# Kalman and Bayesian Filters in Python

Roger R Labbe Jr

May 23, 2020

# Contents

**Preface** **5**
0.1 Kalman and Bayesian Filters . . . 5
0.2 Motivation for this Book . . . 6
0.3 Reading Online . . . 7
0.4 PDF Version . . . 7
0.5 Downloading and Running the Book . . . 7
0.6 Jupyter . . . 8
0.7 SciPy, NumPy, and Matplotlib . . . 9
0.7.1 Exercise - Create arrays . . . 13
0.7.2 Solution . . . 13
0.8 Companion Software . . . 14
0.9 Thoughts on Python and Coding Math . . . 14
0.10 License . . . 15
0.11 Contact . . . 15
0.12 Resources . . . 15

**1 The g-h Filter** **17**
1.1 Building Intuition via Thought Experiments . . . 17
1.2 The g-h Filter . . . 29
1.3 Notation . . . 32
1.4 Exercise: Write Generic Algorithm . . . 32
1.4.1 Solution and Discussion . . . 33
1.5 Choice of $g$ and $h$ . . . 34
1.6 Exercise: create measurement function . . . 34
1.6.1 Solution . . . 34
1.6.2 Discussion . . . 35
1.7 Exercise: Bad Initial Conditions . . . 35
1.7.1 Solution and Discussion . . . 36
1.8 Exercise: Extreme Noise . . . 36
1.8.1 Solution and Discussion . . . 36
1.9 Exercise: The Effect of Acceleration . . . 37
1.9.1 Solution and Discussion . . . 37
1.10 Exercise: Varying $g$ . . . 38
1.10.1 Solution and Discussion . . . 38
1.11 Varying $h$ . . . 40
1.12 Interactive Example . . . 41
1.13 Don't Lie to the Filter . . . 42
1.14 Tracking a Train . . . 43
1.15 g-h Filters with FilterPy . . . 47
1.16 Summary . . . 49

**2 Discrete Bayes Filter** **51**
2.1 Tracking a Dog . . . 51
2.2 Extracting Information from Sensor Readings . . . 53
2.3 Noisy Sensors . . . 54
2.4 Incorporating Movement . . . 57
2.5 Terminology . . . 59
2.6 Adding Uncertainty to the Prediction . . . 60
2.7 Generalizing with Convolution . . . 63
2.8 Integrating Measurements and Movement Updates . . . 65
2.9 The Discrete Bayes Algorithm . . . 68
2.10 The Effect of Bad Sensor Data . . . 71
2.11 Drawbacks and Limitations . . . 73
2.12 Tracking and Control . . . 74
2.12.1 Simulating the Train Behavior . . . 74
2.13 Bayes Theorem and the Total Probability Theorem . . . 78
2.14 Summary . . . 79
2.15 References . . . 79

**3 Probabilities, Gaussians, and Bayes' Theorem** **81**
3.1 Introduction . . . 81
3.2 Mean, Variance, and Standard Deviations . . . 81
3.2.1 Random Variables . . . 81
3.3 Probability Distribution . . . 82
3.3.1 The Mean, Median, and Mode of a Random Variable . . . 83
3.4 Expected Value of a Random Variable . . . 84
3.4.1 Exercise . . . 85
3.4.2 Solution . . . 85
3.4.3 Exercise . . . 86
3.4.4 Solution . . . 86
3.4.5 Variance of a Random Variable . . . 86
3.4.6 Why the Square of the Differences . . . 90
3.5 Gaussians . . . 91
3.6 Nomenclature . . . 92
3.7 Gaussian Distributions . . . 93
3.8 The Variance and Belief . . . 96
3.9 The 68-95-99.7 Rule . . . 97
3.10 Interactive Gaussians . . . 98
3.11 Computational Properties of Gaussians . . . 99
3.12 Putting it all Together . . . 101
3.12.1 Bayes Theorem . . . 102
3.12.2 Total Probability Theorem . . . 104
3.13 Computing Probabilities with scipy.stats . . . 105
3.14 Limitations of Using Gaussians to Model the World . . . 106
3.15 Product of Gaussians (Optional) . . . 109
3.16 Sum of Gaussians (Optional) . . . 110
3.17 Summary and Key Points . . . 111
3.18 References . . . 111
3.19 Useful Wikipedia Links . . . 112

**4 One Dimensional Kalman Filters 113**
4.1 Problem Description . . . 113
4.2 Beliefs as Gaussians . . . 114
4.3 Tracking with Gaussian Probabilities . . . 115
4.4 Predictions with Gaussians . . . 116
4.5 Updates with Gaussians . . . 118
4.5.1 Understanding Gaussian Multiplication . . . 119
4.5.2 Interactive Example . . . 122
4.6 First Kalman Filter . . . 123
4.7 Code Walkthrough . . . 126
4.7.1 Exercise: Modify Variance Values . . . 128
4.7.2 KF Animation . . . 128
4.8 Kalman Gain . . . 129
4.9 Full Description of the Algorithm . . . 131
4.10 Comparison with g-h and discrete Bayes Filters . . . 132
4.11 Introduction to Designing a Filter . . . 134
4.11.1 Animation . . . 137
4.12 Example: Extreme Amounts of Noise . . . 137
4.13 Example: Incorrect Process Variance . . . 138
4.14 Example: Bad Initial Estimate . . . 139
4.15 Example: Large Noise and Bad Initial Estimate . . . 140
4.16 Exercise: Interactive Plots . . . 142
4.16.1 Solution . . . 143
4.17 Exercise - Nonlinear Systems . . . 144
4.17.1 Solution . . . 144
4.17.2 Discussion . . . 145
4.18 Fixed Gain Filters . . . 145
4.19 FilterPy's Implementation . . . 146
4.20 Summary . . . 146

**5 Multivariate Gaussians 149**
5.1 Introduction . . . 149
5.2 Multivariate Normal Distributions . . . 149
5.3 Correlation and Covariance . . . 150
5.4 Multivariate Normal Distribution Equation . . . 155
5.4.1 Pearson's Correlation Coefficient . . . 165
5.5 Using Correlations to Improve Estimates . . . 165
5.6 Multiplying Multidimensional Gaussians . . . 166
5.7 Hidden Variables . . . 172
5.8 Higher Dimensions . . . 176
5.9 Summary . . . 177
5.10 References . . . 177

**6 Multivariate Kalman Filters 179**
6.1 Introduction . . . 179
6.2 Newton's Equations of Motion . . . 179
6.3 Kalman Filter Algorithm . . . 180
6.4 Tracking a Dog . . . 182
6.5 Predict Step . . . 182
6.5.1 Design State Variable . . . 182
6.5.2 Design State Covariance . . . 184
6.5.3 Design the Process Model . . . 185
6.5.4 Design Process Noise . . . 188
6.5.5 Design the Control Function . . . 189

6.6 Update Step . . . 190
6.6.1 Design the Measurement Function . . . 190
6.7 Implementing the Kalman Filter . . . 192
6.8 The Saver Class . . . 197
6.9 The Kalman Filter Equations . . . 199
6.9.1 Prediction Equations . . . 199
6.9.2 Update Equations . . . 203
6.9.3 An Example not using FilterPy . . . 205
6.9.4 Summary . . . 206
6.10 Exercise: Show Effect of Hidden Variables . . . 207
6.10.1 Solution . . . 207
6.10.2 Discussion . . . 209
6.11 How Velocity is Calculated . . . 210
6.12 Adjusting the Filter . . . 211
6.13 A Detailed Examination of the Covariance Matrix . . . 215
6.14 Question: Explain Ellipse Differences . . . 219
6.14.1 Solution . . . 220
6.15 Filter Initialization . . . 221
6.16 Batch Processing . . . 222
6.17 Smoothing the Results . . . 223
6.18 Exercise: Compare Velocities . . . 225
6.18.1 Solution . . . 225
6.19 Discussion and Summary . . . 226
6.20 References . . . 227

**7 Kalman Filter Math** **229**
7.1 Modeling a Dynamic System . . . 229
7.2 State-Space Representation of Dynamic Systems . . . 231
7.2.1 Forming First Order Equations from Higher Order Equations . . . 232
7.2.2 First Order Differential Equations In State-Space Form . . . 232
7.2.3 Finding the Fundamental Matrix for Time Invariant Systems . . . 233
7.2.4 The Matrix Exponential . . . 233
7.2.5 Time Invariance . . . 235
7.2.6 Linear Time Invariant Theory . . . 236
7.2.7 Numerical Solutions . . . 236
7.3 Design of the Process Noise Matrix . . . 237
7.3.1 Continuous White Noise Model . . . 237
7.3.2 Piecewise White Noise Model . . . 239
7.3.3 Using FilterPy to Compute Q . . . 241
7.3.4 Simplification of Q . . . 241
7.4 Stable Compution of the Posterior Covariance . . . 242
7.5 Deriving the Kalman Gain Equation . . . 244
7.6 Numeric Integration of Differential Equations . . . 245
7.6.1 Euler's Method . . . 245
7.6.2 Runge Kutta Methods . . . 248
7.7 Bayesian Filtering . . . 250
7.8 Converting Kalman Filter to a g-h Filter . . . 251
7.9 References . . . 253

**8 Designing Kalman Filters** **255**
8.1 Introduction . . . 255
8.2 Tracking a Robot . . . 255
8.2.1 Choose the State Variables . . . 256
8.2.2 Design State Transition Function . . . 257
8.2.3 Design the Process Noise Matrix . . . 258
8.2.4 Design the Control Function . . . 258
8.2.5 Design the Measurement Function . . . 258
8.2.6 Design the Measurement Noise Matrix . . . 259
8.2.7 Initial Conditions . . . 259
8.2.8 Implement the Filter . . . 259
8.3 Filter Order . . . 262
8.3.1 Zero Order Kalman Filter . . . 264
8.3.2 First Order Kalman Filter . . . 264
8.3.3 Second Order Kalman Filter . . . 265
8.4 Evaluating Filter Order . . . 266
8.5 Exercise: State Variable Design . . . 279
8.5.1 Solution . . . 280
8.6 Detecting and Rejecting Bad Measurement . . . 282
8.6.1 Gating and Data Association Strategies . . . 287
8.7 Evaluating Filter Performance . . . 289
8.7.1 Normalized Estimated Error Squared (NEES) . . . 289
8.7.2 Likelihood Function . . . 290
8.8 Control Inputs . . . 292
8.9 Sensor Fusion . . . 294
8.9.1 Exercise: Can you Filter GPS outputs? . . . 299
8.9.2 Exercise: Prove that the position sensor improves the filter . . . 302
8.10 Nonstationary Processes . . . 303
8.10.1 Sensor fusion: Different Data Rates . . . 304
8.11 Tracking a Ball . . . 306
8.11.1 Choose the State Variables . . . 308
8.11.2 Design State Transition Function . . . 309
8.11.3 Design the Control Input Function . . . 310
8.11.4 Design the Measurement Function . . . 310
8.11.5 Design the Measurement Noise Matrix . . . 311
8.11.6 Design the Process Noise Matrix . . . 311
8.11.7 Design the Initial Conditions . . . 311
8.12 Tracking a Ball in Air . . . 313
8.12.1 Implementing Air Drag . . . 313
8.13 References . . . 321

**9 Nonlinear Filtering** **323**
9.1 Introduction . . . 323
9.2 The Problem with Nonlinearity . . . 324
9.3 An Intuitive Look at the Problem . . . 324
9.4 The Effect of Nonlinear Functions on Gaussians . . . 325
9.5 A 2D Example . . . 330
9.6 The Algorithms . . . 333
9.7 Summary . . . 333
9.8 References . . . 334

**10 The Unscented Kalman Filter** **335**
10.1 Sigma Points - Sampling from a Distribution . . . 337
10.2 A Quick Example . . . 338
10.3 Choosing Sigma Points . . . 340
10.4 The Unscented Transform . . . 343
10.4.1 Accuracy of the Unscented Transform . . . 344
10.5 The Unscented Kalman Filter . . . 345
10.5.1 Predict Step . . . 345
10.5.2 Update Step . . . 346
10.6 Van der Merwe's Scaled Sigma Point Algorithm . . . 347
10.6.1 Sigma Point Computation . . . 347
10.6.2 Weight Computation . . . 348
10.6.3 Reasonable Choices for the Parameters . . . 348
10.7 Using the UKF . . . 348
10.8 Tracking an Airplane . . . 352
10.8.1 Tracking Maneuevering Aircraft . . . 356
10.8.2 Sensor Fusion . . . 358
10.8.3 Multiple Position Sensors . . . 361
10.9 Effects of Sensor Error and Geometry . . . 364
10.10 Exercise: Explain Filter Performance . . . 365
10.10.1 Solution . . . 365
10.11 Implementation of the UKF . . . 366
10.11.1 Weights . . . 367
10.11.2 Sigma Points . . . 367
10.11.3 Predict Step . . . 369
10.11.4 Update Step . . . 369
10.11.5 FilterPy's Implementation . . . 370
10.12 Batch Processing . . . 370
10.13 Smoothing the Results . . . 373
10.14 Choosing the Sigma Parameters . . . 374
10.15 Robot Localization - A Fully Worked Example . . . 376
10.15.1 Robot Motion Model . . . 376
10.15.2 Design the State Variables . . . 378
10.15.3 Design the System Model . . . 378
10.15.4 Design the Measurement Model . . . 378
10.15.5 Design Measurement Noise . . . 379
10.15.6 Implementation . . . 379
10.15.7 Steering the Robot . . . 383
10.16 Discussion . . . 385
10.17 References . . . 386

**11 The Extended Kalman Filter** **387**
11.1 Linearizing the Kalman Filter . . . 387
11.2 Example: Tracking a Airplane . . . 389
11.2.1 Design the State Variables . . . 390
11.2.2 Design the Process Model . . . 390
11.2.3 Design the Measurement Model . . . 391
11.2.4 Design Process and Measurement Noise . . . 393
11.2.5 Implementation . . . 393
11.3 Using SymPy to compute Jacobians . . . 396
11.4 Robot Localization . . . 396
11.4.1 Robot Motion Model . . . 396
11.4.2 Design the State Variables . . . 397
11.4.3 Design the System Model . . . 397

11.4.4 Design the Measurement Model . . . 399
11.4.5 Design Measurement Noise . . . 400
11.4.6 Implementation . . . 401
11.4.7 Discussion . . . 407
11.5 UKF vs EKF . . . 407

**12 Particle Filters** **411**
12.1 Motivation . . . 411
12.2 Monte Carlo Sampling . . . 412
12.3 Generic Particle Filter Algorithm . . . 413
12.4 Probability distributions via Monte Carlo . . . 414
12.5 The Particle Filter . . . 417
12.5.1 Predict Step . . . 418
12.5.2 Update Step . . . 418
12.5.3 Computing the State Estimate . . . 419
12.5.4 Particle Resampling . . . 419
12.6 SIR Filter - A Complete Example . . . 420
12.6.1 Effect of Sensor Errors on the Filter . . . 424
12.6.2 Filter Degeneracy From Inadequate Samples . . . 425
12.7 Importance Sampling . . . 426
12.8 Resampling Methods . . . 427
12.8.1 Multinomial Resampling . . . 428
12.8.2 Residual Resampling . . . 429
12.8.3 Stratified Resampling . . . 430
12.8.4 Systematic Resampling . . . 431
12.8.5 Choosing a Resampling Algorithm . . . 431
12.9 Summary . . . 432
12.10References . . . 433

**13 Smoothing** **435**
13.1 Introduction . . . 435
13.2 An Overview of How Smoothers Work . . . 437
13.3 Types of Smoothers . . . 437
13.4 Choice of Filters . . . 438
13.5 Fixed-Interval Smoothing . . . 438
13.6 Fixed-Lag Smoothing . . . 442
13.7 References . . . 444

**14 Adaptive Filtering** **445**
14.1 Introduction . . . 445
14.2 Maneuvering Targets . . . 445
14.3 Detecting a Maneuver . . . 452
14.4 Adjustable Process Noise . . . 454
14.4.1 Continuous Adjustment . . . 454
14.4.2 Continuous Adjustment - Standard Deviation Version . . . 456
14.5 Fading Memory Filter . . . 462
14.6 Multiple Model Estimation . . . 465
14.6.1 A Two Filter Adaptive Filter . . . 466
14.7 MMAE . . . 468
14.7.1 Limitations of the MMAE Filter . . . 470
14.8 Interacting Multiple Models (IMM) . . . 471
14.8.1 Mode Probabilities . . . 472
14.8.2 Mode Transitions . . . 472
14.8.3 Computing the Mode Probabilities . . . 474

14.9 Mixing Probabilities . . . 475
14.9.1 IMM Estimate . . . 476
14.9.2 Tracking Maneuvering Target with the IMM . . . 476
14.9.3 Limitations of the IMM . . . 478
14.10Summary . . . 480
14.11References . . . 480

**A Installation** **483**
A.1 Installing the SciPy Stack . . . 483
A.2 Installing FilterPy . . . 484
A.3 Downloading and Running the Book . . . 484
A.4 Companion Software . . . 485
A.5 Using Jupyter Notebook . . . 485
A.6 SymPy . . . 485
A.7 Various Links . . . 487

**B Symbols and Notations** **489**
B.1 Labbe . . . 489
B.2 Wikipedia . . . 489
B.3 Brookner . . . 489
B.4 Gelb . . . 490
B.5 Brown . . . 490
B.6 Zarchan . . . 490

**C H Infinity filter** **491**

**D Ensemble Kalman Filters** **493**
D.1 The Algorithm . . . 494
D.1.1 Initialize Step . . . 495
D.1.2 Predict Step . . . 495
D.1.3 Update Step . . . 495
D.2 Implementation and Example . . . 497
D.3 Outstanding Questions . . . 499
D.4 References . . . 500

# Preface

```
In [1]: from __future__ import division, print_function
        %matplotlib inline
```

```
In [2]: #format the book
        import book_format
        book_format.set_style()
```

```
Out[2]:
```

Introductory textbook for Kalman filters and Bayesian filters. The book is written using Jupyter Notebook so you may read the book in your browser and also run and modify the code, seeing the results inside the book. What better way to learn?

## 0.1 Kalman and Bayesian Filters

Sensors are noisy. The world is full of data and events that we want to measure and track, but we cannot rely on sensors to give us perfect information. The GPS in my car reports altitude. Each time I pass the same point in the road it reports a slightly different altitude. My kitchen scale gives me different readings if I weigh the same object twice.

In simple cases the solution is obvious. If my scale gives slightly different readings I can just take a few readings and average them. Or I can replace it with a more accurate scale. But what do we do when the sensor is very noisy, or the environment makes data collection difficult? We may be trying to track the movement of a low flying aircraft. We may want to create an autopilot for a drone, or ensure that our farm tractor seeded the entire field. I work on computer vision, and I need to track moving objects in images, and the computer vision algorithms create very noisy and unreliable results.

This book teaches you how to solve these sorts of filtering problems. I use many different algorithms, but they are all based on *Bayesian probability*. In simple terms Bayesian probability determines what is likely to be true based on past information.

If I asked you the heading of my car at this moment you would have no idea. You'd proffer a number between 1° and 360° degrees, and have a 1 in 360 chance of being right. Now suppose I told you that 2 seconds ago its heading was 243°. In 2 seconds my car could not turn very far so you could make a far more accurate prediction. You are using past information to more accurately infer information about the present or future.

The world is also noisy. That prediction helps you make a better estimate, but it also subject to noise. I may have just braked for a dog or swerved around a pothole. Strong winds and ice on the road are external influences on the path of my car. In control literature we call this *noise* though you may not think of it that way.

There is more to Bayesian probability, but you have the main idea. Knowledge is uncertain, and we alter our beliefs based on the strength of the evidence. Kalman and Bayesian filters blend our noisy and limited knowledge of how a system behaves with the noisy and limited sensor readings to produce the best possible estimate of the state of the system. Our principle is to never discard information.

Say we are tracking an object and a sensor reports that it suddenly changed direction. Did it really turn, or is the data noisy? It depends. If this is a jet fighter we'd be very inclined to believe the report of a

sudden maneuver. If it is a freight train on a straight track we would discount it. We'd further modify our belief depending on how accurate the sensor is. Our beliefs depend on the past and on our knowledge of the system we are tracking and on the characteristics of the sensors.

The Kalman filter was invented by Rudolf Emil Kálmán to solve this sort of problem in a mathematically optimal way. Its first use was on the Apollo missions to the moon, and since then it has been used in an enormous variety of domains. There are Kalman filters in aircraft, on submarines, and on cruise missiles. Wall street uses them to track the market. They are used in robots, in IoT (Internet of Things) sensors, and in laboratory instruments. Chemical plants use them to control and monitor reactions. They are used to perform medical imaging and to remove noise from cardiac signals. If it involves a sensor and/or time-series data, a Kalman filter or a close relative to the Kalman filter is usually involved.

## 0.2 Motivation for this Book

I'm a software engineer that spent almost two decades in aerospace, and so I have always been 'bumping elbows' with the Kalman filter, but never implemented one. They've always had a fearsome reputation for difficulty. The theory is beautiful, but quite difficult to learn if you are not already well trained in topics such as signal processing, control theory, probability and statistics, and guidance and control theory. As I moved into solving tracking problems with computer vision the need to implement them myself became urgent.

There are excellent textbooks in the field, such as Grewal and Andrew's *Kalman Filtering*. But sitting down and trying to read many of these books is a dismal and trying experience if you do not have the necessary background. Typically the first few chapters fly through several years of undergraduate math, blithely referring you to textbooks on Itō calculus, and presenting an entire semester's worth of statistics in a few brief paragraphs. They are textbooks for an upper undergraduate or graduate level course, and an invaluable reference to researchers and professionals, but the going is truly difficult for the more casual reader. Notation is introduced without explanation, different texts use different words and variable names for the same concept, and the books are almost devoid of examples or worked problems. I often found myself able to parse the words and comprehend the mathematics of a definition, but had no idea as to what real world phenomena these words and math were attempting to describe. "But what does that *mean?*" was my repeated thought. Here are typical examples which once puzzled me:

$$\hat{x}_k = \Phi_k\hat{x}_{k-1} + G_k u_{k-1} + K_k[z_k - H\Phi_k\hat{x}_{k-1} - HG_k u_{k-1}]$$
$$\mathbf{P}_{k|k} = (I - \mathbf{K}_k\mathbf{H}_k)\text{cov}(\mathbf{x}_k - \hat{\mathbf{x}}_{k|k-1})(I - \mathbf{K}_k\mathbf{H}_k)^{\mathrm{T}} + \mathbf{K}_k\text{cov}(\mathbf{v}_k)\mathbf{K}_k^{\mathrm{T}}$$

However, as I began to finally understand the Kalman filter I realized the underlying concepts are quite straightforward. If you know a few simple probability rules, and have some intuition about how we fuse uncertain knowledge, the concepts of the Kalman filter are accessible. Kalman filters have a reputation for difficulty, but shorn of much of the formal terminology the beauty of the subject and of their math became clear to me, and I fell in love with the topic.

As I began to understand the math and theory more difficulties appeared. A book or paper will make some statement of fact and presents a graph as proof. Unfortunately, why the statement is true is not clear to me, or I cannot reproduce the plot. Or maybe I wonder "is this true if R=0?" Or the author provides pseudocode at such a high level that the implementation is not obvious. Some books offer Matlab code, but I do not have a license to that expensive package. Finally, many books end each chapter with many useful exercises. Exercises which you need to understand if you want to implement Kalman filters for yourself, but exercises with no answers. If you are using the book in a classroom, perhaps this is okay, but it is terrible for the independent reader. I loathe that an author withholds information from me, presumably to avoid 'cheating' by the student in the classroom.

All of this impedes learning. I want to track an image on a screen, or write some code for my Arduino project. I want to know how the plots in the book are made, and to choose different parameters than the author chose. I want to run simulations. I want to inject more noise into the signal and see how a filter performs. There are thousands of opportunities for using Kalman filters in everyday code, and yet this fairly straightforward topic is the provenance of rocket scientists and academics.

I wrote this book to address all of those needs. This is not the sole book for you if you design military radars. Go get a Masters or PhD at a great STEM school, because you'll need it. This book is for the hobbyist, the curious, and the working engineer that needs to filter or smooth data. If you are a hobbyist this book should provide everything you need. If you are serious about Kalman filters you'll need more. My intention is to introduce enough of the concepts and mathematics to make the textbooks and papers approachable.

This book is interactive. While you can read it online as static content, I urge you to use it as intended. It is written using Jupyter Notebook. This allows me to combine text, math, Python, and Python output in one place. Every plot, every piece of data in this book is generated from Python inside the notebook. Want to double the value of a parameter? Just change the parameter's value, and press CTRL-ENTER. A new plot or printed output will appear.

This book has exercises, but it also has the answers. I trust you. If you just need an answer, go ahead and read the answer. If you want to internalize this knowledge, try to implement the exercise before you read the answer. Since the book is interactive, you enter and run your solution inside the book - you don't have to move to a different environment, or deal with importing a bunch of stuff before starting.

This book is free. I've spent several thousand dollars on Kalman filtering books. I cannot believe they are within the reach of someone in a depressed economy or a financially struggling student. I have gained so much from free software like Python, and free books like those from Allen B. Downey [1]. It's time to repay that. So, the book is free, it is hosted on free servers at GitHub, and it uses only free and open software such as IPython and MathJax.

## 0.3 Reading Online

GitHub

The book is hosted on GitHub, and you can read any chapter by clicking on its name. GitHub statically renders Jupyter Notebooks. You will not be able to run or alter the code, but you can read all of the content.

The GitHub pages for this project are at

```
https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python
```

binder

binder serves interactive notebooks online, so you can run the code and change the code within your browser without downloading the book or installing Jupyter. Use this link to access the book via binder:

```
http://mybinder.org/repo/rlabbe/Kalman-and-Bayesian-Filters-in-Python
```

nbviewer

The nbviewer website will render any Notebook in a static format. I find it does a slightly better job than the GitHub renderer, but it is slighty harder to use. It accesses GitHub directly; whatever I have checked into GitHub will be rendered by nbviewer.

You may access this book via nbviewer here:

http://nbviewer.ipython.org/github/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/table_of_contents.ipynb

## 0.4 PDF Version

I periodically generate a PDF of the book from the notebooks. You can access it here:

https://drive.google.com/file/d/0By_SW19c1BfhSVFzNHc0SjduNzg/view?usp=sharing

## 0.5 Downloading and Running the Book

However, this book is intended to be interactive and I recommend using it in that form. It's a little more effort to set up, but worth it. If you install IPython and some supporting libraries on your computer and then

clone this book you will be able to run all of the code in the book yourself. You can perform experiments, see how filters react to different data, see how different filters react to the same data, and so on. I find this sort of immediate feedback both vital and invigorating. You do not have to wonder "what happens if". Try it and see!

Instructions for installation can be found in the Installation appendix, found here.

Once the software is installed you can navigate to the installation directory and run Juptyer notebook with the command line instruction

```
jupyter notebook
```

This will open a browser window showing the contents of the base directory. The book is organized into chapters. Each chapter is named *xx-name*.ipynb, where *xx* is the chapter number. .ipynb is the Notebook file extension. To read Chapter 2, click on the link for chapter 2. This will cause the browser to open that subdirectory. In each subdirectory there will be one or more IPython Notebooks (all notebooks have a .ipynb file extension). The chapter contents are in the notebook with the same name as the chapter name. There are sometimes supporting notebooks for doing things like generating animations that are displayed in the chapter. These are not intended to be read by the end user, but of course if you are curious as to how an animation is made go ahead and take a look.

Admittedly this is a cumbersome interface to a book. I am following in the footsteps of several other projects that are re-purposing Jupyter Notebook to generate entire books. I feel the slight annoyances have a huge payoff - instead of having to download a separate code base and run it in an IDE while you try to read a book, all of the code and text is in one place. If you want to alter the code, you may do so and immediately see the effects of your change. If you find a bug, you can make a fix, and push it back to my repository so that everyone in the world benefits. And, of course, you will never encounter a problem I face all the time with traditional books - the book and the code are out of sync with each other, and you are left scratching your head as to which source to trust.

## 0.6 Jupyter

First, some words about using Jupyter Notebooks with this book. This book is interactive. If you want to run code examples, and especially if you want to see animated plots, you will need to run the code cells. I cannot teach you everything about Jupyter Notebooks. However, a few things trip readers up. You can go to http://jupyter.org/ for detailed documentation.

First, you must always run the topmost code cell, the one with the comment `#format the book`. It is directly above. This does not just set up formatting, which you might not care about, but it also loads some necessary modules and makes some global settings regarding plotting and printing. So, always run this cell unless you are just passively reading.

The line

```
%matplotlib inline
```

causes plots to be displayed inside the notebook. Matplotlib is a plotting package which is described below. For reasons I don't understand the default behavior of Jupyter Notebooks is to generate plots in an external window.

The percent sign in `%matplotlib` is used for IPython *magic* - these are commands to the kernel to do things that are not part of the Python language. There are many useful magic commands, and you can read about them here: http://ipython.readthedocs.io/en/stable/interactive/magics.html

Running the code inside a cell is easy. Click on it so that it has focus (a box will be drawn around it), and then press CTRL-Enter.

Second, cells must be run in order. I break problems up over several cells; if you try to just skip down and run the tenth code cell it almost certainly won't work. If you haven't run anything yet just choose *Run All Above* from the **Cell** menu item. That's the easiest way to ensure everything has been run.

Once cells are run you can often jump around and rerun cells in different orders, but not always. I'm trying to fix this, but there is a tradeoff. I'll define a variable in cell 10 (say), and then run code that modifies

that variable in cells 11 and 12. If you go back and run cell 11 again the variable will have the value that was set in cell 12, and the code expects the value that was set in cell 10. So, occasionally you'll get weird results if you run cells out of order. My advise is to backtrack a bit, and run cells in order again to get back to a proper state. It's annoying, but the interactive aspect of Jupyter notebooks more than makes up for it. Better yet, submit an issue on GitHub so I know about the problem and fix it!

Finally, some readers have reported problems with the animated plotting features in some browsers. I have not been able to reproduce this. In parts of the book I use the `%matplotlib notebook` magic, which enables interactive plotting. If these plots are not working for you, try changing this to read `%matplotlib inline`. You will lose the animated plotting, but it seems to work on all platforms and browsers.

## 0.7 SciPy, NumPy, and Matplotlib

SciPy is a open source collection of software for mathematics. Included in SciPy are NumPy, which provides array objects, linear algebra, random numbers, and more. Matplotlib provides plotting of NumPy arrays. SciPy's modules duplicate some of the functionality in NumPy while adding features such as optimization, image processing, and more.

To keep my efforts for this book managable I have elected to assume that you know how to program in Python, and that you also are familiar with these packages. Nonetheless, I will take a few moments to illustrate a few features of each; realistically you will have to find outside sources to teach you the details. The home page for SciPy, https://scipy.org, is the perfect starting point, though you will soon want to search for relevant tutorials and/or videos.

NumPy, SciPy, and Matplotlib do not come with the default Python distribution; see the *Installation* Appendix if you do not have them installed.

I use NumPy's array data structure throughout the book, so let's learn about them now. I will teach you enough to get started; refer to NumPy's documentation if you want to become an expert.

`numpy.array` implements a one or more dimensional array. Its type is `numpy.ndarray`, and we will refer to this as an ndarray for short. You can construct it with any list-like object. The following constructs a 1-D array from a list:

```
In [3]: import numpy as np
        x = np.array([1, 2, 3])
        print(type(x))
        x
```

```
<class 'numpy.ndarray'>
```

```
Out[3]: array([1, 2, 3])
```

It has become a industry standard to use `import numpy as np`.

You can also use tuples:

```
In [4]: x = np.array((4,5,6))
        x
```

```
Out[4]: array([4, 5, 6])
```

Create multidimensional arrays with nested brackets:

```
In [5]: x = np.array([[1, 2, 3],
                      [4, 5, 6]])
        print(x)
```

```
[[1 2 3]
 [4 5 6]]
```

You can create arrays of 3 or more dimensions, but we have no need for that here, and so I will not elaborate.

By default the arrays use the data type of the values in the list; if there are multiple types then it will choose the type that most accurately represents all the values. So, for example, if your list contains a mix of `int` and `float` the data type of the array would be of type `float`. You can override this with the `dtype` parameter.

```
In [6]: x = np.array([1, 2, 3], dtype=float)
        print(x)
```

```
[1. 2. 3.]
```

You can access the array elements using subscript location:

```
In [7]: x = np.array([[1, 2, 3],
                      [4, 5, 6]])

        print(x[1,2])
```

```
6
```

You can access a column or row by using slices. A colon (:) used as a subscript is shorthand for all data in that row or column. So `x[:,0]` returns an array of all data in the first column (the 0 specifies the first column):

```
In [8]: x[:, 0]
```

```
Out[8]: array([1, 4])
```

We can get the second row with:

```
In [9]: x[1, :]
```

```
Out[9]: array([4, 5, 6])
```

Get the last two elements of the second row with:

```
In [10]: x[1, 1:]
```

```
Out[10]: array([5, 6])
```

As with Python `lists`, you can use negative indexes to refer to the end of the array. -1 refers to the last index. So another way to get the last two elements of the second (last) row would be:

```
In [11]: x[-1, -2:]
```

```
Out[11]: array([5, 6])
```

You can perform matrix addition with the `+` operator, but matrix multiplication requires the `dot` method or function. The `*` operator performs element-wise multiplication, which is **not** what you want for linear algebra.

```
In [12]: x = np.array([[1., 2.],
                       [3., 4.]])
         print('addition:\n', x + x)
         print('\nelement-wise multiplication\n', x * x)
         print('\nmultiplication\n', np.dot(x, x))
         print('\ndot is also a member of np.array\n', x.dot(x))
```

```
addition:
 [[2. 4.]
 [6. 8.]]

element-wise multiplication
 [[ 1.  4.]
 [ 9. 16.]]

multiplication
 [[ 7. 10.]
 [15. 22.]]

dot is also a member of np.array
 [[ 7. 10.]
 [15. 22.]]
```

Python 3.5 introduced the `@` operator for matrix multiplication.

```
In [13]: x @ x

Out[13]: array([[ 7., 10.],
                [15., 22.]])
```

This will only work if you are using Python 3.5+. This book requires 3.5 or later, so I will use it whenever I can. Note that the operator requires that both values are arrays. Hence, `x @ 3.` raises a ValueError whereas `np.dot(X, 3.)` works fine.

You can get the transpose with `.T`, and the inverse with `numpy.linalg.inv`. The SciPy package also provides the inverse function.

```
In [14]: import scipy.linalg as linalg
         print('transpose\n', x.T)
         print('\nNumPy ninverse\n', np.linalg.inv(x))
         print('\nSciPy inverse\n', linalg.inv(x))

transpose
 [[1. 3.]
 [2. 4.]]

NumPy ninverse
 [[-2.   1. ]
 [ 1.5 -0.5]]

SciPy inverse
 [[-2.   1. ]
 [ 1.5 -0.5]]
```

There are helper functions like `zeros` to create a matrix of all zeros, `ones` to get all ones, and `eye` to get the identity matrix. If you want a multidimensional array, use a tuple to specify the shape.

```
In [15]: print('zeros\n', np.zeros(7))
         print('\nzeros(3x2)\n', np.zeros((3, 2)))
         print('\neye\n', np.eye(3))
```

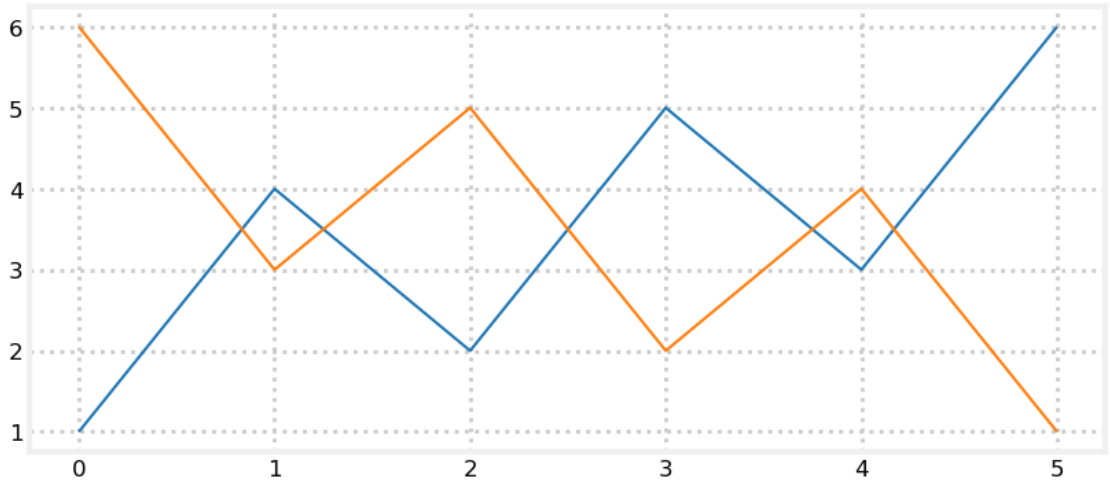
```
zeros
 [0. 0. 0. 0. 0. 0. 0.]

zeros(3x2)
 [[0. 0.]
 [0. 0.]
 [0. 0.]]

eye
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

We have functions to create equally spaced data. `arange` works much like Python's `range` function, except it returns a NumPy array. `linspace` works slightly differently, you call it with `linspace(start, stop, num)`, where `num` is the length of the array that you want.

```
In [16]: np.arange(0, 2, 0.1)

Out[16]: array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. , 1.1,
                1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9])

In [17]: np.linspace(0, 2, 20)

Out[17]: array([0.   , 0.105, 0.211, 0.316, 0.421, 0.526, 0.632, 0.737, 0.842,
                0.947, 1.053, 1.158, 1.263, 1.368, 1.474, 1.579, 1.684, 1.789,
                1.895, 2.   ])
```

Now let's plot some data. For the most part it is very simple. Matplotlib contains a plotting library `pyplot`. It is industry standard to import it as `plt`. Once imported, plot numbers by calling `plt.plot` with a list or array of numbers. If you make multiple calls it will plot multiple series, each with a different color.

```
In [18]: import matplotlib.pyplot as plt
         a = np.array([6, 3, 5, 2, 4, 1])
         plt.plot([1, 4, 2, 5, 3, 6])
         plt.plot(a)

Out[18]: [<matplotlib.lines.Line2D at 0x298bcc7b4c8>]
```



The output `[<matplotlib.lines.Line2D at 0x2ba160bed68>]` is because `plt.plot` returns the object that was just created. Ordinarily we do not want to see that, so I add a `;` to my last plotting command to suppress that output.

By default `plot` assumes that the x-series is incremented by one. You can provide your own x-series by passing in both x and y.

```
In [19]: plt.plot(np.arange(0,1, 0.1), [1,4,3,2,6,4,7,3,4,5]);
```


There are many more features to these packages which I use in this book. Normally I will introduce them without explanation, trusting that you can infer the usage from context, or search online for an explanation. As always, if you are unsure, create a new cell in the Notebook or fire up a Python console and experiment!

### 0.7.1 Exercise - Create arrays

I want you to create a NumPy array of 10 elements with each element containing 1/10. There are several ways to do this; try to implement as many as you can think of.

```
In [20]: # your solution
```

### 0.7.2 Solution

Here are three ways to do this. The first one is the one I want you to know. I used the '/' operator to divide all of the elements of the array with 10. We will shortly use this to convert the units of an array from meters to km.

```
In [21]: print(np.ones(10) / 10.)
         print(np.array([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1]))
         print(np.array([.1] * 10))
```

```
[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
```

Here is one I haven't covered yet. The function `numpy.asarray()` will convert its argument to an ndarray if it isn't already one. If it is, the data is unchanged. This is a handy way to write a function that can accept either Python lists or ndarrays, and it is very efficient if the type is already ndarray as nothing new is created.

```
In [22]: def one_tenth(x):
             x = np.asarray(x)
             return x / 10.

         print(one_tenth([1, 2, 3]))             # I work!
         print(one_tenth(np.array([4, 5, 6])))   # so do I!
```

```
[0.1 0.2 0.3]
[0.4 0.5 0.6]
```

## 0.8 Companion Software

I am writing an open source Bayesian filtering Python library called FilterPy. Installation instructions are given above.

FilterPy is hosted GitHub at (https://github.com/rlabbe/filterpy) but the `pip` installed version should serve your needs.

Code that is specific to the book is stored with the book in the subdirectory **/kf_book**. It contains code for formatting the book. It also contains python files with names like *xxx*_internal.py. I use these to store functions that are useful for a specific chapter. This allows me to hide Python code that is not particularly interesting to read - I may be generating a plot or chart, and I want you to focus on the contents of the chart, not the mechanics of how I generate that chart with Python. If you are curious as to the mechanics of that, just go and browse the source.

Some chapters introduce functions that are useful for the rest of the book. Those functions are initially defined within the Notebook itself, but the code is also stored in a Python file in **/kf_book** that is imported if needed in later chapters. I do document when I do this where the function is first defined, but this is still a work in progress. I try to avoid this because then I always face the issue of code in the directory becoming out of sync with the code in the book. However, Jupyter Notebook does not give us a way to refer to code cells in other notebooks, so this is the only mechanism I know of to share functionality across notebooks.

There is an undocumented directory called **/experiments**. This is where I write and test code prior to putting it in the book. There is some interesting stuff in there, and feel free to look at it. As the book evolves I plan to create examples and projects, and a lot of this material will end up there. Small experiments will eventually just be deleted. If you are just interested in reading the book you can safely ignore this directory.

The directory **/kf_book** contains a css file containing the style guide for the book. The default look and feel of Jupyter Notebook is rather plain. I have followed the examples set by books such as Probabilistic Programming and Bayesian Methods for Hackers [2]. I have also been very influenced by Professor Lorena Barba's fantastic work, available here [3]. I owe all of my look and feel to the work of these projects.

## 0.9 Thoughts on Python and Coding Math

Most Kalman filtering and other engineering texts are written by mathematicians or academics. When there is software (rarely), it is not production quality. Take Paul Zarchan's book *Fundamentals of Kalman Filtering* as an example. This is a fantastic book which belongs in your library, and is one of the few books that provides full source for every example and chart. But the code is Fortran without any subroutines beyond calls to functions like `MATMUL`. Kalman filters are re-implemented throughout the book. The same listing mixes simulation with filtering code, making it hard to distinguish them. Some chapters implement the same filter in subtly different ways, and uses bold text to highlight the few lines that changed. If Runge Kutta is needed it is embedded in the code, without comments.

There's a better way. If I want to perform Runge Kutta I call `ode45`, I do not embed an Runge Kutta implementation in my code. I don't want to implement Runge Kutta multiple times and debug it several times. if I do find a bug, I can fix it once and be assured that it now works across all my different projects. And, it is readable. It is rare that I care about the implementation of Runge Kutta.

This is a textbook on Kalman filtering, and you can argue that we *do* care about the implementation of Kalman filters. That is true, but the code that performs the filtering uses about 10 lines of code. The code to implement the math is fairly trivial. Most of the work that Kalman filter requires is the design of the matrices that get fed into the math engine.

A possible downside is that the equations that perform the filtering are hidden behind functions, which we could argue is a loss in a pedagogical text. I argue the converse. I want you to learn how to use Kalman filters in the real world, for real projects, and you shouldn't be cutting and pasting established algorithms all over the place.

I use Python classes. I mostly use classes as a way to organize the data that the filters require, not to implement object oriented (OO) features such as inheritence. For example, the `KalmanFilter` class stores matrices and vectors called `x`, `P`, `R`, `Q`, `S`, `y`, `K`. I've seen procedural libraries for Kalman filters, and they require the programmer to maintain all of those matrices. This perhaps isn't so bad for a toy program, but program a bank of Kalman filters and you will not enjoy having to manage all of those matrices and other associated data. I have derived from these classes occasionally in my own work, and find it handy, but I don't want to force OO on people as I know many do not like it.

## 0.10 License



## 0.11 Contact

rlabbejr@gmail.com

## 0.12 Resources

- [1] http://www.greenteapress.com/
- [2] http://nbviewer.ipython.org/github/CamDavidsonPilon/Probabilistic-Programming-and-Bayesian-Methods-for-Hackers/blob/master/Chapter1_Introduction/Chapter1.ipynb
- [3] https://github.com/barbagroup/CFDPython

# Chapter 1

# The g-h Filter

```
In [1]: from __future__ import division, print_function
        %matplotlib inline
```

```
In [2]: #format the book
        import book_format
        book_format.set_style()
```

```
Out[2]:
```

Before we start, be sure you understand how to use Jupyter Notebooks, and are familiar with the SciPy, NumPy, and Matplotlib packages, as they are used throughout this book. The Preface contains an introduction to these packages.

## 1.1 Building Intuition via Thought Experiments

Imagine that we live in a world without scales - the devices you stand on to weigh yourself. One day at work a co-worker comes running up to you and announces her invention of a 'scale' to you. After she explains, you eagerly stand on it and announce the results: "172 lbs". You are ecstatic - for the first time in your life you know what you weigh. More importantly, dollar signs dance in your eyes as you imagine selling this device to weight loss clinics across the world! This is fantastic!

Another co-worker hears the commotion and comes over to find out what has you so excited. You explain the invention and once again step onto the scale, and proudly proclaim the result: "161 lbs." And then you hesitate, confused.

"It read 172 lbs a few seconds ago", you complain to your co-worker.

"I never said it was accurate," she replies.

Sensors are inaccurate. This is the motivation behind a huge body of work in filtering, and solving this problem is the topic of this book. I could just provide the solutions that have been developed over the last half century, but these solutions were developed by asking very basic, fundamental questions into the nature of what we know and how we know it. Before we attempt the math, let's follow that journey of discovery, and see if it informs our intuition about filtering.

**Try Another Scale**

Is there any way we can improve upon this result? The obvious, first thing to try is get a better sensor. Unfortunately, your co-worker informs you that she has built 10 scales, and they all operate with about the same accuracy. You have her bring out another scale, and you weigh yourself on one, and then on the other. The first scale (A) reads "160 lbs", and the second (B) reads "170 lbs". What can we conclude about your weight?

Well, what are our choices?

- We could choose to only believe A, and assign 160lbs to our weight estimate.
- We could choose to only believe B, and assign 170lbs to our weight.

- We could choose a number less than both A and B.
- We could choose a number greater than both A and B.
- We could choose a number between A and B.

The first two choices are plausible, but we have no reason to favor one scale over the other. Why would we choose to believe A instead of B? We have no reason for such a belief. The third and fourth choices are irrational. The scales are admittedly not very accurate, but there is no reason at all to choose a number outside of the range of what they both measured. The final choice is the only reasonable one. If both scales are inaccurate, and as likely to give a result above my actual weight as below it, more often than not the answer is somewhere between A and B.

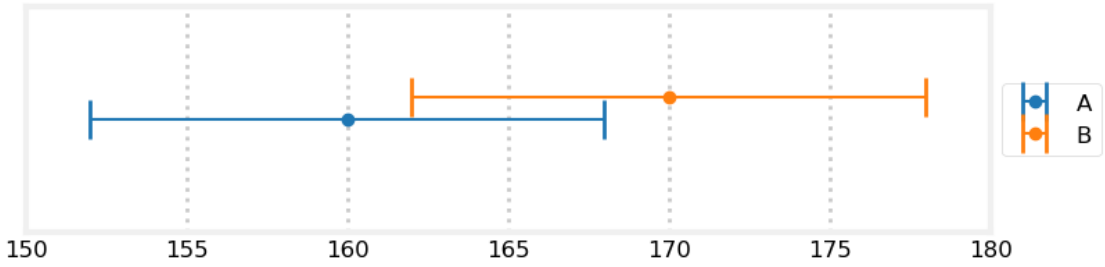
In mathematics this concept is formalized as *expected value*, and we will cover it in depth later. For now ask yourself what would be the 'usual' thing to happen if we took one million readings. Some of the times both scales will read too low, sometimes both will read too high, and the rest of the time they will straddle the actual weight. If they straddle the actual weight then certainly we should choose a number between A and B. If they don't straddle then we don't know if they are both too high or low, but by choosing a number between A and B we at least mitigate the effect of the worst measurement. For example, suppose our actual weight is 180 lbs. 160 lbs is a big error. But if we choose a weight between 160 lbs and 170 lbs our estimate will be better than 160 lbs. The same argument holds if both scales returned a value greater than the actual weight.

We will deal with this more formally later, but for now I hope it is clear that our best estimate is the average of A and B.

$$\frac{160 + 170}{2} = 165$$

We can look at this graphically. I have plotted the measurements of A and B with an assumed error of ± 8 lbs. The measurements falls between 160 and 170 so the only weight that makes sense must lie within 160 and 170 lbs.

```
In [3]: import kf_book.book_plots as book_plots
        from kf_book.book_plots import plot_errorbars
        plot_errorbars([(160, 8, 'A'), (170, 8, 'B')], xlims=(150, 180))
```




A word on how I generated this plot. I import code from the module book_plots in the `kf_book` subdirectory. Generating this plot takes a lot of boilerplate Python that isn't interesting to read. I take this tack often in the book. When the cell is run `plot_errorbars()` gets called and the plot is inserted into the book.

If this is your first time using Jupyter Notebook, the code above is in a *cell*. The text "In [2]:" labels this as a cell where you can enter input, and the number in the bracket denotes that this cell was run second. To run the cell, click on it with your mouse so that it has focus, then press CTRL+ENTER on the keyboard. As we continue you will be able to alter the code inside the cells and rerun them. Try changing the values "160", "170", and "8" to some other value and run the cell. The printed output should change depending on what you entered.

If you want to view the code for plot_errorbars, either open it in an editor, or create a new cell and type the function name followed by two question marks. Press Ctrl+Enter, and your browser will open a

window displaying the source code. This is a feature of Jupyter Notebooks. If you want to just view the documentation for the function, do the same but with one question mark.

```
plot_errorbars??
```

or

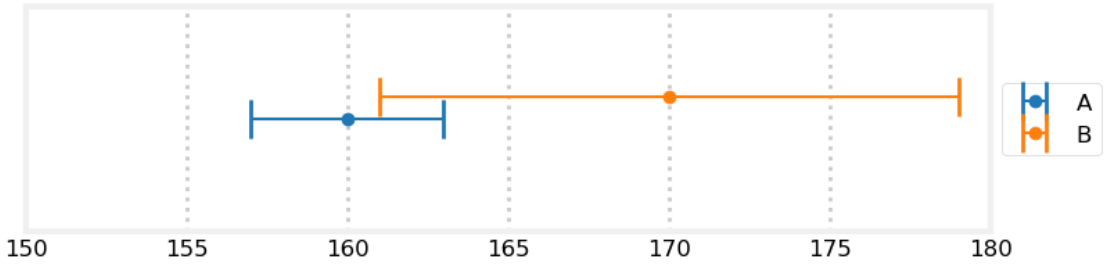
```
plot_errorbars?
```

So 165 lbs looks like a reasonable estimate, but there is more information here that we might be able to take advantage of. The only weights that are possible lie in the intersection between the error bars of A and B. For example, a weight of 161 lbs is impossible because scale B could not give a reading of 170 lbs with a maximum error of 8 pounds. Likewise a weight of 169 lbs is impossible because scale A could not give a reading of 160 lbs with a maximum error of 8 lbs. In this example the only possible weights lie in the range of 162 to 168 lbs.

That doesn't yet allow us to find a better weight estimate, but let's play 'what if' some more. What if we are now told that A is three times more accurate than B? Consider the 5 options we listed above. It still makes no sense to choose a number outside the range of A and B, so we will not consider those. It perhaps seems more compelling to choose A as our estimate - after all, we know it is more accurate, why not use it instead of B? Can B possibly improve our knowledge over A alone?

The answer, perhaps counter intuitively, is yes, it can. First, let's look at the same measurements of A=160 and B=170, but with the error of A $\pm$ 3 lbs and the error of B is 3 times as much, $\pm$ 9 lbs.

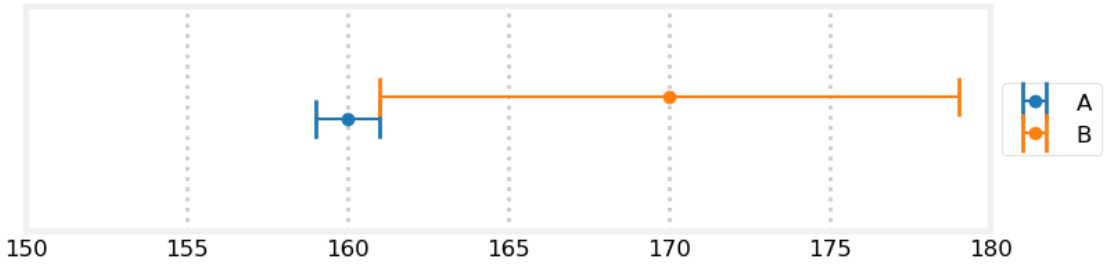
```
In [4]: plot_errorbars([(160, 3, 'A'), (170, 9, 'B')], xlims=(150, 180))
```




The overlap of the error bars of A and B are the only possible true weight. This overlap is smaller than the error in A alone. More importantly, in this case we can see that the overlap doesn't include 160 lbs or 165 lbs. If we only used the measurement from A because it is more accurate than B we would give an estimate of 160 lbs. If we average A and B we would get 165 lbs. Neither of those weights are possible given our knowledge of the accuracy of the scales. By including the measurement of B we would give an estimate somewhere between 161 lbs and 163 lbs, the limits of the intersections of the two error bars.

Let's take this to the extreme limits. Assume we know scale A is accurate to 1 lb. In other words, if we truly weigh 170 lbs, it could report 169, 170, or 171 lbs. We also know that scale B is accurate to 9 lbs. We do a weighing on each scale, and get A=160, and B=170. What should we estimate our weight to be? Let's look at that graphically.

```
In [5]: plot_errorbars([(160, 1, 'A'), (170, 9, 'B')], xlims=(150, 180))
```




Here we can see that the only possible weight is 161 lbs. This is an important result. With two relatively inaccurate sensors we are able to deduce an extremely accurate result.

**So two sensors, even if one is less accurate than the other, is better than one.** I will harp on this for the remainder of the book. We never throw information away, no matter how poor it is. We will be developing math and algorithms that allow us to include all possible sources of information to form the best estimate possible.

However, we have strayed from our problem. No customer is going to want to buy multiple scales, and besides, we initially started with an assumption that all scales were equally (in)accurate. This insight of using all measurements regardless of accuracy will play a large role later, so don't forget it.

What if I have one scale, but I weigh myself many times? We concluded that if we had two scales of equal accuracy we should average the results of their measurements. What if I weigh myself 10,000 times with one scale? We have already stated that the scale is equally likely to return a number too large as it is to return one that is too small. It is not that hard to prove that the average of a large number of weights will be very close to the actual weight, but let's write a simulation for now. I will use NumPy, part of the SciPy ecosystem for numerical computation.

```
In [6]: import numpy as np
        measurements = np.random.uniform(160, 170, size=10000)
        mean = measurements.mean()
        print('Average of measurements is {:.4f}'.format(mean))

Average of measurements is 164.9980
```

The exact number printed depends on your random number generator, but it should be very close to 165.

This code makes one assumption that probably isn't true - that the scale is as likely to read 160 as 165 for a true weight of 165 lbs. This is almost never true. Real sensors are more likely to get readings nearer the true value, and are less and less likely to get readings the further away from the true value it gets. We will cover this in detail in the Gaussian chapter. For now, I will use without further explanation the `numpy.random.normal()` function, which will produce more values nearer 165 lbs, and fewer further away. Take it on faith for now that this will produce noisy measurements similar to how a real scale works.

```
In [7]: mean = np.random.normal(165, 5, size=10000).mean()
        print('Average of measurements is {:.4f}'.format(mean))

Average of measurements is 164.9918
```

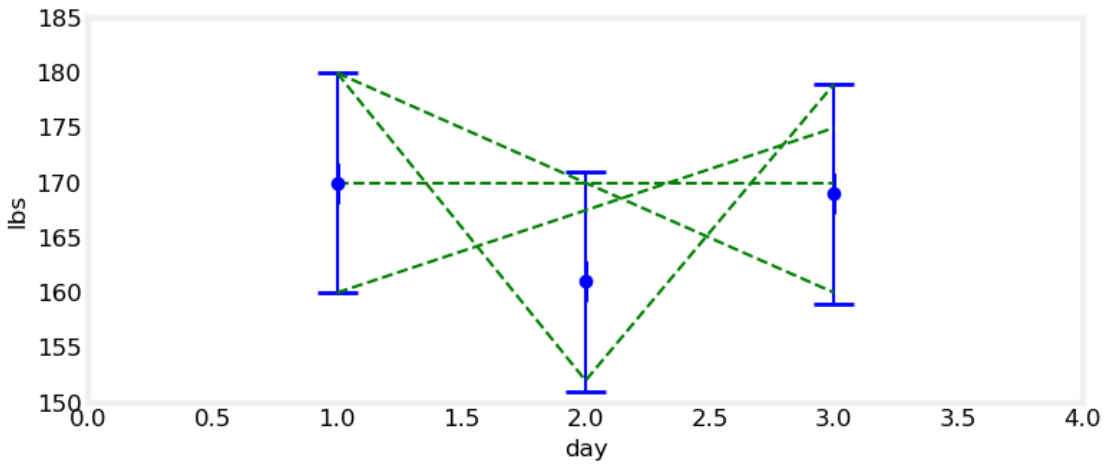
Again the answer is very close to 165.

Okay, great, we have an answer to our sensor problem! But it is not a very practical answer. No one has the patience to weigh themselves ten thousand, or even a dozen times.

So, let's play 'what if'. What if you measured your weight once a day, and got the readings 170, 161, and then 169. Did you gain weight, lose weight, or is this all just noisy measurements?

We really can't say. The first measurement was 170, and the last was 169, implying a 1 lb loss. But if the scale is only accurate to 10 lbs, that is explainable by noise. I could have actually gained weight; maybe my weight on day one was 165 lbs, and on day three it was 172. It is possible to get those weight readings with that weight gain. My scale tells me I am losing weight, and I am actually gaining weight! Let's look at that in a chart. I've plotted the measurements along with the error bars, and then some possible weight gain/losses that could be explained by those measurements in dotted green lines.

```
In [8]: import kf_book.gh_internal as gh
        gh.plot_hypothesis1()
```

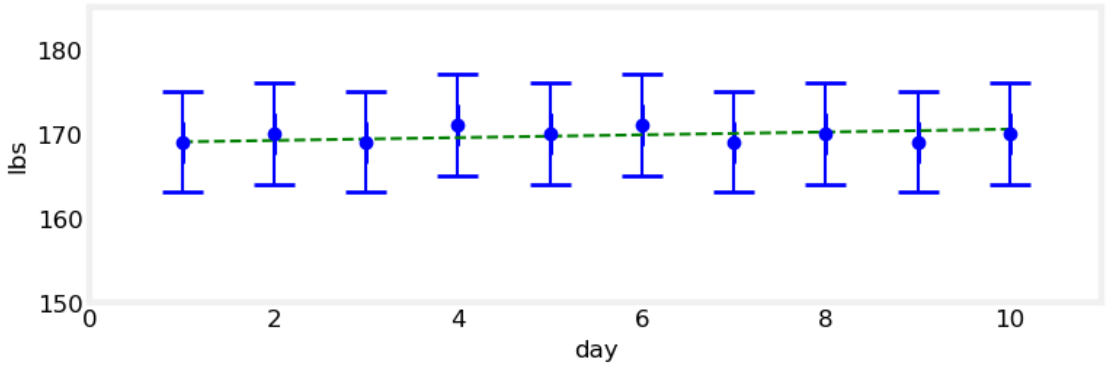



As we can see there is an extreme range of weight changes that could be explained by these three measurements. In fact, there are an infinite number of choices. Shall we give up? Not me! Recall that we are talking about measuring a human's weight. There is no reasonable way for a human to weigh 180 lbs on day 1 and 160 lbs on day 3. or to lose 30 lbs in one day only to gain it back the next (we will assume no amputations or other trauma has happened to the person).

The behavior of the physical system we are measuring should influence how we interpret the measurements. If we were weighing a rock each day we'd attribute all of the variance to noise. If we were weighing a cistern fed by rain and used for household chores we might believe such weight changes are real.

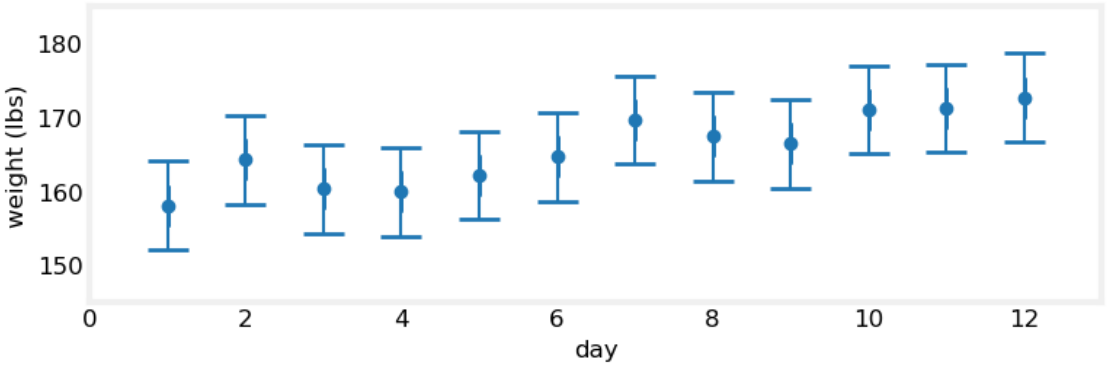
Suppose I take a different scale, and I get the following measurements: 169, 170, 169, 171, 170, 171, 169, 170, 169, 170. What does your intuition tell you? It is possible, for example, that you gained 1 lb each day, and the noisy measurements just happens to look like you stayed the same weight. Equally, you could have lost 1 lb a day and gotten the same readings. But is that likely? How likely is it to flip a coin and get 10 heads in a row? Not very likely. We can't prove it based solely on these readings, but it seems pretty likely that my weight held steady. In the chart below I've plotted the measurements with error bars, and a likely true weight in dashed green. This dashed line is not meant to be the 'correct' answer to this problem, merely one that is reasonable and could be explained by the measurement.

```
In [9]: gh.plot_hypothesis2()
```




Another what if: what if the readings were 158.0, 164.2, 160.3, 159.9, 162.1, 164.6, 169.6, 167.4, 166.4, 171.0? Let's look at a chart of that and then answer some questions.

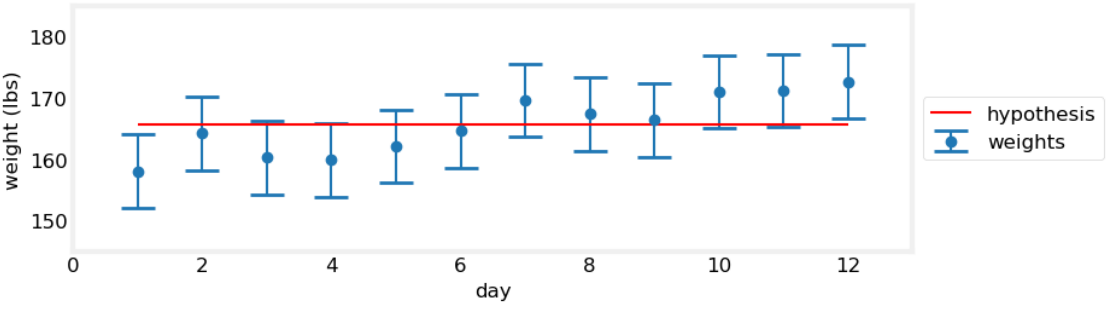
```
In [10]: gh.plot_hypothesis3()
```




Does it 'seem' likely that I lost weight and this is just really noisy data? Not really. Does it seem likely that I held the same weight? Again, no. This data trends upwards over time; not evenly, but definitely upwards. We can't be sure, but that looks like a weight gain, and a significant weight gain at that. Let's test this assumption with some more plots. It is often easier to 'eyeball' data in a chart versus a table.

So let's look at two hypotheses. First, let's assume our weight did not change. To get that number we agreed that we should average the measurements. Let's look at that.

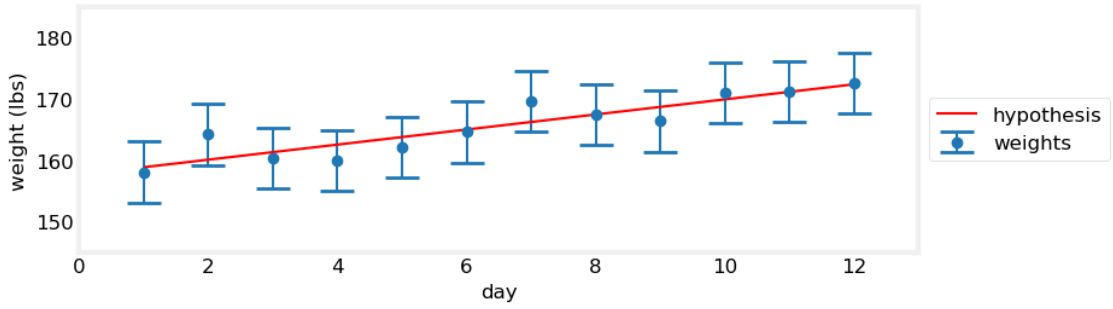
```
In [11]: gh.plot_hypothesis4()
```



That doesn't look very convincing. In fact, we can see that there is no horizontal line that we could draw that is inside all of the error bars.

Now, let's assume we gained weight. How much? I don't know, but NumPy does! We want to draw a line through the measurements that looks 'about' right. NumPy has functions that will do this according to a rule called "least squares fit". Let's not worry about the details of that computation (I use polyfit() if you are interested), and just plot the results.

```
In [12]: gh.plot_hypothesis5()
```



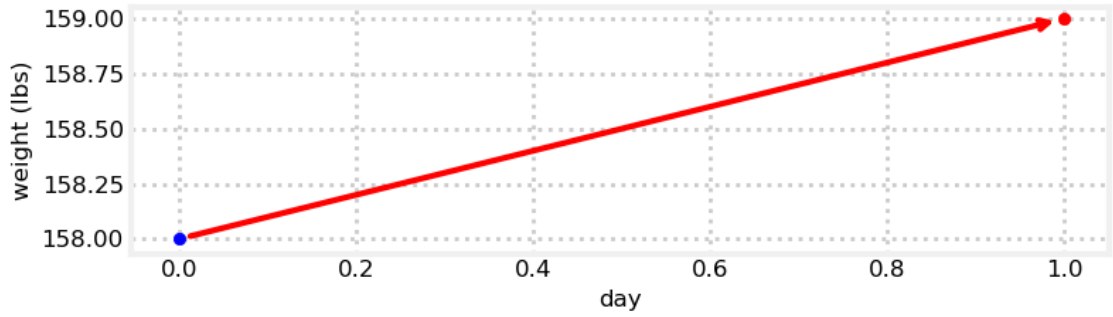

This looks much better, at least to my eyes. Notice now the hypothesis lies very close to each measurement, whereas in the previous plot the hypothesis was often quite far from the measurement. It seems far more likely to be true that I gained weight than I didn't gain any weight. Did I actually gain 13 lbs? Who can say? That seems impossible to answer.

"But is it impossible?" pipes up a co-worker.

Let's try something crazy. Let's assume that I know I am gaining about one lb a day. It doesn't matter how I know that right now, assume I know it is approximately correct. Maybe I am on a 6000 calorie a day diet, which would result in such a weight gain. Or maybe there is another way to estimate the weight gain. This is a thought experiment, the details are not important. Let's see if we can make use of such information if it was available.

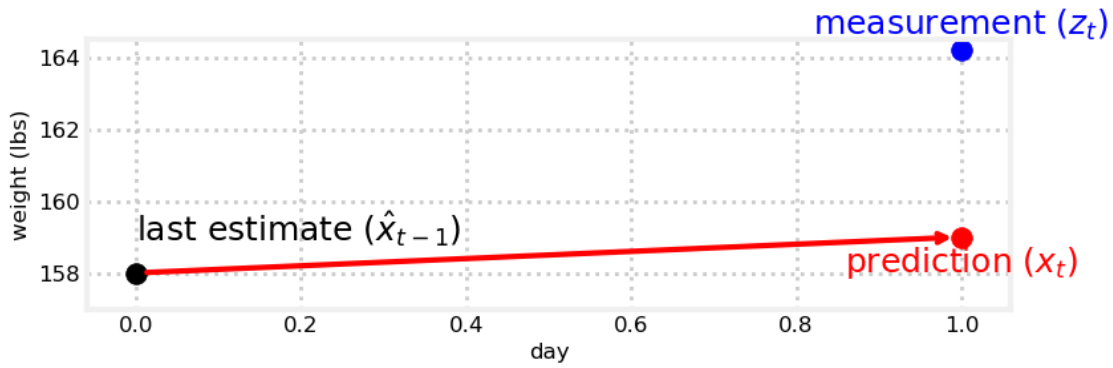
The first measurement was 158. We have no way of knowing any different, so let's accept that as our estimate. If our weight today is 158, what will it be tomorrow? Well, we think we are gaining weight at 1 lb/day, so our prediction is 159, like so:

```
In [13]: gh.plot_estimate_chart_1()
```




Okay, but what good is this? Sure, we could assume the 1 lb/day is accurate, and predict our weight for the next 10 days, but then why use a scale at all if we don't incorporate its readings? So let's look at the next measurement. We step on the scale again and it displays 164.2 lbs.

```
In [14]: gh.plot_estimate_chart_2()
```




We have a problem. Our prediction doesn't match our measurement. But, that is what we expected, right? If the prediction was always exactly the same as the measurement, it would not be capable of adding any information to the filter. And, of course, there would be no reason to ever measure since our predictions are perfect.

**The key insight to this entire book is in the next paragraph. Read it carefully!**

So what do we do? If we only form estimates from the measurement then the prediction will not affect the result. If we only form estimates from the prediction then the measurement will be ignored. If this is to work we need to take some kind of **blend of the prediction and measurement** (I've bolded the key point).

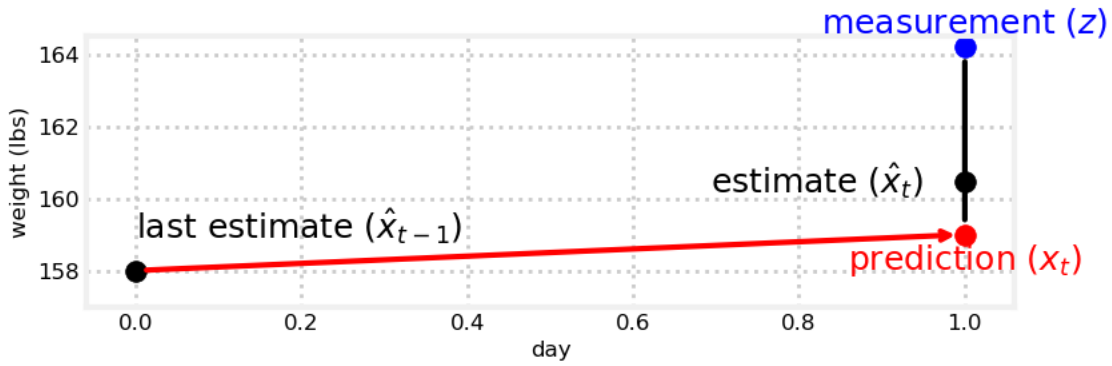
Blending two values - this sounds a lot like the two scale problem earlier. Using the same reasoning as before we can see that the only thing that makes sense is to choose a number between the prediction and the measurement. For example, an estimate of 165 makes no sense, nor does 157. Our estimates should lie between 159 (the prediction) and 164.2 (the measurement).

One more time, this is so important. We agreed that when presented two values with errors, we should form an estimate part way between the two values. It does not matter how those values were generated. In the start of the chapter we had two measurements, but now we have one measurement and one prediction. The reasoning, and hence the math is the same in both cases. We *never* throw information away. I mean it. I see so much commercial software that throws away noisy data. Don't do it! Our prediction of a weight gain might not be very accurate, but so long as there is some information we should use it.

I have to insist you stop and really think about this. All I have done is replaced an inaccurate scale with an inaccurate weight prediction based on human physiology. It is still data. Math doesn't know if the data came from a scale or a prediction. We have two pieces of data with a certain amount of noise, and we want to combine them. In the remainder of this book we are going to develop some fairly complicated math to perform this computation, but the math never cares where the data come from, it only makes computations based on the value and accuracy of those values.

Should the estimate be half way between the measurement and prediction? Maybe, but in general it seems like we might know that our prediction is more or less accurate compared to the measurements. Probably the accuracy of our prediction differs from the accuracy of the scale. Recall what we did when scale A was much more accurate than scale B - we scaled the answer to be closer to A than B. Let's look at that in a chart.

```
In [15]: gh.plot_estimate_chart_3()
```




Now let's try a randomly chosen number to scale our estimate: $\frac{4}{10}$. Our estimate will be four tenths the measurement and the rest will be from the prediction. In other words, we are expressing a belief here, a belief that the prediction is somewhat more likely to be correct than the measurement. We compute that as

$$\texttt{estimate} = \texttt{prediction} + \frac{4}{10}(\texttt{measurement} - \texttt{prediction})$$

The difference between the measurement and prediction is called the *residual*, which is depicted by the black vertical line in the plot above. This will become an important value to use later on, as it is an exact computation of the difference between measurements and the filter's output. Smaller residuals imply better performance.

Let's code that and see the results when we test it against the series of weights from above. We have to take into account one other factor. Weight gain has units of lbs/time, so to be general we will need to add a time step $t$, which we will set to 1 (day).

I hand generated the weight data to correspond to a true starting weight of 160 lbs, and a weight gain of 1 lb per day. In other words on the first day (day zero) the true weight is 160lbs, on the second day (day one, the first day of weighing) the true weight is 161 lbs, and so on.

We need to make a guess for the initial weight. It is too early to talk about initialization strategies, so for now I will assume 160 lbs.

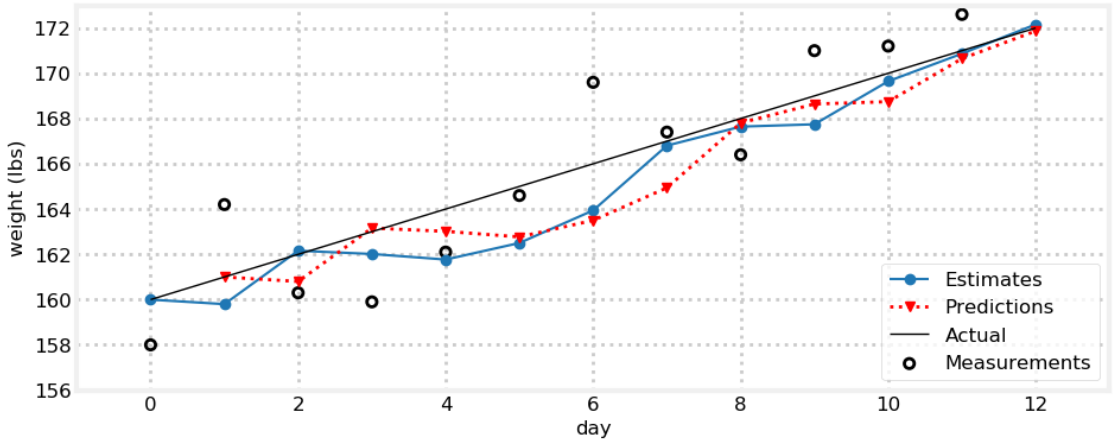
```
In [16]: from kf_book.book_plots import figsize
         import matplotlib.pyplot as plt

         weights = [158.0, 164.2, 160.3, 159.9, 162.1, 164.6,
                    169.6, 167.4, 166.4, 171.0, 171.2, 172.6]

         time_step = 1.0  # day
         scale_factor = 4.0/10

         def predict_using_gain_guess(estimated_weight, gain_rate, do_print=False):
             # storage for the filtered results
             estimates, predictions = [estimated_weight], []

             # most filter literature uses 'z' for measurements
             for z in weights:
                 # predict new position
                 predicted_weight = estimated_weight + gain_rate * time_step

                 # update filter
                 estimated_weight = predicted_weight + scale_factor * (z - predicted_weight)
```

```
            # save and log
            estimates.append(estimated_weight)
            predictions.append(predicted_weight)
            if do_print:
                gh.print_results(estimates, predicted_weight, estimated_weight)

        return estimates, predictions

    initial_estimate = 160.
    estimates, predictions = predict_using_gain_guess(
        estimated_weight=initial_estimate, gain_rate=1, do_print=True)
```

```
previous estimate: 160.00, prediction: 161.00, estimate 159.80
previous estimate: 159.80, prediction: 160.80, estimate 162.16
previous estimate: 162.16, prediction: 163.16, estimate 162.02
previous estimate: 162.02, prediction: 163.02, estimate 161.77
previous estimate: 161.77, prediction: 162.77, estimate 162.50
previous estimate: 162.50, prediction: 163.50, estimate 163.94
previous estimate: 163.94, prediction: 164.94, estimate 166.80
previous estimate: 166.80, prediction: 167.80, estimate 167.64
previous estimate: 167.64, prediction: 168.64, estimate 167.75
previous estimate: 167.75, prediction: 168.75, estimate 169.65
previous estimate: 169.65, prediction: 170.65, estimate 170.87
previous estimate: 170.87, prediction: 171.87, estimate 172.16
```

```
In [17]: # plot results
         book_plots.set_figsize(10)
         gh.plot_gh_results(weights, estimates, predictions, [160, 172])
         weights
```

```
Out[17]: [158.0,
          164.2,
          160.3,
          159.9,
          162.1,
          164.6,
          169.6,
          167.4,
          166.4,
          171.0,
          171.2,
          172.6]
```



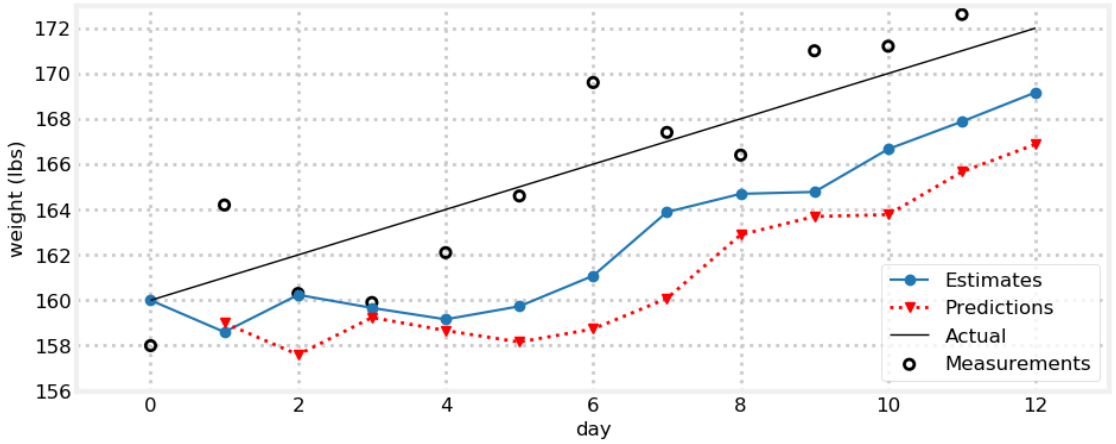

That is pretty good! There is a lot of data here, so let's talk about how to interpret it. The thick blue line shows the estimate from the filter. It starts at day 0 with the initial guess of 160 lbs. The red line shows the prediction that is made from the previous day's weight. So, on day one the previous weight was 160 lbs, the weight gain is 1 lb, and so the first prediction is 161 lbs. The estimate on day one is then part way between the prediction and measurement at 159.8 lbs. Below the chart is a print out of the previous weight, predicted weight, and new estimate for each day. Finally, the thin black line shows the actual weight gain of the person being weighed.

Walk through this for each day, ensuring you understand how the prediction and estimates were formed at each step. Note how the estimate always falls between the measurement and prediction.

The estimates are not a straight line, but they are straighter than the measurements and somewhat close to the trend line we created. Also, it seems to get better over time.

The results of the filter may strike you as quite silly; of course the data will look good if we assume the conclusion, that our weight gain is around 1 lb/day! Let's see what the filter does if our initial guess is bad. Let's predict that there is a weight loss of 1 lb a day:

```
In [18]: e, p = predict_using_gain_guess(initial_estimate, -1.)
         gh.plot_gh_results(weights, e, p, [160, 172])
```




That is not so impressive. The estimates quickly divert from the measurements. Clearly a filter that requires us to correctly guess a rate of change is not very useful. Even if our initial guess was correct, the

filter will fail as soon as that rate of change changes. If I stop overeating the filter will have extreme difficulty in adjusting to that change. Note that it is adjusting! The estimates are climbing even though we tell it we are losing 1 lb a day. It just can't adjust fast enough.

But, 'what if'? What if instead of leaving the weight gain at the initial guess of 1 lb (or whatever), we compute it from the existing measurements and estimates. On day one our estimate for the weight is:

$$(160 + 1) + \frac{4}{10}(158 - 161) = 159.8$$

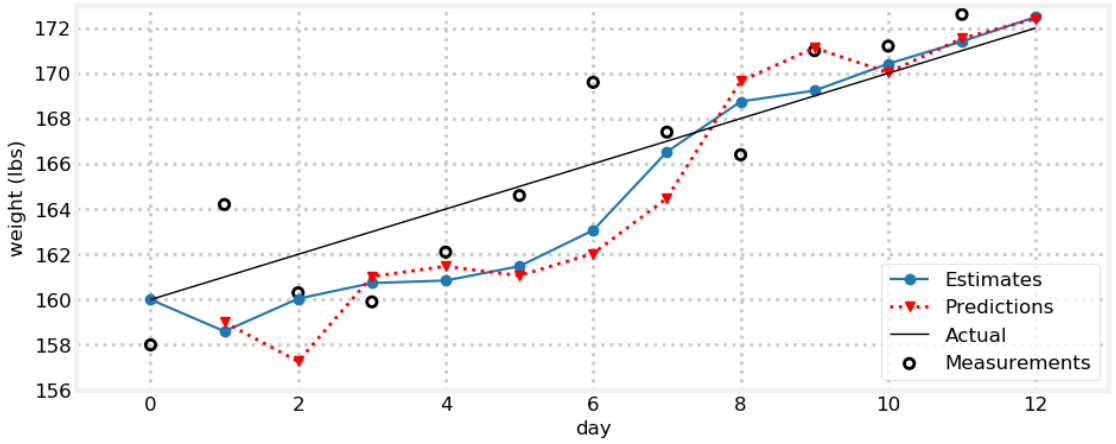
On the next day we measure 164.2, which implies a weight gain of 4.4 lbs (since 164.2 - 159.8 = 4.4), not 1. Can we use this information somehow? It seems plausible. After all, the weight measurement itself is based on a real world measurement of our weight, so there is useful information. Our estimate of our weight gain may not be perfect, but it is surely better than just guessing our gain is 1 lb. Data is better than a guess, even if it is noisy.

People really balk at this point, so make sure you are in agreement. Two noisy measurements of weight give us an implied weight gain/loss. That estimate is going to be very inaccurate if the measurements are inaccurate, but there is still information in this computation. Imagine weighing a cow with a scale accurate to 1 lb, and it shows that the cow gained 10 lbs. The cow might have gained 8 lbs up to 12 lbs, depending on the errors, but we know it gained weight, and roughly how much. This is information. What do we do with information? Never throw it away!

Back to my diet. Should we set the new gain/day to 4.4 lbs? Yesterday we thought the weight gain was 1 lb, today we think it is 4.4 lbs. We have two numbers, and want to combine them somehow. Hmm, sounds like our same problem again. Let's use our same tool, and the only tool we have so far - pick a value part way between the two. This time I will use another arbitrarily chosen number, $\frac{1}{3}$. The equation is identical as for the weight estimate except we have to incorporate time because this is a rate (gain/day):

$$\text{new gain} = \text{old gain} + \frac{1}{3}\frac{\text{measurement - predicted weight}}{\text{1 day}}$$

In [19]:
```
weight = 160.  # initial guess
gain_rate = -1.0  # initial guess

time_step = 1.
weight_scale = 4./10
gain_scale = 1./3
estimates = [weight]
predictions = []

for z in weights:
    # prediction step
    weight = weight + gain_rate*time_step
    gain_rate = gain_rate
    predictions.append(weight)

    # update step
    residual = z - weight

    gain_rate = gain_rate + gain_scale   * (residual/time_step)
    weight    = weight    + weight_scale * residual

    estimates.append(weight)

gh.plot_gh_results(weights, estimates, predictions, [160, 172])
```




I think this is starting to look really good. Because of the poor initial guess of the weight gain being -1 it takes the filter several days to accurately predict the weight, but once it does that it starts to accurately track the weight. We used no methodology for choosing our scaling factors of $\frac{4}{10}$ and $\frac{1}{3}$ (actually, they are poor choices for this problem), but otherwise all of the math followed from very reasonable assumptions. Recall that you can change the value of the parameter `time_step` to a larger value and re-run the cell if you want to see the plot drawn step-by-step.

One final point before we go on. In the prediction step I wrote the line

```
gain_rate = gain_rate
```

This obviously has no effect, and can be removed. I wrote this to emphasize that in the prediction step you need to predict the next value for all variables, both `weight` and `gain_rate`. This will become relevant shortly. In this case we are assuming that the gain does not vary, but when we generalize this algorithm we will remove that assumption.

## 1.2 The g-h Filter

This algorithm is known as the g-h filter or the $\alpha$-$\beta$ filter. $g$ and $h$ refer to the two scaling factors that we used in our example. $g$ is the scaling we used for the measurement (weight in our example), and $h$ is the scaling for the change in measurement over time (lbs/day in our example). $\alpha$ and $\beta$ are just different names used for this factors.

This filter is the basis for a huge number of filters, including the Kalman filter. In other words, the Kalman filter is a form of the g-h filter, which I will prove later in the book. So is the Least Squares filter, which you may have heard of, and so is the Benedict-Bordner filter, which you probably have not. Each filter has a different way of assigning values to $g$ and $h$, but otherwise the algorithms are identical. For example, the Benedict-Bordner filter assigns a constant to $g$ and $h$, constrained to a certain range of values. Other filters such as the Kalman will vary $g$ and $h$ dynamically at each time step.

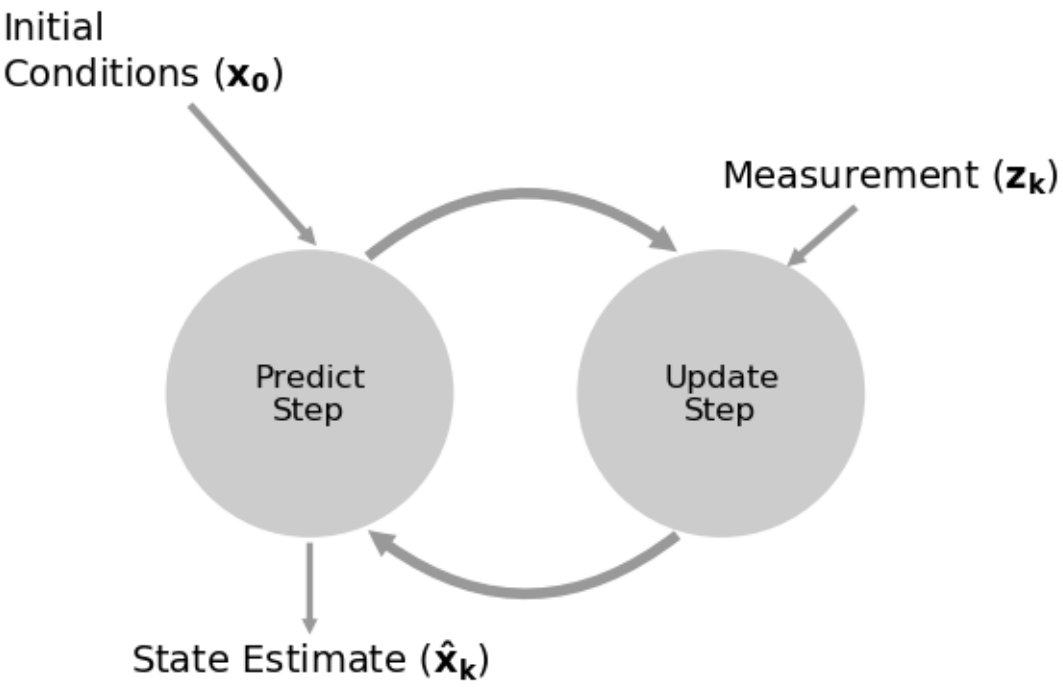
**Let me repeat the key points as they are so important**. If you do not understand these you will not understand the rest of the book. If you do understand them, then the rest of the book will unfold naturally for you as mathematical elaborations to various 'what if' questions we will ask about $g$ and $h$. The math may look profoundly different, but the algorithm will be exactly the same.

- Multiple data points are more accurate than one data point, so throw nothing away no matter how inaccurate it is.
- Always choose a number part way between two data points to create a more accurate estimate.
- Predict the next measurement and rate of change based on the current estimate and how much we think it will change.

- The new estimate is then chosen as part way between the prediction and next measurement scaled by how accurate each is.

Let's look at a visual depiction of the algorithm.

In [20]:
```
book_plots.predict_update_chart()
```




Let me introduce some more formal terminology. The *system* is the object that we want to estimate. In this chapter the system is whatever we are trying to weigh. Some texts call this the *plant*. That terminology comes from control system theory. https://en.wikipedia.org/wiki/Plant_(control_theory)

The *state* of the system is the current configuration or values of that system that is of interest to us. We are interested only in the weight reading. If I put a 100 kg weight on the scale, the state is 100kg. We define the state based on what is relevant to us. The color of the scale is irrelevant to us so we do not include those values in the state. A QA engineer for the manufacturer might include color in the state so that she can track and control the manufacturing process.

The *measurement* is a measured value of the system. Measurements can be inaccurate, so it may not have the same value as the state.

The *state estimate* is our filter's estimate of the state. For example, for the 100 kg weight our estimate might be 99.327 kg due to sensor errors. This is commonly abbreviated to *estimate*, and I have done that in this chapter.

In other words, the state should be understood as the actual value of the system. This value is usually *hidden* to us. If I stepped on a scale you'd then have a *measurement*. We call this *observable* since you can directly observe this measurement. In contrast, you can never directly observe my weight, you can only measure it.

This language of *hidden* and *observable* is important. Any estimation problem consists of forming an estimate of a hidden state via observable measurements. If you read the literature these terms are used when defining a problem, so you need to be comfortable with them.

We use a *process model* to mathematically model the system. In this chapter our process model is the assumption that my weight today is yesterday's weight plus my weight gain for the last day. The process model does not model or otherwise account for the sensors. Another example would be a process model for an automobile. The process model might be "distance equals velocity times time. This model is not perfect as the velocity of a car can vary over a non-zero amount of time, the tires can slip on the road, and so on. The *system error* or *process error* is the error in this model. We never know this value exactly; if we did we could refine our model to have zero error. Some texts use *plant model* and *plant error*. You may also see *system model*. They all mean the same thing.

The predict step is known as *system propagation*. It uses the *process model* to form a new *state estimate*. Because of the *process error* this estimate is imperfect. Assuming we are tracking data over time, we say we *propagate* the state into the future. Some texts call this the *evolution*.

The update step is known as the *measurement update*. One iteration of the system propagation and measurement update is known as an *epoch*.

Now let's explore a few different problem domains to better understand this algorithm. Consider the problem of trying to track a train on a track. The track constrains the position of the train to a very specific region. Furthermore, trains are large and slow. It takes many minutes for them to slow down or speed up significantly. So, if I know that the train is at kilometer marker 23 km at time t and moving at 18 kph, I can be extremely confident in predicting its position at time t + 1 second. Why is that important? Suppose we can only measure its position with an accuracy of ± 250 meters. The train is moving at 18 kph, which is 5 meters per second. At t+1 seconds the train will be at 23.005 km yet the measurement could be anywhere from 22.755 km to 23.255 km. So if the next measurement says the position is at 23.4 we know that must be inaccurate. Even if at time t the engineer slammed on the brakes the train will still be very near to 23.005 km because a train cannot slow down very much in 1 second. If we were to design a filter for this problem (and we will a bit further in the chapter!) we would want to design a filter that gave a very high weighting to the prediction vs the measurement.

Now consider the problem of tracking a thrown ball. We know that a ballistic object moves in a parabola in a vacuum when in a gravitational field. But a ball thrown on Earth is influenced by air drag, so it does not travel in a perfect parabola. Baseball pitchers take advantage of this fact when they throw curve balls. Let's say that we are tracking the ball inside a stadium using computer vision, something I do at work. The accuracy of the computer vision tracking might be modest, but predicting the ball's future positions by assuming that it is moving on a parabola is not extremely accurate either. In this case we'd probably design a filter that gave roughly equal weight to the measurement and the prediction.

Now consider trying to track a helium party balloon in a hurricane. We have no legitimate model that would allow us to predict the balloon's behavior except over very brief time scales (we know the balloon cannot go 10 miles in 1 second, for example). In this case we would design a filter that emphasized the measurements over the predictions.

Most of this book is devoted to expressing the concerns in the last three paragraphs mathematically, which then allows us to find an optimal solution (in some mathematical sense). In this chapter we will merely be assigning different values to $g$ and $h$ in a more intuitive, and thus less optimal way. But the fundamental idea is to blend somewhat inaccurate measurements with somewhat inaccurate models of how the systems behaves to get a filtered estimate that is better than either information source by itself.

We can express this as an algorithm:

**Initialization**

```
1. Initialize the state of the filter
2. Initialize our belief in the state
```

**Predict**

```
1. Use system behavior to predict state at the next time step
2. Adjust belief to account for the uncertainty in prediction
```

**Update**

```
1. Get a measurement and associated belief about its accuracy
```

```
2. Compute residual between estimated state and measurement
3. New estimate is somewhere on the residual line
```

We will use this same algorithm throughout the book, albeit with some modifications.

## 1.3 Notation

I'll begin to introduce the notations and variable names used in the literature. Some of this was already used in the above charts. Measurement is typically denoted $z$ and that is what we will use in this book (some literature uses $y$). Subscript $k$ indicates the time step, so $z_k$ is the data for this time step. A bold font denotes a vector or matrix. So far we have only considered having one sensor, and hence one sensor measurement, but in general we may have $n$ sensors and $n$ measurements. $\mathbf{x}$ denotes our state, and is bold to denote that it is a vector. For our scale example, it represents both the initial weight and initial weight gain rate, like so:

$$\mathbf{x} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

Here I use Newton's notation of a dot over the x to denote velocity. More precisely, the dot implies the derivative of x with respect to time, which of course is the velocity. For a weight of 62 kg with a gain of 0.3 kg/day we have

$$\mathbf{x} = \begin{bmatrix} 62 \\ 0.3 \end{bmatrix}$$

So, the algorithm is simple. The state is initialized with $\mathbf{x_0}$, the initial estimate. We then enter a loop, predicting the state for time or step $k$ from the values from time (or step) $k-1$. We then get the measurement $z_k$ and choose some intermediate point between the measurements and prediction, creating the estimate $\mathbf{x}_k$.

## 1.4 Exercise: Write Generic Algorithm

In the example above, I explicitly coded this to solve the weighing problem that we've been discussing throughout the chapter. For example, the variables are named "weight_scale", "gain", and so on. I did this to make the algorithm easy to follow - you can easily see that we correctly implemented each step. But, that is code written for exactly one problem, and the algorithm is the same for any problem. So let's rewrite the code to be generic - to work with any problem. Use this function signature:

```
def g_h_filter(data, x0, dx, g, h, dt):
    """
    Performs g-h filter on 1 state variable with a fixed g and h.

    'data' contains the data to be filtered.
    'x0' is the initial value for our state variable
    'dx' is the initial change rate for our state variable
    'g' is the g-h's g scale factor
    'h' is the g-h's h scale factor
    'dt' is the length of the time step
    """
```

Return the data as a NumPy array, not a list. Test it by passing in the same weight data as before, plot the results, and visually determine that it works.

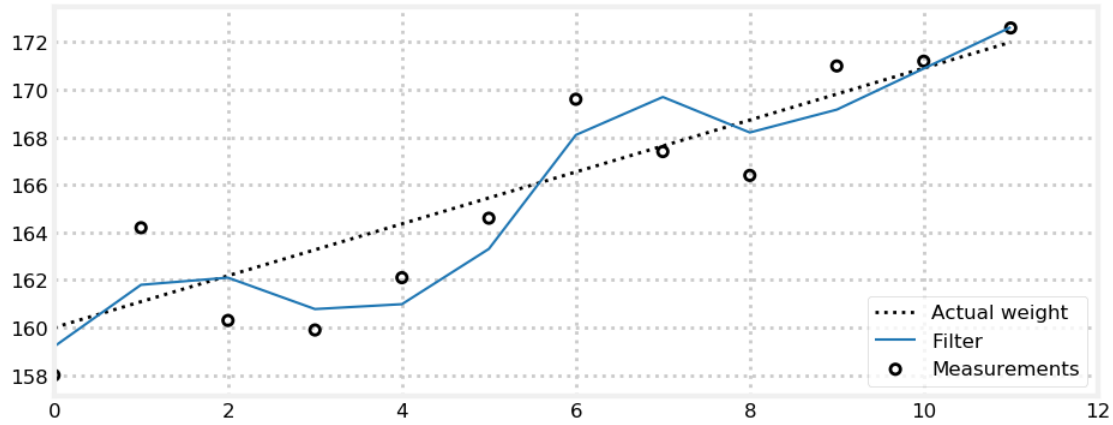
```
In [21]: from kf_book.gh_internal import plot_g_h_results
         def g_h_filter(data, x0, dx, g, h, dt):
             pass #  your solution here
```

```
# uncomment to run the filter and plot the results
#book_plots.plot_track([0, 11], [160, 172], label='Actual weight')
#data = g_h_filter(data=weights, x0=160., dx=1., g=6./10, h=2./3, dt=1.)
#plot_g_h_results(weights, data)
```

### 1.4.1 Solution and Discussion

```
In [22]: import matplotlib.pylab as pylab

         def g_h_filter(data, x0, dx, g, h, dt=1.):
             x_est = x0
             results = []
             for z in data:
                 # prediction step
                 x_pred = x_est + (dx*dt)
                 dx = dx

                 # update step
                 residual = z - x_pred
                 dx = dx + h * (residual) / dt
                 x_est = x_pred + g * residual
                 results.append(x_est)
             return np.array(results)

         book_plots.plot_track([0, 11], [160, 172], label='Actual weight')
         data = g_h_filter(data=weights, x0=160., dx=1., g=6./10, h=2./3, dt=1.)
         plot_g_h_results(weights, data)
         print(weights)
         print(data)
```

```
[158.0, 164.2, 160.3, 159.9, 162.1, 164.6, 169.6, 167.4, 166.4, 171.0, 171.2, 172.6]
[159.2   161.8   162.1   160.78  160.985 163.311 168.1   169.696
 168.204 169.164 170.892 172.629]
```




This should have been straightforward. I just subtituted the variable names `x0`, `dx`, etc., for the variable names in the weight gain code. Nothing else needed to change.

## 1.5 Choice of $g$ and $h$

The g-h filter is not one filter - it is a classification for a family of filters. Eli Brookner in *Tracking and Kalman Filtering Made Easy* lists 11, and I am sure there are more. Not only that, but each type of filter has numerous subtypes. Each filter is differentiated by how $g$ and $h$ are chosen. So there is no 'one size fits all' advice that I can give here. Some filters set $g$ and $h$ as constants, others vary them dynamically. The Kalman filter varies them dynamically at each step. Some filters allow $g$ and $h$ to take any value within a range, others constrain one to be dependent on the other by some function $f()$, where $g = f(h)$.

The topic of this book is not the entire family of g-h filters; more importantly, we are interested in the *Bayesian* aspect of these filters, which I have not addressed yet. Therefore I will not cover selection of $g$ and $h$ in depth. *Tracking and Kalman Filtering Made Easy* is an excellent resource for that topic. If this strikes you as an odd position for me to take, recognize that the typical formulation of the Kalman filter does not use $g$ and $h$ at all. The Kalman filter is a g-h filter because it mathematically reduces to this algorithm. When we design the Kalman filter we use design criteria that can be mathematically reduced to $g$ and $h$, but the Kalman filter form is usually a much more powerful way to think about the problem. Don't worry if this is not too clear right now, it will clear once we develop the Kalman filter theory.

It is worth seeing how varying $g$ and $h$ affects the results, so we will work through some examples. This will give us strong insight into the fundamental strengths and limitations of this type of filter, and help us understand the behavior of the rather more sophisticated Kalman filter.

## 1.6 Exercise: create measurement function

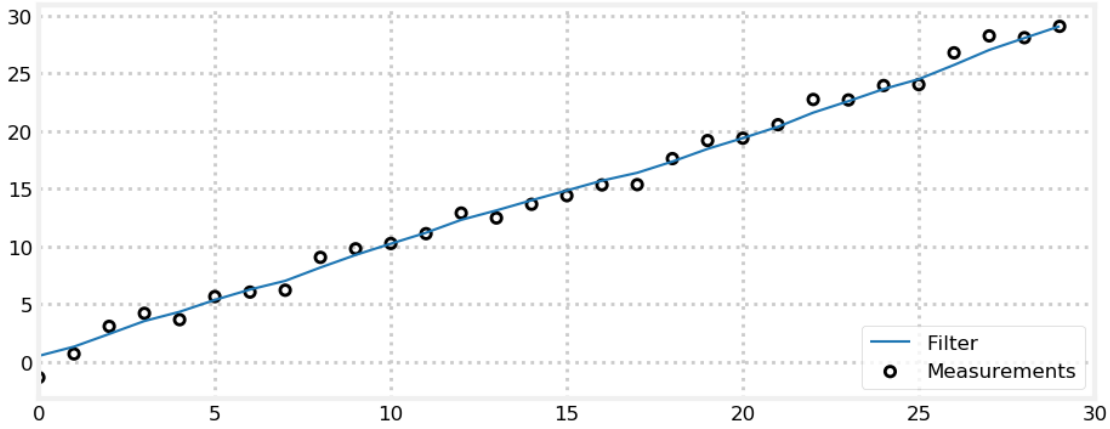
Now let's write a function that generates noisy data for us. In this book I model a noisy signal as the signal plus white noise. We've not yet covered the statistics to fully understand the definition of white noise. In essence, think of it as data that randomly varies higher and lower than the signal with no pattern. We say that it is a serially uncorrelated random variable with zero mean and finite variance. If you don't follow that, you will by the end of the *Gaussians* chapter. You may not be successful at this exercise if you have no knowledge of statistics. If so, just read the solution and discussion.

White noise can be generated by `numpy.random.randn()`. We want a function that we call with the starting value, the amount of change per step, the number of steps, and the amount of noise we want to add. It should return a list of the data. Test it by creating 30 points, filtering it with `g_h_filter()`, and plot the results with `plot_g_h_results()`.

```
In [23]: # your code here
```

### 1.6.1 Solution

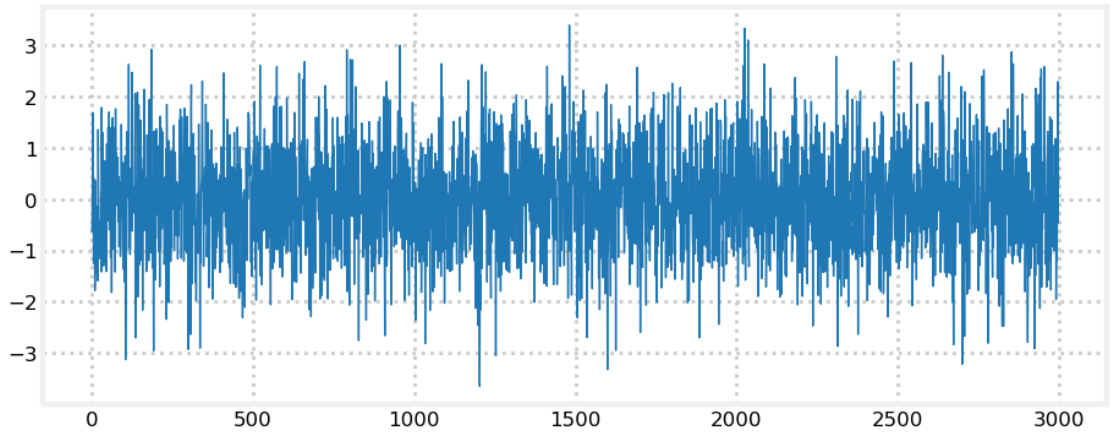
```
In [24]: from numpy.random import randn
         def gen_data(x0, dx, count, noise_factor):
             return [x0 + dx*i + randn()*noise_factor for i in range(count)]

         measurements = gen_data(0, 1, 30, 1)
         data = g_h_filter(data=measurements, x0=0., dx=1., dt=1., g=.2, h=0.02)
         plot_g_h_results(measurements, data)
```




### 1.6.2 Discussion

`randn()` returns random numbers centered around 0 - it is just as likely to be greater than zero as under zero. It varies by *one standard deviation* - don't worry if you don't know what that means. I've plotted 3000 calls to `randn()` - you can see that the values are centered around zero and mostly range from a bit under -1 to a bit more than +1, though occasionally they are much larger.

```
In [25]: plt.plot([randn() for _ in range(3000)], lw=1);
```



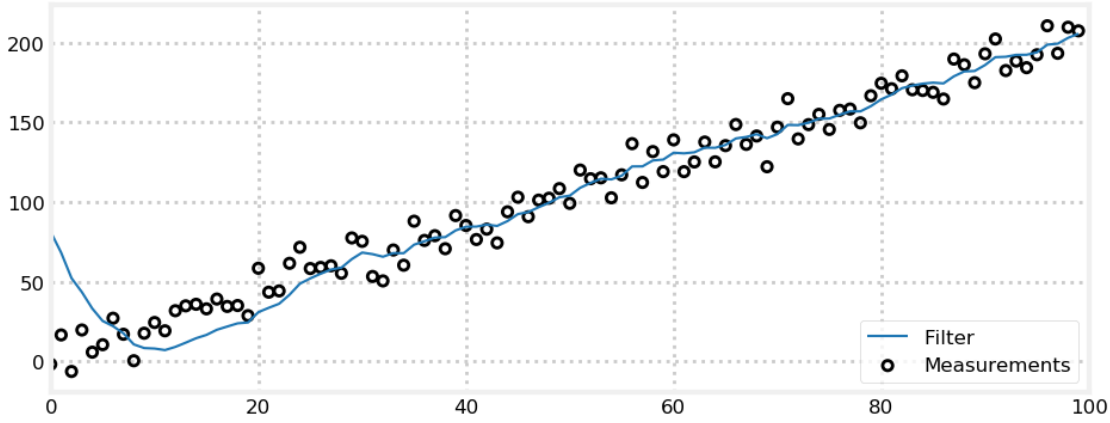
## 1.7 Exercise: Bad Initial Conditions

Now write code that uses `gen_data` and `g_h_filter` to filter 100 data points that starts at 5, has a derivative of 2, a noise scaling factor of 10, and uses g=0.2 and h=0.02. Set your initial guess for x to be 100.

```
In [26]: # your code here
```

### 1.7.1 Solution and Discussion

```
In [27]: zs = gen_data(x0=5., dx=2., count=100, noise_factor=10)
         data = g_h_filter(data=zs, x0=100., dx=2., dt=1., g=0.2, h=0.02)
         plot_g_h_results(measurements=zs, filtered_data=data)
```




The filter starts out with estimates that are far from the measured data due to the bad initial guess of 100. You can see that it 'rings' before settling in on the measured data. 'Ringing' means that the signal overshoots and undershoots the data in a sinusoidal type pattern. This is a very common phenomena in filters, and a lot of work in filter design is devoted to minimizing ringing. That is a topic that we are not yet prepared to address, but I wanted to show you the phenomenon.

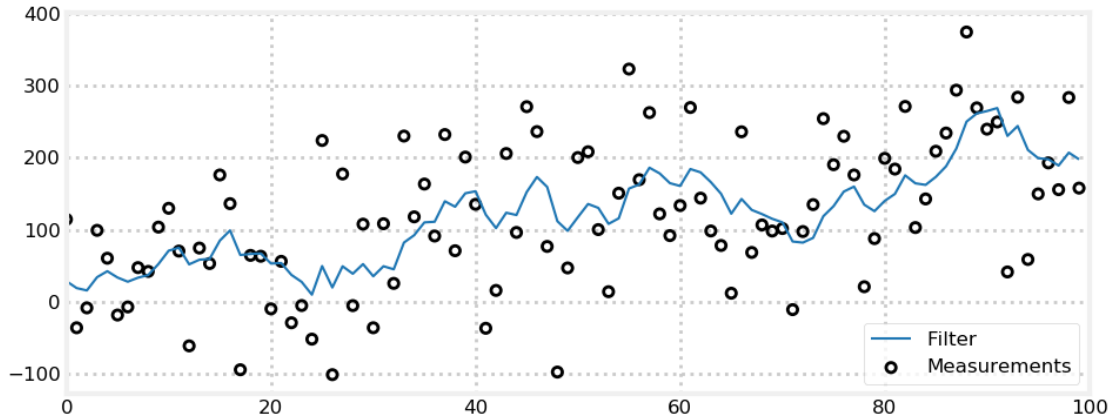
## 1.8 Exercise: Extreme Noise

Rerun the same test, but this time use a noise factor of 100. Remove the initial condition ringing by changing the initial condition from 100 down to 5.

```
In [28]: # your code here
```

### 1.8.1 Solution and Discussion

```
In [29]: zs = gen_data(x0=5., dx=2., count=100, noise_factor=100)
         data = g_h_filter(data=zs, x0=5., dx=2., g=0.2, h=0.02)
         plot_g_h_results(measurements=zs, filtered_data=data)
```



This doesn't look so wonderful to me. We can see that perhaps the filtered signal varies less than the noisy signal, but it is far from the straight line. If we were to plot just the filtered result no one would guess that the signal starts at 5 and increments by 2 at each time step. And while in locations the filter does seem to reduce the noise, in other places it seems to overshoot and undershoot.

At this point we don't know enough to really judge this. We added **a lot** of noise; maybe this is as good as filtering can get. However, the existence of the multitude of chapters beyond this one should suggest that we can do much better.

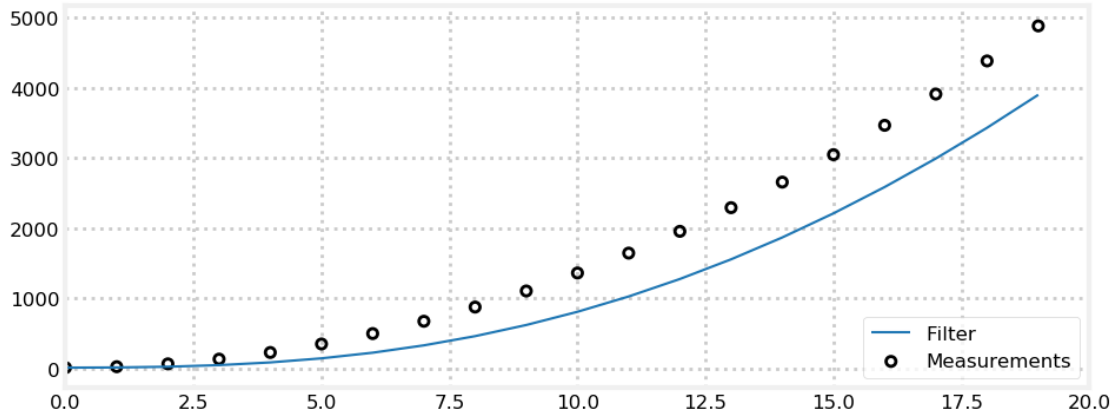
## 1.9 Exercise: The Effect of Acceleration

Write a new data generation function that adds in a constant acceleration factor to each data point. In other words, increment dx as you compute each data point so that the velocity (dx) is ever increasing. Set the noise to 0, $g = 0.2$ and $h = 0.02$ and plot the results using `plot_g_h_results` or your own routine. Play around with different accererations and times steps. Explain what you see.

```
In [30]: # your code here
```

### 1.9.1 Solution and Discussion

```
In [31]: def gen_data(x0, dx, count, noise_factor, accel=0.):
             zs = []
             for i in range(count):
                 zs.append(x0 + accel * (i**2) / 2 + dx*i + randn()*noise_factor)
                 dx += accel
             return zs

         predictions = []
         zs = gen_data(x0=10., dx=0., count=20, noise_factor=0, accel=9.)
         data = g_h_filter(data=zs, x0=10., dx=0., g=0.2, h=0.02)
         plot_g_h_results(measurements=zs, filtered_data=data)
```




Each prediction lags behind the signal. If you think about what is happening this makes sense. Our model assumes that velocity is constant. The g-h filter computes the first derivative of $x$ (we use $\dot{x}$ to denote the derivative) but not the second derivative $\ddot{x}$. So we are assuming that $\ddot{x} = 0$. At each prediction step we predict the new value of x as $x + \dot{x} * t$. But because of the acceleration the prediction must necessarily fall

behind the actual value. We then try to compute a new value for $\dot{x}$, but because of the $h$ factor we only partially adjust $\dot{x}$ to the new velocity. On the next iteration we will again fall short.

Note that there is no adjustment to $g$ or $h$ that we can make to correct this problem. This is called the *lag error* or *systemic error* of the system. It is a fundamental property of g-h filters. Perhaps your mind is already suggesting solutions or workarounds to this problem. As you might expect, a lot of research has been devoted to this problem, and we will be presenting various solutions to this problem in this book. > The 'take home' point is that the filter is only as good as the mathematical model used to express the system.

## 1.10 Exercise: Varying $g$

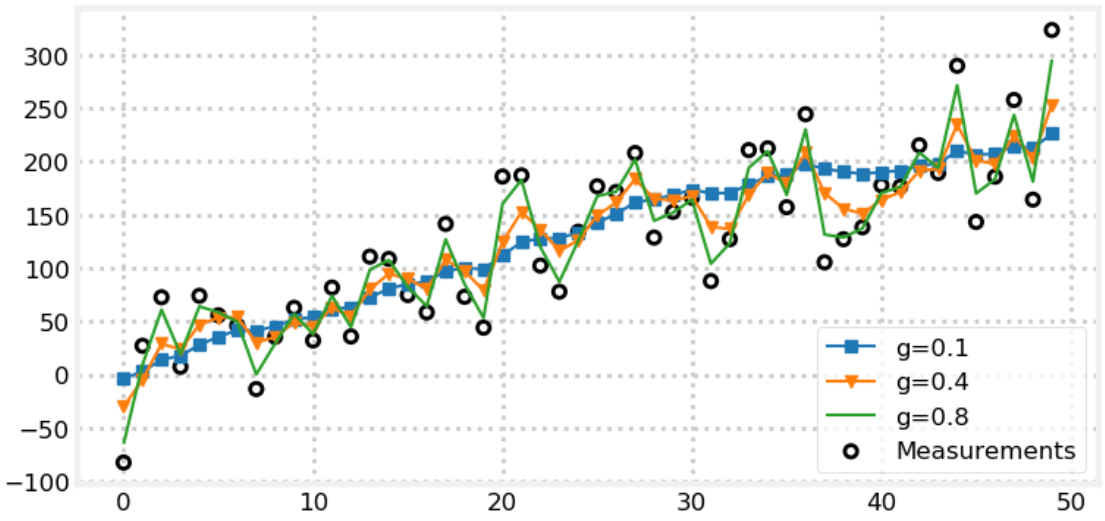
Now let's look at the effect of varying $g$. Before you perform this exercise, recall that $g$ is the scale factor for choosing between the measurement and prediction. What do you think the effect of a large value of $g$ will be? A small value?

Now, let the `noise_factor=50` and `dx=5`. Plot the results of $g = 0.1,\ 0.4,$ and $0.8$.

```
In [32]: # your code here
```

### 1.10.1 Solution and Discussion

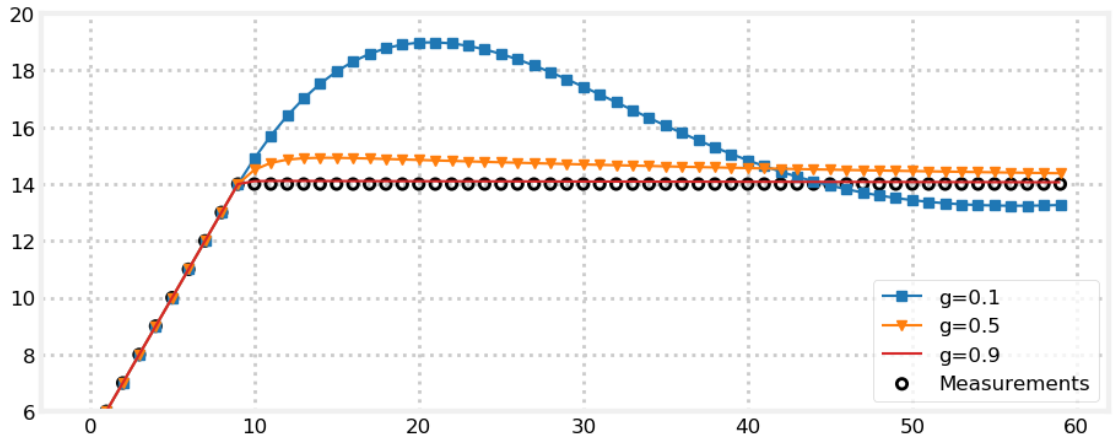
```
In [33]: np.random.seed(100)
         zs = gen_data(x0=5., dx=5., count=50, noise_factor=50)
         data1 = g_h_filter(data=zs, x0=0., dx=5., dt=1., g=0.1, h=0.01)
         data2 = g_h_filter(data=zs, x0=0., dx=5., dt=1., g=0.4, h=0.01)
         data3 = g_h_filter(data=zs, x0=0., dx=5., dt=1., g=0.8, h=0.01)

         with book_plots.figsize(y=4):
             book_plots.plot_measurements(zs, color='k')
             book_plots.plot_filter(data1, label='g=0.1', marker='s', c='C0')
             book_plots.plot_filter(data2, label='g=0.4', marker='v', c='C1')
             book_plots.plot_filter(data3, label='g=0.8', c='C2')
             plt.legend(loc=4)
```




It is clear that as $g$ is larger we more closely follow the measurement instead of the prediction. When $g = 0.8$ we follow the signal almost exactly, and reject almost none of the noise. One might naively conclude

that $g$ should always be very small to maximize noise rejection. However, that means that we are mostly ignoring the measurements in favor of our prediction. What happens when the signal changes not due to noise, but an actual state change? Let's have a look. I will create data that has $\dot{x} = 1$ for 9 steps before changing to $\dot{x} = 0$.

```
In [34]: zs = [5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
         for i in range(50):
             zs.append(14)

         data1 = g_h_filter(data=zs, x0=4., dx=1., dt=1., g=0.1, h=0.01)
         data2 = g_h_filter(data=zs, x0=4., dx=1., dt=1., g=0.5, h=0.01)
         data3 = g_h_filter(data=zs, x0=4., dx=1., dt=1., g=0.9, h=0.01)

         book_plots.plot_measurements(zs)
         book_plots.plot_filter(data1, label='g=0.1', marker='s', c='C0')
         book_plots.plot_filter(data2, label='g=0.5', marker='v', c='C1')
         book_plots.plot_filter(data3, label='g=0.9', c='C3')
         plt.legend(loc=4)
         plt.ylim([6, 20]);
```

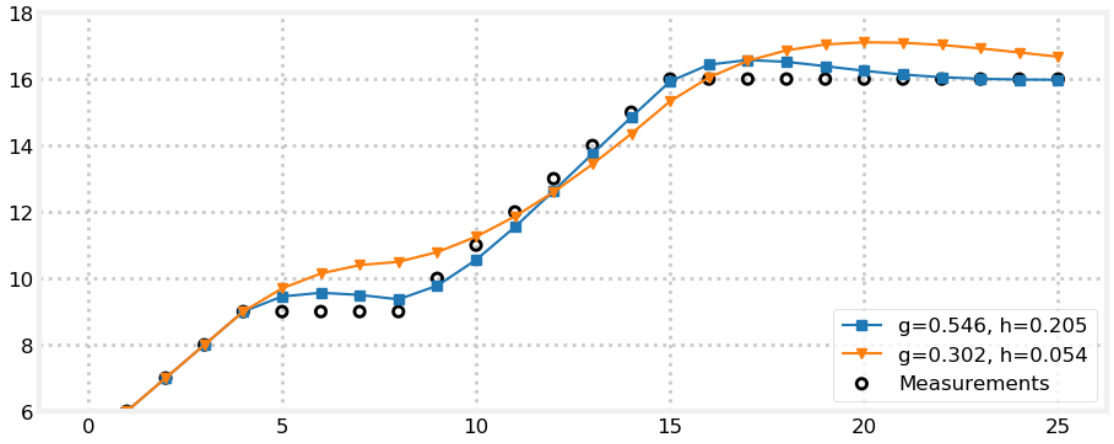



Here we can see the effects of ignoring the signal. We not only filter out noise, but legitimate changes in the signal as well.

Maybe we need a 'Goldilocks' filter, where $g$ is not too large, not too small, but just right? Well, not exactly. As alluded to earlier, different filters choose $g$ and $h$ in different ways depending on the mathematical properties of the problem. For example, the Benedict-Bordner filter was invented to minimize the transient error in this example, where $\dot{x}$ makes a step jump. We will not discuss this filter in this book, but here are two plots chosen with different allowable pairs of $g$ and $h$. This filter design minimizes transient errors for step jumps in $\dot{x}$ at the cost of not being optimal for other types of changes in $\dot{x}$.

```
In [35]: zs = [5,6,7,8,9,9,9,9,9,10,11,12,13,14,
               15,16,16,16,16,16,16,16,16,16,16,16]
         data1 = g_h_filter(data=zs, x0=4., dx=1., dt=1., g=.302, h=.054)
         data2 = g_h_filter(data=zs, x0=4., dx=1., dt=1., g=.546, h=.205)

         book_plots.plot_measurements(zs)
         book_plots.plot_filter(data2, label='g=0.546, h=0.205', marker='s', c='C0')
         book_plots.plot_filter(data1, label='g=0.302, h=0.054', marker='v', c='C1')
```

```
plt.legend(loc=4)
plt.ylim([6, 18]);
```

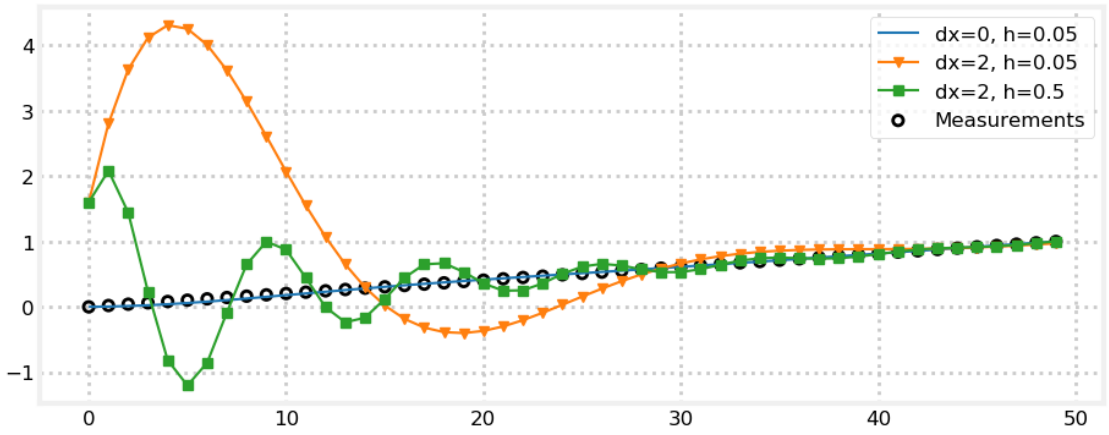



## 1.11 Varying $h$

Now let's leave $g$ unchanged and investigate the effect of modifying $h$. We know that $h$ affects how much we favor the measurement of $\dot{x}$ vs our prediction. But what does this *mean*? If our signal is changing a lot (quickly relative to the time step of our filter), then a large $h$ will cause us to react to those transient changes rapidly. A smaller $h$ will cause us to react more slowly.

We will look at three examples. We have a noiseless measurement that slowly goes from 0 to 1 in 50 steps. Our first filter uses a nearly correct initial value for $\dot{x}$ and a small $h$. You can see from the output that the filter output is very close to the signal. The second filter uses the very incorrect guess of $\dot{x} = 2$. Here we see the filter 'ringing' until it settles down and finds the signal. The third filter uses the same conditions but it now sets $h = 0.5$. If you look at the amplitude of the ringing you can see that it is much smaller than in the second chart, but the frequency is greater. It also settles down a bit quicker than the second filter, though not by much.

```
In [36]: zs = np.linspace(0, 1, 50)

         data1 = g_h_filter(data=zs, x0=0, dx=0., dt=1., g=.2, h=0.05)
         data2 = g_h_filter(data=zs, x0=0, dx=2., dt=1., g=.2, h=0.05)
         data3 = g_h_filter(data=zs, x0=0, dx=2., dt=1., g=.2, h=0.5)

         book_plots.plot_measurements(zs)
         book_plots.plot_filter(data1, label='dx=0, h=0.05', c='C0')
         book_plots.plot_filter(data2, label='dx=2, h=0.05', marker='v', c='C1')
         book_plots.plot_filter(data3, label='dx=2, h=0.5',  marker='s', c='C2')
         plt.legend(loc=1);
```




## 1.12 Interactive Example

For those of you running this in Jupyter Notebook I've written an interactive version of the filter so you can see the effect of changing $\dot{x}$, $g$ and $h$ in real time. As you adjust the sliders for $\dot{x}$, $g$ and $h$ the data will be refiltered and the results plotted for you.

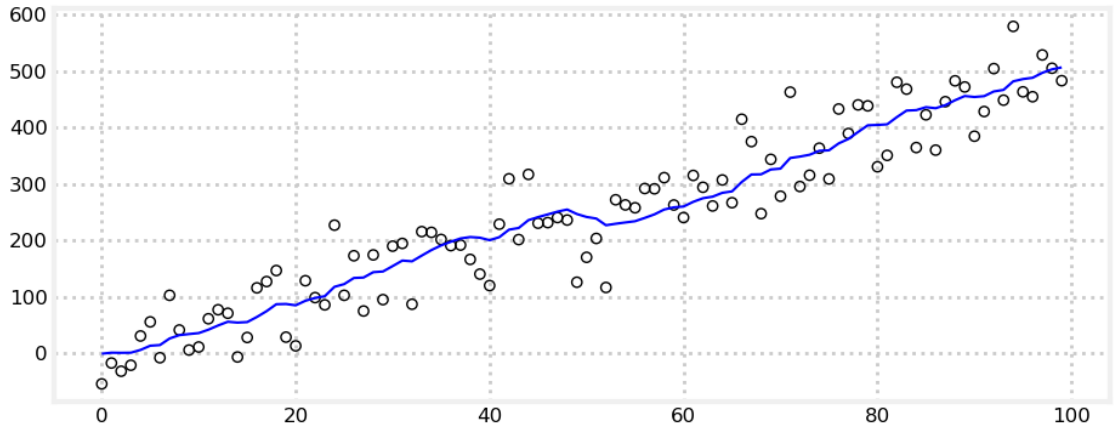
If you really want to test yourself, read the next paragraph and try to predict the results before you move the sliders.

Some things to try include setting $g$ and $h$ to their minimum values. See how perfectly the filter tracks the data! This is only because we are perfectly predicting the weight gain. Adjust $\dot{x}$ to larger or smaller than 5. The filter should diverge from the data and never reacquire it. Start adding back either $g$ or $h$ and see how the filter snaps back to the data. See what the difference in the line is when you add only $g$ vs only $h$. Can you explain the reason for the difference? Then try setting $g$ greater than 1. Can you explain the results? Put $g$ back to a reasonable value (such as 0.1), and then make $h$ very large. Can you explain these results? Finally, set both $g$ and $h$ to their largest values.

If you want to explore with this more, change the value of the array `zs` to the values used in any of the charts above and rerun the cell to see the result.

In [37]:
```
from ipywidgets import interact

# my FloatSlider returns an ipywidgets.FloatSlider with
# continuous_update=False. Filtering code runs too slowly
# to instantly react to slider changes.
from kf_book.book_plots import FloatSlider

zs1 = gen_data(x0=5, dx=5., count=100, noise_factor=50)

fig = None
def interactive_gh(x, dx, g, h):
    global fig
    if fig is not None: plt.close(fig)
    fig = plt.figure()
    data = g_h_filter(data=zs1, x0=x, dx=dx, g=g, h=h)
    plt.scatter(range(len(zs1)), zs1, edgecolor='k',
                facecolors='none', marker='o', lw=1)
    plt.plot(data, color='b')
```

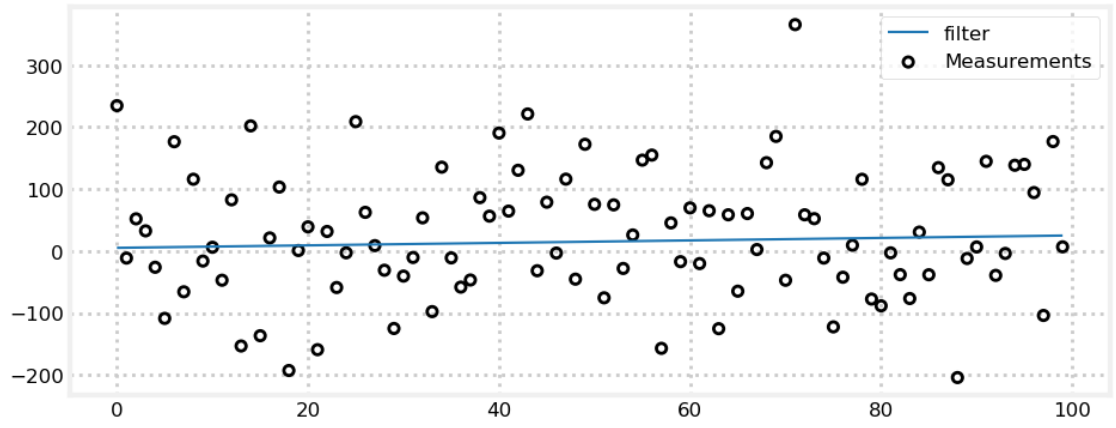
```
interact(interactive_gh,
         x=FloatSlider(value=0, min=-200, max=200),
         dx=FloatSlider(value=5, min=-50, max=50),
         g=FloatSlider(value=.1, min=.01, max=2, step=.02),
         h=FloatSlider(value=.02, min=.0, max=.5, step=.01));
```




## 1.13 Don't Lie to the Filter

You are free to set $g$ and $h$ to any value. Here is a filter that performs perfectly despite extreme noise.

```
In [38]: zs = gen_data(x0=5., dx=.2, count=100, noise_factor=100)
         data = g_h_filter(data=zs, x0=5., dx=.2, dt=1., g=0., h=0.)

         book_plots.plot_measurements(zs)
         book_plots.plot_filter(data, label='filter')
         plt.legend(loc=1);
```

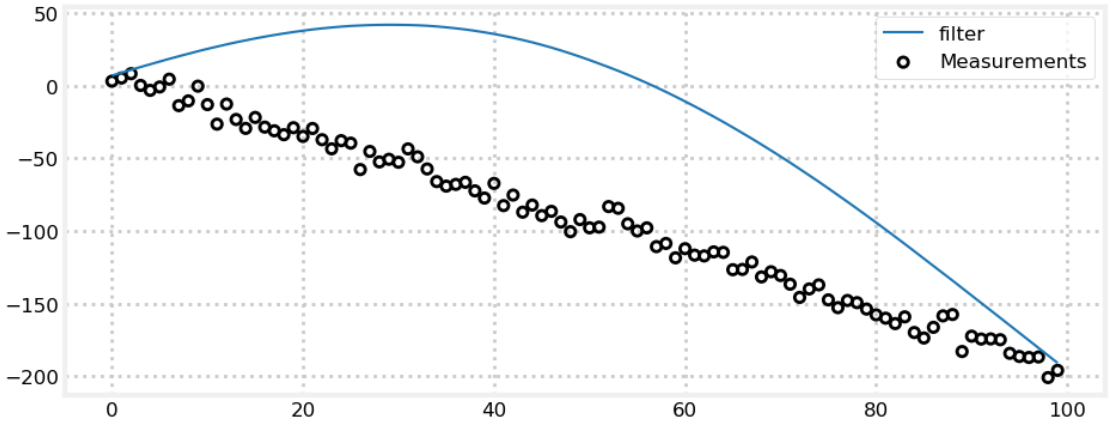



I brilliantly extracted a straight line out of very noisy data! Maybe I shouldn't try to collect my Fields Medal in mathematics just yet. I did this by setting both $g$ and $h$ to 0. What does this do? It makes the

filter ignore the measurements, and so for each update it computes the new position as $x + \Delta x \Delta t$. Of course the result is a straight line if we ignore the measurements.

A filter that ignores measurements is useless. I know you would never set both $g$ and $h$ to zero as that takes a special kind of genius that only I possess, but I promise that if you are not careful you will set them lower than they should be. You can always make great looking results from test data. When you try your filter on different data you will be disappointed in the results because you finely tuned the constants for a specific data set. $g$ and $h$ must reflect the real world behavior of the system you are filtering, not the behavior of one specific data set. In later chapters we will learn a lot about how to do that. For now I can only say be careful, or you will be getting perfect results with your test data, but results like this once you switch to real data:

```
In [39]: zs = gen_data(x0=5, dx=-2, count=100, noise_factor=5)
         data = g_h_filter(data=zs, x0=5., dx=2., dt=1., g=.005, h=0.001)
         book_plots.plot_measurements(zs)
         book_plots.plot_filter(data, label='filter')
         plt.legend(loc=1);
```




## 1.14 Tracking a Train

We are ready for a practical example. Earlier in the chapter we talked about tracking a train. Trains are heavy and slow, thus they cannot change speed quickly. They are on a track, so they cannot change direction except by slowing to a stop and then reversing course. Hence, we can conclude that if we already know the train's approximate position and velocity then we can predict its position in the near future with a great deal of accuracy. A train cannot change its velocity much in a second or two.

So let's write a filter for a train. Its position is expressed as its position on the track in relation to some fixed point which we say is 0 km. I.e., a position of 1 means that the train is 1 km away from the fixed point. Velocity is expressed as meters per second. We perform measurement of position once per second, and the error is ± 500 meters. How should we implement our filter?

First, let's simulate the situation without a filter. We will assume that the train is currently at kilometer 23, and moving at 15 m/s. We can code this as

```
pos = 23*1000
vel = 15
```

Now we can compute the position of the train at some future time, *assuming* no change in velocity, with

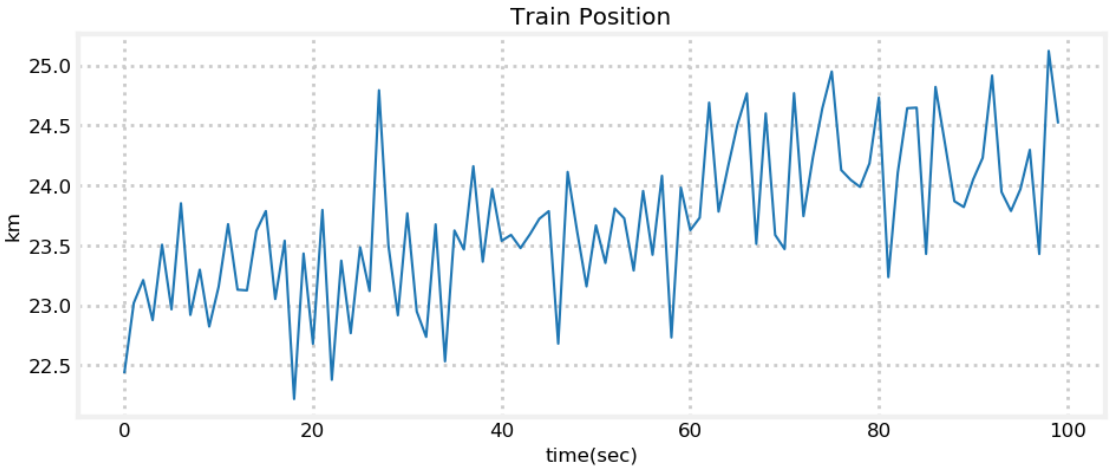
```
def compute_new_position(pos, vel, dt=1):
    return pos + (vel * dt)
```

We can simulate the measurement by adding in some random noise to the position. Here our error is 500m, so the code might look like:

```
def measure_position(pos):
        return pos + random.randn()*500
```

Let's put that in a cell and plot the results of 100 seconds of simulation. I will use NumPy's `asarray` function to convert the data into an NumPy array. This will allow me to divide all of the elements of the array at once by using the '/' operator.

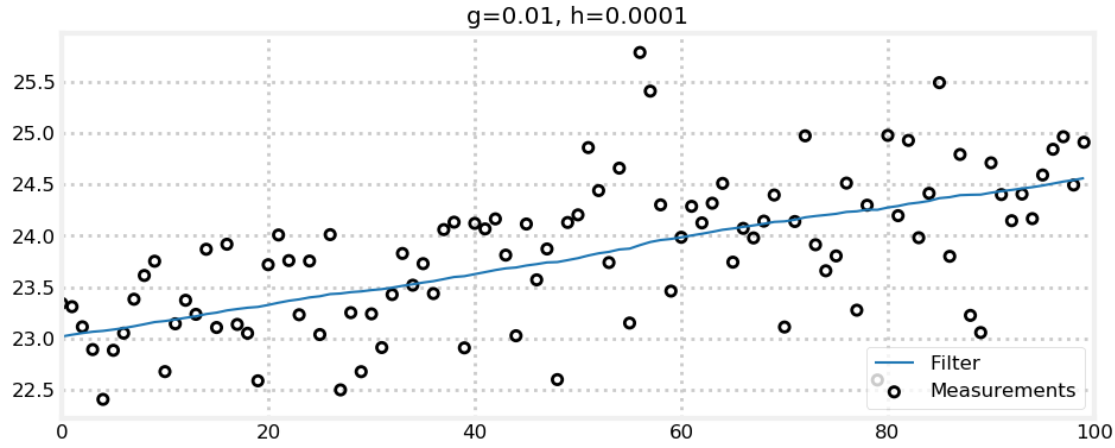
```
In [40]: from numpy.random import randn

         def compute_new_position(pos, vel, dt=1.):
             """ dt is the time delta in seconds."""
             return pos + (vel * dt)

         def measure_position(pos):
             return pos + randn()*500

         def gen_train_data(pos, vel, count):
             zs = []
             for t in range(count):
                 pos = compute_new_position(pos, vel)
                 zs.append(measure_position(pos))
             return np.asarray(zs)

         pos, vel = 23.*1000, 15.
         zs = gen_train_data(pos, vel, 100)

         plt.plot(zs / 1000.)  # convert to km
         book_plots.set_labels('Train Position', 'time(sec)', 'km')
```




We can see from the chart how poor the measurements are. No real train could ever move like that.

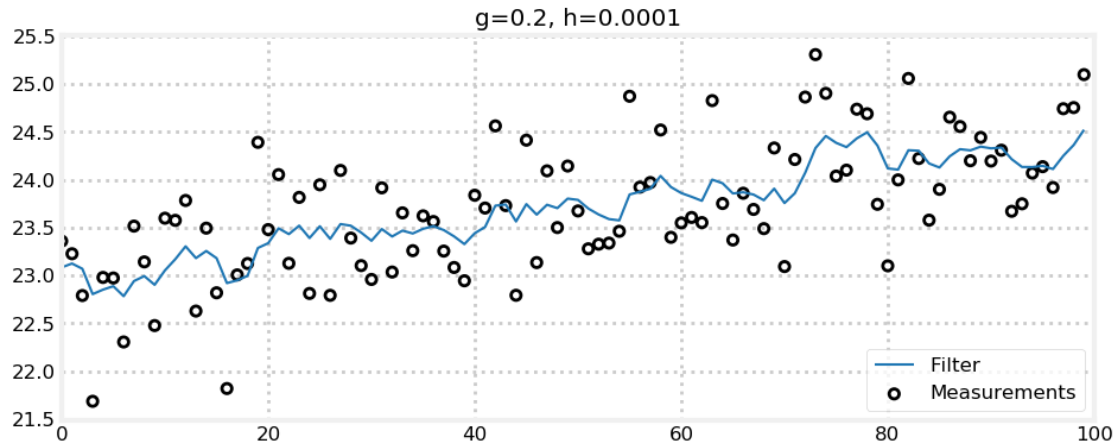
So what should we set $g$ and $h$ to if we want to filter this data? We have not developed the theory for this, but let's try to get a reasonable answer by the seat of our pants. We know that the measurements are very inaccurate, so we don't want to give them much weight at all. To do this we need to choose a very small $g$. We also know that trains can not accelerate or decelerate quickly, so we also want a very small $h$. For example:

```
In [41]: zs = gen_train_data(pos=pos, vel=15., count=100)
         data = g_h_filter(data=zs, x0=pos, dx=15., dt=1., g=.01, h=0.0001)
         plot_g_h_results(zs/1000., data/1000., 'g=0.01, h=0.0001')
```




That is pretty good for an initial guess. Let's make $g$ larger to see the effect.

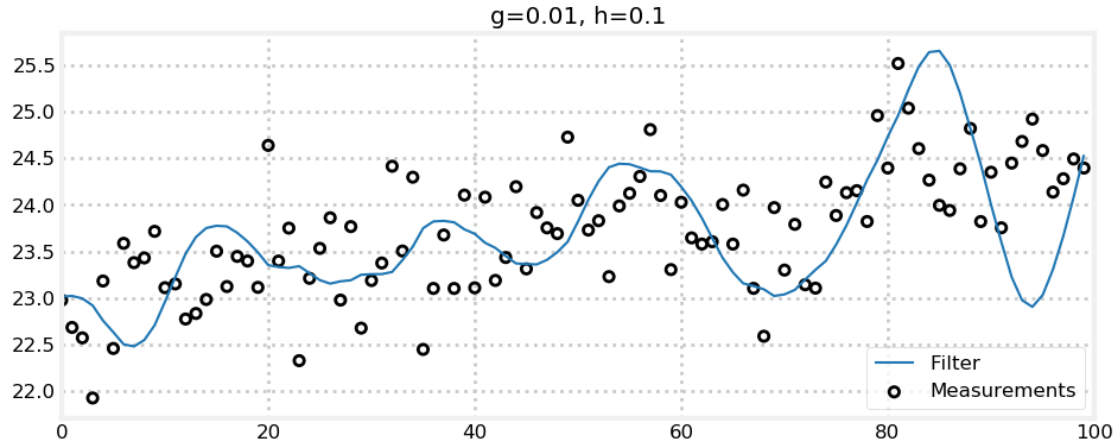
```
In [42]: zs = gen_train_data(pos=pos, vel=15., count=100)
         data = g_h_filter(data=zs, x0=pos, dx=15., dt=1., g=.2, h=0.0001)
         plot_g_h_results(zs/1000., data/1000., 'g=0.2, h=0.0001')
```




We made `g=0.2` and we can see that while the train's position is smoothed, the estimated position (and hence velocity) fluctuates a lot in a very tiny frame, far more than a real train can do. So empirically we know that we want `g<<0.2`.

Now let's see the effect of a poor choice for $h$.

```
In [43]: zs = gen_train_data(pos=pos, vel=15., count=100)
         data = g_h_filter(data=zs, x0=pos, dx=15., dt=1., g=0.01, h=0.1)
         plot_g_h_results(zs/1000., data/1000., 'g=0.01, h=0.1')
```

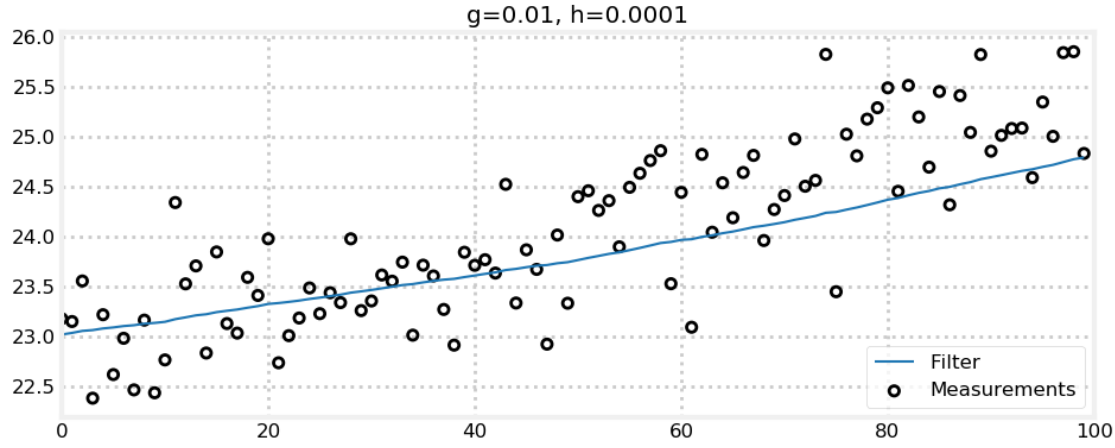



Here the position changes smoothly thanks to the small $g$, but the large $h$ makes the filter very reactive to the measurements. This happens because in the course of a few seconds the rapidly changing measurement implies a very large velocity change, and a large $h$ tells the filter to react to those changes quickly. Trains cannot change velocity quickly, so the filter is not doing a good job of filtering the data - the filter is changing velocity faster than a train can.

Finally, let's add some acceleration to the train. I don't know how fast a train can actually accelerate, but let's say it is accelerating at 0.2 m/sec^2.

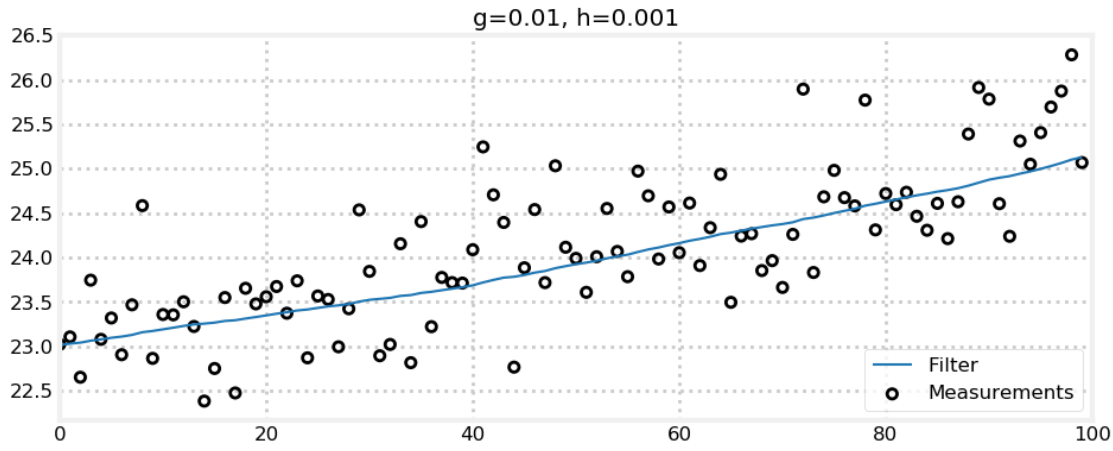
```
In [44]: def gen_train_data_with_acc(pos, vel, count):
             zs = []
             for t in range(count):
                 pos = compute_new_position(pos, vel)
                 vel += 0.2
                 zs.append(measure_position(pos))
             return np.asarray(zs)
```

```
In [45]: zs = gen_train_data_with_acc(pos=pos, vel=15., count=100)
         data = g_h_filter(data=zs, x0=pos, dx=15., dt=1., g=.01, h=0.0001)
         plot_g_h_results(zs/1000., data/1000., 'g=0.01, h=0.0001')
```



Here we see that the filter is not quite tracking the train anymore due to the acceleration. We can fiddle with $h$ to let it track better, at the expense of a less smooth filtered estimate.

```
In [46]: zs = gen_train_data_with_acc(pos=pos, vel=15., count=100)
         data = g_h_filter(data=zs, x0=pos, dx=15., dt=1., g=.01, h=0.001)
         plot_g_h_results(zs/1000., data/1000., 'g=0.01, h=0.001')
```




There are two lessons to be learned here. First, use the $h$ term to respond to changes in velocity that you are not modeling. But, far more importantly, there is a trade off here between responding quickly and accurately to changes in behavior and producing ideal output for when the system is in a steady state that you have. If the train never changes velocity we would make $h$ extremely small to avoid having the filtered estimate unduly affected by the noise in the measurement. But in an interesting problem there are almost always changes in state, and we want to react to them quickly. The more quickly we react to them, the more we are affected by the noise in the sensors.

I could go on, but my aim is not to develop g-h filter theory here so much as to build insight into how combining measurements and predictions leads to a filtered solution. There is extensive literature on choosing $g$ and $h$ for problems such as this, and there are optimal ways of choosing them to achieve various goals. As I explained earlier it is easy to 'lie' to the filter when experimenting with test data like this. In the subsequent chapters we will learn how the Kalman filter solves this problem in the same basic manner, but with far more sophisticated mathematics.

## 1.15 g-h Filters with FilterPy

FilterPy is an open source filtering library that I wrote. It has all of the filters in this book, along with others. It is rather easy to program your own g-h filter, but as we progress we will rely on FilterPy more. As a quick introduction, let's look at the g-h filter in FilterPy.

If you do not have FilterPy installed just issue the following command from the command line.

```
pip install filterpy
```

Read Appendix A for more information on installing or downloading FilterPy from GitHub.

To use the g-h filter import it and create an object from the class `GHFilter`.

```
In [47]: from filterpy.gh import GHFilter
         f = GHFilter(x=0., dx=0., dt=1., g=.8, h=.2)
```

When you construct the object you specify the initial value and rate of change for the signal (`x` and 'dx'), the time step between updates(`dt`) and the two filter parameter (`g` and `h`). `dx` must have the same units of `x/dt` - if `x` is in meters and `dt` is in seconds then `dx` must be in meters per second.

To run the filter call update, passing the measurement in the parameter `z`, which you'll recall is a standard name for measurements in the literature.

```
In [48]: f.update(z=1.2)
```

```
Out[48]: (0.96, 0.24)
```

`update()` returns the new value of `x` and `dx` in a tuple, but you can also access them from the object.

```
In [49]: print(f.x, f.dx)
```

```
0.96 0.24
```

You can dynamically alter `g` and `h`.

```
In [50]: print(f.update(z=2.1, g=.85, h=.15))
```

```
(1.965, 0.375)
```

You can filter a sequence of measurements in a batch.

```
In [51]: print(f.batch_filter([3., 4., 5.]))
```

```
[[1.965 0.375]
 [2.868 0.507]
 [3.875 0.632]
 [4.901 0.731]]
```

You can filter multiple independent variables. If you are tracking an aircraft you'll need to track it in 3D space. Use NumPy arrays for `x`, `dx`, and the measurements.

```
In [52]: x_0  = np.array([1., 10., 100.])
         dx_0 = np.array([10., 12., .2])

         f_air = GHFilter(x=x_0, dx=dx_0, dt=1., g=.8, h=.2)
         f_air.update(z=np.array((2., 11., 102.)))
         print(' x =', f_air.x)
         print('dx =', f_air.dx)
```

```
 x = [  3.8   13.2  101.64]
dx = [8.2  9.8  0.56]
```

The class `GHFilterOrder` allows you to create a filter of order 0, 1, or 2. A g-h filter is order 1. The g-h-k filter, which we haven't talked about, also tracks accelerations. Both classes have functionality required by real applications, such as computing the Variance Reduction Factor (VRF), which we haven't discussed in this chapter. I could fill a book just on the theory and applications of g-h filters, but we have other goals in this book. If you are interested, explore the FilterPy code and do some further reading.

The documentation for FilterPy is at https://filterpy.readthedocs.org/.

## 1.16 Summary

I encourage you to experiment with this filter to develop your understanding of how it reacts. It shouldn't take too many attempts to come to the realization that ad-hoc choices for $g$ and $h$ do not perform very well. A particular choice might perform well in one situation, but very poorly in another. Even when you understand the effect of $g$ and $h$ it can be difficult to choose proper values. In fact, it is extremely unlikely that you will choose values for $g$ and $h$ that is optimal for any given problem. Filters are *designed*, not selected *ad hoc*.

In some ways I do not want to end the chapter here, as there is a significant amount that we can say about selecting $g$ and $h$. But the g-h filter in this form is not the purpose of this book. Designing the Kalman filter requires you to specify a number of parameters - indirectly they do relate to choosing $g$ and $h$, but you will never refer to them directly when designing Kalman filters. Furthermore, $g$ and $h$ will vary at every time step in a very non-obvious manner.

There is another feature of these filters we have barely touched upon - Bayesian statistics. You will note that the term 'Bayesian' is in the title of this book; this is not a coincidence! For the time being we will leave $g$ and $h$ behind, largely unexplored, and develop a very powerful form of probabilistic reasoning about filtering. Yet suddenly this same g-h filter algorithm will appear, this time with a formal mathematical edifice that allows us to create filters from multiple sensors, to accurately estimate the amount of error in our solution, and to control robots.

# Chapter 2

# Discrete Bayes Filter

```
In [1]: from __future__ import division, print_function
        %matplotlib inline
```

```
In [2]: #format the book
        import book_format
        book_format.set_style()
```

```
Out[2]:
```

The Kalman filter belongs to a family of filters called *Bayesian filters*. Most textbook treatments of the Kalman filter present the Bayesian formula, perhaps shows how it factors into the Kalman filter equations, but mostly keeps the discussion at a very abstract level.

That approach requires a fairly sophisticated understanding of several fields of mathematics, and it still leaves much of the work of understanding and forming an intuitive grasp of the situation in the hands of the reader.

I will use a different way to develop the topic, to which I owe the work of Dieter Fox and Sebastian Thrun a great debt. It depends on building an intuition on how Bayesian statistics work by tracking an object through a hallway - they use a robot, I use a dog. I like dogs, and they are less predictable than robots which imposes interesting difficulties for filtering. The first published example of this that I can find seems to be Fox 1999 [1], with a fuller example in Fox 2003 [2]. Sebastian Thrun also uses this formulation in his excellent Udacity course Artificial Intelligence for Robotics [3]. In fact, if you like watching videos, I highly recommend pausing reading this book in favor of first few lessons of that course, and then come back to this book for a deeper dive into the topic.

Let's now use a simple thought experiment, much like we did with the g-h filter, to see how we might reason about the use of probabilities for filtering and tracking.

## 2.1 Tracking a Dog

Let's begin with a simple problem. We have a dog friendly workspace, and so people bring their dogs to work. Occasionally the dogs wander out of offices and down the halls. We want to be able to track them. So during a hackathon somebody invented a sonar sensor to attach to the dog's collar. It emits a signal, listens for the echo, and based on how quickly an echo comes back we can tell whether the dog is in front of an open doorway or not. It also senses when the dog walks, and reports in which direction the dog has moved. It connects to the network via wifi and sends an update once a second.

I want to track my dog Simon, so I attach the device to his collar and then fire up Python, ready to write code to track him through the building. At first blush this may appear impossible. If I start listening to the sensor of Simon's collar I might read **door**, **hall**, **hall**, and so on. How can I use that information to determine where Simon is?

To keep the problem small enough to plot easily we will assume that there are only 10 positions in the hallway, which we will number 0 to 9, where 1 is to the right of 0. For reasons that will be clear later, we

will also assume that the hallway is circular or rectangular. If you move right from position 9, you will be at position 0.

When I begin listening to the sensor I have no reason to believe that Simon is at any particular position in the hallway. From my perspective he is equally likely to be in any position. There are 10 positions, so the probability that he is in any given position is 1/10.

Let's represent our belief of his position in a NumPy array. I could use a Python list, but NumPy arrays offer functionality that we will be using soon.

```
In [3]: import numpy as np
        belief = np.array([1/10]*10)
        print(belief)
```

```
[0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1 0.1]
```

In Bayesian statistics this is called a *prior*. It is the probability prior to incorporating measurements or other information. More completely, this is called the *prior probability distribution*. A *probability distribution* is a collection of all possible probabilities for an event. Probability distributions always sum to 1 because something had to happen; the distribution lists all possible events and the probability of each.

I'm sure you've used probabilities before - as in "the probability of rain today is 30%". The last paragraph sounds like more of that. But Bayesian statistics was a revolution in probability because it treats probability as a belief about a single event. Let's take an example. I know that if I flip a fair coin infinitely many times I will get 50% heads and 50% tails. This is called *frequentist statistics* to distinguish it from Bayesian statistics. Computations are based on the frequency in which events occur.

I flip the coin one more time and let it land. Which way do I believe it landed? Frequentist probability has nothing to say about that; it will merely state that 50% of coin flips land as heads. In some ways it is meaningless to assign a probability to the current state of the coin. It is either heads or tails, we just don't know which. Bayes treats this as a belief about a single event - the strength of my belief or knowledge that this specific coin flip is heads is 50%. Some object to the term "belief"; belief can imply holding something to be true without evidence. In this book it always is a measure of the strength of our knowledge. We'll learn more about this as we go.

Bayesian statistics takes past information (the prior) into account. We observe that it rains 4 times every 100 days. From this I could state that the chance of rain tomorrow is 1/25. This is not how weather prediction is done. If I know it is raining today and the storm front is stalled, it is likely to rain tomorrow. Weather prediction is Bayesian.

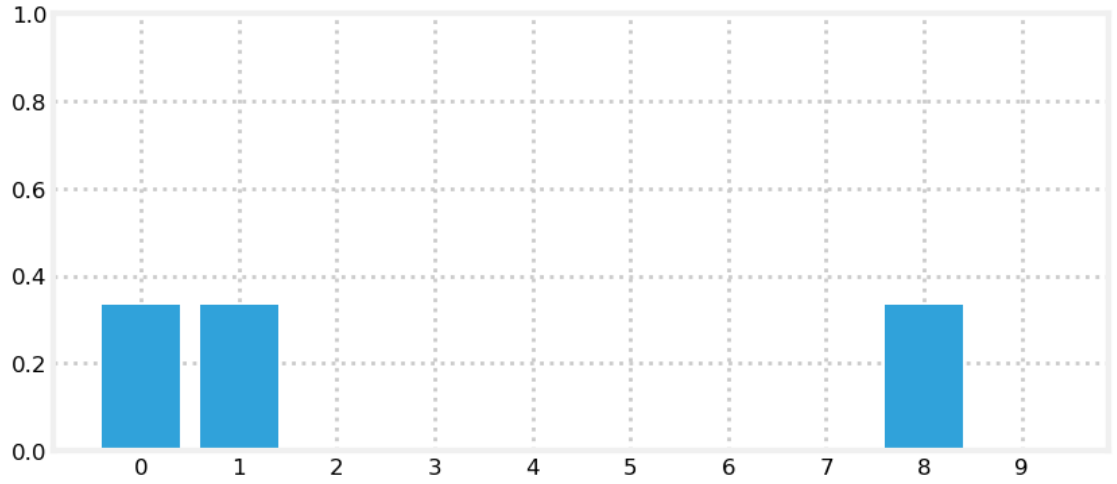
In practice statisticians use a mix of frequentist and Bayesian techniques. Sometimes finding the prior is difficult or impossible, and frequentist techniques rule. In this book we can find the prior. When I talk about the probability of something I am referring to the probability that some specific thing is true given past events. When I do that I'm taking the Bayesian approach.

Now let's create a map of the hallway. We'll place the first two doors close together, and then another door further away. We will use 1 for doors, and 0 for walls:

```
In [4]: hallway = np.array([1, 1, 0, 0, 0, 0, 0, 0, 1, 0])
```

I start listening to Simon's transmissions on the network, and the first data I get from the sensor is **door**. For the moment assume the sensor always returns the correct answer. From this I conclude that he is in front of a door, but which one? I have no reason to believe he is in front of the first, second, or third door. What I can do is assign a probability to each door. All doors are equally likely, and there are three of them, so I assign a probability of 1/3 to each door.

```
In [5]: import kf_book.book_plots as book_plots
        from kf_book.book_plots import figsize, set_figsize
        import matplotlib.pyplot as plt

        belief = np.array([1/3, 1/3, 0, 0, 0, 0, 0, 0, 1/3, 0])
        book_plots.bar_plot(belief)
```




This distribution is called a *categorical distribution*, which is a discrete distribution describing the probability of observing $n$ outcomes. It is a *multimodal distribution* because we have multiple beliefs about the position of our dog. Of course we are not saying that we think he is simultaneously in three different locations, merely that we have narrowed down our knowledge to one of these three locations. My (Bayesian) belief is that there is a 33.3% chance of being at door 0, 33.3% at door 1, and a 33.3% chance of being at door 8.

This is an improvement in two ways. I've rejected a number of hallway positions as impossible, and the strength of my belief in the remaining positions has increased from 10% to 33%. This will always happen. As our knowledge improves the probabilities will get closer to 100%.

A few words about the *mode* of a distribution. Given a list of numbers, such as {1, 2, 2, 2, 3, 3, 4}, the *mode* is the number that occurs most often. For this set the mode is 2. A distribution can contain more than one mode. The list {1, 2, 2, 2, 3, 3, 4, 4, 4} contains the modes 2 and 4, because both occur three times. We say the former list is *unimodal*, and the latter is *multimodal*.

Another term used for this distribution is a *histogram*. Histograms graphically depict the distribution of a set of numbers. The bar chart above is a histogram.

I hand coded the `belief` array in the code above. How would we implement this in code? We represent doors with 1, and walls as 0, so we will multiply the hallway variable by the percentage, like so;

```
In [6]: belief = hallway * (1/3)
        print(belief)
```

```
[0.333 0.333 0.    0.    0.    0.    0.    0.    0.333 0.   ]
```

## 2.2 Extracting Information from Sensor Readings

Let's put Python aside and think about the problem a bit. Suppose we were to read the following from Simon's sensor:

- door
- move right
- door

Can we deduce Simon's location? Of course! Given the hallway's layout there is only one place from which you can get this sequence, and that is at the left end. Therefore we can confidently state that Simon is in front of the second doorway. If this is not clear, suppose Simon had started at the second or third door.

After moving to the right, his sensor would have returned 'wall'. That doesn't match the sensor readings, so we know he didn't start there. We can continue with that logic for all the remaining starting positions. The only possibility is that he is now in front of the second door. Our belief is:

```
In [7]: belief = np.array([0., 1., 0., 0., 0., 0., 0., 0., 0., 0.])
```

I designed the hallway layout and sensor readings to give us an exact answer quickly. Real problems are not so clear cut. But this should trigger your intuition - the first sensor reading only gave us low probabilities (0.333) for Simon's location, but after a position update and another sensor reading we know more about where he is. You might suspect, correctly, that if you had a very long hallway with a large number of doors that after several sensor readings and positions updates we would either be able to know where Simon was, or have the possibilities narrowed down to a small number of possibilities. This is possible when a set of sensor readings only matches one to a few starting locations.

We could implement this solution now, but instead let's consider a real world complication to the problem.

## 2.3 Noisy Sensors

Perfect sensors are rare. Perhaps the sensor would not detect a door if Simon sat in front of it while scratching himself, or misread if he is not facing down the hallway. Thus when I get **door** I cannot use 1/3 as the probability. I have to assign less than 1/3 to each door, and assign a small probability to each blank wall position. Something like

```
[.31, .31, .01, .01, .01, .01, .01, .01, .31, .01]
```

At first this may seem insurmountable. If the sensor is noisy it casts doubt on every piece of data. How can we conclude anything if we are always unsure?

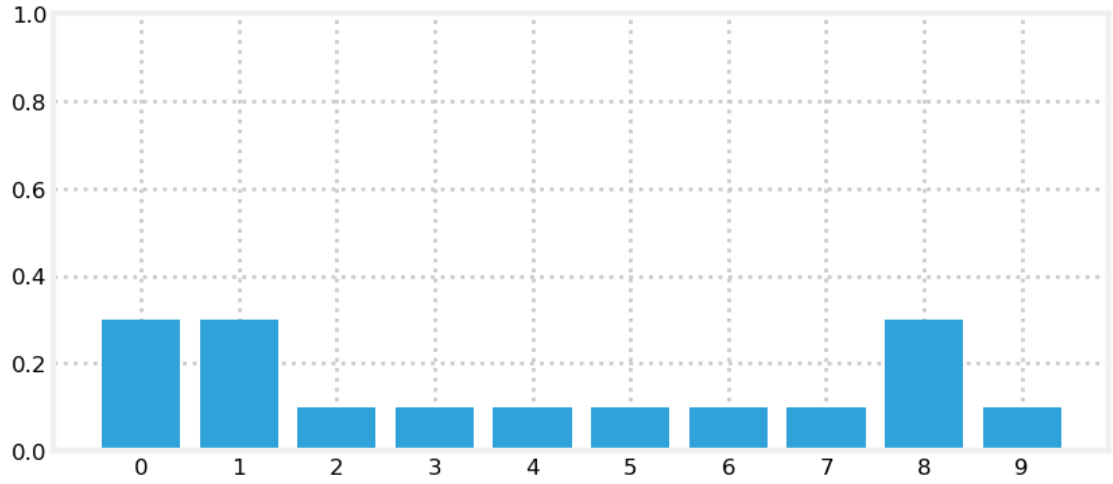
The answer, as for the problem above, is with probabilities. We are already comfortable assigning a probabilistic belief to the location of the dog; now we have to incorporate the additional uncertainty caused by the sensor noise.

Say we get a reading of **door**, and suppose that testing shows that the sensor is 3 times more likely to be right than wrong. We should scale the probability distribution by 3 where there is a door. If we do that the result will no longer be a probability distribution, but we will learn how to fix that in a moment.

Let's look at that in Python code. Here I use the variable `z` to denote the measurement. `z` or `y` are customary choices in the literature for the measurement. As a programmer I prefer meaningful variable names, but I want you to be able to read the literature and/or other filtering code, so I will start introducing these abbreviated names now.

```
In [8]: def update_belief(hall, belief, z, correct_scale):
            for i, val in enumerate(hall):
                if val == z:
                    belief[i] *= correct_scale

        belief = np.array([0.1] * 10)
        reading = 1 # 1 is 'door'
        update_belief(hallway, belief, z=reading, correct_scale=3.)
        print('belief:', belief)
        print('sum =', sum(belief))
        plt.figure()
        book_plots.bar_plot(belief)
```

```
belief: [0.3 0.3 0.1 0.1 0.1 0.1 0.1 0.1 0.3 0.1]
sum = 1.6000000000000003
```




This is not a probability distribution because it does not sum to 1.0. But the code is doing mostly the right thing - the doors are assigned a number (0.3) that is 3 times higher than the walls (0.1). All we need to do is normalize the result so that the probabilities correctly sum to 1.0. Normalization is done by dividing each element by the sum of all elements in the list. That is easy with NumPy:

```
In [9]: belief / sum(belief)

Out[9]: array([0.188, 0.188, 0.062, 0.062, 0.062, 0.062, 0.062, 0.062, 0.188,
               0.062])
```

FilterPy implements this with the `normalize` function:

```
from filterpy.discrete_bayes import normalize
normalize(belief)
```

It is a bit odd to say "3 times as likely to be right as wrong". We are working in probabilities, so let's specify the probability of the sensor being correct, and compute the scale factor from that. The equation for that is

$$scale = \frac{prob_{correct}}{prob_{incorrect}} = \frac{prob_{correct}}{1 - prob_{correct}}$$

Also, the `for` loop is cumbersome. As a general rule you will want to avoid using `for` loops in NumPy code. NumPy is implemented in C and Fortran, so if you avoid for loops the result often runs 100x faster than the equivalent loop.

How do we get rid of this `for` loop? NumPy lets you index arrays with boolean arrays. You create a boolean array with logical operators. We can find all the doors in the hallway with:

```
In [10]: hallway == 1

Out[10]: array([ True,  True, False, False, False, False, False, False,  True,
                False])
```

When you use the boolean array as an index to another array it returns only the elements where the index is `True`. Thus we can replace the `for` loop with

```
belief[hall==z] *= scale
```

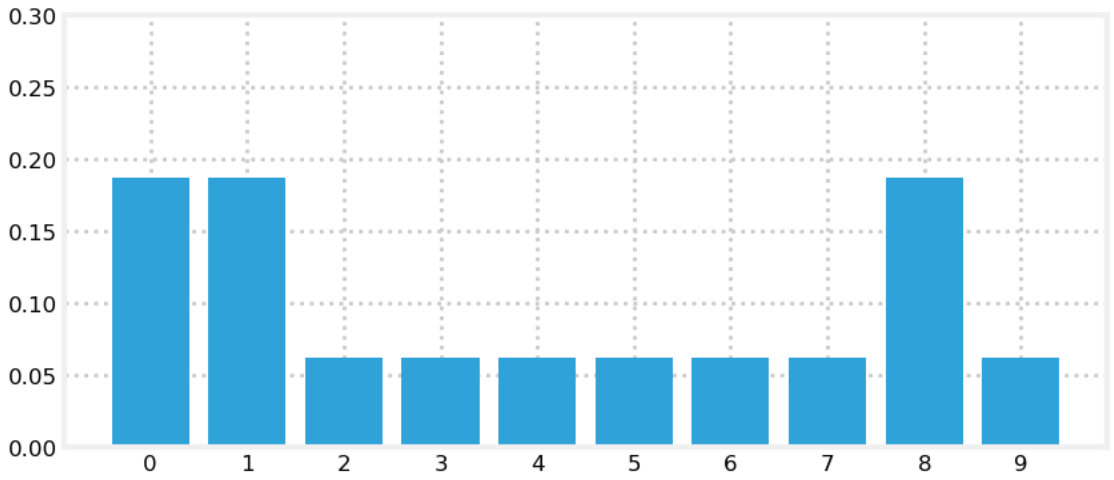
and only the elements which equal `z` will be multiplied by `scale`.

Teaching you NumPy is beyond the scope of this book. I will use idiomatic NumPy constructs and explain them the first time I present them. If you are new to NumPy there are many blog posts and videos on how to use NumPy efficiently and idiomatically.

Here is our improved version:

```
In [11]: from filterpy.discrete_bayes import normalize

         def scaled_update(hall, belief, z, z_prob):
             scale = z_prob / (1. - z_prob)
             belief[hall==z] *= scale
             normalize(belief)

         belief = np.array([0.1] * 10)
         scaled_update(hallway, belief, z=1, z_prob=.75)

         print('sum =', sum(belief))
         print('probability of door =', belief[0])
         print('probability of wall =', belief[2])
         book_plots.bar_plot(belief, ylim=(0, .3))
```

```
sum = 1.0
probability of door = 0.1875
probability of wall = 0.06249999999999999
```




We can see from the output that the sum is now 1.0, and that the probability of a door vs wall is still three times larger. The result also fits our intuition that the probability of a door must be less than 0.333, and that the probability of a wall must be greater than 0.0. Finally, it should fit our intuition that we have not yet been given any information that would allow us to distinguish between any given door or wall position, so all door positions should have the same value, and the same should be true for wall positions.

This result is called the *posterior*, which is short for *posterior probability distribution*. All this means is a probability distribution *after* incorporating the measurement information (posterior means 'after' in this context). To review, the *prior* is the probability distribution before including the measurement's information.

Another term is the *likelihood*. When we computed `belief[hall==z] *= scale` we were computing how *likely* each position was given the measurement. The likelihood is not a probability distribution because it does not sum to one.

The combination of these gives the equation

$$\text{posterior} = \frac{\text{likelihood} \times \text{prior}}{\text{normalization}}$$

When we talk about the filter's output we typically call the state after performing the prediction the *prior* or *prediction*, and we call the state after the update either the *posterior* or the *estimated state*.

It is very important to learn and internalize these terms as most of the literature uses them extensively.

Does `scaled_update()` perform this computation? It does. Let me recast it into this form:

```
In [12]: def scaled_update(hall, belief, z, z_prob):
             scale = z_prob / (1. - z_prob)
             likelihood = np.ones(len(hall))
             likelihood[hall==z] *= scale
             return normalize(likelihood * belief)
```

This function is not fully general. It contains knowledge about the hallway, and how we match measurements to it. We always strive to write general functions. Here we will remove the computation of the likelihood from the function, and require the caller to compute the likelihood themselves.

Here is a full implementation of the algorithm:

```
def update(likelihood, prior):
    return normalize(likelihood * prior)
```

Computation of the likelihood varies per problem. For example, the sensor might not return just 1 or 0, but a `float` between 0 and 1 indicating the probability of being in front of a door. It might use computer vision and report a blob shape that you then probabilistically match to a door. It might use sonar and return a distance reading. In each case the computation of the likelihood will be different. We will see many examples of this throughout the book, and learn how to perform these calculations.

FilterPy implements `update`. Here is the previous example in a fully general form:

```
In [13]: from filterpy.discrete_bayes import update

         def lh_hallway(hall, z, z_prob):
             """ compute likelihood that a measurement matches
             positions in the hallway."""

             try:
                 scale = z_prob / (1. - z_prob)
             except ZeroDivisionError:
                 scale = 1e8

             likelihood = np.ones(len(hall))
             likelihood[hall==z] *= scale
             return likelihood

         belief = np.array([0.1] * 10)
         likelihood = lh_hallway(hallway, z=1, z_prob=.75)
         update(likelihood, belief)
```

```
Out[13]: array([0.188, 0.188, 0.062, 0.062, 0.062, 0.062, 0.062, 0.062, 0.188,
                0.062])
```

## 2.4 Incorporating Movement

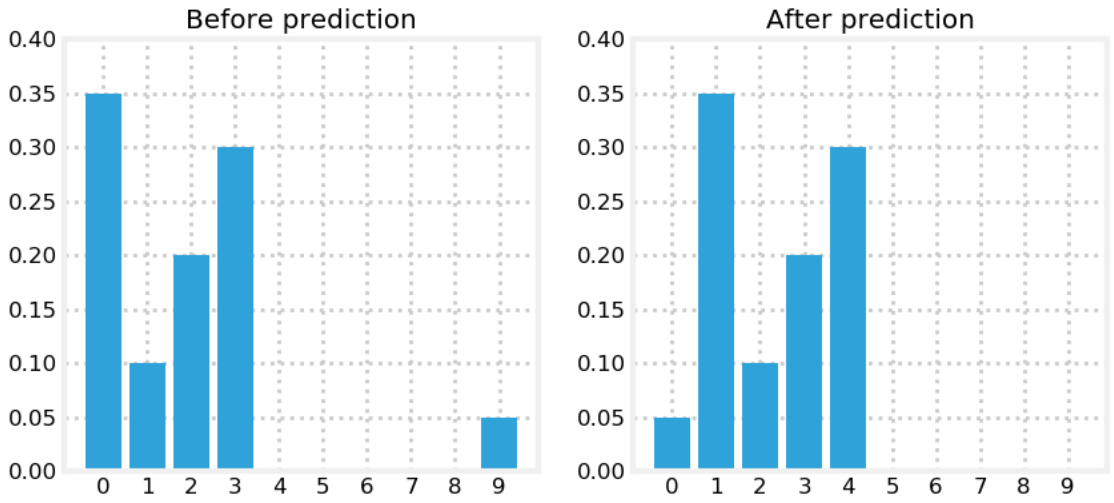
Recall how quickly we were able to find an exact solution when we incorporated a series of measurements and movement updates. However, that occurred in a fictional world of perfect sensors. Might we be able to find an exact solution with noisy sensors?

Unfortunately, the answer is no. Even if the sensor readings perfectly match an extremely complicated hallway map, we cannot be 100% certain that the dog is in a specific position - there is, after all, a tiny possibility that every sensor reading was wrong! Naturally, in a more typical situation most sensor readings will be correct, and we might be close to 100% sure of our answer, but never 100% sure. This may seem complicated, but let's go ahead and program the math.

First let's deal with the simple case - assume the movement sensor is perfect, and it reports that the dog has moved one space to the right. How would we alter our `belief` array?

I hope that after a moment's thought it is clear that we should shift all the values one space to the right. If we previously thought there was a 50% chance of Simon being at position 3, then after he moved one position to the right we should believe that there is a 50% chance he is at position 4. The hallway is circular, so we will use modulo arithmetic to perform the shift.

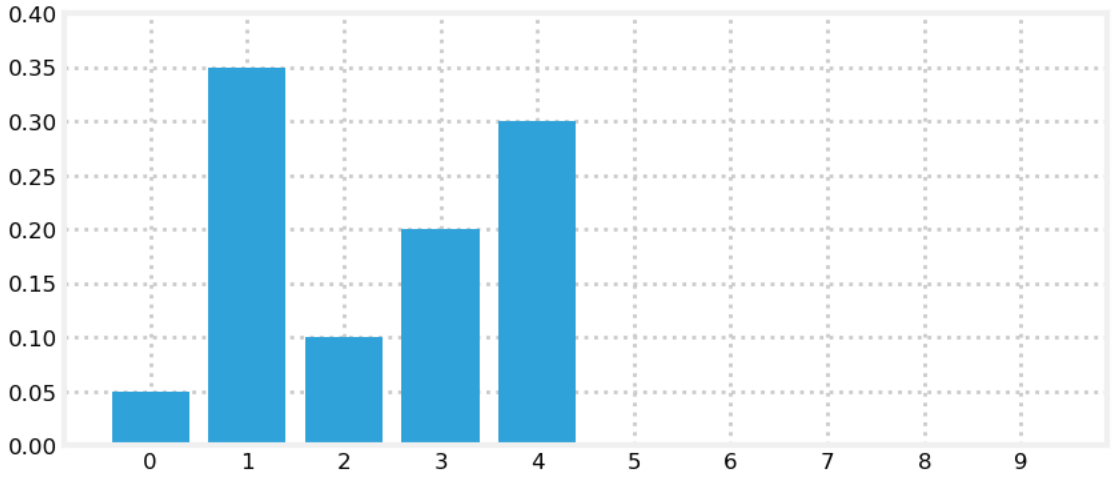
```
In [14]: def perfect_predict(belief, move):
             """ move the position by `move` spaces, where positive is
             to the right, and negative is to the left
             """
             n = len(belief)
             result = np.zeros(n)
             for i in range(n):
                 result[i] = belief[(i-move) % n]
             return result

         belief = np.array([.35, .1, .2, .3, 0, 0, 0, 0, 0, .05])
         plt.subplot(121)
         book_plots.bar_plot(belief, title='Before prediction', ylim=(0, .4))

         belief = perfect_predict(belief, 1)
         plt.subplot(122)
         book_plots.bar_plot(belief, title='After prediction', ylim=(0, .4))
```




We can see that we correctly shifted all values one position to the right, wrapping from the end of the array back to the beginning.

The next cell animates this so you can see it in action. Use the slider to move forwards and backwards in time. This simulates Simon walking around and around the hallway. It does not yet incorporate new measurements so the probability distribution does not change shape, only position.

```
In [15]: from ipywidgets import interact, IntSlider

         belief = np.array([.35, .1, .2, .3, 0, 0, 0, 0, 0, .05])
         perfect_beliefs = []

         for _ in range(20):
             # Simon takes one step to the right
             belief = perfect_predict(belief, 1)
             perfect_beliefs.append(belief)

         def simulate(time_step):
             book_plots.bar_plot(perfect_beliefs[time_step], ylim=(0, .4))

         interact(simulate, time_step=IntSlider(value=0, max=len(perfect_beliefs)-1));
```




## 2.5 Terminology

Let's pause a moment to review terminology. I introduced this terminology in the last chapter, but let's take a second to help solidify your knowledge.

The *system* is what we are trying to model or filter. Here the system is our dog. The *state* is its current configuration or value. In this chapter the state is our dog's position. We rarely know the actual state, so we say our filters produce the *estimated state* of the system. In practice this often gets called the state, so be careful to understand the context.

One cycle of prediction and updating with a measurement is called the state or system *evolution*, which is short for *time evolution* [7]. Another term is *system propagation*. It refers to how the state of the system changes over time. For filters, time is usually a discrete step, such as 1 second. For our dog tracker the system state is the position of the dog, and the state evolution is the position after a discrete amount of time has passed.

We model the system behavior with the *process model*. Here, our process model is that the dog moves one or more positions at each time step. This is not a particularly accurate model of how dogs behave. The error in the model is called the *system error* or *process error*.

The prediction is our new *prior*. Time has moved forward and we made a prediction without benefit of knowing the measurements.

Let's work an example. The current position of the dog is 17 m. Our epoch is 2 seconds long, and the dog is traveling at 15 m/s. Where do we predict he will be in two seconds?

Clearly,

$$\begin{aligned}\bar{x} &= 17 + (15 * 2) \\ &= 47\end{aligned}$$

I use bars over variables to indicate that they are priors (predictions). We can write the equation for the process model like this:

$$\bar{x}_{k+1} = f_x(\bullet) + x_k$$

$x_k$ is the current position or state. If the dog is at 17 m then $x_k = 17$.

$f_x(\bullet)$ is the state propagation function for x. It describes how much the $x_k$ changes over one time step. For our example it performs the computation $15 \cdot 2$ so we would define it as

$$f_x(v_x, t) = v_k t$$

.

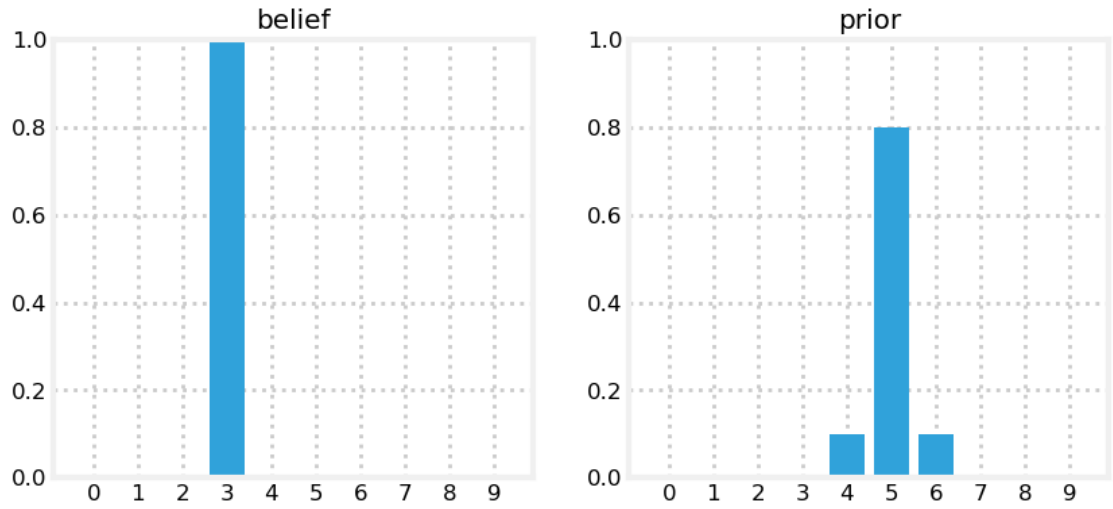
## 2.6 Adding Uncertainty to the Prediction

`perfect_predict()` assumes perfect measurements, but all sensors have noise. What if the sensor reported that our dog moved one space, but he actually moved two spaces, or zero? This may sound like an insurmountable problem, but let's model it and see what happens.

Assume that the sensor's movement measurement is 80% likely to be correct, 10% likely to overshoot one position to the right, and 10% likely to undershoot to the left. That is, if the movement measurement is 4 (meaning 4 spaces to the right), the dog is 80% likely to have moved 4 spaces to the right, 10% to have moved 3 spaces, and 10% to have moved 5 spaces.

Each result in the array now needs to incorporate probabilities for 3 different situations. For example, consider the reported movement of 2. If we are 100% certain the dog started from position 3, then there is an 80% chance he is at 5, and a 10% chance for either 4 or 6. Let's try coding that:

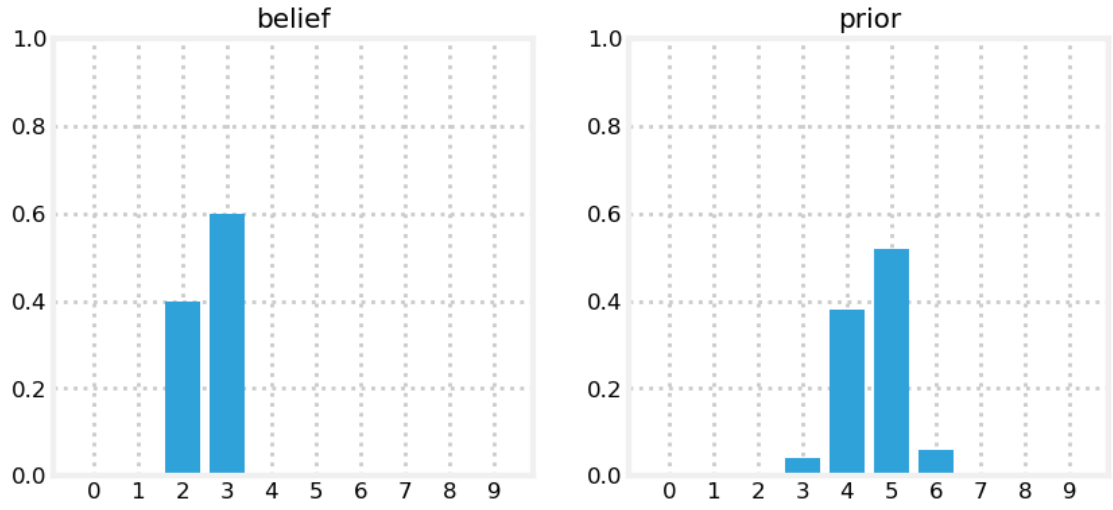
```
In [16]: def predict_move(belief, move, p_under, p_correct, p_over):
             n = len(belief)
             prior = np.zeros(n)
             for i in range(n):
                 prior[i] = (
                     belief[(i-move) % n]   * p_correct +
                     belief[(i-move-1) % n] * p_over +
                     belief[(i-move+1) % n] * p_under)
             return prior

         belief = [0., 0., 0., 1., 0., 0., 0., 0., 0., 0.]
         prior = predict_move(belief, 2, .1, .8, .1)
         book_plots.plot_belief_vs_prior(belief, prior)
```




It appears to work correctly. Now what happens when our belief is not 100% certain?

```
In [17]: belief = [0, 0, .4, .6, 0, 0, 0, 0, 0, 0]
         prior = predict_move(belief, 2, .1, .8, .1)
         book_plots.plot_belief_vs_prior(belief, prior)
         prior
```

```
Out[17]: array([0.  , 0.  , 0.  , 0.04, 0.38, 0.52, 0.06, 0.  , 0.  , 0.  ])
```

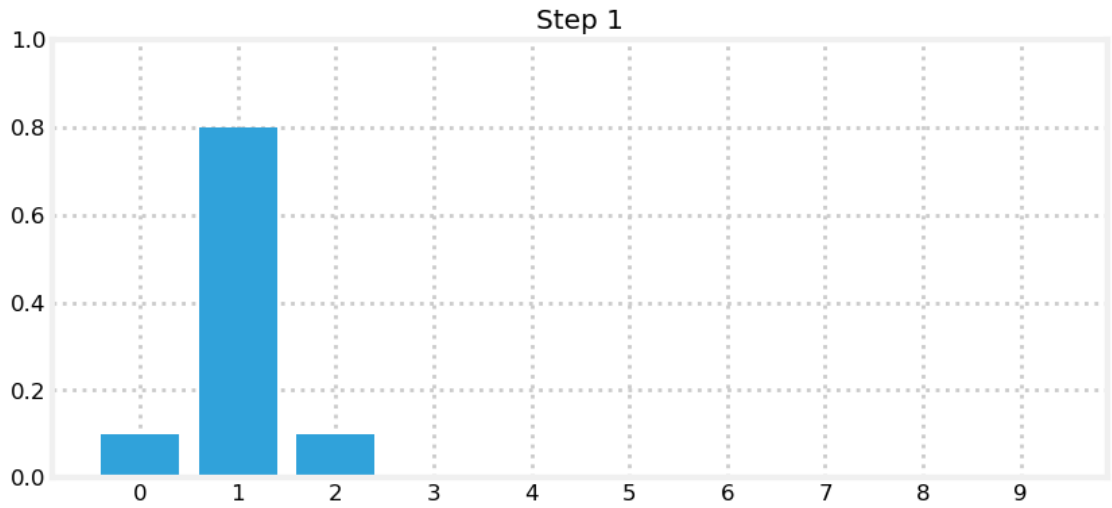



Here the results are more complicated, but you should still be able to work it out in your head. The 0.04 is due to the possibility that the 0.4 belief undershot by 1. The 0.38 is due to the following: the 80% chance that we moved 2 positions ($0.4 \times 0.8$) and the 10% chance that we undershot ($0.6 \times 0.1$). Overshooting plays no role here because if we overshot both 0.4 and 0.6 would be past this position. **I strongly suggest working some examples until all of this is very clear, as so much of what follows depends on understanding this step.**

If you look at the probabilities after performing the update you might be dismayed. In the example above we started with probabilities of 0.4 and 0.6 in two positions; after performing the update the probabilities are not only lowered, but they are strewn out across the map.

This is not a coincidence, or the result of a carefully chosen example - it is always true of the prediction. If the sensor is noisy we lose some information on every prediction. Suppose we were to perform the prediction an infinite number of times - what would the result be? If we lose information on every step, we must eventually end up with no information at all, and our probabilities will be equally distributed across the `belief` array. Let's try this with 100 iterations. The plot is animated; use the slider to change the step number.

```
In [18]: belief = np.array([1.0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
         predict_beliefs = []

         for i in range(100):
             belief = predict_move(belief, 1, .1, .8, .1)
             predict_beliefs.append(belief)

         print('Final Belief:', belief)

         # make interactive plot
         def show_prior(step):
             book_plots.bar_plot(predict_beliefs[step-1])
             plt.title('Step {}'.format(step))

         interact(show_prior, step=IntSlider(value=1, max=len(predict_beliefs)));
```




```
In [19]: print('Final Belief:', belief)

Final Belief: [0.104 0.103 0.101 0.099 0.097 0.096 0.097 0.099 0.101 0.103]
```

After 100 iterations we have lost almost all information, even though we were 100% sure that we started in position 0. Feel free to play with the numbers to see the effect of differing number of updates. For example, after 100 updates a small amount of information is left, after 50 a lot is left, but by 200 iterations essentially all information is lost.

And, if you are viewing this online here is an animation of that output.

I will not generate these standalone animations through the rest of the book. Please see the preface for instructions to run this book on the web, for free, or install IPython on your computer. This will allow you to run all of the cells and see the animations. It's very important that you practice with this code, not just read passively.

## 2.7 Generalizing with Convolution

We made the assumption that the movement error is at most one position. But it is possible for the error to be two, three, or more positions. As programmers we always want to generalize our code so that it works for all cases.

This is easily solved with *convolution*. Convolution modifies one function with another function. In our case we are modifying a probability distribution with the error function of the sensor. The implementation of `predict_move()` is a convolution, though we did not call it that. Formally, convolution is defined as

$$(f * g)(t) = \int_0^t f(\tau)\, g(t-\tau)\, \mathrm{d}\tau$$

where $f * g$ is the notation for convolving f by g. It does not mean multiply.

Integrals are for continuous functions, but we are using discrete functions. We replace the integral with a summation, and the parenthesis with array brackets.

$$(f * g)[t] = \sum_{\tau=0}^{t} f[\tau]\, g[t-\tau]$$

Comparison shows that `predict_move()` is computing this equation - it computes the sum of a series of multiplications.

Khan Academy [4] has a good introduction to convolution, and Wikipedia has some excellent animations of convolutions [5]. But the general idea is already clear. You slide an array called the *kernel* across another array, multiplying the neighbors of the current cell with the values of the second array. In our example above we used 0.8 for the probability of moving to the correct location, 0.1 for undershooting, and 0.1 for overshooting. We make a kernel of this with the array `[0.1, 0.8, 0.1]`. All we need to do is write a loop that goes over each element of our array, multiplying by the kernel, and summing the results. To emphasize that the belief is a probability distribution I have named it `pdf`.

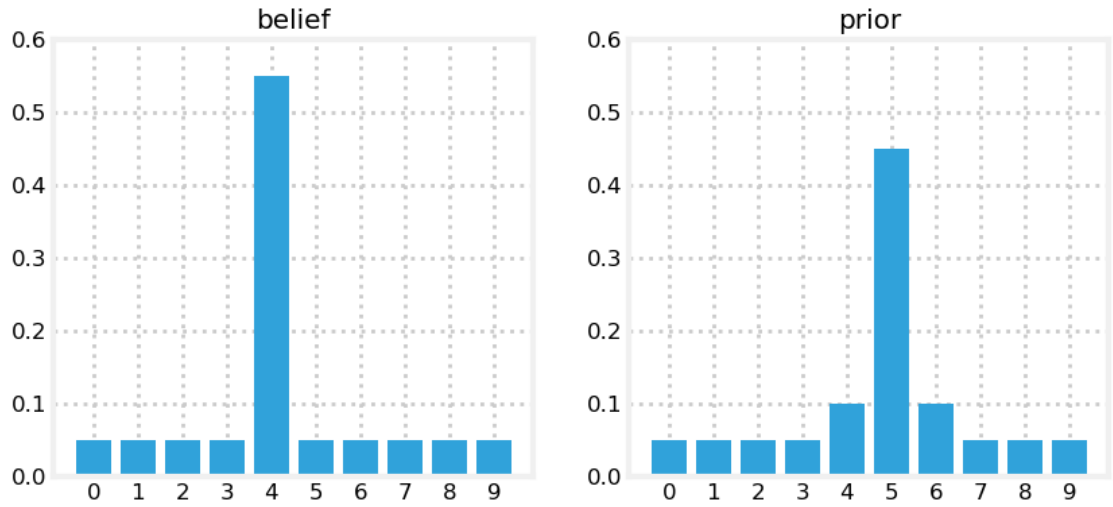
```
In [20]: def predict_move_convolution(pdf, offset, kernel):
             N = len(pdf)
             kN = len(kernel)
             width = int((kN - 1) / 2)

             prior = np.zeros(N)
             for i in range(N):
                 for k in range (kN):
                     index = (i + (width-k) - offset) % N
                     prior[i] += pdf[index] * kernel[k]
             return prior
```

This illustrates the algorithm, but it runs very slow. SciPy provides a convolution routine `convolve()` in the `ndimage.filters` module. We need to shift the pdf by `offset` before convolution; `np.roll()` does that. The move and predict algorithm can be implemented with one line:

```
convolve(np.roll(pdf, offset), kernel, mode='wrap')
```

FilterPy implements this with `discrete_bayes`' `predict()` function.

```
In [21]: from filterpy.discrete_bayes import predict

         belief = [.05, .05, .05, .05, .55, .05, .05, .05, .05, .05]
         prior = predict(belief, offset=1, kernel=[.1, .8, .1])
         book_plots.plot_belief_vs_prior(belief, prior, ylim=(0,0.6))
```




All of the elements are unchanged except the middle ones. The values in position 4 and 6 should be

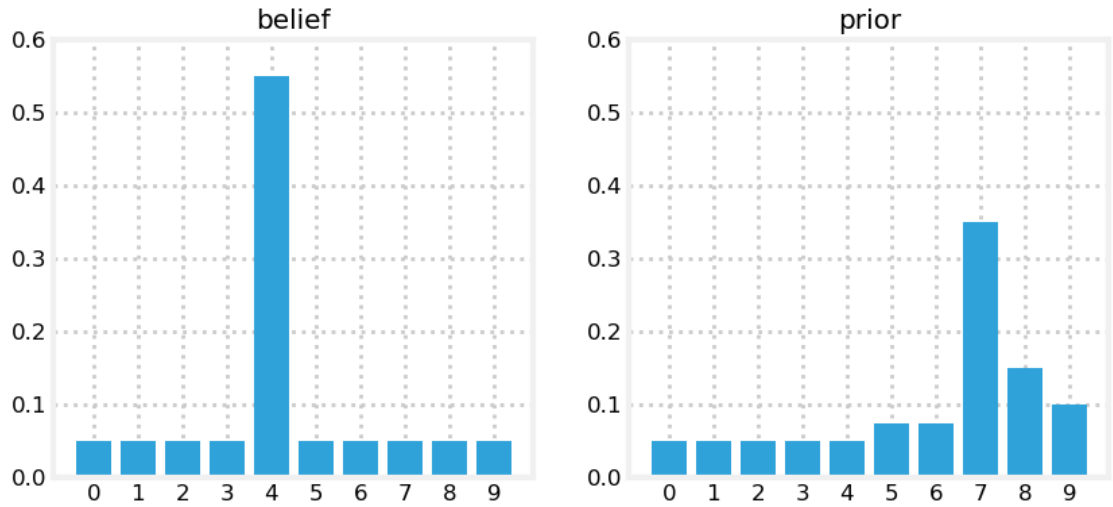
$$(0.1 \times 0.05) + (0.8 \times 0.05) + (0.1 \times 0.55) = 0.1$$

Position 5 should be

$$(0.1 \times 0.05) + (0.8 \times 0.55) + (0.1 \times 0.05) = 0.45$$

Let's ensure that it shifts the positions correctly for movements greater than one and for asymmetric kernels.

```
In [22]: prior = predict(belief, offset=3, kernel=[.05, .05, .6, .2, .1])
         book_plots.plot_belief_vs_prior(belief, prior, ylim=(0,0.6))
```



The position was correctly shifted by 3 positions and we give more weight to the likelihood of an overshoot vs an undershoot, so this looks correct.

Make sure you understand what we are doing. We are making a prediction of where the dog is moving, and convolving the probabilities to get the prior.

If we weren't using probabilities we would use this equation that I gave earlier:

$$\bar{x}_{k+1} = x_k + f_{\mathbf{x}}(\bullet)$$

The prior, our prediction of where the dog will be, is the amount the dog moved plus his current position. The dog was at 10, he moved 5 meters, so he is now at 15 m. It couldn't be simpler. But we are using probabilities to model this, so our equation is:

$$\bar{\mathbf{x}}_{k+1} = \mathbf{x}_k * f_{\mathbf{x}}(\bullet)$$

We are *convolving* the current probabilistic position estimate with a probabilistic estimate of how much we think the dog moved. It's the same concept, but the math is slightly different. $\mathbf{x}$ is bold to denote that it is an array of numbers.

## 2.8 Integrating Measurements and Movement Updates

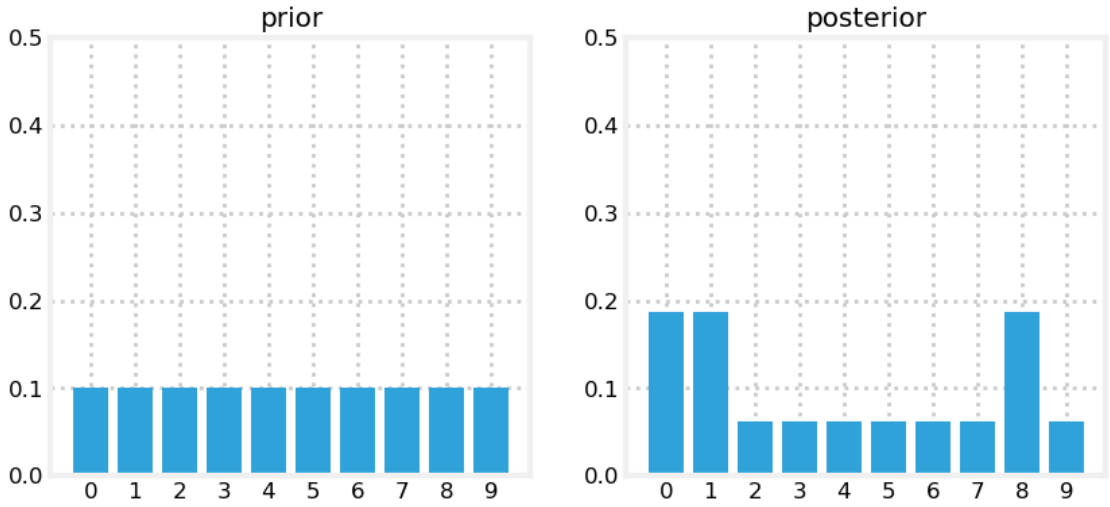
The problem of losing information during a prediction may make it seem as if our system would quickly devolve into having no knowledge. However, each prediction is followed by an update where we incorporate the measurement into the estimate. The update improves our knowledge. The output of the update step is fed into the next prediction. The prediction degrades our certainty. That is passed into another update, where certainty is again increased.

Let's think about this intuitively. Consider a simple case - you are tracking a dog while he sits still. During each prediction you predict he doesn't move. Your filter quickly *converges* on an accurate estimate of his position. Then the microwave in the kitchen turns on, and he goes streaking off. You don't know this, so at the next prediction you predict he is in the same spot. But the measurements tell a different story. As you incorporate the measurements your belief will be smeared along the hallway, leading towards the kitchen. On every epoch (cycle) your belief that he is sitting still will get smaller, and your belief that he is inbound towards the kitchen at a startling rate of speed increases.

That is what intuition tells us. What does the math tell us?

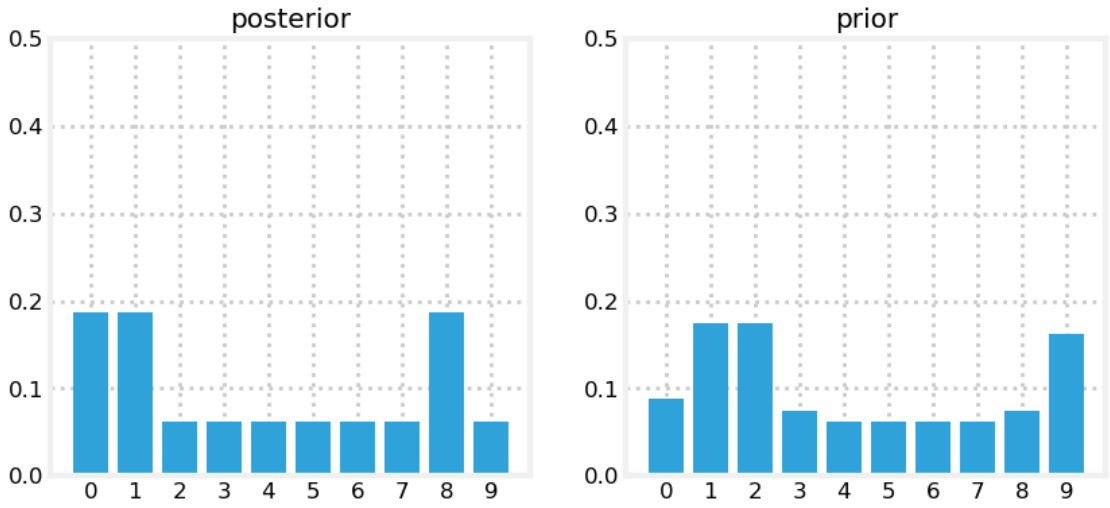
We have already programmed the update and predict steps. All we need to do is feed the result of one into the other, and we will have implemented a dog tracker!!! Let's see how it performs. We will input measurements as if the dog started at position 0 and moved right one position each epoch. As in a real world application, we will start with no knowledge of his position by assigning equal probability to all positions.

In [23]:
```
from filterpy.discrete_bayes import update

hallway = np.array([1, 1, 0, 0, 0, 0, 0, 0, 1, 0])
prior = np.array([.1] * 10)
likelihood = lh_hallway(hallway, z=1, z_prob=.75)
posterior = update(likelihood, prior)
book_plots.plot_prior_vs_posterior(prior, posterior, ylim=(0,.5))
```



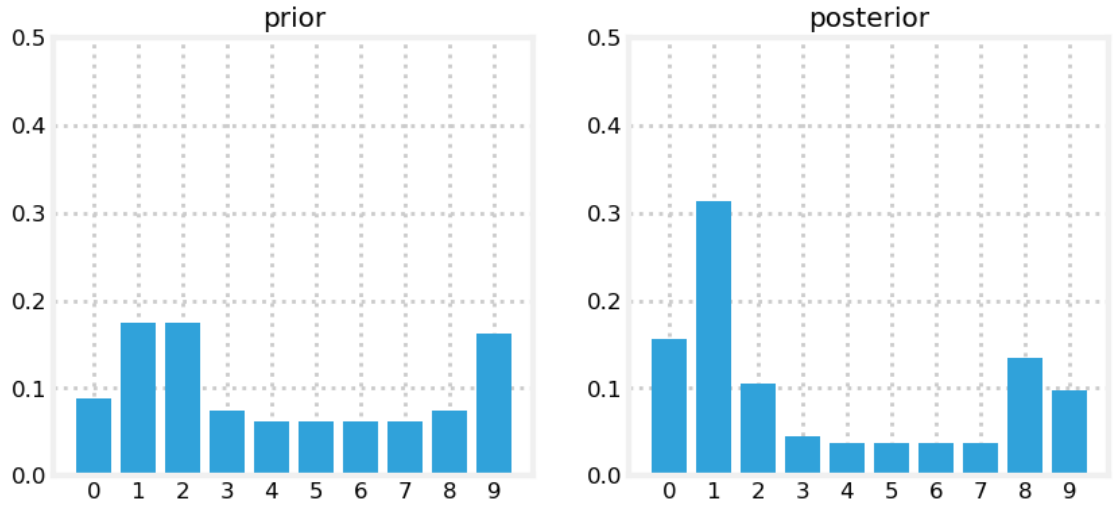

After the first update we have assigned a high probability to each door position, and a low probability to each wall position.

```
In [24]: kernel = (.1, .8, .1)
         prior = predict(posterior, 1, kernel)
         book_plots.plot_prior_vs_posterior(prior, posterior, True, ylim=(0,.5))
```



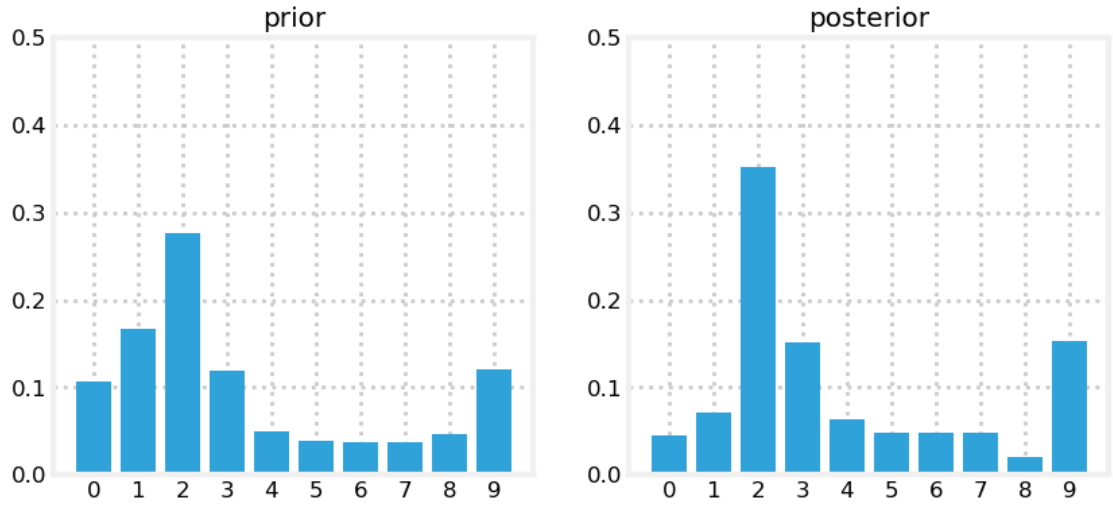

The predict step shifted these probabilities to the right, smearing them about a bit. Now let's look at what happens at the next sense.

```
In [25]: likelihood = lh_hallway(hallway, z=1, z_prob=.75)
         posterior = update(likelihood, prior)
         book_plots.plot_prior_vs_posterior(prior, posterior, ylim=(0,.5))
```



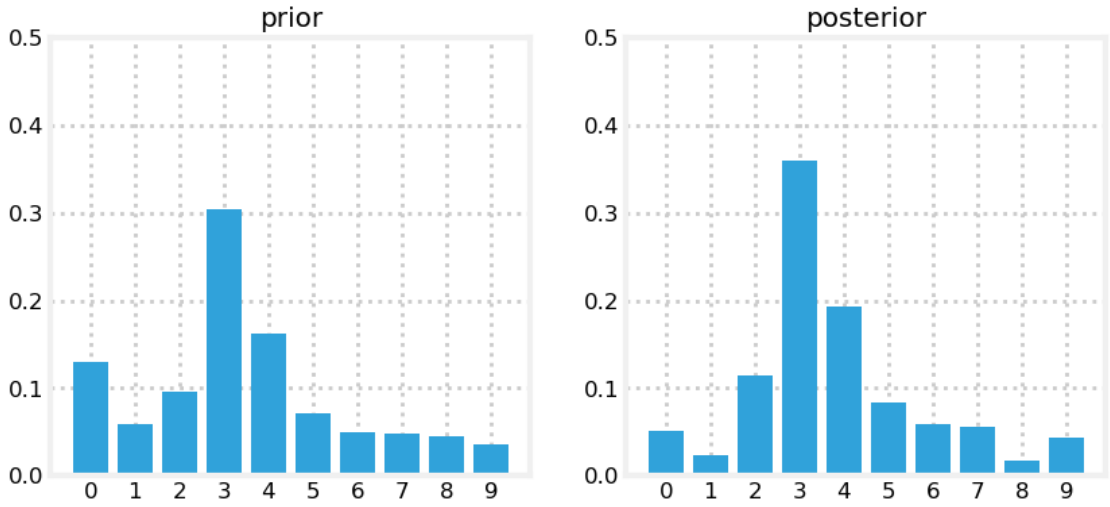

Notice the tall bar at position 1. This corresponds with the (correct) case of starting at position 0, sensing a door, shifting 1 to the right, and sensing another door. No other positions make this set of observations as likely. Now we will add an update and then sense the wall.

```
In [26]: prior = predict(posterior, 1, kernel)
         likelihood = lh_hallway(hallway, z=0, z_prob=.75)
         posterior = update(likelihood, prior)
         book_plots.plot_prior_vs_posterior(prior, posterior, ylim=(0,.5))
```




This is exciting! We have a very prominent bar at position 2 with a value of around 35%. It is over twice the value of any other bar in the plot, and is about 4% larger than our last plot, where the tallest bar was around 31%. Let's see one more cycle.

```
In [27]: prior = predict(posterior, 1, kernel)
         likelihood = lh_hallway(hallway, z=0, z_prob=.75)
         posterior = update(likelihood, prior)
         book_plots.plot_prior_vs_posterior(prior, posterior, ylim=(0,.5))
```



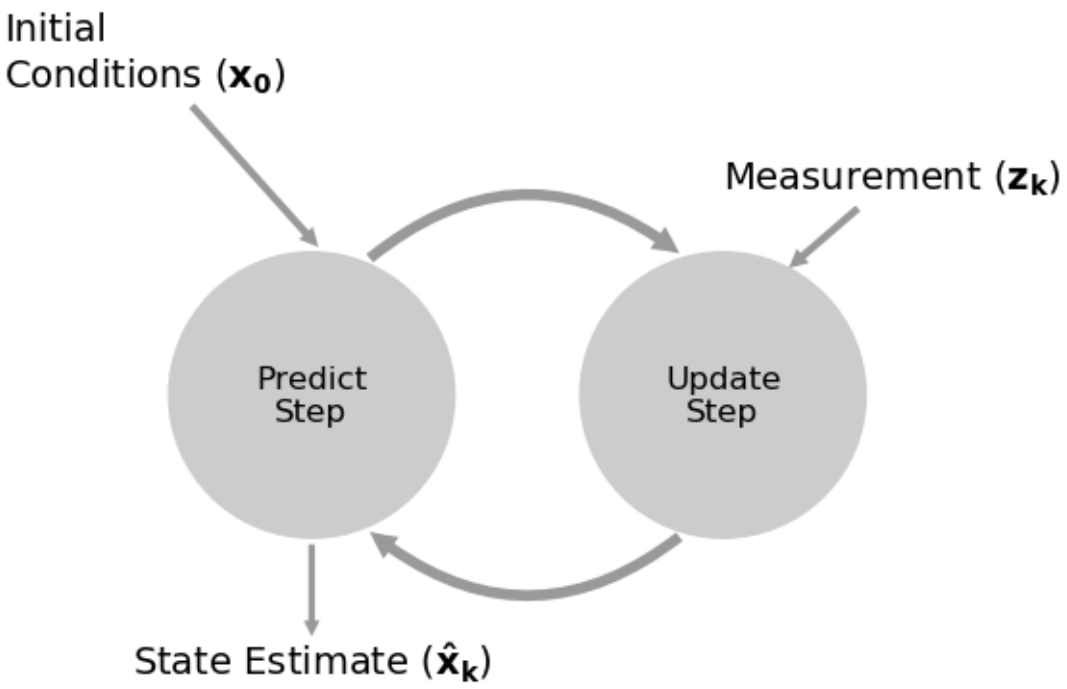

I ignored an important issue. Earlier I assumed that we had a motion sensor for the predict step; then, when talking about the dog and the microwave I assumed that you had no knowledge that he suddenly began running. I mentioned that your belief that the dog is running would increase over time, but I did not provide any code for this. In short, how do we detect and/or estimate changes in the process model if we aren't directly measuring it?

For now I want to ignore this problem. In later chapters we will learn the mathematics behind this estimation; for now it is a large enough task just to learn this algorithm. It is profoundly important to solve this problem, but we haven't yet built enough of the mathematical apparatus that is required, and so for the remainder of the chapter we will ignore the problem by assuming we have a sensor that senses movement.

## 2.9 The Discrete Bayes Algorithm

This chart illustrates the algorithm:

```
In [28]: book_plots.predict_update_chart()
```




This filter is a form of the g-h filter. Here we are using the percentages for the errors to implicitly compute the $g$ and $h$ parameters. We could express the discrete Bayes algorithm as a g-h filter, but that would obscure the logic of this filter.

The filter equations are:

$$\begin{aligned} \bar{\mathbf{x}} &= \mathbf{x} * f_{\mathbf{x}}(\bullet) && \text{Predict Step} \\ \mathbf{x} &= \|\mathcal{L} \cdot \bar{\mathbf{x}}\| && \text{Update Step} \end{aligned}$$

$\mathcal{L}$ is the usual way to write the likelihood function, so I use that. The $\|\|$ notation denotes taking the norm. We need to normalize the product of the likelihood with the prior to ensure $x$ is a probability distribution that sums to one.

We can express this in pseudocode.

**Initialization**

```
1. Initialize our belief in the state
```

**Predict**

```
1. Based on the system behavior, predict state for the next time step
2. Adjust belief to account for the uncertainty in prediction
```

**Update**

```
1. Get a measurement and associated belief about its accuracy
2. Compute how likely it is the measurement matches each state
3. Update state belief with this likelihood
```

When we cover the Kalman filter we will use this exact same algorithm; only the details of the computation will differ.

Algorithms in this form are sometimes called *predictor correctors*. We make a prediction, then correct them.

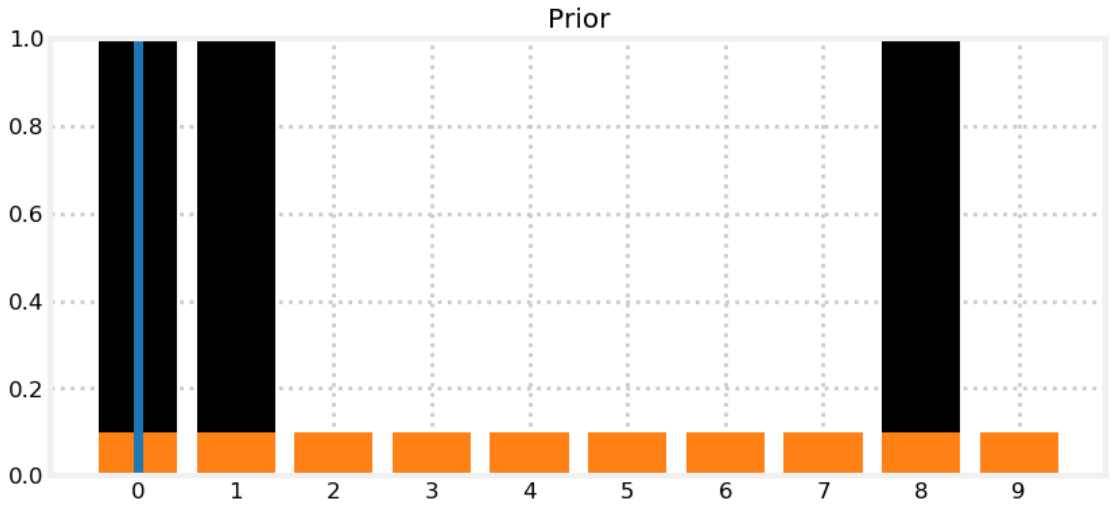
Let's animate this. First Let's write functions to perform the filtering and to plot the results at any step. I've plotted the position of the doorways in black. Prior are drawn in orange, and the posterior in blue. I draw a thick vertical line to indicate where Simon really is. This is not an output of the filter - we know where Simon is only because we are simulating his movement.

```
In [29]: def discrete_bayes_sim(prior, kernel, measurements, z_prob, hallway):
             posterior = np.array([.1]*10)
             priors, posteriors = [], []
             for i, z in enumerate(measurements):
                 prior = predict(posterior, 1, kernel)
                 priors.append(prior)

                 likelihood = lh_hallway(hallway, z, z_prob)
                 posterior = update(likelihood, prior)
                 posteriors.append(posterior)
             return priors, posteriors


         def plot_posterior(hallway, posteriors, i):
             plt.title('Posterior')
             book_plots.bar_plot(hallway, c='k')
             book_plots.bar_plot(posteriors[i], ylim=(0, 1.0))
             plt.axvline(i % len(hallway), lw=5)

         def plot_prior(hallway, priors, i):
             plt.title('Prior')
             book_plots.bar_plot(hallway, c='k')
             book_plots.bar_plot(priors[i], ylim=(0, 1.0), c='#ff8015')
             plt.axvline(i % len(hallway), lw=5)

         def animate_discrete_bayes(hallway, priors, posteriors):
             def animate(step):
                 step -= 1
                 i = step // 2
                 if step % 2 == 0:
                     plot_prior(hallway, priors, i)
                 else:
                     plot_posterior(hallway, posteriors, i)

             return animate
```

Let's run the filter and animate it.

```
In [30]: # change these numbers to alter the simulation
         kernel = (.1, .8, .1)
         z_prob = 1.0
         hallway = np.array([1, 1, 0, 0, 0, 0, 0, 0, 1, 0])

         # measurements with no noise
         zs = [hallway[i % len(hallway)] for i in range(50)]
```

```
priors, posteriors = discrete_bayes_sim(prior, kernel, zs, z_prob, hallway)
interact(animate_discrete_bayes(hallway, priors, posteriors), step=IntSlider(value=1, max=len(
```




Now we can see the results. You can see how the prior shifts the position and reduces certainty, and the posterior stays in the same position and increases certainty as it incorporates the information from the measurement. I've made the measurement perfect with the line `z_prob = 1.0`; we will explore the effect of imperfect measurements in the next section. Finally,

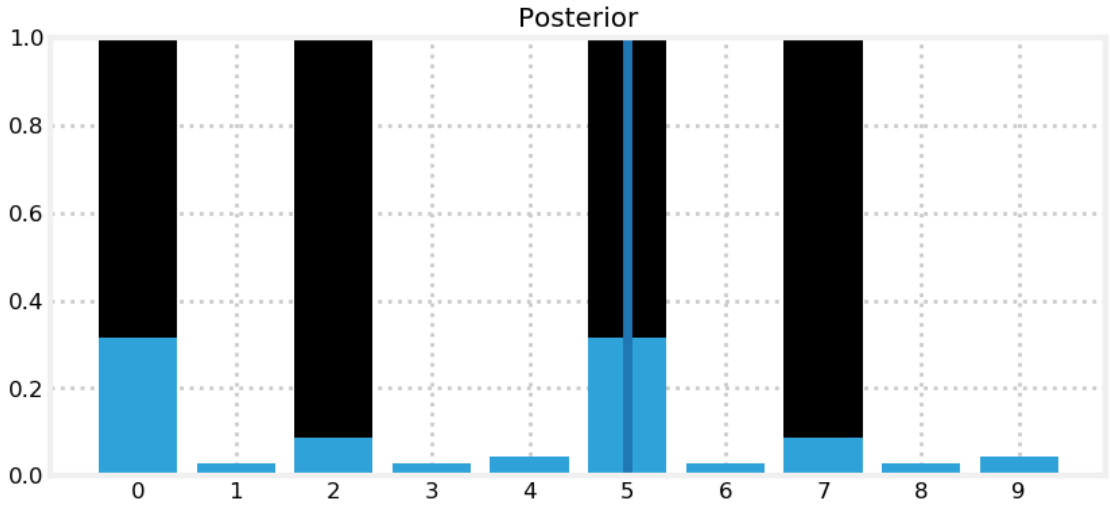
Another thing to note is how accurate our estimate becomes when we are in front of a door, and how it degrades when in the middle of the hallway. This should make intuitive sense. There are only a few doorways, so when the sensor tells us we are in front of a door this boosts our certainty in our position. A long stretch of no doors reduces our certainty.

## 2.10 The Effect of Bad Sensor Data

You may be suspicious of the results above because I always passed correct sensor data into the functions. However, we are claiming that this code implements a *filter* - it should filter out bad sensor measurements. Does it do that?

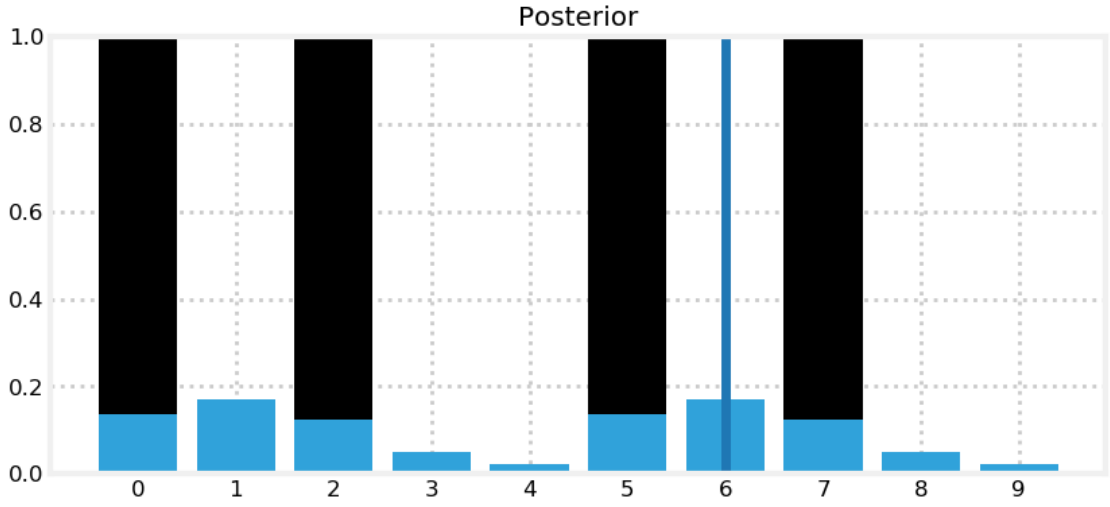
To make this easy to program and visualize I will change the layout of the hallway to mostly alternating doors and hallways, and run the algorithm on 6 correct measurements:

```
In [31]: hallway = np.array([1, 0, 1, 0, 0]*2)
         kernel = (.1, .8, .1)
         prior = np.array([.1] * 10)
         zs = [1, 0, 1, 0, 0, 1]
         z_prob = 0.75
         priors, posteriors = discrete_bayes_sim(prior, kernel, zs, z_prob, hallway)
         interact(animate_discrete_bayes(hallway, priors, posteriors), step=IntSlider(value=12, max=len
```




We have identified the likely cases of having started at position 0 or 5, because we saw this sequence of doors and walls: 1,0,1,0,0. Now I inject a bad measurement. The next measurement should be 0, but instead we get a 1:

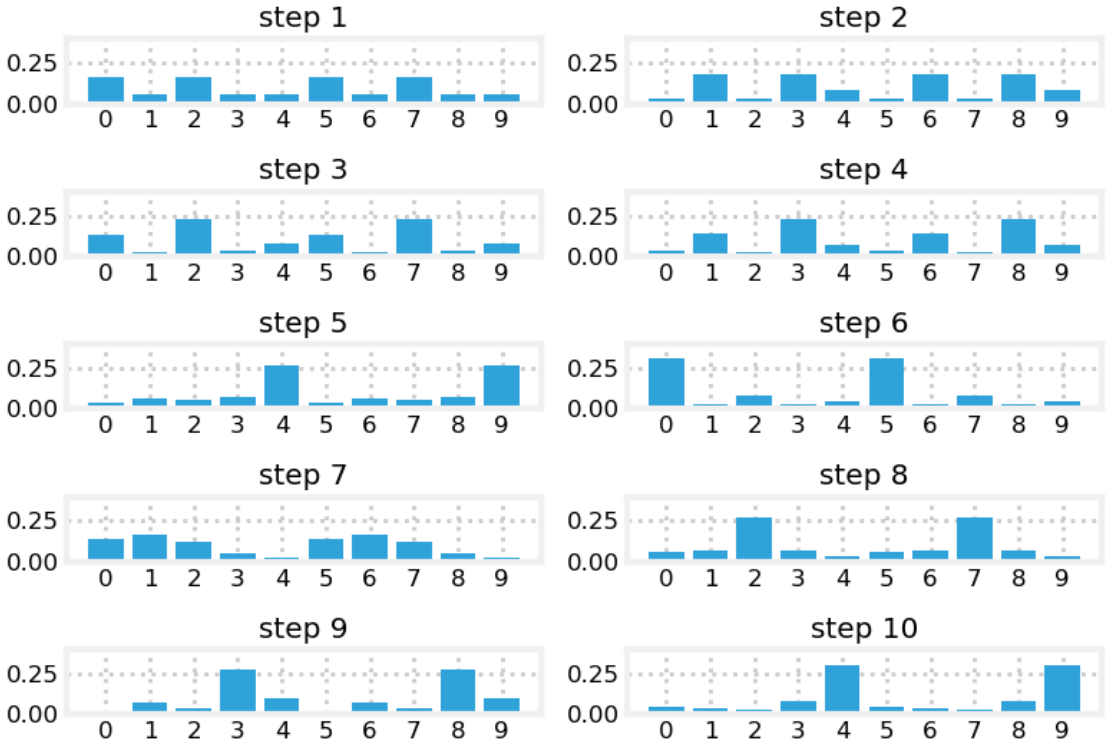
```
In [32]: measurements = [1, 0, 1, 0, 0, 1, 1]
         priors, posteriors = discrete_bayes_sim(prior, kernel, measurements, z_prob, hallway);
         plot_posterior(hallway, posteriors, 6)
```




That one bad measurement has significantly eroded our knowledge. Now let's continue with a series of correct measurements.

```
In [33]: with figsize(y=5.5):
             measurements = [1, 0, 1, 0, 0, 1, 1, 1, 0, 0]
             for i, m in enumerate(measurements):
                 likelihood = lh_hallway(hallway, z=m, z_prob=.75)
                 posterior = update(likelihood, prior)
```

```
        prior = predict(posterior, 1, kernel)
        plt.subplot(5, 2, i+1)
        book_plots.bar_plot(posterior, ylim=(0, .4), title='step {}'.format(i+1))
    plt.tight_layout()
```




We quickly filtered out the bad sensor reading and converged on the most likely positions for our dog.

## 2.11 Drawbacks and Limitations

Do not be mislead by the simplicity of the examples I chose. This is a robust and complete filter, and you may use the code in real world solutions. If you need a multimodal, discrete filter, this filter works.

With that said, this filter it is not used often because it has several limitations. Getting around those limitations is the motivation behind the chapters in the rest of this book.

The first problem is scaling. Our dog tracking problem used only one variable, *pos*, to denote the dog's position. Most interesting problems will want to track several things in a large space. Realistically, at a minimum we would want to track our dog's $(x, y)$ coordinate, and probably his velocity $(\dot{x}, \dot{y})$ as well. We have not covered the multidimensional case, but instead of an array we use a multidimensional grid to store the probabilities at each discrete location. Each `update()` and `predict()` step requires updating all values in the grid, so a simple four variable problem would require $O(n^4)$ running time *per time step*. Realistic filters can have 10 or more variables to track, leading to exorbitant computation requirements.

The second problem is that the filter is discrete, but we live in a continuous world. The histogram requires that you model the output of your filter as a set of discrete points. A 100 meter hallway requires 10,000 positions to model the hallway to 1cm accuracy. So each update and predict operation would entail performing calculations for 10,000 different probabilities. It gets exponentially worse as we add dimensions. A 100x100 $m^2$ courtyard requires 100,000,000 bins to get 1cm accuracy.

A third problem is that the filter is multimodal. In the last example we ended up with strong beliefs that the dog was in position 4 or 9. This is not always a problem. Particle filters, which we will study later,

are multimodal and are often used because of this property. But imagine if the GPS in your car reported to you that it is 40% sure that you are on D street, and 30% sure you are on Willow Avenue.

A forth problem is that it requires a measurement of the change in state. We need a motion sensor to detect how much the dog moves. There are ways to work around this problem, but it would complicate the exposition of this chapter, so, given the aforementioned problems, I will not discuss it further.

With that said, if I had a small problem that this technique could handle I would choose to use it; it is trivial to implement, debug, and understand, all virtues.

## 2.12 Tracking and Control

We have been passively tracking an autonomously moving object. But consider this very similar problem. I am automating a warehouse and want to use robots to collect all of the items for a customer's order. Perhaps the easiest way to do this is to have the robots travel on a train track. I want to be able to send the robot a destination and have it go there. But train tracks and robot motors are imperfect. Wheel slippage and imperfect motors means that the robot is unlikely to travel to exactly the position you command. There is more than one robot, and we need to know where they all are so we do not cause them to crash.

So we add sensors. Perhaps we mount magnets on the track every few feet, and use a Hall sensor to count how many magnets are passed. If we count 10 magnets then the robot should be at the 10th magnet. Of course it is possible to either miss a magnet or to count it twice, so we have to accommodate some degree of error. We can use the code from the previous section to track our robot since magnet counting is very similar to doorway sensing.

But we are not done. We've learned to never throw information away. If you have information you should use it to improve your estimate. What information are we leaving out? We know what control inputs we are feeding to the wheels of the robot at each moment in time. For example, let's say that once a second we send a movement command to the robot - move left 1 unit, move right 1 unit, or stand still. If I send the command 'move left 1 unit' I expect that in one second from now the robot will be 1 unit to the left of where it is now. This is a simplification because I am not taking acceleration into account, but I am not trying to teach control theory. Wheels and motors are imperfect. The robot might end up 0.9 units away, or maybe 1.2 units.

Now the entire solution is clear. We assumed that the dog kept moving in whatever direction he was previously moving. That is a dubious assumption for my dog! Robots are far more predictable. Instead of making a dubious prediction based on assumption of behavior we will feed in the command that we sent to the robot! In other words, when we call `predict()` we will pass in the commanded movement that we gave the robot along with a kernel that describes the likelihood of that movement.

### 2.12.1 Simulating the Train Behavior

We need to simulate an imperfect train. When we command it to move it will sometimes make a small mistake, and its sensor will sometimes return the incorrect value.

```
In [34]: class Train(object):

             def __init__(self, track_len, kernel=[1.], sensor_accuracy=.9):
                 self.track_len = track_len
                 self.pos = 0
                 self.kernel = kernel
                 self.sensor_accuracy = sensor_accuracy

             def move(self, distance=1):
                 """ move in the specified direction
                 with some small chance of error"""

                 self.pos += distance
                 # insert random movement error according to kernel
```

```
            r = random.random()
            s = 0
            offset = -(len(self.kernel) - 1) / 2
            for k in self.kernel:
                s += k
                if r <= s:
                    break
                offset += 1
            self.pos = int((self.pos + offset) % self.track_len)
            return self.pos

        def sense(self):
            pos = self.pos
             # insert random sensor error
            if random.random() > self.sensor_accuracy:
                if random.random() > 0.5:
                    pos += 1
                else:
                    pos -= 1
            return pos
```

With that we are ready to write the filter. We will put it in a function so that we can run it with different assumptions. I will assume that the robot always starts at the beginning of the track. The track is implemented as being 10 units long, but think of it as a track of length, say 10,000, with the magnet pattern repeated every 10 units. A length of 10 makes it easier to plot and inspect.

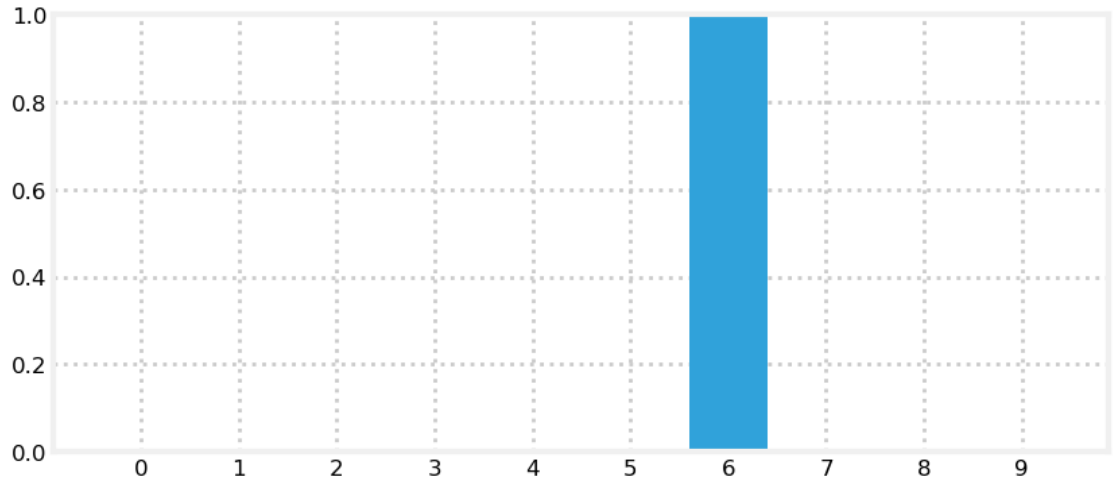
```
In [35]: def train_filter(iterations, kernel, sensor_accuracy,
                         move_distance, do_print=True):
            track = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
            prior = np.array([.9] + [0.01]*9)
            posterior = prior[:]
            normalize(prior)

            robot = Train(len(track), kernel, sensor_accuracy)
            for i in range(iterations):
                # move the robot and
                robot.move(distance=move_distance)

                # peform prediction
                prior = predict(posterior, move_distance, kernel)

                #  and update the filter
                m = robot.sense()
                likelihood = lh_hallway(track, m, sensor_accuracy)
                posterior = update(likelihood, prior)
                index = np.argmax(posterior)

                if do_print:
                    print('''time {}: pos {}, sensed {}, '''
                          '''at position {}'''.format(
                            i, robot.pos, m, track[robot.pos]))

                    print('''        estimated position is {}'''
                          ''' with confidence {:.4f}%:'''.format(
                          index, posterior[index]*100))
```

```
            book_plots.bar_plot(posterior)
            if do_print:
                print()
                print('final position is', robot.pos)
                index = np.argmax(posterior)
                print('''Estimated position is {} with '''
                      '''confidence {:.4f}%:'''.format(
                        index, posterior[index]*100))
```

Read the code and make sure you understand it. Now let's do a run with no sensor or movement error. If the code is correct it should be able to locate the robot with no error. The output is a bit tedious to read, but if you are at all unsure of how the update/predict cycle works make sure you read through it carefully to solidify your understanding.

```
In [36]: import random

         random.seed(3)
         np.set_printoptions(precision=2, suppress=True, linewidth=60)
         train_filter(4, kernel=[1.], sensor_accuracy=.999,
                      move_distance=4, do_print=True)
```

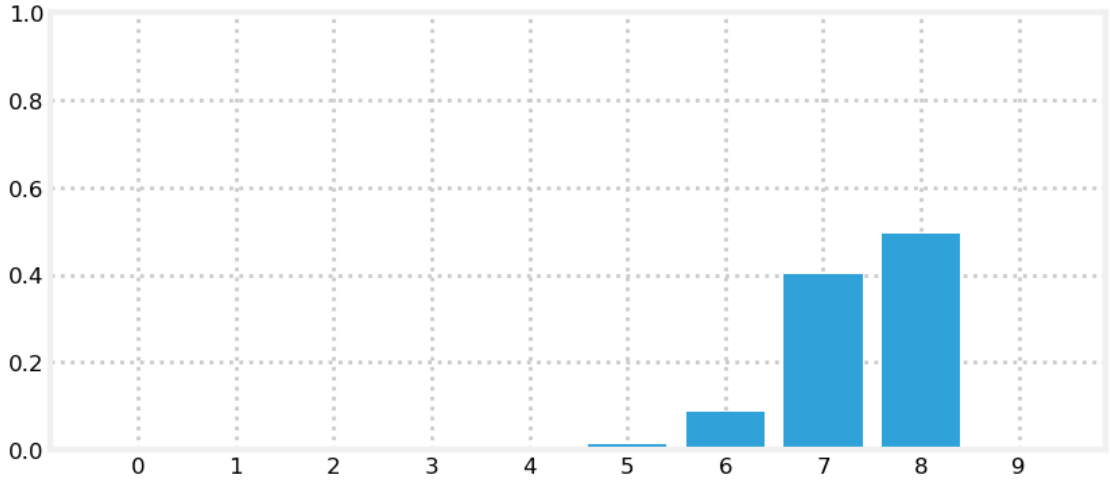
```
time 0: pos 4, sensed 4, at position 4
        estimated position is 4 with confidence 99.9900%:
time 1: pos 8, sensed 8, at position 8
        estimated position is 8 with confidence 100.0000%:
time 2: pos 2, sensed 2, at position 2
        estimated position is 2 with confidence 100.0000%:
time 3: pos 6, sensed 6, at position 6
        estimated position is 6 with confidence 100.0000%:

final position is 6
Estimated position is 6 with confidence 100.0000%:
```




We can see that the code was able to perfectly track the robot so we should feel reasonably confident that the code is working. Now let's see how it fairs with some errors.

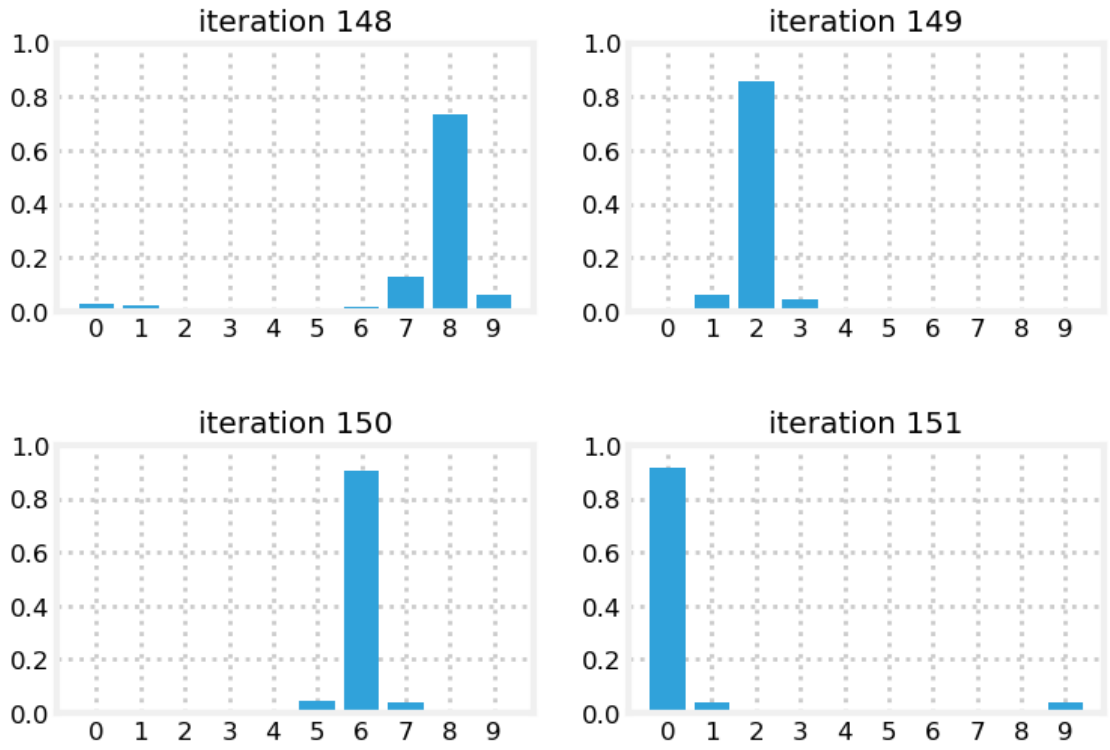
```
In [37]: random.seed(5)
         train_filter(4, kernel=[.1, .8, .1], sensor_accuracy=.9,
                  move_distance=4, do_print=True)
```

```
time 0: pos 4, sensed 4, at position 4
        estimated position is 4 with confidence 96.0390%:
time 1: pos 8, sensed 9, at position 8
        estimated position is 9 with confidence 52.1180%:
time 2: pos 3, sensed 3, at position 3
        estimated position is 3 with confidence 88.3993%:
time 3: pos 7, sensed 8, at position 7
        estimated position is 8 with confidence 49.3174%:

final position is 7
Estimated position is 8 with confidence 49.3174%:
```




There was a sensing error at time 1, but we are still quite confident in our position. Now let's run a very long simulation and see how the filter responds to errors.

```
In [38]: with figsize(y=5.5):
             for i in range (4):
                 random.seed(3)
                 plt.subplot(221+i)
                 train_filter(148+i, kernel=[.1, .8, .1],
                              sensor_accuracy=.8,
                              move_distance=4, do_print=False)
                 plt.title ('iteration {}'.format(148+i))
```




We can see that there was a problem on iteration 149 as the confidence degrades. But within a few iterations the filter is able to correct itself and regain confidence in the estimated position.

## 2.13 Bayes Theorem and the Total Probability Theorem

We developed the math in this chapter merely by reasoning about the information we have at each moment. In the process we discovered *Bayes' Theorem* and the *Total Probability Theorem*.

Bayes theorem tells us how to compute the probability of an event given previous information.

We implemented the `update()` function with this probability calculation:

$$\texttt{posterior} = \frac{\texttt{likelihood} \times \texttt{prior}}{\texttt{normalization factor}}$$

We haven't developed the mathematics to discuss Bayes yet, but this is Bayes' theorem. Every filter in this book is an expression of Bayes' theorem. In the next chapter we will develop the mathematics, but in many ways that obscures the simple idea expressed in this equation:

$$\textit{updated knowledge} = \|\textit{likelihood of new knowledge} \times \textit{prior knowledge}\|$$

where $\|\cdot\|$ expresses normalizing the term.

We came to this with simple reasoning about a dog walking down a hallway. Yet, as we will see the same equation applies to a universe of filtering problems. We will use this equation in every subsequent chapter.

Likewise, the `predict()` step computes the total probability of multiple possible events. This is known as the *Total Probability Theorem* in statistics, and we will also cover this in the next chapter after developing some supporting math.

For now I need you to understand that Bayes' theorem is a formula to incorporate new information into existing information.

## 2.14 Summary

The code is very short, but the result is impressive! We have implemented a form of a Bayesian filter. We have learned how to start with no information and derive information from noisy sensors. Even though the sensors in this chapter are very noisy (most sensors are more than 80% accurate, for example) we quickly converge on the most likely position for our dog. We have learned how the predict step always degrades our knowledge, but the addition of another measurement, even when it might have noise in it, improves our knowledge, allowing us to converge on the most likely result.

This book is mostly about the Kalman filter. The math it uses is different, but the logic is exactly the same as used in this chapter. It uses Bayesian reasoning to form estimates from a combination of measurements and process models.

**If you can understand this chapter you will be able to understand and implement Kalman filters.** I cannot stress this enough. If anything is murky, go back and reread this chapter and play with the code. The rest of this book will build on the algorithms that we use here. If you don't understand why this filter works you will have little success with the rest of the material. However, if you grasp the fundamental insight - multiplying probabilities when we measure, and shifting probabilities when we update leads to a converging solution - then after learning a bit of math you are ready to implement a Kalman filter.

## 2.15 References


- [1] D. Fox, W. Burgard, and S. Thrun. "Monte carlo localization: Efficient position estimation for mobile robots." In *Journal of Artifical Intelligence Research*, 1999.

  http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume11/fox99a-html/jair-localize.html

- [2] Dieter Fox, et. al. "Bayesian Filters for Location Estimation". In *IEEE Pervasive Computing*, September 2003.

  http://swarmlab.unimaas.nl/wp-content/uploads/2012/07/fox2003bayesian.pdf

- [3] Sebastian Thrun. "Artificial Intelligence for Robotics".

  https://www.udacity.com/course/cs373

- [4] Khan Acadamy. "Introduction to the Convolution"

  https://www.khanacademy.org/math/differential-equations/laplace-transform/convolution-integral/v/introduction-to-the-convolution

- [5] Wikipedia. "Convolution"

  http://en.wikipedia.org/wiki/Convolution

- [6] Wikipedia. "Law of total probability"
  http://en.wikipedia.org/wiki/Law_of_total_probability

- [7] Wikipedia. "Time Evolution"

  https://en.wikipedia.org/wiki/Time_evolution

- [8] We need to rethink how we teach statistics from the ground up

  http://www.statslife.org.uk/opinion/2405-we-need-to-rethink-how-we-teach-statistics-from-the-ground-up

# Chapter 3

# Probabilities, Gaussians, and Bayes' Theorem

```
In [1]: from __future__ import division, print_function
        %matplotlib inline

In [2]: #format the book
        import book_format
        book_format.set_style()

Out[2]:
```

## 3.1 Introduction

The last chapter ended by discussing some of the drawbacks of the Discrete Bayesian filter. For many tracking and filtering problems our desire is to have a filter that is *unimodal* and *continuous*. That is, we want to model our system using floating point math (continuous) and to have only one belief represented (unimodal). For example, we want to say an aircraft is at (12.34, -95.54, 2389.5) where that is latitude, longitude, and altitude. We do not want our filter to tell us "it might be at (1.65, -78.01, 2100.45) or it might be at (34.36, -98.23, 2543.79)." That doesn't match our physical intuition of how the world works, and as we discussed, it can be prohibitively expensive to compute the multimodal case. And, of course, multiple position estimates makes navigating impossible.

We desire a unimodal, continuous way to represent probabilities that models how the real world works, and that is computationally efficient to calculate. Gaussian distributions provide all of these features.

## 3.2 Mean, Variance, and Standard Deviations

Most of you will have had exposure to statistics, but allow me to cover this material anyway. I ask that you read the material even if you are sure you know it well. I ask for two reasons. First, I want to be sure that we are using terms in the same way. Second, I strive to form an intuitive understanding of statistics that will serve you well in later chapters. It's easy to go through a stats course and only remember the formulas and calculations, and perhaps be fuzzy on the implications of what you have learned.

### 3.2.1 Random Variables

Each time you roll a die the *outcome* will be between 1 and 6. If we rolled a fair die a million times we'd expect to get a one 1/6 of the time. Thus we say the *probability*, or *odds* of the outcome 1 is 1/6. Likewise, if I asked you the chance of 1 being the result of the next roll you'd reply 1/6.

This combination of values and associated probabilities is called a *random variable*. Here *random* does not mean the process is nondeterministic, only that we lack information about the outcome. The result of a die toss is deterministic, but we lack enough information to compute the result. We don't know what will happen, except probabilistically.

While we are defining terms, the range of values is called the *sample space*. For a die the sample space is {1, 2, 3, 4, 5, 6}. For a coin the sample space is {H, T}. *Space* is a mathematical term which means a set with structure. The sample space for the die is a subset of the natural numbers in the range of 1 to 6.

Another example of a random variable is the heights of students in a university. Here the sample space is a range of values in the real numbers between two limits defined by biology.

Random variables such as coin tosses and die rolls are *discrete random variables*. This means their sample space is represented by either a finite number of values or a countably infinite number of values such as the natural numbers. Heights of humans are called *continuous random variables* since they can take on any real value between two limits.

Do not confuse the *measurement* of the random variable with the actual value. If we can only measure the height of a person to 0.1 meters we would only record values from 0.1, 0.2, 0.3…2.7, yielding 27 discrete choices. Nonetheless a person's height can vary between any arbitrary real value between those ranges, and so height is a continuous random variable.

In statistics capital letters are used for random variables, usually from the latter half of the alphabet. So, we might say that $X$ is the random variable representing the die toss, or $Y$ are the heights of the students in the freshmen poetry class. Later chapters use linear algebra to solve these problems, and so there we will follow the convention of using lower case for vectors, and upper case for matrices. Unfortunately these conventions clash, and you will have to determine which an author is using from context. I always use bold symbols for vectors and matrices, which helps distinguish between the two.

## 3.3 Probability Distribution

The *probability distribution* gives the probability for the random variable to take any value in a sample space. For example, for a fair six sided die we might say:

| Value | Probability |
|---|---|
| 1 | 1/6 |
| 2 | 1/6 |
| 3 | 1/6 |
| 4 | 1/6 |
| 5 | 1/6 |
| 6 | 1/6 |

We denote this distribution with a lower case p: $p(x)$. Using ordinary function notation, we would write:

$$P(X{=}4) = p(4) = \frac{1}{6}$$

This states that the probability of the die landing on 4 is $\frac{1}{6}$. $P(X{=}x_k)$ is notation for "the probability of $X$ being $x_k$". Note the subtle notational difference. The capital $P$ denotes the probability of a single event, and the lower case $p$ is the probability distribution function. This can lead you astray if you are not observent. Some texts use $Pr$ instead of $P$ to ameliorate this.

Another example is a fair coin. It has the sample space {H, T}. The coin is fair, so the probability for heads (H) is 50%, and the probability for tails (T) is 50%. We write this as

$$P(X{=}H) = 0.5$$
$$P(X{=}T) = 0.5$$

Sample spaces are not unique. One sample space for a die is {1, 2, 3, 4, 5, 6}. Another valid sample space would be {even, odd}. Another might be {dots in all corners, not dots in all corners}. A sample space

is valid so long as it covers all possibilities, and any single event is described by only one element. {even, 1, 3, 4, 5} is not a valid sample space for a die since a value of 4 is matched both by 'even' and '4'.

The probabilities for all values of a *discrete random value* is known as the *discrete probability distribution* and the probabilities for all values of a *continuous random value* is known as the *continuous probability distribution*.

To be a probability distribution the probability of each value $x_i$ must be $x_i \geq 0$, since no probability can be less than zero. Secondly, the sum of the probabilities for all values must equal one. This should be intuitively clear for a coin toss: if the odds of getting heads is 70%, then the odds of getting tails must be 30%. We formulize this requirement as

$$\sum_u P(X{=}u) = 1$$

for discrete distributions, and as

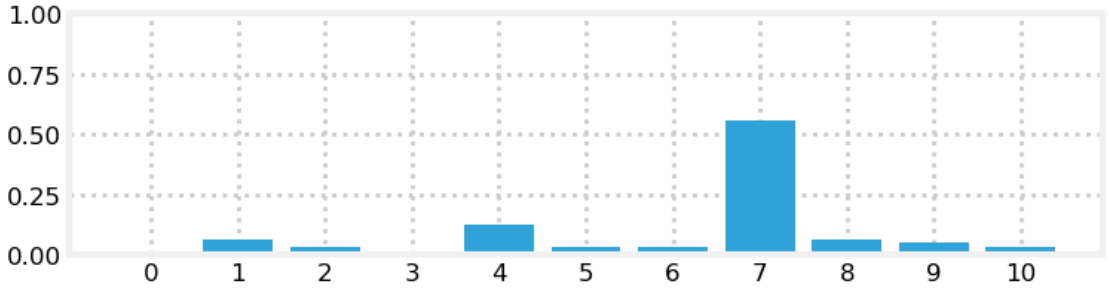
$$\int_u P(X{=}u)\, du = 1$$

for continuous distributions.

In the previous chapter we used probability distributions to estimate the position of a dog in a hallway. For example:

```
In [3]: import numpy as np
        import kf_book.book_plots as book_plots

        belief = np.array([1, 4, 2, 0, 8, 2, 2, 35, 4, 3, 2])
        belief = belief / np.sum(belief)
        with book_plots.figsize(y=2):
            book_plots.bar_plot(belief)
        print('sum = ', np.sum(belief))
```

```
sum =  1.0
```




Each position has a probability between 0 and 1, and the sum of all equals one, so this makes it a probability distribution. Each probability is discrete, so we can more precisely call this a discrete probability distribution. In practice we leave out the terms discrete and continuous unless we have a particular reason to make that distinction.

### 3.3.1 The Mean, Median, and Mode of a Random Variable

Given a set of data we often want to know a representative or average value for that set. There are many measures for this, and the concept is called a *measure of central tendency*. For example we might want to

know the *average* height of the students in a class. We all know how to find the average of a set of data, but let me belabor the point so I can introduce more formal notation and terminology. Another word for average is the *mean*. We compute the mean by summing the values and dividing by the number of values. If the heights of the students in meters is

$$X = \{1.8, 2.0, 1.7, 1.9, 1.6\}$$

we compute the mean as

$$\mu = \frac{1.8 + 2.0 + 1.7 + 1.9 + 1.6}{5} = 1.8$$

It is traditional to use the symbol $\mu$ (mu) to denote the mean.

We can formalize this computation with the equation

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

NumPy provides `numpy.mean()` for computing the mean.

```
In [4]: x = [1.8, 2.0, 1.7, 1.9, 1.6]
        np.mean(x)
```

```
Out[4]: 1.8
```

As a convenience NumPy arrays provide the method `mean()`.

```
In [5]: x = np.array([1.8, 2.0, 1.7, 1.9, 1.6])
        x.mean()
```

```
Out[5]: 1.8
```

The *mode* of a set of numbers is the number that occurs most often. If only one number occurs most often we say it is a *unimodal* set, and if two or more numbers occur the most with equal frequency than the set is *multimodal*. For example the set {1, 2, 2, 2, 3, 4, 4, 4} has modes 2 and 4, which is multimodal, and the set {5, 7, 7, 13} has the mode 7, and so it is unimodal. We will not be computing the mode in this manner in this book, but we do use the concepts of unimodal and multimodal in a more general sense. For example, in the **Discrete Bayes** chapter we talked about our belief in the dog's position as a *multimodal distribution* because we assigned different probabilities to different positions.

Finally, the *median* of a set of numbers is the middle point of the set so that half the values are below the median and half are above the median. Here, above and below is in relation to the set being sorted. If the set contains an even number of values then the two middle numbers are averaged together.

Numpy provides `numpy.median()` to compute the median. As you can see the median of {1.8, 2.0, 1.7, 1.9, 1.6} is 1.8, because 1.8 is the third element of this set after being sorted. In this case the median equals the mean, but that is not generally true.

```
In [6]: np.median(x)
```

```
Out[6]: 1.8
```

## 3.4 Expected Value of a Random Variable

The *expected value* of a random variable is the average value it would have if we took an infinite number of samples of it and then averaged those samples together. Let's say we have $x = [1, 3, 5]$ and each value is equally probable. What value would we *expect* $x$ to have, on average?

It would be the average of 1, 3, and 5, of course, which is 3. That should make sense; we would expect equal numbers of 1, 3, and 5 to occur, so $(1 + 3 + 5)/3 = 3$ is clearly the average of that infinite series of samples. In other words, here the expected value is the *mean* of the sample space.

Now suppose that each value has a different probability of happening. Say 1 has an 80% chance of occurring, 3 has an 15% chance, and 5 has only a 5% chance. In this case we compute the expected value by multiplying each value of $x$ by the percent chance of it occurring, and summing the result. For this case we could compute

$$\mathbb{E}[X] = (1)(0.8) + (3)(0.15) + (5)(0.05) = 1.5$$

Here I have introduced the notation $\mathbb{E}[X]$ for the expected value of $x$. Some texts use $E(x)$. The value 1.5 for $x$ makes intuitive sense because $x$ is far more likely to be 1 than 3 or 5, and 3 is more likely than 5 as well.

We can formalize this by letting $x_i$ be the $i^{th}$ value of $X$, and $p_i$ be the probability of its occurrence. This gives us

$$\mathbb{E}[X] = \sum_{i=1}^{n} p_i x_i$$

A trivial bit of algebra shows that if the probabilities are all equal, the expected value is the same as the mean:

$$\mathbb{E}[X] = \sum_{i=1}^{n} p_i x_i = \frac{1}{n}\sum_{i=1}^{n} x_i = \mu_x$$

If $x$ is continuous we substitute the sum for an integral, like so

$$\mathbb{E}[X] = \int_a^b x f(x)\, dx$$

where $f(x)$ is the probability distribution function of $x$. We won't be using this equation yet, but we will be using it in the next chapter.

We can write a bit of Python to simulate this. Here I take 1,000,000 samples and compute the expected value of the distribution we just computed analytically.

```
In [7]: total = 0
        N = 1000000
        for r in np.random.rand(N):
            if r <= .80: total += 1
            elif r < .95: total += 3
            else: total += 5

        total / N
```

```
Out[7]: 1.501402
```

You can see that the computed value is close to the analytically derived value. It is not exact because getting an exact values requires an infinite sample size.

### 3.4.1 Exercise

What is the expected value of a die role?

### 3.4.2 Solution

Each side is equally likely, so each has a probability of 1/6. Hence

$$\begin{aligned}\mathbb{E}[X] &= 1/6 \times 1 + 1/6 \times 2 + 1/6 \times 3 + 1/6 \times 4 + 1/6 \times 5 + 1/6 \times 6 \\ &= 1/6(1 + 2 + 3 + 4 + 5 + 6) \\ &= 3.5\end{aligned}$$

### 3.4.3 Exercise

Given the uniform continuous distribution

$$f(x) = \frac{1}{b-a}$$

compute the expected value for $a = 0$ and $b = 20$.

### 3.4.4 Solution

$$\begin{aligned}\mathbb{E}[X] &= \int_0^{20} x\frac{1}{20}\,dx \\ &= \left[\frac{x^2}{40}\right]_0^{20} \\ &= 10 - 0 \\ &= 10\end{aligned}$$

### 3.4.5 Variance of a Random Variable

The computation above tells us the average height of the students, but it doesn't tell us everything we might want to know. For example, suppose we have three classes of students, which we label $X$, $Y$, and $Z$, with these heights:

```
In [8]: X = [1.8, 2.0, 1.7, 1.9, 1.6]
        Y = [2.2, 1.5, 2.3, 1.7, 1.3]
        Z = [1.8, 1.8, 1.8, 1.8, 1.8]
```

Using NumPy we see that the mean height of each class is the same.

```
In [9]: print(np.mean(X), np.mean(Y), np.mean(Z))
```

```
1.8 1.8 1.8
```

The mean of each class is 1.8 meters, but notice that there is a much greater amount of variation in the heights in the second class than in the first class, and that there is no variation at all in the third class.

The mean tells us something about the data, but not the whole story. We want to be able to specify how much *variation* there is between the heights of the students. You can imagine a number of reasons for this. Perhaps a school district needs to order 5,000 desks, and they want to be sure they buy sizes that accommodate the range of heights of the students.

Statistics has formalized this concept of measuring variation into the notion of *standard deviation* and *variance*. The equation for computing the variance is

$$VAR(X) = \mathbb{E}[(X - \mu)^2]$$

Ignoring the square for a moment, you can see that the variance is the *expected value* for how much the sample space $X$ varies from the mean $\mu : (X - \mu)$. I will explain the purpose of the squared term later. The formula for the expected value is $\mathbb{E}[X] = \sum_{i=1}^{n} p_i x_i$ so we can substitute that into the equation above to get

$$VAR(X) = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

Let's compute the variance of the three classes to see what values we get and to become familiar with this concept.

The mean of $X$ is 1.8 ($\mu_x = 1.8$) so we compute

$$\begin{aligned} VAR(X) &= \frac{(1.8-1.8)^2 + (2-1.8)^2 + (1.7-1.8)^2 + (1.9-1.8)^2 + (1.6-1.8)^2}{5} \\ &= \frac{0 + 0.04 + 0.01 + 0.01 + 0.04}{5} \\ VAR(X) &= 0.02\,m^2 \end{aligned}$$

NumPy provides the function `var()` to compute the variance:

```
In [10]: print("{:.2f} meters squared".format(np.var(X)))
```

```
0.02 meters squared
```

This is perhaps a bit hard to interpret. Heights are in meters, yet the variance is meters squared. Thus we have a more commonly used measure, the *standard deviation*, which is defined as the square root of the variance:

$$\sigma = \sqrt{VAR(X)} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$$

It is typical to use $\sigma$ for the *standard deviation* and $\sigma^2$ for the *variance*. In most of this book I will be using $\sigma^2$ instead of $VAR(X)$ for the variance; they symbolize the same thing.

For the first class we compute the standard deviation with

$$\begin{aligned} \sigma_x &= \sqrt{\frac{(1.8-1.8)^2 + (2-1.8)^2 + (1.7-1.8)^2 + (1.9-1.8)^2 + (1.6-1.8)^2}{5}} \\ &= \sqrt{\frac{0 + 0.04 + 0.01 + 0.01 + 0.04}{5}} \\ \sigma_x &= 0.1414 \end{aligned}$$

We can verify this computation with the NumPy method `numpy.std()` which computes the standard deviation. 'std' is a common abbreviation for standard deviation.

```
In [11]: print('std {:.4f}'.format(np.std(X)))
         print('var {:.4f}'.format(np.std(X)**2))
```

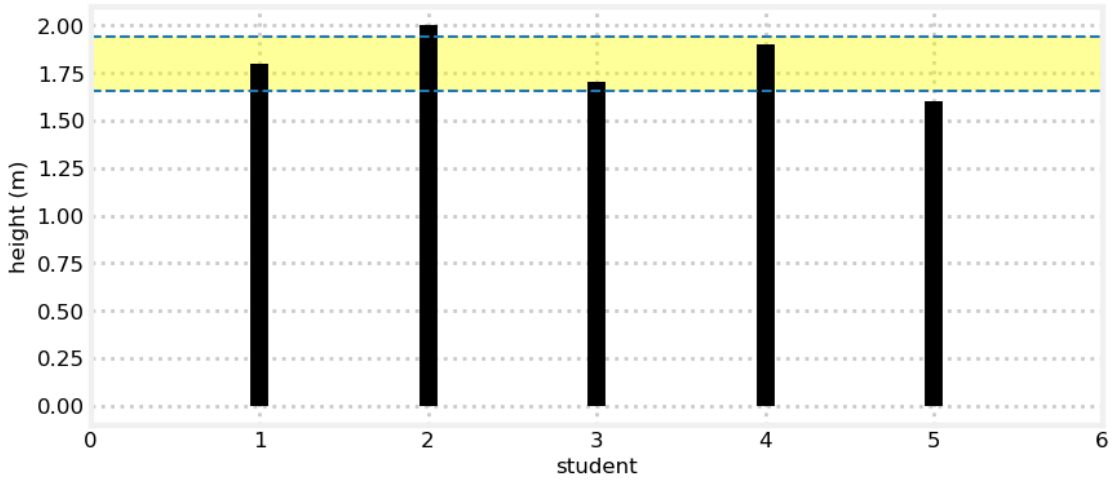
```
std 0.1414
var 0.0200
```

And, of course, $0.1414^2 = 0.02$, which agrees with our earlier computation of the variance.

What does the standard deviation signify? It tells us how much the heights vary amongst themselves. "How much" is not a mathematical term. We will be able to define it much more precisely once we introduce the concept of a Gaussian in the next section. For now I'll say that for many things 68% of all values lie within one standard deviation of the mean. In other words we can conclude that for a random class 68% of the students will have heights between 1.66 (1.8-0.1414) meters and 1.94 (1.8+0.1414) meters.

We can view this in a plot:

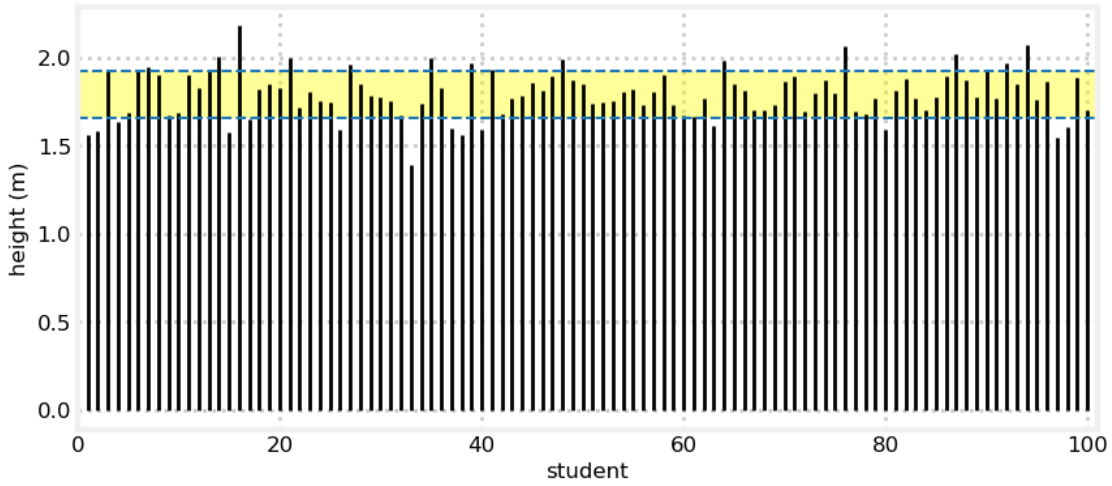
```
In [12]: from kf_book.gaussian_internal import plot_height_std
         import matplotlib.pyplot as plt

         plot_height_std(X)
```




For only 5 students we obviously will not get exactly 68% within one standard deviation. We do see that 3 out of 5 students are within $\pm 1\sigma$, or 60%, which is as close as you can get to 68% with only 5 samples. Let's look at the results for a class with 100 students.

> We write one standard deviation as $1\sigma$, which is pronounced "one standard deviation", not "one sigma". Two standard deviations is $2\sigma$, and so on.

```
In [13]: from numpy.random import randn
         data = 1.8 + randn(100)*.1414
         mean, std = data.mean(), data.std()

         plot_height_std(data, lw=2)
         print('mean = {:.3f}'.format(mean))
         print('std  = {:.3f}'.format(std))
```




```
mean = 1.794
std  = 0.134
```

By eye roughly 68% of the heights lie within $\pm 1\sigma$ of the mean 1.8, but we can verify this with code.

```
In [14]: np.sum((data > mean-std) & (data < mean+std)) / len(data) * 100.

Out[14]: 69.0
```

We'll discuss this in greater depth soon. For now let's compute the standard deviation for

$$Y = [2.2, 1.5, 2.3, 1.7, 1.3]$$

The mean of $Y$ is $\mu = 1.8$ m, so

$$\begin{aligned}\sigma_y &= \sqrt{\frac{(2.2-1.8)^2 + (1.5-1.8)^2 + (2.3-1.8)^2 + (1.7-1.8)^2 + (1.3-1.8)^2}{5}} \\ &= \sqrt{0.152} = 0.39\ m\end{aligned}$$

We will verify that with NumPy with

```
In [15]: print('std of Y is {:.2f} m'.format(np.std(Y)))

std of Y is 0.39 m
```

This corresponds with what we would expect. There is more variation in the heights for $Y$, and the standard deviation is larger.

Finally, let's compute the standard deviation for $Z$. There is no variation in the values, so we would expect the standard deviation to be zero.

$$\begin{aligned}\sigma_z &= \sqrt{\frac{(1.8-1.8)^2 + (1.8-1.8)^2 + (1.8-1.8)^2 + (1.8-1.8)^2 + (1.8-1.8)^2}{5}} \\ &= \sqrt{\frac{0+0+0+0+0}{5}} \\ \sigma_z &= 0.0\ m\end{aligned}$$

```
In [16]: print(np.std(Z))

0.0
```

Before we continue I need to point out that I'm ignoring that on average men are taller than women. In general the height variance of a class that contains only men or women will be smaller than a class with both sexes. This is true for other factors as well. Well nourished children are taller than malnourished children. Scandinavians are taller than Italians. When designing experiments statisticians need to take these factors into account.

I suggested we might be performing this analysis to order desks for a school district. For each age group there are likely to be two different means - one clustered around the mean height of the females, and a second mean clustered around the mean heights of the males. The mean of the entire class will be somewhere between the two. If we bought desks for the mean of all students we are likely to end up with desks that fit neither the males or females in the school!

We will not to consider these issues in this book. Consult any standard probability text if you need to learn techniques to deal with these issues.

### 3.4.6 Why the Square of the Differences

Why are we taking the *square* of the differences for the variance? I could go into a lot of math, but let's look at this in a simple way. Here is a chart of the values of $X$ plotted against the mean for $X = [3, -3, 3, -3]$

```
In [17]: X = [3, -3, 3, -3]
         mean = np.average(X)
         for i in range(len(X)):
             plt.plot([i ,i], [mean, X[i]], color='k')
         plt.axhline(mean)
         plt.xlim(-1, len(X))
         plt.tick_params(axis='x', labelbottom=False)
```


If we didn't take the square of the differences the signs would cancel everything out:

$$\frac{(3-0)+(-3-0)+(3-0)+(-3-0)}{4} = 0$$

This is clearly incorrect, as there is more than 0 variance in the data.

Maybe we can use the absolute value? We can see by inspection that the result is $12/4 = 3$ which is certainly correct — each value varies by 3 from the mean. But what if we have $Y = [6, -2, -3, 1]$? In this case we get $12/4 = 3$. $Y$ is clearly more spread out than $X$, but the computation yields the same variance. If we use the formula using squares we get a variance of 3.5 for $Y$, which reflects its larger variation.

This is not a proof of correctness. Indeed, Carl Friedrich Gauss, the inventor of the technique, recognized that it is somewhat arbitrary. If there are outliers then squaring the difference gives disproportionate weight to that term. For example, let's see what happens if we have:

```
In [18]: X = [1, -1, 1, -2, -1, 2, 1, 2, -1, 1, -1, 2, 1, -2, 100]
         print('Variance of X with outlier    = {:6.2f}'.format(np.var(X)))
         print('Variance of X without outlier = {:6.2f}'.format(np.var(X[:-1])))
```

```
Variance of X with outlier    = 621.45
Variance of X without outlier =   2.03
```

Is this "correct"? You tell me. Without the outlier of 100 we get $\sigma^2 = 2.03$, which accurately reflects how $X$ is varying absent the outlier. The one outlier swamps the variance computation. Do we want to swamp the computation so we know there is an outlier, or robustly incorporate the outlier and still provide an estimate close to the value absent the outlier? Again, you tell me. Obviously it depends on your problem.

I will not continue down this path; if you are interested you might want to look at the work that James Berger has done on this problem, in a field called *Bayesian robustness*, or the excellent publications on *robust statistics* by Peter J. Huber [4]. In this book we will always use variance and standard deviation as defined by Gauss.

The point to gather from this is that these *summary* statistics always tell an incomplete story about our data. In this example variance as defined by Gauss does not tell us we have a single large outlier. However, it is a powerful tool, as we can concisely describe a large data set with a few numbers. If we had 1 billion data points we would not want to inspect plots by eye or look at lists of numbers; summary statistics give us a way to describe the shape of the data in a useful way.

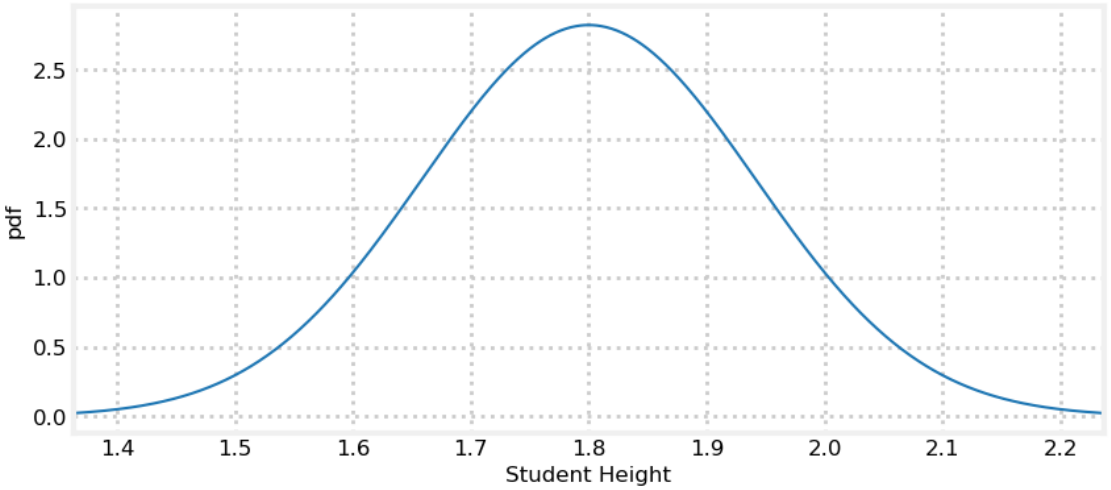
## 3.5 Gaussians

We are now ready to learn about Gaussians. Let's remind ourselves of the motivation for this chapter.

> We desire a unimodal, continuous way to represent probabilities that models how the real world works, and that is computationally efficient to calculate.

Let's look at a graph of a Gaussian distribution to get a sense of what we are talking about.

```
In [19]: from filterpy.stats import plot_gaussian_pdf
         plot_gaussian_pdf(mean=1.8, variance=0.1414**2,
                           xlabel='Student Height', ylabel='pdf');
```




This curve is a *probability density function* or *pdf* for short. It shows the relative likelihood for the random variable to take on a value. We can tell from the chart student is somewhat more likely to have a height near 1.8 m than 1.7 m, and far more likely to have a height of 1.9 m vs 1.4 m. Put another way, many students will have a height near 1.8 m, and very few students will have a height of 1.4 m or 2.2 meters. Finally, notice that the curve is centered over the mean of 1.8 m.

> I explain how to plot Gaussians, and much more, in the Notebook *Computing_and_Plotting_PDFs* in the Supporting_Notebooks folder. You can read it online here [1].

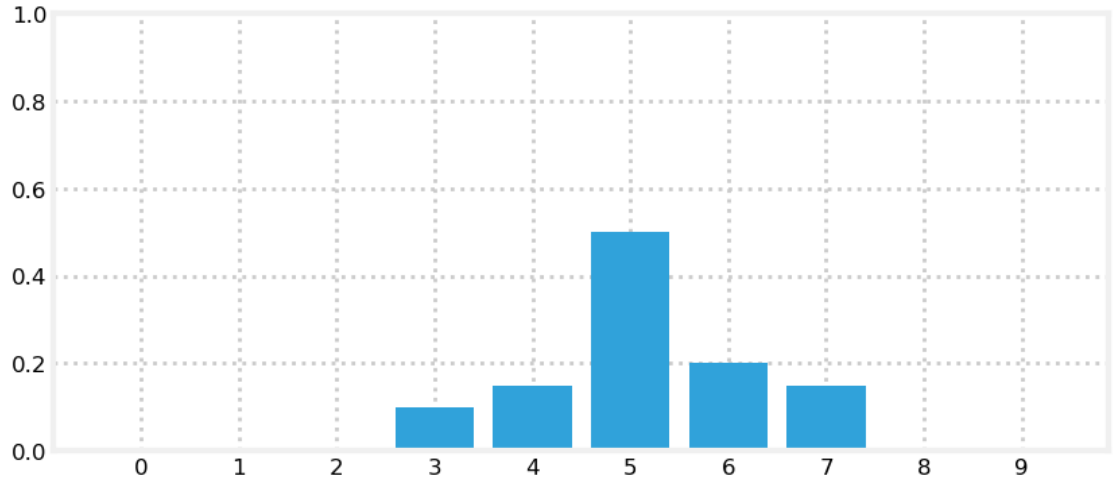
This may be recognizable to you as a 'bell curve'. This curve is ubiquitous because under real world conditions many observations are distributed in such a manner. I will not use the term 'bell curve' to refer to a Gaussian because many probability distributions have a similar bell curve shape. Non-mathematical

sources might not be as precise, so be judicious in what you conclude when you see the term used without definition.

This curve is not unique to heights — a vast amount of natural phenomena exhibits this sort of distribution, including the sensors that we use in filtering problems. As we will see, it also has all the attributes that we are looking for — it represents a unimodal belief or value as a probability, it is continuous, and it is computationally efficient. We will soon discover that it also has other desirable qualities which we may not realize we desire.

To further motivate you, recall the shapes of the probability distributions in the *Discrete Bayes* chapter:

```
In [20]: import kf_book.book_plots as book_plots
         belief = [0., 0., 0., 0.1, 0.15, 0.5, 0.2, .15, 0, 0]
         book_plots.bar_plot(belief)
```

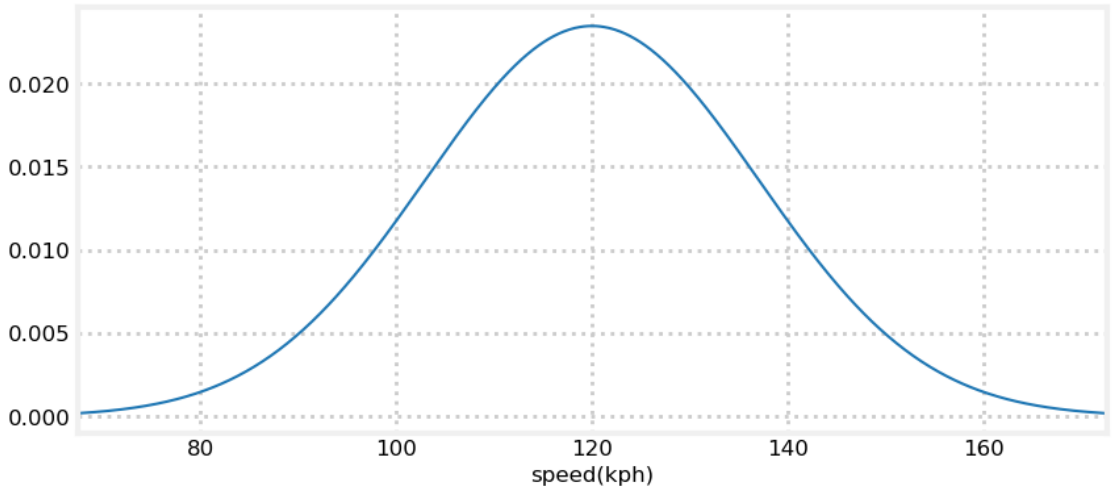



They were not perfect Gaussian curves, but they were similar. We will be using Gaussians to replace the discrete probabilities used in that chapter!

## 3.6 Nomenclature

A bit of nomenclature before we continue - this chart depicts the *probability density* of a *random variable* having any value between $(-\infty..\infty)$. What does that mean? Imagine we take an infinite number of infinitely precise measurements of the speed of automobiles on a section of highway. We could then plot the results by showing the relative number of cars going past at any given speed. If the average was 120 kph, it might look like this:

```
In [21]: plot_gaussian_pdf(mean=120, variance=17**2, xlabel='speed(kph)');
```




The y-axis depicts the *probability density* — the relative amount of cars that are going the speed at the corresponding x-axis. I will explain this further in the next section.

The Gaussian model is imperfect. Though these charts do not show it, the *tails* of the distribution extend out to infinity. *Tails* are the far ends of the curve where the values are the lowest. Of course human heights or automobile speeds cannot be less than zero, let alone $-\infty$ or $\infty$. "The map is not the territory" is a common expression, and it is true for Bayesian filtering and statistics. The Gaussian distribution above models the distribution of the measured automobile speeds, but being a model it is necessarily imperfect. The difference between model and reality will come up again and again in these filters. Gaussians are used in many branches of mathematics, not because they perfectly model reality, but because they are easier to use than any other relatively accurate choice. However, even in this book Gaussians will fail to model reality, forcing us to use computationally expensive alternatives.

You will hear these distributions called *Gaussian distributions* or *normal distributions*. *Gaussian* and *normal* both mean the same thing in this context, and are used interchangeably. I will use both throughout this book as different sources will use either term, and I want you to be used to seeing both. Finally, as in this paragraph, it is typical to shorten the name and talk about a *Gaussian* or *normal* — these are both typical shortcut names for the *Gaussian distribution*.

## 3.7 Gaussian Distributions

Let's explore how Gaussians work. A Gaussian is a *continuous probability distribution* that is completely described with two parameters, the mean ($\mu$) and the variance ($\sigma^2$). It is defined as:

$$f(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right]$$

$\exp[x]$ is notation for $e^x$.

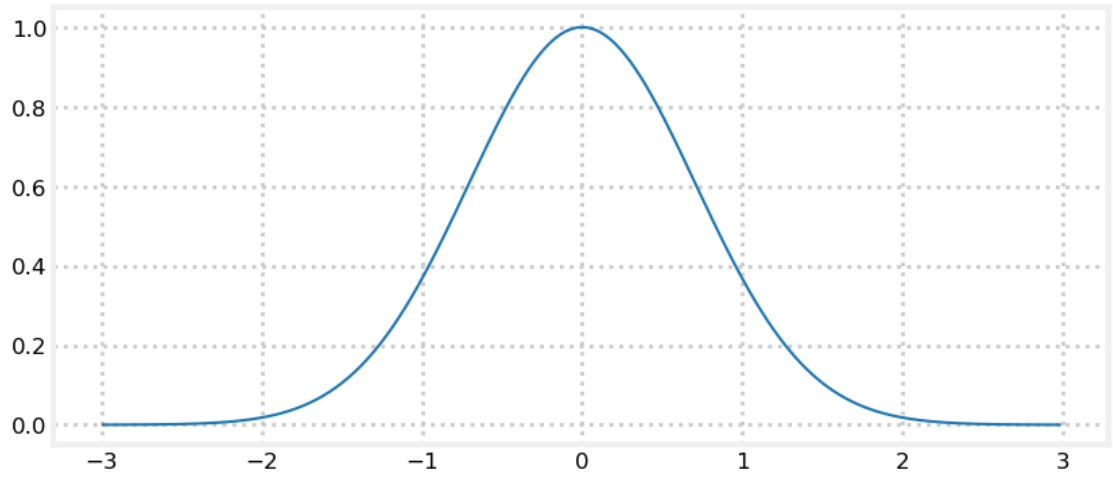
Don't be dissuaded by the equation if you haven't seen it before; you will not need to memorize or manipulate it. The computation of this function is stored in `stats.py` with the function `gaussian(x, mean, var, normed=True)`.

Shorn of the constants, you can see it is a simple exponential:

$$f(x) \propto e^{-x^2}$$

which has the familiar bell curve shape

```
In [22]: x = np.arange(-3, 3, .01)
         plt.plot(x, np.exp(-x**2));
```

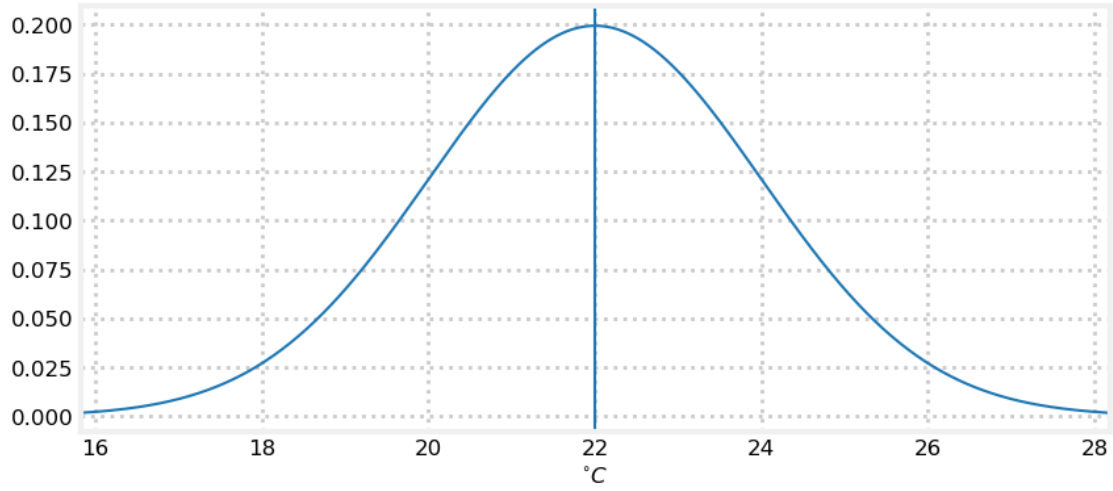



Let's remind ourselves how to look at the code for a function. In a cell, type the function name followed by two question marks and press CTRL+ENTER. This will open a popup window displaying the source. Uncomment the next cell and try it now.

```
In [23]: from filterpy.stats import gaussian
         #gaussian??
```

Let's plot a Gaussian with a mean of 22 ($\mu = 22$), with a variance of 4 ($\sigma^2 = 4$).

```
In [24]: plot_gaussian_pdf(22, 4, mean_line=True, xlabel='$^{\circ}C$');
```




What does this curve *mean*? Assume we have a thermometer which reads 22°C. No thermometer is perfectly accurate, and so we expect that each reading will be slightly off the actual value. However, a theorem called *Central Limit Theorem* states that if we make many measurements that the measurements will be normally distributed. When we look at this chart we can see it is proportional to the probability of the thermometer reading a particular value given the actual temperature of 22°C.

Recall that a Gaussian distribution is *continuous*. Think of an infinitely long straight line - what is the probability that a point you pick randomly is at 2. Clearly 0%, as there is an infinite number of choices to

choose from. The same is true for normal distributions; in the graph above the probability of being *exactly* 2°C is 0% because there are an infinite number of values the reading can take.

What is this curve? It is something we call the *probability density function*. The area under the curve at any region gives you the probability of those values. So, for example, if you compute the area under the curve between 20 and 22 the resulting area will be the probability of the temperature reading being between those two temperatures.

Here is another way to understand it. What is the *density* of a rock, or a sponge? It is a measure of how much mass is compacted into a given space. Rocks are dense, sponges less so. So, if you wanted to know how much a rock weighed but didn't have a scale, you could take its volume and multiply by its density. This would give you its mass. In practice density varies in most objects, so you would integrate the local density across the rock's volume.

$$M = \iiint_R p(x, y, z)\, dV$$

We do the same with *probability density*. If you want to know the temperature being between 20°C and 21°C you would integrate the curve above from 20 to 21. As you know the integral of a curve gives you the area under the curve. Since this is a curve of the probability density, the integral of the density is the probability.

What is the probability of the temperature being exactly 22°C? Intuitively, 0. These are real numbers, and the odds of 22°C vs, say, 22.00000000000017°C is infinitesimal. Mathematically, what would we get if we integrate from 22 to 22? Zero.

Thinking back to the rock, what is the weight of an single point on the rock? An infinitesimal point must have no weight. It makes no sense to ask the weight of a single point, and it makes no sense to ask about the probability of a continuous distribution having a single value. The answer for both is obviously zero.

In practice our sensors do not have infinite precision, so a reading of 22°C implies a range, such as 22 ± 0.1°C, and we can compute the probability of that range by integrating from 21.9 to 22.1.

We can think of this in Bayesian terms or frequentist terms. As a Bayesian, if the thermometer reads exactly 22°C, then our belief is described by the curve - our belief that the actual (system) temperature is near 22°C is very high, and our belief that the actual temperature is near 18 is very low. As a frequentist we would say that if we took 1 billion temperature measurements of a system at exactly 22°C, then a histogram of the measurements would look like this curve.

How do you compute the probability, or area under the curve? You integrate the equation for the Gaussian

$$\int_{x_0}^{x_1} \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(x-\mu)^2/\sigma^2} dx$$

This is called the *cumulative probability distribution*, commonly abbreviated *cdf*.

I wrote `filterpy.stats.norm_cdf` which computes the integral for you. For example, we can compute

```
In [25]: from filterpy.stats import norm_cdf
         print('Cumulative probability of range 21.5 to 22.5 is {:.2f}%'.format(
               norm_cdf((21.5, 22.5), 22,4)*100))
         print('Cumulative probability of range 23.5 to 24.5 is {:.2f}%'.format(
               norm_cdf((23.5, 24.5), 22,4)*100))
```

```
Cumulative probability of range 21.5 to 22.5 is 19.74%
Cumulative probability of range 23.5 to 24.5 is 12.10%
```

The mean ($\mu$) is what it sounds like — the average of all possible probabilities. Because of the symmetric shape of the curve it is also the tallest part of the curve. The thermometer reads 22°C, so that is what we used for the mean.

The notation for a normal distribution for a random variable $X$ is $X \sim \mathcal{N}(\mu, \sigma^2)$ where $\sim$ means *distributed according to*. This means I can express the temperature reading of our thermometer as

$$\text{temp} \sim \mathcal{N}(22, 4)$$

This is an extremely important result. Gaussians allow me to capture an infinite number of possible values with only two numbers! With the values $\mu = 22$ and $\sigma^2 = 4$ I can compute the distribution of measurements over any range.

Some sources use $\mathcal{N}(\mu, \sigma)$ instead of $\mathcal{N}(\mu, \sigma^2)$. Either is fine, they are both conventions. You need to keep in mind which form is being used if you see a term such as $\mathcal{N}(22, 4)$. In this book I always use $\mathcal{N}(\mu, \sigma^2)$, so $\sigma = 2$, $\sigma^2 = 4$ for this example.

## 3.8 The Variance and Belief

Since this is a probability density distribution it is required that the area under the curve always equals one. This should be intuitively clear — the area under the curve represents all possible outcomes, *something* happened, and the probability of *something happening* is one, so the density must sum to one. We can prove this ourselves with a bit of code. (If you are mathematically inclined, integrate the Gaussian equation from $-\infty$ to $\infty$)

```
In [26]: print(norm_cdf((-1e8, 1e8), mu=0, var=4))
```

```
1.0
```

This leads to an important insight. If the variance is small the curve will be narrow. this is because the variance is a measure of *how much* the samples vary from the mean. To keep the area equal to 1, the curve must also be tall. On the other hand if the variance is large the curve will be wide, and thus it will also have to be short to make the area equal to 1.

Let's look at that graphically. We will use the aforementioned `filterpy.stats.gaussian` which can take either a single value or array of values.

```
In [27]: from filterpy.stats import gaussian

         print(gaussian(x=3.0, mean=2.0, var=1))
         print(gaussian(x=[3.0, 2.0], mean=2.0, var=1))
```

```
0.24197072451914337
[0.378 0.622]
```

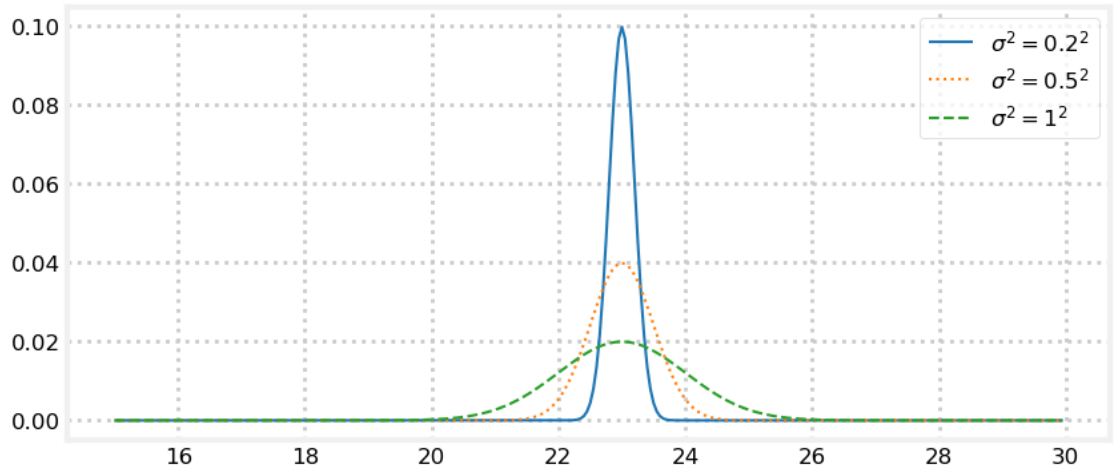
By default `gaussian` normalizes the output, which turns the output back into a probability distribution. Use the argument`normed` to control this.

```
In [28]: print(gaussian(x=[3.0, 2.0], mean=2.0, var=1, normed=False))
```

```
[0.242 0.399]
```

If the Gaussian is not normalized it is called a *Gaussian function* instead of *Gaussian distribution*.

```
In [29]: xs = np.arange(15, 30, 0.05)
         plt.plot(xs, gaussian(xs, 23, 0.2**2), label='$\sigma^2=0.2^2$')
         plt.plot(xs, gaussian(xs, 23, .5**2), label='$\sigma^2=0.5^2$', ls=':')
         plt.plot(xs, gaussian(xs, 23, 1**2), label='$\sigma^2=1^2$', ls='--')
         plt.legend();
```




What is this telling us? The Gaussian with $\sigma^2 = 0.2^2$ is very narrow. It is saying that we believe $x = 23$, and that we are very sure about that: within $\pm 0.2$ std. In contrast, the Gaussian with $\sigma^2 = 1^2$ also believes that $x = 23$, but we are much less sure about that. Our belief that $x = 23$ is lower, and so our belief about the likely possible values for $x$ is spread out — we think it is quite likely that $x = 20$ or $x = 26$, for example. $\sigma^2 = 0.2^2$ has almost completely eliminated 22 or 24 as possible values, whereas $\sigma^2 = 1^2$ considers them nearly as likely as 23.

If we think back to the thermometer, we can consider these three curves as representing the readings from three different thermometers. The curve for $\sigma^2 = 0.2^2$ represents a very accurate thermometer, and curve for $\sigma^2 = 1^2$ represents a fairly inaccurate one. Note the very powerful property the Gaussian distribution affords us — we can entirely represent both the reading and the error of a thermometer with only two numbers — the mean and the variance.

An equivalent formation for a Gaussian is $\mathcal{N}(\mu, 1/\tau)$ where $\mu$ is the *mean* and $\tau$ the *precision*. $1/\tau = \sigma^2$; it is the reciprocal of the variance. While we do not use this formulation in this book, it underscores that the variance is a measure of how precise our data is. A small variance yields large precision — our measurement is very precise. Conversely, a large variance yields low precision — our belief is spread out across a large area. You should become comfortable with thinking about Gaussians in these equivalent forms. In Bayesian terms Gaussians reflect our *belief* about a measurement, they express the *precision* of the measurement, and they express how much *variance* there is in the measurements. These are all different ways of stating the same fact.

I'm getting ahead of myself, but in the next chapters we will use Gaussians to express our belief in things like the estimated position of the object we are tracking, or the accuracy of the sensors we are using.

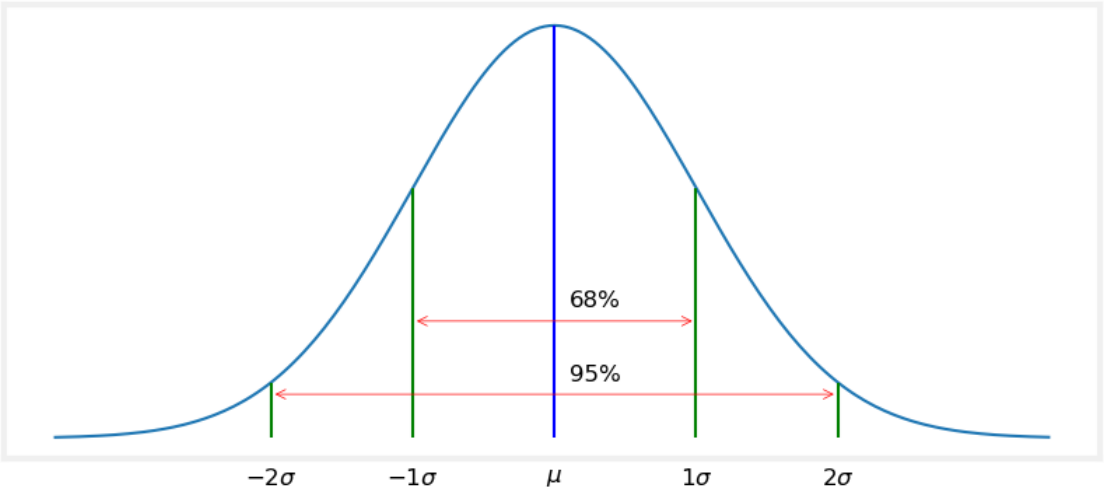
## 3.9 The 68-95-99.7 Rule

It is worth spending a few words on standard deviation now. The standard deviation is a measure of how much the data deviates from the mean. For Gaussian distributions, 68% of all the data falls within one standard deviation ($\pm 1\sigma$) of the mean, 95% falls within two standard deviations ($\pm 2\sigma$), and 99.7% within three ($\pm 3\sigma$). This is often called the 68-95-99.7 rule. If you were told that the average test score in a class was 71 with a standard deviation of 9.4, you could conclude that 95% of the students received a score between 52.2 and 89.8 if the distribution is normal (that is calculated with $71 \pm (2 * 9.4)$).

Finally, these are not arbitrary numbers. If the Gaussian for our position is $\mu = 22$ meters, then the standard deviation also has units meters. Thus $\sigma = 0.2$ implies that 68% of the measurements range from 21.8 to 22.2 meters. Variance is the standard deviation squared, thus $\sigma^2 = .04$ meters$^2$. As you saw in the last section, writing $\sigma^2 = 0.2^2$ can make this somewhat more meaningful, since the 0.2 is in the same units as the data.

The following graph depicts the relationship between the standard deviation and the normal distribution.

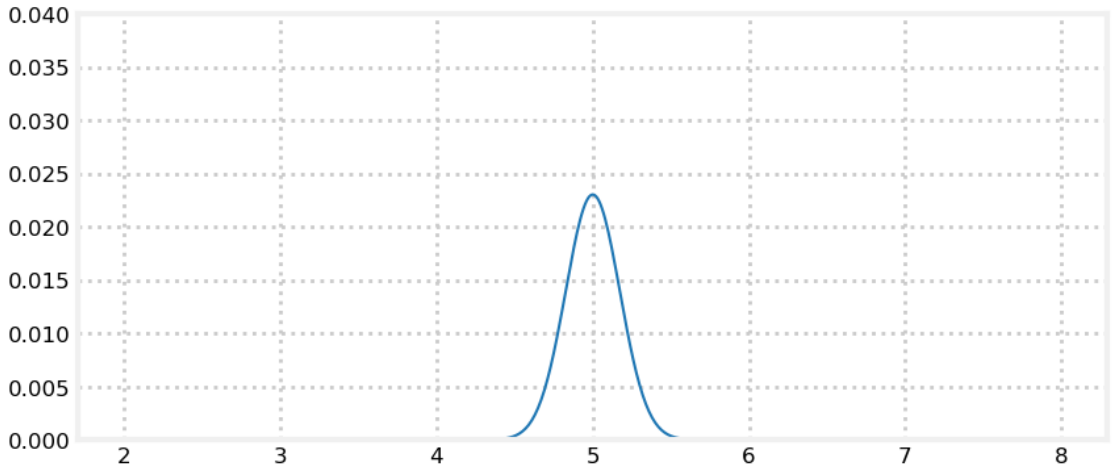
```
In [30]: from kf_book.gaussian_internal import display_stddev_plot
         display_stddev_plot()
```




## 3.10 Interactive Gaussians

For those that are reading this in a Jupyter Notebook, here is an interactive version of the Gaussian plots. Use the sliders to modify $\mu$ and $\sigma^2$. Adjusting $\mu$ will move the graph to the left and right because you are adjusting the mean, and adjusting $\sigma^2$ will make the bell curve thicker and thinner.

```
In [31]: import math
         from ipywidgets import interact, FloatSlider

         def plt_g(mu,variance):
             plt.figure()
             xs = np.arange(2, 8, 0.01)
             ys = gaussian(xs, mu, variance)
             plt.plot(xs, ys)
             plt.ylim(0, 0.04)

         interact(plt_g, mu=FloatSlider(value=5, min=3, max=7),
                  variance=FloatSlider(value = .03, min=.01, max=1.));
```




Finally, if you are reading this online, here is an animation of a Gaussian. First, the mean is shifted to the right. Then the mean is centered at $\mu = 5$ and the variance is modified.

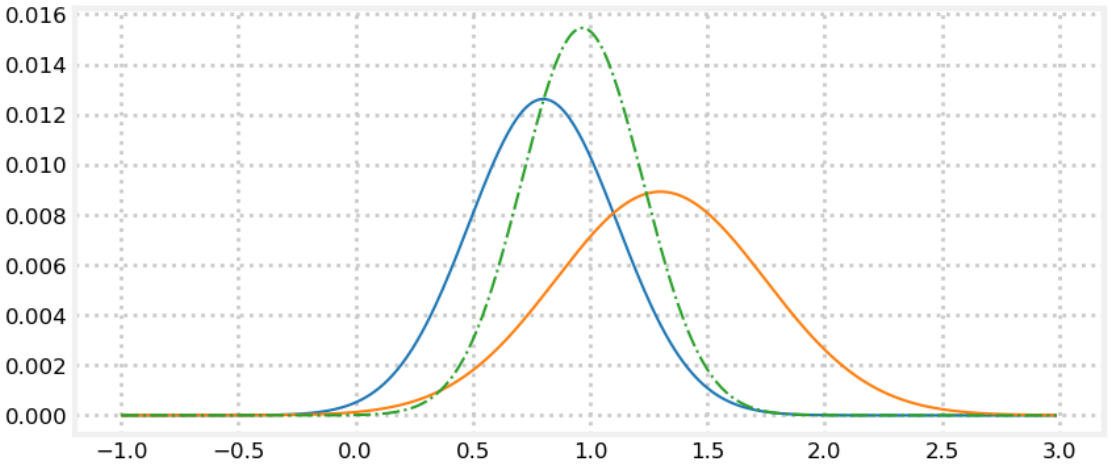
## 3.11 Computational Properties of Gaussians

The discrete Bayes filter works by multiplying and adding arbitrary probability distributions. The Kalman filter uses Gaussians instead of arbitrary distributions, but the rest of the algorithm remains the same. This means we will need to multiply and add Gaussians.

A remarkable property of Gaussian distributions is that the sum of two independent Gaussians is another Gaussian! The product is not Gaussian, but proportional to a Gaussian. There we can say that the result of multipying two Gaussian distributions is a Gaussian function (recall function in this context means that the property that the values sum to one is not guaranteed).

Before we do the math, let's test this visually.

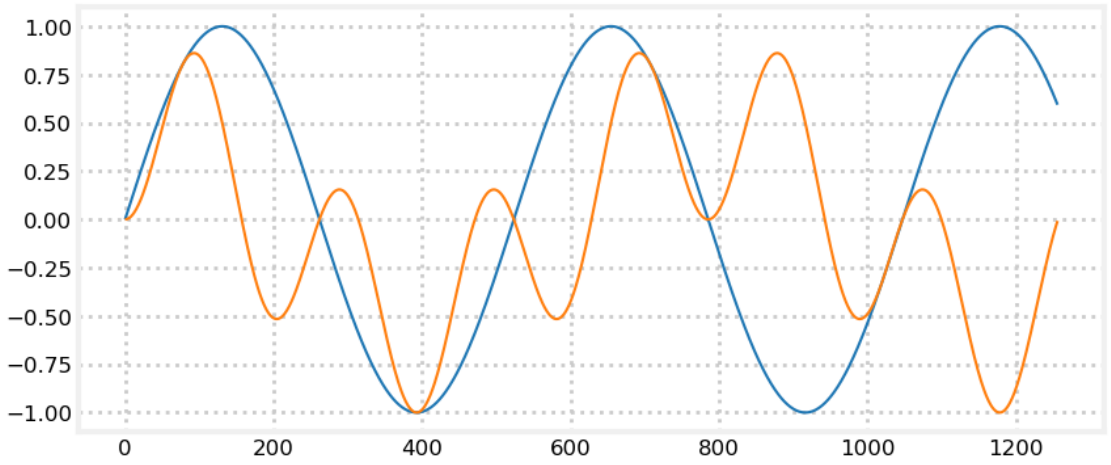
```
In [32]: x = np.arange(-1, 3, 0.01)
         g1 = gaussian(x, mean=0.8, var=.1)
         g2 = gaussian(x, mean=1.3, var=.2)
         plt.plot(x, g1, x, g2)

         g = g1 * g2  # element-wise multiplication
         g = g / sum(g)  # normalize
         plt.plot(x, g, ls='-.');
```




Here I created two Gaussians, g1=$\mathcal{N}(0.8, 0.1)$ and g2=$\mathcal{N}(1.3, 0.2)$ and plotted them. Then I multiplied them together and normalized the result. As you can see the result *looks* like a Gaussian distribution.

Gaussians are nonlinear functions. Typically, if you multiply a nonlinear equations you end up with a different type of function. For example, the shape of multiplying two sins is very different from `sin(x)`.

```
In [33]: x = np.arange(0, 4*np.pi, 0.01)
         plt.plot(np.sin(1.2*x))
         plt.plot(np.sin(1.2*x) * np.sin(2*x));
```




But the result of multiplying two Gaussians distributions is a Gaussian function. This is a key reason why Kalman filters are computationally feasible. Said another way, Kalman filters use Gaussians *because* they are computationally nice.

The product of two independent Gaussians is given by:

$$\mu = \frac{\sigma_1^2\mu_2 + \sigma_2^2\mu_1}{\sigma_1^2 + \sigma_2^2}$$

$$\sigma^2 = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2 + \sigma_2^2}$$

The sum of two Gaussians is given by

$$\mu = \mu_1 + \mu_2$$
$$\sigma^2 = \sigma_1^2 + \sigma_2^2$$

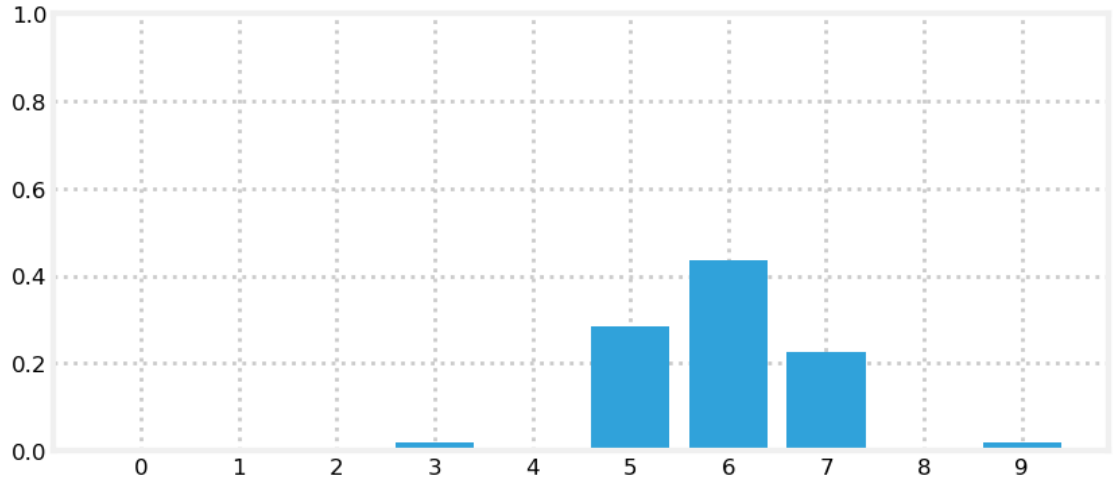
At the end of the chapter I derive these equations. However, understanding the deriviation is not very important.

## 3.12 Putting it all Together

Now we are ready to talk about how Gaussians can be used in filtering. In the next chapter we will implement a filter using Gaussins. Here I will explain why we would want to use Gaussians.

In the previous chapter we represented probability distributions with an array. We performed the update computation by computing the element-wise product of that distribution with another distribution representing the likelihood of the measurement at each point, like so:

```
In [34]: def normalize(p):
             return p / sum(p)

         def update(likelihood, prior):
             return normalize(likelihood * prior)

         prior =      normalize(np.array([4, 2, 0, 7, 2, 12, 35, 20, 3, 2]))
         likelihood = normalize(np.array([3, 4, 1, 4, 2, 38, 20, 18, 1, 16]))
         posterior = update(likelihood, prior)
         book_plots.bar_plot(posterior)
```



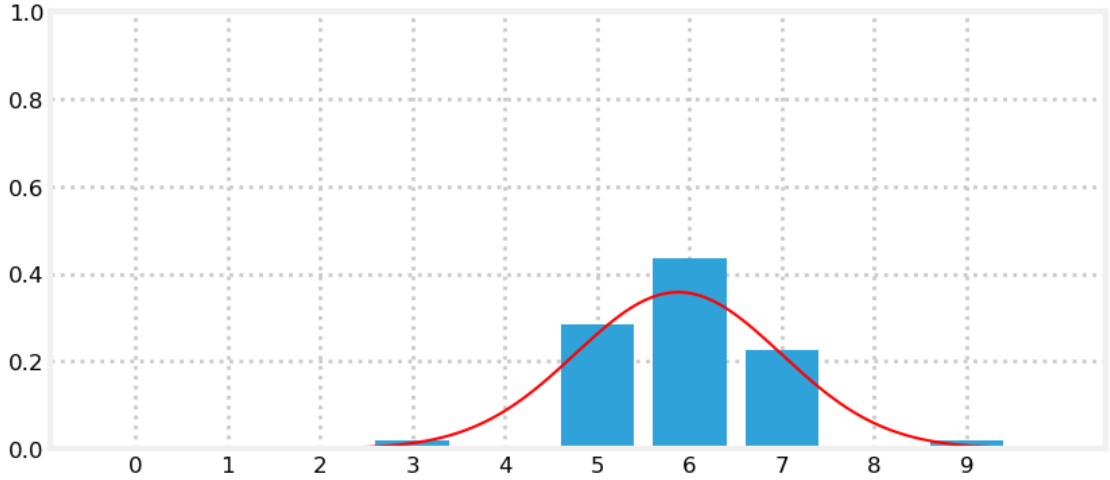

In other words, we have to compute 10 multiplications to get this result. For a real filter with large arrays in multiple dimensions we'd require billions of multiplications, and vast amounts of memory.

But this distribution looks like a Gaussian. What if we use a Gaussian instead of an array? I'll compute the mean and variance of the posterior and plot it against the bar chart.

```
In [35]: xs = np.arange(0, 10, .01)

         def mean_var(p):
             x = np.arange(len(p))
```

```
        mean = np.sum(p * x,dtype=float)
        var = np.sum((x - mean)**2 * p)
        return mean, var

    mean, var = mean_var(posterior)
    book_plots.bar_plot(posterior)
    plt.plot(xs, gaussian(xs, mean, var, normed=False), c='r');
    print('mean: %.2f' % mean, 'var: %.2f' % var)
```

```
mean: 5.88 var: 1.24
```




This is impressive. We can describe an entire distribution of numbers with only two numbers. Perhaps this example is not persuasive, given there are only 10 numbers in the distribution. But a real problem could have millions of numbers, yet still only require two numbers to describe it.

Next, recall that our filter implements the update function with

```
def update(likelihood, prior):
    return normalize(likelihood * prior)
```

If the arrays contain a million elements, that is one million multiplications. However, if we replace the arrays with a Gaussian then we would perform that calculation with

$$\mu = \frac{\sigma_1^2\mu_2 + \sigma_2^2\mu_1}{\sigma_1^2 + \sigma_2^2}$$

$$\sigma^2 = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2 + \sigma_2^2}$$

which is three multiplications and two divisions.

### 3.12.1 Bayes Theorem

In the last chapter we developed an algorithm by reasoning about the information we have at each moment, which we expressed as discrete probability distributions. In the process we discovered *Bayes' Theorem*. Bayes theorem tells us how to compute the probability of an event given prior information.

We implemented the `update()` function with this probability calculation:

$$\texttt{posterior} = \frac{\texttt{likelihood} \times \texttt{prior}}{\texttt{normalization}}$$

It turns out that this is Bayes' theorem. In a second I will develop the mathematics, but in many ways that obscures the simple idea expressed in this equation. We read this as:

$$updated\,knowledge = \|likelihood\,of\,new\,knowledge \times prior\,knowledge\|$$

where $\|\cdot\|$ expresses normalizing the term.

We came to this with simple reasoning about a dog walking down a hallway. Yet, as we will see, the same equation applies to a universe of filtering problems. We will use this equation in every subsequent chapter.

To review, the *prior* is the probability of something happening before we include the probability of the measurement (the *likelihood*) and the *posterior* is the probability we compute after incorporating the information from the measurement.

Bayes theorem is

$$P(A \mid B) = \frac{P(B \mid A)\,P(A)}{P(B)}$$

$P(A \mid B)$ is called a *conditional probability*. That is, it represents the probability of $A$ happening *if* $B$ happened. For example, it is more likely to rain today compared to a typical day if it also rained yesterday because rain systems usually last more than one day. We'd write the probability of it raining today given that it rained yesterday as $P(\text{rain today} \mid \text{rain yesterday})$.

I've glossed over an important point. In our code above we are not working with single probabilities, but an array of probabilities - a *probability distribution*. The equation I just gave for Bayes uses probabilities, not probability distributions. However, it is equally valid with probability distributions. We use a lower case $p$ for probability distributions

$$p(A \mid B) = \frac{p(B \mid A)\,p(A)}{p(B)}$$

In the equation above $B$ is the *evidence*, $p(A)$ is the *prior*, $p(B \mid A)$ is the *likelihood*, and $p(A \mid B)$ is the *posterior*. By substituting the mathematical terms with the corresponding words you can see that Bayes theorem matches our update equation. Let's rewrite the equation in terms of our problem. We will use $x_i$ for the position at $i$, and $z$ for the measurement. Hence, we want to know $P(x_i \mid z)$, that is, the probability of the dog being at $x_i$ given the measurement $z$.

So, let's plug that into the equation and solve it.

$$p(x_i \mid z) = \frac{p(z \mid x_i)p(x_i)}{p(z)}$$

That looks ugly, but it is actually quite simple. Let's figure out what each term on the right means. First is $p(z \mid x_i)$. This is the likelihood, or the probability for the measurement at every cell $x_i$. $p(x_i)$ is the *prior* - our belief before incorporating the measurements. We multiply those together. This is just the unnormalized multiplication in the `update()` function:

```
def update(likelihood, prior):
    posterior = prior * likelihood    # p(z|x) * p(x)
    return normalize(posterior)
```

The last term to consider is the denominator $p(z)$. This is the probability of getting the measurement $z$ without taking the location into account. It is often called the *evidence*. We compute that by taking the sum of $x$, or `sum(belief)` in the code. That is how we compute the normalization! So, the `update()` function is doing nothing more than computing Bayes' theorem.

The literature often gives you these equations in the form of integrals. After all, an integral is just a sum over a continuous function. So, you might see Bayes' theorem written as

$$p(A \mid B) = \frac{p(B \mid A)\, p(A)}{\int p(B \mid A_j) p(A_j)\, \mathtt{d}A_j}.$$

This denominator is usually impossible to solve analytically; when it can be solved the math is fiendishly difficult. A recent opinion piecefor the Royal Statistical Society called it a "dog's breakfast" [8]. Filtering textbooks that take a Bayesian approach are filled with integral laden equations with no analytic solution. Do not be cowed by these equations, as we trivially handled this integral by normalizing our posterior. We will learn more techniques to handle this in the **Particle Filters** chapter. Until then, recognize that in practice it is just a normalization term over which we can sum. What I'm trying to say is that when you are faced with a page of integrals, just think of them as sums, and relate them back to this chapter, and often the difficulties will fade. Ask yourself "why are we summing these values", and "why am I dividing by this term". Surprisingly often the answer is readily apparent. Surprisingly often the author neglects to mention this interpretation.

It's probable that the strength of Bayes' theorem is not yet fully apparent to you. We want to compute $p(x_i \mid Z)$. That is, at step i, what is our probable state given a measurement. That's an extraordinarily difficult problem in general. Bayes' Theorem is general. We may want to know the probability that we have cancer given the results of a cancer test, or the probability of rain given various sensor readings. Stated like that the problems seem unsolvable.

But Bayes' Theorem lets us compute this by using the inverse $p(Z \mid x_i)$, which is often straightforward to compute

$$p(x_i \mid Z) \propto p(Z \mid x_i)\, p(x_i)$$

That is, to compute how likely it is to rain given specific sensor readings we only have to compute the likelihood of the sensor readings given that it is raining! That's a ***much*** easier problem! Well, weather prediction is still a difficult problem, but Bayes makes it tractable.

Likewise, as you saw in the Discrete Bayes chapter, we computed the likelihood that Simon was in any given part of the hallway by computing how likely a sensor reading is given that Simon is at position `x`. A hard problem becomes easy.

### 3.12.2 Total Probability Theorem

We now know the formal mathematics behind the `update()` function; what about the `predict()` function? `predict()` implements the *total probability theorem*. Let's recall what `predict()` computed. It computed the probability of being at any given position given the probability of all the possible movement events. Let's express that as an equation. The probability of being at any position $i$ at time $t$ can be written as $P(X_i^t)$. We computed that as the sum of the prior at time $t-1$ $P(X_j^{t-1})$ multiplied by the probability of moving from cell $x_j$ to $x_i$. That is

$$P(X_i^t) = \sum_j P(X_j^{t-1}) P(x_i|x_j)$$

That equation is called the *total probability theorem*. Quoting from Wikipedia [6] "It expresses the total probability of an outcome which can be realized via several distinct events". I could have given you that equation and implemented `predict()`, but your chances of understanding why the equation works would be slim. As a reminder, here is the code that computes this equation

```
for i in range(N):
    for k in range (kN):
        index = (i + (width-k) - offset) % N
        result[i] += prob_dist[index] * kernel[k]
```

## 3.13 Computing Probabilities with scipy.stats

In this chapter I used code from FilterPy to compute and plot Gaussians. I did that to give you a chance to look at the code and see how these functions are implemented. However, Python comes with "batteries included" as the saying goes, and it comes with a wide range of statistics functions in the module `scipy.stats`. So let's walk through how to use scipy.stats to compute statistics and probabilities.

The `scipy.stats` module contains a number of objects which you can use to compute attributes of various probability distributions. The full documentation for this module is here: http://docs.scipy.org/doc/scipy/reference/stats.html. We will focus on the norm variable, which implements the normal distribution. Let's look at some code that uses `scipy.stats.norm` to compute a Gaussian, and compare its value to the value returned by the `gaussian()` function from FilterPy.

```
In [36]: from scipy.stats import norm
         import filterpy.stats
         print(norm(2, 3).pdf(1.5))
         print(filterpy.stats.gaussian(x=1.5, mean=2, var=3*3))
```

```
0.13114657203397997
0.13114657203397995
```

The call `norm(2, 3)` creates what scipy calls a 'frozen' distribution - it creates and returns an object with a mean of 2 and a standard deviation of 3. You can then use this object multiple times to get the probability density of various values, like so:

```
In [37]: n23 = norm(2, 3)
         print('pdf of 1.5 is       %.4f' % n23.pdf(1.5))
         print('pdf of 2.5 is also  %.4f' % n23.pdf(2.5))
         print('pdf of 2 is         %.4f' % n23.pdf(2))
```

```
pdf of 1.5 is       0.1311
pdf of 2.5 is also  0.1311
pdf of 2 is         0.1330
```

The documentation for scipy.stats.norm [2] lists many other functions. For example, we can generate $n$ samples from the distribution with the `rvs()` function.

```
In [38]: np.set_printoptions(precision=3, linewidth=50)
         print(n23.rvs(size=15))
```

```
[ 3.727  2.153  3.402 -0.984  1.832 -2.181  3.088
  7.797  3.35  -0.951  2.976  1.314  4.391  2.932
  0.143]
```

We can get the *cumulative distribution function (CDF)*, which is the probability that a randomly drawn value from the distribution is less than or equal to $x$.

```
In [39]: # probability that a random value is less than the mean 2
         print(n23.cdf(2))
```

```
0.5
```

We can get various properties of the distribution:

```
In [40]: print('variance is', n23.var())
         print('standard deviation is', n23.std())
         print('mean is', n23.mean())

variance is 9.0
standard deviation is 3.0
mean is 2.0
```

## 3.14 Limitations of Using Gaussians to Model the World

Earlier I mentioned the *central limit theorem*, which states that under certain conditions the arithmetic sum of any independent random variable will be normally distributed, regardless of how the random variables are distributed. This is important to us because nature is full of distributions which are not normal, but when we apply the central limit theorem over large populations we end up with normal distributions.

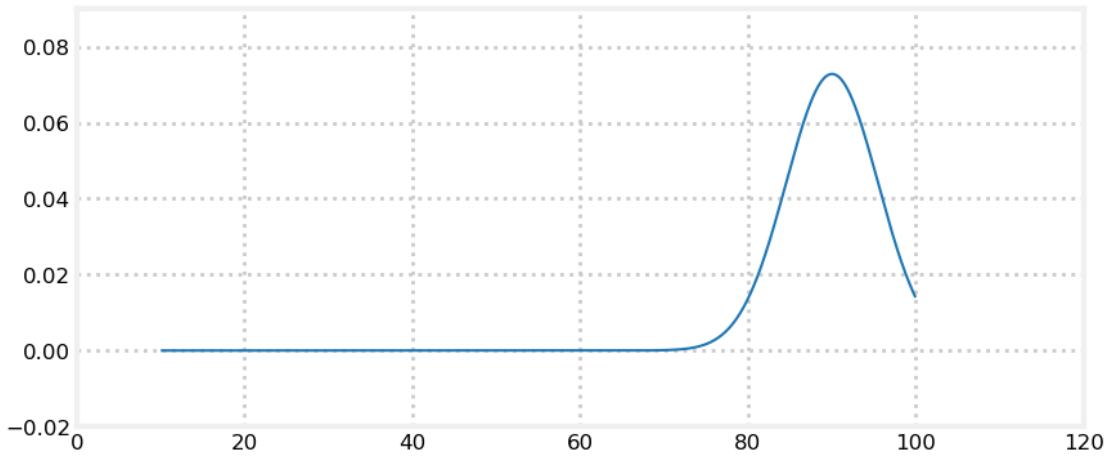
However, a key part of the proof is "under certain conditions". These conditions often do not hold for the physical world. For example, a kitchen scale cannot read below zero, but if we represent the measurement error as a Gaussian the left side of the curve extends to negative infinity, implying a very small chance of giving a negative reading.

This is a broad topic which I will not treat exhaustively.

Let's consider a trivial example. We think of things like test scores as being normally distributed. If you have ever had a professor "grade on a curve" you have been subject to this assumption. But of course test scores cannot follow a normal distribution. This is because the distribution assigns a nonzero probability distribution for *any* value, no matter how far from the mean. So, for example, say your mean is 90 and the standard deviation is 13. The normal distribution assumes that there is a large chance of somebody getting a 90, and a small chance of somebody getting a 40. However, it also implies that there is a tiny chance of somebody getting a grade of -10, or 150. It assigns an extremely small chance of getting a score of $-10^{300}$ or $10^{32986}$. The tails of a Gaussian distribution are infinitely long.

But for a test we know this is not true. Ignoring extra credit, you cannot get less than 0, or more than 100. Let's plot this range of values using a normal distribution to see how poorly this represents real test scores distributions.

```
In [41]: xs = np.arange(10, 100, 0.05)
         ys = [gaussian(x, 90, 30) for x in xs]
         plt.plot(xs, ys, label='var=0.2')
         plt.xlim(0, 120)
         plt.ylim(-0.02, 0.09);
```



The area under the curve cannot equal 1, so it is not a probability distribution. What actually happens is that more students than predicted by a normal distribution get scores nearer the upper end of the range (for example), and that tail becomes "fat". Also, the test is probably not able to perfectly distinguish minute differences in skill in the students, so the distribution to the left of the mean is also probably a bit bunched up in places.

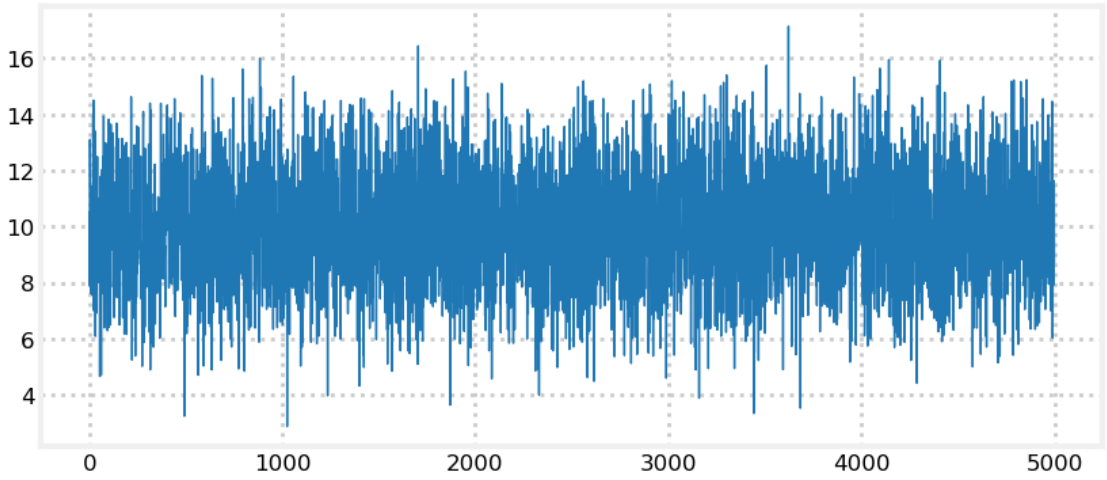
Sensors measure the world. The errors in a sensor's measurements are rarely truly Gaussian. It is far too early to be talking about the difficulties that this presents to the Kalman filter designer. It is worth keeping in the back of your mind the fact that the Kalman filter math is based on an idealized model of the world. For now I will present a bit of code that I will be using later in the book to form distributions to simulate various processes and sensors. This distribution is called the *Student's t-distribution*.

Let's say I want to model a sensor that has some white noise in the output. For simplicity, let's say the signal is a constant 10, and the standard deviation of the noise is 2. We can use the function `numpy.random.randn()` to get a random number with a mean of 0 and a standard deviation of 1. I can simulate this with:

```
In [42]: from numpy.random import randn
         def sense():
             return 10 + randn()*2
```

Let's plot that signal and see what it looks like.

```
In [43]: zs = [sense() for i in range(5000)]
         plt.plot(zs, lw=1);
```



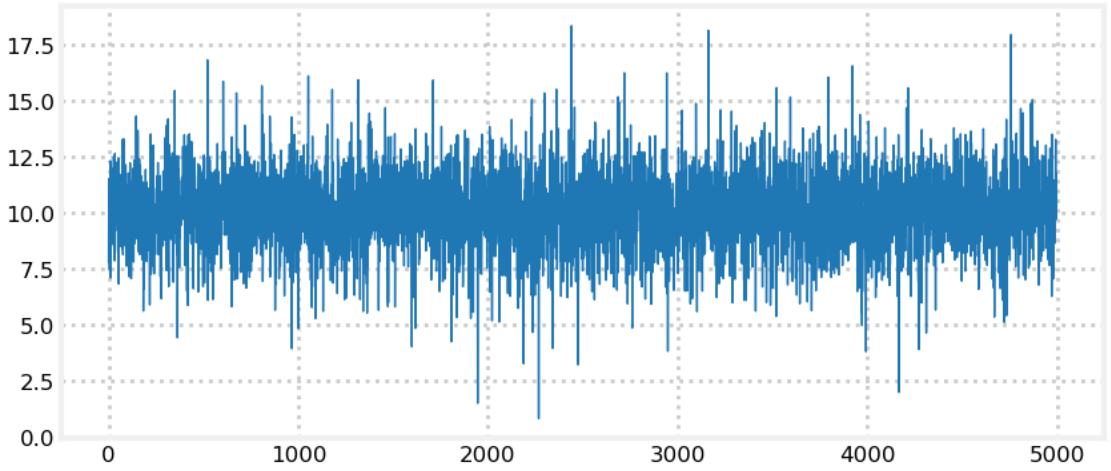

That looks like what I would expect. The signal is centered around 10. A standard deviation of 2 means that 68% of the measurements will be within ± 2 of 10, and 99% will be within ± 6 of 10, and that looks like what is happening.

Now let's look at distribution generated with the Student's $t$-distribution. I will not go into the math, but just give you the source code for it and then plot a distribution using it.

```
In [44]: import random
         import math

         def rand_student_t(df, mu=0, std=1):
             """return random number distributed by Student's t
```

```
        distribution with `df` degrees of freedom with the
        specified mean and standard deviation.
        """
        x = random.gauss(0, std)
        y = 2.0*random.gammavariate(0.5*df, 2.0)
        return x / (math.sqrt(y / df)) + mu
```

```
In [45]: def sense_t():
             return 10 + rand_student_t(7)*2

         zs = [sense_t() for i in range(5000)]
         plt.plot(zs, lw=1);
```




We can see from the plot that while the output is similar to the normal distribution there are outliers that go far more than 3 standard deviations from the mean (7 to 13).

It is unlikely that the Student's $t$-distribution is an accurate model of how your sensor (say, a GPS or Doppler) performs, and this is not a book on how to model physical systems. However, it does produce reasonable data to test your filter's performance when presented with real world noise. We will be using distributions like these throughout the rest of the book in our simulations and tests.

This is not an idle concern. The Kalman filter equations assume the noise is normally distributed, and perform sub-optimally if this is not true. Designers for mission critical filters, such as the filters on spacecraft, need to master a lot of theory and empirical knowledge about the performance of the sensors on their spacecraft. For example, a presentation I saw on a NASA mission stated that while theory states that they should use 3 standard deviations to distinguish noise from valid measurements in practice they had to use 5 to 6 standard deviations. This was something they determined by experiments.

The code for rand_student_t is included in `filterpy.stats`. You may use it with

```
from filterpy.stats import rand_student_t
```

While I'll not cover it here, statistics has defined ways of describing the shape of a probability distribution by how it varies from an exponential distribution. The normal distribution is shaped symmetrically around the mean - like a bell curve. However, a probability distribution can be asymmetrical around the mean. The measure of this is called *skew*. The tails can be shortened, fatter, thinner, or otherwise shaped differently from an exponential distribution. The measure of this is called *kurtosis*. the `scipy.stats` module contains the function `describe` which computes these statistics, among others.

```
In [46]: import scipy
         scipy.stats.describe(zs)
```

```
Out[46]: DescribeResult(nobs=5000, minmax=(0.7895640739177043, 18.33535370483629), mean=10.047232700025
```

Let's examine two normal populations, one small, one large:

```
In [47]: print(scipy.stats.describe(np.random.randn(10)))
         print()
         print(scipy.stats.describe(np.random.randn(300000)))
```

```
DescribeResult(nobs=10, minmax=(-1.7032677594040733, 1.986068539959645), mean=0.051555632450208375, var

DescribeResult(nobs=300000, minmax=(-4.430736129177644, 4.377722305677003), mean=0.0038782068765823375,
```

The small sample has very non-zero skew and kurtosis because the small number of samples is not well distributed around the mean of 0. You can see this also by comparing the computed mean and variance with the theoretical mean of 0 and variance 1. In comparison the large sample's mean and variance are very close to the theoretical values, and both the skew and kurtosis are near zero.

## 3.15 Product of Gaussians (Optional)

It is not important to read this section. Here I derive the equations for the product of two Gaussians.

You can find this result by multiplying the equation for two Gaussians together and combining terms. The algebra gets messy. I will derive it using Bayes theorem. We can state the problem as: let the prior be $N(\bar{\mu}, \bar{\sigma}^2)$, and measurement be $z \propto N(z, \sigma_z^2)$. What is the posterior x given the measurement z?

Write the posterior as $p(x \mid z)$. Now we can use Bayes Theorem to state

$$p(x \mid z) = \frac{p(z \mid x)p(x)}{p(z)}$$

$p(z)$ is a normalizing constant, so we can create a proportinality

$$p(x \mid z) \propto p(z|x)p(x)$$

Now we subtitute in the equations for the Gaussians, which are

$$p(z \mid x) = \frac{1}{\sqrt{2\pi\sigma_z^2}} \exp\Big[-\frac{(z-x)^2}{2\sigma_z^2}\Big]$$

$$p(x) = \frac{1}{\sqrt{2\pi\bar{\sigma}^2}} \exp\Big[-\frac{(x-\bar{\mu})^2}{2\bar{\sigma}^2}\Big]$$

We can drop the leading terms, as they are constants, giving us

$$\begin{aligned} p(x \mid z) &\propto \exp\Big[-\frac{(z-x)^2}{2\sigma_z^2}\Big] \exp\Big[-\frac{(x-\bar{\mu})^2}{2\bar{\sigma}^2}\Big] \\ &\propto \exp\Big[-\frac{(z-x)^2}{2\sigma_z^2} - \frac{(x-\bar{\mu})^2}{2\bar{\sigma}^2}\Big] \\ &\propto \exp\Big[-\frac{1}{2\sigma_z^2\bar{\sigma}^2}[\bar{\sigma}^2(z-x)^2 + \sigma_z^2(x-\bar{\mu})^2]\Big] \end{aligned}$$

Now we multiply out the squared terms and group in terms of the posterior $x$.

$$\begin{aligned} p(x \mid z) &\propto \exp\Big[-\frac{1}{2\sigma_z^2\bar{\sigma}^2}[\bar{\sigma}^2(z^2 - 2xz + x^2) + \sigma_z^2(x^2 - 2x\bar{\mu} + \bar{\mu}^2)]\Big] \\ &\propto \exp\Big[-\frac{1}{2\sigma_z^2\bar{\sigma}^2}[x^2(\bar{\sigma}^2 + \sigma_z^2) - 2x(\sigma_z^2\bar{\mu} + \bar{\sigma}^2 z) + (\bar{\sigma}^2 z^2 + \sigma_z^2\bar{\mu}^2)]\Big] \end{aligned}$$

The last parentheses do not contain the posterior $x$, so it can be treated as a constant and discarded.

$$p(x \mid z) \propto \exp\Big[-\frac{1}{2}\frac{x^2(\bar{\sigma}^2+\sigma_z^2) - 2x(\sigma_z^2\bar{\mu}+\bar{\sigma}^2 z)}{\sigma_z^2\bar{\sigma}^2}\Big]$$

Divide numerator and denominator by $\bar{\sigma}^2+\sigma_z^2$ to get

$$p(x \mid z) \propto \exp\Big[-\frac{1}{2}\frac{x^2 - 2x(\frac{\sigma_z^2\bar{\mu}+\bar{\sigma}^2 z}{\bar{\sigma}^2+\sigma_z^2})}{\frac{\sigma_z^2\bar{\sigma}^2}{\bar{\sigma}^2+\sigma_z^2}}\Big]$$

Proportionality allows us create or delete constants at will, so we can factor this into

$$p(x \mid z) \propto \exp\Big[-\frac{1}{2}\frac{(x - \frac{\sigma_z^2\bar{\mu}+\bar{\sigma}^2 z}{\bar{\sigma}^2+\sigma_z^2})^2}{\frac{\sigma_z^2\bar{\sigma}^2}{\bar{\sigma}^2+\sigma_z^2}}\Big]$$

A Gaussian is

$$N(\mu, \sigma^2) \propto \exp\Big[-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\Big]$$

So we can see that $p(x \mid z)$ has a mean of

$$\mu_{\texttt{posterior}} = \frac{\sigma_z^2\bar{\mu} + \bar{\sigma}^2 z}{\bar{\sigma}^2+\sigma_z^2}$$

and a variance of

$$\sigma_{\texttt{posterior}} = \frac{\sigma_z^2\bar{\sigma}^2}{\bar{\sigma}^2+\sigma_z^2}$$

I've dropped the constants, and so the result is not a normal, but proportional to one. Bayes theorem normalizes with the $p(z)$ divisor, ensuring that the result is normal. We normalize in the update step of our filters, ensuring the filter estimate is Gaussian.

$$\mathcal{N}_1 = \|\mathcal{N}_2 \cdot \mathcal{N}_3\|$$

## 3.16 Sum of Gaussians (Optional)

Likewise, this section is not important to read. Here I derive the equations for the sum of two Gaussians.

The sum of two Gaussians is given by

$$\mu = \mu_1 + \mu_2$$
$$\sigma^2 = \sigma_1^2 + \sigma_2^2$$

There are several proofs for this. I will use convolution since we used convolution in the previous chapter for the histograms of probabilities.

To find the density function of the sum of two Gaussian random variables we sum the density functions of each. They are nonlinear, continuous functions, so we need to compute the sum with an integral. If the random variables $p$ and $z$ (e.g. prior and measurement) are independent we can compute this with

$p(x) = \int_{-\infty}^{\infty} f_p(x-z) f_z(z)\, dx$

This is the equation for a convolution. Now we just do some math:

$p(x) = \int_{-\infty}^{\infty} f_2(x-x_1) f_1(x_1)\, dx$

$= \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\sigma_z} \exp\Big[-\frac{(x-z-\mu_z)^2}{2\sigma_z^2}\Big] \frac{1}{\sqrt{2\pi}\sigma_p} \exp\Big[-\frac{(x-\mu_p)^2}{2\sigma_p^2}\Big]\, dx$

$$= \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\sqrt{\sigma_p^2+\sigma_z^2}} \exp\left[-\frac{(x-(\mu_p+\mu_z)))^2}{2(\sigma_z^2+\sigma_p^2)}\right] \frac{1}{\sqrt{2\pi}\frac{\sigma_p\sigma_z}{\sqrt{\sigma_p^2+\sigma_z^2}}} \exp\left[-\frac{(x-\frac{\sigma_p^2(x-\mu_z)+\sigma_z^2\mu_p}{\sigma_z^2+\sigma_p^2}))^2}{2\left(\frac{\sigma_p\sigma_x}{\sqrt{\sigma_z^2+\sigma_p^2}}\right)^2}\right] dx$$

$$= \frac{1}{\sqrt{2\pi}\sqrt{\sigma_p^2+\sigma_z^2}} \exp\left[-\frac{(x-(\mu_p+\mu_z)))^2}{2(\sigma_z^2+\sigma_p^2)}\right] \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\frac{\sigma_p\sigma_z}{\sqrt{\sigma_p^2+\sigma_z^2}}} \exp\left[-\frac{(x-\frac{\sigma_p^2(x-\mu_z)+\sigma_z^2\mu_p}{\sigma_z^2+\sigma_p^2}))^2}{2\left(\frac{\sigma_p\sigma_x}{\sqrt{\sigma_z^2+\sigma_p^2}}\right)^2}\right] dx$$

The expression inside the integral is a normal distribution. The sum of a normal distribution is one, hence the integral is one. This gives us

$$p(x) = \frac{1}{\sqrt{2\pi}\sqrt{\sigma_p^2+\sigma_z^2}} \exp\left[-\frac{(x-(\mu_p+\mu_z)))^2}{2(\sigma_z^2+\sigma_p^2)}\right]$$

This is in the form of a normal, where

$$\mu_x = \mu_p + \mu_z$$
$$\sigma_x^2 = \sigma_z^2 + \sigma_p^2 \,\square$$

## 3.17 Summary and Key Points

This chapter is a poor introduction to statistics in general. I've only covered the concepts that needed to use Gaussians in the remainder of the book, no more. What I've covered will not get you very far if you intend to read the Kalman filter literature. If this is a new topic to you I suggest reading a statistics textbook. I've always liked the Schaum series for self study, and Alan Downey's *Think Stats* [5] is also very good and freely available online.

The following points **must** be understood by you before we continue:

- Normals express a continuous probability distribution
- They are completely described by two parameters: the mean ($\mu$) and variance ($\sigma^2$)
- $\mu$ is the average of all possible values
- The variance $\sigma^2$ represents how much our measurements vary from the mean
- The standard deviation ($\sigma$) is the square root of the variance ($\sigma^2$)
- Many things in nature approximate a normal distribution, but the math is not perfect.
- In filtering problems computing $p(x \mid z)$ is nearly impossible, but computing $p(z \mid x)$ is straightforward. Bayes' lets us compute the former from the latter.

The next several chapters will be using Gaussians with Bayes' theorem to help perform filtering. As noted in the last section, sometimes Gaussians do not describe the world very well. Latter parts of the book are dedicated to filters which work even when the noise or system's behavior is very non-Gaussian.

## 3.18 References

[1] https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/Supporting_Notebooks/Computing_and_plotting_PDFs.ipynb

[2] http://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.norm.html

[3] http://docs.scipy.org/doc/scipy/reference/tutorial/stats.html

[4] Huber, Peter J. *Robust Statistical Procedures*, Second Edition. Society for Industrial and Applied Mathematics, 1996.

[5] Downey, Alan. *Think Stats*, Second Edition. O'Reilly Media.
https://github.com/AllenDowney/ThinkStats2
http://greenteapress.com/thinkstats/

## 3.19 Useful Wikipedia Links

https://en.wikipedia.org/wiki/Probability_distribution
- https://en.wikipedia.org/wiki/Random_variable
- https://en.wikipedia.org/wiki/Sample_space
- https://en.wikipedia.org/wiki/Central_tendency
- https://en.wikipedia.org/wiki/Expected_value
- https://en.wikipedia.org/wiki/Standard_deviation
- https://en.wikipedia.org/wiki/Variance
- https://en.wikipedia.org/wiki/Probability_density_function
- https://en.wikipedia.org/wiki/Central_limit_theorem
- https://en.wikipedia.org/wiki/68%E2%80%9395%E2%80%9399.7_rule
- https://en.wikipedia.org/wiki/Cumulative_distribution_function
- https://en.wikipedia.org/wiki/Skewness
- https://en.wikipedia.org/wiki/Kurtosis

# Chapter 4

# One Dimensional Kalman Filters

```
In [1]: from __future__ import division, print_function
        %matplotlib inline

In [2]: #format the book
        import book_format
        book_format.set_style()

Out[2]:
```

Now that we understand the discrete Bayes filter and Gaussians we are prepared to implement a Kalman filter. We will do this exactly as we did the discrete Bayes filter - rather than starting with equations we will develop the code step by step based on reasoning about the problem.

"One dimensional" means that the filter only tracks one state variable, such as position on the x-axis. In subsequent chapters we will learn a more general multidimensional form of the filter that can track many state variables simultaneously, such as position, velocity, and acceleration. Recall that we used velocity in the g-h filter to get better estimates than by tracking position alone. The same is true for the Kalman filter.

So why not just jump into the multidimensional form of the filter? To be honest, the math is difficult, and my intuitive approach to developing the filter starts to break down. This math obscures the rather simple principles that allow the Kalman filter to work.

So, in this chapter we learn how to use Gaussians to implement a Bayesian filter. That's all the Kalman filter is - a Bayesian filter that uses Gaussians. In the next chapter we will switch to a multidimensional form and the full power of the Kalman filter will be unleashed!

## 4.1 Problem Description

As in the **Discrete Bayes Filter** chapter we will be tracking a moving object in a long hallway at work. Assume that in our latest hackathon someone created an RFID tracker that provides a reasonably accurate position of the dog. The sensor returns the distance of the dog from the left end of the hallway in meters. So, 23.4 would mean the dog is 23.4 meters from the left end of the hallway.

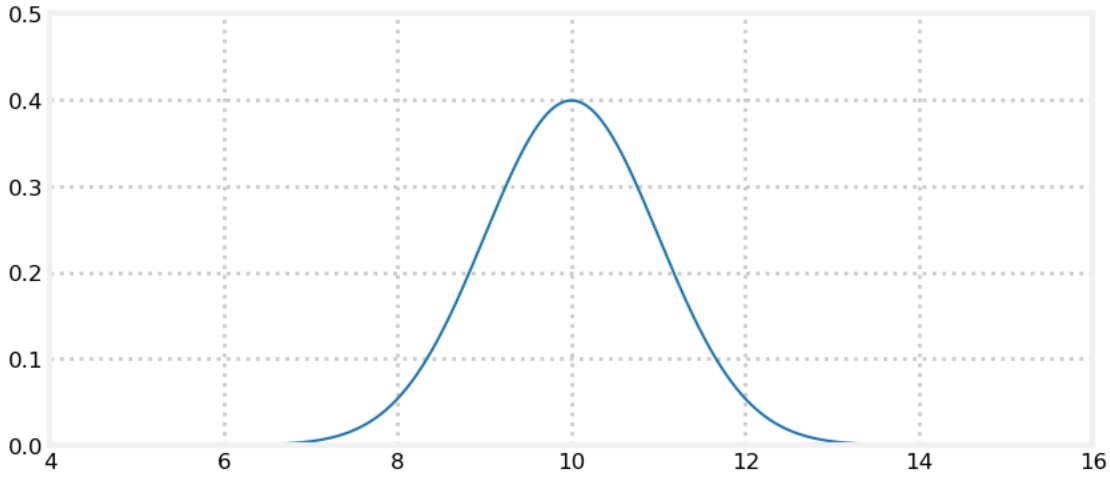
The sensor is not perfect. A reading of 23.4 could correspond to the dog being at 23.7, or 23.0. However, it is very unlikely to correspond to a position of 47.6. Testing during the hackathon confirmed this result - the sensor is 'reasonably' accurate, and while it had errors, the errors are small. Furthermore, the errors seemed to be evenly distributed on both sides of the true position; a position of 23 m would equally likely be measured as 22.9 or 23.1. Perhaps we can model this with a Gaussian.

We predict that the dog is moving. This prediction is not perfect. Sometimes our prediction will overshoot, sometimes it will undershoot. We are more likely to undershoot or overshoot by a little than a lot. Perhaps we can also model this with a Gaussian.

## 4.2 Beliefs as Gaussians

We can express our belief in the dog's position with a Gaussian. Say we believe that our dog is at 10 meters, and the variance in that belief is 1 $m^2$, or $\mathcal{N}(10, 1)$. A plot of the pdf follows:

```
In [3]: import filterpy.stats as stats
        stats.plot_gaussian_pdf(mean=10., variance=1.,
                                xlim=(4, 16), ylim=(0, .5));
```

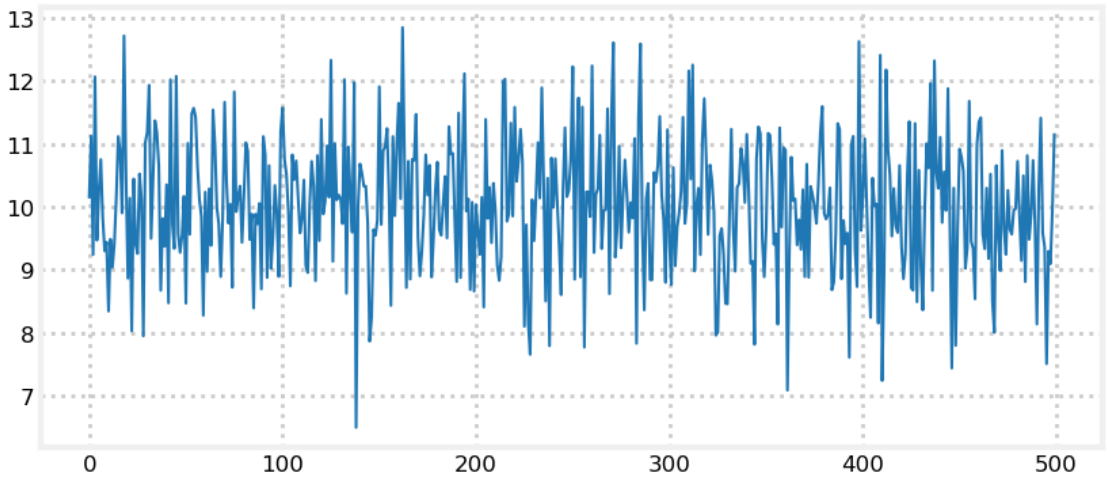



This plot depicts our uncertainty about the dog's position. It represents a fairly inexact belief. While we believe that it is most likely that the dog is at 10 m, any position from 9 m to 11 m or so are quite likely as well. Assume the dog is standing still, and we query the sensor again. This time it returns 10.2 m. Can we use this additional information to improve our estimate?

Intuition suggests we can. Consider: if we read the sensor 500 times and each time it returned a value between 8 and 12, all centered around 10, we should be very confident that the dog is near 10. Of course, a different interpretation is possible. Perhaps our dog was randomly wandering back and forth in a way that exactly emulated random draws from a normal distribution. But that seems extremely unlikely - I've never seen a dog do that. Let's look at 500 draws from $\mathcal{N}(10, 1)$:

```
In [4]: import numpy as np
        from numpy.random import randn
        import matplotlib.pyplot as plt

        xs = range(500)
        ys = randn(500)*1. + 10.
        plt.plot(xs, ys)
        print('Mean of readings is {:.3f}'.format(np.mean(ys)))
```

```
Mean of readings is 10.043
```




Eyeballing this confirms our intuition - no dog moves like this. However, noisy sensor data certainly looks this way. The computed mean of the readings is almost exactly 10. Assuming the dog is standing still, we say the dog is at position 10 with a variance of 1.

## 4.3 Tracking with Gaussian Probabilities

The discrete Bayes filter used a histogram of probabilities to track the dog. Each bin in the histogram represents a position, and the value is the probability of the dog being in that position.

Tracking was performed with a cycle of predictions and updates. We used the equations

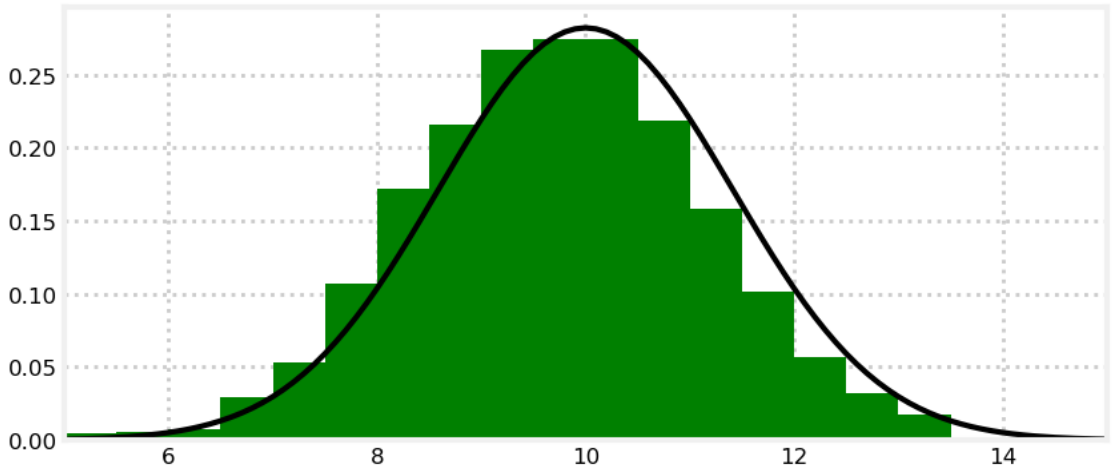
$$\begin{aligned} \bar{\mathbf{x}} &= \mathbf{x} * f_{\mathbf{x}}(\bullet) && \text{Predict} \\ \mathbf{x} &= \mathcal{L} \cdot \bar{\mathbf{x}} && \text{Update} \end{aligned}$$

to compute the new probability distributions. Recall that $\bar{\mathbf{x}}$ is the *prior*, $\mathcal{L}$ is the *likelihood* of a measurement given the prior $\bar{\mathbf{x}}$, $f_{\mathbf{x}}(\bullet)$ is the *process model*, and $*$ denotes *convolution*. $\mathbf{x}$ is bold to denote that it is a histogram of numbers, or a vector.

This method works, but led to histograms that implied the dog could be in multiple places at once. Also, the computations are very slow for large problems.

Can we replace $\mathbf{x}$, the histogram, with a Gaussian $\mathcal{N}(x, \sigma^2)$? Absolutely! We've learned how to express belief as a Gaussian. A Gaussian, which is a single number pair $\mathcal{N}(\mu, \sigma^2)$, can replace an entire histogram of probabilities:

```
In [5]: import kf_book.kf_internal as kf_internal
        kf_internal.gaussian_vs_histogram()
```




I hope you see the power of this. We can replace hundreds to thousands of numbers with a single pair of numbers: $x = \mathcal{N}(\mu, \sigma^2)$.

The tails of the Gaussian extend to infinity on both sides, so it incorporates arbitrarily many bars in the histogram. If this represents our belief in the position of the dog in the hallway, this one Gaussian covers the entire hallway (and the entire universe on that axis). We think that it is likely the dog is at 10, but he could be at 8, 14, or, with infinitesimally small probability, at $10^{80}$.

In this chapter we replace histograms with Gaussians:

| discrete Bayes | Gaussian | Step |
|---|---|---|
| $\bar{\mathbf{x}} = \mathbf{x} * f(\mathbf{x})$ | $\bar{x}_\mathcal{N} = x_\mathcal{N} \oplus f_{x_\mathcal{N}}(\bullet)$ | Predict |
| $\mathbf{x} = \|\mathcal{L}\bar{\mathbf{x}}\|$ | $x_\mathcal{N} = L \otimes \bar{x}_\mathcal{N}$ | Update |

where $\oplus$ and $\otimes$ is meant to express some unknown operator on Gaussians. I won't do it in the rest of the book, but the subscript indicates that $x_\mathcal{N}$ is a Gaussian.

The discrete Bayes filter used convolution for the prediction. We showed that it used the *total probabability theorem*, computed as a sum, so maybe we can add the Gaussians. It used multiplications to incorporate the measurement into the prior, so maybe we can multiply the Gaussians. Could it be this easy:

$$\bar{x} \stackrel{?}{=} x + f_x(\bullet)$$
$$x \stackrel{?}{=} \mathcal{L} \cdot \bar{x}$$

This will only work if the sum and product of two Gaussians is another Gaussian. Otherwise after the first epoch $x$ would not be Gaussian, and this scheme falls apart.

## 4.4 Predictions with Gaussians

We use Newton's equation of motion to compute current position based on the current velocity and previous position:

$$\begin{aligned}\bar{x}_k &= x_{k-1} + v_k\Delta t \\ &= x_{k-1} + f_x\end{aligned}$$

I've dropped the notation $f_x(\bullet)$ in favor of $f_x$ to keep the equations uncluttered.

If the dog is at 10 m, his velocity is 15 m/s, and the epoch is 2 seconds long, we have

$$\begin{aligned}f_x &= v\Delta t = 15 \cdot 2 \\ \bar{x}_k &= 10 + (15 \cdot 2) = 40\end{aligned}$$

We are uncertain about his current position and velocity, so this will not do. We need to express the uncertainty with a Gaussian.

Position is easy. We define $x$ as a Gaussian. If we think the dog is at 10 m, and the standard deviation of our uncertainty is 0.2 m, we get $x = \mathcal{N}(10, 0.2^2)$.

What about our uncertainty in his movement? We define $f_x$ as a Gaussian. If the dog's velocity is 15 m/s, the epoch is 1 second, and the standard deviation of our uncertainty is 0.7 m/s, we get $f_x = \mathcal{N}(15, 0.7^2)$.

The equation for the prior is

$$\bar{x} = x + f_x$$

What is the sum of two Gaussians? In the last chapter I proved that:

$$\mu = \mu_1 + \mu_2$$
$$\sigma^2 = \sigma_1^2 + \sigma_2^2$$

This is fantastic news; the sum of two Gaussians is another Gaussian!

The math works, but does this make intuitive sense? Think of the physical representation of this abstract equation. We have

$$x = \mathcal{N}(10, 0.2^2)$$
$$f_x = \mathcal{N}(15, 0.7^2)$$

If we add these we get:

$$\begin{aligned}\bar{x} = \mu_x + \mu_{f_x} &= 10 + 15 &&= 25 \\ \bar{\sigma}^2 = \sigma_x^2 + \sigma_{f_x}^2 &= 0.2^2 + 0.7^2 &&= 0.53\end{aligned}$$

It makes sense that the predicted position is the previous position plus the movement. What about the variance? It is harder to form an intuition about this. However, recall that with the `predict()` function for the discrete Bayes filter we always lost information. We don't really know where the dog is moving, so the confidence should get smaller (variance gets larger). $\sigma_{f_x}^2$ is the amount of uncertainty added to the system due to the imperfect prediction about the movement, and so we would add that to the existing uncertainty.

Let's take advantage of the `namedtuple` class in Python's `collection` module to implement a Gaussian object. We could implement a Gaussian using a tuple, where $\mathcal{N}(10, 0.04)$ is implemented in Python as `g = (10., 0.04)`. We would access the mean with `g[0]` and the variance with `g[1]`.

`namedtuple` works the same as a tuple, except you provide it with a type name and field names. It's not important to understand, but I modified the `__repr__` method to display its value using the notation in this chapter.

```
In [6]: from collections import namedtuple
        gaussian = namedtuple('Gaussian', ['mean', 'var'])
        gaussian.__repr__ = lambda s: '(=\{:.3f\}, ²=\{:.3f\})'.format(s[0], s[1])
```

Now we can create a print a Gaussian with:

```
In [7]: g1 = gaussian(3.4, 10.1)
        g2 = gaussian(mean=4.5, var=0.2**2)
        print(g1)
        print(g2)
```

```
(=3.400, ²=10.100)
(=4.500, ²=0.040)
```

We can access the mean and variance with either subscripts or field names:

```
In [8]: g1.mean, g1[0], g1[1], g1.var
```

```
Out[8]: (3.4, 3.4, 10.1, 10.1)
```

Here is our implementation of the predict function, where `pos` and `movement` are Gaussian tuples in the form $(\mu, \sigma^2)$:

```
In [9]: def predict(pos, movement):
            return gaussian(pos.mean + movement.mean, pos.var + movement.var)
```

Let's test it. What is the prior if the intitial position is the Gaussian $\mathcal{N}(10, 0.2^2)$ and the movement is the Gaussian $\mathcal{N}(15, 0.7^2)$?

```
In [10]: pos = gaussian(10., .2**2)
         move = gaussian(15., .7**2)
         predict(pos, move)
```

```
Out[10]: (=25.000, ²=0.530)
```

The prior states that the dog is at 25 m with a variance of 0.53 $m^2$, which is what we computed by hand.

## 4.5 Updates with Gaussians

The discrete Bayes filter encodes our belief about the position of our dog in a histogram of probabilities. The distribution is discrete and multimodal. It can express strong belief that the dog is in two positions at once, and the positions are discrete.

We are proposing that we replace the histogram with a Gaussian. The discrete Bayes filter used this code to compute the posterior:

```
def update(likelihood, prior):
    posterior = likelihood * prior
    return normalize(posterior)
```

which is an implementation of the equation:

$$x = \|\mathcal{L}\bar{x}\|$$

We've just shown that we can represent the prior with a Gaussian. What about the likelihood? The likelihood is the probability of the measurement given the current state. We've learned how to represent measurements as Gaussians. For example, maybe our sensor states that the dog is at 23 m, with a standard deviation of 0.4 meters. Our measurement, expressed as a likelihood, is $z = \mathcal{N}(23, 0.16)$.

Both the likelihood and prior are modeled with Gaussians. Can we multiply Gaussians? Is the product of two Gaussians another Gaussian?

Yes to the former, and almost to the latter! In the last chapter I proved that the product of two Gaussians is proportional to another Gausian.

$$\mu = \frac{\sigma_1^2\mu_2 + \sigma_2^2\mu_1}{\sigma_1^2 + \sigma_2^2},$$
$$\sigma^2 = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2 + \sigma_2^2}$$

We can immediately infer several things. If we normalize the result, the product is another Gaussian. If one Gaussian is the likelihood, and the second is the prior, then the mean is a scaled sum of the prior and the measurement. The variance is a combination of the variances of the prior and measurement. Finally, the variances are completely unaffected by the values of the mean!

We put this in Bayesian terms like so:

$$\begin{aligned}\mathcal{N}(\mu, \sigma^2) &= \|prior \cdot likelihood\| \\ &= \|\mathcal{N}(\bar{\mu}, \bar{\sigma}^2) \cdot \mathcal{N}(\mu_z, \sigma_z^2)\| \\ &= \mathcal{N}(\frac{\bar{\sigma}^2\mu_z + \sigma_z^2\bar{\mu}}{\bar{\sigma}^2 + \sigma_z^2}, \frac{\bar{\sigma}^2\sigma_z^2}{\bar{\sigma}^2 + \sigma_z^2})\end{aligned}$$

If we implemented that in a function `gaussian_multiply()` we could implement our filter's update step as

```
In [11]: def gaussian_multiply(g1, g2):
             mean = (g1.var * g2.mean + g2.var * g1.mean) / (g1.var + g2.var)
             variance = (g1.var * g2.var) / (g1.var + g2.var)
             return gaussian(mean, variance)

         def update(prior, likelihood):
             posterior = gaussian_multiply(likelihood, prior)
             return posterior

         # test the update function
         predicted_pos = gaussian(10., .2**2)
         measured_pos = gaussian(11., .1**2)
         estimated_pos = update(predicted_pos, measured_pos)
         estimated_pos
```

```
Out[11]: (=10.800, ²=0.008)
```

Perhaps this would be clearer if we used more specific names:

```
def update_dog(dog_pos, measurement):
    estimated_pos = gaussian_multiply(measurement, dog_pos)
    return estimated_pos
```

That is less abstract, which perhaps helps with comprehension, but it is poor coding practice. We are writing a Kalman filter that works for any problem, not just tracking dogs in a hallway, so we won't use variable names with 'dog' in them. Also, this form obscures the fact that we are multiplying the likelihood by the prior.

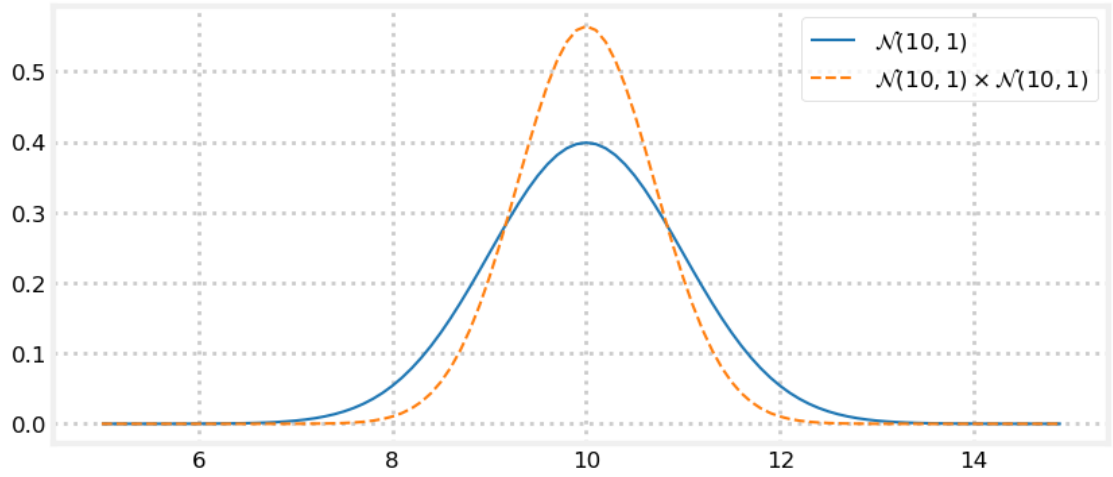
We have the majority of our filter implemented, but I fear this step is still a bit confusing. I've asserted that we can multiply Gaussians and that it correctly performs the update step, but why is this true? Let's take a detour and spend some time multiplying Gaussians.

### 4.5.1 Understanding Gaussian Multiplication

Let's plot the pdf of $\mathcal{N}(10, 1) \times \mathcal{N}(10, 1)$. Can you determine its shape without looking at the result? What should the new mean be? Will the curve be wider, narrower, or the same as $\mathcal{N}(10, 1)$?

```
In [12]: z = gaussian(10., 1.)  # Gaussian N(10, 1)

         product = gaussian_multiply(z, z)

         xs = np.arange(5, 15, 0.1)
         ys = [stats.gaussian(x, z.mean, z.var) for x in xs]
         plt.plot(xs, ys, label='$\mathcal{N}(10,1)$')

         ys = [stats.gaussian(x, product.mean, product.var) for x in xs]
         plt.plot(xs, ys, label='$\mathcal{N}(10,1) \\times \mathcal{N}(10,1)$', ls='--')
         plt.legend()
         print(product)
```

```
(=10.000, ²=0.500)
```




The result of the multiplication is taller and narrow than the original Gaussian but the mean is unchanged. Does this match your intuition?

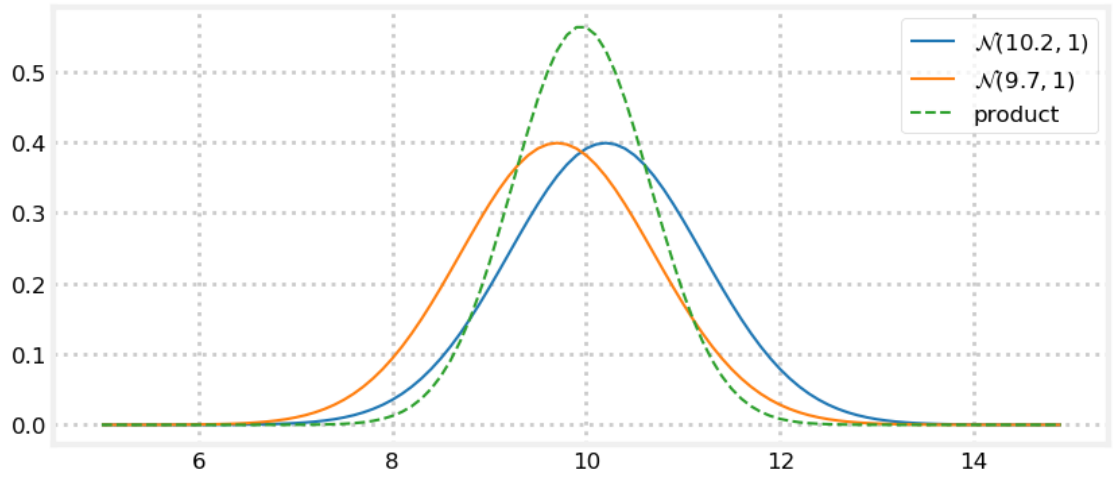
Think of the Gaussians as two measurements. If I measure twice and get 10 meters each time, I should conclude that the length is close to 10 meters. Thus the mean should be 10. It would make no sense to conclude the length is actually 11, or 9.5. Also, I am more confident with two measurements than with one, so the variance of the result should be smaller.

"Measure twice, cut once" is a well known saying. Gaussian multiplication is a mathematical model of this physical fact.

I'm unlikely to get the same measurement twice in a row. Now let's plot the pdf of $\mathcal{N}(10.2,\ 1) \times \mathcal{N}(9.7,\ 1)$. What do you think the result will be? Think about it, and then look at the graph.

The result of the multiplication is taller and narrow than the original Gaussian but the mean is unchanged. Does this match your intuition

```
In [13]: def plot_products(g1, g2):
             plt.figure()
             product = gaussian_multiply(g1, g2)

             xs = np.arange(5, 15, 0.1)
             ys = [stats.gaussian(x, g1.mean, g1.var) for x in xs]
             plt.plot(xs, ys, label='$\mathcal{N}$'+'$({},{})$'.format(g1.mean, g1.var))

             ys = [stats.gaussian(x, g2.mean, g2.var) for x in xs]
             plt.plot(xs, ys, label='$\mathcal{N}$'+'$({},{})$'.format(g2.mean, g2.var))

             ys = [stats.gaussian(x, product.mean, product.var) for x in xs]
             plt.plot(xs, ys, label='product', ls='--')
             plt.legend();

         z1 = gaussian(10.2, 1)
         z2 = gaussian(9.7, 1)

         plot_products(z1, z2)
```




If you ask two people to measure the distance of a table from a wall, and one gets 10.2 meters, and the other got 9.7 meters, your best guess must be the average, 9.95 meters if you trust the skills of both equally.

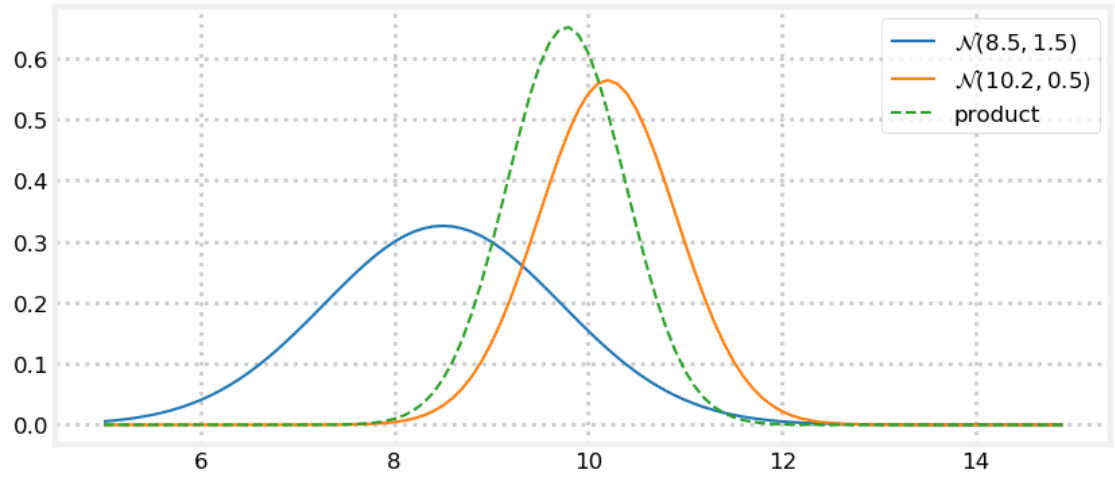
Recall the g-h filter. We agreed that if I weighed myself on two scales, and the first read 160 lbs while the second read 170 lbs, and both were equally accurate, the best estimate was 165 lbs. Furthermore I should be a bit more confident about 165 lbs vs 160 lbs or 170 lbs because I now have two readings, both near this estimate, increasing my confidence that neither is wildly wrong.

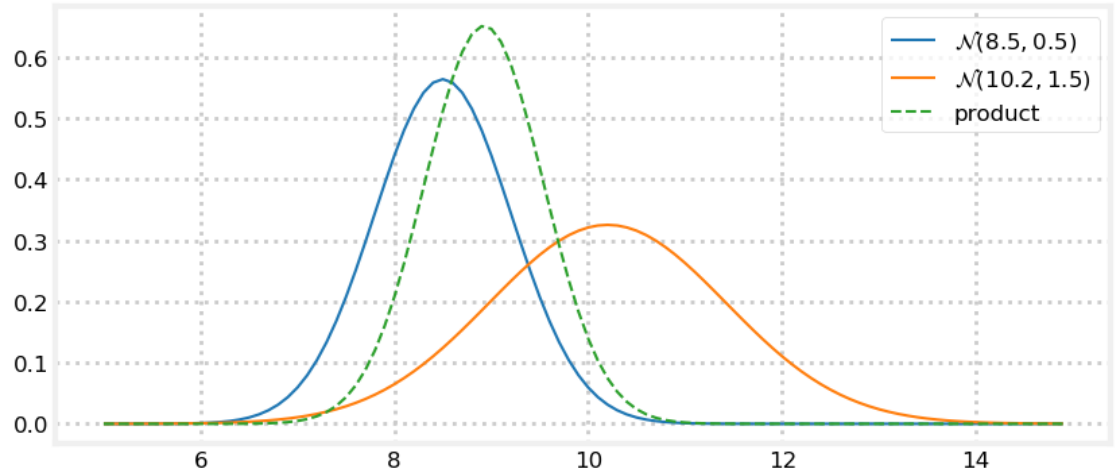
This becomes counter-intuitive in more complicated situations, so let's consider it further. Perhaps a more reasonable assumption would be that one person made a mistake, and the true distance is either 10.2 or 9.7, but certainly not 9.95. Surely that is possible. But we know we have noisy measurements, so we have no reason to think one of the measurements has no noise, or that one person made a gross error that allows us to discard their measurement. Given all available information, the best estimate must be 9.95.

In the update step of the Kalman filter we are not combining two measurements, but one measurement and the prior, our estimate before incorporating the measurement. We went through this logic for the g-h filter. It doesn't matter if we are incorporating information from two measurements, or a measurement and a prediction, the math is the same.

Let's look at that. I'll create a fairly inaccurate prior of $\mathcal{N}(8.5, 1.5)$ and a more accurate measurement of $\mathcal{N}(10.2, 0.5)$. By "accurate" I mean the sensor variance is smaller than the prior's variance, not that I somehow know that the dog is closer to 10.2 than 8.5. Next I'll plot the reverse relationship: an accurate prior of $\mathcal{N}(8.5, 0.5)$ and a inaccurate measurement of $\mathcal{N}(10.2, 1.5)$.

```
In [14]: prior, z = gaussian(8.5, 1.5), gaussian(10.2, 0.5)
         plot_products(prior, z)

         prior, z = gaussian(8.5, 0.5), gaussian(10.2, 1.5)
         plot_products(prior, z)
```




The result is a Gaussian that is taller than either input. This makes sense - we have incorporated information, so our variance should have been reduced. And notice how the result is far closer to the the input with the smaller variance. We have more confidence in that value, so it makes sense to weight it more heavily.

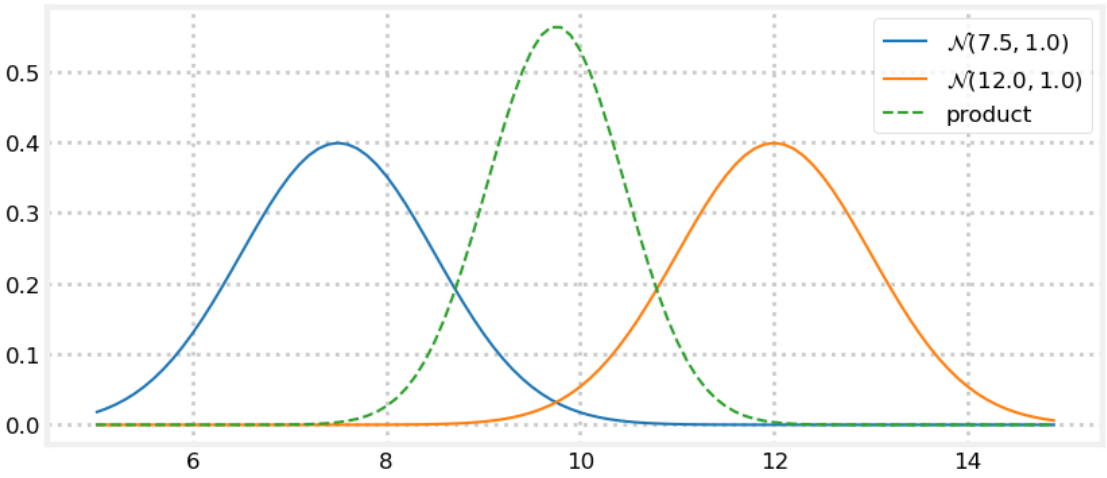
It *seems* to work, but is it really correct? There is more to say about this, but I want to get a working filter going so you can it experience it in concrete terms. After that we will revisit Gaussian multiplication and determine why it is correct.

### 4.5.2 Interactive Example

This interactive code provides sliders to alter the mean and variance of two Gaussians that are being multiplied together. As you move the sliders the plot is redrawn. Place your cursor inside the code cell and press CTRL+Enter to execute it.

```
In [15]: from ipywidgets import interact

         def interactive_gaussian(m1, m2, v1, v2):
```

```
    g1 = gaussian(m1, v1)
    g2 = gaussian(m2, v2)
    plot_products(g1, g2)

interact(interactive_gaussian,
         m1=(5, 10., .5), m2=(10, 15, .5),
         v1=(.1, 2, .1), v2=(.1, 2, .1));
```




## 4.6 First Kalman Filter

Let's get back to concrete terms and implement a Kalman filter. We've implemented the `update()` and `predict()` functions. We just need to write some boilerplate code to simulate a dog and create the measurements. I've put a `DogSimulation` class in `kf_internal` to avoid getting distracted with that task.

This boilerplate code sets up the problem by definine the means, variances, and generating the simulated dog movement.

```
In [16]: import kf_book.kf_internal as kf_internal
         from kf_book.kf_internal import DogSimulation

         np.random.seed(13)

         process_var = 1. # variance in the dog's movement
         sensor_var = 2. # variance in the sensor

         x = gaussian(0., 20.**2)  # dog's position, N(0, 20**2)
         velocity = 1
         dt = 1. # time step in seconds
         process_model = gaussian(velocity*dt, process_var) # displacement to add to x

         # simulate dog and get measurements
         dog = DogSimulation(
             x0=x.mean,
             velocity=process_model.mean,
             measurement_var=sensor_var,
             process_var=process_model.var)
```

```
        # create list of measurements
        zs = [dog.move_and_sense() for _ in range(10)]
```

And here is the Kalman filter.

```
In [17]: print('PREDICT\t\t\tUPDATE')
         print('     x      var\t\t  z\t    x      var')

         # perform Kalman filter on measurement z
         for z in zs:
             prior = predict(x, process_model)
             likelihood = gaussian(z, sensor_var)
             x = update(prior, likelihood)

             kf_internal.print_gh(prior, x, z)

         print()
         print('final estimate:        {:10.3f}'.format(x.mean))
         print('actual final position: {:10.3f}'.format(dog.x))
```

```
PREDICT                        UPDATE
     x      var                   z            x      var
  1.000  401.000         1.354          1.352   1.990
  2.352    2.990         1.882          2.070   1.198
  3.070    2.198         4.341          3.736   1.047
  4.736    2.047         7.156          5.960   1.012
  6.960    2.012         6.939          6.949   1.003
  7.949    2.003         6.844          7.396   1.001
  8.396    2.001         9.847          9.122   1.000
 10.122    2.000        12.553         11.338   1.000
 12.338    2.000        16.273         14.305   1.000
 15.305    2.000        14.800         15.053   1.000

final estimate:            15.053
actual final position:     14.838
```

Here is an animation of the filter. Predictions are plotted with a red triangle. After the prediction, the filter receives the next measurement, plotted as a black circle. The filter then forms an estimate part way between the two.

```
In [18]: from kf_book import book_plots as book_plots
         from ipywidgets.widgets import IntSlider

         # save output in these lists for plotting
         xs, predictions = [], []

         process_model = gaussian(velocity, process_var)

         # perform Kalman filter
         x = gaussian(0., 20.**2)
         for z in zs:
             prior = predict(x, process_model)
             likelihood = gaussian(z, sensor_var)
```

```
        x = update(prior, likelihood)

        # save results
        predictions.append(prior.mean)
        xs.append(x.mean)

    def plot_filter(step):
        plt.cla()
        step -= 1
        i = step // 3 + 1

        book_plots.plot_predictions(predictions[:i])
        if step % 3 == 0:
            book_plots.plot_measurements(zs[:i-1])
            book_plots.plot_filter(xs[:i-1])
        elif step % 3 == 1:
            book_plots.plot_measurements(zs[:i])
            book_plots.plot_filter(xs[:i-1])
        else:
            book_plots.plot_measurements(zs[:i])
            book_plots.plot_filter(xs[:i])

        plt.xlim(-1, 10)
        plt.ylim(0, 20)
        plt.legend(loc=2);
    interact(plot_filter, step=IntSlider(value=1, min=1, max=len(predictions)*3));
```


I've plotted the prior (labeled *prediction*), the measurements, and the filter output. For each iteration of the loop we form a prior, take a measurement, form a likelihood from the measurement, and then incorporate the likelihood into the prior.

If you look at the plot you can see that the filter estimate is always between the measurement and prediction. Recall that for the g-h filter we argued that the estimate must always be between the measurement and prior. It makes no sense to choose a value outside of the two values. If I predict I am at 10, but measure that I am at 9, it would be foolish to decide that I must be at 8, or 11.

## 4.7 Code Walkthrough

Now let's walk through the code.

```
process_var = 1.
sensor_var = 2.
```

These are the variances for the process model and sensor. The meaning of sensor variance should be clear - it is how much variance there is in each measurement. The process variance is how much error there is in the process model. We are predicting that at each time step the dog moves forward one meter. Dogs rarely do what we expect, and things like hills or the whiff of a squirrel will change his progress. If this was a robot responding to digital commands the performance would be much better, and perhaps the variance would be $\sigma^2 = .05$. These are not 'magic' numbers; the square root of the variance is the distance error in meters. It is easy to get a Kalman filter working by just plugging in numbers, but if the numbers do not reflect reality the performance of the filter will be poor.

```
x = gaussian(0., 20.**2)
```

This is the dog's initial position expressed as a Gaussian. The position is 0 meters, and the variance to 400 $m^2$, which is a standard deviation of 20 meters. You can think of this as saying "I believe with 99.7% accuracy the position is 0 plus or minus 60 meters". This is because with Gaussians ~99.7% of values fall within $\pm 3\sigma$ of the mean.

```
process_model = gaussian(velocity, process_var)
```

This is the process model - the description of how we think the dog moves. How do I know the velocity? Magic? Consider it a prediction, or perhaps we have a secondary velocity sensor. If this is a robot then this would be a control input to the robot. In subsequent chapters we will learn how to handle situations where you don't have a velocity sensor or input, so please accept this simplification for now.

Next we initialize the simulation and create 10 measurements:

```
dog = DogSimulation(
    x0=x.mean,
    velocity=process_model.mean,
    measurement_var=sensor_var,
    process_var=process_model.var)

zs = [dog.move_and_sense() for _ in range(10)]
```

Now we enter our `predict() ... update()` loop.

```
for z in zs:
    prior = predict(x, process_model)
    likelihood = gaussian(z, sensor_var)
    x = update(prior, likelihood)
```

The first time through the loop `prior` is `(1.0, 401.0)`, as can be seen in the printed table. After the prediction, we believe that we are at 1.0, and the variance is now 401, up from 400. The variance got worse, which is what always happens during the prediction step because it involves a loss of information.

Then we call the update function using `prior` as the current position.

For this I get this as the result: `pos = (1.352, 1.990), z = 1.354`.

What is happening? The dog is actually at 1.0 but the measured position is 1.354 due to sensor noise. That is pretty far from the predicted value of 1. The variance of the prior is 401 $m^2$. A large variance implies that confidence is very low, so the filter estimates the position to be very close to the measurement: 1.352.

Now look at the variance: 1.99 $m^2$. It has dropped tremendously from 401 $m^2$. Why? Well, the RFID has a reasonably small variance of 2.0 $m^2$, so we trust it far more than the prior. However, the previous belief does contain a bit of useful information, so our variance is now slightly smaller than 2.0.

Now the software loops, calling `predict()` and `update()` in turn. By the end the final estimated position is 15.053 vs the actual position of 14.838. The variance has converged to 1.0 $m^2$.

Now look at the plot. The noisy measurements are plotted with black circles, and the filter results are drawn with a solid blue line. Both are quite noisy, but notice how much noisier the measurements are. I plotted the prediction (prior) with red triangles. The estimate always lies between the prior and the measurement. This is your first Kalman filter and it seems to work!

The filtering is implemented in only a few lines of code. Most of the code is either initialization, storing of data, simulating the dog movement, and printing results. The code that performs the filtering is very succinct:

```
prior = predict(x, process_model)
likelihood = gaussian(z, sensor_var)
x = update(prior, likelihood)
```

If we didn't use the `predict` and `update` functions the code might be:

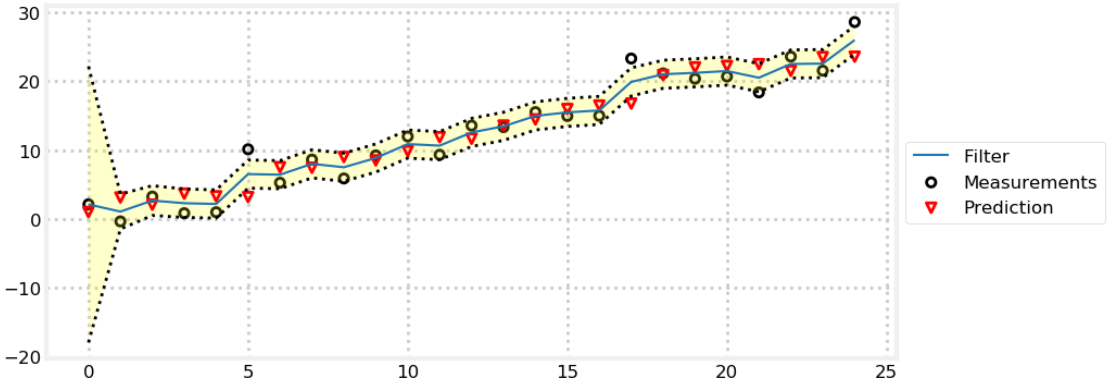
```
for z in zs:
    # predict
    dx = velocity*dt
    pos = pos + dx
    var = var + process_var

    # update
    pos  = (var*z + sensor_var*pos) / (var + sensor_var)
    var = (var * sensor_var) / (var + sensor_var)
```

Just 5 lines of very simple math implements the entire filter!

In this example I only plotted 10 data points so the output from the print statements would not overwhelm us. Now let's look at the filter's performance with more data. The variance is plotted as a lightly shaded yellow area between dotted lines. I've increased the size of the process and sensor variance so they are easier to see on the chart - for a real Kalman filter of course you will not be randomly changing these values.

```
In [19]: process_var = 2.
         sensor_var = 4.5
         x = gaussian(0., 400.)
         process_model = gaussian(1., process_var)
         N = 25

         dog = DogSimulation(x.mean, process_model.mean, sensor_var, process_var)
         zs = [dog.move_and_sense() for _ in range(N)]

         xs, priors = np.zeros((N, 2)), np.zeros((N, 2))
         for i, z in enumerate(zs):
             prior = predict(x, process_model)
             x = update(prior, gaussian(z, sensor_var))
             priors[i] = prior

             xs[i] = x

         book_plots.plot_measurements(zs)
         book_plots.plot_filter(xs[:, 0], var=priors[:, 1])
         book_plots.plot_predictions(priors[:, 0])
         book_plots.show_legend()
         kf_internal.print_variance(xs)
```

```
4.4502 2.6507 2.2871 2.1955 2.1712
2.1647 2.1629 2.1625 2.1623 2.1623
2.1623 2.1623 2.1623 2.1623 2.1623
2.1623 2.1623 2.1623 2.1623 2.1623
2.1623 2.1623 2.1623 2.1623 2.1623
```




Here we can see that the variance converges to 2.1623 in 9 steps. This means that we have become very confident in our position estimate. It is equal to $\sigma = 1.47$ meters. Contrast this to the sensor's $\sigma = 2.12$ meters. The first few measurements are unsure due to our uncertainty of the initial position, but the filter quickly converges to an estimate with lower variance than the sensor!

This code fully implements a Kalman filter. If you have tried to read the literature you are perhaps surprised, because this looks nothing like the endless pages of math in those books. So long as we worry about *using* the equations rather than *deriving* them the topic is approachable. Moreover, I hope you'll agree that you have a decent intuitive grasp of what is happening. We represent beliefs with Gaussians, and they get better over time because more measurements means we have more data to work with.

### 4.7.1 Exercise: Modify Variance Values

Modify the values of `process_var` and `sensor_var` and note the effect on the filter and on the variance. Which has a larger effect on the variance convergence? For example, which results in a smaller variance:

```
process_var = 40
sensor_var = 2
```

or:

```
process_var = 2
sensor_var = 40
```

### 4.7.2 KF Animation

If you are reading this in a browser you will be able to see an animation of the filter tracking the dog directly below this sentence.

The top plot shows the output of the filter in green, and the measurements with a dashed red line. The bottom plot shows the Gaussian at each step.

When the track first starts you can see that the measurements varies quite a bit from the initial prediction. At this point the Gaussian probability is small (the curve is low and wide) so the filter does not trust its prediction. As a result, the filter adjusts its estimate a large amount. As the filter innovates you can see that as the Gaussian becomes taller, indicating greater certainty in the estimate, the filter's output becomes

very close to a straight line. At `x = 15` and greater you can see that there is a large amount of noise in the measurement, but the filter does not react much to it compared to how much it changed for the first noisy measurement.

## 4.8 Kalman Gain

We see that the filter works. Now let's go back to the math to understand what is happening. The posterior $x$ is computed as the likelihood times the prior ($\mathcal{L}\bar{x}$), where both are Gaussians.

Therefore the mean of the posterior is given by:

$$\mu = \frac{\bar{\sigma}^2\,\mu_z + \sigma_z^2\,\bar{\mu}}{\bar{\sigma}^2 + \sigma_z^2}$$

I use the subscript $z$ to denote the measurement. We can rewrite this as:

$$\mu = \left(\frac{\bar{\sigma}^2}{\bar{\sigma}^2 + \sigma_z^2}\right)\mu_z + \left(\frac{\sigma_z^2}{\bar{\sigma}^2 + \sigma_z^2}\right)\bar{\mu}$$

In this form it is easy to see that we are scaling the measurement and the prior by weights:

$$\mu = W_1\mu_z + W_2\bar{\mu}$$

The weights sum to one because the denominator is a normalization term. We introduce a new term, $K = W_1$, giving us:

$$\begin{aligned}\mu &= K\mu_z + (1-K)\bar{\mu} \\ &= \bar{\mu} + K(\mu_z - \bar{\mu})\end{aligned}$$

where

$$K = \frac{\bar{\sigma}^2}{\bar{\sigma}^2 + \sigma_z^2}$$

$K$ is the *Kalman gain*. It's the crux of the Kalman filter. It is a scaling term that chooses a value partway between $\mu_z$ and $\bar{\mu}$.

Let's work a few examples. If the measurement is nine times more accurate than the prior, then $\bar{\sigma}^2 = 9\sigma_z^2$, and

$$\begin{aligned}\mu &= \frac{9\sigma_z^2\mu_z + \sigma_z^2\,\bar{\mu}}{9\sigma_z^2 + \sigma_\mathbf{z}^2} \\ &= \left(\frac{9}{10}\right)\mu_z + \left(\frac{1}{10}\right)\bar{\mu}\end{aligned}$$

Hence $K = \frac{9}{10}$, and to form the posterior we take nine tenths of the measurement and one tenth of the prior.

If the measurement and prior are equally accurate, then $\bar{\sigma}^2 = \sigma_z^2$ and

$$\begin{aligned}\mu &= \frac{\sigma_z^2\,(\bar{\mu} + \mu_z)}{2\sigma_\mathbf{z}^2} \\ &= \left(\frac{1}{2}\right)\bar{\mu} + \left(\frac{1}{2}\right)\mu_z\end{aligned}$$

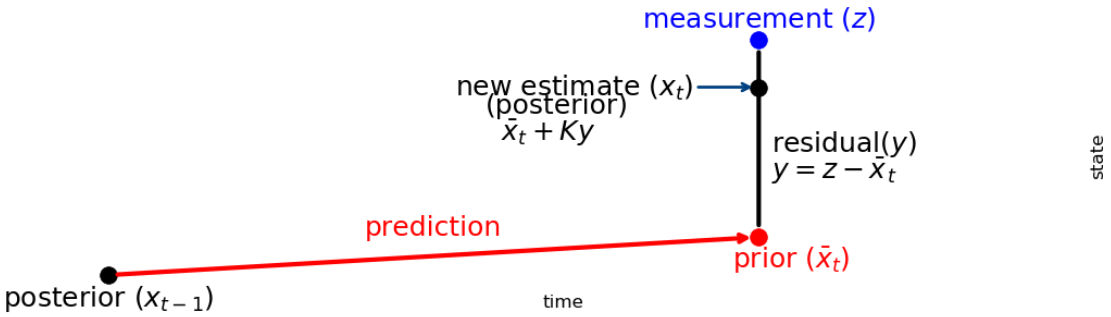
which is the average of the two means. It makes intuitive sense to take the average of two equally accurate values.

We can also express the variance in terms of the Kalman gain:

$$\begin{aligned}\sigma^2 &= \frac{\bar{\sigma}^2\sigma_z^2}{\bar{\sigma}^2 + \sigma_z^2} \\ &= K\sigma_z^2 \\ &= (1-K)\bar{\sigma}^2\end{aligned}$$

We can understand this by looking at this chart:

```
In [20]: import kf_book.book_plots as book_plots
         book_plots.show_residual_chart()
```




The Kalman gain $K$ is a scale factor that chooses a value along the residual. This leads to an alternative but equivalent implementation for `update()` and `predict()`:

```
In [21]: def update(prior, measurement):
             x, P = prior        # mean and variance of prior
             z, R = measurement  # mean and variance of measurement

             y = z - x        # residual
             K = P / (P + R)  # Kalman gain

             x = x + K*y      # posterior
             P = (1 - K) * P  # posterior variance
             return gaussian(x, P)

         def predict(posterior, movement):
             x, P = posterior # mean and variance of posterior
             dx, Q = movement # mean and variance of movement
             x = x + dx
             P = P + Q
             return gaussian(x, P)
```

Why have I written it in this form, and why have I chosen these terrible variable names? A few related reasons. A majority of books and papers present the Kalman filter in this form. My derivation of the filter from Bayesian principles is not unknown, but it is not used nearly as often. Alternative derivations naturally lead to this form of the equations. Also, the equations for the multivariate Kalman filter look almost exactly like these equations. So, you need to learn and understand them.

Where do the names `z`, `P`, `Q`, and `R` come from? You will see them used in the rest of this book. In the literature $R$ is nearly universally used for the measurement noise, $Q$ for the process noise and $P$ for the variance of the state. Using $z$ for the measurement is common, albeit not universal. Almost every book and paper you read will use these variable names. Get used to them.

This is also a powerful way to think about filtering. This is the way we reasoned about the g-h filter. It emphasizes taking the residual $y = \mu_z - \bar{\mu}$, finding the Kalman gain as a ratio of our uncertainty in the prior and measurement $K = P/(P + R)$, and computing the posterior by adding $Ky$ to the prior.

The Bayesian aspect is obscured in this form, as is the fact that we are multiplying the likelihood by the prior. Both viewpoints are equivalent because the math is identical. I chose the Bayesian approach because I think it give a much more intuitive yet deep understanding of the probabilistic reasoning. This alternative form using $K$ gives a deep understanding of what is known as the *orthogonal projection* approach. Dr. Kalman used that derivation, not Bayesian reasoning, when he invented this filter. You will understand more about this in the next few chapters.

## 4.9 Full Description of the Algorithm

Recall the diagram we used for the g-h filter:

We've been doing the same thing in this chapter. The Kalman filter makes a prediction, takes a measurement, and then forms a new estimate somewhere between the two.

**This is extremely important to understand**: Every filter in this book implements the same algorithm, just with different mathematical details. The math can become challenging in later chapters, but the idea is easy to understand.

It is important to see past the details of the equations of a specific filter and understand *what* the equations are calculating and *why*. There are a tremendous number of filters. They all use different math to implement the same algorithm. The choice of math affects the quality of results and what problems can be represented, but not the underlying ideas.

Here is the generic algorithm:

**Initialization**

```
1. Initialize the state of the filter
2. Initialize our belief in the state
```

**Predict**

```
1. Use system behavior to predict state at the next time step
2. Adjust belief to account for the uncertainty in prediction
```

**Update**

```
1. Get a measurement and associated belief about its accuracy
2. Compute residual between estimated state and measurement
3. Compute scaling factor based on whether the measurement
or prediction is more accurate
4. set state between the prediction and measurement based
on scaling factor
5. update belief in the state based on how certain we are
in the measurement
```

You will be hard pressed to find a Bayesian filter algorithm that does not fit into this form. Some filters will not include some aspects, such as error in the prediction, and others will have very complicated methods of computation, but this is what they all do.

The equations for the univariate Kalman filter are:

Predict

| Equation | Implementation | Kalman Form |
|---|---|---|
| $\bar{x} = x + f_x$ | $\bar{\mu} = \mu + \mu_{f_x}$<br>$\bar{\sigma}^2 = \sigma^2 + \sigma^2_{f_x}$ | $\bar{x} = x + dx$<br>$\bar{P} = P + Q$ |

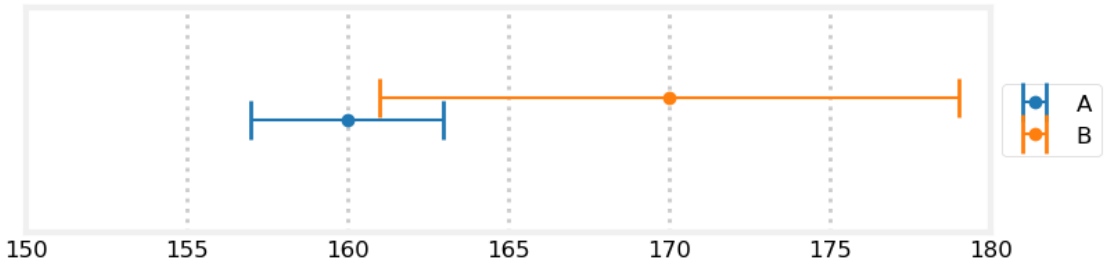
Update

| Equation | Implementation | Kalman Form |
|---|---|---|
| $x = \|\mathcal{L}\bar{x}\|$ | $y = z - \bar{\mu}$<br>$K = \frac{\bar{\sigma}^2}{\bar{\sigma}^2+\sigma_z^2}$<br>$\mu = \bar{\mu} + Ky$<br>$\sigma^2 = \frac{\bar{\sigma}^2\sigma_z^2}{\bar{\sigma}^2+\sigma_z^2}$ | $y = z - \bar{x}$<br>$K = \frac{\bar{P}}{\bar{P}+R}$<br>$x = \bar{x} + Ky$<br>$P = (1-K)\bar{P}$ |

## 4.10 Comparison with g-h and discrete Bayes Filters

Now is a good time to understand the differences between these three filters in terms of how we model errors. For the g-h filter we modeled our measurements as shown in this graph:

```
In [22]: book_plots.plot_errorbars([(160, 3, 'A'), (170, 9, 'B')], xlims=(150, 180))
```

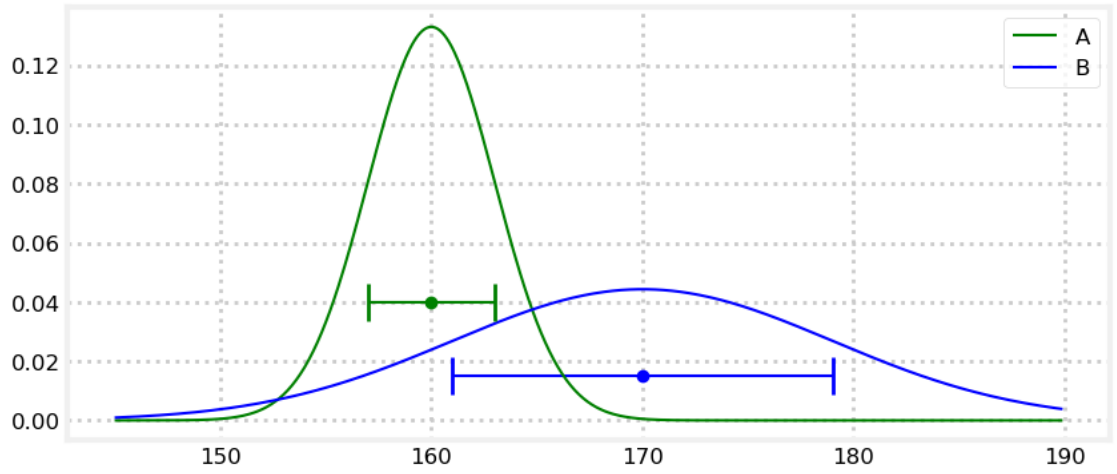



Sensor A returned a measurement of 160, and sensor B returned 170. The bars are *error bars* - they illustrate the possible range of error for the measurement. Hence, the actual value that A is measuring can be between 157 to 163, and B is measuring a value between 161 to 179.

I did not define it at the time, but this is a [*uniform distribution*](https://en.wikipedia.org/wiki/Uniform_distribution_(continuous)). A uniform distribution assigns equal probability to any event in the range. According to this model it is equally likely for sensor A to read 157, 160, or 163. Any value outside these ranges have 0 probability.

We can model this situation with Gaussians. I'll use $\mathcal{N}(160, 3^2)$ for sensor A, and $\mathcal{N}(170, 9^2)$ for sensor B. I've plotted these below with the uniform distribution error bars for comparison.

```
In [23]: xs = np.arange(145, 190, 0.1)
         ys = [stats.gaussian(x, 160, 3**2) for x in xs]
         plt.plot(xs, ys, label='A', color='g')

         ys = [stats.gaussian(x, 170, 9**2) for x in xs]
         plt.plot(xs, ys, label='B', color='b')
         plt.legend();
         plt.errorbar(160, [0.04], xerr=[3], fmt='o', color='g', capthick=2, capsize=10)
         plt.errorbar(170, [0.015], xerr=[9], fmt='o', color='b', capthick=2, capsize=10);
```

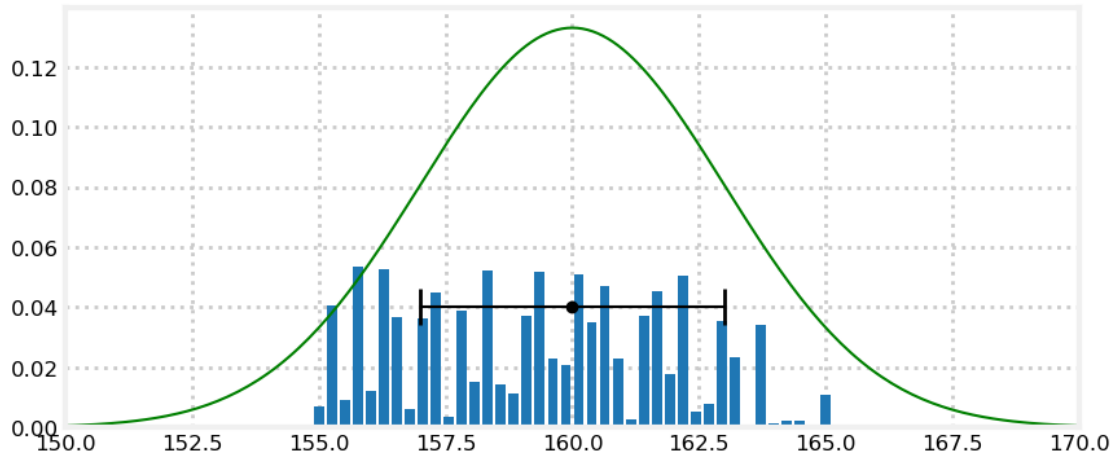



Using a uniform or Gaussian distribution is a modeling choice. Neither exactly describes reality. In most cases the Gaussian distribution is more realistic. Most sensors are more likely to return readings near the value being measured, and unlikely to return a reading far from that value. The Gaussian models this tendency. In contrast the uniform distribution assumes that any measurement within a range is equally likely.

Now let's see the *discrete distribution* used in the discrete Bayes filter. This model divides the range of possible values into discrete ranges and assigns a probability to each bucket. This assignment can be entirely arbitrary so long as the probabilities sum to one.

Let's plot the data for one sensor using a uniform distribution, a Gaussian distribution, and a discrete distribution.

```
In [24]: from random import random
         xs = np.arange(145, 190, 0.1)
         ys = [stats.gaussian(x, 160, 3**2) for x in xs]
         belief = np.array([random() for _ in range(40)])
         belief = belief / sum(belief)

         x = np.linspace(155, 165, len(belief))
         plt.gca().bar(x, belief, width=0.2)
         plt.plot(xs, ys, label='A', color='g')
         plt.errorbar(160, [0.04], xerr=[3], fmt='o', color='k', capthick=2, capsize=10)
         plt.xlim(150, 170);
```




I used random numbers to form the discrete distribution to illustrate that it can model any arbitrary probability distribution. This provides it with enormous power. With enough discrete buckets we can model the error characteristics of any sensor no matter how complicated. But with this power comes mathematical intractability. Multiplying or adding Gaussians takes two lines of math, and the result is another Gaussian. This regularity allows us to perform powerful analysis on the performance and behavior of our filters. Multiplying or adding a discrete distribution requires looping over the data, and we have no easy way to characterize the result. Analyzing the performance characteristics of a filter based on a discrete distribution is extremely difficult to impossible.

There is no 'correct' choice here. Later in the book we will introduce the *particle filter* which uses a discrete distribution. It is an extremely powerful technique because it can handle arbitrarily complex situations. This comes at the cost of slow performance, and resistance to analytical analysis.

For now we will ignore these matters and return to using Gaussians for the next several chapters. As we progress you will learn the strengths and limitations of using Gaussians in our mathematical models.

## 4.11 Introduction to Designing a Filter

So far we have developed filters for a position sensor. We are used to this problem by now, and may feel ill-equipped to implement a Kalman filter for a different problem. To be honest, there is still quite a bit of information missing from this presentation. Following chapters will fill in the gaps. Still, let's get a feel for it by designing and implementing a Kalman filter for a thermometer. The sensor for the thermometer outputs a voltage that corresponds to the temperature that is being measured. We have read the manufacturer's specifications for the sensor, and it tells us that the sensor exhibits white noise with a standard deviation of 0.13 volts.

We can simulate the temperature sensor measurement with this function:

```
In [25]: def volt(voltage, std):
             return voltage + (randn() * std)
```

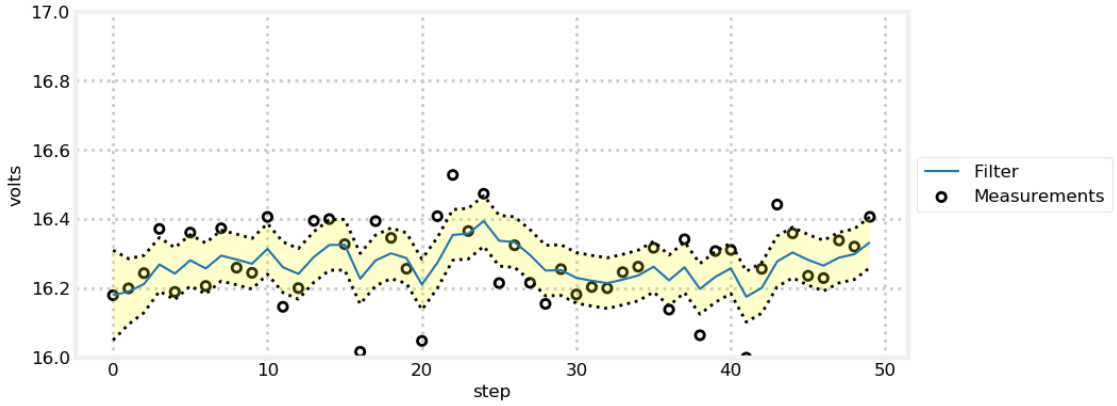
Now we need to write the Kalman filter processing loop. As with our previous problem, we need to perform a cycle of predicting and updating. The sensing step probably seems clear - call `volt()` to get the measurement, pass the result into `update()` method, but what about the predict step? We do not have a sensor to detect 'movement' in the voltage, and for any small duration we expect the voltage to remain constant. How shall we handle this?

As always, we will trust in the math. We have no known movement, so we will set that to zero. However, that means that we are predicting that the temperature will never change. If that is true, then over time we should become extremely confident in our results. Once the filter has enough measurements it will become

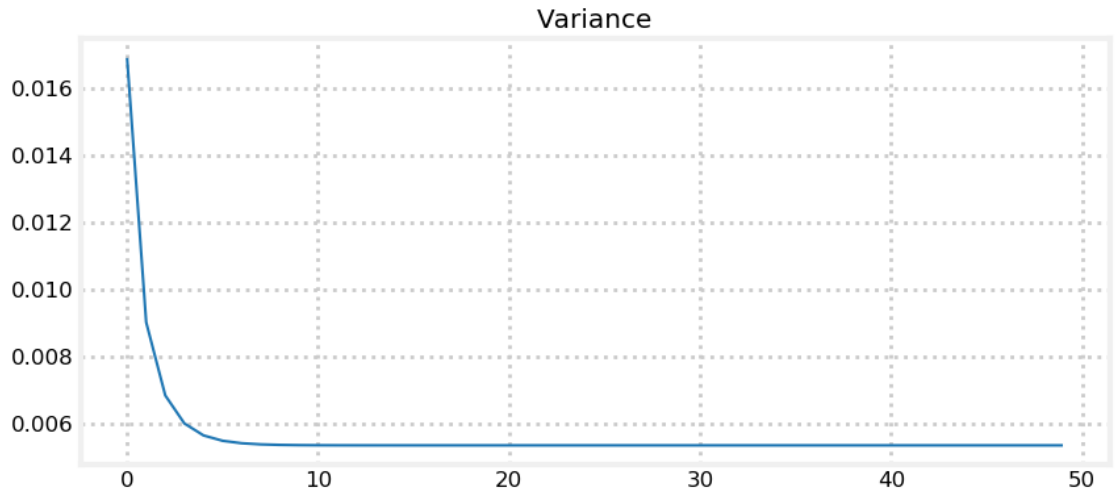
very confident that it can predict the subsequent temperatures, and this will lead it to ignoring measurements that result due to an actual temperature change. This is called a *smug* filter, and is something you want to avoid. So we will add a bit of error to our prediction step to tell the filter not to discount changes in voltage over time. In the code below I set `process_var = .05**2`. This is the expected variance in the change of voltage over each time step. I chose this value merely to be able to show how the variance changes through the update and predict steps. For a real sensor you would set this value for the actual amount of change you expect. For example, this would be an extremely small number if it is a thermometer for ambient air temperature in a house, and a high number if this is a thermocouple in a chemical reaction chamber. We will say more about selecting the actual value in the later chapters.

Let's see what happens.

```
In [26]: temp_change = 0
         voltage_std = .13
         process_var = .05**2
         actual_voltage = 16.3

         x = gaussian(25., 1000.) # initial state
         process_model = gaussian(0., process_var)

         N = 50
         zs = [volt(actual_voltage, voltage_std) for i in range(N)]
         ps = []
         estimates = []

         for z in zs:
             prior = predict(x, process_model)
             x = update(prior, gaussian(z, voltage_std**2))

             # save for latter plotting
             estimates.append(x.mean)
             ps.append(x.var)

         # plot the filter output and the variance
         book_plots.plot_measurements(zs)
         book_plots.plot_filter(estimates, var=np.array(ps))
         book_plots.show_legend()
         plt.ylim(16, 17)
         book_plots.set_labels(x='step', y='volts')
         plt.show()

         plt.plot(ps)
         plt.title('Variance')
         print('Variance converges to {:.3f}'.format(ps[-1]))
```




```
Variance converges to 0.005
```




The first plot shows the individual sensor measurements vs the filter output. Despite a lot of noise in the sensor we quickly discover the approximate voltage of the sensor. In the run I just completed at the time of authorship, the last voltage output from the filter is 16.213, which is quite close to the 16.4 used by the `volt()` function. On other runs I have gotten larger and smaller results.

Spec sheets are what they sound like - specifications. Any individual sensor will exhibit different performance based on normal manufacturing variations. Values are often maximums - the spec is a guarantee that the performance will be at least that good. If you buy an expensive piece of equipment it often comes with a sheet of paper displaying the test results of your specific item; this is usually very trustworthy. On the other hand, if this is a cheap sensor it is likely it received little to no testing prior to being sold. Manufacturers typically test a small subset of their output to verify that a sample falls within the desired performance range. If you have a critical application you will need to read the specification sheet carefully to figure out exactly what they mean by their ranges. Do they guarantee their number is a maximum, or is it, say, the $3\sigma$ error rate? Is every item tested? Is the variance normal, or some other distribution? Finally, manufacturing is not perfect. Your part might be defective and not match the performance on the sheet.

For example, I am looking at a data sheet for an airflow sensor. There is a field *Repeatability*, with the value $\pm 0.50\%$. Is this a Gaussian? Is there a bias? For example, perhaps the repeatability is nearly 0.0% at

low temperatures, and always nearly +0.50% at high temperatures. Data sheets for electrical components often contain a section of "Typical Performance Characteristics". These are used to capture information that cannot be easily conveyed in a table. For example, I am looking at a chart showing output voltage vs current for a LM555 timer. There are three curves showing the performance at different temperatures. The response is ideally linear, but all three lines are curved. This clarifies that errors in voltage outputs are probably not Gaussian - in this chip's case higher temperatures lead to lower voltage output, and the voltage output is quite nonlinear if the input current is very high.

As you might guess, modeling the performance of your sensors is one of the harder parts of creating a Kalman filter that performs well.

### 4.11.1 Animation

For those reading this in a browser here is an animation showing the filter working. If you are not using a browser you can see this plot at https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/animations/05_volt_animate.gif.

The top plot in the animation draws a green line for the predicted next voltage, then a red '+' for the actual measurement, draws a light red line to show the residual, and then draws a blue line to the filter's output. You can see that when the filter starts the corrections made are quite large, but after only a few updates the filter only adjusts its output by a small amount even when the measurement is far from it.

The lower plot shows the Gaussian belief as the filter innovates. When the filter starts the Gaussian curve is centered over 25, our initial guess for the voltage, and is very wide and short due to our initial uncertainty. But as the filter innovates, the Gaussian quickly moves to about 16.0 and becomes taller, reflecting the growing confidence that the filter has in it's estimate for the voltage. You will also note that the Gaussian's height bounces up and down a little bit. If you watch closely you will see that the Gaussian becomes a bit shorter and more spread out during the prediction step, and becomes taller and narrower as the filter incorporates another measurement.

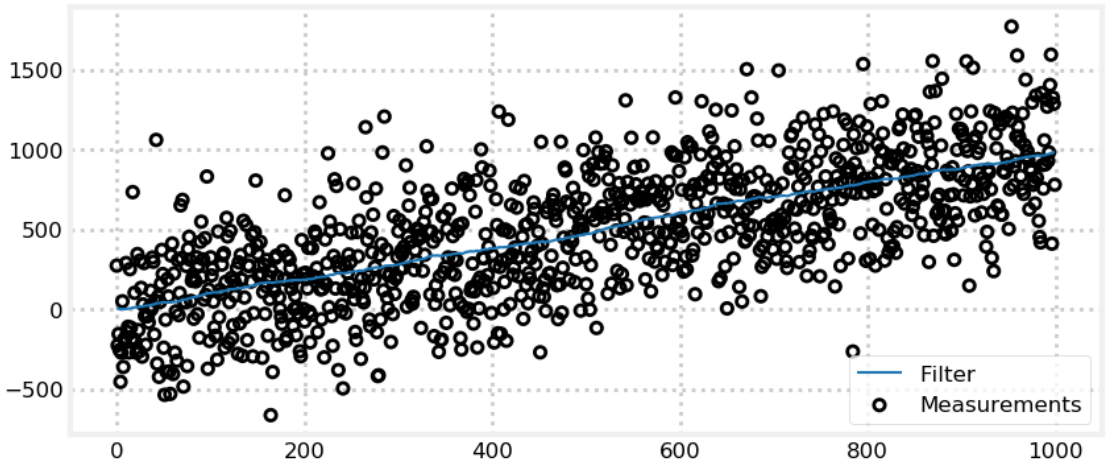
Think of this animation in terms of the g-h filter. At each step the g-h filter makes a prediction, takes a measurement, computes the residual (the difference between the prediction and the measurement), and then selects a point on the residual line based on the scaling factor $g$. The Kalman filter is doing exactly the same thing, except that the scaling factor $g$ varies with time. As the filter becomes more confident in its state the scaling factor favors the filter's prediction over the measurement.

## 4.12 Example: Extreme Amounts of Noise

With the dog filter I didn't put a lot of noise in the signal, and I 'guessed' that the dog was at position 0. How does the filter perform in real world conditions? I will start by injecting more noise in the RFID sensor while leaving the process variance at 2 $m^2$. I will inject an extreme amount of noise - noise that apparently swamps the actual measurement. What does your intuition say about the filter's performance if the sensor has a standard deviation of 300 meters? In other words, an actual position of 1.0 m might be reported as 287.9 m, or -589.6 m, or any other number in roughly that range. Think about it before you scroll down.

```
In [27]: sensor_var = 300.**2
         process_var = 2.
         process_model = gaussian(1., process_var)
         pos = gaussian(0., 500.)
         N = 1000
         dog = DogSimulation(pos.mean, 1., sensor_var, process_var)
         zs = [dog.move_and_sense() for _ in range(N)]
         ps = []

         for i in range(N):
             prior = predict(pos, process_model)
             pos = update(prior, gaussian(zs[i], sensor_var))
             ps.append(pos.mean)
```

```
book_plots.plot_measurements(zs, lw=1)
book_plots.plot_filter(ps)
plt.legend(loc=4);
```




In this example the noise is extreme yet the filter still outputs a nearly straight line! This is an astonishing result! What do you think might be the cause of this performance?

We get a nearly straight line because our process error is small. A small process error tells the filter that the prediction is very trustworthy, and the prediction is a straight line, so the filter outputs a nearly straight line.

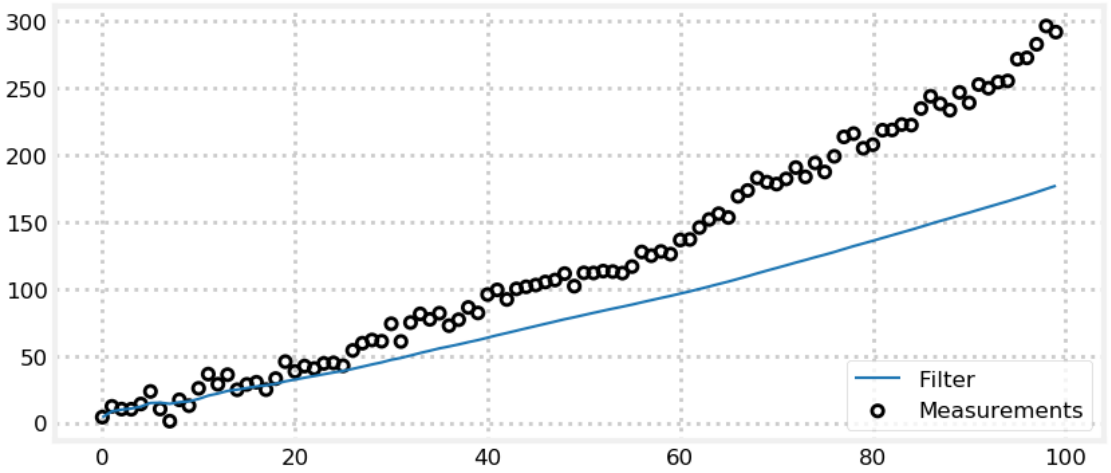
## 4.13 Example: Incorrect Process Variance

That last filter looks fantastic! Why wouldn't we set the process variance very low, as it guarantees the result will be straight and smooth?

The process variance tells the filter how much the system is changing over time. If you lie to the filter by setting this number artificially low the filter will not be able to react to changes that are happening. Let's have the dog increase his velocity by a small amount at each time step and see how the filter performs with a process variance of 0.001 $m^2$.

```
In [28]: sensor_var = 20.
         process_var = .001
         process_model = gaussian(1., process_var)
         pos = gaussian(0., 500.)
         N = 100
         dog = DogSimulation(pos.mean, 1, sensor_var, process_var*10000)
         zs, ps = [], []
         for _ in range(N):
             dog.velocity += 0.04
             zs.append(dog.move_and_sense())

         for z in zs:
             prior = predict(pos, process_model)
             pos = update(prior, gaussian(z, sensor_var))
             ps.append(pos.mean)
```

```
book_plots.plot_measurements(zs, lw=1)
book_plots.plot_filter(ps)
plt.legend(loc=4);
```

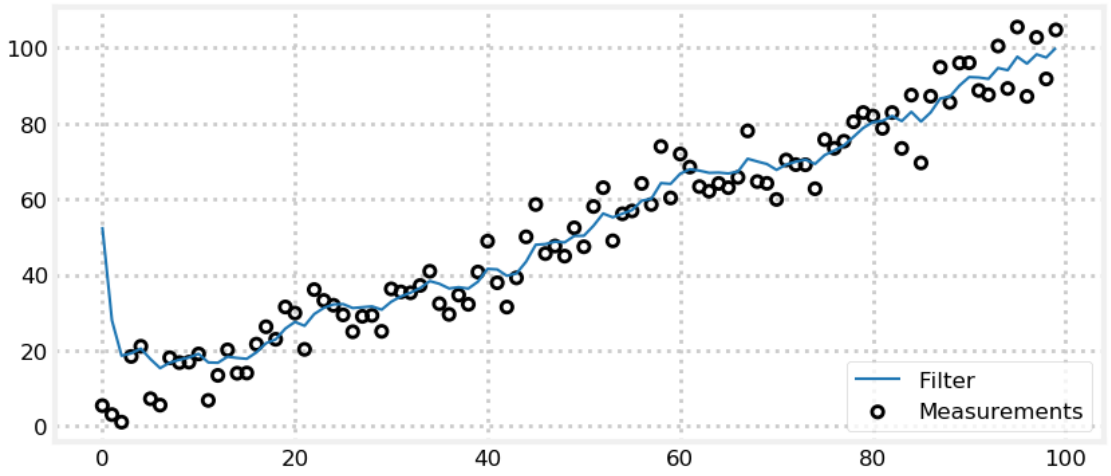



It is easy to see that the filter is not correctly responding to the measurements. The measurements clearly indicate that the dog is changing speed but the filter has been told that it's predictions are nearly perfect so it almost entirely ignores them. I encourage you to adjust the amount of movement in the dog vs process variance. We will also be studying this topic much more in the later chapters. The key point is to recognize that math requires that the variances correctly describe your system. The filter does not 'notice' that it is diverging from the measurements and correct itself. It computes the Kalman gain from the variance of the prior and the measurement, and forms the estimate depending on which is more accurate.

## 4.14 Example: Bad Initial Estimate

Now let's look at the results when we make a bad initial estimate of position. To avoid obscuring the results I'll reduce the sensor variance to 30, but set the initial position to 1000 meters. Can the filter recover from a 1000 meter error?

```
In [29]: sensor_var = 5.**2
         process_var = 2.
         pos = gaussian(1000., 500.)
         process_model = gaussian(1., process_var)
         N = 100
         dog = DogSimulation(0, 1, sensor_var, process_var)
         zs = [dog.move_and_sense() for _ in range(N)]
         ps = []

         for z in zs:
             prior = predict(pos, process_model)
             pos = update(prior, gaussian(z, sensor_var))
             ps.append(pos.mean)

         book_plots.plot_measurements(zs, lw=1)
         book_plots.plot_filter(ps)
         plt.legend(loc=4);
```

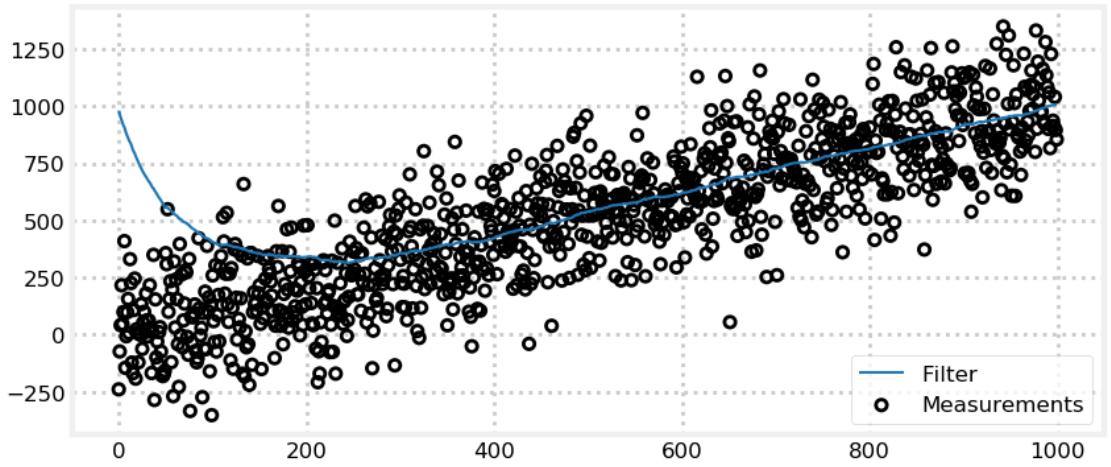



Again the answer is yes! Because we are relatively sure about our belief in the sensor ($\sigma^2 = 5^2$) after only the first step we have changed our position estimate from 1000 m to roughly 50 m. After another 5-10 measurements we have converged to the correct value. This is how we get around the chicken and egg problem of initial guesses. In practice we would likely assign the first measurement from the sensor as the initial value, but you can see it doesn't matter much if we wildly guess at the initial conditions - the Kalman filter still converges so long as the filter variances are chosen to match the actual process and measurement variances.

## 4.15 Example: Large Noise and Bad Initial Estimate

What about the worst of both worlds, large noise and a bad initial estimate?

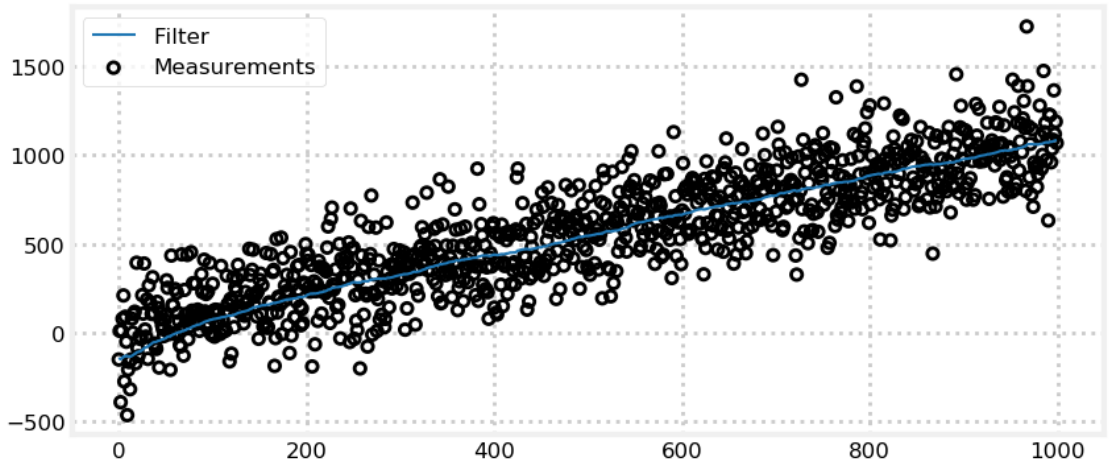
```
In [30]: sensor_var = 30000.
         process_var = 2.
         pos = gaussian(1000., 500.)
         process_model = gaussian(1., process_var)

         N = 1000
         dog = DogSimulation(0, 1, sensor_var, process_var)
         zs = [dog.move_and_sense() for _ in range(N)]
         ps = []

         for z in zs:
             prior = predict(pos, process_model)
             pos = update(prior, gaussian(z, sensor_var))
             ps.append(pos.mean)

         book_plots.plot_measurements(zs, lw=1)
         book_plots.plot_filter(ps)
         plt.legend(loc=4);
```




This time the filter struggles. Notice that the previous example only computed 100 updates, whereas this example uses 1000. By my eye it takes the filter 400 or so iterations to become reasonable accurate, but maybe over 600 before the results are good. Kalman filters are good, but we cannot expect miracles. If we have extremely noisy data and extremely bad initial conditions, this is as good as it gets.

Finally, let's implement the suggestion of using the first measurement as the initial position.

```
In [31]: sensor_var = 30000.
         process_var = 2.
         process_model = gaussian(1., process_var)
         N = 1000
         dog = DogSimulation(0, 1, sensor_var, process_var)
         zs = [dog.move_and_sense() for _ in range(N)]

         pos = gaussian(zs[0], 500.)
         ps = []
         for z in zs:
             prior = predict(pos, process_model)
             pos = update(prior, gaussian(z, sensor_var))
             ps.append(pos.mean)

         book_plots.plot_measurements(zs, lw=1)
         book_plots.plot_filter(ps)
         plt.legend(loc='best');
```




This simple change significantly improves the results. On some runs it takes 200 iterations or so to settle to a good solution, but other runs it converges very rapidly. This all depends on the amount of noise in the first measurement. A large amount of noise causes the initial estimate to be far from the dog's position.

200 iterations may seem like a lot, but the amount of noise we are injecting is truly huge. In the real world we use sensors like thermometers, laser range finders, GPS satellites, computer vision, and so on. None have the enormous errors in these examples. A reasonable variance for a cheap thermometer might be 0.2 $C^{\circ 2}$, and our code is using 30,000 $C^{\circ 2}$.

## 4.16 Exercise: Interactive Plots

Implement the Kalman filter using Jupyter Notebook's animation features to allow you to modify the various constants in real time using sliders. Refer to the section **Interactive Gaussians** in the **Gaussians** chapter to see how to do this. You will use the `interact()` function to call a calculation and plotting function. Each parameter passed into `interact()` automatically gets a slider created for it. I have written the boilerplate for this; you fill in the required code.

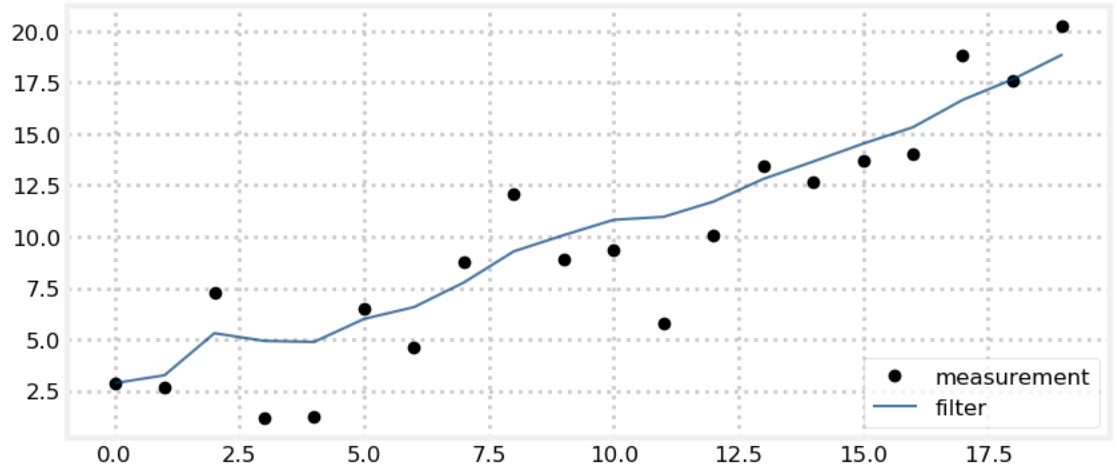
```
In [32]: from ipywidgets import interact
         from kf_book.book_plots import FloatSlider

         def plot_kalman_filter(start_pos,
                                sensor_noise,
                                velocity,
                                process_noise):
             plt.figure();
             # your code goes here

         interact(plot_kalman_filter,
                  start_pos=(-10, 10),
                  sensor_noise=FloatSlider(value=5, min=0, max=100),
                  velocity=FloatSlider(value=1, min=-2., max=2.),
                  process_noise=FloatSlider(value=5, min=0, max=100.));
```

```
<Figure size 900x400 with 0 Axes>
```

### 4.16.1 Solution

One possible solution follows. We have sliders for the start position, the amount of noise in the sensor, the amount we move in each time step, and how much movement error there is. Process noise is perhaps the least clear - it models how much the dog wanders off course at each time step, so we add that into the dog's position at each step. I set the random number generator seed so that each redraw uses the same random numbers, allowing us to compare the graphs as we move the sliders.

```
In [33]: from numpy.random import seed
         from ipywidgets import interact

         def plot_kalman_filter(start_pos,
                                sensor_noise,
                                velocity,
                                process_noise):
             N = 20
             zs, ps = [], []
             seed(303)
             dog = DogSimulation(start_pos, velocity, sensor_noise, process_noise)
             zs = [dog.move_and_sense() for _ in range(N)]
             pos = gaussian(0., 1000.) # mean and variance
             process_model = gaussian(velocity, process_noise)

             for z in zs:
                 pos = predict(pos, process_model)
                 pos = update(pos, gaussian(z, sensor_noise))
                 ps.append(pos.mean)

             plt.figure()
             plt.plot(zs, c='k', marker='o', linestyle='', label='measurement')
             plt.plot(ps, c='#004080', alpha=0.7, label='filter')
             plt.legend(loc=4);

         interact(plot_kalman_filter,
                  start_pos=(-10, 10),
                  sensor_noise=FloatSlider(value=5, min=0., max=100),
                  velocity=FloatSlider(value=1, min=-2., max=2.),
                  process_noise=FloatSlider(value=.1, min=0, max=40));
```



## 4.17 Exercise - Nonlinear Systems

Our equations for the Kalman filter are linear:

$$\mathcal{N}(\bar{\mu}, \bar{\sigma}^2) = \mathcal{N}(\mu, \sigma^2) + \mathcal{N}(\mu_{\mathtt{move}}, \sigma^2_{\mathtt{move}})$$
$$\mathcal{N}(\mu, \sigma^2) = \mathcal{N}(\bar{\mu}, \bar{\sigma}^2) \times \mathcal{N}(\mu_{\mathtt{z}}, \sigma^2_{\mathtt{z}})$$

Do you suppose that this filter works well or poorly with nonlinear systems?

Implement a Kalman filter that uses the following equation to generate the measurement value

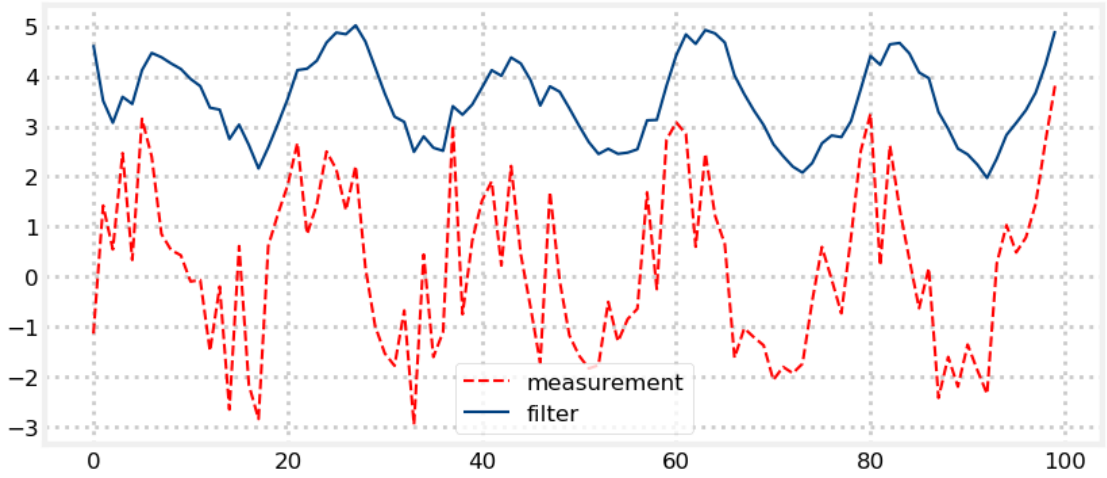
```
for i in range(100):
    z = math.sin(i/3.) * 2
```

Adjust the variance and initial positions to see the effect. What is, for example, the result of a very bad initial guess?

```
In [34]: #enter your code here.
```

### 4.17.1 Solution

```
In [35]: import math

         sensor_var = 30.
         process_var = 2.
         pos = gaussian(100., 500.)
         process_model = gaussian(1., process_var)

         zs, ps = [], []

         for i in range(100):
             pos = predict(pos, process_model)

             z = math.sin(i/3.)*2 + randn()*1.2
             zs.append(z)

             pos = update(pos, gaussian(z, sensor_var))
             ps.append(pos.mean)

         plt.plot(zs, c='r', linestyle='dashed', label='measurement')
         plt.plot(ps, c='#004080', label='filter')
         plt.legend(loc='best');
```




### 4.17.2 Discussion

This is terrible! The output is not at all like a sin wave, except in the grossest way. With linear systems we could add extreme amounts of noise to our signal and still extract a very accurate result, but here even modest noise creates a very bad result.

If we recall the **g-h Filter** chapter we can understand what is happening here. The structure of the g-h filter requires that the filter output chooses a value part way between the prediction and measurement. A varying signal like this one is always accelerating, whereas our process model assumes constant velocity, so the filter is mathematically guaranteed to always lag the input signal.

Very shortly after practitioners began implementing Kalman filters they recognized the poor performance of them for nonlinear systems and began devising ways of dealing with it. Later chapters are devoted to this problem.

## 4.18 Fixed Gain Filters

Embedded computers usually have extremely limited processors. Many do not have floating point circuitry. These simple equations can impose a heavy burden on the chip. This is less true as technology advances, but do not underestimate the value of spending one dollar less on a processor when you will be buying millions of them.

In the example above the variance of the filter converged to a fixed value. This will always happen if the variance of the measurement and process is a constant. You can take advantage of this fact by running simulations to determine what the variance converges to. Then you can hard code this value into your filter. So long as you initialize the filter to a good starting guess (I recommend using the first measurement as your initial value) the filter will perform very well. For example, the dog tracking filter can be reduced to this:

```
def update(x, z):
    K = .13232  # experimentally derived Kalman gain
    y = z - x   # residual
    x = x + K*y # posterior
    return x

def predict(x):
    return x + vel*dt
```

I used the Kalman gain form of the update function to emphasize that we do not need to consider the variances at all. If the variances converge to a single value so does the Kalman gain.

## 4.19 FilterPy's Implementation

FilterPy implements `predict()` and `update()`. They work not only for the univariate case developed in this chapter, but the more general multivariate case that we learn in subsequent chapters. Because of this their interface is slightly different. They do not take Gaussians as tuples, but as two separately named variables.

`predict()` takes several arguments, but we will only need to use these four:

```
predict(x, P, u, Q)
```

`x` is the state of the system. `P` is the variance of the system. `u` is the movement due to the process, and `Q` is the noise in the process. You will need to used named arguments when you call `predict()` because most of the arguments are optional. The third argument to `predict()` is **not** `u`.

These may strike you as terrible variable names. They are! As I already mentioned they come from a long history of control theory, and every paper or book you read will use these names. So, we just have to get used to it. Refusing to memorize them means you will never be able to read the literature.

Let's try it for the state $\mathcal{N}(10, 3)$ and the movement $\mathcal{N}(1, 4)$. We'd expect a final position of 11 (10+1) with a variance of 7 (3+4).

```
In [36]: import filterpy.kalman as kf
         kf.predict(x=10., P=3., u=1., Q=4.)
```

```
Out[36]: (11.0, 7.0)
```

`update` also takes several arguments, but for now you will be interested in these four:

```
update(x, P, z, R)
```

As before, `x` and `P` are the state and variance of the system. `z` is the measurement, and `R` is the measurement variance. Let's perform the last predict statement to get our prior, and then perform an update:

```
In [37]: x, P = kf.predict(x=10., P=3., u=1., Q=2.**2)
         print('%.3f' % x)

         x, P = kf.update(x=x, P=P, z=12., R=3.5**2)
         print('%.3f' % x, '%.3f' % P)
```

```
11.000
11.364 4.455
```

I gave it a noisy measurement with a big variance, so the estimate remained close to the prior of 11.

One final point. I did not use the variable name `prior` for the output of the predict step. I will not use that variable name in the rest of the book. The Kalman filter equations just use **x**. Both the prior and the posterior are the estimated state of the system, the former is the estimate before the measurement is incorporated, and the latter is after the measurement has been incorporated.

## 4.20 Summary

The Kalman filter that we describe in this chapter is a special, restricted case of the more general filter we will learn next. Most texts do not discuss this one dimensional form. However, I think it is a vital stepping stone. We started the book with the g-h filter, then implemented the discrete Bayes filter, and now implemented the one dimensional Kalman filter. I have tried to show you that each of these filters use the same algorithm and reasoning. The mathematics of the Kalman filter that we will learn shortly is fairly sophisticated, and it can be difficult to understand the underlying simplicity of the filter. That sophistication comes with significant benefits: the generalized filter will markedly outperform the filters in this chapter.

This chapter takes time to assimilate. To truly understand it you will probably have to work through this chapter several times. I encourage you to change the various constants in the code and observe the results. Convince yourself that Gaussians are a good representation of a unimodal belief of the position of a dog in a hallway, the position of an aircraft in the sky, or the temperature of a chemical reaction chamber. Then convince yourself that multiplying Gaussians truly does compute a new belief from your prior belief and the new measurement. Finally, convince yourself that if you are measuring movement, that adding the Gaussians together updates your belief.

Most of all, spend enough time with the **Full Description of the Algorithm** section to ensure you understand the algorithm and how it relates to the g-h filter and discrete Bayes filter. There is just one 'trick' here - selecting a value somewhere between a prediction and a measurement. Each algorithm performs that trick with different math, but all use the same logic.

# Chapter 5

# Multivariate Gaussians

Modeling Uncertainty in Multiple Dimensions

```
In [1]: from __future__ import division, print_function
        %matplotlib inline
```

```
In [2]: #format the book
        import book_format
        book_format.set_style()
```

```
Out[2]:
```

## 5.1 Introduction

The techniques in the last chapter are very powerful, but they only work with one variable or dimension. They provide no way to represent multidimensional data, such as the position and velocity of a dog in a field. Position and velocity are related to each other, and as we learned in the g-h chapter we should never throw away information. In this chapter we learn how to describe this relationship probabilistically. Through this key insight we will achieve markedly better filter performance.

## 5.2 Multivariate Normal Distributions

We've been using Gaussians for a scalar random variable, expressed as $\mathcal{N}(\mu, \sigma^2)$. A more formal term for this is *univariate normal*, where univariate means 'one variable'. The probability distribution of the Gaussian is known as the *univariate normal distribution*.

What might a *multivariate normal distribution* be? *Multivariate* means multiple variables. Our goal is to be able to represent a normal distribution with multiple dimensions. I don't necessarily mean spatial dimensions - if we track the position, velocity, and acceleration of an aircraft in (x, y, z) that gives us a nine dimensional problem. Consider a two dimensional case. It might be the $x$ and $y$ coordinates of a robot, it might be the position and velocity of a dog on the x-axis, or milk production and feed rate at a dairy. It doesn't really matter. We can see that for $N$ dimensions, we need $N$ means, which we will arrange in a column matrix (vector) like so:

$$\mu = \begin{bmatrix}\mu_1 \\ \mu_2 \\ \vdots \\ \mu_n\end{bmatrix}$$

Let's say we believe that $x = 2$ and $y = 17$. We would have

$$\mu = \begin{bmatrix} 2 \\ 17 \end{bmatrix}$$

The next step is representing our variances. At first blush we might think we would also need N variances for N dimensions. We might want to say the variance for x is 10 and the variance for y is 4, like so.

$$\sigma^2 = \begin{bmatrix} 10 \\ 4 \end{bmatrix}$$

This is incomplete because it does not consider the more general case. In the **Gaussians** chapter we computed the variance in the heights of students. That is a measure of how the heights vary relative to each other. If all students are the same height, then the variance is 0, and if their heights are wildly different, then the variance will be large.

There is also a relationship between height and weight. In general, a taller person weighs more than a shorter person. Height and weight are *correlated*. We want a way to express not only what we think the variance is in the height and the weight, but also the degree to which they are correlated. In other words, we want to know how weight varies compared to the heights. We call that the *covariance*.

Before we can understand multivariate normal distributions we need to understand the mathematics behind correlations and covariances.

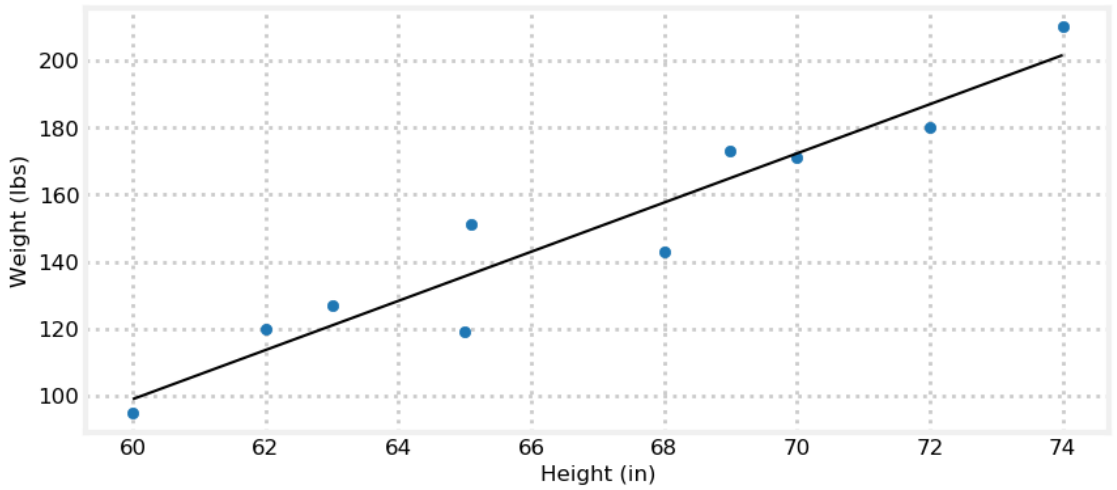
## 5.3 Correlation and Covariance

*Covariance* describes how much two variables vary together. Covariance is short for *correlated variances*. In other words, *variance* is a measure for how a population vary amongst themselves, and *covariance* is a measure for how much two variables change in relation to each other. For example, as height increases weight also generally increases. These variables are *correlated*. They are *positively correlated* because as one variable gets larger so does the other. As the outdoor temperature decreases home heating bills increase. These are *inversely correlated* or *negatively correlated* because as one variable gets larger the other variable lowers. The price of tea and the number of tail wags my dog makes have no relation to each other, and we say they are *uncorrelated* or *independent*- each can change independent of the other.

Correlation allows prediction. If you are significantly taller than me I can predict that you also weigh more than me. As winter comes I predict that I will be spending more to heat my house. If my dog wags his tail more I don't conclude that tea prices will be changing.

For example, here is a plot of height and weight of students on the school's track team. If a student is 68 inches tall I can predict they weigh roughly 160 pounds. Since the correlation is not perfect neither is my prediction.

```
In [3]: from kf_book.gaussian_internal import plot_correlated_data

        height = [60, 62, 63, 65, 65.1, 68, 69, 70, 72, 74]
        weight = [95, 120, 127, 119, 151, 143, 173, 171, 180, 210]
        plot_correlated_data(height, weight, 'Height (in)', 'Weight (lbs)', False)
```




In this book we only consider *linear correlation*. We assume that the relationship between variables is linear. That is, a straight line is a good fit for the data. I've fit a straight line through the data in the above chart. The concept of *nonlinear correlation* exists, but we will not be using it.

The equation for the covariance between $X$ and $Y$ is

$$COV(X,Y) = \sigma_{xy} = \mathbb{E}\big[(X - \mu_x)(Y - \mu_y)\big]$$

Where $\mathbb{E}[X]$ is the *expected value* of X, defined as

$$\mathbb{E}[X] = \begin{cases} \sum_{i=1}^{n} p_i x_i & \text{discrete} \\ \int_{-\infty}^{\infty} f(x)\, x & \text{continuous} \end{cases}$$

We assume each data point is equally likely, so the probability of each is $\frac{1}{N}$, giving

$$\mathbb{E}[X] = \frac{1}{N}\sum_{i=1}^{n} x_i$$

for the discrete case we will be considering.

Compare the covariance equation to the equation for the variance. As you can see they are very similar:

$$VAR(X) = \sigma_x^2 = \mathbb{E}[(X - \mu)^2]$$
$$COV(X,Y) = \sigma_{xy} = \mathbb{E}\big[(X - \mu_x)(Y - \mu_y)\big]$$

In particular, if you compute $COV(X, X)$ you get the equation for $VAR(X)$, which supports my statement that the variance computes how a random variable varies amongst itself.

We use a *covariance matrix* to denote covariances of a multivariate normal distribution, and it looks like this:

$$\Sigma = \begin{bmatrix} \sigma_1^2 & \sigma_{12} & \cdots & \sigma_{1n} \\ \sigma_{21} & \sigma_2^2 & \cdots & \sigma_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{n1} & \sigma_{n2} & \cdots & \sigma_n^2 \end{bmatrix}$$

The diagonal contains the variance for each variable, and the off-diagonal elements contain the covariance between the $i^{th}$ and $j^{th}$ variables. So $\sigma_3^2$ is the variance of the third variable, and $\sigma_{13}$ is the covariance between the first and third variables.

A covariance of 0 indicates no correlation. If the variance for $x$ is 10, the variance for $y$ is 4, and there is no linear correlation between $x$ and $y$, then we would write

$$\Sigma = \begin{bmatrix} 10 & 0 \\ 0 & 4 \end{bmatrix}$$

If there was a small amount of positive correlation between $x$ and $y$ we might have

$$\Sigma = \begin{bmatrix} 10 & 1.2 \\ 1.2 & 4 \end{bmatrix}$$

where 1.2 is the covariance between $x$ and $y$. I say the correlation is "small" because the covariance of 1.2 is small relative to the variances of 10.

If there was a large amount of negative correlation between between $x$ and $y$ we might have

$$\Sigma = \begin{bmatrix} 10 & -9.7 \\ -9.7 & 4 \end{bmatrix}$$

The covariance matrix is symmetric. After all, the covariance between $x$ and $y$ is always equal to the covariance between $y$ and $x$. That is, $\sigma_{xy} = \sigma_{yx}$ for any $x$ and $y$.

I fear I might be losing you, so let's work an example. In the **Gaussians** chapter we had a class of students with heights H=[1.8, 2.0, 1.7, 1.9, 1.6] meters. We computed:

$$\begin{aligned} VAR(H) &= E[(H - \mu_H)^2] \\ &= \frac{1}{N}\sum_{i=1}^{n}(H_i - \mu_H)^2 \\ &= \frac{1}{5}\left[(1.8-1.8)^2 + (2-1.8)^2 + (1.7-1.8)^2 + (1.9-1.8)^2 + (1.6-1.8)^2\right] \\ &= 0.02 \end{aligned}$$

Easy, right? If we weigh the students we might find their weights to be W = [70.1, 91.2, 59.5, 93.2, 53.5]. Can we use the covariance equation to create the covariance matrix? Sure. It will look like:

$$\Sigma = \begin{bmatrix} \sigma_H^2 & \sigma_{H,W} \\ \sigma_{W,H} & \sigma_W^2 \end{bmatrix}$$

We just computed the variance of the height, and it will go in the upper left hand corner of the matrix. The lower right corner contains the variance in weights. Using the same equation we get:

$$\begin{aligned} \mu_W &= \frac{1}{5}(70.1 + 91.2 + 59.5 + 93.2 + 53.5) = 73.5 \\ \sigma_W^2 &= \frac{1}{5}\left[(70.1-73.5)^2 + (91.2-73.5)^2 + (59.5-73.5)^2 + (93.2-73.5)^2 + (53.5-73.5)^2\right] \\ &= 261.8 \end{aligned}$$

Now the covariances. Using the formula above, we compute:

$$\begin{aligned} \sigma_{H,W} &= \mathbb{E}\big[(H - \mu_H)(W - \mu_W)\big] \\ &= \frac{1}{N}\sum_{i=1}^{n}(H_i - \mu_H)(W_i - \mu_W) \\ &= \frac{1}{5}[(1.8-1.8)(70.1-73.5) + (2-1.8)(91.2-73.5) + (1.7-1.8)(59.5-73.5) + \\ &\quad (1.9-1.8)(93.2-73.5) + (1.6-1.8)(53.5-73.5)] \\ &= 2.18 \end{aligned}$$

That was tedious, but easy enough. We will never do that again because, of course, NumPy will compute it for you.

```
In [4]: import numpy as np

        W = [70.1, 91.2, 59.5, 93.2, 53.5]
        H = [1.8, 2.0, 1.7, 1.9, 1.6]
        np.cov(H, W)

Out[4]: array([[  0.025,   2.727],
               [  2.727, 327.235]])
```

That doesn't agree with our calculation! What went wrong? Nothing. NumPy applies a correction for small sample sizes; it uses $\frac{1}{N-1}$ as the normalization term instead of $\frac{1}{N}$.

This is a bit beyond the scope of this book. Briefly, suppose the actual class size is 200 students, and we took a sample of 5 students to perform this computation because we couldn't afford to measure and weigh all 200 students. It is nearly certain that there will be some error in our estimator because the sample is unlikely to perfectly represent the class. As our sample size approaches 200 the error will approach 0. We say there is no *bias* in the latter, and that we have an *unbiased estimator*. In contrast, when we take a small sample there is bias (error is nonzero), and we have a *biased estimator*.

If the error is zero it makes sense to divide by $N$. I will not prove why, but for biased estimators we use $\frac{1}{N-1}$ to correct for the small sample size. NumPy does this by default because in practice we are almost always working from data samples from a larger collection. If you want the unbiased estimator, which we computed above, use `bias=1` in the call to 'np.cov'.

```
In [5]: np.cov(H, W, bias=1)

Out[5]: array([[  0.02 ,   2.182],
               [  2.182, 261.788]])
```

This agrees with our computation. We will not use `bias=1` again in this book since we are using *random variables* which are sampling from the infinite set of positions of the objects we are tracking. Here we are computing the variance and covariance for the entire population, so `bias=1` is correct.

What does this matrix tell us? It tells us the variance in heights is 0.02 $m^2$ and the variance in weights is 261.788 $kg^2$. Furthermore, it tells us the weights and heights are positively correlated - as heights increase so do the weights.

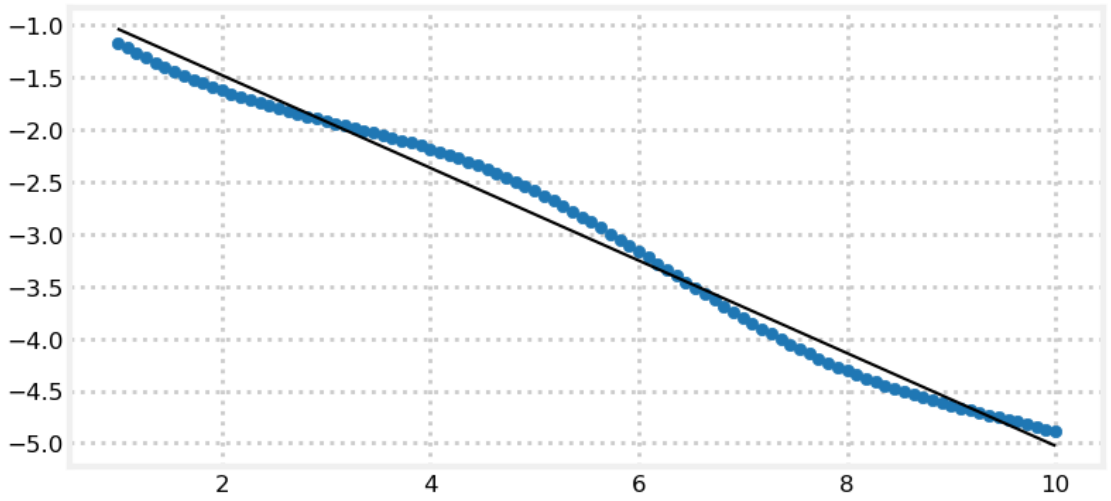
Let's create perfectly correlated data. By this I mean that the data perfectly fits on a line - there is no variance from the line.

```
In [6]: X = np.linspace(1, 10, 100)
        Y = np.linspace(1, 10, 100)
        np.cov(X, Y)

Out[6]: array([[6.956, 6.956],
               [6.956, 6.956]])
```

We can see from the covariance matrix that the covariance is equal to the variance in x and in y.

Now let's add some noise to one of the variables so that they are no longer perfectly correlated. I will make $Y$ negative to create a negative correlation.

```
In [7]: X = np.linspace(1, 10, 100)
        Y = -(np.linspace(1, 5, 100) + np.sin(X)*.2)
        plot_correlated_data(X, Y)
        print(np.cov(X, Y))
```

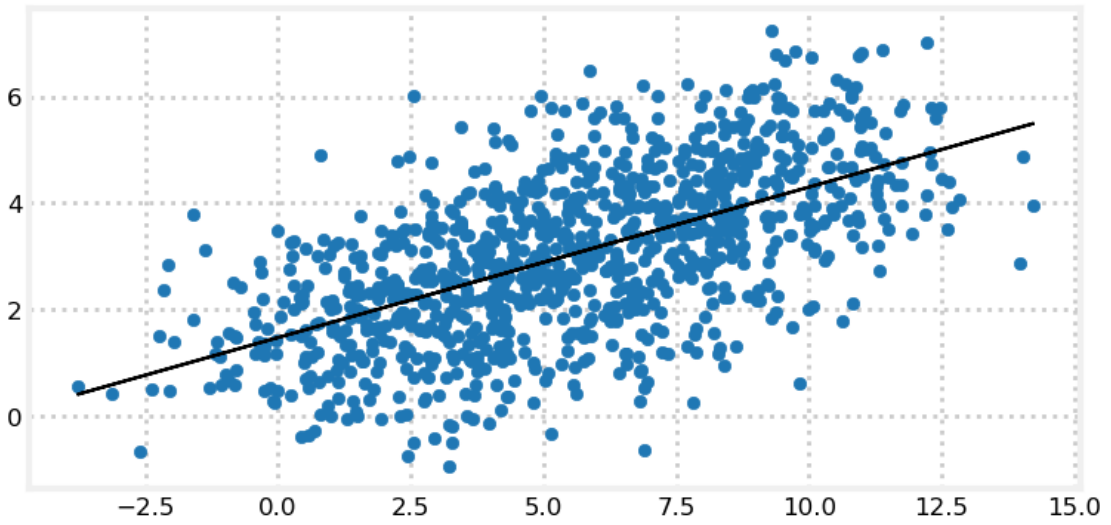



```
[[ 6.956 -3.084]
 [-3.084  1.387]]
```

The data no longer forms a straight line. The covariance is $\sigma_{xy} = -3.08$. It is not close to zero compared to the magnitudes of $\sigma_x^2$ and $\sigma_y^2$, and so we know there is still a high degree of correlation. We can verify this by looking at the chart. The data forms nearly a straight line.

Now I will add random noise to a straight line.

```
In [8]: from numpy.random import randn
        X = np.linspace(1, 10, 1000) + randn(1000)*2
        Y = np.linspace(1, 5, 1000) + randn(1000)
        plot_correlated_data(X, Y)
        print(np.cov(X, Y))
```

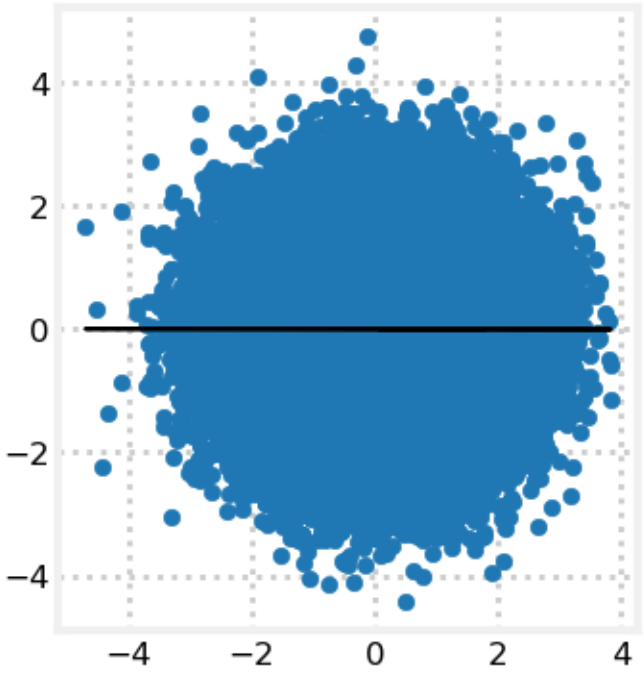


```
[[10.834  3.064]
 [ 3.064  2.406]]
```

We see that the covariance is smaller in relation to the variances, reflecting the lower correlation between $X$ and $Y$. We can still fit a straight line through this data, but there is much greater variation in the data.

Finally, here is the covariance between completely random data.

```
In [9]: X = randn(100000)
        Y = randn(100000)
        plot_correlated_data(X, Y)
        print(np.cov(X, Y))
```




```
[[ 1.001 -0.001]
 [-0.001  1.   ]]
```

Here the covariances are very near zero. As you can see with the plot, there is no clear way to draw a line to fit the data. A vertical line would be as unconvincing as the horizontal line I've shown.

## 5.4 Multivariate Normal Distribution Equation

Recall the equation for the normal distribution from the **Gaussians** chapter:

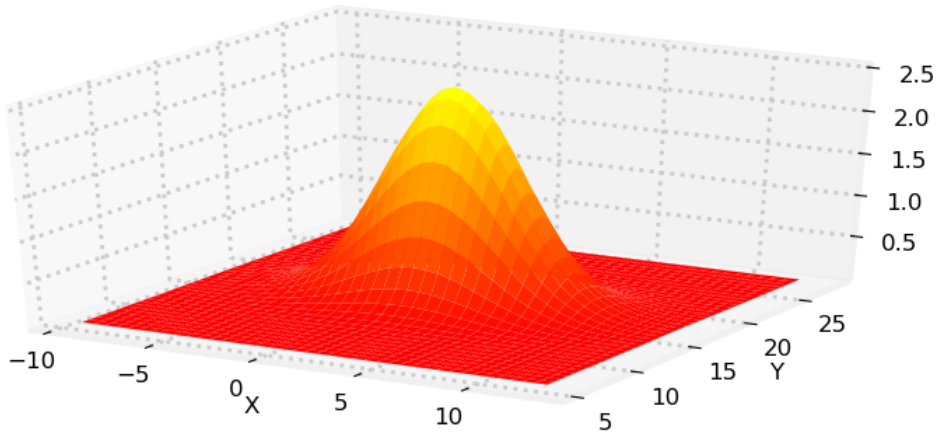
$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[-\frac{1}{2}(x-\mu)^2/\sigma^2\right]$$

Here is the multivariate normal distribution in $n$ dimensions.

$$f(\mathbf{x}, \mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^n|\Sigma|}} \exp\left[-\frac{1}{2}(\mathbf{x}-\mu)^\mathsf{T}\Sigma^{-1}(\mathbf{x}-\mu)\right]$$

The multivariate version merely replaces the scalars of the univariate equations with matrices. If you are reasonably well-versed in linear algebra this equation should look quite manageable. If not, don't worry, both FilterPy and SciPy provide functions to compute it for you. Let's ignore the computation for a moment and plot it to see what it looks like.

```
In [10]: import kf_book.mkf_internal as mkf_internal

         mean = [2., 17.]
         cov = [[10., 0.],
                [0., 4.]]

         mkf_internal.plot_3d_covariance(mean, cov)
```




This is a plot of multivariate Gaussian with a mean of $\mu = \begin{bmatrix} 2 \\ 17 \end{bmatrix}$ and a covariance of $\Sigma = \begin{bmatrix} 10 & 0 \\ 0 & 4 \end{bmatrix}$. The three dimensional shape shows the probability density for any value of $(X, Y)$ in the z-axis. I have projected the variance for x and y onto the walls of the chart - you can see that they take on the Gaussian bell curve shape. The curve for $X$ is wider than the curve for $Y$, which is explained by $\sigma_x^2 = 10$ and $\sigma_y^2 = 4$. The highest point of the 3D surface is at the the means for $X$ and $Y$.

All multivariate Gaussians have this shape. If we think of this as the Gaussian for the position of a dog, the z-value at each point of $(X, Y)$ is the probability density of the dog being at that position. Strictly speaking this is the *joint probability density function*, which I will define soon. So, the dog has the highest probability of being near (2, 17), a modest probability of being near (5, 14), and a very low probability of being near (10, 10). As with the univariate case this is a *probability density*, not a *probability*. Continuous distributions have an infinite range, and so the probability of being exactly at (2, 17), or any other point, is 0%. We can compute the probability of being within a given range by computing the volume under the surface with an integral.

FilterPy [2] implements the equation with the function `multivariate_gaussian()` in the `filterpy.stats.` module. SciPy's `stats` module implements the multivariate normal equation with `multivariate_normal()`. It implements a 'frozen' form where you set the mean and covariance once, and then calculate the probability density for any number of values for x over any arbitrary number of calls. I named my function `multivariate_gaussian()` to ensure it is never confused with the SciPy version.

> The tutorial[1] for the `scipy.stats` module explains 'freezing' distributions and other very useful features.

```
In [11]: from filterpy.stats import gaussian, multivariate_gaussian
```

I'll demonstrate using it, and then move on to more interesting things.

First, let's find the probability density for our dog being at (2.5, 7.3) if we believe he is at (2, 7) with a variance of 8 for $x$ and a variance of 3 for $y$.

Start by setting $x$ to (2.5, 7.3). You can use a tuple, list, or NumPy array.

```
In [12]: x = [2.5, 7.3]
```

Next, we set the mean of our belief:

```
In [13]: mu = [2.0, 7.0]
```

Finally, we have to define our covariance matrix. In the problem statement we did not mention any correlation between $x$ and $y$, and we will assume there is none. This makes sense; a dog can choose to independently wander in either the $x$ direction or $y$ direction without affecting the other. I will use the variable name `P`. Kalman filters use the name $\mathbf{P}$ for the covariance matrix, and we need to become familiar with the conventions.

```
In [14]: P = [[8., 0.],
              [0., 3.]]
```

Now call the function

```
In [15]: %precision 4
         multivariate_gaussian(x, mu, P)
```

```
Out[15]: 0.0315
```

We can get the same result from the `scipy.stats` module.

```
In [16]: import scipy
         try:
             print('{:.4f}'.format(scipy.stats.multivariate_normal(mu, P).pdf(x)))
         except:
             print('you have an old version of scipy, upgrade it now!')
```

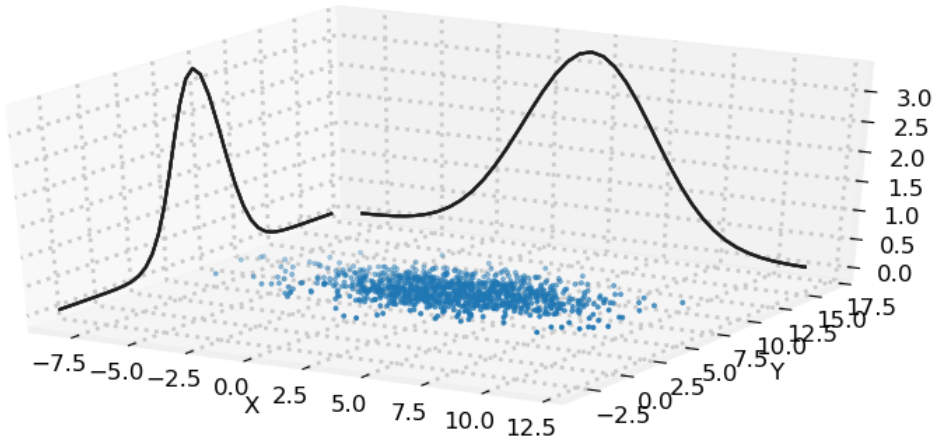
```
0.0315
```

It's time to define some terms. The *joint probability*, denoted $P(x, y)$, is the probability of both $x$ and $y$ happening. For example, if you roll two die $P(2, 5)$ is the probability of the first die rolling a 2 and the second die rolling a 5. Assuming the die are six sided and fair, the probability $P(2, 5) = \frac{1}{6} \times \frac{1}{6} = \frac{1}{36}$. The 3D chart above shows the *joint probability density function*.

The *marginal probability* is the probability of an event happening without regard of any other event. In the chart above the Gaussian curve drawn to the left is the marginal for $Y$. This is the probability for the dog being at any position in $Y$ disregarding the value for $X$. Earlier I wrote "I have projected the variance for x and y onto the walls of the chart"; these are the marginal probabilities for $x$ and $y$. Another computational benefit of Gaussians is that the marginal of a multivariate Gaussian is another Gaussian!

Let's look at this in a slightly different way. Instead of plotting a surface showing the probability distribution I will generate 1,000 points with the distribution of $\left[\begin{smallmatrix} 8 & 0 \\ 0 & 3 \end{smallmatrix}\right]$.

```
In [17]: mkf_internal.plot_3d_sampled_covariance(mu, P)
```



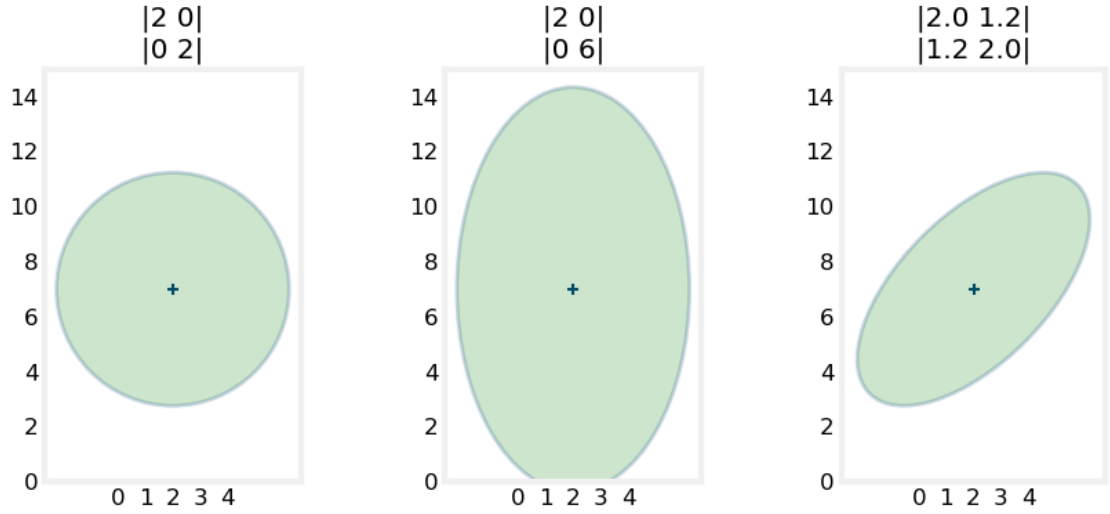

We can think of the sampled points as being possible locations for our dog given those particular mean and covariances. The contours on the side show the marginal probability for $X$ and $Y$. We can see that he is far more likely to be at (2, 7) where there are many points, than at (-5, 5) where there are few.

As beautiful as these plots are, it is hard to get useful information from them. For example, it is not easy to tell if $X$ and $Y$ both have the same variance, and how much they are correlated. In most of the book I'll display Gaussians as contour plots.

The contour plots display the range of values that the multivariate Gaussian takes for a specific standard deviation. This is like taking a horizontal slice out of the 3D plot.

These plots show the shape of the slice for 3 standard deviations.

```
In [18]: mkf_internal.plot_3_covariances()
```



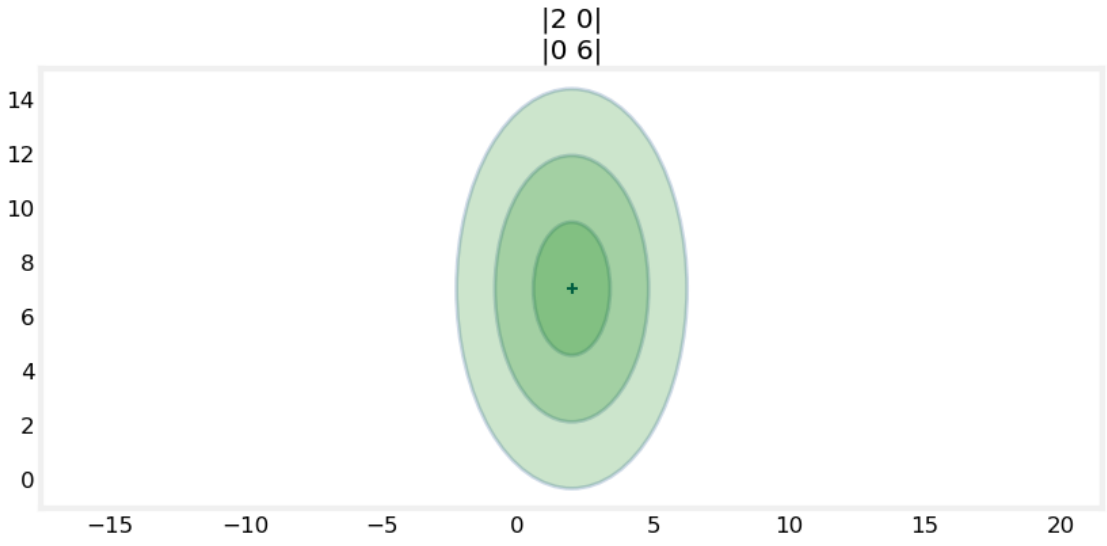

For those of you viewing this online or in Juptyer Notebook on your computer, here is an animation of varying the covariance while holding the variance constant.

(source: http://git.io/vqxLS)

These plots look like circles and ellipses. Indeed, it turns out that any slice through the multivariate Gaussian is an ellipse. Hence, in statistics we do not call these 'contour plots', but either *error ellipses* or *confidence ellipses*; the terms are interchangable.

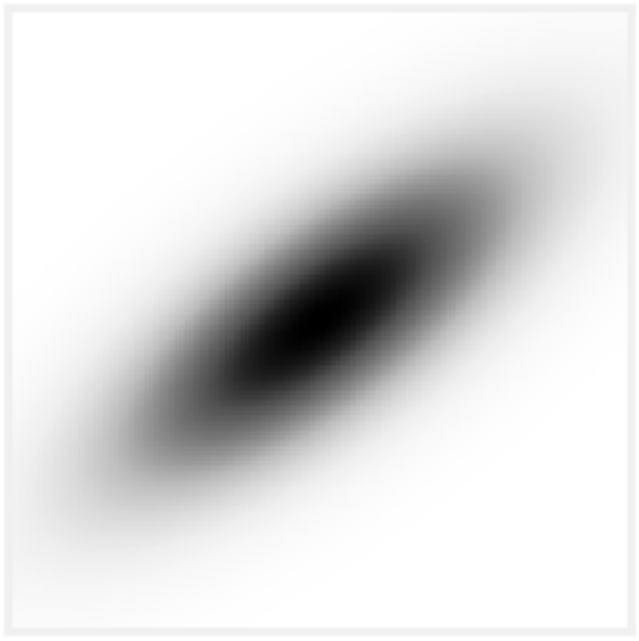
This code uses the function `plot_covariance_ellipse()` from `filterpy.stats`. By default the function displays one standard deviation, but you can use either the `variance` or `std` parameter to control what is displayed. For example, `variance=3**2` or `std=3` would display the 3rd standard deviation, and `variance=[1,4,9]` or `std=[1,2,3]` would display the 1st, 2nd, and 3rd standard deviations.

```
In [19]: from filterpy.stats import plot_covariance_ellipse
         import matplotlib.pyplot as plt

         P = [[2, 0], [0, 6]]
         plot_covariance_ellipse((2, 7), P, fc='g', alpha=0.2,
                                 std=[1, 2, 3],
                                 title='|2 0|\n|0 6|')
         plt.gca().grid(b=False);
```




The solid colors may suggest to you that the probability distribution is constant between the standard deviations. This is not true, as you can tell from the 3D plot of the Gaussian. Here is a 2D shaded representation of the probability distribution for the covariance $\left(\begin{smallmatrix} 2 & 1.2 \\ 1.2 & 1.3 \end{smallmatrix}\right)$. Darker gray corresponds to higher probability density.

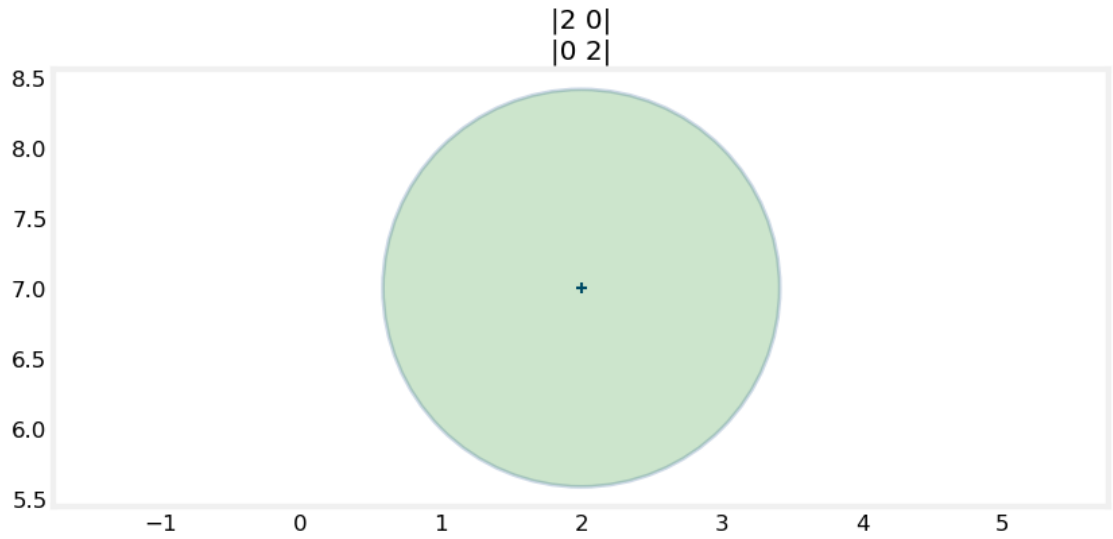
```
In [20]: from kf_book.nonlinear_plots import plot_cov_ellipse_colormap
         plot_cov_ellipse_colormap(cov=[[2, 1.2], [1.2, 1.3]]);
```



Thinking about the physical interpretation of these plots clarifies their meaning. The mean and covariance of the first plot is

$$\mu = \begin{bmatrix}2\\7\end{bmatrix},\ \Sigma = \begin{bmatrix}2 & 0\\0 & 2\end{bmatrix}$$

```
In [21]: x = [2, 7]
         P = [[2, 0], [0, 2]]
         plot_covariance_ellipse(x, P, fc='g', alpha=0.2,
                                 title='|2 0|\n|0 2|')
         plt.gca().grid(b=False)
```

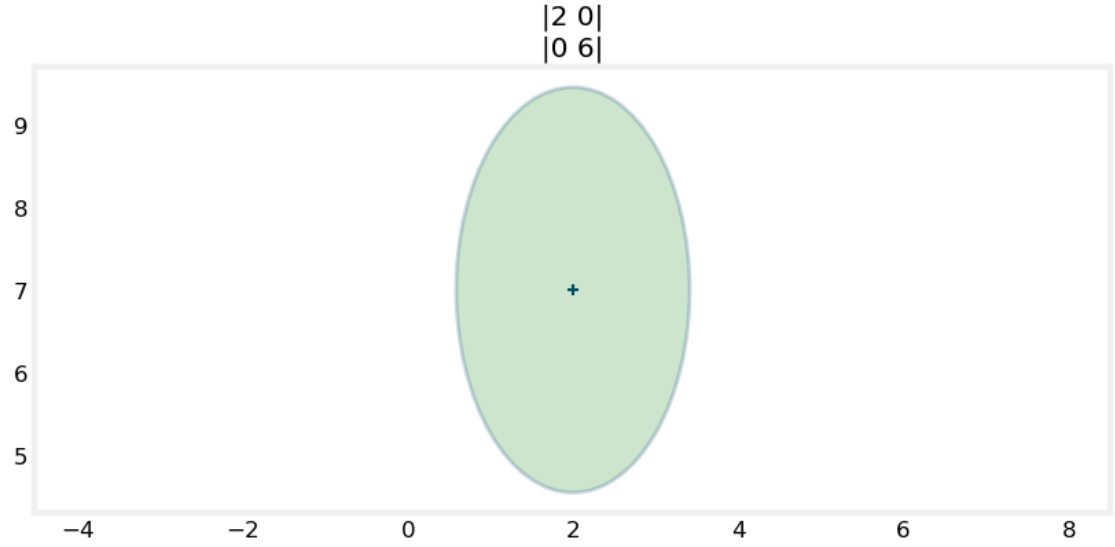


A Bayesian way of thinking about this is that the ellipse shows us the amount of error in our belief. A tiny circle would indicate that we have a very small error, and a very large circle indicates a lot of error in our belief. The shape of the ellipse shows us the geometric relationship of the errors in $X$ and $Y$. Here we have a circle so errors in $X$ and $Y$ are equally likely.

The mean and covariance of the second plot are

$$\mu = \begin{bmatrix}2\\7\end{bmatrix},\ \Sigma = \begin{bmatrix}2 & 0\\0 & 6\end{bmatrix}$$

```
In [22]: x = [2, 7]
         P = [[2, 0], [0, 6]]
         plot_covariance_ellipse(x, P, fc='g', alpha=0.2,
                                 title='|2 0|\n|0 6|')
         plt.gca().grid(b=False)
```

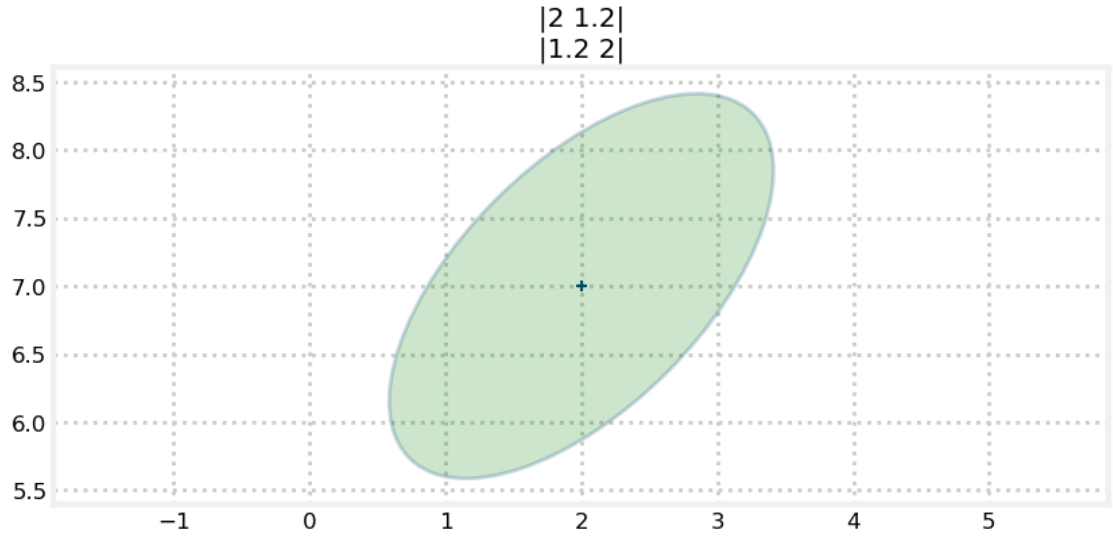



This time we use a different variance for $X$ ($\sigma_x^2 = 2$) vs $Y$ ($\sigma_y^2 = 6$). The result is a tall and narrow ellipse. We can see that a lot more uncertainty in $Y$ vs $X$. In both cases we believe the dog is at (2, 7), but the uncertainties are different.

The third plot shows the mean and covariance

$$\mu = \begin{bmatrix}2\\7\end{bmatrix},\ \Sigma = \begin{bmatrix}2 & 1.2\\1.2 & 2\end{bmatrix}$$

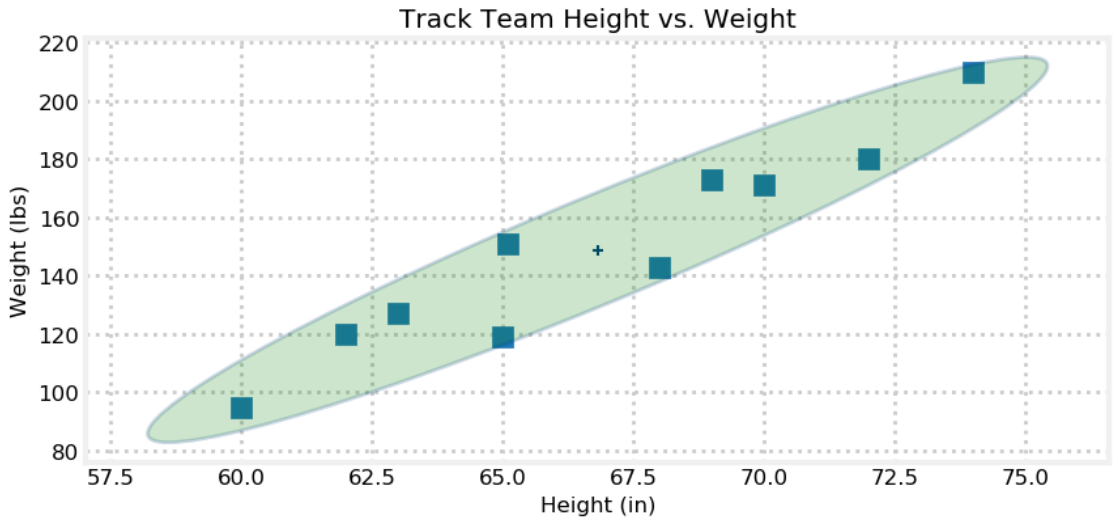
```
In [23]: x = [2, 7]
         P = [[2, 1.2], [1.2, 2]]
         plot_covariance_ellipse(x, P, fc='g', alpha=0.2,
                                 title='|2 1.2|\n|1.2 2|')
```




This is the first contour that has values in the off-diagonal elements of the covariance, and this is the first contour plot with a slanted ellipse. This is not a coincidence. The two facts are telling us the same thing. A slanted ellipse tells us that the $x$ and $y$ values are somehow correlated. The off-diagonal elements in the covariance matrix are non-zero, indicating that a correlation exists.

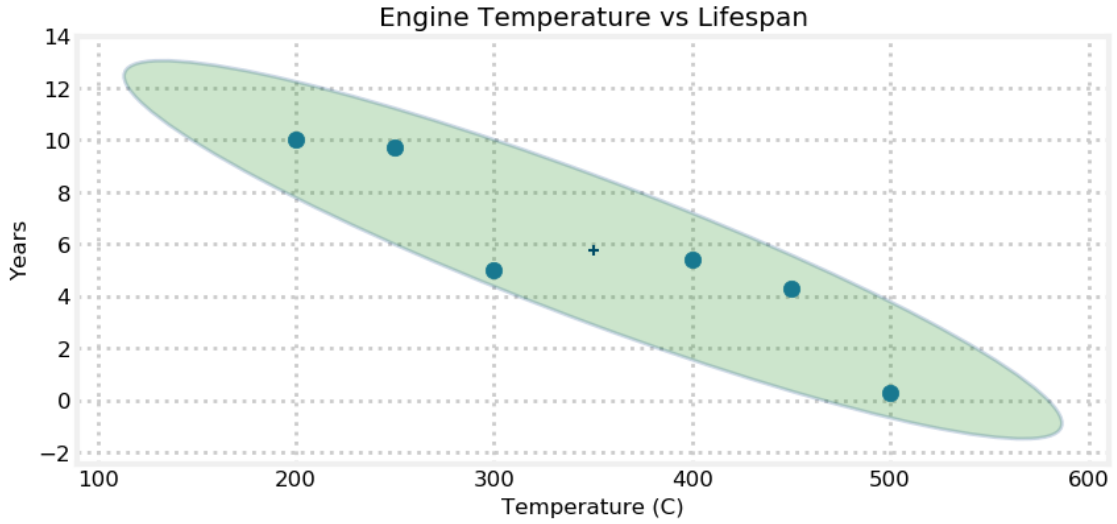
Recall the plot for height versus weight. It formed a slanted grouping of points. We can use NumPy's `cov()` function to compute the covariance of two or more variables by placing them into a 2D array. Let's do that, then plot the $2\sigma$ covariance ellipse on top of the data. We will need to use `bias=1` because the data represents the entire population; it is not a sample.

```
In [24]: cov_hw = np.cov(np.vstack((height, weight)), bias=1)
         cov_hw

Out[24]: array([[  18.5249,  135.701 ],
                [ 135.701 , 1092.29  ]])

In [25]: plt.scatter(height, weight, s=120, marker='s')
         plt.title('Track Team Height vs. Weight')
         plt.xlabel('Height (in)'); plt.ylabel('Weight (lbs)')
         plot_covariance_ellipse((np.mean(height), np.mean(weight)), cov_hw, fc='g',
                                 alpha=0.2, axis_equal=False, std=2)
```



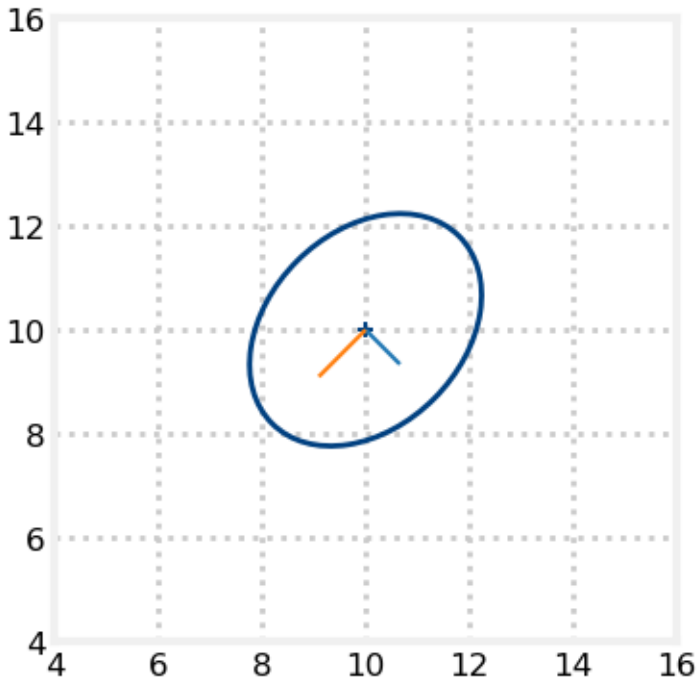

This should help you form a strong intuition on the meaning and use of covariances. The covariance ellipse shows you how the data is 'scattered' in relation to each other. A narrow ellipse like this tells you that the data is very correlated. There is only a narrow range of weights for any given height. The ellipse leans towards the right, telling us there is a positive correlation - as x increases y also increases. If the ellipse leaned towards the left then the correlation would be negative - as x increases y decreases. We can see this in the following plot:

```
In [26]: max_temp = [200, 250, 300, 400, 450, 500]
         lifespan = [10, 9.7, 5, 5.4, 4.3, 0.3]

         plt.scatter(max_temp, lifespan, s=80)
         cov = np.cov(np.vstack((max_temp, lifespan)))
         plot_covariance_ellipse((np.mean(max_temp), np.mean(lifespan)), cov, fc='g',
                     alpha=0.2, axis_equal=False, std=2)
         plt.title('Engine Temperature vs Lifespan')
         plt.xlabel('Temperature (C)'); plt.ylabel('Years');
```



The relationships between variances and covariances can be hard to puzzle out by inspection, so here is an interactive plot. (If you are reading this in a static form instructions to run this online are here: https://git.io/vza7b)

```
In [27]: from ipywidgets import interact
         from kf_book.book_plots import figsize, FloatSlider

         fig = None
         def plot_covariance(var_x, var_y, cov_xy):
             global fig
             if fig: plt.close(fig)
             fig = plt.figure(figsize=(4,4))
             P1 = [[var_x, cov_xy], [cov_xy, var_y]]

             plot_covariance_ellipse((10, 10), P1, axis_equal=False,
                                     show_semiaxis=True)

             plt.xlim(4, 16)
             plt.gca().set_aspect('equal')
             plt.ylim(4, 16)


         with figsize(y=6):
             interact (plot_covariance,
                   var_x=FloatSlider(5, min=0, max=20),
                   var_y=FloatSlider(5, min=0, max=20),
                   cov_xy=FloatSlider(1.5, min=0, max=50, step=.2));
```



### 5.4.1 Pearson's Correlation Coefficient

We will not be using this coefficient in this book, but you may see it elsewhere. You can safely skip this section if uninterested.

The correlation between two variables can be given a numerical value with *Pearson's Correlation Coefficient*. It is defined as

$$\rho_{xy} = \frac{COV(X,Y)}{\sigma_x\sigma_y}$$

This value can range in value from -1 to 1. If the covariance is 0 than $\rho = 0$. A value greater than 0 indicates that the relationship is a positive correlation, and a negative value indicates that there is a negative correlation. Values near -1 or 1 indicate a very strong correlation, and values near 0 indicate a very weak correlation.

Correlation and covariance are very closely related. Covariance has units associated with it, and correlation is a unitless ratio. For example, for our dog $\sigma_{xy}$ has units of meters squared.

We can use `scipy.stats.pearsonr` function to compute the Pearson coefficient. It returns a tuple of the Pearson coefficient and of the 2 tailed p-value. The latter is not used in this book. Here we compute $\rho$ for height vs weight of student athletes:

```
In [28]: from scipy.stats import pearsonr
         pearsonr(height, weight)[0]
```

```
Out[28]: 0.9539731096080194
```

Here we compute the correlation between engine temperature and lifespan.

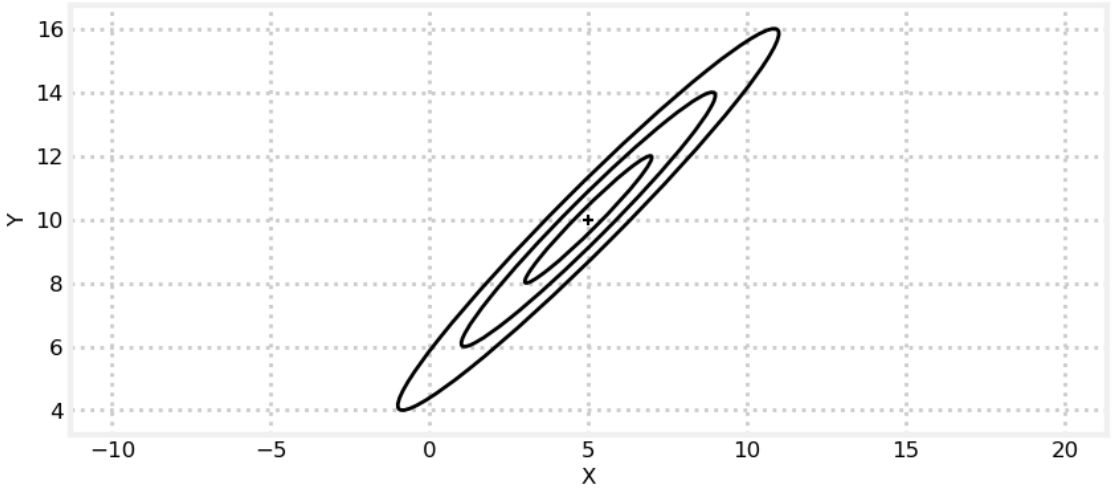
```
In [29]: pearsonr(max_temp, lifespan)[0]
```

```
Out[29]: -0.9178223453527254
```

## 5.5 Using Correlations to Improve Estimates

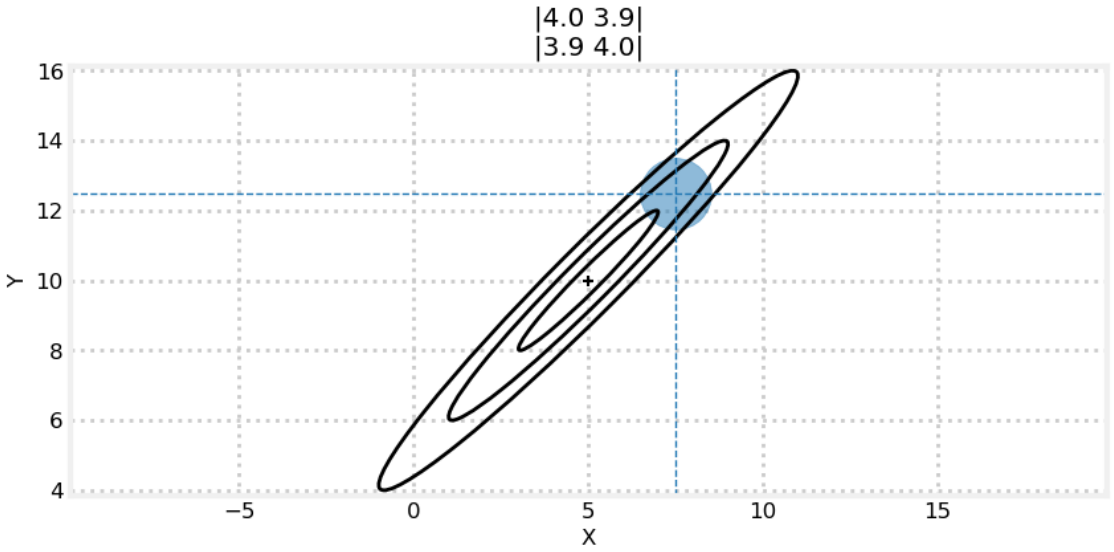
Suppose we believe our dog is at position (5, 10) with some given covariance. If the standard deviation in x and y is each 2 meters, but they are strongly correlated, the covariance contour would look something like this.

```
In [30]: P = [[4, 3.9], [3.9, 4]]

         plot_covariance_ellipse((5, 10), P, ec='k', std=[1, 2, 3])
         plt.xlabel('X')
         plt.ylabel('Y');
```



Now suppose I were to tell you that we know that $x = 7.5$. What can we infer about the value for $y$? The position is extremely likely to lie within the $3\sigma$ covariance ellipse. We can infer the position in $y$ based on the covariance matrix because there is a correlation between $x$ and $y$. I've illustrated the likely range of values for y as a blue filled circle.

```
In [31]: mkf_internal.plot_correlation_covariance()
```



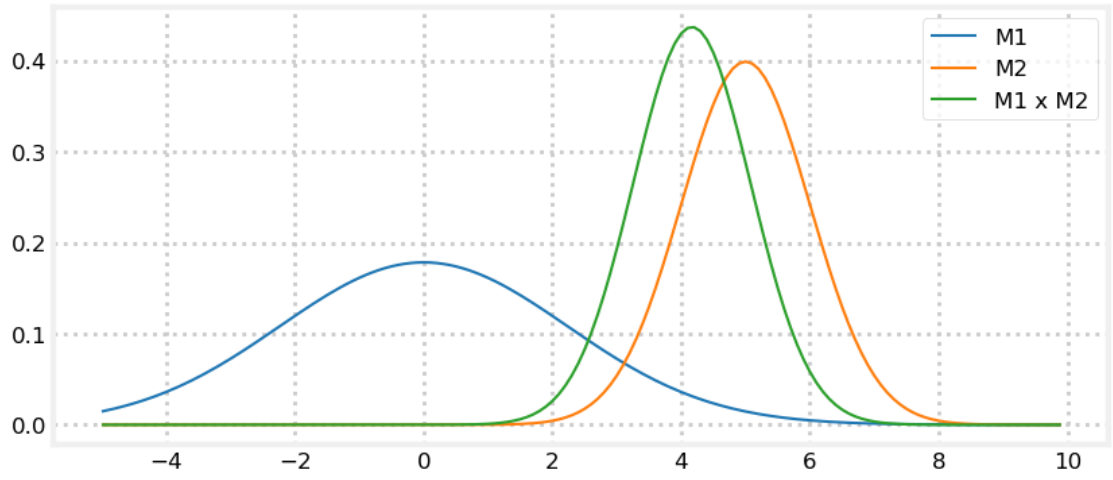

The circle not mathematically correct, but it gets the idea across. We will tackle the mathematics in the next section. For now recognize that we can predict that $y$ is likely near 12. A value of $y = -10$ is extremely improbable.

A word about *correlation* and *independence*. If variables are *independent* they can vary separately. If you walk in an open field, you can move in the $x$ direction (east-west), the $y$ direction(north-south), or any combination thereof. Independent variables are always also *uncorrelated*. Except in special cases, the reverse does not hold true. Variables can be uncorrelated, but dependent. For example, consider $y = x^2$. Correlation is a linear measurement, so $x$ and $y$ are uncorrelated. However, $y$ is dependent on $x$.

## 5.6 Multiplying Multidimensional Gaussians

In the previous chapter we incorporated an uncertain measurement with an uncertain estimate by multiplying their Gaussians together. The result was another Gaussian with a smaller variance. If two pieces of uncertain information corroborate each other we should be more certain in our conclusion. The graphs look like this:

```
In [32]: mkf_internal.plot_gaussian_multiply()
```




The combination of measurements 1 and 2 yields more certainty, so the new Gaussian is taller and narrower - the variance became smaller. The same happens in multiple dimensions with multivariate Gaussians.

Here are the equations for multiplying multivariate Gaussians. The capital sigma ($\Sigma$) indicates that these are matrices, not scalars. Specifically, they are covariance matrices:

$$\mu = \Sigma_2(\Sigma_1 + \Sigma_2)^{-1}\mu_1 + \Sigma_1(\Sigma_1 + \Sigma_2)^{-1}\mu_2$$
$$\Sigma = \Sigma_1(\Sigma_1 + \Sigma_2)^{-1}\Sigma_2$$

They are generated by plugging the multivariate Gaussians for the prior and the estimate into Bayes Theorem. I gave you the algebra for the univariate case in the **Gaussians** chapter.

You will not need to remember these equations as they are computed by Kalman filter equations that will be presented shortly. This computation is also available in FilterPy using the `multivariate_multiply()` method, which you can import from `filterpy.stats`.

To give you some intuition about this, recall the equations for multiplying univariate Gaussians:

$$\mu = \frac{\sigma_1^2\mu_2 + \sigma_2^2\mu_1}{\sigma_1^2 + \sigma_2^2},$$
$$\sigma^2 = \frac{\sigma_1^2\sigma_2^2}{\sigma_1^2 + \sigma_2^2}$$

This looks similar to the equations for the multivariate equations. This will be more obvious if you recognize that matrix inversion, denoted by the -1 power, is *like* a reciprocal since $AA^{-1} = I$. I will rewrite the inversions as divisions - this is not a mathematically correct thing to do as division for matrices is not defined, but it does help us compare the equations.

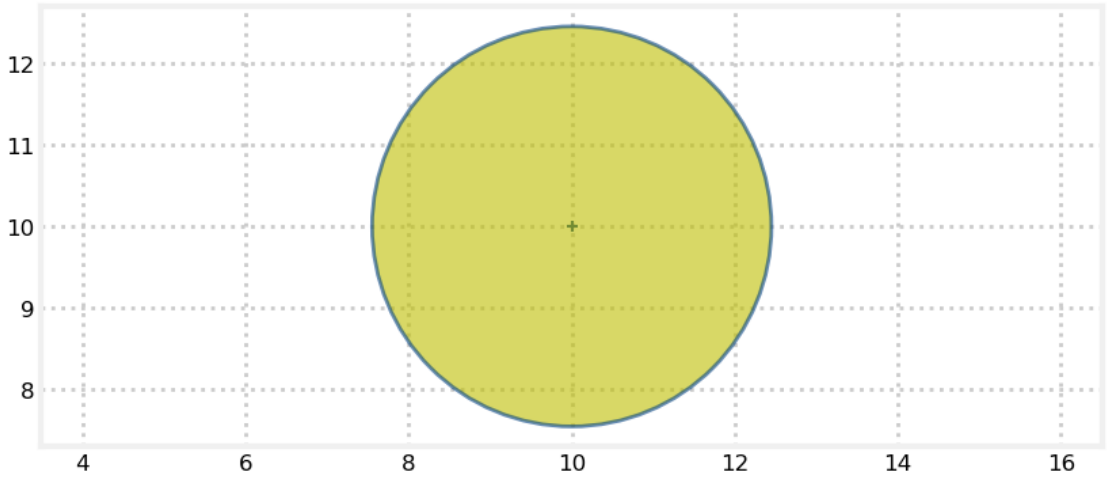
$$\mu \approx \frac{\Sigma_2\mu_1 + \Sigma_1\mu_2}{\Sigma_1 + \Sigma_2}$$

$$\Sigma \approx \frac{\Sigma_1\Sigma_2}{(\Sigma_1 + \Sigma_2)}$$

In this form the relationship between the univariate and multivariate equations is clear.

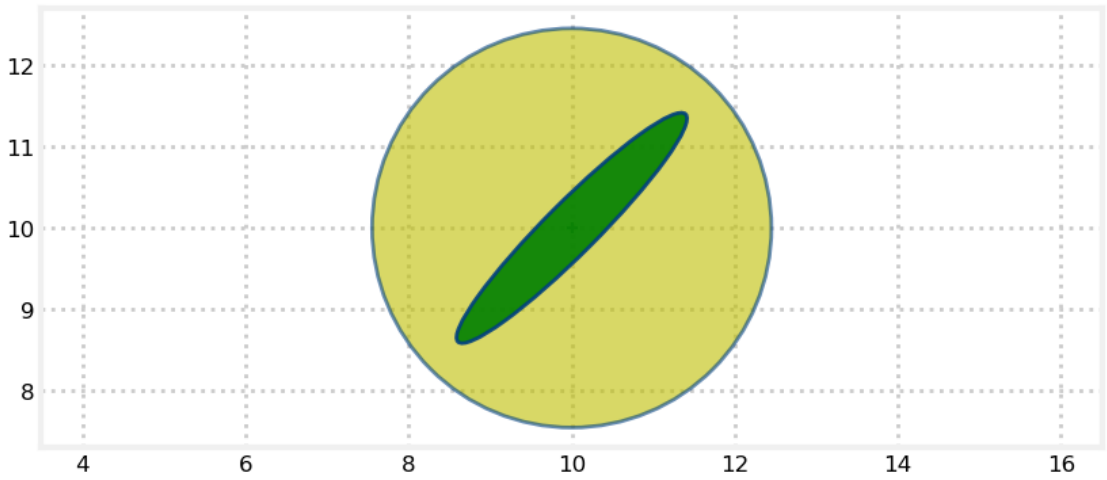
Now let's explore multivariate Gaussians in terms of a concrete example. Suppose that we are tracking an aircraft with two radar systems. I will ignore altitude so I can use two dimensional plots. Radar provides the range and bearing to a target. We start out being uncertain about the position of the aircraft, so the covariance, which is our uncertainty about the position, might look like this. In the language of Bayesian statistics this is our *prior*.

```
In [33]: P0 = [[6, 0], [0, 6]]
         plot_covariance_ellipse((10, 10), P0, fc='y', alpha=0.6)
```




Now suppose that there is a radar to the lower left of the aircraft. Further suppose that the radar's bearing measurement is accurate, but the range measurement is inaccurate. The covariance for the error in the measurement might look like this (plotted in green on top of the yellow prior):

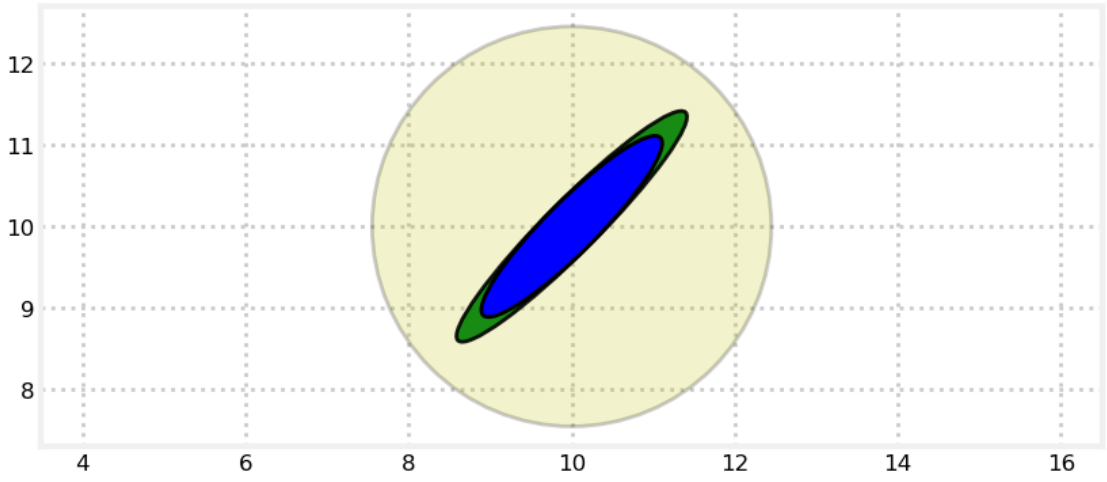
```
In [34]: P1 = [[2, 1.9], [1.9, 2]]
         plot_covariance_ellipse((10, 10), P0, fc='y', alpha=0.6)
         plot_covariance_ellipse((10, 10), P1, fc='g', alpha=0.9)
```




Recall that Bayesian statistics calls this the *evidence*. The ellipse points towards the radar. It is very long because the range measurement is inaccurate, and the aircraft could be within a considerable distance of the measured range. It is very narrow because the bearing estimate is very accurate and thus the aircraft must be very close to the bearing estimate.

We want to find the *posterior* - the mean and covariance that results from incorporating the evidence into the prior. As in every other chapter we combine evidence by multiplying them together.

```
In [35]: from filterpy.stats import multivariate_multiply

         P2 = multivariate_multiply((10, 10), P0, (10, 10), P1)[1]
         plot_covariance_ellipse((10, 10), P0, ec='k', fc='y', alpha=0.2)
         plot_covariance_ellipse((10, 10), P1, ec='k', fc='g', alpha=0.9)
         plot_covariance_ellipse((10, 10), P2, ec='k', fc='b')
```

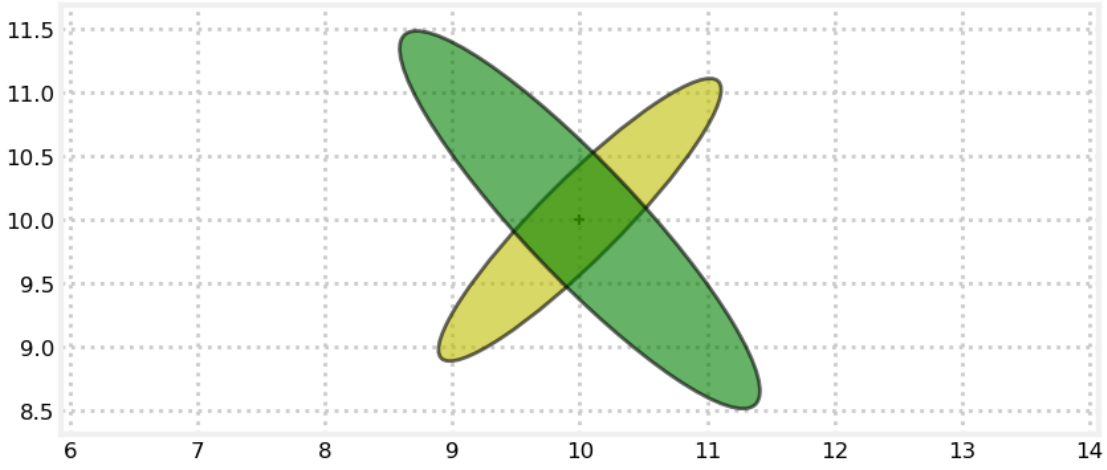



I have plotted the original estimate (prior) in a very transparent yellow, the radar reading in green (evidence), and the finale estimate (posterior) in blue.

The posterior retained the same shape and position as the radar measurement, but is smaller. We've seen this with one dimensional Gaussians. Multiplying two Gaussians makes the variance smaller because we are incorporating more information, hence we are less uncertain. Another point to recognize is that the covariance shape reflects the physical layout of the aircraft and the radar system. The importance of this will become clear in the next step.

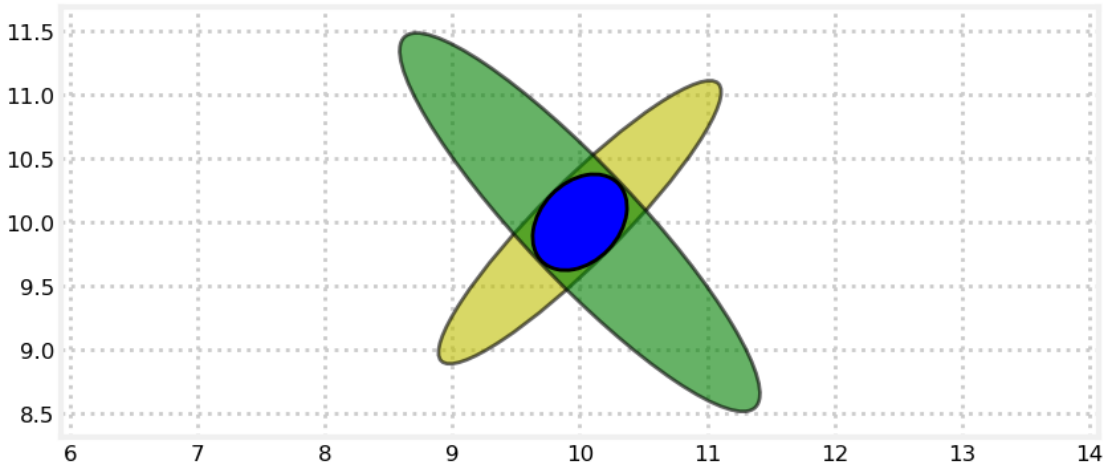
Now let's say we get a measurement from a second radar, this one to the lower right. The posterior from the last step becomes our new prior, which I plot in yellow. The new measurement is plotted in green.

```
In [36]: P3 = [[2, -1.9], [-1.9, 2.2]]
         plot_covariance_ellipse((10, 10), P2, ec='k', fc='y', alpha=0.6)
         plot_covariance_ellipse((10, 10), P3, ec='k', fc='g', alpha=0.6)
```



We incorporate this information by multiplying the Gaussians:

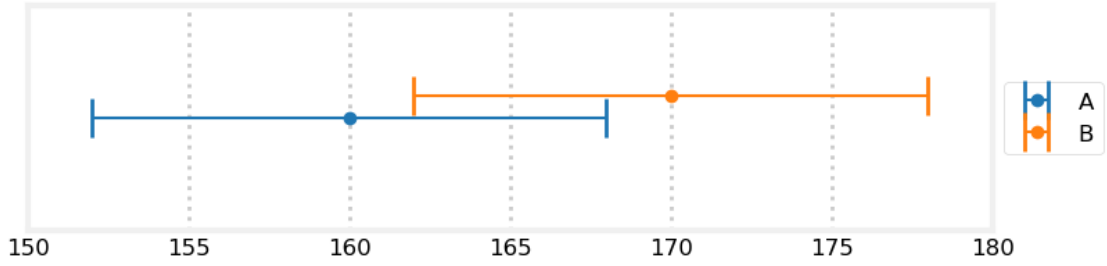
```
In [37]: P4 = multivariate_multiply((10, 10), P2, (10, 10), P3)[1]
         plot_covariance_ellipse((10, 10), P2, ec='k', fc='y', alpha=0.6)
         plot_covariance_ellipse((10, 10), P3, ec='k', fc='g', alpha=0.6)
         plot_covariance_ellipse((10, 10), P4, ec='k', fc='b')
```




The only likely place for the aircraft is where the two ellipses intersect. The intersection, formed by multiplying the prior and measurement, is a new Gaussian. The shapes reflects the geometry of the problem. This allows us to *triangulate* on the aircraft, resulting in a very accurate estimate. We didn't explicitly write any code to perform triangulation; it was a natural outcome of multiplying the Gaussians of each measurement together.

Think back to the **g-h Filter** chapter where we displayed the error bars of two weighings on a scale. The estimate must fall somewhere within the region where the error bars overlap. Here the estimate must fall between 161 to 163 pounds.

```
In [38]: import kf_book.book_plots as book_plots
         book_plots.plot_errorbars([(160, 8, 'A'), (170, 8, 'B')], xlims=(150, 180))
```

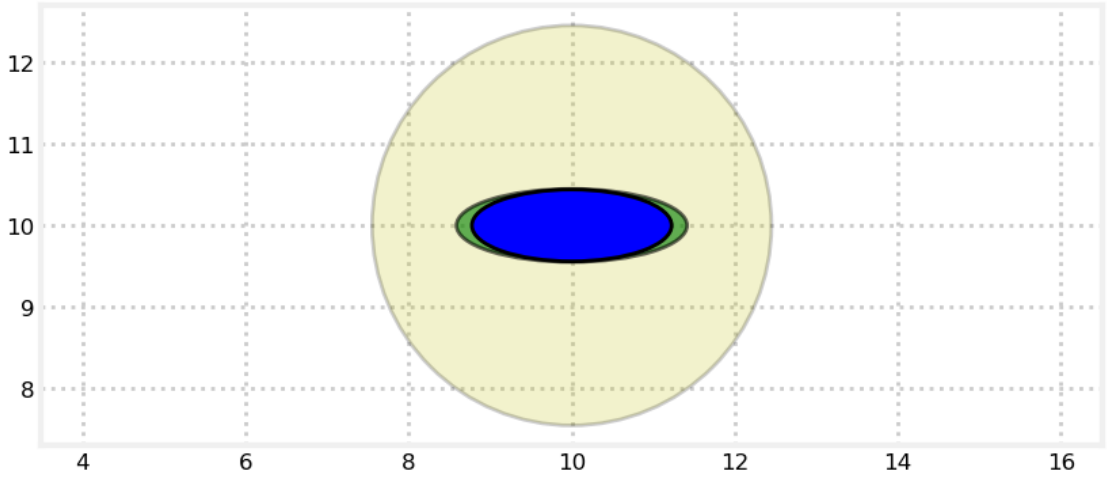



Let's consider a different layout. Suppose the first radar is directly to the left of the aircraft. I can model the measurement error with

$$\Sigma = \begin{bmatrix} 2 & 0 \\ 0 & 0.2 \end{bmatrix}$$

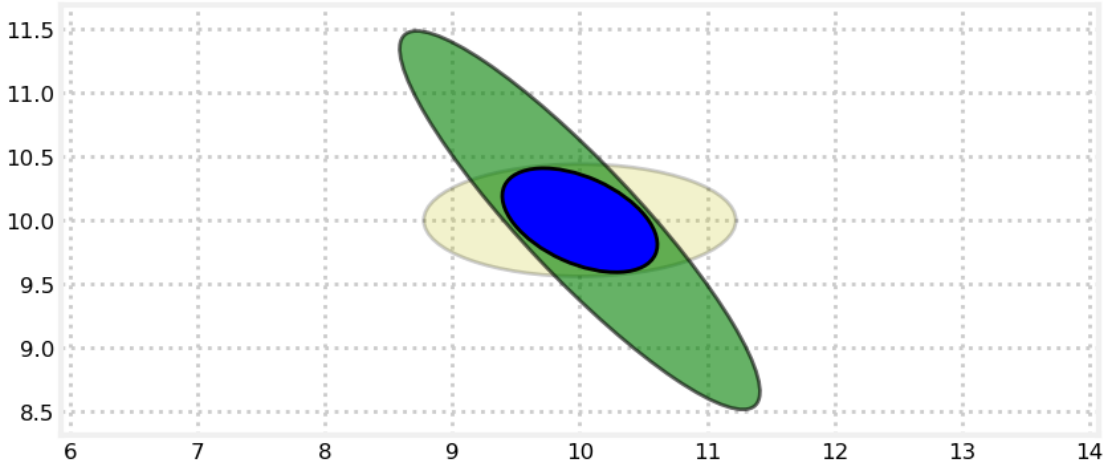
Here we see the result of multiplying the prior with the measurement.

```
In [39]: P1 = [[2, 0], [0, .2]]
         P2 = multivariate_multiply((10, 10), P0, (10, 10), P1)[1]
         plot_covariance_ellipse((10, 10), P0, ec='k', fc='y', alpha=0.2)
         plot_covariance_ellipse((10, 10), P1, ec='k', fc='g', alpha=0.6)
         plot_covariance_ellipse((10, 10), P2, ec='k', fc='b')
```




Now we can incorporate the measurement from the second radar system, which we will leave in the same position as before.

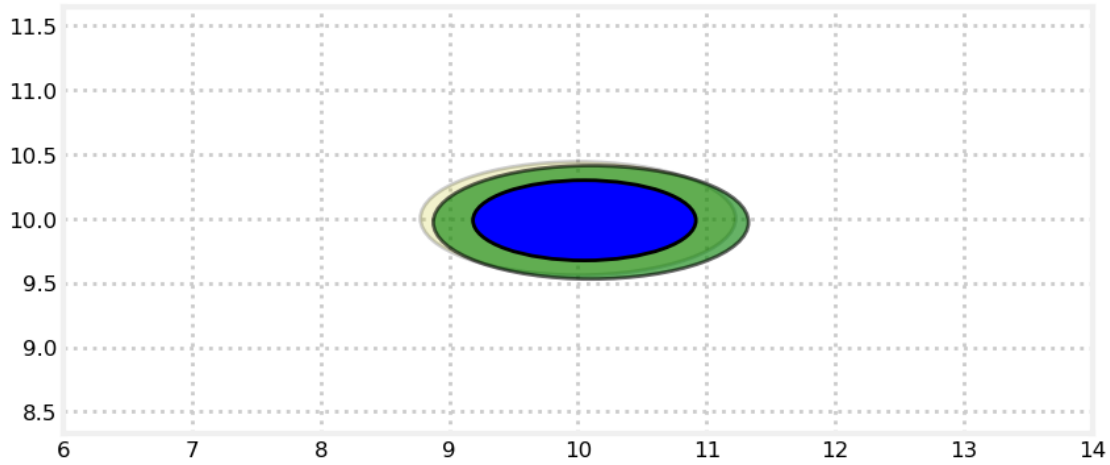
```
In [40]: P3 = [[2, -1.9], [-1.9, 2.2]]
         P4 = multivariate_multiply((10, 10), P2, (10, 10), P3)[1]
         plot_covariance_ellipse((10, 10), P2, ec='k', fc='y', alpha=0.2)
         plot_covariance_ellipse((10, 10), P3, ec='k', fc='g', alpha=0.6)
         plot_covariance_ellipse((10, 10), P4, ec='k', fc='b')
```




Our estimate is not as accurate as the previous example. The two radar stations are no longer orthogonal to each other relative to the aircraft's position so the triangulation is not optimal.

For a final example, imagine taking two measurements from the same radar a short time apart. The covariance ellipses will nearly overlap, leaving a very large error in our new estimate:

```
In [41]: P5 = multivariate_multiply((10,10), P2, (10.1, 9.97), P2)
         plot_covariance_ellipse((10, 10), P2, ec='k', fc='y', alpha=0.2)
         plot_covariance_ellipse((10.1, 9.97), P2, ec='k', fc='g', alpha=0.6)
         plot_covariance_ellipse(P5[0], P5[1], ec='k', fc='b')
         plt.xlim(6, 14);
```

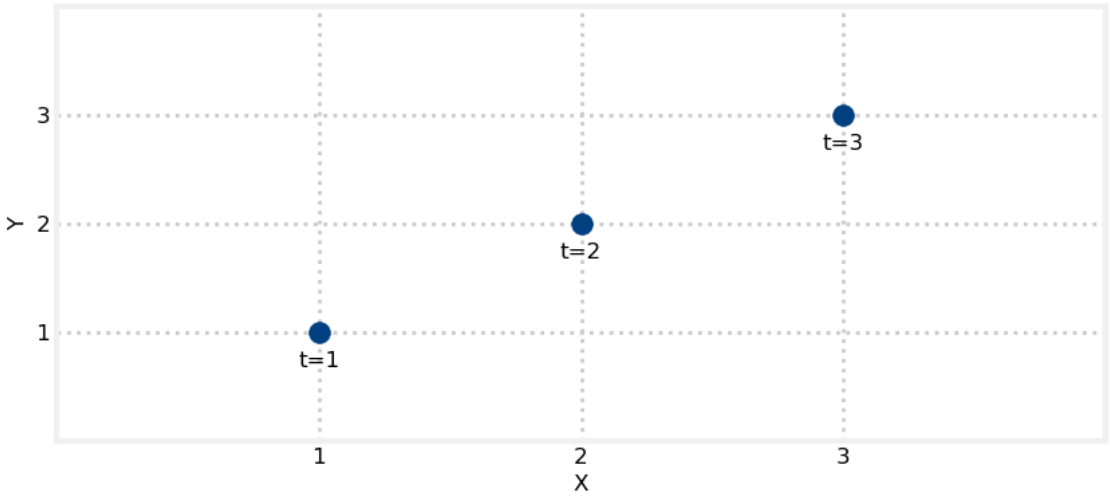



## 5.7 Hidden Variables

You can already see why a multivariate Kalman filter can perform better than a univariate one. Correlations between variables can significantly improve our estimates. We can take this much further. **This section contains the key insight to this chapter, so read carefully**.

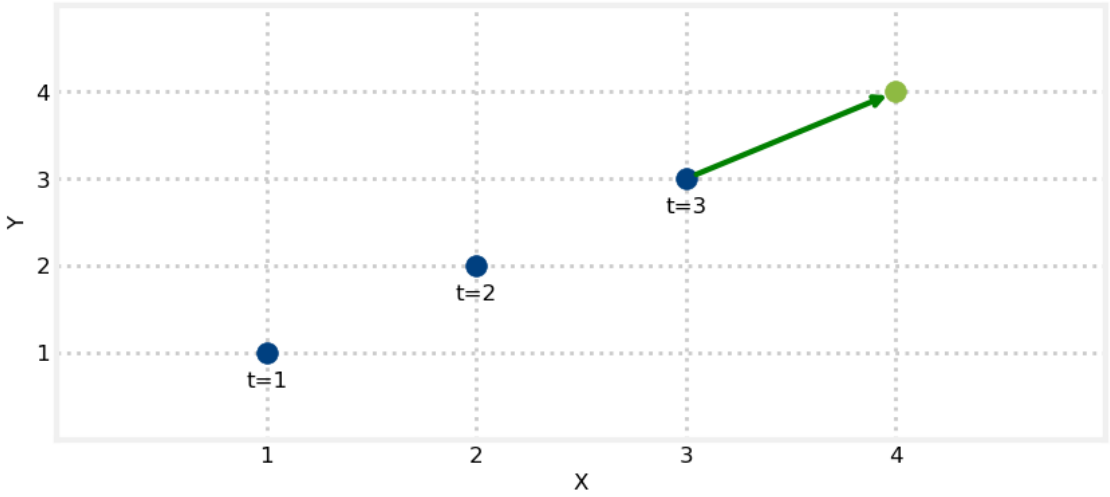
Let's say we are tracking an aircraft and we get the following data for the $x$ and $y$ coordinates at time $t$=1, 2, and 3 seconds. What does your intuition tell you the value of $x$ will be at time $t$=4 seconds?

```
In [42]: mkf_internal.show_position_chart()
```



It appears that the aircraft is flying in a straight line and we know that aircraft cannot turn on a dime. The most reasonable guess is that at $t$=4 the aircraft is at (4,4). I will depict that with a green arrow.

```
In [43]: mkf_internal.show_position_prediction_chart()
```



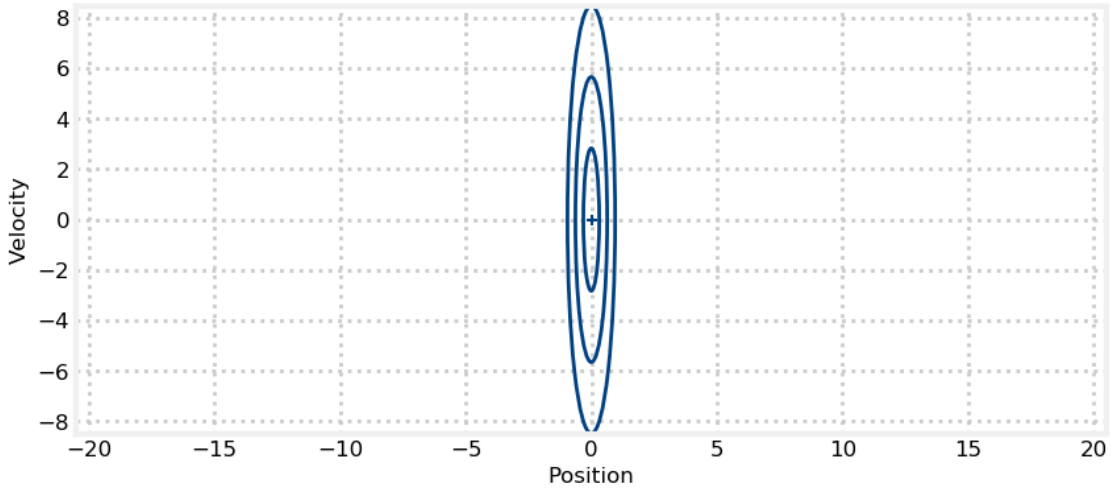

You made this inference because you *inferred* a constant velocity for the airplane. The reasonable assumption is that the aircraft is moving one unit each in $x$ and $y$ per time step.

Think back to the **g-h Filter** chapter when we were trying to improve the weight predictions of a noisy scale. We incorporated *weight gain* into the equations because it allowed us to make a better prediction of the weight the next day. The g-h filter uses the $g$ parameter to scale the amount of significance given to the current weight measurement, and the $h$ parameter scaled the amount of significance given to the weight gain.

We are going to do the same thing with our Kalman filter. After all, the Kalman filter is a form of a g-h filter. In this case we are tracking an airplane, so instead of weight and weight gain we need to track position and velocity. Weight gain is the *derivative* of weight, and of course velocity is the derivative of position. It's impossible to plot and understand the 4D chart that would be needed to plot $x$ and $y$ and their respective velocities so let's do it for $x$, knowing that the math generalizes to more dimensions.

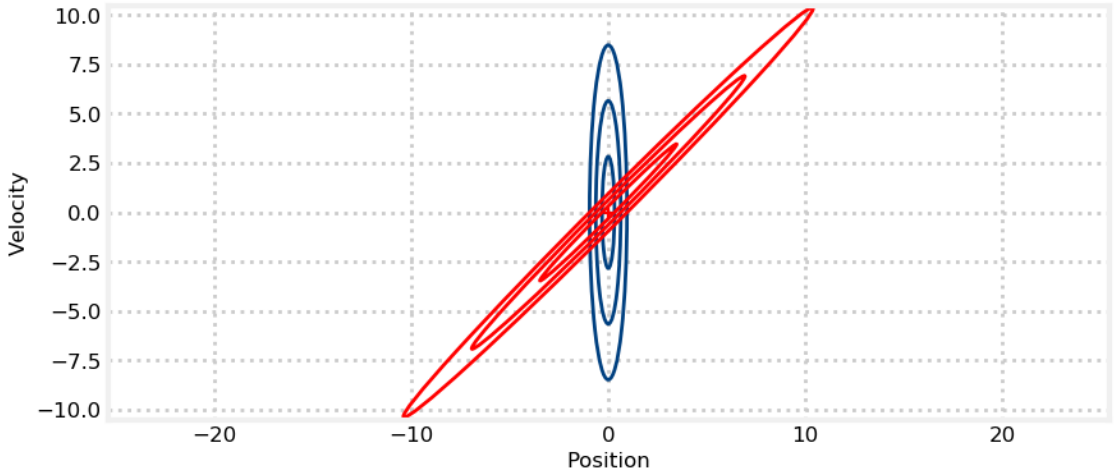
At time 1 we might be fairly certain about the position (x=0) but have no idea about the velocity. We can plot that with a covariance matrix like this. The narrower width expresses our relative certainty about position, and the tall height expresses our lack of knowledge about velocity.

```
In [44]: mkf_internal.show_x_error_chart(1)
```



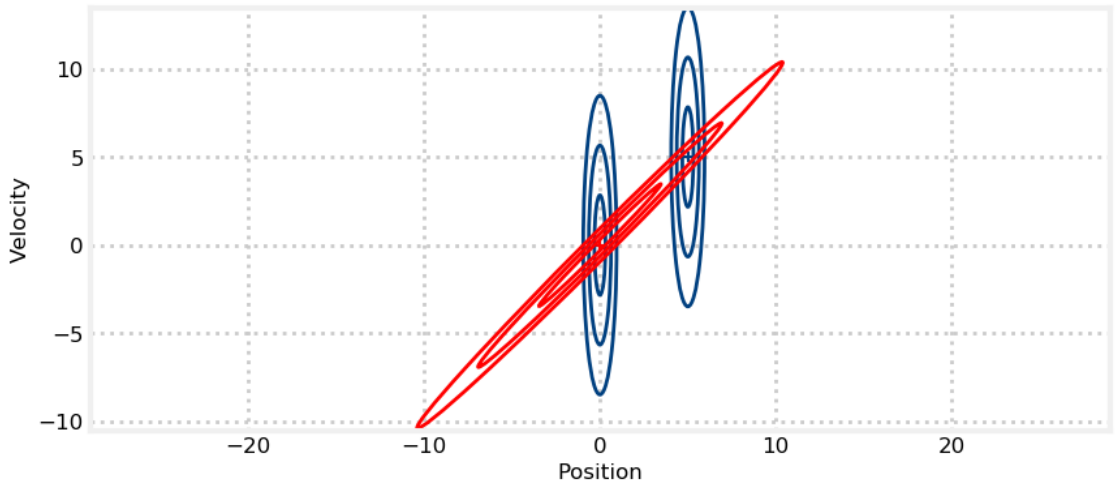

However, position and velocity are correlated. If the velocity is 5 m/s, then in 1 second the position will be 5 m. If the velocity is -10 m/s then in 1 second the position will be -10 m. Let's visualize this with a velocity covariance drawn on the diagonal.

```
In [45]: mkf_internal.show_x_error_chart(2)
```




At this point the velocity covariance doesn't help us. We can't predict a new position because we have no idea what the velocity is. But after one second we get a position update of x=5.

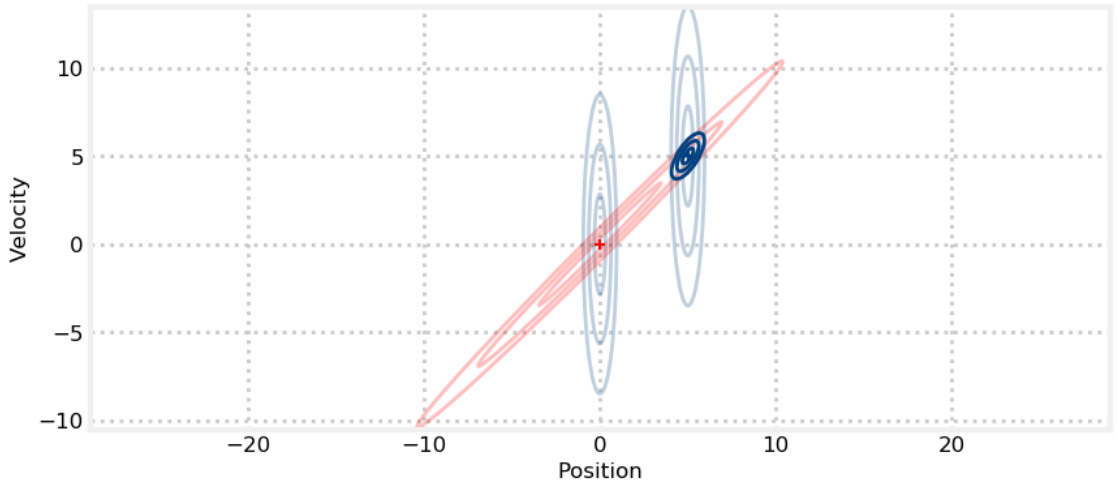
```
In [46]: mkf_internal.show_x_error_chart(3)
```




It won't be clear until the next chapter how I calculate what I'm about to say. Ignore the calculation, and think about what this implies. We have no easy way to say where the object really is because we are so uncertain about the velocity. Hence the ellipse stretches very far in the x-axis. Our uncertainty in velocity of course means it is also very spread in the y-axis. But as I said in the last paragraph, position is correlated to velocity. If the velocity is 5 m/s the next position would be 5, and if the velocity is 10 the next position would be 10. They are very correlated, so the ellipse must be very narrow.

This superposition of the two covariances is where the magic happens. The only reasonable estimate at time t=1 (where position=5) is roughly the intersection between the velocity covariance and measurement covariance! More exactly, we can use the math from the last section and multiply the two covariances together. From a Bayesian point of view we multiply the prior with the probability of the evidence (the *likelihood*) to get the posterior. If we multiply the position covariance with the velocity covariance using the Bayesian equations we get this result:

```
In [47]: mkf_internal.show_x_error_chart(4)
```




The new covariance (the posterior) lies at the intersection of the position covariance and the velocity covariance. It is slightly tilted, showing that there is some correlation between the position and velocity. Far

more importantly, it is much smaller than either the position or velocity covariances. In the previous chapter our variance would get smaller each time we performed an `update()` because the previous estimate was multiplied by the new measurement. The same happens here. However, here the improvement is markedly better. This is because we are using two different pieces of information which are nevertheless correlated. Knowing the velocity approximately and their correlation and the position approximately allows us to make a very accurate estimate. It is not easy to see in this diagram, but the uncertainty in position has become smaller. We know this because the new covariance doesn't reach as far in the x-axis as the measurement covariance at t=1. So we are not just more certain about velocity, but we are also more certain about the position than we would be if we only used position measurement without considering the velocity!

This is a key point, so read carefully! The radar is only detecting the position of the aircraft. This is called an *observed variable*. Based on the position estimates we can compute velocity. We call the velocity a *hidden variable*. Hidden means what it sounds like - there is no sensor that is measuring velocity, thus its value is hidden from us. We are able to use the correlation between position and velocity to infer its value very accurately.

To round out the terminology there are also *unobserved variables*. For example, the aircraft's state includes things such as as heading, engine RPM, weight, color, the first name of the pilot, and so on. We cannot sense these directly using the position sensor so they are not *observed*. There is no way to *infer* them from the sensor measurements and correlations (red planes don't go faster than white planes), so they are not *hidden*. Instead, they are *unobservable*. If you include an unobserved variable in your filter state the estimate for that variable will be nonsense.

What makes this possible? Imagine for a moment that we superimposed the velocity from a different airplane over the position graph. Clearly the two are not related, and there is no way that combining the two could possibly yield any additional information. In contrast, the velocity of this airplane tells us something very important - the direction and speed of travel. So long as the aircraft does not alter its velocity the velocity allows us to predict where the next position is. After a relatively small amount of error in velocity the probability that it is a good match with the position is very small. Think about it - if you suddenly change direction your position is also going to change a lot. If the measurement of the position is not in the direction of the velocity change it is very unlikely to be true. The two are correlated, so if the velocity changes so must the position, and in a predictable way.

It is important to understand that we are taking advantage of the fact that velocity and position are correlated. We get a rough estimate of velocity from the distance and time between two measurements, and use Bayes theorem to produce very accurate estimates after only a few observations. Please reread this section if you have any doubts. If you do not understand this you will quickly find it impossible to reason about what you will learn in the following chapters.

The effect of including velocity appears to me minor if only care about the position. But this is only after one update. In the next chapter we will see what a dramatic increase in certainty we have after multiple updates. The measurment variance will be large, but the estimated position variance will be small. Each time you intersect the velocity covariance with position it gets narrower on the x-axis, hence the variance is also smaller each time.

## 5.8 Higher Dimensions

So far I have shown you two dimensional Gaussians, but the math does not limit you to two dimensions. In later chapters we will be working in 9, or even 12 dimensions. If you work in areas such as weather prediction, you can end up with thousands of dimensions.

What do these higher dimensions 'look like? Well, a two dimensional Gaussian can be represented by an error ellipse, so it stands to reason a three dimensional Gaussian could be represented by a 3D error ellipsoid. We won't delve into the math here, but this turns out to be true. `FilterPy` provides a function to plot this ellipse.

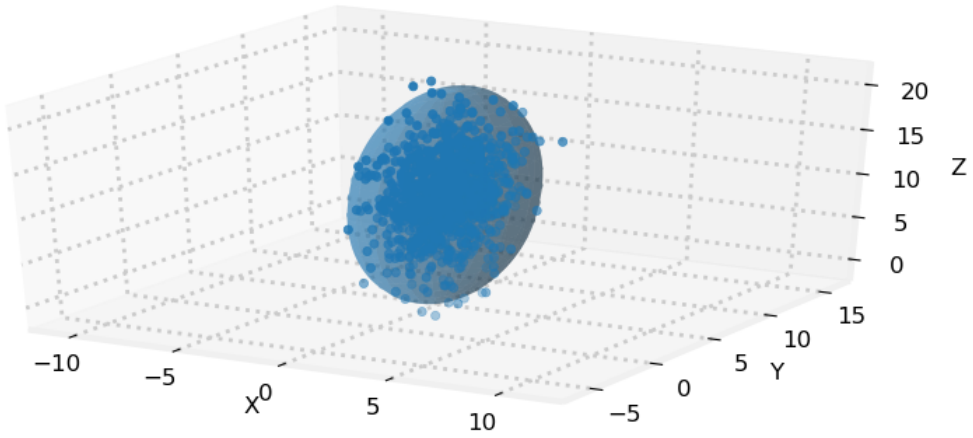
First, let's make some noisy data with a given covariance, just so we can plot it inside the ellipsoid.

```
In [48]: from filterpy.stats import plot_3d_covariance

         mu = [0.3, 5., 10.]
```

```
C = np.array([[1.0, .03, .2],
              [.03,  4.0, .0],
              [.2,  .0, 16.1]])

sample = np.random.multivariate_normal(mu, C, size=1000)
```

Now we plot the ellipsoid with the `FilterPy` function `plot_3d_covariance`, and then scatter plot the samples.

```
In [49]: ax = plot_3d_covariance(mu, C, alpha=.4, std=3, limit_xyz=True)
         ax.scatter(sample[:, 0], sample[:, 1], zs=sample[:, 2],);
```




Theory states that roughly 99% of a distribution will fall withing 3 standard deviations, and this appears to be the case.

Nine dimensions? I haven't quite figured out how to plot a 9D ellipsoid on a 2D screen, so there will be no graphs. The concept is the same; the standard deviation error of the distribution can be described by a 9D ellipsoid.

## 5.9 Summary

We have taken advantage of the geometry and correlations of the system to produce a very accurate estimate. The math does not care whether we are working with two positions, or a position and a correlated velocity, or if these are spatial dimensions. If floor space is correlated to house price you can write a Kalman filter to track house prices. If age is correlated to disease incidence you can write a Kalman filter to track diseases. If the zombie population is inversely correlated with the number of shotguns then you can write a Kalman filter to track zombie populations. I showed you this in terms of geometry and talked about *triangulation*. That was just to build your intuition. You can write a Kalman filter for state variables that have no geometric representation, such as filters for stock prices or milk production of cows (I received an email from someone tracking milk production!) Get used to thinking of these as Gaussians with correlations. If we can express our uncertainties as a multidimensional Gaussian we can then multiply the prior with the likelihood and get a much more accurate result.

## 5.10 References


- [1] http://docs.scipy.org/doc/scipy/reference/tutorial/stats.html
- [2] `FilterPy` library. Roger Labbe. https://github.com/rlabbe/filterpy

# Chapter 6

# Multivariate Kalman Filters

Filtering Multiple Random Variables

```
In [1]: from __future__ import division, print_function
        %matplotlib inline
```

```
In [2]: #format the book
        import book_format
        book_format.set_style()
```

```
Out[2]:
```

## 6.1 Introduction

We are now ready to study and implement the full, multivariate form of the Kalman filter. In the last chapter we learned how multivariate Gaussians express the correlation between multiple random variables, such as the position and velocity of an aircraft. We also learned how correlation between variables drastically improves the posterior. If we only roughly know position and velocity, but they are correlated, then our new estimate can be very accurate.

I prefer that you develop an intuition for how these filters work through several worked examples. I'm going to gloss over many issues. Some things I show you will only work for special cases, others will be 'magical' - it will not be clear how I derived a certain result. If I started with rigorous, generalized equations you would be left scratching your head about what all these terms mean and how you might apply them to your problem. In later chapters I will provide a more rigorous mathematical foundation, and at that time I will have to either correct approximations that I made in this chapter or provide additional information that I did not cover here.

To make this possible we will restrict ourselves to a subset of problems which we can describe with Newton's equations of motion. These filters are called *discretized continuous-time kinematic filters*. In the **Kalman Filter Math** chapter we will develop the math for non-Newtonian systems.

## 6.2 Newton's Equations of Motion

Newton's equations of motion tells us that given a constant velocity $v$ of a system we can compute its position $x$ after time $t$ with:

$$x = vt + x_0$$

For example, if we start at position 13, our velocity is 10 m/s, and we travel for 12 seconds our final position is 133 ($10 \times 12 + 13$).

We can incorporate constant acceleration with this equation

$$x = \frac{1}{2}at^2 + v_0t + x_0$$

And if we assume constant jerk we get

$$x = \frac{1}{6}jt^3 + \frac{1}{2}a_0t^2 + v_0t + x_0$$

These equations were generated by integrating a differential equation. Given a constant velocity v we can compute the distance traveled over time with the equation

$$x = vt + x_0$$

which we can derive with

$$\begin{aligned}
v &= \frac{dx}{dt} \\
dx &= v\,dt \\
\int_{x_0}^{x} dx &= \int_0^t v\,dt \\
x - x_0 &= vt - 0 \\
x &= vt + x_0
\end{aligned}$$

When you design a Kalman filter you start with a system of differential equations that describe the dynamics of the system. Most systems of differential equations do not easily integrate in this way. We start with Newton's equation because we can integrate and get a closed form solution, which makes the Kalman filter easier to design. An added benefit is that Newton's equations are the right equations to use to track moving objects, one of the main uses of Kalman filters.

## 6.3 Kalman Filter Algorithm

The algorithm is the same Bayesian filter algorithm that we have used in every chapter. The update step is slightly more complicated, but I will explain why when we get to it.

**Initialization**

```
1. Initialize the state of the filter
2. Initialize our belief in the state
```

**Predict**

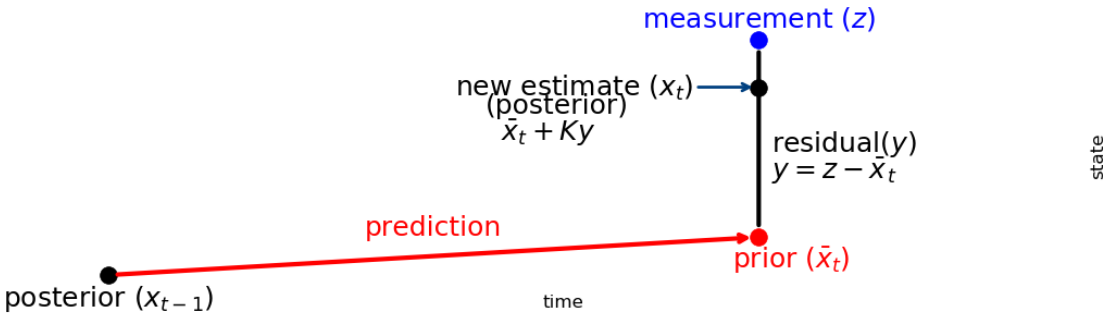
```
1. Use process model to predict state at the next time step
2. Adjust belief to account for the uncertainty in prediction
```

**Update**

```
1. Get a measurement and associated belief about its accuracy
2. Compute residual between estimated state and measurement
3. Compute scaling factor based on whether the measurement
or prediction is more accurate
4. set state between the prediction and measurement based
on scaling factor
5. update belief in the state based on how certain we are
in the measurement
```

As a reminder, here is a graphical depiction of the algorithm:

```
In [3]: import kf_book.book_plots as book_plots
        book_plots.show_residual_chart()
```




The univariate Kalman filter represented the state with a univariate Gaussian. Naturally the multivariate Kalman filter will use a multivariate Gaussian for the state. We learned in the last chapter that multivariate Gaussians use a vector for the mean and a matrix for the covariances. That means that the Kalman filter needs to use linear algebra to perform the estimations.

I don't want you to memorize these equations, but I have listed the univariate and multivariate equations below. They are quite similar.

**Predict**

| Univariate | Univariate (Kalman form) | Multivariate |
|---|---|---|
| $\bar\mu = \mu + \mu_{f_x}$<br>$\bar\sigma^2 = \sigma_x^2 + \sigma_{f_x}^2$ | $\bar x = x + dx$<br>$\bar P = P + Q$ | $\bar{\mathbf{x}} = \mathbf{Fx} + \mathbf{Bu}$<br>$\bar{\mathbf{P}} = \mathbf{FPF}^\mathsf{T} + \mathbf{Q}$ |

Without worrying about the specifics of the linear algebra, we can see that:

$\mathbf{x}$, $\mathbf{P}$ are the state mean and covariance. They correspond to $x$ and $\sigma^2$.

$\mathbf{F}$ is the *state transition function*. When multiplied by $\mathbf{x}$ it computes the prior.

$\mathbf{Q}$ is the process covariance. It corresponds to $\sigma_{f_x}^2$.

$\mathbf{B}$ and $\mathbf{u}$ are new to us. They let us model control inputs to the system.

**Update**

| Univariate | Univariate (Kalman form) | Multivariate |
|---|---|---|
| | $y = z - \bar x$ | $\mathbf{y} = \mathbf{z} - \mathbf{Hx}$ |
| | $K = \frac{\bar P}{\bar P + R}$ | $\mathbf{K} = \mathbf{PH}^\mathsf{T}(\mathbf{HPH}^\mathsf{T} + \mathbf{R})^{-1}$ |
| $\mu = \frac{\bar\sigma^2 \mu_z + \sigma_z^2 \bar\mu}{\bar\sigma^2 + \sigma_z^2}$ | $x = \bar x + Ky$ | $\mathbf{x} = \bar{\mathbf{x}} + \mathbf{Ky}$ |
| $\sigma^2 = \frac{\sigma_1^2 \sigma_2^2}{\sigma_1^2 + \sigma_2^2}$ | $P = (1 - K)\bar P$ | $\mathbf{P} = (\mathbf{I} - \mathbf{KH})\mathbf{P}$ |

$\mathbf{H}$ is the measurement function. We haven't seen this yet in this book and I'll explain it later. If you mentally remove $\mathbf{H}$ from the equations, you should be able to see these equations are similar as well.

$\mathbf{z}$, $\mathbf{R}$ are the measurement mean and noise covariance. They correspond to $z$ and $\sigma_z^2$ in the univariate filter (I've substituted $\mu$ with $x$ for the univariate equations to make the notation as similar as possible).

$\mathbf{y}$ and $\mathbf{K}$ are the residual and Kalman gain.

The details will be different than the univariate filter because these are vectors and matrices, but the concepts are exactly the same:

- Use a Gaussian to represent our estimate of the state and error
- Use a Gaussian to represent the measurement and its error
- Use a Gaussian to represent the process model
- Use the process model to predict the next state (the prior)
- Form an estimate part way between the measurement and the prior

Your job as a designer will be to design the state $(\mathbf{x}, \mathbf{P})$, the process $(\mathbf{F}, \mathbf{Q})$, the measurement $(\mathbf{z}, \mathbf{R})$, and the measurement function $\mathbf{H}$. If the system has control inputs, such as a robot, you will also design $\mathbf{B}$ and $\mathbf{u}$.

I have programmed the equations of the Kalman filter into the `predict` and `update` functions in FilterPy. You will import them with:

```
from filterpy.kalman import predict, update
```

## 6.4 Tracking a Dog

Let's go back to our tried and true problem of tracking a dog. This time we will include the fundamental insight of the previous chapter and use *hidden variables* to improve our estimates. I could start with the math, but instead let's implement a filter, learning as we go. On the surface the math is different and perhaps more complicated than the previous chapters, but the ideas are all the same - we are just multiplying and adding Gaussians.

We start by writing a simulation for the dog. The simulation will run for `count` steps, moving the dog forward approximately 1 meter for each step. At each step the velocity will vary according to the process variance `process_var`. After updating the position we compute a measurement with an assumed sensor variance of `z_var`. The function returns an NumPy array of the positions and another of the measurements.

```
In [4]: import math
        import numpy as np
        from numpy.random import randn

        def compute_dog_data(z_var, process_var, count=1, dt=1.):
            "returns track, measurements 1D ndarrays"
            x, vel = 0., 1.
            z_std = math.sqrt(z_var)
            p_std = math.sqrt(process_var)
            xs, zs = [], []
            for _ in range(count):
                v = vel + (randn() * p_std)
                x += v*dt
                xs.append(x)
                zs.append(x + randn() * z_std)
            return np.array(xs), np.array(zs)
```

## 6.5 Predict Step

For the prediction we need to design the state and covariance, the process model and the process noise, and optionally the control input. We'll take them in order.

### 6.5.1 Design State Variable

We previously tracked a dog in one dimension by using a Gaussian. The mean $(\mu)$ represented the most likely position, and the variance $(\sigma^2)$ represented the probability distribution of the position. The position is the *state* of the system, and we call $\mu$ the *state variable*.

In this problem we will be tracking both the position and velocity of the dog. This requires us to use a multivariate Gaussian represented with the state vector $\mathbf{x}$ and its corresponding covariance matrix $\mathbf{P}$.

State variables can either be *observed variables* - directly measured by a sensor, or *hidden variables* - inferred from the observed variables. For our dog tracking problem the sensor only reads position, so position is observed and velocity is hidden. We will learn how to track hidden variables soon.

It is important to understand that tracking position and velocity is a design choice with implications and assumptions that we are not yet prepared to explore. For example, we could also track acceleration, or even

jerk. For now, recall that in the last chapter we showed that including velocity in the covariance matrix resulted in much smaller variances in position. We will learn how the Kalman filter computes estimates for hidden variables later in this chapter.

In the univariate chapter we represented the dog's position with a scalar value (e.g. $\mu = 3.27$). In the last chapter we learned to use a multivariate Gaussian for multiple variables. For example, if we wanted to specify a position of 10.0 m and a velocity of 4.5 m/s, we would write:

$$\mu = \begin{bmatrix}10.0\\4.5\end{bmatrix}$$

The Kalman filter is implemented using linear algebra. We use an $n \times 1$ matrix (called a *vector*) to store $n$ state variables. For the dog tracking problem, we use $x$ to denote position, and the first derivative of $x$, $\dot{x}$, for velocity. I use Newton's dot notation for derivatives; $\dot{x}$ represents the first derivative of x with respect to t: $\dot{x} = \frac{dx}{dt}$. Kalman filter equations use $\mathbf{x}$ for the state, so we define $\mathbf{x}$ as:

$$\mathbf{x} = \begin{bmatrix}x\\\dot{x}\end{bmatrix}$$

We use $\mathbf{x}$ instead of $\mu$, but recognize this is the mean of the multivariate Gaussian.

Another way to write this is $\mathbf{x} = \begin{bmatrix}x & \dot{x}\end{bmatrix}^\mathsf{T}$ because the transpose of a row vector is a column vector. This notation is easier to use in text because it takes less vertical space.

$\mathbf{x}$ and the position $x$ coincidentally have the same name. If we were tracking the dog in the y-axis we would write $\mathbf{x} = \begin{bmatrix}y & \dot{y}\end{bmatrix}^\mathsf{T}$, not $\mathbf{y} = \begin{bmatrix}y & \dot{y}\end{bmatrix}^\mathsf{T}$. $\mathbf{x}$ is the standard name for the state variable used in the Kalman filter literature and we will not vary it to give it a more meaningful name. This consistency in naming allows us to communicate with our peers.

Let's code this. Initialization of `x` is as simple as

```
In [5]: x = np.array([[10.0],
                      [4.5]])
        x

Out[5]: array([[10. ],
               [ 4.5]])
```

I often use the transpose in my code to turn a row matrix into a column vector, as I find it easier to type and read:

```
In [6]: x = np.array([[10., 4.5]]).T
        x

Out[6]: array([[10. ],
               [ 4.5]])
```

However, NumPy recognizes 1D arrays as vectors, so I can simplify this line to use a 1D array.

```
In [7]: x = np.array([10.0, 4.5])
        x

Out[7]: array([10. ,  4.5])
```

All of the array elements have the same type, typically either `float` or `int`. If the list contains all `int`s then the created array will also have a data type of `int`, otherwise it will be `float`. I will often take advantage of this by only specifying one number as a floating point:

```
In [8]: np.array([1., 0, 0, 0, 0, 0])

Out[8]: array([1., 0., 0., 0., 0., 0.])
```

Here are some examples.

```
In [9]: A = np.array([[1, 2], [3, 4]])
        x = np.array([[10.0], [4.5]])

        # matrix multiply
        print(np.dot(A, x))
        print()
```

```
[[19.]
 [48.]]
```

In Python 3.5+ we have the matrix multiplier @, where `np.dot(A, B) == A @ B`. It is somewhat less useful then you might realize because it requires both `A` and `B` to be arrays. It is entirely valid in the math in this book for some of these variables to be scalars, therefore the utility of `@` is often lost.

```
In [10]: # alternative matrix multiply)
         print(A @ x)
         print()

         x = np.array([[10.0, 4.5]]).T
         print(A @ x)
         print()

         x = np.array([10.0, 4.5])
         print(A @ x)
```

```
[[19.]
 [48.]]

[[19.]
 [48.]]

[19. 48.]
```

The last returns a 1D array, but I have written the Kalman filter class to be able to handle this. In retrospect that might lead to confusion, but it does work.

### 6.5.2 Design State Covariance

The other half of the state Gaussian is the covariance matrix $\mathbf{P}$. In the univariate Kalman filter we specified an initial value for $\sigma^2$, and then the filter took care of updating its value as measurements were added to the filter. The same thing happens in the multidimensional Kalman filter. We specify an initial value for $\mathbf{P}$ and the filter updates it during each epoch.

We need to set the variances to reasonable values. For example, we may choose $\sigma^2_{\mathtt{pos}} = 500m^2$ if we are quite uncertain about the initial position. Top speed for a dog is around 21 m/s, so in the absence of any other information about the velocity we can set $3\sigma_{\mathtt{vel}} = 21$, or $\sigma^2_{\mathtt{vel}} = 7^2 = 49$.

In the last chapter we showed that the position and velocities are correlated. But how correlated are they for a dog? I have no idea. As we will see the filter computes this for us, so I initialize the covariances to zero. Of course, if you know the covariances you should use them.

Recall that the diagonals of the covariance matrix contains the variance of each variable, and the off-diagonal elements contains the covariances. Thus we have:

$$\mathbf{P} = \begin{bmatrix} 500 & 0 \\ 0 & 49 \end{bmatrix}$$

We can use `numpy.diag`, which creates a diagonal matrix from the values for the diagonal. Recall from linear algebra that a diagonal matrix is one with zeros in the off-diagonal elements.

```
In [11]: P = np.diag([500., 49.])
         P

Out[11]: array([[500.,   0.],
                [  0.,  49.]])
```

I could have written:

```
In [12]: P = np.array([[500., 0.],
                       [0., 49.]])
         P

Out[12]: array([[500.,   0.],
                [  0.,  49.]])
```

We are done. We've expressed the state of the filter as a multivariate Gaussian and implemented it in code.

### 6.5.3 Design the Process Model

The next step is designing the *process model*. It is a mathematical model which describes the behavior of the system. The filter uses it to predict the state after a discrete time step. We do this with a set of equations that describe the dynamics of the system.

In the univariate chapter we modeled the dog's motion with

$$x = v\Delta t + x_0$$

We implemented this as follows:

```
def predict(pos, movement):
    return gaussian(pos.mean + movement.mean,
                    pos.var + movement.var)
```

We will do the same thing in this chapter, using multivariate Gaussians instead of univariate Gaussians. You might imagine this sort of implementation:

$$\mathbf{x} = \begin{bmatrix} 5.4 \\ 4.2 \end{bmatrix},\ \dot{\mathbf{x}} = \begin{bmatrix} 1.1 \\ 0. \end{bmatrix}\ \mathbf{x} = \dot{\mathbf{x}}t + \mathbf{x}$$

But we need to generalize this. The Kalman filter equations work with any linear system, not just Newtonian ones. Maybe the system you are filtering is the plumbing system in a chemical plant, and the flow in a given pipe is determined by a linear combination of the settings of different valves.

$$\texttt{pipe}_1 = 0.134(\texttt{valve}_1) + 0.41(\texttt{valve}_2 - \texttt{valve}_3) + 1.34$$
$$\texttt{pipe}_2 = 0.210(\texttt{valve}_2) - 0.62(\texttt{valve}_1 - \texttt{valve}_5) + 1.86$$

Linear algebra has a powerful way to express systems of equations. Take this system

$$\begin{cases} 2x + 3y = 8 \\ 4x - y = 2 \end{cases}$$

We can put this in matrix form by writing:

$$\begin{bmatrix}2 & 3\\ 4 & -1\end{bmatrix}\begin{bmatrix}x\\ y\end{bmatrix} = \begin{bmatrix}8\\ 2\end{bmatrix}$$

If you perform the matrix multiplication in this equation the result will be the two equations above. In linear algebra we would write this as $\mathbf{Ax} = \mathbf{B}$, where

$$\mathbf{A} = \begin{bmatrix}2 & 3\\ 4 & -1\end{bmatrix}, \mathbf{x} = \begin{bmatrix}x\\ y\end{bmatrix}, \mathbf{B} = \begin{bmatrix}8\\ 2\end{bmatrix}$$

And then we can use the SciPy's `linalg` package to solve for $\mathbf{x}$:

```
In [13]: from scipy.linalg import solve
         A = np.array([[2, 3],[4, -1]])
         b = np.array([[8], [2]])
         x = solve(A, b)
         x

Out[13]: array([[1.],
                [2.]])
```

We use the process model to perform the *innovation*, because the equations tell us what the next state will be given the current state. Kalman filters implement this using this linear equation, where $\mathbf{x}$ is the *prior*, or predicted state:

$$\mathbf{x} = \mathbf{Fx}$$

which we can make explicit as

$$\begin{bmatrix}\bar{x}\\ \dot{\bar{x}}\end{bmatrix} = \begin{bmatrix}? & ?\\ ? & ?\end{bmatrix}\begin{bmatrix}x\\ \dot{x}\end{bmatrix}$$

Our job as Kalman filters designers is to specify $\mathbf{F}$ such that $\bar{\mathbf{x}} = \mathbf{Fx}$ performs the innovation (prediction) for our system. To do this we need one equation for each state variable. In our problem $\mathbf{x} = \begin{bmatrix}x & \dot{x}\end{bmatrix}^\mathsf{T}$, so we need one equation to compute the position $x$ and another to compute the velocity $\dot{x}$ . We already know the equation for the position innovation:

$$\bar{x} = x + \dot{x}\Delta t$$

What is our equation for velocity? We have no predictive model for how our dog's velocity will change over time. In this case we assume that it remains constant between innovations. Of course this is not exactly true, but so long as the velocity doesn't change too much over each innovation you will see that the filter performs very well. So we say

$$\dot{\bar{x}} = \dot{x}$$

This gives us the process model for our system

$$\begin{cases}\bar{x} = x + \dot{x}\Delta t\\ \dot{\bar{x}} = \dot{x}\end{cases}$$

This correctly has one equation for each variable in the state, isolated on the left hand side. We need to express this set of equations in the form $\bar{\mathbf{x}} = \mathbf{Fx}$. Rearranging terms makes it easier to see what to do.

$$\begin{cases}\bar{x} = 1x + & \Delta t\,\dot{x}\\ \dot{\bar{x}} = 0x + & 1\,\dot{x}\end{cases}$$

We can rewrite this in matrix form as

$$\begin{bmatrix}\bar{x} \\ \bar{\dot{x}}\end{bmatrix} = \begin{bmatrix}1 & \Delta t \\ 0 & 1\end{bmatrix}\begin{bmatrix}x \\ \dot{x}\end{bmatrix}$$
$$\mathbf{x} = \mathbf{F}\mathbf{x}$$

$\mathbf{F}$ is called the *state transition function* or the *state transition matrix*. In later chapters it will be a true function, not a matrix, so calling it a function is a bit more general.

```
In [14]: dt = 0.1
         F = np.array([[1, dt],
                       [0, 1]])
         F

Out[14]: array([[1. , 0.1],
                [0. , 1. ]])
```

Let's test this! FilterPy has a `predict` method that performs the prediction by computing $\mathbf{x} = \mathbf{F}\mathbf{x}$. Let's call it and see what happens. We've set the position to 10.0 and the velocity to 4.5 meter/sec. We've defined `dt = 0.1`, which means the time step is 0.1 seconds, so we expect the new position to be 10.45 meters after the innovation. The velocity should be unchanged.

```
In [15]: from filterpy.kalman import predict

         x = np.array([10.0, 4.5])
         P = np.diag([500, 49])
         F = np.array([[1, dt], [0, 1]])

         # Q is the process noise
         x, P = predict(x=x, P=P, F=F, Q=0)
         print('x =', x)

x = [10.45  4.5 ]
```

This worked. If we call `predict()` several times in a row the value will be updated each time.

```
In [16]: for _ in range(4):
             x, P = predict(x=x, P=P, F=F, Q=0)
             print('x =', x)

x = [10.9  4.5]
x = [11.35  4.5 ]
x = [11.8  4.5]
x = [12.25  4.5 ]
```

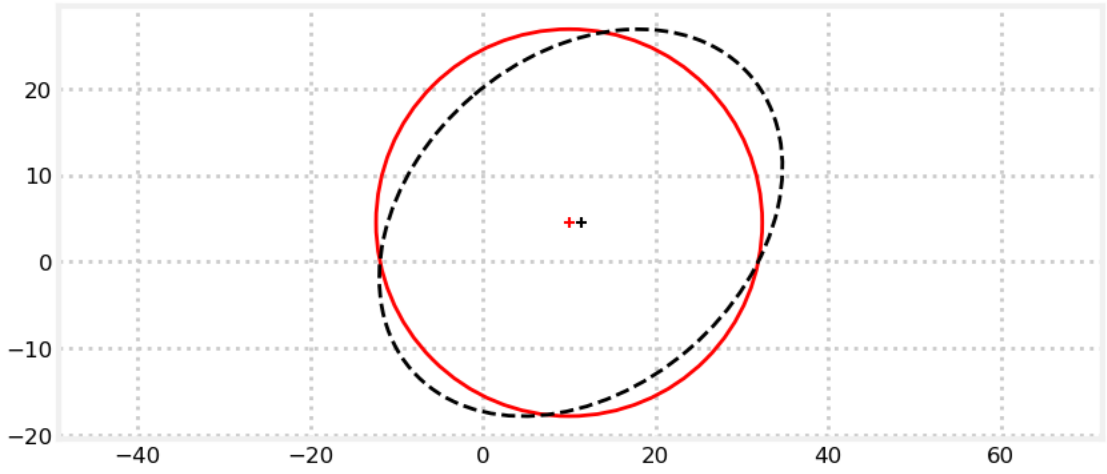
`predict()` computes both the mean and covariance of the innovation. This is the value of $\mathbf{P}$ after five innovations (predictions), which we denote $\mathbf{P}$ in the Kalman filter equations.

```
In [17]: print(P)

[[512.25  24.5 ]
 [ 24.5   49.  ]]
```

Inspecting the diagonals shows us that the position variance got larger. We've performed five prediction steps with no measurements, and our uncertainty grew. The off-diagonal elements became non-zero - the Kalman filter detected a correlation between position and velocity! The variance of the velocity did not change.

Here I plot the covariance before and after the prediction. The initial value is in solid red, and the prior (prediction) is in dashed black. I've altered the covariance and time step to better illustrate the change.

```
In [18]: from filterpy.stats import plot_covariance_ellipse

         dt = 0.3
         F = np.array([[1, dt], [0, 1]])
         x = np.array([10.0, 4.5])
         P = np.diag([500, 500])
         plot_covariance_ellipse(x, P, edgecolor='r')
         x, P = predict(x, P, F, Q=0)
         plot_covariance_ellipse(x, P, edgecolor='k', ls='dashed')
```




You can see that the center of the ellipse shifted by a small amount (from 10 to 11.35) because the position changed. The ellipse also elongated, showing the correlation between position and velocity. How does the filter compute new values for $\mathbf{P}$, and what is it based on? Note that I set the process noise `Q` to zero each time, so it is not due to me adding noise. It's a little to early to discuss this, but recall that in every filter so far the predict step entailed a loss of information. The same is true here. I will give you the details once we have covered a bit more ground.

### 6.5.4 Design Process Noise

A quick review on *process noise*. A car is driving along the road with the cruise control on; it should travel at a constant speed. We model this with $\bar{x}_k = \dot{x}_k \Delta t + x_{k-1}$. However, it is affected by a number of unknown factors. The cruise control cannot perfectly maintain a constant velocity. Winds affect the car, as do hills and potholes. Passengers roll down windows, changing the drag profile of the car.

We can model this system with the differential equation

$$\dot{\mathbf{x}} = f(\mathbf{x}) + w$$

where $f(\mathbf{x})$ models the state transition and $w$ is *white process noise*.

We will learn how to go from a set of differential equations to the Kalman filter matrices in the **Kalman Filter Math** chapter. In this chapter we take advantage of the fact that Newton already derived the

equations of motion for us. For now you just need to know that we account for the noise in the system by adding a process noise covariance matrix $\mathbf{Q}$ to the covariance $\mathbf{P}$. We do not add anything to $\mathbf{x}$ because the noise is *white* - which means that the mean of the noise will be 0. If the mean is 0, $\mathbf{x}$ will not change.

The univariate Kalman filter used `variance = variance + process_noise` to compute the variance for the variance of the prediction step. The multivariate Kalman filter does the same, essentially `P = P + Q`. I say 'essentially' because there are other terms unrelated to noise in the covariance equation that we will see later.

Deriving the process noise matrix can be quite demanding, and we will put it off until the Kalman math chapter. For now know that $\mathbf{Q}$ equals the expected value of the white noise $w$, computed as $\mathbf{Q} = \mathbb{E}[\mathbf{w}\mathbf{w}^\mathsf{T}]$. In this chapter we will focus on building an intuitive understanding on how modifying this matrix alters the behavior of the filter.

FilterPy provides functions which compute $\mathbf{Q}$ for the kinematic problems of this chapter. `Q_discrete_white_noise` takes 3 parameters. `dim`, which specifies the dimension of the matrix, `dt`, which is the time step in seconds, and `var`, the variance in the noise. Briefly, it discretizes the noise over the given time period under assumptions that we will discuss later. This code computes $\mathbf{Q}$ for white noise with a variance of 2.35 and a time step of 1 seconds:

```
In [19]: from filterpy.common import Q_discrete_white_noise
         Q = Q_discrete_white_noise(dim=2, dt=1., var=2.35)
         print(Q)
```

```
[[0.588 1.175]
 [1.175 2.35 ]]
```

### 6.5.5 Design the Control Function

The Kalman filter does not just filter data, it allows us to incorporate the control inputs of systems like robots and airplanes. Suppose we are controlling a robot. At each time step we would send steering and velocity signals to the robot based on its current position vs desired position. Kalman filter equations incorporate that knowledge into the filter equations, creating a predicted position based both on current velocity and control inputs to the drive motors. Remember, we *never* throw information away.

For a linear system the effect of control inputs can be described as a set of linear equations, which we can express with linear algebra as

$$\Delta\mathbf{x} = \mathbf{B}\mathbf{u}$$

Here $\mathbf{u}$ is the *control input*, and $\mathbf{B}$ is the *control input model* or *control function*. For example, $\mathbf{u}$ might be a voltage controlling how fast the wheel's motor turns, and multiplying by $\mathbf{B}$ yields $\Delta\left[\begin{smallmatrix}x\\\dot{x}\end{smallmatrix}\right]$. In other words, it must compute how much $\mathbf{x}$ changes due to the control input.

Therefore the complete Kalman filter equation for the prior mean is

$$\mathbf{x} = \mathbf{F}\mathbf{x} + \mathbf{B}\mathbf{u}$$

and this is the equation that is computed when you call `KalmanFilter.predict()`.

Your dog may be trained to respond to voice commands. All available evidence suggests that my dog has no control inputs whatsoever, so I set $\mathbf{B}$ to zero. In Python we write:

```
In [20]: B = 0.  # my dog doesn't listen to me!
         u = 0
         x, P = predict(x, P, F, Q, B, u)
         print('x =', x)
         print('P =', P)
```

```
x = [12.7  4.5]
P = [[680.587 301.175]
```

```
 [301.175 502.35 ]]
```

Setting $\mathbf{B}$ and $\mathbf{u}$ to zero is not necessary since `predict` uses 0 for their default value:

```
In [21]: predict(x, P, F, Q)[0] == predict(x, P, F, Q, B, u)[0]

Out[21]: array([ True,  True])

In [22]: predict(x, P, F, Q)[1] == predict(x, P, F, Q, B, u)[1]

Out[22]: array([[ True,  True],
                [ True,  True]])
```

**Prediction: Summary**

Your job as a designer is to specify the matrices for

- $\mathbf{x}$, $\mathbf{P}$: the state and covariance
- $\mathbf{F}$, $\mathbf{Q}$: the process model and noise covariance
- $\mathbf{B}, \mathbf{u}$: Optionally, the control input and function

## 6.6 Update Step

Now we can implement the update step of the filter. You only have to supply two more matrices, and they are easy to understand.

### 6.6.1 Design the Measurement Function

The Kalman filter computes the update step in what we call *measurement space*. We mostly ignored this issue in the univariate chapter because of the complication it adds. We tracked our dog's position using a sensor that reported his position. Computing the *residual* was easy - subtract the filter's predicted position from the measurement:

$$\texttt{residual} = \texttt{measured position} - \texttt{predicted position}$$

We need to compute the residual because we scale it by the Kalman gain to get the new estimate.

What would happen if we were trying to track temperature using a thermometer that outputs a voltage corresponding to the temperature reading? The equation for the residual computation would be meaningless; you can't subtract a temperature from a voltage.

$$\texttt{residual} = \texttt{voltage} - \texttt{temperature} \quad (NONSENSE!)$$

We need to convert the temperature into a voltage so we can perform the subtraction. For the thermometer we might write:

```
CELSIUS_TO_VOLTS = 0.21475
residual = voltage - (CELSIUS_TO_VOLTS * predicted_temperature)
```

The Kalman filter generalizes this problem by having you supply a *measurement function* that converts a state into a measurement.

Why are we working in measurement space? Why not work in state space by converting the voltage into a temperature, allowing the residual to be a difference in temperature?

We cannot do that because most measurements are not *invertible*. The state for the tracking problem contains the hidden variable $\dot{x}$. There is no way to convert a measurement of position into a state containing velocity. On the other hand, it is trivial to convert a state containing position and velocity into a equivalent

"measurement" containing only position. We have to work in measurement space to make the computation of the residual possible.

Both the measurement $\mathbf{z}$ and state $\mathbf{x}$ are vectors so we need to use a matrix to perform the conversion. The Kalman filter equation that performs this step is:

$$\mathbf{y} = \mathbf{z} - \mathbf{H}\mathbf{x}$$

where $\mathbf{y}$ is the residual, $\mathbf{x}$ is the prior, $\mathbf{z}$ is the measurement, and $\mathbf{H}$ is the measurement function. So we take the prior, convert it to a measurement by multiplying it with $\mathbf{H}$, and subtract that from the measurement. This gives us the difference between our prediction and measurement in measurement space!

We need to design $\mathbf{H}$ so that $\mathbf{Hx}$ yields a measurement. For this problem we have a sensor that measures position, so $\mathbf{z}$ will be a one variable vector:

$$\mathbf{z} = \begin{bmatrix} z \end{bmatrix}$$

The residual equation will have the form

$$\begin{aligned} \mathbf{y} &= \mathbf{z} - \mathbf{H}\mathbf{x} \\ \begin{bmatrix} y \end{bmatrix} &= \begin{bmatrix} z \end{bmatrix} - \begin{bmatrix} ? & ? \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix} \end{aligned}$$

$\mathbf{H}$ has to be a 1x2 matrix for $\mathbf{Hx}$ to be 1x1. Recall that multiplying matrices $m \times n$ by $n \times p$ yields a $m \times p$ matrix.

We will want to multiply the position $x$ by 1 to get the corresponding measurement of the position. We do not need to use velocity to find the corresponding measurement so we multiply $\dot{x}$ by 0.

$$\begin{aligned} \mathbf{y} &= \mathbf{z} - \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix} \\ &= [z] - [x] \end{aligned}$$

And so, for our Kalman filter we set

$$\mathbf{H} = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

```
In [23]: H = np.array([[1., 0.]])
```

We have designed the majority of our Kalman filter. All that is left is to model the noise in the sensors.

### Design the Measurement

The measurement is implemented with $\mathbf{z}$, the measurement mean, and $\mathbf{R}$, the measurement covariance.

$\mathbf{z}$ is easy. it contains the measurement(s) as a vector. We have only one measurement, so we have:

$$\mathbf{z} = \begin{bmatrix} z \end{bmatrix}$$

If we have two sensors or measurements we'd have:

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$$

The *measurement noise matrix* models the noise in our sensors as a covariance matrix. In practice this can be difficult. A complicated system may have many sensors, the correlation between them might not be clear, and usually their noise is not a pure Gaussian. For example, a sensor might be biased to read high if the temperature is high, and so the noise is not distributed equally on both sides of the mean. We will learn to deal with these problems later.

The Kalman filter equations uses a covariance matrix $\mathbf{R}$ for the measurement noise. The matrix will have dimension $m \times m$, where $m$ is the number of sensors. It is a covariance matrix to account for correlations between the sensors. We have only 1 sensor so R is:

$$R = \begin{bmatrix}\sigma_z^2\end{bmatrix}$$

If $\sigma_z^2$ is 5 meters squared we'd have $R = \begin{bmatrix}5\end{bmatrix}$.

If we had two position sensors, the first with a variance of 5 $m^2$, the second with a variance of 3 $m^2$, we would write

$$R = \begin{bmatrix}5 & 0\\ 0 & 3\end{bmatrix}$$

We put the variances on the diagonal because this is a *covariance* matrix, where the variances lie on the diagonal, and the covariances, if any, lie in the off-diagonal elements. Here we assume there is no correlation in the noise between the two sensors, so the covariances are 0.

For our problem we only have one sensor, so we can implement this as

```
In [24]: R = np.array([[5.]])
```

We perform the update by calling `update`.

```
In [25]: from filterpy.kalman import update
         z = 1.
         x, P = update(x, P, z, R, H)
         print('x =', x)
```

```
x = [ 1.085 -0.64 ]
```

Keeping track of all of these variables is burdensome, so FilterPy also implements the filter with the class `KalmanFilter`. I will use the class in the rest of this book, but I wanted you to see the procedural form of these functions since I know some of you are not fans of object oriented programming.

## 6.7 Implementing the Kalman Filter

I've given you all of the code for the filter, but now let's collect it in one place. First we construct a `KalmanFilter` object. We have to specify the number of variables in the state with the `dim_x` parameter, and the number of measurements with `dim_z`. We have two random variables in the state and one measurement, so we write:

```
from filterpy.kalman import KalmanFilter
dog_filter = KalmanFilter(dim_x=2, dim_z=1)
```

This creates an object with default values for all the Kalman filter matrices:

```
In [26]: from filterpy.kalman import KalmanFilter
         dog_filter = KalmanFilter(dim_x=2, dim_z=1)
         print('x = ', dog_filter.x.T)
         print('R = ', dog_filter.R)
         print('Q = \n', dog_filter.Q)
         # etc...
```

```
x =  [[0. 0.]]
R =  [[1.]]
Q =
 [[1. 0.]
 [0. 1.]]
```

Now we initialize the filter's matrices and vectors with values valid for our problem. I've put this in a function to allow you to specify different initial values for `R`, `P`, and `Q` and put it in a helper function. We will be creating and running many of these filters, and this saves us a lot of headaches.

```
In [27]: from filterpy.kalman import KalmanFilter
         from filterpy.common import Q_discrete_white_noise

         def pos_vel_filter(x, P, R, Q=0., dt=1.0):
             """ Returns a KalmanFilter which implements a
             constant velocity model for a state [x dx].T
             """

             kf = KalmanFilter(dim_x=2, dim_z=1)
             kf.x = np.array([x[0], x[1]]) # location and velocity
             kf.F = np.array([[1., dt],
                              [0.,  1.]])  # state transition matrix
             kf.H = np.array([[1., 0]])    # Measurement function
             kf.R *= R                     # measurement uncertainty
             if np.isscalar(P):
                 kf.P *= P                 # covariance matrix
             else:
                 kf.P[:] = P               # [:] makes deep copy
             if np.isscalar(Q):
                 kf.Q = Q_discrete_white_noise(dim=2, dt=dt, var=Q)
             else:
                 kf.Q[:] = Q
             return kf
```

`KalmanFilter` initializes `R`, `P`, and `Q` to the identity matrix, so `kf.P *= P` is one way to quickly assign all of the diagonal elements to the same scalar value. Now we create the filter:

```
In [28]: dt = .1
         x = np.array([0., 0.])
         kf = pos_vel_filter(x, P=500, R=5, Q=0.1, dt=dt)
```

You can inspect the current values of all attributes of the filter by entering the variable on the command line.

```
In [29]: kf
```

```
Out[29]: KalmanFilter object
         dim_x = 2
         dim_z = 1
         dim_u = 0
         x = [0. 0.]
         P = [[500.   0.]
              [  0. 500.]]
         x_prior = [[0. 0.]].T
         P_prior = [[1. 0.]
                    [0. 1.]]
         x_post = [[0. 0.]].T
         P_post = [[1. 0.]
                   [0. 1.]]
         F = [[1.  0.1]
              [0.  1. ]]
```

```
Q = [[0.    0.   ]
     [0.    0.001]]
R = [[5.]]
H = [[1. 0.]]
K = [[0. 0.]].T
y = [[0.]]
S = [[0.]]
SI = [[0.]]
M = [[0. 0.]].T
B = None
z = [[None]]
log-likelihood = -708.3964185322641
likelihood = 2.2250738585072014e-308
mahalanobis = 0.0
alpha = 1.0
inv = <function inv at 0x000001CEE99D48B8>
```

All that is left is to write the code to run the Kalman filter.

```
In [30]: from kf_book.mkf_internal import plot_track

         def run(x0=(0.,0.), P=500, R=0, Q=0, dt=1.0,
                 track=None, zs=None,
                 count=0, do_plot=True, **kwargs):
             """
             track is the actual position of the dog, zs are the
             corresponding measurements.
             """

             # Simulate dog if no data provided.
             if zs is None:
                 track, zs = compute_dog_data(R, Q, count)

             # create the Kalman filter
             kf = pos_vel_filter(x0, R=R, P=P, Q=Q, dt=dt)

             # run the kalman filter and store the results
             xs, cov = [], []
             for z in zs:
                 kf.predict()
                 kf.update(z)
                 xs.append(kf.x)
                 cov.append(kf.P)

             xs, cov = np.array(xs), np.array(cov)
             if do_plot:
                 plot_track(xs[:, 0], track, zs, cov, **kwargs)
             return xs, cov
```

This is the complete code for the filter, and most of it is boilerplate. I've made it flexible enough to support several uses in this chapter, so it is a bit verbose. Let's work through it line by line.

The first lines checks to see if you provided it with measurement data in `data`. If not, it creates the data using the `compute_dog_data` function we wrote earlier.

The next lines uses our helper function to create a Kalman filter.

```
# create the Kalman filter
kf = pos_vel_filter(x0, R=R, P=P, Q=Q, dt=dt)
```

All we need to do is perform the update and predict steps of the Kalman filter for each measurement. The `KalmanFilter` class provides the two methods `update()` and `predict()` for this purpose. `update()` performs the measurement update step of the Kalman filter, and so it takes a variable containing the sensor measurement.

Absent the work of storing the results, the loop reads:

```
for z in zs:
    kf.predict()
    kf.update(z)
```

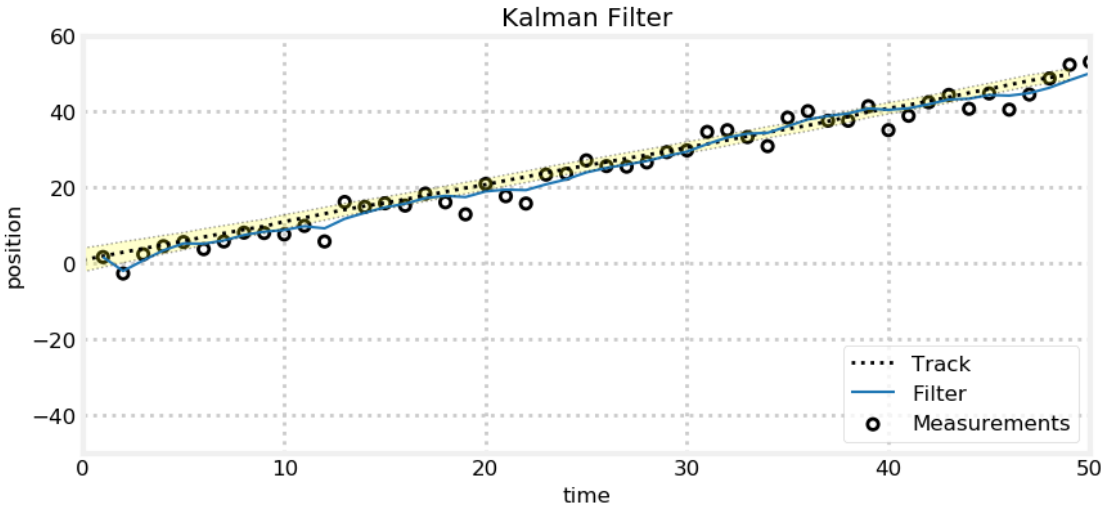
Each call to `predict` and `update` modifies the state variables `x` and `P`. Therefore, after the call to `predict`, `kf.x` contains the prior. After the call to update, `kf.x` contains the posterior. `data` contains the actual position and measurement of the dog, so we use `[:, 1]` to get an array of measurements.

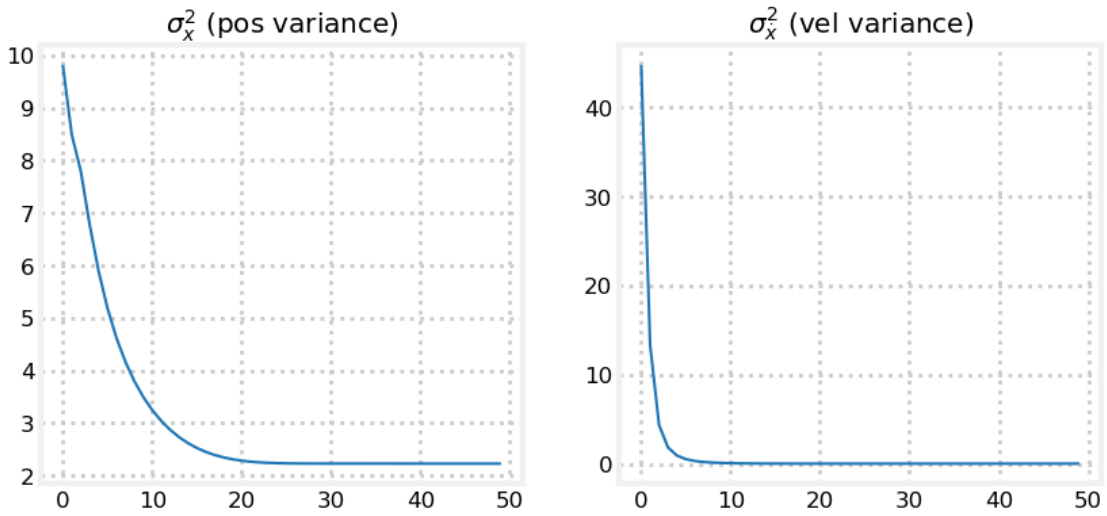
It really cannot get much simpler than that. As we tackle more complicated problems this code will remain largely the same; all of the work goes into setting up the `KalmanFilter` matrices; executing the filter is trivial.

The rest of the code optionally plots the results and then returns the saved states and covariances.

Let's run it. We have 50 measurements with a noise variance of 10 and a process variance of 0.01.

```
In [31]: P = np.diag([500., 49.])
         Ms, Ps = run(count=50, R=10, Q=0.01, P=P)
```






There is still a lot to learn, but we have implemented a Kalman filter using the same theory and equations as published by Rudolf Kalman! Code very much like this runs inside of your GPS, airliners, robots, and so on.

The first plot plots the output of the Kalman filter against the measurements and the actual position of our dog (labelled *Track*). After the initial settling in period the filter should track the dog's position very closely. The yellow shaded portion between the black dotted lines shows 1 standard deviations of the filter's variance, which I explain in the next paragraph.

The next two plots show the variance of $x$ and of $\dot{x}$. I have plotted the diagonals of $\mathbf{P}$ over time. Recall that the diagonal of a covariance matrix contains the variance of each state variable. So $\mathbf{P}[0,0]$ is the variance of $x$, and $\mathbf{P}[1,1]$ is the variance of $\dot{x}$. You can see that we quickly converge to small variances for both.

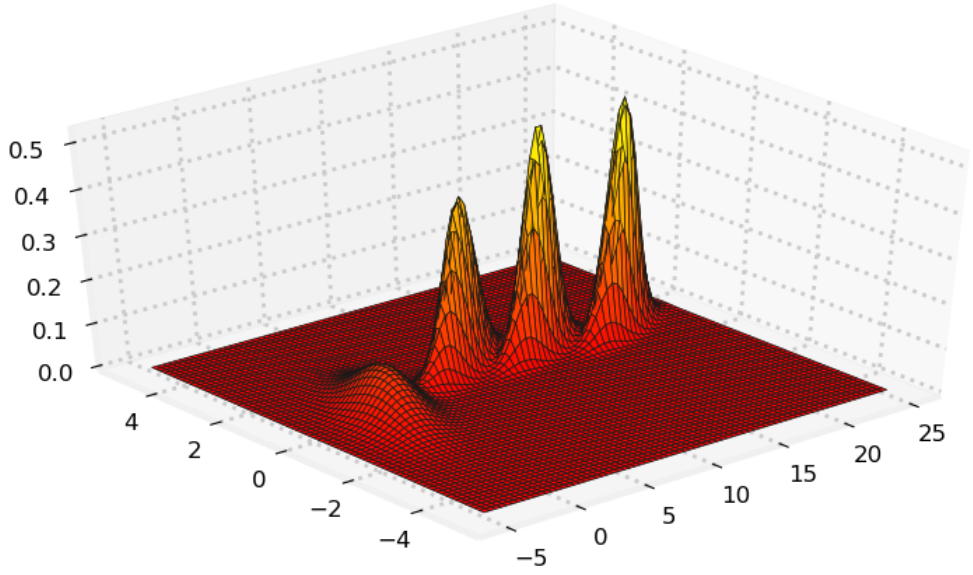
The covariance matrix $\mathbf{P}$ tells us the *theoretical* performance of the filter *assuming* everything we tell it is true. Recall that the standard deviation is the square root of the variance, and that approximately 68% of a Gaussian distribution occurs within one standard deviation. If at least 68% of the filter output is within one standard deviation the filter may be performing well. In the top chart I have displayed the one standard deviation as the yellow shaded area between the two dotted lines. To my eye it looks like perhaps the filter is slightly exceeding that bounds, so the filter probably needs some tuning.

In the univariate chapter we filtered very noisy signals with much simpler code than the code above. However, realize that right now we are working with a very simple example - an object moving through 1-D space and one sensor. That is about the limit of what we can compute with the code in the last chapter. In contrast, we can implement very complicated, multidimensional filters with this code merely by altering our assignments to the filter's variables. Perhaps we want to track 100 dimensions in financial models. Or we have an aircraft with a GPS, INS, TACAN, radar altimeter, baro altimeter, and airspeed indicator, and we want to integrate all those sensors into a model that predicts position, velocity, and acceleration in 3D space. We can do that with the code in this chapter.

I want you to get a better feel for how the Gaussians change over time, so here is a 3D plot showing the Gaussians every 7th epoch (time step). Every 7th separates them enough so can see each one independently. The first Gaussian at $t = 0$ is to the left.

```
In [32]: from kf_book.book_plots import set_figsize, figsize
         from kf_book.nonlinear_plots import plot_gaussians

         P = np.diag([3., 1.])
         np.random.seed(3)
         Ms, Ps = run(count=25, R=10, Q=0.01, P=P, do_plot=False)
```

```
with figsize(x=9, y=5):
    plot_gaussians(Ms[::7], Ps[::7], (-5,25), (-5, 5), 75)
```




## 6.8 The Saver Class

In the `run()` method I wrote boilerplate code to save the results of the filter

```
xs, cov = [], []
for z in zs:
    kf.predict()
    kf.update(z)
    xs.append(kf.x)
    cov.append(kf.P)

xs, cov = np.array(xs), np.array(cov)
```

There's an easy way to avoid this. `filtery.common` provides the `Saver` class which will save all attributes in the Kalman filter class each time `Saver.save()` is called. Let's see it in action and then we will talk about it more.

```
In [33]: from filterpy.common import Saver
         kf = pos_vel_filter([0, .1], R=R, P=P, Q=Q, dt=1.)
         s = Saver(kf)
         for i in range(1, 6):
             kf.predict()
             kf.update([i])
             s.save()  # save the current state
```

The `Saver` object now contains lists of all the attributes of the KalmanFilter object. `kf.x` is the current state estimate of the filter. Therefore `s.x` contains the saved state estimate that was computed inside the loop:

```
In [34]: s.x
```

```
Out[34]: [array([0.531, 0.304]),
          array([1.555, 0.763]),
          array([2.784, 1.036]),
          array([3.944, 1.105]),
          array([5.015, 1.086])]
```

You can see all the available attributes with the `keys` attribute:

```
In [35]: s.keys
```

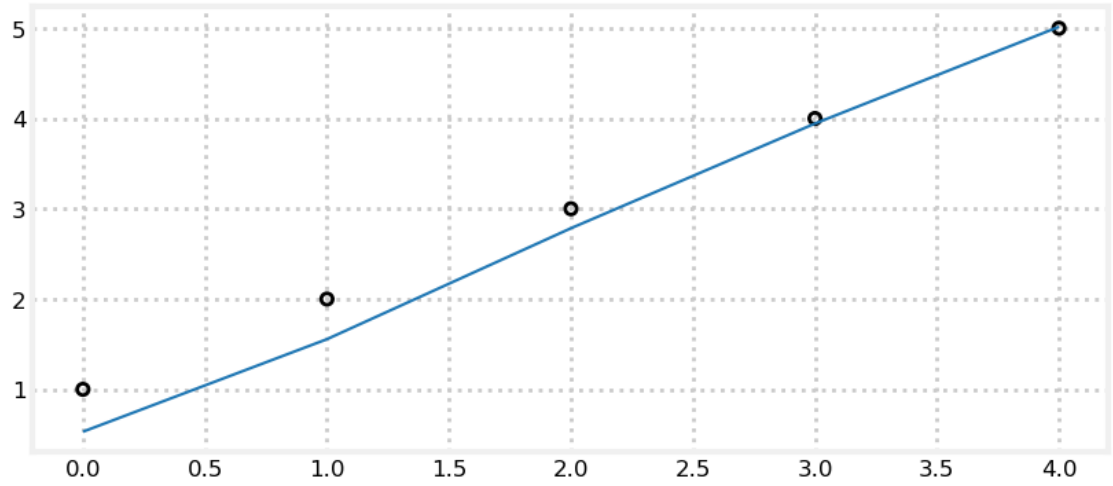
```
Out[35]: ['alpha',
          'likelihood',
          'log_likelihood',
          'mahalanobis',
          'dim_x',
          'dim_z',
          'dim_u',
          'x',
          'P',
          'Q',
          'B',
          'F',
          'H',
          'R',
          '_alpha_sq',
          'M',
          'z',
          'K',
          'y',
          'S',
          'SI',
          '_I',
          'x_prior',
          'P_prior',
          'x_post',
          'P_post',
          '_log_likelihood',
          '_likelihood',
          '_mahalanobis',
          'inv']
```

There are many attributes there that we haven't discussed yet, but many should be familar.

At this point you could write code to plot any of these variables. However, it is often more useful to use `np.array` instead of lists. Calling `Saver.to_array()` will convert the lists into `np.array`. There is one caveat: if the shape of any of the attributes changes during the run, the `to_array` will raise an exception since `np.array` requires all of the elements to be of the same type and size.

If you look at the keys again you'll see that `z` is one of the choices. This is promising; apparently the measurement `z` is saved for us. Let's plot it against the estimate.

```
In [36]: import matplotlib.pyplot as plt

         s.to_array()
         book_plots.plot_measurements(s.z);
         plt.plot(s.x[:, 0]);
```




While I've demonstrated this with the `KalmanFilter` class, it will work with all filter classes implemented by `FilterPy`. It will probably work with any class you write as well, as it inspects the object to retrieve the attribute names. We will use this class throughout the book to keep the code readable and short. Using the `Saver` will slow down your code because a lot happens behind the scenes, but for learning and exploring the convience cannot be beat.

## 6.9 The Kalman Filter Equations

We are now ready to learn how `predict()` and `update()` perform their computations.

A word about notation. I'm a programmer, and I am used to code that reads

```
x = x + 1
```

That is not an equation as the sides are not equal, but an *assignment*. If we wanted to write this in mathematical notation we'd write

$$x_k = x_{k-1} + 1$$

Kalman filter equations are littered with subscripts and superscripts to keep the equations mathematically consistent. I find this makes them extremely hard to read. In most of the book I opt for subscriptless assignments. As a programmer you should understand that I am showing you assignments which implement an algorithm that is to be executed step by step. I'll elaborate on this once we have a concrete example.

### 6.9.1 Prediction Equations

The Kalman filter uses these equations to compute the *prior* - the predicted next state of the system. They compute the prior mean ($\bar{\mathbf{x}}$) and covariance ($\bar{\mathbf{P}}$) of the system.

$$\mathbf{x} = \mathbf{F}\mathbf{x} + \mathbf{B}\mathbf{u}$$
$$\mathbf{P} = \mathbf{F}\mathbf{P}\mathbf{F}^\mathsf{T} + \mathbf{Q}$$

**Mean**

$\mathbf{x} = \mathbf{F}\mathbf{x} + \mathbf{B}\mathbf{u}$

As a reminder, the linear equation $\mathbf{A}\mathbf{x} = \mathbf{b}$ represents a system of equations, where $\mathbf{A}$ holds the coefficients set of equations, $\mathbf{x}$ is the vector of variables. Performing the multiplication $\mathbf{A}\mathbf{x}$ computes the right hand side values for that set of equations, represented by $\mathbf{b}$.

If $\mathbf{F}$ contains the state transition for a given time step, then the product $\mathbf{F}\mathbf{x}$ computes the state after that transition. Easy! Likewise, $\mathbf{B}$ is the control function, $\mathbf{u}$ is the control input, so $\mathbf{B}\mathbf{u}$ computes the

contribution of the controls to the state after the transition. Thus, the prior $\mathbf{x}$ is computed as the sum of $\mathbf{Fx}$ and $\mathbf{Bu}$.

The equivalent univariate equation is

$$\bar{\mu} = \mu + \mu_{move}$$

If you perform the matrix multiplication $\mathbf{Fx}$ it generates this equation for $x$.

Let's make this explicit. Recall the value for $\mathbf{F}$ from the last chapter:

$$\mathbf{F} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$$

Thus $\mathbf{x} = \mathbf{Fx}$ corresponds to the set of linear equations:

$$\begin{cases} \bar{x} = 1x + \Delta t\, \dot{x} \\ \bar{\dot{x}} = 0x + 1\, \dot{x} \end{cases}$$

**Covariance**

$\mathbf{P} = \mathbf{FPF}^\mathsf{T} + \mathbf{Q}$

This equation is not as easy to understand so we will spend more time on it.

In univariate version of this equation is:

$$\bar{\sigma}^2 = \sigma^2 + \sigma^2_{move}$$

We add the variance of the movement to the variance of our estimate to reflect the loss of knowlege. We need to do the same thing here, except it isn't quite that easy with multivariate Gaussians.

We can't simply write $\mathbf{P} = \mathbf{P} + \mathbf{Q}$. In a multivariate Gaussians the state variables are *correlated*. What does this imply? Our knowledge of the velocity is imperfect, but we are adding it to the position with

$$\bar{x} = \dot{x}\Delta t + x$$

Since we do not have perfect knowledge of the value of $\dot{x}$ the sum $\bar{x} = \dot{x}\Delta t + x$ gains uncertainty. Because the positions and velocities are correlated we cannot simply add the covariance matrices. For example, if $\mathbf{P}$ and $\mathbf{Q}$ are diagonal matrices the sum would also be diagonal. But we know position is correlated to velocity so the off-diagonal elements should be non-zero.

The correct equation is

$$\mathbf{P} = \mathbf{FPF}^\mathsf{T} + \mathbf{Q}$$

Expressions in the form $\mathbf{ABA}^\mathsf{T}$ are common in linear algebra. You can think of it as *projecting* the middle term by the outer term. We will be using this many times in the rest of the book. I admit this may be a 'magical' equation to you. Let's explore it.

When we initialize $\mathbf{P}$ with

$$\mathbf{P} = \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_v^2 \end{bmatrix}$$

the value for $\mathbf{FPF}^\mathsf{T}$ is:

$$\begin{aligned} \mathbf{FPF}^\mathsf{T} &= \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \sigma_x^2 & 0 \\ 0 & \sigma_v^2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \Delta t & 1 \end{bmatrix} \\ &= \begin{bmatrix} \sigma_x^2 & \sigma_v^2 \Delta t \\ 0 & \sigma_v^2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ \Delta t & 1 \end{bmatrix} \\ &= \begin{bmatrix} \sigma_x^2 + \sigma_v^2 \Delta t^2 & \sigma_v^2 \Delta t \\ \sigma_v^2 \Delta t & \sigma_v^2 \end{bmatrix} \end{aligned}$$

The initial value for $\mathbf{P}$ had no covariance between the position and velocity. Position is computed as $\dot{x}\Delta t + x$, so there is a correlation between the position and velocity. The multiplication $\mathbf{FPF}^\mathsf{T}$ computes

a covariance of $\sigma_v^2\Delta t$. The exact value is not important; you just need to recognize that $\mathbf{FPF}^\mathsf{T}$ uses the process model to automatically compute the covariance between the position and velocity!

Another way to think of this is to reflect on the $\mathbf{Fx}$ multiplication. That projected $\mathbf{x}$ forward in time. $\mathbf{FP}$ might seem to be the equivalent operation, but $\mathbf{P}$ is a matrix while $\mathbf{x}$ is a vector. The trailing $\mathbf{F}^\mathsf{T}$ term ensures that we multiply by both the rows and columns of $\mathbf{F}$. In the second line of the computation of $\mathbf{FPF}^\mathsf{T}$ we have the value for $\mathbf{FP}$. You can see that it is an upper triangular matrix because we haven't fully incorporated $\mathbf{F}$ into the multiplication.

If you have some experience with linear algebra and statistics, this may help. The covariance due to the prediction can be modeled as the expected value of the error in the prediction step, given by this equation.

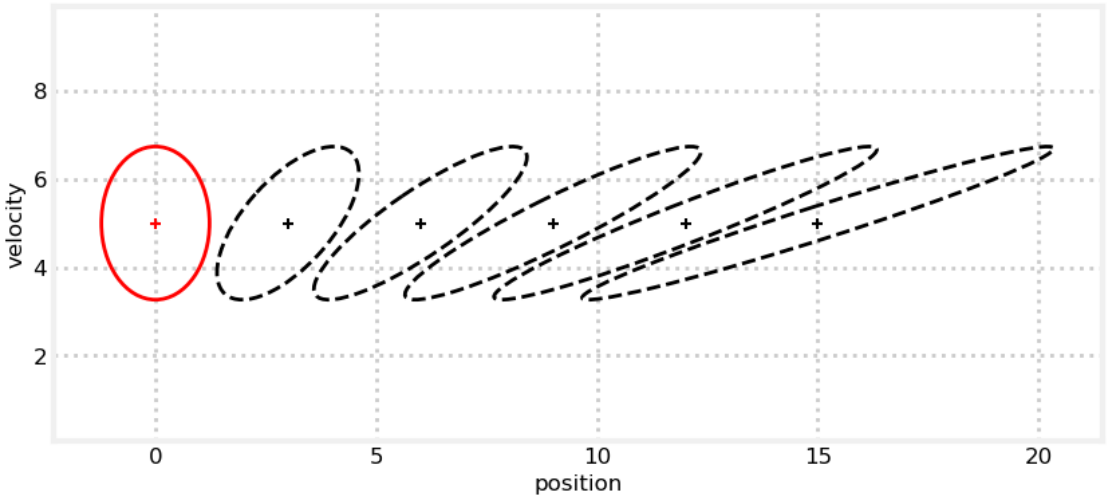
$$\begin{aligned}\bar{\mathbf{P}} &= \mathbb{E}[(\mathbf{Fx} - \mu)(\mathbf{Fx} - \mu)^\mathsf{T}] \\ &= \mathbf{F}\,\mathbb{E}[(\mathbf{x} - \mu)(\mathbf{x} - \mu)^\mathsf{T}]\,\mathbf{F}^\mathsf{T}\end{aligned}$$

Of course, $\mathbb{E}[(\mathbf{x} - \mu)(\mathbf{x} - \mu)^\mathsf{T}]$ is just $\mathbf{P}$, giving us

$$\bar{\mathbf{P}} = \mathbf{FPF}^\mathsf{T}$$

Let's look at its effect. Here I use $\mathbf{F}$ from our filter and project the state forward 6/10ths of a second. I do this five times so you can see how $\mathbf{P}$ continues to change.

```
In [37]: dt = 0.6
         x = np.array([0., 5.])
         F = np.array([[1., dt], [0, 1.]])
         P = np.array([[1.5, 0], [0, 3.]])
         plot_covariance_ellipse(x, P, edgecolor='r')

         for _ in range(5):
             x = np.dot(F, x)
             P = np.dot(F, P).dot(F.T)
             plot_covariance_ellipse(x, P, edgecolor='k', ls='dashed')
         book_plots.set_labels(x='position', y='velocity')
```



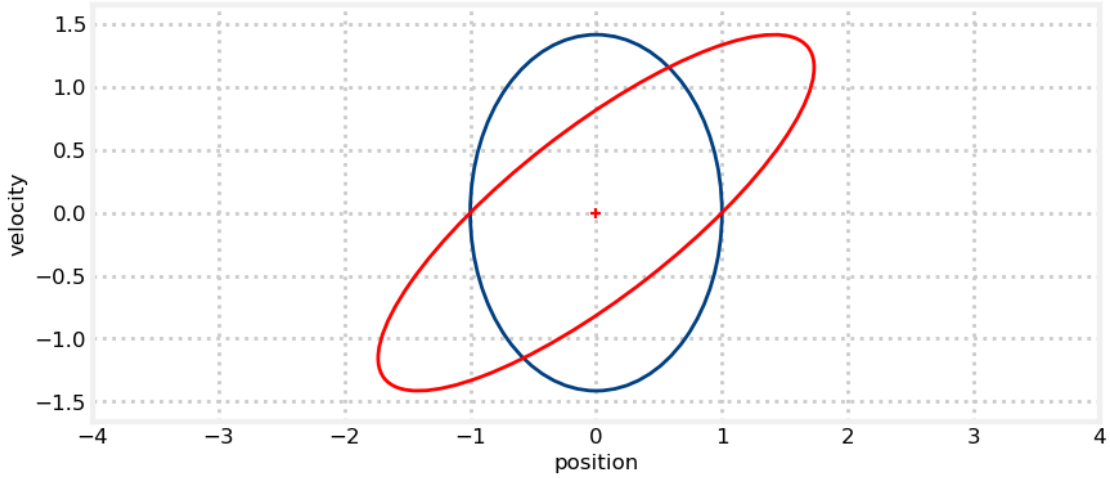

You can see that with a velocity of 5 the position correctly moves 3 units in each 6/10ths of a second step. At each step the width of the ellipse is larger, indicating that we have lost information about the position due to adding $\dot{x}\Delta t$ to x at each step. The height has not changed - our system model says the velocity does not change, so the belief we have about the velocity cannot change. As time continues you can see that the

ellipse becomes more and more tilted. Recall that a tilt indicates *correlation*. $\mathbf{F}$ linearly correlates $x$ with $\dot{x}$ with the expression $\bar{x} = \dot{x}\Delta t + x$. The $\mathbf{FPF}^\mathsf{T}$ computation correctly incorporates this correlation into the covariance matrix.

Here is an animation of this equation that allows you to change the design of $\mathbf{F}$ to see how it affects shape of $\mathbf{P}$. The `F00` slider affects the value of F[0, 0]. `covar` sets the intial covariance between the position and velocity($\sigma_x\sigma_{\dot{x}}$). I recommend answering these questions at a minimum

- what if $x$ is not correlated to $\dot{x}$? (set F01 to 0, the rest at defaults)
- what if $x = 2\dot{x}\Delta t + x_0$? (set F01 to 2, the rest at defaults)
- what if $x = \dot{x}\Delta t + 2x_0$? (set F00 to 2, the rest at defaults)
- what if $x = \dot{x}\Delta t$? (set F00 to 0, the rest at defaults)

```
In [38]: from ipywidgets import interact
         from kf_book.book_plots import IntSlider, FloatSlider

         def plot_FPFT(F00, F01, F10, F11, covar):
             plt.figure()
             dt = 1.
             x = np.array((0, 0.))
             P = np.array(((1, covar), (covar, 2)))
             F = np.array(((F00, F01), (F10, F11)))
             plot_covariance_ellipse(x, P)
             plot_covariance_ellipse(x, np.dot(F, P).dot(F.T), ec='r')
             plt.gca().set_aspect('equal')
             plt.xlim(-4, 4)
             plt.ylim(-4, 4)
             #plt.title(str(F))
             plt.xlabel('position')
             plt.ylabel('velocity')

         interact(plot_FPFT,
                  F00=IntSlider(value=1, min=0, max=2),
                  F01=FloatSlider(value=1, min=0, max=2, description='F01(dt)'),
                  F10=FloatSlider(value=0, min=0, max=2),
                  F11=FloatSlider(value=1, min=0, max=2),
                  covar=FloatSlider(value=0, min=0, max=1));
```



(If you are reading this in a static form: instructions to run this online are here: https://git.io/vza7b). Or, go to binder using the link below, and open this notebook from there.

http://mybinder.org/repo/rlabbe/Kalman-and-Bayesian-Filters-in-Python

### 6.9.2 Update Equations

The update equations look messier than the predict equations, but that is mostly due to the Kalman filter computing the update in measurement space. This is because measurements are not *invertible*. For example, consider a sensor that gives the range to a target. It is impossible to convert a range into a position - an infinite number of positions in a circle will yield the same range. On the other hand, we can always compute the range (measurement) given a position (state).

Before I continue, recall that we are trying to do something very simple: choose a new estimate chosen somewhere between a measurement and a prediction, as in this chart:

The equations will be complicated because the state has multiple dimensions, but this operations is what we are doing. Don't let the equations distract you from the simplicity of this idea.

**System Uncertainty**

$\mathbf{S} = \mathbf{HPH}^\mathsf{T} + \mathbf{R}$

To work in measurement space the Kalman filter has to project the covariance matrix into measurement space. The math for this is $\mathbf{HPH}^\mathsf{T}$, where $\mathbf{P}$ is the *prior* covariance and $\mathbf{H}$ is the measurement function.

You should recognize this $\mathbf{ABA}^\mathsf{T}$ form - the prediction step used $\mathbf{FPF}^\mathsf{T}$ to update $\mathbf{P}$ with the state transition function. Here, we use the same form to update it with the measurement function. The linear algebra is changing the coordinate system for us.

Once the covariance is in measurement space we need to account for the sensor noise. This is very easy - we just add matrices. The result is variously called the *system uncertainty* or *innovation covariance*.

If you ignore the $\mathbf{H}$ term this equation is the equivalent to the denominator in the univariate equation for the Kalman gain:

$$K = \frac{\bar\sigma^2}{\bar\sigma^2 + \sigma_z^2}$$

Compare the equations for the system uncertainty and the covariance

$$\mathbf{S} = \mathbf{HPH}^\mathsf{T} + \mathbf{R}$$
$$\mathbf{P} = \mathbf{FPF}^\mathsf{T} + \mathbf{Q}$$

In each equation $\mathbf{P}$ is put into a different space with either the function $\mathbf{H}$ or $\mathbf{F}$. Then we add the noise matrix associated with that space.

**Kalman Gain**

$\mathbf{K} = \mathbf{PH}^\mathsf{T}\mathbf{S}^{-1}$

Look back at the residual diagram. Once we have a prediction and a measurement we need to select an estimate somewhere between the two. If we have more certainty about the measurement the estimate will be closer to it. If instead we have more certainty about the prediction then the estimate will be closer to it.

In the univariate chapter we scaled the mean using this equation

$$\mu = \frac{\bar\sigma^2\mu_z + \sigma_\mathbf{z}^2\bar\mu}{\bar\sigma^2 + \sigma_\mathbf{z}^2}$$

which we simplified to

$$\mu = (1-K)\bar\mu + K\mu_\mathbf{z}$$

which gave us

$$K = \frac{\bar\sigma^2}{\bar\sigma^2 + \sigma_z^2}$$

$K$ is the *Kalman gain*, and it is a real number between 0 and 1. Ensure you understand how it selects a mean somewhere between the prediction and measurement. The Kalman gain is a *percentage* or *ratio* - if K is .9 it takes 90% of the measurement and 10% of the prediction.

For the multivariate Kalman filter $\mathbf{K}$ is a vector, not a scalar. Here is the equation again: $\mathbf{K} = \mathbf{P}\mathbf{H}^\mathsf{T}\mathbf{S}^{-1}$. Is this a *ratio*? We can think of the inverse of a matrix as linear algebra's way of finding the reciprocal. Division is not defined for matrices, but it is useful to think of it in this way. So we can read the equation for $\mathbf{K}$ as meaning

$$\mathbf{K} \approx \frac{\mathbf{P}\mathbf{H}^\mathsf{T}}{\mathbf{S}}$$

$$\mathbf{K} \approx \frac{\mathsf{uncertainty}_{\mathsf{prediction}}}{\mathsf{uncertainty}_{\mathsf{prediction}} + \mathsf{uncertainty}_{\mathsf{measurement}}}\mathbf{H}^\mathsf{T}$$

The Kalman gain equation computes a ratio based on how much we trust the prediction vs the measurement. We did the same thing in every prior chapter. The equation is complicated because we are doing this in multiple dimensions via matrices, but the concept is simple. The $\mathbf{H}^\mathsf{T}$ term is less clear, I'll explain it soon. If you ignore that term the equation for the Kalman gain is the same as the univariate case: divide the uncertainty of the prior with the of the sum of the uncertainty of the prior and measurement.

**Residual**

$\mathbf{y} = \mathbf{z} - \mathbf{H}\mathbf{x}$

This is an easy one as we've covered this equation while designing the measurement function $\mathbf{H}$. Recall that the measurement function converts a state into a measurement. So $\mathbf{H}\mathbf{x}$ converts $\mathbf{x}$ into an equivalent measurement. Once that is done, we can subtract it from the measurement $\mathbf{z}$ to get the residual - the difference between the measurement and prediction.

The univariate equation is

$$y = z - \bar{x}$$

and clearly computes the same thing, but only in one dimension.

**State Update**

$\mathbf{x} = \mathbf{x} + \mathbf{K}\mathbf{y}$

We select our new state to be along the residual, scaled by the Kalman gain. The scaling is performed by $\mathbf{K}\mathbf{y}$, which both scales the residual and converts it back into state space with the $\mathbf{H}^\mathsf{T}$ term which is in $\mathbf{K}$. This is added to the prior, yielding the equation: $\mathbf{x} = \mathbf{x} + \mathbf{K}\mathbf{y}$. Let me write out $\mathbf{K}$ so we can see the entire computation:

$$\begin{aligned}
\mathbf{x} &= \mathbf{x} + \mathbf{K}\mathbf{y} \\
&= \mathbf{x} + \mathbf{P}\mathbf{H}^\mathsf{T}\mathbf{S}^{-1}\mathbf{y} \\
&\approx \mathbf{x} + \frac{\mathsf{uncertainty}_{\mathsf{prediction}}}{\mathsf{uncertainty}_{\mathsf{measurement}}}\mathbf{H}^\mathsf{T}\mathbf{y}
\end{aligned}$$

Perhaps a better way to *see* the ratio is to rewrite the estimate equation:

$$\begin{aligned}
\mathbf{x} &= \mathbf{x} + \mathbf{K}\mathbf{y} \\
&= \mathbf{x} + \mathbf{K}(\mathbf{z} - \mathbf{H}\mathbf{x}) \\
&= (\mathbf{I} - \mathbf{K}\mathbf{H})\mathbf{x} + \mathbf{K}\mathbf{z}
\end{aligned}$$

The similarity between this and the univariate form should be obvious:

$$\mu = (1 - K)\bar{\mu} + K\mu_\mathbf{z}$$

**Covariance Update**

$\mathbf{P} = (\mathbf{I} - \mathbf{K}\mathbf{H})\mathbf{P}$

$\mathbf{I}$ is the identity matrix, and is the way we represent 1 in multiple dimensions. $\mathbf{H}$ is our measurement function, and is a constant. We can think of the equation as $\mathbf{P} = (1 - c\mathbf{K})\mathbf{P}$. $\mathbf{K}$ is our ratio of how much prediction vs measurement we use. If $\mathbf{K}$ is large then $(1 - \mathbf{c}\mathbf{K})$ is small, and $\mathbf{P}$ will be made smaller than it

was. If $\mathbf{K}$ is small, then $(1 - \mathbf{cK})$ is large, and $\mathbf{P}$ will be relatively larger. This means that we adjust the size of our uncertainty by some factor of the Kalman gain.

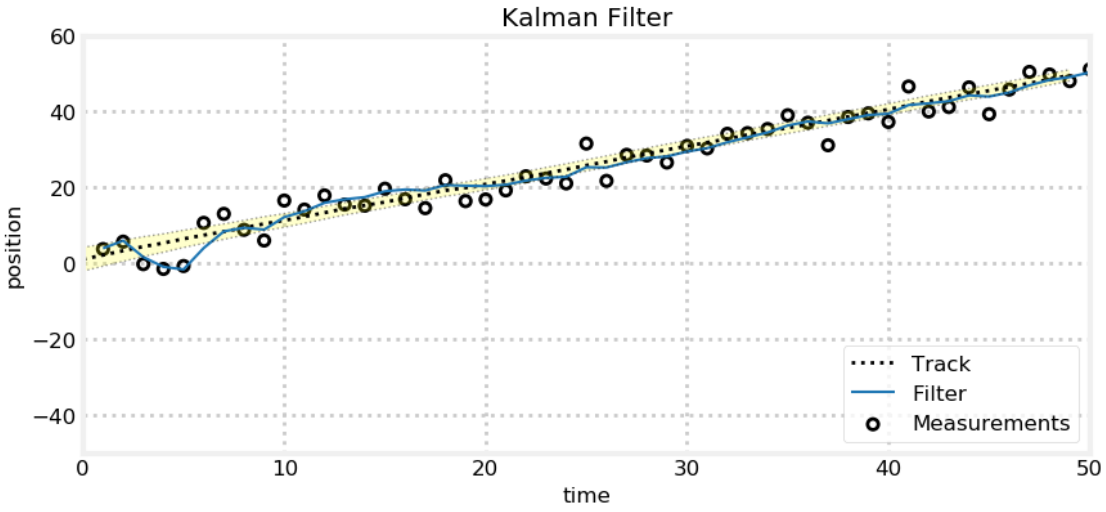
This equation can be numerically unstable and I don't use it in FilterPy. The subtraction can destroy symmetry and lead to floating point errors over time. Later I'll share more complicated but numerically stable forms of this equation.

### 6.9.3 An Example not using FilterPy

FilterPy hides the details of the implementation from us. Normally you will appreciate this, but let's implement the last filter without FilterPy. To do so we need to define our matrices as variables, and then implement the Kalman filter equations explicitly.

Here we initialize our matrices:

```
In [39]: dt = 1.
         R_var = 10
         Q_var = 0.01
         x = np.array([[10.0, 4.5]]).T
         P = np.diag([500, 49])
         F = np.array([[1, dt],
                       [0,  1]])
         H = np.array([[1., 0.]])
         R = np.array([[R_var]])
         Q = Q_discrete_white_noise(dim=2, dt=dt, var=Q_var)

In [40]: from numpy import dot
         from scipy.linalg import inv

         count = 50
         track, zs = compute_dog_data(R_var, Q_var, count)
         xs, cov = [], []
         for z in zs:
             # predict
             x = dot(F, x)
             P = dot(F, P).dot(F.T) + Q

             #update
             S = dot(H, P).dot(H.T) + R
             K = dot(P, H.T).dot(inv(S))
             y = z - dot(H, x)
             x += dot(K, y)
             P = P - dot(K, H).dot(P)

             xs.append(x)
             cov.append(P)

         xs, cov = np.array(xs), np.array(cov)
         plot_track(xs[:, 0], track, zs, cov, plot_P=False)
```




The results are identical to the FilterPy version. Which you prefer is up to you. I prefer not polluting my namespace with variables such as `x`, `P`, and so on; `dog_filter.x` is, to me, more readable.

More importantly, this example requires you to remember and program the equations for the Kalman filter. Sooner or later you will make a mistake. FilterPy's version ensures that your code will be correct. On the other hand, if you make a mistake in your definitions, such as making $\mathbf{H}$ a column vector instead of a row vector, FilterPy's error message will be harder to debug than this explicit code.

FilterPy's KalmanFilter class provides additional functionality such as smoothing, batch processing, faded memory filtering, computation of the maximum likelihood function, and more. You get all of this functionality without having to explicitly program it.

### 6.9.4 Summary

We have learned the Kalman filter equations. Here they are all together for your review. There was a lot to learn, but I hope that as you went through each you recognized it's kinship with the equations in the univariate filter. In the *Kalman Math* chapter I will show you that if we set the dimension of $\mathbf{x}$ to one that these equations revert back to the equations for the univariate filter. This is not "like" the univariate filter - it is a multidimensional implementation of it.

Predict Step

$$\mathbf{x} = \mathbf{Fx} + \mathbf{Bu}$$
$$\mathbf{P} = \mathbf{FPF}^\mathsf{T} + \mathbf{Q}$$

Update Step

$$\mathbf{S} = \mathbf{HPH}^\mathsf{T} + \mathbf{R}$$
$$\mathbf{K} = \mathbf{PH}^\mathsf{T}\mathbf{S}^{-1}$$
$$\mathbf{y} = \mathbf{z} - \mathbf{Hx}$$
$$\mathbf{x} = \mathbf{x} + \mathbf{Ky}$$
$$\mathbf{P} = (\mathbf{I} - \mathbf{KH})\mathbf{P}$$

I want to share a form of the equations that you will see in the literature. There are many different notation systems used, but this gives you an idea of what to expect.

$$
\begin{aligned}
\hat{\mathbf{x}}_{k|k-1} &= \mathbf{F}_k\hat{\mathbf{x}}_{k-1|k-1} + \mathbf{B}_k\mathbf{u}_k \\
\mathbf{P}_{k|k-1} &= \mathbf{F}_k\mathbf{P}_{k-1|k-1}\mathbf{F}_k^\mathsf{T} + \mathbf{Q}_k \\
\tilde{\mathbf{y}}_k &= \mathbf{z}_k - \mathbf{H}_k\hat{\mathbf{x}}_{k|k-1} \\
\mathbf{S}_k &= \mathbf{H}_k\mathbf{P}_{k|k-1}\mathbf{H}_k^\mathsf{T} + \mathbf{R}_k \\
\mathbf{K}_k &= \mathbf{P}_{k|k-1}\mathbf{H}_k^\mathsf{T}\mathbf{S}_k^{-1} \\
\hat{\mathbf{x}}_{k|k} &= \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k\tilde{\mathbf{y}}_k \\
\mathbf{P}_{k|k} &= (I - \mathbf{K}_k\mathbf{H}_k)\mathbf{P}_{k|k-1}
\end{aligned}
$$

This notation uses the Bayesian $a \mid b$ notation, which means $a$ given the evidence of $b$. The hat means estimate. Thus $\hat{\mathbf{x}}_{k|k}$ means the estimate of the state $\mathbf{x}$ at step $k$ (the first k) given the evidence from step $k$ (the second k). The posterior, in other words. $\hat{\mathbf{x}}_{k|k-1}$ means the estimate for the state $\mathbf{x}$ at step $k$ given the estimate from step $k-1$. The prior, in other words.

This notation, copied from Wikipedia Section **??**, allows a mathematician to express himself exactly. In formal publications presenting new results this precision is necessary. As a programmer I find it fairly unreadable. I am used to thinking about variables changing state as a program runs, and do not use a different variable name for each new computation. There is no agreed upon format in the literature, so each author makes different choices. I find it challenging to switch quickly between books and papers, and so have adopted my admittedly less precise notation. Mathematicians may write scathing emails to me, but I hope programmers and students will rejoice at my simplified notation.

The **Symbology** Appendix lists the notation used by various authors. This brings up another difficulty. Different authors use different variable names. $\mathbf{x}$ is fairly universal, but after that it is anybody's guess. For example, it is common to use $\mathbf{A}$ for what I call $\mathbf{F}$. You must read carefully, and hope that the author defines their variables (they often do not).

If you are a programmer trying to understand a paper's equations, I suggest starting by removing all of the superscripts, subscripts, and diacriticals, replacing them with a single letter. If you work with equations like this every day this is superfluous advice, but when I read I am usually trying to understand the flow of computation. To me it is far more understandable to remember that $P$ in this step represents the updated value of $P$ computed in the last step, as opposed to trying to remember what $P_{k-1}(+)$ denotes, and what its relation to $P_k(-)$ is, if any, and how any of that relates to the completely different notation used in the paper I read 5 minutes ago.

## 6.10 Exercise: Show Effect of Hidden Variables

In our filter velocity is a hidden variable. How would a filter perform if we did not use velocity in the state?

Write a Kalman filter that uses the state $\mathbf{x} = \begin{bmatrix}x\end{bmatrix}$ and compare it against a filter that uses $\mathbf{x} = \begin{bmatrix}x & \dot{x}\end{bmatrix}^\mathsf{T}$.

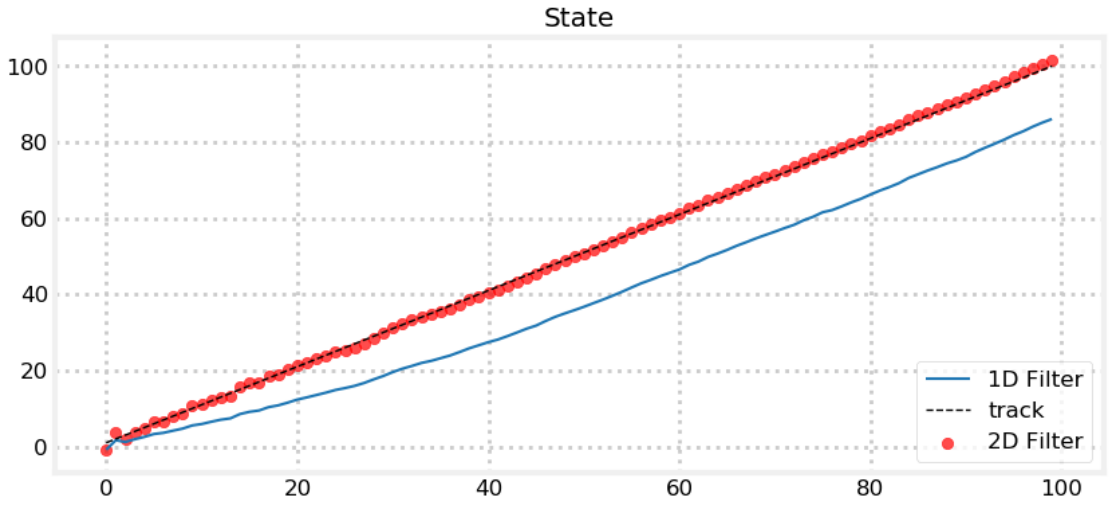
```
In [41]: # your code here
```

### 6.10.1 Solution

We've already implemented a Kalman filter for position and velocity, so I will provide the code without much comment, and then plot the result.

```
In [42]: from math import sqrt
         from numpy.random import randn

         def univariate_filter(x0, P, R, Q):
             f = KalmanFilter(dim_x=1, dim_z=1, dim_u=1)
             f.x = np.array([[x0]])
             f.P *= P
             f.H = np.array([[1.]])
             f.F = np.array([[1.]])
```

```
    f.B = np.array([[1.]])
    f.Q *= Q
    f.R *= R
    return f

def plot_1d_2d(xs, xs1d, xs2d):
    plt.plot(xs1d, label='1D Filter')
    plt.scatter(range(len(xs2d)), xs2d, c='r', alpha=0.7, label='2D Filter')
    plt.plot(xs, ls='--', color='k', lw=1, label='track')
    plt.title('State')
    plt.legend(loc=4)
    plt.show()

def compare_1D_2D(x0, P, R, Q, vel, u=None):
    # storage for filter output
    xs, xs1, xs2 = [], [], []

    # 1d KalmanFilter
    f1D = univariate_filter(x0, P, R, Q)

    #2D Kalman filter
    f2D = pos_vel_filter(x=(x0, vel), P=P, R=R, Q=0)
    if np.isscalar(u):
        u = [u]
    pos = 0 # true position
    for i in range(100):
        pos += vel
        xs.append(pos)

        # control input u - discussed below
        f1D.predict(u=u)
        f2D.predict()

        z = pos + randn()*sqrt(R) # measurement
        f1D.update(z)
        f2D.update(z)

        xs1.append(f1D.x[0])
        xs2.append(f2D.x[0])
    plt.figure()
    plot_1d_2d(xs, xs1, xs2)

compare_1D_2D(x0=0, P=50., R=5., Q=.02, vel=1.)
```




### 6.10.2 Discussion

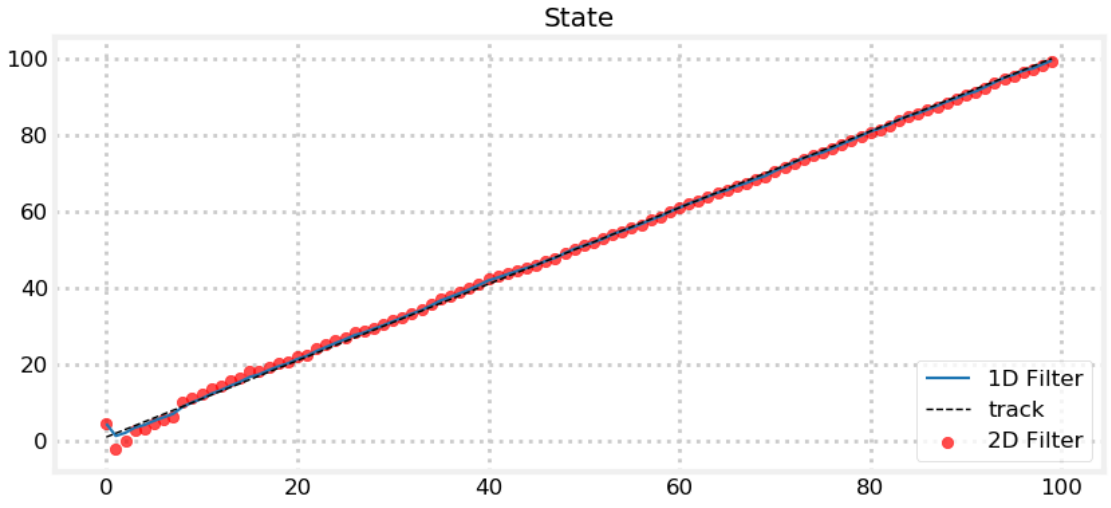
The filter that incorporates velocity into the state produces much better estimates than the filter that only tracks position. The univariate filter has no way to estimate the velocity or change in position, so it lags the tracked object.

In the univarate Kalman filter chapter we had a control input `u` to the predict equation:

```
def predict(self, u=0.0):
    self.x += u
    self.P += self.Q
```

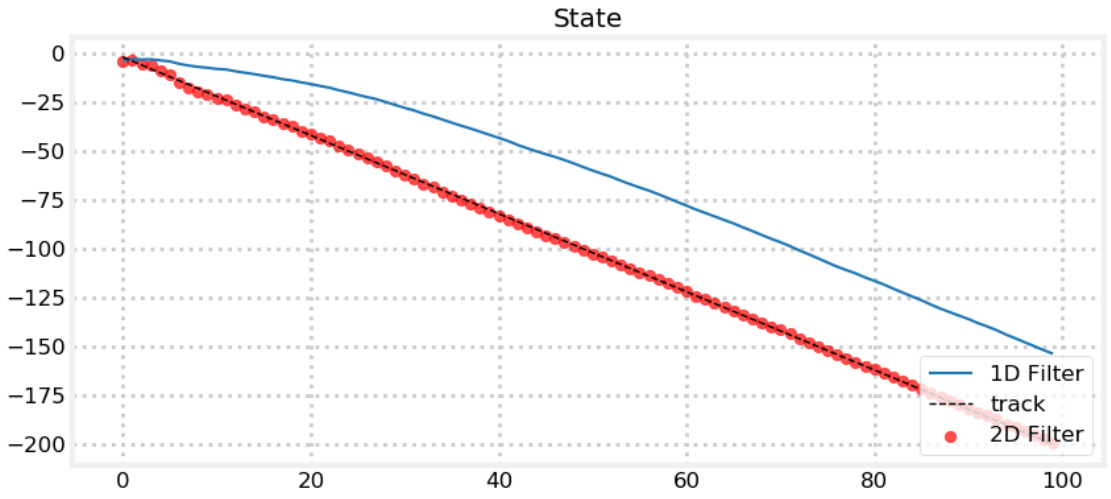
Let's try specifying the control input:

```
In [43]: compare_1D_2D(x0=0, P=50., R=5., Q=.02, vel=1., u=1.)
```




Here the performance of the two filters are similar, and perhaps the univariate filter is tracking more closely. But let's see what happens when the actual velocity `vel` is different from the control input `u`:

```
In [44]: compare_1D_2D(x0=0, P=50., R=5., Q=.02, vel=-2., u=1.)
```




If we are tracking a robot which we are also controlling the univariate filter can do a very good job because the control input allows the filter to make an accurate prediction. But if we are tracking passively the control input is not much help unless we can make an accurate *apriori* guess as to the velocity. This is rarely possible.

## 6.11 How Velocity is Calculated

I haven't explained how the filter computes the velocity, or any hidden variable. If we plug in the values we calculated for each of the filter's matrices we can see what happens.

First we need to compute the system uncertainty.

$$\begin{aligned}
\mathbf{S} &= \mathbf{HPH}^\mathsf{T} + \mathbf{R} \\
&= \begin{bmatrix}1 & 0\end{bmatrix} \begin{bmatrix}\sigma_x^2 & \sigma_{xv} \\ \sigma_{xv} & \sigma_v^2\end{bmatrix} \begin{bmatrix}1 \\ 0\end{bmatrix} + \begin{bmatrix}\sigma_z^2\end{bmatrix} \\
&= \begin{bmatrix}\sigma_x^2 & \sigma_{xv}\end{bmatrix} \begin{bmatrix}1 \\ 0\end{bmatrix} + \begin{bmatrix}\sigma_z^2\end{bmatrix} \\
&= \begin{bmatrix}\sigma_x^2 + \sigma_z^2\end{bmatrix}
\end{aligned}$$

Now that we have $\mathbf{S}$ we can find the value for the Kalman gain:

$$\begin{aligned}
\mathbf{K} &= \mathbf{PH}^\mathsf{T}\mathbf{S}^{-1} \\
&= \begin{bmatrix}\sigma_x^2 & \sigma_{xv} \\ \sigma_{xv} & \sigma_v^2\end{bmatrix} \begin{bmatrix}1 \\ 0\end{bmatrix} \begin{bmatrix}\frac{1}{\sigma_x^2 + \sigma_z^2}\end{bmatrix} \\
&= \begin{bmatrix}\sigma_x^2 \\ \sigma_{xv}\end{bmatrix} \begin{bmatrix}\frac{1}{\sigma_x^2 + \sigma_z^2}\end{bmatrix} \\
&= \begin{bmatrix}\sigma_x^2/(\sigma_x^2 + \sigma_z^2) \\ \sigma_{xv}/(\sigma_x^2 + \sigma_z^2)\end{bmatrix}
\end{aligned}$$

In other words, the Kalman gain for $x$ is

$$K_x = \frac{VAR(x)}{VAR(x) + VAR(z)}$$

This should be very familiar to you from the univariate case.
The Kalman gain for the velocity $\dot{x}$ is

$$K_{\dot{x}} = \frac{COV(x, \dot{x})}{VAR(x) + VAR(z)}$$

What is the effect of this? Recall that we compute the state as

$$\begin{aligned}\mathbf{x} &= \mathbf{x} + \mathbf{K}(z - \mathbf{H}\mathbf{x}) \\ &= \mathbf{x} + \mathbf{K}y\end{aligned}$$

Here the residual $y$ is a scalar. Therefore it is multiplied into each element of $\mathbf{K}$. Therefore we have

$$\begin{bmatrix} x \\ \dot{x} \end{bmatrix} = \begin{bmatrix} \bar{x} \\ \bar{\dot{x}} \end{bmatrix} + \begin{bmatrix} K_x \\ K_{\dot{x}} \end{bmatrix} y$$

Which gives this system of equations:

$$\begin{aligned} x &= \bar{x} + yK_x \\ \dot{x} &= \bar{\dot{x}} + yK_{\dot{x}} \end{aligned}$$

The prediction $\bar{x}$ was computed as $x + \bar{x}\Delta t$. If the prediction was perfect then the residual will be $y = 0$ (ignoring noise in the measurement) and the velocity estimate will be unchanged. On the other hand, if the velocity estimate was very bad then the prediction will be very bad, and the residual will be large: $y >> 0$. In this case we update the velocity estimate with $yK_{\dot{x}}$. $K_{\dot{x}}$ is proportional to $COV(x, \dot{x})$. Therefore the velocity is updated by the error in the position times the value proportional to the covariance between the position and velocity. The higher the correlation the larger the correction.

To bring this full circle, $COV(x, \dot{x})$ are the off-diagonal elements of $\mathbf{P}$. Recall that those values were computed with $\mathbf{FPF}^\mathsf{T}$. So the covariance of position and velocity is computed during the predict step. The Kalman gain for the velocity is proportional to this covariance, and we adjust the velocity estimate based on how inaccurate it was during the last epoch times a value proportional to this covariance.

In summary, these linear algebra equations may be unfamiliar to you, but computation is actually very simple. It is essentially the same computation that we performed in the g-h filter. Our constants are different in this chapter because we take the noise in the process model and sensors into account, but the math is the same.

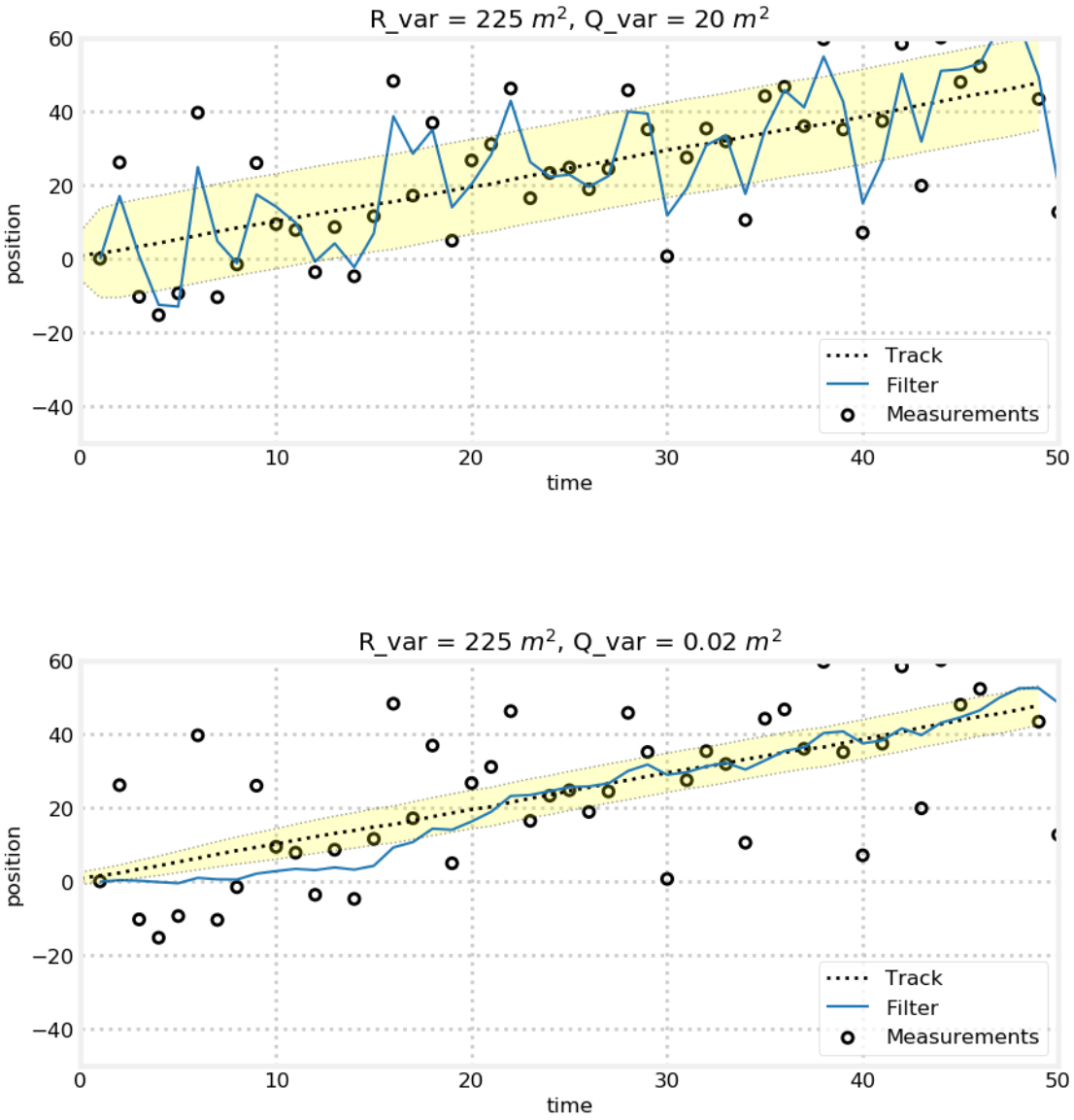
## 6.12 Adjusting the Filter

Let's start varying our parameters to see the effect of various changes. This is a very normal thing to be doing with Kalman filters. It is difficult, and often impossible to exactly model our sensors. An imperfect model means imperfect output from our filter. Engineers spend a lot of time tuning Kalman filters so that they perform well with real world sensors. We will spend time now to learn the effect of these changes. As you learn the effect of each change you will develop an intuition for how to design a Kalman filter. Designing a Kalman filter is as much art as science. We are modeling a physical system using math, and models are imperfect.

Let's look at the effects of the measurement noise $\mathbf{R}$ and process noise $\mathbf{Q}$. We will want to see the effect of different settings for $\mathbf{R}$ and $\mathbf{Q}$, so I have given the measurements a variance of 225 meters squared. That is very large, but it magnifies the effects of various design choices on the graphs, making it easier to recognize what is happening. Our first experiment holds $\mathbf{R}$ constant while varying $\mathbf{Q}$.

```
In [45]: from numpy.random import seed
         seed(2)
         trk, zs = compute_dog_data(z_var=225, process_var=.02, count=50)

         run(track=trk, zs=zs, R=225, Q=200, P=P, plot_P=False,
             title='R_var = 225 $m^2$, Q_var = 20 $m^2$')
```

```
run(track=trk, zs=zs, R=225, Q=.02, P=P, plot_P=False,
    title='R_var = 225 $m^2$, Q_var = 0.02 $m^2$');
```




The filter in the first plot should follow the noisy measurement closely. In the second plot the filter should vary from the measurement quite a bit, and be much closer to a straight line than in the first graph. Why does **Q** affect the plots this way?

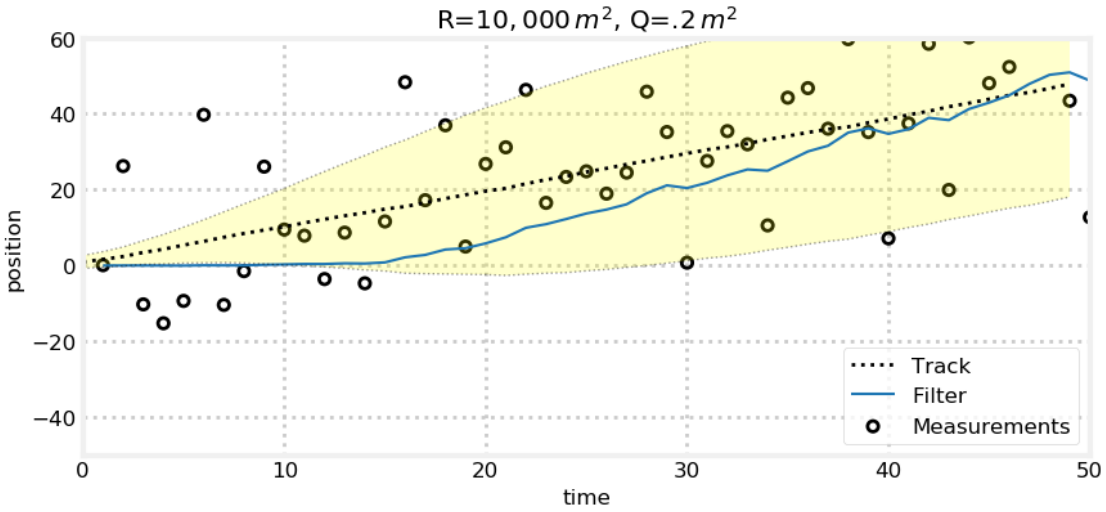
Let's remind ourselves of what the term *process uncertainty* means. Consider the problem of tracking a ball. We can accurately model its behavior in a vacuum with math, but with wind, varying air density, temperature, and a spinning ball with an imperfect surface our model will diverge from reality.

In the first case we set `Q_var=20 m^2`, which is quite large. In physical terms this is telling the filter "I don't trust my motion prediction step" as we are saying that the variance in the velocity is 20. Strictly speaking, we are telling the filter there is a lot of external noise that we are not modeling with **F**, but the upshot of that is to not trust the motion prediction step. The filter will be computing velocity ($\dot{x}$), but then mostly ignoring it because we are telling the filter that the computation is extremely suspect. Therefore the filter has nothing to trust but the measurements, and thus it follows the measurements closely.

In the second case we set `Q_var=0.02 m^2`, which is quite small. In physical terms we are telling the filter "trust the prediction, it is really good!". More strictly this actually says there is very small amounts of process noise (variance 0.02 $m^2$), so the process model is very accurate. So the filter ends up ignoring some of the measurement as it jumps up and down, because the variation in the measurement does not match our trustworthy velocity prediction.

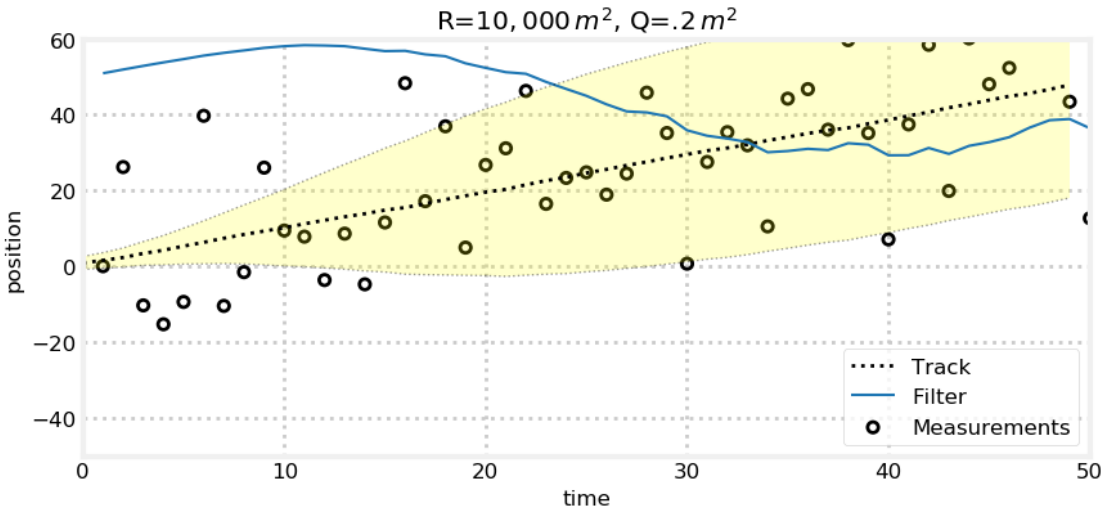
Now let's set `Q_var` to $0.2\,m^2$, and bump `R_var` up to $10,000\,m^2$. This is telling the filter that the measurement noise is very large.

```
In [46]: run(track=trk, zs=zs, R=10000, Q=.2, P=P, plot_P=False,
              title='R=$10,000\, m^2$, Q=$.2\, m^2$');
```



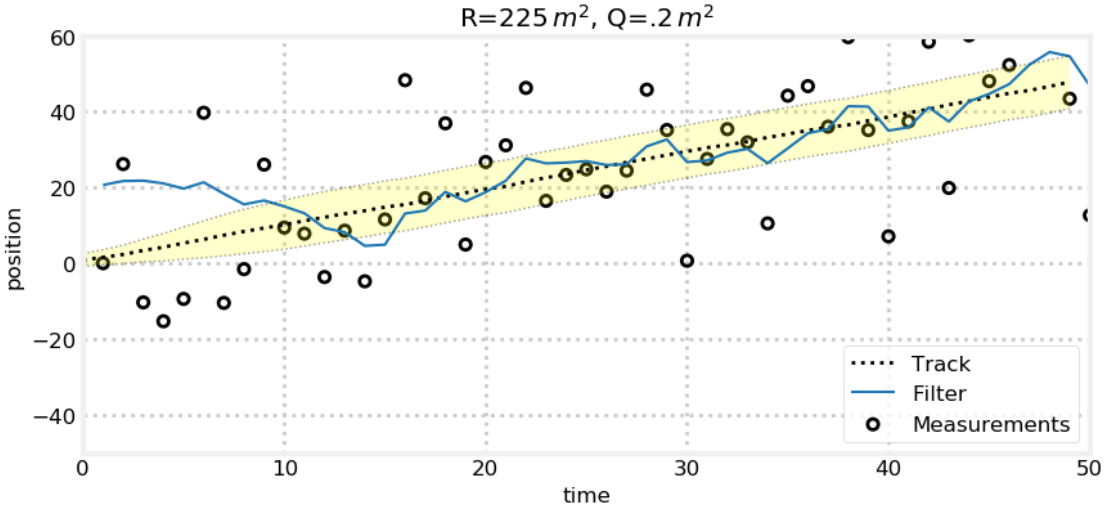

The effect of this can be subtle. We have created an suboptimal filter because the actual measurement noise variance is 225 $m^2$, not 10,000 $m^2$. By setting the filter's noise variance so high we force the filter to favor the prediction over the measurement. This can lead to apparently very smooth and good looking results. In the chart above the track may look extremely good to you since it follows the ideal path very closely. But, the 'great' behavior at the start should give you pause - the filter has not converged yet ($\mathbf{P}$ is still large) so it should not be able to be so close to the actual position. We can see that $\mathbf{P}$ has not converged because the entire chart is colored with the yellow background denoting the size of $\mathbf{P}$. Let's see the result of a bad initial guess for the position by guessing the initial position to be 50 m and the initial velocity to be 1 m/s.

```
In [47]: run(track=trk, zs=zs, R=10000, Q=.2, P=P, plot_P=False,
              x0=np.array([50., 1.]),
              title='R=$10,000\, m^2$, Q=$.2\, m^2$');
```




Here we can see that the filter cannot acquire the track. This happens because even though the filter is getting reasonably good measurements it assumes that the measurements are bad, and eventually predicts forward from a bad position at each step. If you think that perhaps that bad initial position would give similar results for a smaller measurement noise, let's set it back to the correct value of 225 $m^2$.

```
In [48]: run(track=trk, zs=zs, R=225, Q=.2, P=P, plot_P=False,
              x0=np.array([20., 1.]),
              title='R=$225\, m^2$, Q=$.2\, m^2$');
```



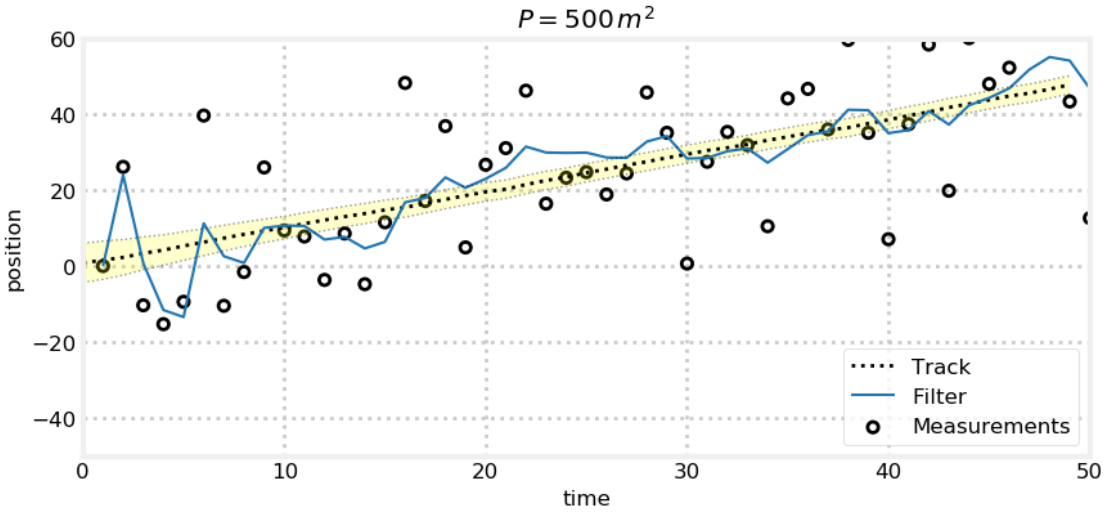

Here we see that the filter initially struggles for several iterations to acquire the track, but then it accurately tracks our dog. In fact, this is nearly optimum - we have not designed $\mathbf{Q}$ optimally, but $\mathbf{R}$ is optimal. A rule of thumb for $\mathbf{Q}$ is to set it between $\frac{1}{2}\Delta a$ to $\Delta a$, where $\Delta a$ is the maximum amount that the acceleration will change between sample periods. This only applies for the assumption we are making in this chapter - that acceleration is constant and uncorrelated between each time period. In the Kalman Math chapter we will discuss several different ways of designing $\mathbf{Q}$.

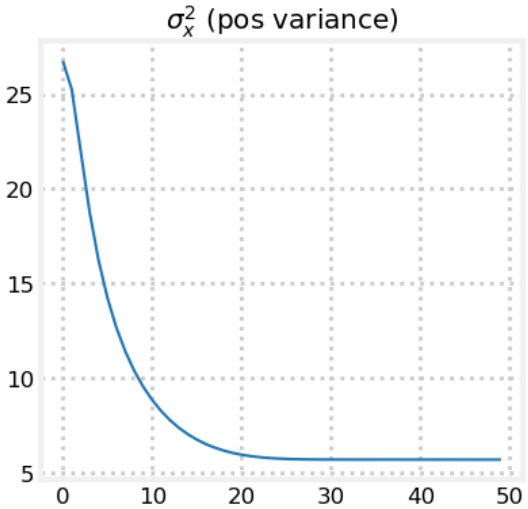
To some extent you can get similar looking output by varying either **R** or **Q**, but I urge you to not 'magically' alter these until you get output that you like. Always think about the physical implications of these assignments, and vary **R** and/or **Q** based on your knowledge of the system you are filtering. Back that up with extensive simulations and/or trial runs of real data.

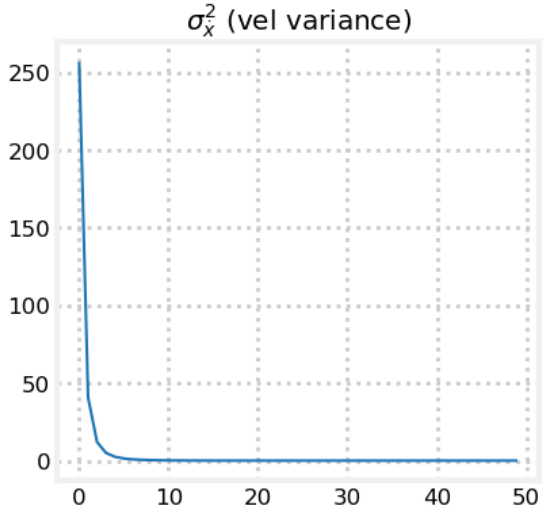
## 6.13 A Detailed Examination of the Covariance Matrix

Let's start by revisiting plotting a track. I've hard coded the data and noise in `zs_var_275` to avoid being at the mercy of the random number generator, which might generate data that does not illustrate what I want to talk about. I will start with `P=500`.

In [49]:
```
import kf_book.mkf_internal as mkf_internal

var = 27.5
data = mkf_internal.zs_var_275()
run(track=trk, zs=zs, R=var, Q=.02, P=500., plot_P=True,
    title='$P=500\, m^2$');
```



Looking at the output we see a very large spike in the filter output at the beginning. We set $\mathrm{P} = 500\,\mathbf{I}_2$ (this is shorthand notation for a 2x2 diagonal matrix with 500 in the diagonal). We now have enough information to understand what this means, and how the Kalman filter treats it. The 500 in the upper left hand corner corresponds to $\sigma_x^2$; therefore we are saying the standard deviation of $\mathbf{x}$ is $\sqrt{500}$, or roughly 22.36 m. Roughly 99% of the samples occur withing $3\sigma$, therefore $\sigma_x^2 = 500$ is telling the Kalman filter that the prediction (the prior) could be up to 67 meters off. That is a large error, so when the measurement spikes the Kalman filter distrusts its own estimate and jumps wildly to try to incorporate the measurement. Then, as the filter evolves $\mathbf{P}$ quickly converges to a more realistic value.

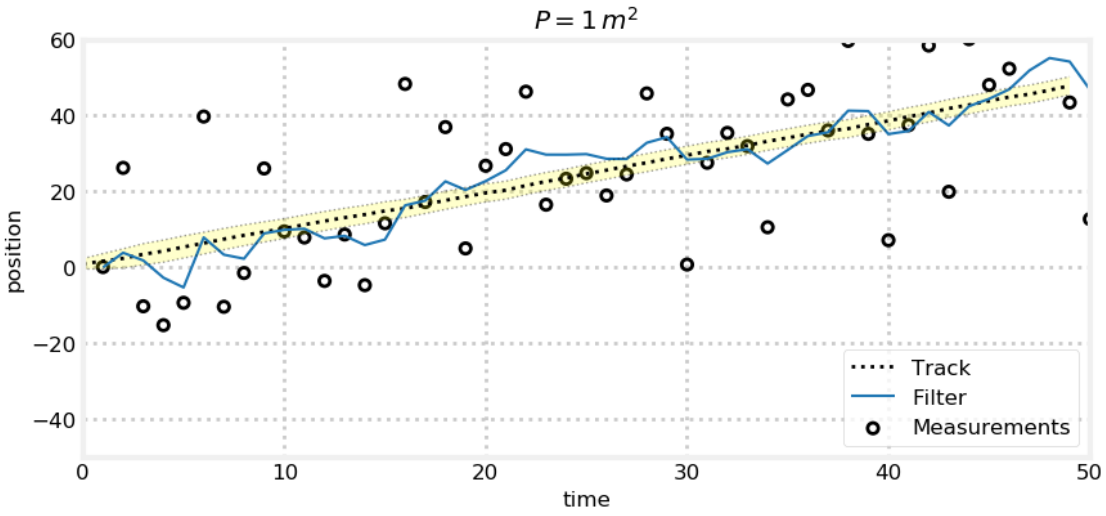
Let's look at the math behind this. The equation for the Kalman gain is

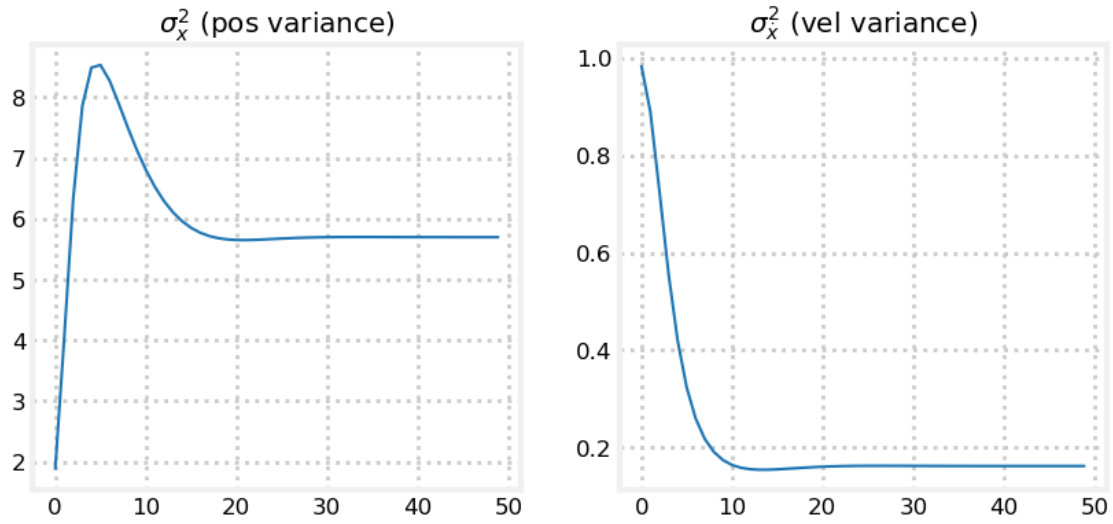
$$\mathbf{K} = \mathbf{P}\mathbf{H}^\mathsf{T}\mathbf{S}^{-1} \approx \frac{\mathbf{P}\mathbf{H}^\mathsf{T}}{\mathbf{S}} \approx \frac{\text{uncertainty}_{\text{prediction}}}{\text{uncertainty}_{\text{measurement}}}\mathbf{H}^\mathsf{T}$$

It is a ratio of the uncertainty of the prediction vs measurement. Here the uncertainty in the prediction is large, so $\mathbf{K}$ is large (near 1 if this was a scalar). $\mathbf{K}$ is multiplied by the residual $\mathbf{y} = \mathbf{z} - \mathbf{H}\mathbf{x}$, which is the measurement minus the prediction, so a large $\mathbf{K}$ favors the measurement. Therefore if $\mathbf{P}$ is large relative to the sensor uncertainty $\mathbf{R}$ the filter will form most of the estimate from the measurement.

Now let us see the effect of a smaller initial value of $\mathbf{P} = 1.0\,\mathbf{I}_2$.

```
In [50]: run(track=trk, zs=zs, R=var, Q=.02, P=1., plot_P=True,
              title='$P=1\, m^2$');
```



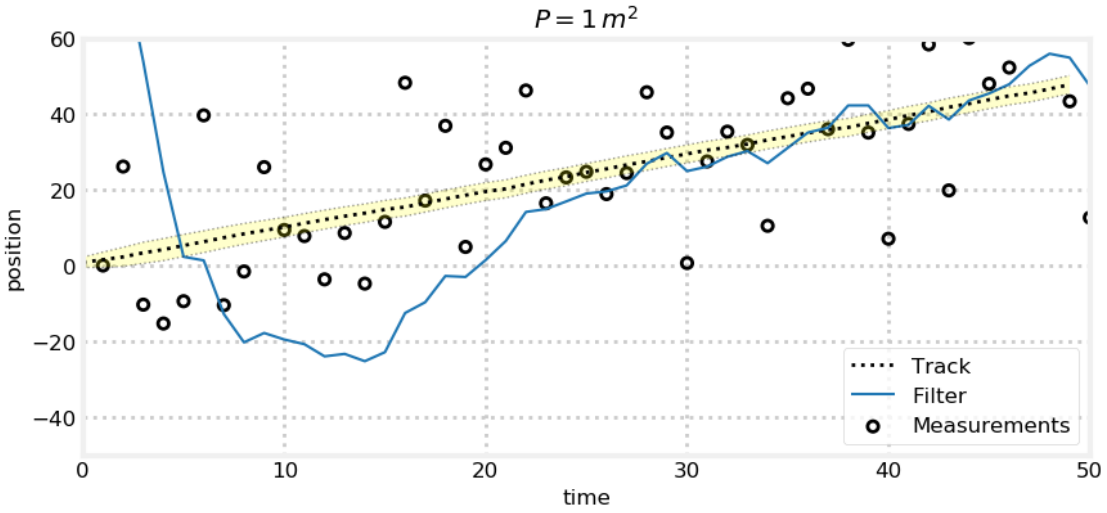



This *looks* good at first blush. The plot does not have the spike that the former plot did; the filter starts tracking the measurements and doesn't take any time to settle to the signal. However, if we look at the plots for P you can see that there is an initial spike for the variance in position, and that it never really converges. Poor design leads to a long convergence time, and suboptimal results.

So despite the filter tracking very close to the actual signal we cannot conclude that the 'magic' is to use a small **P**. Yes, this will avoid having the Kalman filter take time to accurately track the signal, but if we are truly uncertain about the initial measurements this can cause the filter to generate very bad results. If we are tracking a living object we are probably very uncertain about where it is before we start tracking it. On the other hand, if we are filtering the output of a thermometer, we are as certain about the first measurement as the 1000th. For your Kalman filter to perform well you must set **P** to a value that truly reflects your knowledge about the data.

Let's see the result of a bad initial estimate coupled with a very small **P**. We will set our initial estimate at x = 100 m (whereas the dog actually starts at 0m), but set `P=1` $m^2$. This is clearly an incorrect value for **P** as the estimate is off by 100 m but we tell the filter that it the $3\sigma$ error is 3 m.

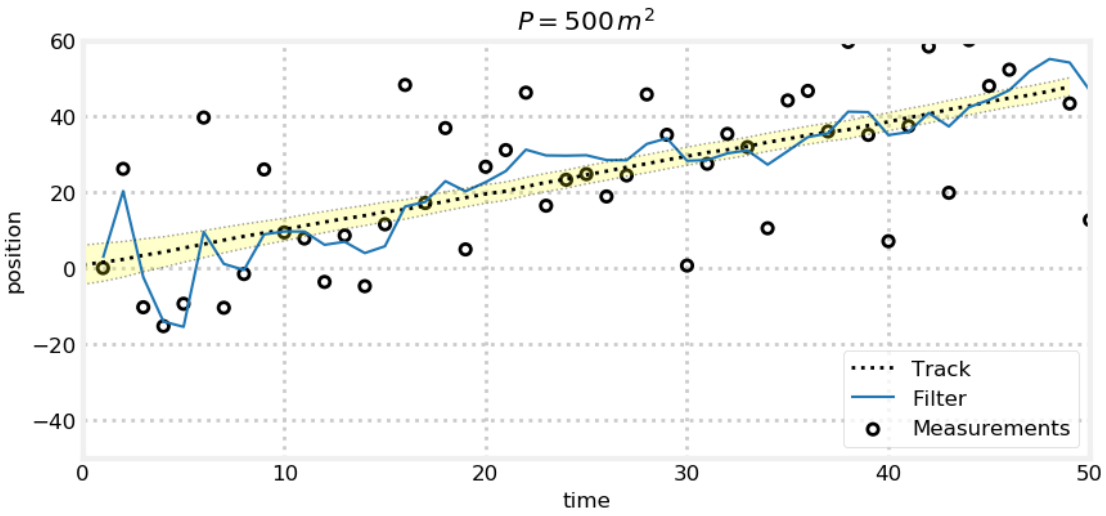
```
In [51]: x = np.array([100., 0.])
         run(track=trk, zs=zs, R=var, Q=.02, P=1., x0=x,
             plot_P=False, title='$P=1\, m^2$');
```



We can see that the initial estimates are terrible and that it takes the filter a long time to start converging onto the signal . This is because we told the Kalman filter that we strongly believe in our initial estimate of 100 m and were incorrect in that belief.

Now, let's provide a more reasonable value for P and see the difference.

```
In [52]: x = np.array([100., 0.])
         run(track=trk, zs=zs, R=var, Q=.02, P=500., x0=x,
            plot_P=False, title='$P=500\, m^2$');
```

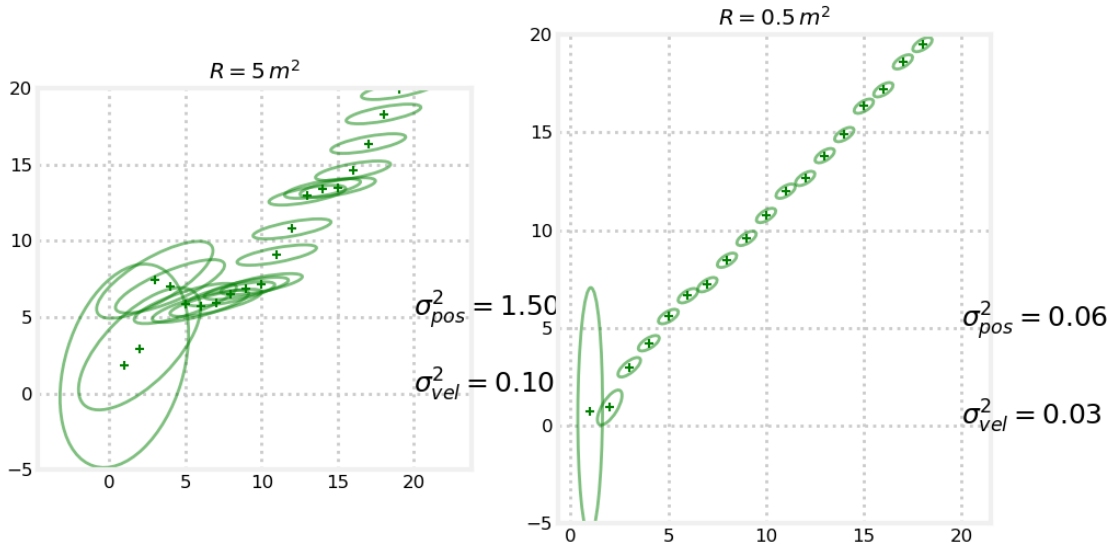



In this case the Kalman filter is very uncertain about the initial state, so it converges onto the signal much faster. It is producing good output after only 5 to 6 epochs. With the theory we have developed so far this is about as good as we can do. However, this scenario is a bit artificial; if we do not know where the object is when we start tracking we do not initialize the filter to some arbitrary value, such as 0 m or 100 m. I address this in the **Filter Initialization** section below.

Let's do another Kalman filter for our dog, and this time plot the covariance ellipses on the same plot as the position.

```
In [53]: from kf_book.mkf_internal import plot_track_ellipses

         def plot_covariances(count, R, Q=0, P=20., title=''):
             track, zs = compute_dog_data(R, Q, count)
             f = pos_vel_filter(x=(0., 0.), R=R, Q=Q, P=P)

             xs, cov = [], []
             for z in zs:
                 f.predict()
                 f.update(z)

                 xs.append(f.x[0])
                 cov.append(f.P)
             plot_track_ellipses(count, zs, xs, cov, title)

         plt.figure(figsize=(10,6))
```

```
plt.subplot(121)
plot_covariances(R=5, Q=.02, count=20, title='$R = 5\, m^2$')
plt.subplot(122)
plot_covariances(R=.1, Q=.02, count=20, title='$R = 0.5\, m^2$')
```




If you are viewing this in Jupyter Notebook or on the web, here is an animation of the filter filtering the data. I've tuned the filter parameters such that it is easy to see a change in $\mathbf{P}$ as the filter progresses.

The output on these is a bit messy, but you should be able to see what is happening. In both plots we are drawing the covariance matrix for each point. We start with the covariance $\mathbf{P} = \begin{pmatrix} 20 & 0 \\ 0 & 20 \end{pmatrix}$, which signifies a lot of uncertainty about our initial belief. After we receive the first measurement the Kalman filter updates this belief, and so the variance is no longer as large. In the top plot the first ellipse (the one on the far left) should be a slightly squashed ellipse. As the filter continues processing the measurements the covariance ellipse quickly shifts shape until it settles down to being a long, narrow ellipse tilted in the direction of movement.

Think about what this means physically. The x-axis of the ellipse denotes our uncertainty in position, and the y-axis our uncertainty in velocity. So, an ellipse that is taller than it is wide signifies that we are more uncertain about the velocity than the position. Conversely, a wide, narrow ellipse shows high uncertainty in position and low uncertainty in velocity. Finally, the amount of tilt shows the amount of correlation between the two variables.

The first plot, with $R = 5m^2$, finishes up with an ellipse that is wider than it is tall. If that is not clear I have printed out the variances for the last ellipse in the lower right hand corner.

In contrast, the second plot, with `R=0.5` $m^2$, has a final ellipse that is taller than wide. The ellipses in the second plot are all much smaller than the ellipses in the first plot. This stands to reason because a small $\mathbf{R}$ implies a small amount of noise in our measurements. Small noise means accurate predictions, and thus a strong belief in our position.

## 6.14 Question: Explain Ellipse Differences

Why are the ellipses for $\mathbf{R} = 5m^2$ more tilted towards the horizontal than the ellipses for $\mathbf{R} = 0.5m^2$. Hint: think about this in the context of what these ellipses mean physically, not in terms of the math. If you aren't sure about the answer,change $\mathbf{R}$ to truly large and small numbers such as 100 $m^2$ and 0.1 $m^2$, observe the changes, and think about what this means.

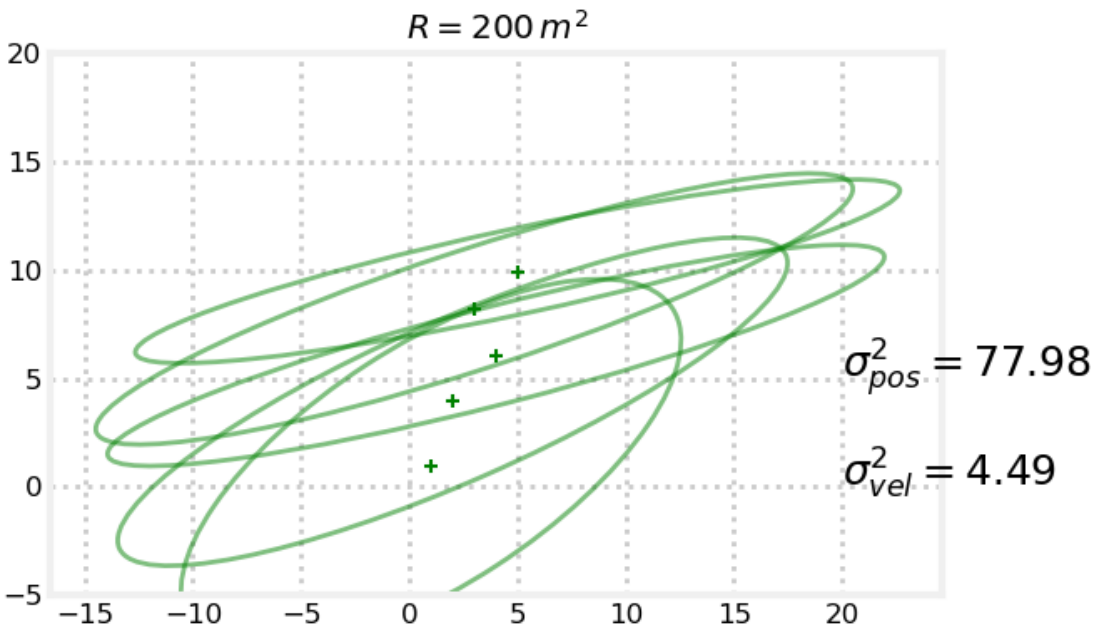
### 6.14.1 Solution

The x-axis is for position, and y-axis is velocity. An ellipse that is vertical, or nearly so, says there is no correlation between position and velocity, and an ellipse that is diagonal says that there is a lot of correlation. Phrased that way, the results sound unlikely. The tilt of the ellipse changes, but the correlation shouldn't be changing over time. But this is a measure of the *output of the filter*, not a description of the actual, physical world. When $\mathbf{R}$ is very large we are telling the filter that there is a lot of noise in the measurements. In that case the Kalman gain $\mathbf{K}$ is set to favor the prediction over the measurement, and the prediction comes from the velocity state variable. Thus there is a large correlation between $x$ and $\dot{x}$. Conversely, if $\mathbf{R}$ is small, we are telling the filter that the measurement is very trustworthy, and $\mathbf{K}$ is set to favor the measurement over the prediction. Why would the filter want to use the prediction if the measurement is nearly perfect? If the filter is not using much from the prediction there will be very little correlation reported.

**This is a critical point to understand!**. The Kalman filter is a mathematical model for a real world system. A report of little correlation *does not mean* there is no correlation in the physical system, just that there was no *linear* correlation in the mathematical model. It's a report of how much measurement vs prediction was incorporated into the model.

Let's bring that point home with a truly large measurement error. We will set $\mathbf{R} = 200\,m^2$. Think about what the plot will look like before looking at it.

```
In [54]: plot_covariances(R=200., Q=.2, count=5, title='$R = 200\, m^2$')
```




I hope the result was what you were expecting. The ellipse quickly became very wide and not very tall. It did this because the Kalman filter mostly used the prediction vs the measurement to produce the filtered result. We can also see how the filter output is slow to acquire the track. The Kalman filter assumes that the measurements are extremely noisy, and so it is very slow to update its estimate for $\dot{x}$.

Keep looking at these plots until you grasp how to interpret the covariance matrix $\mathbf{P}$. When you work with a 9×9 matrix it may seem overwhelming - there are 81 numbers to interpret. Just break it down - the diagonal contains the variance for each state variable, and all off diagonal elements are the product of two variances and a scaling factor $p$. You cannot plot a 9×9 matrix on the screen so you have to develop your intuition and understanding in this simple, 2-D case.

> When plotting covariance ellipses, make sure to always use ax.set_aspect('equal') or plt.axis('equal') in your code (the former lets you set the xlim and ylim values). If the axis

use different scales the ellipses will be drawn distorted. For example, the ellipse may be drawn as being taller than it is wide, but it may actually be wider than tall.

## 6.15 Filter Initialization

There are many schemes for initializing the filter. The following approach performs well in most situations. In this scheme you do not initialize the filter until you get the first measurement, $\mathbf{z}_0$. From this you can compute the initial value for $\mathbf{x}$ with $\mathbf{x}_0 = \mathbf{z}_0$. If $\mathbf{z}$ is not of the same size, type, and units as $\mathbf{x}$, which is usually the case, we can use our measurement function as follow.

We know

$$\mathbf{z} = \mathbf{H}\mathbf{x}$$

Hence,

$$\begin{aligned}\mathbf{H}^{-1}\mathbf{H}\mathbf{x} &= \mathbf{H}^{-1}\mathbf{z} \\ \mathbf{x} &= \mathbf{H}^{-1}\mathbf{z}\end{aligned}$$

Matrix inversion requires a square matrix, but $\mathbf{H}$ is rarely square. SciPy will compute the Moore-Penrose pseudo-inverse of a matrix with `scipy.linalg.pinv`, so your code might look like

```
In [55]: from scipy.linalg import pinv

         H = np.array([[1, 0.]])
         z0 = 3.2
         x = np.dot(pinv(H), z0)
         print(x)
```

```
[[3.2]
 [0. ]]
```

Specialized knowledge of your problem domain may lead you to a different computation, but this is one way to do it. For example, if the state includes velocity you might take the first two measurements of position, compute the difference, and use that as the initial velocity.

Now we need to compute a value for $\mathbf{P}$. This will vary by problem, but in general you will use the measurement error $\mathbf{R}$ for identical terms, and maximum values for the rest of the terms. Maybe that isn't clear. In this chapter we have been tracking and object using position and velocity as the state, and the measurements have been positions. In that case we would initialize $\mathbf{P}$ with

$$\mathbf{P} = \begin{bmatrix}\mathbf{R}_0 & 0 \\ 0 & vel^2_{max}\end{bmatrix}$$

The diagonal of $\mathbf{P}$ contains the variance of each state variable, so we populate it with reasonable values. $\mathbf{R}$ is a reasonable variance for the position, and the maximum velocity squared is a reasonable variance for the velocity. It is squared because variance is squared: $\sigma^2$.

You really need to understand the domain in which you are working and initialize your filter on the best available information. For example, suppose we were trying to track horses in a horse race. The initial measurements might be very bad, and provide you with a position far from the starting gate. We know that the horse must start at the starting gate; initializing the filter to the initial measurement would lead to suboptimal results. In this scenario we would want to always initialize the Kalman filter with the starting gate position of the horse.

## 6.16 Batch Processing

The Kalman filter is designed as a recursive algorithm - as new measurements come in we immediately create a new estimate. But it is very common to have a set of data that have been already collected which we want to filter. Kalman filters can be run in a batch mode, where all of the measurements are filtered at once. We have implemented this in `KalmanFilter.batch_filter()`. Internally, all the function does is loop over the measurements and collect the resulting state and covariance estimates in arrays. It simplifies your logic and conveniently gathers all of the outputs into arrays. I often use this function, but waited until the end of the chapter so you would become very familiar with the predict/update cyle that you must run.

First collect your measurements into an array or list. Maybe it is in a CSV file:

```
zs = read_altitude_from_csv('altitude_data.csv')
```

Or maybe you will generate it using a generator:

```
zs = [some_func(i) for i in range(1000)]
```

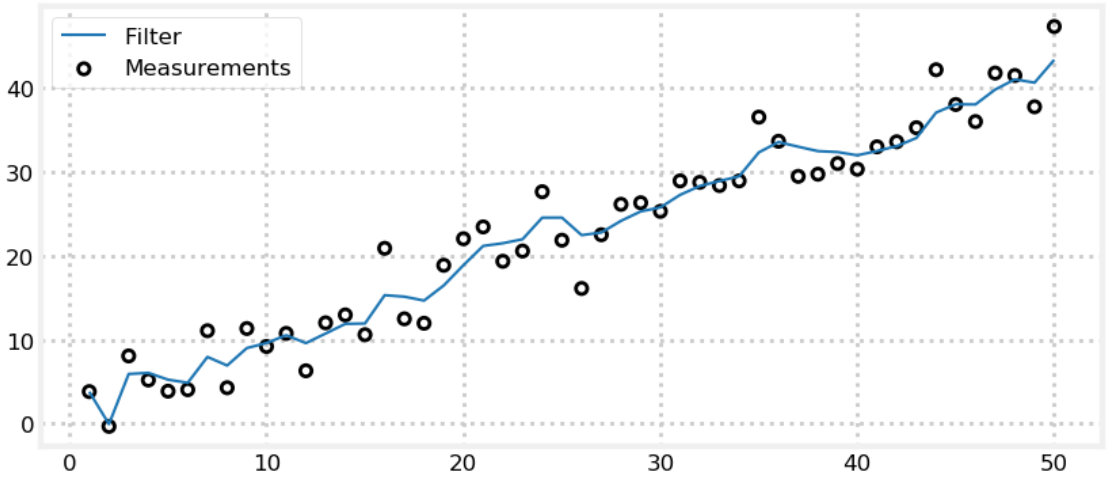
Then call the `batch_filter()` method.

```
Xs, Ps, Xs_prior, Ps_prior = kfilter.batch_filter(zs)
```

The function takes the list of measurements, filters it, and returns an NumPy array of state estimates (Xs), covariance matrices (Ps), and the priors for the same (Xs_prior, Ps_prior).

Here is a complete example.

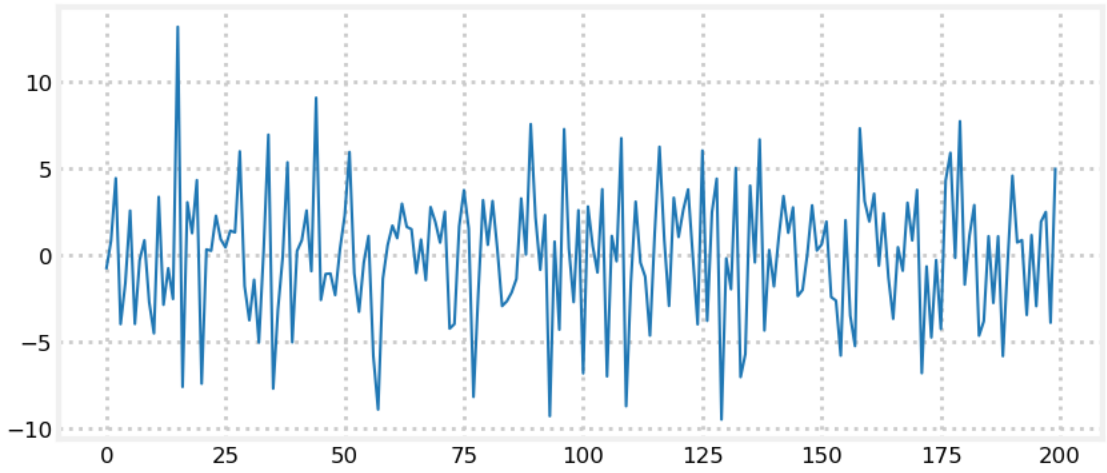
```
In [56]: count = 50
         track, zs = compute_dog_data(10, .2, count)
         P = np.diag([500., 49.])
         f = pos_vel_filter(x=(0., 0.), R=3., Q=.02, P=P)
         xs, _, _, _ = f.batch_filter(zs)

         book_plots.plot_measurements(range(1, count + 1), zs)
         book_plots.plot_filter(range(1, count + 1), xs[:, 0])
         plt.legend(loc='best');
```




The batch filter takes an optional `filterpy.common.Saver` object. If provided, all of the filter's attributes will be saved as well. This is useful if you want to inspect values other than the state and covariance. Here

I plot the residual to see if it appears like noise centered around 0. This is a quick visual inspection to see if the filter is well designed. If if drifts from zero, or doesn't look like noise, the filter is poorly designed and/or the processes are not Gaussian. We will discuss this in detail in later chapters. For now consider this a demonstration of the `Saver` class.

```
In [57]: track, zs = compute_dog_data(10, .2, 200)
         P = np.diag([500., 49.])
         f = pos_vel_filter(x=(0., 0.), R=3., Q=.02, P=P)
         s = Saver(f)
         xs, _, _, _ = f.batch_filter(zs, saver=s)
         s.to_array()
         plt.plot(s.y);
```




## 6.17 Smoothing the Results

This book includes a chapter on smoothing the results; I will not repeat the information here. However, it is so easy to use, and offers such a profoundly improved output that I will tease you with a few examples. The smoothing chapter is not especially difficult; you are sufficiently prepared to read it now.

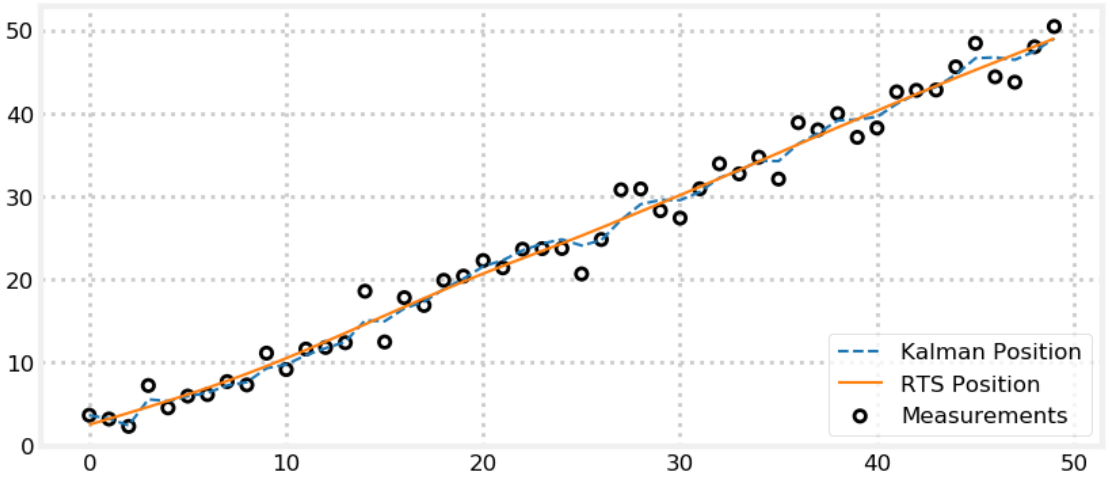
Let's assume that we are tracking a car that has been traveling in a straight line. We get a measurement that implies that the car is starting to turn to the left. The Kalman filter moves the state estimate somewhat towards the measurement, but it cannot judge whether this is a particularly noisy measurement or the true start of a turn.

However, if we have future measurements we can decide if a turn was made. Suppose the subsequent measurements all continue turning left. We can then be sure that that a turn was initiated. On the other hand, if the subsequent measurements continued on in a straight line we would know that the measurement was noisy and should be mostly ignored. Instead of making an estimate part way between the measurement and prediction the estimate will either fully incorporate the measurement or ignore it, depending on what the future measurements imply about the object's movement.

`KalmanFilter` implements a form of this algorithm which is called an *RTS smoother*, named after the inventors of the algorithm: Rauch, Tung, and Striebel. The method is `rts_smoother()`. To use it pass in the means and covariances computed from the `batch_filter` step, and receive back the smoothed means, covariances, and Kalman gain.

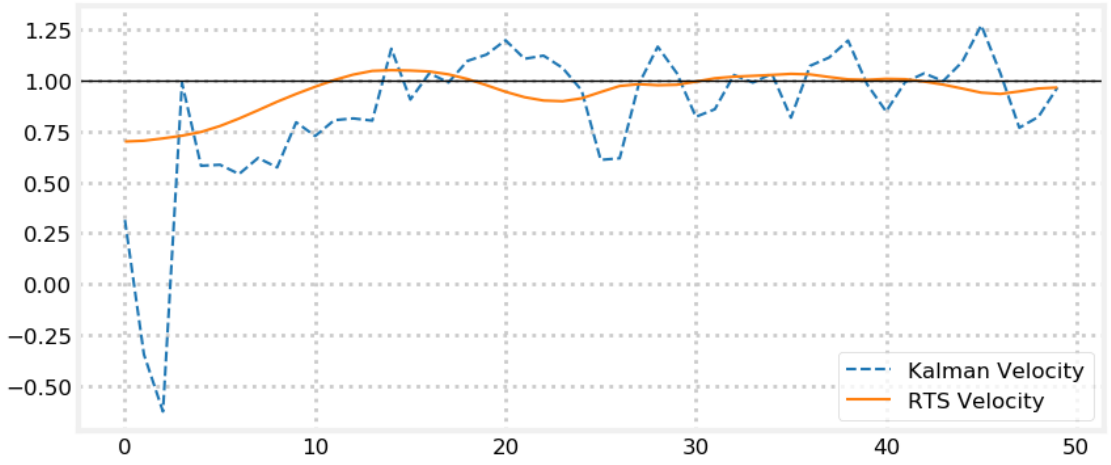
```
In [58]: from numpy.random import seed
         count = 50
```

```
seed(8923)

P = np.diag([500., 49.])
f = pos_vel_filter(x=(0., 0.), R=3., Q=.02, P=P)
track, zs = compute_dog_data(3., .02, count)
Xs, Covs, _, _ = f.batch_filter(zs)
Ms, Ps, _, _ = f.rts_smoother(Xs, Covs)

book_plots.plot_measurements(zs)
plt.plot(Xs[:, 0], ls='--', label='Kalman Position')
plt.plot(Ms[:, 0], label='RTS Position')
plt.legend(loc=4);
```




This output is fantastic! Two things are very apparent to me in this chart. First, the RTS smoother's output is much smoother than the KF output. Second, it is almost always more accurate than the KF output (we will examine this claim in detail in the **Smoothing** chapter). The improvement in the velocity, which is a hidden variable, is even more dramatic:

```
In [59]: plt.plot(Xs[:, 1], ls='--', label='Kalman Velocity')
         plt.plot(Ms[:, 1], label='RTS Velocity')
         plt.legend(loc=4)
         plt.gca().axhline(1, lw=1, c='k');
```




We will explore why this is so in the next exercise.

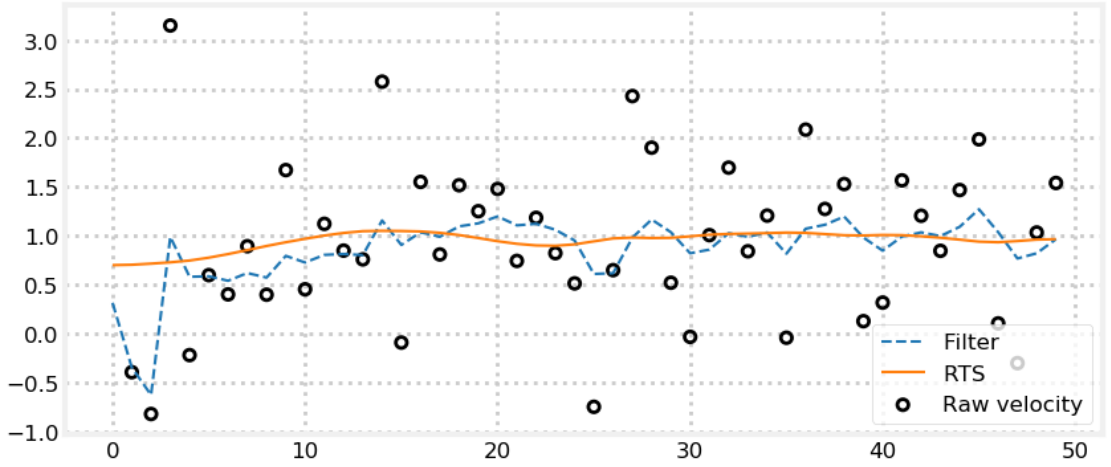
## 6.18 Exercise: Compare Velocities

Since we are plotting velocities let's look at what the 'raw' velocity is, which we can compute by subtracting subsequent measurements. i.e the velocity at time 1 can be approximated by `xs[1] - xs[0]`. Plot the raw value against the values estimated by the Kalman filter and by the RTS filter. Discuss what you see.

```
In [60]: # your code here
```

### 6.18.1 Solution

```
In [61]: dx = np.diff(Xs[:, 0], axis=0)
         plt.scatter(range(1, len(dx) + 1), dx, facecolor='none',
                     edgecolor='k', lw=2, label='Raw velocity')
         plt.plot(Xs[:, 1], ls='--', label='Filter')
         plt.plot(Ms[:, 1], label='RTS')
         plt.legend(loc=4);
```



We see that the noise swamps the signal, causing the raw values to be essentially worthless. The filter is maintaining a separate estimate for velocity. The Kalman gain $\mathbf{K}$ is multidimensional. For example, it might have the value $\mathbf{K} = [0.1274, 0.843]^\mathsf{T}$. the first value is used to scale the residual of the position, and the second value will scale the residual of the velocity. The covariance matrix tells the filter how correlated the position and velocity are, and each will be optimally filtered.

I show this to reiterate the importance of using Kalman filters to compute velocities, accelerations, and even higher order values. I use a Kalman filter even when my measurements are so accurate that I am willing to use them unfiltered because it allows me accurate estimates for velocities and accelerations.

## 6.19 Discussion and Summary

Multivariate Gaussians allow us to simultaneously handle multiple dimensions, both spacial and others (velocity, etc). We made a key insight: hidden variables have the ability to significantly increase the accuracy of the filter. This is possible because the hidden variables are correlated with the observed variables.

I gave an intuitive definition of *observability*. Observability was invented by Dr. Kalman for linear systems, and there is a fair amount of theory behind it. It answers the question of whether a system state can be determined by observing the system's output. For our problems this has been easy to determine, but more complicated systems may require rigorous analysis. Wikipedia's Observability article has an overview; if you need to learn the topic Section D.4 is a good source.

There is one important caveat about hidden variables. It is easy to construct a filter that produces estimates for hidden variables. I could write a filter that estimates the color of a tracked car. But there is no way to compute car color from positions, so the estimate for the color will be nonsense. The designer must verify that these variables are being estimated correctly. If you do not have a velocity sensor and yet are estimating velocity, you will need to test that the velocity estimates are correct.; do not trust that they are. For example, suppose the velocity has a periodic component to it - it looks like a sine wave. If your sample time is less than 2 times the frequency you will not be able to accurately estimate the velocity (due to Nyquist's Theorem). Imagine that the sample period is equal to the frequency of the velocity. The filter will report that the velocity is constant because it samples the system at the same point on the sin wave.

Initialization poses a particularly difficult problem for hidden variables. If you start with a bad initialization the filter can usually recover the observed variables, but may struggle and fail with the hidden one. Estimating hidden variables is a powerful tool, but a dangerous one.

I established a series of steps for designing a Kalman filter. These are not a usual part of the Kalman filter literature, and are only meant as a guide, not a prescription. Designing for a hard problem is an iterative process. You make a guess at the state vector, work out what your measurement and state models are, run some tests, and then alter the design as necessary.

The design of $\mathbf{R}$ and $\mathbf{Q}$ is often quite challenging. I've made it appear to be quite scientific. Your sensor has Gaussian noise of $\mathcal{N}(0, \sigma^2)$, so set $\mathbf{R} = \sigma^2$. Easy! This is a dirty lie. Sensors are not Gaussian. We started the book with a bathroom scale. Suppose $\sigma = 1$ kg, and you try to weigh something that weighs 0.5 kg. Theory tells us we will get negative measurements, but of course the scale will never report weights less than zero. Real world sensors typically have *fat tails* (known as *kurtosis*) and *skew*. In some cases, such as with the scale, one or both tails are truncated.

The case with $\mathbf{Q}$ is more dire. I hope you were skeptical when I blithely assigned a noise matrix to my prediction about the movements of a dog. Who can say what a dog will do next? The Kalman filter in my GPS doesn't know about hills, the outside winds, or my terrible driving skills. Yet the filter requires a precise number to encapsulate all of that information, and it needs to work while I drive off-road in the desert, and when a Formula One champion drives on a track.

These problems led some researchers and engineers to derogatorily call the Kalman filter a 'ball of mud'. In other words, it doesn't always hold together so well. Another term to know - Kalman filters can become *smug*. Their estimates are based solely on what you tell it the noises are. Those values can lead to overly confident estimates. $\mathbf{P}$ gets smaller and smaller while the filter is actually becoming more and more inaccurate! In the worst case the filter diverges. We will see a lot of that when we start studying nonlinear filters.

The Kalman filter is a mathematical model of the world. The output is only as accurate as that model.

To make the math tractable we had to make some assumptions. We assume that the sensors and motion model have Gaussian noise. We assume that everything is linear. If that is true, the Kalman filter is *optimal* in a least squares sense. This means that there is no way to make a better estimate than what the filter gives us. However, these assumption are almost never true, and hence the model is necessarily limited, and a working filter is rarely optimal.

In later chapters we will deal with the problem of nonlinearity. For now I want you to understand that designing the matrices of a linear filter is an experimental procedure more than a mathematical one. Use math to establish the initial values, but then you need to experiment. If there is a lot of unaccounted noise in the world (wind, etc.) you may have to make **Q** larger. If you make it too large the filter fails to respond quickly to changes. In the **Adaptive Filters** chapter you will learn some alternative techniques that allow you to change the filter design in real time in response to the inputs and performance, but for now you need to find one set of values that works for the conditions your filter will encounter. Noise matrices for an acrobatic plane might be different if the pilot is a student than if the pilot is an expert as the dynamics will be quite different.

## 6.20 References


- [1] 'Kalman Filters'. Wikipedia https://en.wikipedia.org/wiki/Kalman_filter#Details
- **[Grewal2008]** Grewal, Mohinder S., Andrews, Angus P. *Kalman Filtering: Theory and Practice Using MATLAB.* Third Edition. John Wiley & Sons. 2008.

# Chapter 7

# Kalman Filter Math

```
In [1]: from __future__ import division, print_function
        %matplotlib inline
```

```
In [2]: #format the book
        import book_format
        book_format.set_style()
```

```
Out[2]:
```

If you've gotten this far I hope that you are thinking that the Kalman filter's fearsome reputation is somewhat undeserved. Sure, I hand waved some equations away, but I hope implementation has been fairly straightforward for you. The underlying concept is quite straightforward - take two measurements, or a measurement and a prediction, and choose the output to be somewhere between the two. If you believe the measurement more your guess will be closer to the measurement, and if you believe the prediction is more accurate your guess will lie closer to it. That's not rocket science (little joke - it is exactly this math that got Apollo to the moon and back!).

To be honest I have been choosing my problems carefully. For an arbitrary problem designing the Kalman filter matrices can be extremely difficult. I haven't been *too tricky*, though. Equations like Newton's equations of motion can be trivially computed for Kalman filter applications, and they make up the bulk of the kind of problems that we want to solve.

I have illustrated the concepts with code and reasoning, not math. But there are topics that do require more mathematics than I have used so far. This chapter presents the math that you will need for the rest of the book.

## 7.1 Modeling a Dynamic System

A *dynamic system* is a physical system whose state (position, temperature, etc) evolves over time. Calculus is the math of changing values, so we use differential equations to model dynamic systems. Some systems cannot be modeled with differential equations, but we will not encounter those in this book.

Modeling dynamic systems is properly the topic of several college courses. To an extent there is no substitute for a few semesters of ordinary and partial differential equations followed by a graduate course in control system theory. If you are a hobbyist, or trying to solve one very specific filtering problem at work you probably do not have the time and/or inclination to devote a year or more to that education.

Fortunately, I can present enough of the theory to allow us to create the system equations for many different Kalman filters. My goal is to get you to the stage where you can read a publication and understand it well enough to implement the algorithms. The background math is deep, but in practice we end up using a few simple techniques.

This is the longest section of pure math in this book. You will need to master everything in this section to understand the Extended Kalman filter (EKF), the most common nonlinear filter. I do cover more modern

filters that do not require as much of this math. You can choose to skim now, and come back to this if you decide to learn the EKF.

We need to start by understanding the underlying equations and assumptions that the Kalman filter uses. We are trying to model real world phenomena, so what do we have to consider?

Each physical system has a process. For example, a car traveling at a certain velocity goes so far in a fixed amount of time, and its velocity varies as a function of its acceleration. We describe that behavior with the well known Newtonian equations that we learned in high school.

$$\begin{aligned} v &= at \\ x &= \frac{1}{2}at^2 + v_0t + x_0 \end{aligned}$$

Once we learned calculus we saw them in this form:

$$\mathbf{v} = \frac{d\mathbf{x}}{dt}, \quad \mathbf{a} = \frac{d\mathbf{v}}{dt} = \frac{d^2\mathbf{x}}{dt^2}$$

A typical automobile tracking problem would have you compute the distance traveled given a constant velocity or acceleration, as we did in previous chapters. But, of course we know this is not all that is happening. No car travels on a perfect road. There are bumps, wind drag, and hills that raise and lower the speed. The suspension is a mechanical system with friction and imperfect springs.

Perfectly modeling a system is impossible except for the most trivial problems. We are forced to make a simplification. At any time $t$ we say that the true state (such as the position of our car) is the predicted value from the imperfect model plus some unknown *process noise*:

$$x(t) = x_{pred}(t) + noise(t)$$

This is not meant to imply that $noise(t)$ is a function that we can derive analytically. It is merely a statement of fact - we can always describe the true value as the predicted value plus the process noise. "Noise" does not imply random events. If we are tracking a thrown ball in the atmosphere, and our model assumes the ball is in a vacuum, then the effect of air drag is process noise in this context.

In the next section we will learn techniques to convert a set of higher order differential equations into a set of first-order differential equations. After the conversion the model of the system without noise is:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$$

$\mathbf{A}$ is known as the *systems dynamics matrix* as it describes the dynamics of the system. Now we need to model the noise. We will call that $\mathbf{w}$, and add it to the equation.

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{w}$$

$\mathbf{w}$ may strike you as a poor choice for the name, but you will soon see that the Kalman filter assumes *white* noise.

Finally, we need to consider any inputs into the system. We assume an input $\mathbf{u}$, and that there exists a linear model that defines how that input changes the system. For example, pressing the accelerator in your car makes it accelerate, and gravity causes balls to fall. Both are control inputs. We will need a matrix $\mathbf{B}$ to convert $u$ into the effect on the system. We add that into our equation:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} + \mathbf{w}$$

And that's it. That is one of the equations that Dr. Kalman set out to solve, and he found an optimal estimator if we assume certain properties of $\mathbf{w}$.

## 7.2 State-Space Representation of Dynamic Systems

We've derived the equation

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u} + \mathbf{w}$$

However, we are not interested in the derivative of $\mathbf{x}$, but in $\mathbf{x}$ itself. Ignoring the noise for a moment, we want an equation that recusively finds the value of $\mathbf{x}$ at time $t_k$ in terms of $\mathbf{x}$ at time $t_{k-1}$:

$$\mathbf{x}(t_k) = \mathbf{F}(\Delta t)\mathbf{x}(t_{k-1}) + \mathbf{B}(t_k)\mathbf{u}(t_k)$$

Convention allows us to write $\mathbf{x}(t_k)$ as $\mathbf{x}_k$, which means the the value of $\mathbf{x}$ at the k$^{th}$ value of $t$.

$$\mathbf{x}_k = \mathbf{F}\mathbf{x}_{k-1} + \mathbf{B}_k\mathbf{u}_k$$

$\mathbf{F}$ is the familiar *state transition matrix*, named due to its ability to transition the state's value between discrete time steps. It is very similar to the system dynamics matrix $\mathbf{A}$. The difference is that $\mathbf{A}$ models a set of linear differential equations, and is continuous. $\mathbf{F}$ is discrete, and represents a set of linear equations (not differential equations) which transitions $\mathbf{x}_{k-1}$ to $\mathbf{x}_k$ over a discrete time step $\Delta t$.

Finding this matrix is often quite difficult. The equation $\dot{x} = v$ is the simplest possible differential equation and we trivially integrate it as:

$$\begin{aligned}
\int_{x_{k-1}}^{x_k} \mathrm{d}x &= \int_0^{\Delta t} v\,\mathrm{d}t \\
x_k - x_0 &= v\Delta t \\
x_k &= v\Delta t + x_0
\end{aligned}$$

This equation is *recursive*: we compute the value of $x$ at time $t$ based on its value at time $t-1$. This recursive form enables us to represent the system (process model) in the form required by the Kalman filter:

$$\begin{aligned}
\mathbf{x}_k &= \mathbf{F}\mathbf{x}_{k-1} \\
&= \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{k-1} \\ \dot{x}_{k-1} \end{bmatrix}
\end{aligned}$$

We can do that only because $\dot{x} = v$ is simplest differential equation possible. Almost all other in physical systems result in more complicated differential equation which do not yield to this approach.

*State-space* methods became popular around the time of the Apollo missions, largely due to the work of Dr. Kalman. The idea is simple. Model a system with a set of $n^{th}$-order differential equations. Convert them into an equivalent set of first-order differential equations. Put them into the vector-matrix form used in the previous section: $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}\mathbf{u}$. Once in this form we use of of several techniques to convert these linear differential equations into the recursive equation:

$$\mathbf{x}_k = \mathbf{F}\mathbf{x}_{k-1} + \mathbf{B}_k\mathbf{u}_k$$

Some books call the state transition matrix the *fundamental matrix*. Many use  instead of $\mathbf{F}$. Sources based heavily on control theory tend to use these forms.

These are called *state-space* methods because we are expressing the solution of the differential equations in terms of the system state.

### 7.2.1 Forming First Order Equations from Higher Order Equations

Many models of physical systems require second or higher order differential equations with control input $u$:

$$a_n\frac{d^n y}{dt^n} + a_{n-1}\frac{d^{n-1} y}{dt^{n-1}} + \cdots + a_2\frac{d^2 y}{dt^2} + a_1\frac{dy}{dt} + a_0 = u$$

State-space methods require first-order equations. Any higher order system of equations can be reduced to first-order by defining extra variables for the derivatives and then solving.

Let's do an example. Given the system $\ddot{x} - 6\dot{x} + 9x = u$ find the equivalent first order equations. I've used the dot notation for the time derivatives for clarity.

The first step is to isolate the highest order term onto one side of the equation.

$$\ddot{x} = 6\dot{x} - 9x + u$$

We define two new variables:

$$x_1(u) = x$$
$$x_2(u) = \dot{x}$$

Now we will substitute these into the original equation and solve. The solution yields a set of first-order equations in terms of these new variables. It is conventional to drop the $(u)$ for notational convenience.

We know that $\dot{x}_1 = x_2$ and that $\dot{x}_2 = \ddot{x}$. Therefore

$$\begin{aligned}\dot{x}_2 &= \ddot{x} \\ &= 6\dot{x} - 9x + t \\ &= 6x_2 - 9x_1 + t\end{aligned}$$

Therefore our first-order system of equations is

$$\dot{x}_1 = x_2$$
$$\dot{x}_2 = 6x_2 - 9x_1 + t$$

If you practice this a bit you will become adept at it. Isolate the highest term, define a new variable and its derivatives, and then substitute.

### 7.2.2 First Order Differential Equations In State-Space Form

Substituting the newly defined variables from the previous section:

$$\frac{dx_1}{dt} = x_2,\ \frac{dx_2}{dt} = x_3,\ ...,\ \frac{dx_{n-1}}{dt} = x_n$$

into the first order equations yields:

$$\frac{dx_n}{dt} = \frac{1}{a_n}\sum_{i=0}^{n-1} a_i x_{i+1} + \frac{1}{a_n}u$$

Using vector-matrix notation we have:

$$\begin{bmatrix} \frac{dx_1}{dt} \\ \frac{dx_2}{dt} \\ \vdots \\ \frac{dx_n}{dt} \end{bmatrix} = \begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \vdots \\ \dot{x}_n \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ -\frac{a_0}{a_n} & -\frac{a_1}{a_n} & -\frac{a_2}{a_n} & \cdots & -\frac{a_{n-1}}{a_n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \frac{1}{a_n} \end{bmatrix} u$$

which we then write as $\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{B}u$.

### 7.2.3 Finding the Fundamental Matrix for Time Invariant Systems

We express the system equations in state-space form with

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$$

where $\mathbf{A}$ is the system dynamics matrix, and want to find the *fundamental matrix* $\mathbf{F}$ that propagates the state $\mathbf{x}$ over the interval $\Delta t$ with the equation

$$\mathbf{x}(t_k) = \mathbf{F}(\Delta t)\mathbf{x}(t_{k-1})$$

In other words, $\mathbf{A}$ is a set of continuous differential equations, and we need $\mathbf{F}$ to be a set of discrete linear equations that computes the change in $\mathbf{A}$ over a discrete time step.

It is conventional to drop the $t_k$ and $(\Delta t)$ and use the notation

$$\mathbf{x}_k = \mathbf{F}\mathbf{x}_{k-1}$$

Broadly speaking there are three common ways to find this matrix for Kalman filters. The technique most often used is the matrix exponential. Linear Time Invariant Theory, also known as LTI System Theory, is a second technique. Finally, there are numerical techniques. You may know of others, but these three are what you will most likely encounter in the Kalman filter literature and praxis.

### 7.2.4 The Matrix Exponential

The solution to the equation $\frac{dx}{dt} = kx$ can be found by:

$$\begin{aligned}
\frac{dx}{dt} &= kx \\
\frac{dx}{x} &= k\,dt \\
\int \frac{1}{x}\,dx &= \int k\,dt \\
\log x &= kt + c \\
x &= e^{kt+c} \\
x &= e^c e^{kt} \\
x &= c_0 e^{kt}
\end{aligned}$$

Using similar math, the solution to the first-order equation

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x},\ \ \mathbf{x}(0) = \mathbf{x}_0$$

where $\mathbf{A}$ is a constant matrix, is

$$\mathbf{x} = e^{\mathbf{A}t}\mathbf{x}_0$$

Substituting $F = e^{\mathbf{A}t}$, we can write

$$\mathbf{x}_k = \mathbf{F}\mathbf{x}_{k-1}$$

which is the form we are looking for! We have reduced the problem of finding the fundamental matrix to one of finding the value for $e^{\mathbf{A}t}$.

$e^{\mathbf{A}t}$ is known as the matrix exponential. It can be computed with this power series:

$$e^{\mathbf{A}t} = \mathbf{I} + \mathbf{A}t + \frac{(\mathbf{A}t)^2}{2!} + \frac{(\mathbf{A}t)^3}{3!} + ...$$

That series is found by doing a Taylor series expansion of $e^{\mathbf{A}t}$, which I will not cover here.

Let's use this to find the solution to Newton's equations. Using $v$ as a substitution for $\dot{x}$, and assuming constant velocity we get the linear matrix-vector form

$$\begin{bmatrix}\dot{x} \\ \dot{v}\end{bmatrix} = \begin{bmatrix}0 & 1 \\ 0 & 0\end{bmatrix}\begin{bmatrix}x \\ v\end{bmatrix}$$

This is a first order differential equation, so we can set $\mathbf{A} = \begin{bmatrix}0 & 1 \\ 0 & 0\end{bmatrix}$ and solve the following equation. I have substituted the interval $\Delta t$ for $t$ to emphasize that the fundamental matrix is discrete:

$$\mathbf{F} = e^{\mathbf{A}\Delta t} = \mathbf{I} + \mathbf{A}\Delta t + \frac{(\mathbf{A}\Delta t)^2}{2!} + \frac{(\mathbf{A}\Delta t)^3}{3!} + ...$$

If you perform the multiplication you will find that $\mathbf{A}^2 = \begin{bmatrix}0 & 0 \\ 0 & 0\end{bmatrix}$, which means that all higher powers of $\mathbf{A}$ are also $\mathbf{0}$. Thus we get an exact answer without an infinite number of terms:

$$\begin{aligned}\mathbf{F} &= \mathbf{I} + \mathbf{A}\Delta t + \mathbf{0} \\ &= \begin{bmatrix}1 & 0 \\ 0 & 1\end{bmatrix} + \begin{bmatrix}0 & 1 \\ 0 & 0\end{bmatrix}\Delta t \\ &= \begin{bmatrix}1 & \Delta t \\ 0 & 1\end{bmatrix}\end{aligned}$$

We plug this into $\mathbf{x}_k = \mathbf{F}\mathbf{x}_{k-1}$ to get

$$x_k = \begin{bmatrix}1 & \Delta t \\ 0 & 1\end{bmatrix} x_{k-1}$$

You will recognize this as the matrix we derived analytically for the constant velocity Kalman filter in the **Multivariate Kalman Filter** chapter.

SciPy's linalg module includes a routine `expm()` to compute the matrix exponential. It does not use the Taylor series method, but the Padé Approximation. There are many (at least 19) methods to computed the matrix exponential, and all suffer from numerical difficulties[1]. You should be aware of the problems, especially when $\mathbf{A}$ is large. If you search for "pade approximation matrix exponential" you will find many publications devoted to this problem.

In practice this may not be of concern to you as for the Kalman filter we normally just take the first two terms of the Taylor series. But don't assume my treatment of the problem is complete and run off and try to use this technique for other problem without doing a numerical analysis of the performance of this technique. Interestingly, one of the favored ways of solving $e^{\mathbf{A}t}$ is to use a generalized ode solver. In other words, they do the opposite of what we do - turn $\mathbf{A}$ into a set of differential equations, and then solve that set using numerical techniques!

Here is an example of using `expm()` to solve $e^{\mathbf{A}t}$.

```
In [3]: import numpy as np
        from scipy.linalg import expm

        dt = 0.1
        A = np.array([[0, 1],
                      [0, 0]])
        expm(A*dt)
```

```
Out[3]: array([[1.  , 0.1],
               [0.  , 1. ]])
```

### 7.2.5 Time Invariance

If the behavior of the system depends on time we can say that a dynamic system is described by the first-order differential equation

$$g(t) = \dot{x}$$

However, if the system is *time invariant* the equation is of the form:

$$f(x) = \dot{x}$$

What does *time invariant* mean? Consider a home stereo. If you input a signal $x$ into it at time $t$, it will output some signal $f(x)$. If you instead perform the input at time $t + \Delta t$ the output signal will be the same $f(x)$, shifted in time.

A counter-example is $x(t) = \sin(t)$, with the system $f(x) = t\,x(t) = t\sin(t)$. This is not time invariant; the value will be different at different times due to the multiplication by t. An aircraft is not time invariant. If you make a control input to the aircraft at a later time its behavior will be different because it will have burned fuel and thus lost weight. Lower weight results in different behavior.

We can solve these equations by integrating each side. I demonstrated integrating the time invariant system $v = \dot{x}$ above. However, integrating the time invariant equation $\dot{x} = f(x)$ is not so straightforward. Using the *separation of variables* techniques we divide by $f(x)$ and move the $dt$ term to the right so we can integrate each side:

$$\frac{dx}{dt} = f(x)$$
$$\int_{x_0}^{x} \frac{1}{f(x)} dx = \int_{t_0}^{t} dt$$

If we let $F(x) = \int \frac{1}{f(x)} dx$ we get

$$F(x) - F(x_0) = t - t_0$$

We then solve for x with

$$F(x) = t - t_0 + F(x_0)$$
$$x = F^{-1}[t - t_0 + F(x_0)]$$

In other words, we need to find the inverse of $F$. This is not trivial, and a significant amount of coursework in a STEM education is devoted to finding tricky, analytic solutions to this problem.

However, they are tricks, and many simple forms of $f(x)$ either have no closed form solution or pose extreme difficulties. Instead, the practicing engineer turns to state-space methods to find approximate solutions.

The advantage of the matrix exponential is that we can use it for any arbitrary set of differential equations which are *time invariant*. However, we often use this technique even when the equations are not time invariant. As an aircraft flies it burns fuel and loses weight. However, the weight loss over one second is negligible, and so the system is nearly linear over that time step. Our answers will still be reasonably accurate so long as the time step is short.

#### Example: Mass-Spring-Damper Model

Suppose we wanted to track the motion of a weight on a spring and connected to a damper, such as an automobile's suspension. The equation for the motion with $m$ being the mass, $k$ the spring constant, and $c$ the damping force, under some input $u$ is

$$m\frac{d^2x}{dt^2} + c\frac{dx}{dt} + kx = u$$

For notational convenience I will write that as

$$m\ddot{x} + c\dot{x} + kx = u$$

I can turn this into a system of first order equations by setting $x_1(t) = x(t)$, and then substituting as follows:

$$\begin{aligned} x_1 &= x \\ x_2 &= \dot{x}_1 \\ \dot{x}_2 &= \ddot{x}_1 = \ddot{x} \end{aligned}$$

As is common I dropped the $(t)$ for notational convenience. This gives the equation

$$m\dot{x}_2 + cx_2 + kx_1 = u$$

Solving for $\dot{x}_2$ we get a first order equation:

$$\dot{x}_2 = -\frac{c}{m}x_2 - \frac{k}{m}x_1 + \frac{1}{m}u$$

We put this into matrix form:

$$\begin{bmatrix}\dot{x}_1 \\ \dot{x}_2\end{bmatrix} = \begin{bmatrix}0 & 1 \\ -k/m & -c/m\end{bmatrix}\begin{bmatrix}x_1 \\ x_2\end{bmatrix} + \begin{bmatrix}0 \\ 1/m\end{bmatrix}u$$

Now we use the matrix exponential to find the state transition matrix:

$$\Phi(t) = e^{\mathbf{A}t} = \mathbf{I} + \mathbf{A}t + \frac{(\mathbf{A}t)^2}{2!} + \frac{(\mathbf{A}t)^3}{3!} + ...$$

The first two terms give us

$$\mathbf{F} = \begin{bmatrix}1 & t \\ -(k/m)t & 1 - (c/m)t\end{bmatrix}$$

This may or may not give you enough precision. You can easily check this by computing $\frac{(\mathbf{A}t)^2}{2!}$ for your constants and seeing how much this matrix contributes to the results.

### 7.2.6 Linear Time Invariant Theory

*Linear Time Invariant Theory*, also known as LTI System Theory, gives us a way to find $\Phi$ using the inverse Laplace transform. You are either nodding your head now, or completely lost. I will not be using the Laplace transform in this book. LTI system theory tells us that

$$\Phi(t) = \mathcal{L}^{-1}[(s\mathbf{I} - \mathbf{A})^{-1}]$$

I have no intention of going into this other than to say that the Laplace transform $\mathcal{L}$ converts a signal into a space $s$ that excludes time, but finding a solution to the equation above is non-trivial. If you are interested, the Wikipedia article on LTI system theory provides an introduction. I mention LTI because you will find some literature using it to design the Kalman filter matrices for difficult problems.

### 7.2.7 Numerical Solutions

Finally, there are numerical techniques to find $\mathbf{F}$. As filters get larger finding analytical solutions becomes very tedious (though packages like SymPy make it easier). C. F. van Loan [2] has developed a technique that finds both $\Phi$ and $\mathbf{Q}$ numerically. Given the continuous model

$$\dot{x} = Ax + Gw$$

where $w$ is the unity white noise, van Loan's method computes both $\mathbf{F}_k$ and $\mathbf{Q}_k$.

I have implemented van Loan's method in `FilterPy`. You may use it as follows:

```
from filterpy.common import van_loan_discretization

A = np.array([[0., 1.], [-1., 0.]])
G = np.array([[0.], [2.]]) # white noise scaling
F, Q = van_loan_discretization(A, G, dt=0.1)
```

In the section *Numeric Integration of Differential Equations* I present alternative methods which are very commonly used in Kalman filtering.

## 7.3 Design of the Process Noise Matrix

In general the design of the $\mathbf{Q}$ matrix is among the most difficult aspects of Kalman filter design. This is due to several factors. First, the math requires a good foundation in signal theory. Second, we are trying to model the noise in something for which we have little information. Consider trying to model the process noise for a thrown baseball. We can model it as a sphere moving through the air, but that leaves many unknown factors - ball rotation and spin decay, the coefficient of drag of a ball with stitches, the effects of wind and air density, and so on. We develop the equations for an exact mathematical solution for a given process model, but since the process model is incomplete the result for $\mathbf{Q}$ will also be incomplete. This has a lot of ramifications for the behavior of the Kalman filter. If $\mathbf{Q}$ is too small then the filter will be overconfident in its prediction model and will diverge from the actual solution. If $\mathbf{Q}$ is too large than the filter will be unduly influenced by the noise in the measurements and perform sub-optimally. In practice we spend a lot of time running simulations and evaluating collected data to try to select an appropriate value for $\mathbf{Q}$. But let's start by looking at the math.

Let's assume a kinematic system - some system that can be modeled using Newton's equations of motion. We can make a few different assumptions about this process.

We have been using a process model of

$$\dot{\mathbf{x}} = \mathbf{Ax} + \mathbf{Bu} + \mathbf{w}$$

where $\mathbf{w}$ is the process noise. Kinematic systems are *continuous* - their inputs and outputs can vary at any arbitrary point in time. However, our Kalman filters are *discrete* (there are continuous forms for Kalman filters, but we do not cover them in this book). We sample the system at regular intervals. Therefore we must find the discrete representation for the noise term in the equation above. This depends on what assumptions we make about the behavior of the noise. We will consider two different models for the noise.

### 7.3.1 Continuous White Noise Model

We model kinematic systems using Newton's equations. We have either used position and velocity, or position, velocity, and acceleration as the models for our systems. There is nothing stopping us from going further - we can model jerk, jounce, snap, and so on. We don't do that normally because adding terms beyond the dynamics of the real system degrades the estimate.

Let's say that we need to model the position, velocity, and acceleration. We can then assume that acceleration is constant for each discrete time step. Of course, there is process noise in the system and so the acceleration is not actually constant. The tracked object will alter the acceleration over time due to external, unmodeled forces. In this section we will assume that the acceleration changes by a continuous time zero-mean white noise $w(t)$. In other words, we are assuming that the small changes in velocity average to 0 over time (zero-mean).

Since the noise is changing continuously we will need to integrate to get the discrete noise for the discretization interval that we have chosen. We will not prove it here, but the equation for the discretization of the noise is

$$\mathbf{Q} = \int_0^{\Delta t} \mathbf{F}(t)\mathbf{Q_c}\mathbf{F}^\mathsf{T}(t)dt$$

where $\mathbf{Q_c}$ is the continuous noise. The general reasoning should be clear. $\mathbf{F}(t)\mathbf{Q_c}\mathbf{F}^\mathsf{T}(t)$ is a projection of the continuous noise based on our process model $\mathbf{F}(t)$ at the instant $t$. We want to know how much noise is added to the system over a discrete intervat $\Delta t$, so we integrate this expression over the interval $[0, \Delta t]$.

We know the fundamental matrix for Newtonian systems is

$$F = \begin{bmatrix} 1 & \Delta t & \Delta t^2/2 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix}$$

We define the continuous noise as

$$\mathbf{Q_c} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \Phi_s$$

where $\Phi_s$ is the spectral density of the white noise. This can be derived, but is beyond the scope of this book. See any standard text on stochastic processes for the details. In practice we often do not know the spectral density of the noise, and so this turns into an "engineering" factor - a number we experimentally tune until our filter performs as we expect. You can see that the matrix that $\Phi_s$ is multiplied by effectively assigns the power spectral density to the acceleration term. This makes sense; we assume that the system has constant acceleration except for the variations caused by noise. The noise alters the acceleration.

We could carry out these computations ourselves, but I prefer using SymPy to solve the equation.

$$\mathbf{Q_c} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \Phi_s$$

In [4]:
```
import sympy
from sympy import (init_printing, Matrix, MatMul,
                   integrate, symbols)

init_printing(use_latex='mathjax')
dt, phi = symbols('\Delta{t} \Phi_s')
F_k = Matrix([[1, dt, dt**2/2],
              [0,  1,      dt],
              [0,  0,       1]])
Q_c = Matrix([[0, 0, 0],
              [0, 0, 0],
              [0, 0, 1]])*phi

Q = integrate(F_k * Q_c * F_k.T, (dt, 0, dt))

# factor phi out of the matrix to make it more readable
Q = Q / phi
MatMul(Q, phi)
```

Out[4]:

$$\begin{bmatrix} \frac{\Delta t^5}{20} & \frac{\Delta t^4}{8} & \frac{\Delta t^3}{6} \\ \frac{\Delta t^4}{8} & \frac{\Delta t^3}{3} & \frac{\Delta t^2}{2} \\ \frac{\Delta t^3}{6} & \frac{\Delta t^2}{2} & \Delta t \end{bmatrix} \Phi_s$$

For completeness, let us compute the equations for the 0th order and 1st order equations.

In [5]:
```
F_k = Matrix([[1]])
Q_c = Matrix([[phi]])

print('0th order discrete process noise')
integrate(F_k*Q_c*F_k.T,(dt, 0, dt))
```

```
0th order discrete process noise
```

Out[5]:

$$\begin{bmatrix}\Delta t \Phi_s\end{bmatrix}$$

In [6]:
```
F_k = Matrix([[1, dt],
              [0, 1]])
Q_c = Matrix([[0, 0],
              [0, 1]]) * phi

Q = integrate(F_k * Q_c * F_k.T, (dt, 0, dt))

print('1st order discrete process noise')
# factor phi out of the matrix to make it more readable
Q = Q / phi
MatMul(Q, phi)
```

```
1st order discrete process noise
```

Out[6]:

$$\begin{bmatrix}\frac{\Delta t^3}{3} & \frac{\Delta t^2}{2} \\ \frac{\Delta t^2}{2} & \Delta t\end{bmatrix} \Phi_s$$

### 7.3.2 Piecewise White Noise Model

Another model for the noise assumes that the that highest order term (say, acceleration) is constant for the duration of each time period, but differs for each time period, and each of these is uncorrelated between time periods. In other words there is a discontinuous jump in acceleration at each time step. This is subtly different than the model above, where we assumed that the last term had a continuously varying noisy signal applied to it.

We will model this as

$$f(x) = Fx + \Gamma w$$

where $\Gamma$ is the *noise gain* of the system, and $w$ is the constant piecewise acceleration (or velocity, or jerk, etc).

Let's start by looking at a first order system. In this case we have the state transition function

$$\mathbf{F} = \begin{bmatrix}1 & \Delta t \\ 0 & 1\end{bmatrix}$$

In one time period, the change in velocity will be $w(t)\Delta t$, and the change in position will be $w(t)\Delta t^2/2$, giving us

$$\Gamma = \begin{bmatrix}\frac{1}{2}\Delta t^2 \\ \Delta t\end{bmatrix}$$

The covariance of the process noise is then

$$Q = \mathbb{E}[\Gamma w(t) w(t) \Gamma^\mathsf{T}] = \Gamma \sigma_v^2 \Gamma^\mathsf{T}$$

.

We can compute that with SymPy as follows

```
In [7]: var = symbols('sigma^2_v')
        v = Matrix([[dt**2 / 2], [dt]])

        Q = v * var * v.T

        # factor variance out of the matrix to make it more readable
        Q = Q / var
        MatMul(Q, var)
```

Out[7]:

$$\begin{bmatrix} \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} \\ \frac{\Delta t^3}{2} & \Delta t^2 \end{bmatrix} \sigma_v^2$$

The second order system proceeds with the same math.

$$\mathbf{F} = \begin{bmatrix} 1 & \Delta t & \Delta t^2/2 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix}$$

Here we will assume that the white noise is a discrete time Wiener process. This gives us

$$\Gamma = \begin{bmatrix} \frac{1}{2}\Delta t^2 \\ \Delta t \\ 1 \end{bmatrix}$$

There is no 'truth' to this model, it is just convenient and provides good results. For example, we could assume that the noise is applied to the jerk at the cost of a more complicated equation.

The covariance of the process noise is then

$$Q = \mathbb{E}[\Gamma w(t) w(t) \Gamma^\mathsf{T}] = \Gamma \sigma_v^2 \Gamma^\mathsf{T}$$

.

We can compute that with SymPy as follows

```
In [8]: var = symbols('sigma^2_v')
        v = Matrix([[dt**2 / 2], [dt], [1]])

        Q = v * var * v.T

        # factor variance out of the matrix to make it more readable
        Q = Q / var
        MatMul(Q, var)
```

Out[8]:

$$\begin{bmatrix} \frac{\Delta t^4}{4} & \frac{\Delta t^3}{2} & \frac{\Delta t^2}{2} \\ \frac{\Delta t^3}{2} & \Delta t^2 & \Delta t \\ \frac{\Delta t^2}{2} & \Delta t & 1 \end{bmatrix} \sigma_v^2$$

We cannot say that this model is more or less correct than the continuous model - both are approximations to what is happening to the actual object. Only experience and experiments can guide you to the appropriate model. In practice you will usually find that either model provides reasonable results, but typically one will perform better than the other.

The advantage of the second model is that we can model the noise in terms of $\sigma^2$ which we can describe in terms of the motion and the amount of error we expect. The first model requires us to specify the spectral density, which is not very intuitive, but it handles varying time samples much more easily since the noise is integrated across the time period. However, these are not fixed rules - use whichever model (or a model of your own devising) based on testing how the filter performs and/or your knowledge of the behavior of the physical model.

A good rule of thumb is to set $\sigma$ somewhere from $\frac{1}{2}\Delta a$ to $\Delta a$, where $\Delta a$ is the maximum amount that the acceleration will change between sample periods. In practice we pick a number, run simulations on data, and choose a value that works well.

### 7.3.3 Using FilterPy to Compute Q

FilterPy offers several routines to compute the $\mathbf{Q}$ matrix. The function `Q_continuous_white_noise()` computes $\mathbf{Q}$ for a given value for $\Delta t$ and the spectral density.

```
In [9]: from filterpy.common import Q_continuous_white_noise
        from filterpy.common import Q_discrete_white_noise

        Q = Q_continuous_white_noise(dim=2, dt=1, spectral_density=1)
        print(Q)
```

```
[[0.333 0.5  ]
 [0.5   1.   ]]
```

```
In [10]: Q = Q_continuous_white_noise(dim=3, dt=1, spectral_density=1)
         print(Q)
```

```
[[0.05  0.125 0.167]
 [0.125 0.333 0.5  ]
 [0.167 0.5   1.   ]]
```

The function `Q_discrete_white_noise()` computes $\mathbf{Q}$ assuming a piecewise model for the noise.

```
In [11]: Q = Q_discrete_white_noise(2, var=1.)
         print(Q)
```

```
[[0.25 0.5 ]
 [0.5  1.  ]]
```

```
In [12]: Q = Q_discrete_white_noise(3, var=1.)
         print(Q)
```

```
[[0.25 0.5  0.5 ]
 [0.5  1.   1.  ]
 [0.5  1.   1.  ]]
```

### 7.3.4 Simplification of Q

Many treatments use a much simpler form for $\mathbf{Q}$, setting it to zero except for a noise term in the lower rightmost element. Is this justified? Well, consider the value of $\mathbf{Q}$ for a small $\Delta t$

```
In [13]: import numpy as np

         np.set_printoptions(precision=8)
         Q = Q_continuous_white_noise(
             dim=3, dt=0.05, spectral_density=1)
         print(Q)
         np.set_printoptions(precision=3)
```

```
[[0.00000002 0.00000078 0.00002083]
 [0.00000078 0.00004167 0.00125   ]
 [0.00002083 0.00125    0.05      ]]
```

We can see that most of the terms are very small. Recall that the only equation using this matrix is

$$\mathbf{P} = \mathbf{FPF}^\mathsf{T} + \mathbf{Q}$$

If the values for $\mathbf{Q}$ are small relative to $\mathbf{P}$ then it will be contributing almost nothing to the computation of $\mathbf{P}$. Setting $\mathbf{Q}$ to the zero matrix except for the lower right term

$$\mathbf{Q} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \sigma^2 \end{bmatrix}$$

while not correct, is often a useful approximation. If you do this for an important application you will have to perform quite a few studies to guarantee that your filter works in a variety of situations.

If you do this, 'lower right term' means the most rapidly changing term for each variable. If the state is $x = \begin{bmatrix} x & \dot{x} & \ddot{x} & y & \dot{y} & \ddot{y} \end{bmatrix}^\mathsf{T}$ Then Q will be 6x6; the elements for both $\ddot{x}$ and $\ddot{y}$ will have to be set to non-zero in $\mathbf{Q}$.

## 7.4 Stable Compution of the Posterior Covariance

I've presented the equation to compute the posterior covariance as

$$\mathbf{P} = (\mathbf{I} - \mathbf{KH})\mathbf{P}$$

and while strictly speaking this is correct, this is not how I compute it in `FilterPy`, where I use the *Joseph* equation

$$\mathbf{P} = (\mathbf{I} - \mathbf{KH})\mathbf{P}(\mathbf{I} - \mathbf{KH})^\mathsf{T} + \mathbf{KRK}^\mathsf{T}$$

I frequently get emails and/or GitHub issues raised, claiming the implementation is a bug. It is not a bug, and I use it for several reasons. First, the subtraction $(\mathbf{I} - \mathbf{KH})$ can lead to nonsymmetric matrices results due to floating point errors. Covariances must be symmetric, and so becoming nonsymmetric usually leads to the Kalman filter diverging, or even for the code to raise an exception because of the checks built into `NumPy`.

A traditional way to preserve symmetry is the following formula:

$$\mathbf{P} = (\mathbf{P} + \mathbf{P}^\mathsf{T})/2$$

This is safe because $\sigma_{ij} = \sigma_{ji}$ for all covariances in the matrix. Hence this operation averages the error between the differences of the two values if they have diverged due to floating point errors.

If you look at the Joseph form for the equation above, you'll see there is a similar $\mathbf{ABA}^\mathsf{T}$ pattern in both terms. So they both preserve symmetry. But where did this equation come from, and why do I use it instead of

$$\mathbf{P} = (\mathbf{I} - \mathbf{KH})\mathbf{PP} = (\mathbf{P} + \mathbf{P}^\mathsf{T})/2$$

Let's just derive the equation from first principles. It's not too bad, and you need to understand the derivation to understand the purpose of the equation, and, more importantly, diagnose issues if you filter diverges due to numerical instability. This derivation comes from Brown[4].

First, some symbology. $\mathbf{x}$ is the true state of our system. $\widehat{\mathbf{x}}$ is the estimated state of our system - the posterior. And $\mathbf{x}$ is the estimated prior of the system.

Given that, we can define our model to be

$$\mathbf{x}_{k+1} = \mathbf{F}_k\mathbf{x}_k + \mathbf{w}_k\mathbf{z}_k = \mathbf{H}_k\mathbf{x}_k + \mathbf{v}_k$$

In words, the next state $\mathbf{x}_{k+1}$ of the system is the current state $k$ moved by some process $\mathbf{F}_k$ plus some noise $\mathbf{w}_k$.

Note that these are definitions. No system perfectly follows a mathematical model, so we model that with the noise term $\mathbf{w}_k$. And no measurement is perfect due to sensor error, so we model that with $\mathbf{v}_k$

I'll dispense with the subscript $k$ since in the remainder of the derivation we will only consider values at step $k$, never step $k + 1$.

Now we define the estimation error as the difference between the true state and the estimated state

$$\mathbf{e} = \mathbf{x} - \widehat{\mathbf{x}}$$

Again, this is a definition; we don't know how to compute $\mathbf{e}$, it is just the defined difference between the true and estimated state.

This allows us to define the covariance of our estimate, which is defined as the expected value of $\mathbf{ee}^\mathsf{T}$:

$$\begin{aligned} P &= E[\mathbf{ee}^\mathsf{T}] \\ &= E[(\mathbf{x} - \widehat{\mathbf{x}})(\mathbf{x} - \widehat{\mathbf{x}})^\mathsf{T}] \end{aligned}$$

Next, we define the posterior estimate as

$$\widehat{\mathbf{x}} = \mathbf{x} + \mathbf{K}(\mathbf{z} - \mathbf{H}\mathbf{x})$$

That looks like the equation from the Kalman filter, and for good reason. But as with the rest of the math so far, this is a **definition**. In particular, we have not defined $\mathbf{K}$, and you shouldn't think of it as the Kalman gain, because we are solving this for *any* problem, not just for linear Kalman filters. Here, $\mathbf{K}$ is just some unspecified blending value between 0 and 1.

Now we have our definitions, let's perform some substitution and algebra.

The term $(\mathbf{x} - \widehat{\mathbf{x}})$ can be expanded by replacing $\widehat{\mathbf{x}}$ with the definition above, yielding

$$(\mathbf{x} - \widehat{\mathbf{x}}) = \mathbf{x} - (\mathbf{x} + \mathbf{K}(\mathbf{z} - \mathbf{H}\mathbf{x}))$$

Now we replace $\mathbf{z}$ with $\mathbf{H}\mathbf{x} + \mathbf{v}$:

$$\begin{aligned} (\mathbf{x} - \widehat{\mathbf{x}}) &= \mathbf{x} - (\mathbf{x} + \mathbf{K}(\mathbf{z} - \mathbf{H}\mathbf{x})) \\ &= \mathbf{x} - (\mathbf{x} + \mathbf{K}(\mathbf{H}\mathbf{x} + \mathbf{v} - \mathbf{H}\mathbf{x})) \\ &= (\mathbf{x} - \mathbf{x}) - \mathbf{K}(\mathbf{H}\mathbf{x} + \mathbf{v} - \mathbf{H}\mathbf{x}) \\ &= (\mathbf{x} - \mathbf{x}) - \mathbf{K}\mathbf{H}(\mathbf{x} - \mathbf{x}) - \mathbf{K}\mathbf{v} \\ &= (\mathbf{I} - \mathbf{K}\mathbf{H})(\mathbf{x} - \mathbf{x}) - \mathbf{K}\mathbf{v} \end{aligned}$$

Now we can solve for $\mathbf{P}$ if we note that the expected value of $(\mathbf{x} - \mathbf{x})$ is the prior covariance $\mathbf{P}$, and that the expected value of $\mathbf{v}$ is $E[\mathbf{v}\mathbf{v}^\mathbf{T}] = \mathbf{R}$:

$$\begin{aligned} \mathbf{P} &= E\big[[(\mathbf{I} - \mathbf{K}\mathbf{H})(\mathbf{x} - \mathbf{x}) - \mathbf{K}\mathbf{v})][(\mathbf{I} - \mathbf{K}\mathbf{H})(\mathbf{x} - \mathbf{x}) - \mathbf{K}\mathbf{v}]^\mathsf{T}\big] \\ &= (\mathbf{I} - \mathbf{K}\mathbf{H})\mathbf{P}(\mathbf{I} - \mathbf{K}\mathbf{H})^\mathsf{T} + \mathbf{K}\mathbf{R}\mathbf{K}^\mathsf{T} \end{aligned}$$

which is what we came here to prove.

Note that this equation is valid for *any* $\mathbf{K}$, not just the optimal $\mathbf{K}$ computed by the Kalman filter. And that is why I use this equation. In practice the Kalman gain computed by the filter is *not* the optimal value both because the real world is never truly linear and Gaussian, and because of floating point errors induced by computation. This equation is far less likely to cause the Kalman filter to diverge in the face of real world conditions.

Where did $\mathbf{P} = (\mathbf{I} - \mathbf{K}\mathbf{H})\mathbf{P}$ come from, then? Let's finish the derivation, which is simple. Recall that the Kalman filter (optimal) gain is given by

$$\mathbf{K} = \mathbf{P}\mathbf{H}^\mathsf{T}(\mathbf{H}\mathbf{P}\mathbf{H}^\mathsf{T} + \mathbf{R})^{-1}$$

Now we substitute this into the equation we just derived:

$$
\begin{aligned}
&= (\mathbf{I} - \mathbf{KH})\mathbf{P}(\mathbf{I} - \mathbf{KH})^\mathsf{T} + \mathbf{KRK}^\mathsf{T} \\
&= \mathbf{P} - \mathbf{KHP} - \mathbf{PH}^\mathsf{T}\mathbf{K}^\mathsf{T} + \mathbf{K}(\mathbf{HPH}^\mathsf{T} + \mathbf{R})\mathbf{K}^\mathsf{T} \\
&= \mathbf{P} - \mathbf{KHP} - \mathbf{PH}^\mathsf{T}\mathbf{K}^\mathsf{T} + \mathbf{PH}^\mathsf{T}(\mathbf{HPH}^\mathsf{T} + \mathbf{R})^{-1}(\mathbf{HPH}^\mathsf{T} + \mathbf{R})\mathbf{K}^\mathsf{T} \\
&= \mathbf{P} - \mathbf{KHP} - \mathbf{PH}^\mathsf{T}\mathbf{K}^\mathsf{T} + \mathbf{PH}^\mathsf{T}\mathbf{K}^\mathsf{T} \\
&= \mathbf{P} - \mathbf{KHP} \\
&= (\mathbf{I} - \mathbf{KH})\mathbf{P}
\end{aligned}
$$

Therefore $\mathbf{P} = (\mathbf{I} - \mathbf{KH})\mathbf{P}$ is mathematically correct when the gain is optimal, but so is $(\mathbf{I} - \mathbf{KH})\mathbf{P}(\mathbf{I} - \mathbf{KH})^\mathsf{T} + \mathbf{KRK}^\mathsf{T}$. As we already discussed the latter is also correct when the gain is suboptimal, and it is also more numerically stable. Therefore I use this computation in FilterPy.

It is quite possible that your filter still diverges, especially if it runs for hundreds or thousands of epochs. You will need to examine these equations. The literature provides yet other forms of this computation which may be more applicable to your problem. As always, if you are solving real engineering problems where failure could mean loss of equipment or life, you will need to move past this book and into the engineering literature. If you are working with 'toy' problems where failure is not damaging, if you detect divergence you can just reset the value of $\mathbf{P}$ to some 'reasonable' value and keep on going. For example, you could zero out the non diagonal elements so the matrix only contains variances, and then maybe multiply by a constant somewhat larger than one to reflect the loss of information you just injected into the filter. Use your imagination, and test.

## 7.5 Deriving the Kalman Gain Equation

If you read the last section, you might as well read this one. With this we will have derived the Kalman filter equations.

Note that this derivation is *not* using Bayes equations. I've seen at least four different ways to derive the Kalman filter equations; this derivation is typical to the literature, and follows from the last section. The source is again Brown [4].

In the last section we used an unspecified scaling factor $\mathbf{K}$ to derive the Joseph form of the covariance equation. If we want an optimal filter, we need to use calculus to minimize the errors in the equations. You should be familiar with this idea. If you want to find the minimum value of a function $f(x)$, you take the derivative and set it equal to zero: $\frac{x}{dx}f(x) = 0$.

In our problem the error is expressed by the covariance matrix $\mathbf{P}$. In particular, the diagonal expresses the error (variance) of each element in the state vector. So, to find the optimal gain we want to take the derivative of the trace (sum) of the diagonal.

Brown reminds us of two formulas involving the derivative of traces:

$$\frac{d\,trace(\mathbf{AB})}{d\mathbf{A}} = \mathbf{B}^\mathsf{T}$$

$$\frac{d\,trace(\mathbf{ACA}^\mathsf{T})}{d\mathbf{A}} = 2\mathbf{AC}$$

where $\mathbf{AB}$ is square and $\mathbf{C}$ is symmetric.

We expand out the Joseph equation to:

$$\mathbf{P} = \mathbf{P} - \mathbf{KHP} - \mathbf{PH}^\mathsf{T}\mathbf{K}^\mathsf{T} + \mathbf{K}(\mathbf{HPH}^\mathsf{T} + \mathbf{R})\mathbf{K}^\mathsf{T}$$

Now we need to the the derivative of the trace of $\mathbf{P}$ with respect to $\mathbf{T}$: $\frac{d\,trace(\mathbf{P})}{d\mathbf{K}}$.

The derivative of the trace the first term with respect to $\mathbf{K}$ is 0, since it does not have $\mathbf{K}$ in the expression.

The derivative of the trace of the second term is $(\mathbf{HP})^\mathsf{T}$.

We can find the derivative of the trace of the third term by noticing that $\mathbf{PH}^\mathsf{T}\mathbf{K}^\mathsf{T}$ is the transpose of $\mathbf{KHP}$. The trace of a matrix is equal to the trace of it's transpose, so it's derivative will be same as the second term.

Finally, the derivative of the trace of the fourth term is $2\mathbf{K}(\mathbf{HPH}^\mathsf{T} + \mathbf{R})$.

This gives us the final value of

$$\frac{d\,trace(\mathbf{P})}{d\mathbf{K}} = -2(\mathbf{HP})^\mathsf{T} + 2\mathbf{K}(\mathbf{HPH}^\mathsf{T} + \mathbf{R})$$

We set this to zero and solve to find the equation for $\mathbf{K}$ which minimizes the error:

$$-2(\mathbf{HP})^\mathsf{T} + 2\mathbf{K}(\mathbf{HPH}^\mathsf{T} + \mathbf{R}) = 0\mathbf{K}(\mathbf{HPH}^\mathsf{T} + \mathbf{R}) = (\mathbf{HP})^\mathsf{T}\mathbf{K}(\mathbf{HPH}^\mathsf{T} + \mathbf{R}) = \mathbf{PH}^\mathsf{T}\mathbf{K} = \mathbf{PH}^\mathsf{T}(\mathbf{HPH}^\mathsf{T} + \mathbf{R})^{-1}$$

This derivation is not quite iron clad as I left out an argument about why minimizing the trace minimizes the total error, but I think it suffices for this book. Any of the standard texts will go into greater detail if you need it.

## 7.6 Numeric Integration of Differential Equations

We've been exposed to several numerical techniques to solve linear differential equations. These include state-space methods, the Laplace transform, and van Loan's method.

These work well for linear ordinary differential equations (ODEs), but do not work well for nonlinear equations. For example, consider trying to predict the position of a rapidly turning car. Cars maneuver by turning the front wheels. This makes them pivot around their rear axle as it moves forward. Therefore the path will be continuously varying and a linear prediction will necessarily produce an incorrect value. If the change in the system is small enough relative to $\Delta t$ this can often produce adequate results, but that will rarely be the case with the nonlinear Kalman filters we will be studying in subsequent chapters.

For these reasons we need to know how to numerically integrate ODEs. This can be a vast topic that requires several books. However, I will cover a few simple techniques which will work for a majority of the problems you encounter.

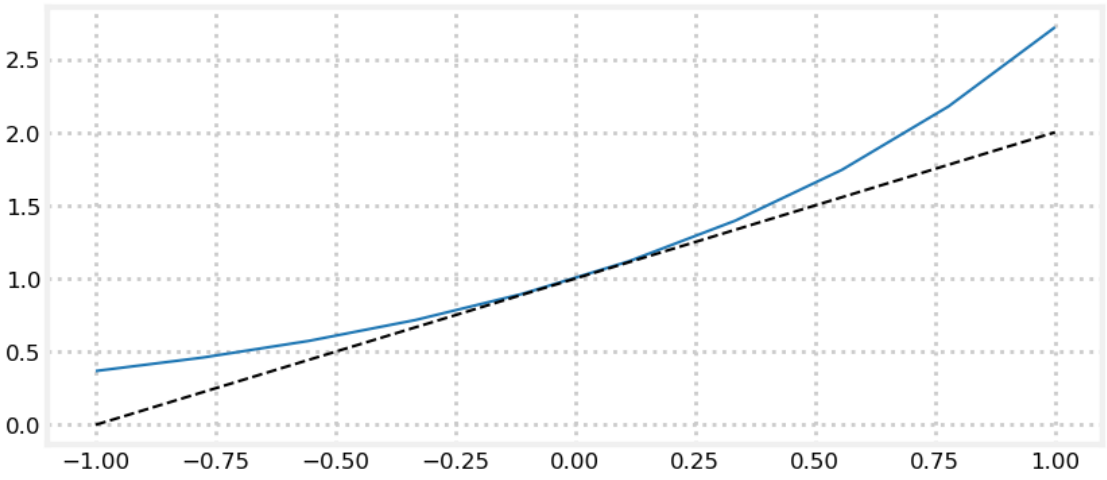
### 7.6.1 Euler's Method

Let's say we have the initial condition problem of

$$y' = y,$$
$$y(0) = 1$$

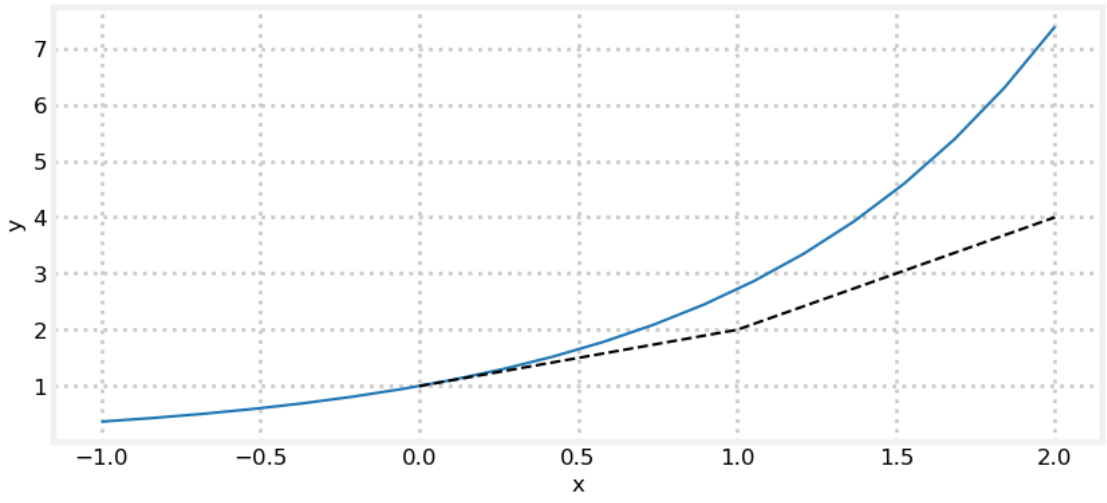
We happen to know the exact answer is $y = e^t$ because we solved it earlier, but for an arbitrary ODE we will not know the exact solution. In general all we know is the derivative of the equation, which is equal to the slope. We also know the initial value: at $t = 0$, $y = 1$. If we know these two pieces of information we can predict the value at $y(t = 1)$ using the slope at $t = 0$ and the value of $y(0)$. I've plotted this below.

```
In [14]: import matplotlib.pyplot as plt

         t = np.linspace(-1, 1, 10)
         plt.plot(t, np.exp(t))
         t = np.linspace(-1, 1, 2)
         plt.plot(t,t+1, ls='--', c='k');
```




You can see that the slope is very close to the curve at $t = 0.1$, but far from it at $t = 1$. But let's continue with a step size of 1 for a moment. We can see that at $t = 1$ the estimated value of $y$ is 2. Now we can compute the value at $t = 2$ by taking the slope of the curve at $t = 1$ and adding it to our initial estimate. The slope is computed with $y' = y$, so the slope is 2.

```
In [15]: import kf_book.book_plots as book_plots

         t = np.linspace(-1, 2, 20)
         plt.plot(t, np.exp(t))
         t = np.linspace(0, 1, 2)
         plt.plot([1, 2, 4], ls='--', c='k')
         book_plots.set_labels(x='x', y='y');
```



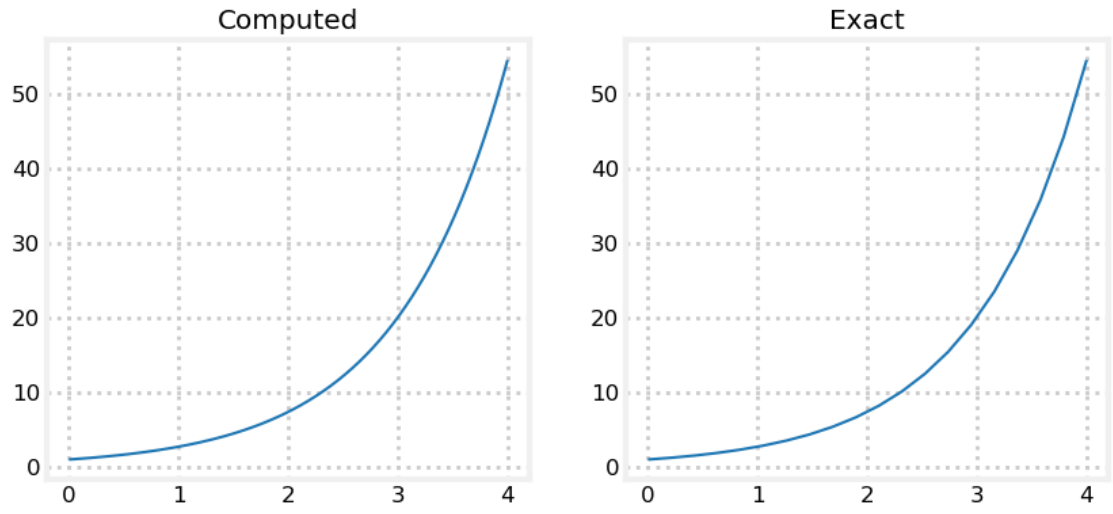

Here we see the next estimate for y is 4. The errors are getting large quickly, and you might be unimpressed. But 1 is a very large step size. Let's put this algorithm in code, and verify that it works by using a small step size.

```
In [16]: def euler(t, tmax, y, dx, step=1.):
             ys = []
             while t < tmax:
                 y = y + step*dx(t, y)
                 ys.append(y)
                 t +=step
             return ys

In [17]: def dx(t, y): return y

         print(euler(0, 1, 1, dx, step=1.)[-1])
         print(euler(0, 2, 1, dx, step=1.)[-1])
```

```
2.0
4.0
```

This looks correct. So now let's plot the result of a much smaller step size.

```
In [18]: ys = euler(0, 4, 1, dx, step=0.00001)
         plt.subplot(1,2,1)
         plt.title('Computed')
         plt.plot(np.linspace(0, 4, len(ys)),ys)
         plt.subplot(1,2,2)
         t = np.linspace(0, 4, 20)
         plt.title('Exact')
         plt.plot(t, np.exp(t));
```




```
In [19]: print('exact answer=', np.exp(4))
         print('euler answer=', ys[-1])
         print('difference =', np.exp(4) - ys[-1])
         print('iterations =', len(ys))
```

```
exact answer= 54.598150033144236
euler answer= 54.59705808834125
```

```
difference = 0.0010019448029866885
iterations = 400000
```

Here we see that the error is reasonably small, but it took a very large number of iterations to get three digits of precision. In practice Euler's method is too slow for most problems, and we use more sophisticated methods.

Before we go on, let's formally derive Euler's method, as it is the basis for the more advanced Runge Kutta methods used in the next section. In fact, Euler's method is the simplest form of Runge Kutta.

Here are the first 3 terms of the Taylor expansion of $y$. An infinite expansion would give an exact answer, so $O(h^4)$ denotes the error due to the finite expansion.

$$y(t_0 + h) = y(t_0) + hy'(t_0) + \frac{1}{2!}h^2y''(t_0) + \frac{1}{3!}h^3y'''(t_0) + O(h^4)$$

Here we can see that Euler's method is using the first two terms of the Taylor expansion. Each subsequent term is smaller than the previous terms, so we are assured that the estimate will not be too far off from the correct value.

### 7.6.2 Runge Kutta Methods

Runge Kutta is the workhorse of numerical integration. There are a vast number of methods in the literature. In practice, using the Runge Kutta algorithm that I present here will solve most any problem you will face. It offers a very good balance of speed, precision, and stability, and it is the 'go to' numerical integration method unless you have a very good reason to choose something different.

Let's dive in. We start with some differential equation

$$\ddot{y} = \frac{d}{dt}\dot{y}$$

.

We can substitute the derivative of y with a function f, like so

$$\ddot{y} = \frac{d}{dt}f(y, t)$$

.

Deriving these equations is outside the scope of this book, but the Runge Kutta RK4 method is defined with these equations.

$$y(t + \Delta t) = y(t) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4) + O(\Delta t^4)$$

$$\begin{aligned}
k_1 &= f(y, t)\Delta t \\
k_2 &= f(y + \frac{1}{2}k_1, t + \frac{1}{2}\Delta t)\Delta t \\
k_3 &= f(y + \frac{1}{2}k_2, t + \frac{1}{2}\Delta t)\Delta t \\
k_4 &= f(y + k_3, t + \Delta t)\Delta t
\end{aligned}$$

Here is the corresponding code:

```
In [20]: def runge_kutta4(y, x, dx, f):
             """computes 4th order Runge-Kutta for dy/dx.
             y is the initial value for y
             x is the initial value for x
             dx is the difference in x (e.g. the time step)
             f is a callable function (y, x) that you supply
             to compute dy/dx for the specified values.
```

```
    """

    k1 = dx * f(y, x)
    k2 = dx * f(y + 0.5*k1, x + 0.5*dx)
    k3 = dx * f(y + 0.5*k2, x + 0.5*dx)
    k4 = dx * f(y + k3, x + dx)

    return y + (k1 + 2*k2 + 2*k3 + k4) / 6.
```

Let's use this for a simple example. Let

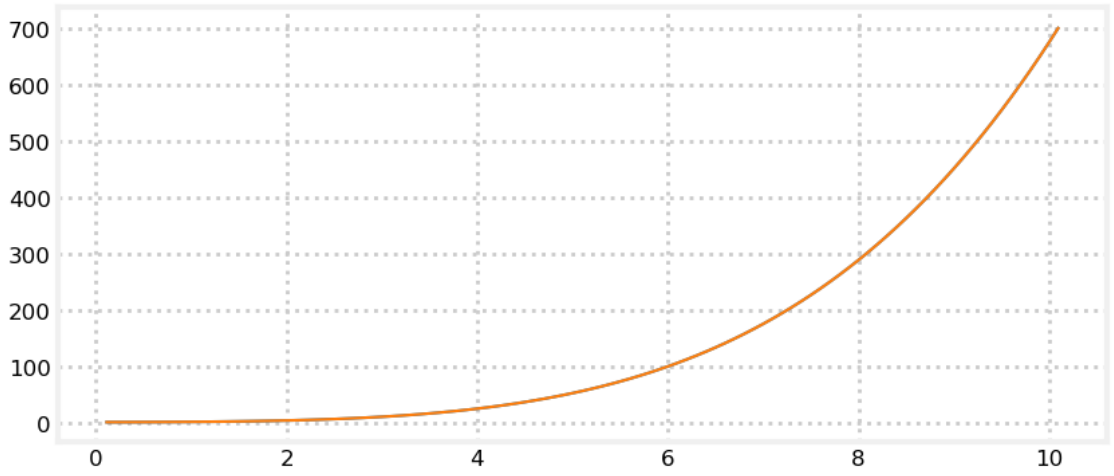
$$\dot{y} = t\sqrt{y(t)}$$

with the initial values

$$t_0 = 0$$
$$y_0 = y(t_0) = 1$$

```
In [21]: import math
         import numpy as np
         t = 0.
         y = 1.
         dt = .1

         ys, ts = [], []

         def func(y,t):
             return t*math.sqrt(y)

         while t <= 10:
             y = runge_kutta4(y, t, dt, func)
             t += dt
             ys.append(y)
             ts.append(t)

         exact = [(t**2 + 4)**2 / 16. for t in ts]
         plt.plot(ts, ys)
         plt.plot(ts, exact)

         error = np.array(exact) - np.array(ys)
         print("max error {:.5f}".format(max(error)))
```

```
max error 0.00005
```




## 7.7 Bayesian Filtering

Starting in the Discrete Bayes chapter I used a Bayesian formulation for filtering. Suppose we are tracking an object. We define its *state* at a specific time as its position, velocity, and so on. For example, we might write the state at time $t$ as $\mathbf{x}_t = \begin{bmatrix} x_t & \dot{x}_t \end{bmatrix}^\mathsf{T}$.

When we take a measurement of the object we are measuring the state or part of it. Sensors are noisy, so the measurement is corrupted with noise. Clearly though, the measurement is determined by the state. That is, a change in state may change the measurement, but a change in measurement will not change the state.

In filtering our goal is to compute an optimal estimate for a set of states $\mathbf{x}_{0:t}$ from time 0 to time $t$. If we knew $\mathbf{x}_{0:t}$ then it would be trivial to compute a set of measurements $\mathbf{z}_{0:t}$ corresponding to those states. However, we receive a set of measurements $\mathbf{z}_{0:t}$, and want to compute the corresponding states $\mathbf{x}_{0:t}$. This is called *statistical inversion* because we are trying to compute the input from the output.

Inversion is a difficult problem because there is typically no unique solution. For a given set of states $\mathbf{x}_{0:t}$ there is only one possible set of measurements (plus noise), but for a given set of measurements there are many different sets of states that could have led to those measurements.

Recall Bayes Theorem:

$$P(x \mid z) = \frac{P(z \mid x)P(x)}{P(z)}$$

where $P(z \mid x)$ is the *likelihood* of the measurement $z$, $P(x)$ is the *prior* based on our process model, and $P(z)$ is a normalization constant. $P(x \mid z)$ is the *posterior*, or the distribution after incorporating the measurement $z$, also called the *evidence*.

This is a *statistical inversion* as it goes from $P(z \mid x)$ to $P(x \mid z)$. The solution to our filtering problem can be expressed as:

$$P(\mathbf{x}_{0:t} \mid \mathbf{z}_{0:t}) = \frac{P(\mathbf{z}_{0:t} \mid \mathbf{x}_{0:t})P(\mathbf{x}_{0:t})}{P(\mathbf{z}_{0:t})}$$

That is all well and good until the next measurement $\mathbf{z}_{t+1}$ comes in, at which point we need to recompute the entire expression for the range $0 : t + 1$.

In practice this is intractable because we are trying to compute the posterior distribution $P(\mathbf{x}_{0:t} \mid \mathbf{z}_{0:t})$ for the state over the full range of time steps. But do we really care about the probability distribution at the third step (say) when we just received the tenth measurement? Not usually. So we relax our requirements and only compute the distributions for the current time step.

The first simplification is we describe our process (e.g., the motion model for a moving object) as a *Markov chain*. That is, we say that the current state is solely dependent on the previous state and a transition probability $P(\mathbf{x}_k \mid \mathbf{x}_{k-1})$, which is just the probability of going from the last state to the current one. We write:

$$\mathbf{x}_k \sim P(\mathbf{x}_k \mid \mathbf{x}_{k-1})$$

In practice this is extremely reasonable, as many things have the *Markov property*. If you are driving in a parking lot, does your position in the next second depend on whether you pulled off the interstate or were creeping along on a dirt road one minute ago? No. Your position in the next second depends solely on your current position, speed, and control inputs, not on what happened a minute ago. Thus, cars have the Markov property, and we can make this simplification with no loss of precision or generality.

The next simplification we make is do define the *measurement model* as depending on the current state $\mathbf{x}_k$ with the conditional probability of the measurement given the current state: $P(\mathbf{z}_t \mid \mathbf{x}_x)$. We write:

$$\mathbf{z}_k \sim P(\mathbf{z}_t \mid \mathbf{x}_x)$$

We have a recurrance now, so we need an initial condition to terminate it. Therefore we say that the initial distribution is the probablity of the state $\mathbf{x}_0$:

$$\mathbf{x}_0 \sim P(\mathbf{x}_0)$$

These terms are plugged into Bayes equation. If we have the state $\mathbf{x}_0$ and the first measurement we can estimate $P(\mathbf{x}_1|\mathbf{z}_1)$. The motion model creates the prior $P(\mathbf{x}_2 \mid \mathbf{x}_1)$. We feed this back into Bayes theorem to compute $P(\mathbf{x}_2|\mathbf{z}_2)$. We continue this predictor-corrector algorithm, recursively computing the state and distribution at time $t$ based solely on the state and distribution at time $t-1$ and the measurement at time $t$.

The details of the mathematics for this computation varies based on the problem. The **Discrete Bayes** and **Univariate Kalman Filter** chapters gave two different formulations which you should have been able to reason through. The univariate Kalman filter assumes that for a scalar state both the noise and process are linear model are affected by zero-mean, uncorrelated Gaussian noise.

The Multivariate Kalman filter make the same assumption but for states and measurements that are vectors, not scalars. Dr. Kalman was able to prove that if these assumptions hold true then the Kalman filter is *optimal* in a least squares sense. Colloquially this means there is no way to derive more information from the noisy measurements. In the remainder of the book I will present filters that relax the constraints on linearity and Gaussian noise.

Before I go on, a few more words about statistical inversion. As Calvetti and Somersalo write in *Introduction to Bayesian Scientific Computing*, "we adopt the Bayesian point of view: *randomness simply means lack of information*."[3] Our state parameterizes physical phenomena that we could in principle measure or compute: velocity, air drag, and so on. We lack enough information to compute or measure their value, so we opt to consider them as random variables. Strictly speaking they are not random, thus this is a subjective position.

They devote a full chapter to this topic. I can spare a paragraph. Bayesian filters are possible because we ascribe statistical properties to unknown parameters. In the case of the Kalman filter we have closed-form solutions to find an optimal estimate. Other filters, such as the discrete Bayes filter or the particle filter which we cover in a later chapter, model the probability in a more ad-hoc, non-optimal manner. The power of our technique comes from treating lack of information as a random variable, describing that random variable as a probability distribution, and then using Bayes Theorem to solve the statistical inference problem.

## 7.8 Converting Kalman Filter to a g-h Filter

I've stated that the Kalman filter is a form of the g-h filter. It just takes some algebra to prove it. It's more straightforward to do with the one dimensional case, so I will do that. Recall

$$\mu_x = \frac{\sigma_1^2\mu_2 + \sigma_2^2\mu_1}{\sigma_1^2 + \sigma_2^2}$$

which I will make more friendly for our eyes as:

$$\mu_x = \frac{ya + xb}{a+b}$$

We can easily put this into the g-h form with the following algebra

$$\begin{aligned}
\mu_x &= (x-x) + \frac{ya + xb}{a+b} \\
\mu_x &= x - \frac{a+b}{a+b}x + \frac{ya + xb}{a+b} \\
\mu_x &= x + \frac{-x(a+b) + xb + ya}{a+b} \\
\mu_x &= x + \frac{-xa + ya}{a+b} \\
\mu_x &= x + \frac{a}{a+b}(y-x)
\end{aligned}$$

We are almost done, but recall that the variance of estimate is given by

$$\begin{aligned}
\sigma_x^2 &= \frac{1}{\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}} \\
&= \frac{1}{\frac{1}{a} + \frac{1}{b}}
\end{aligned}$$

We can incorporate that term into our equation above by observing that

$$\begin{aligned}
\frac{a}{a+b} &= \frac{a/a}{(a+b)/a} = \frac{1}{(a+b)/a} \\
&= \frac{1}{1 + \frac{b}{a}} = \frac{1}{\frac{b}{b} + \frac{b}{a}} \\
&= \frac{1}{b}\frac{1}{\frac{1}{b} + \frac{1}{a}} \\
&= \frac{\sigma_{x'}^2}{b}
\end{aligned}$$

We can tie all of this together with

$$\begin{aligned}
\mu_x &= x + \frac{a}{a+b}(y-x) \\
&= x + \frac{\sigma_{x'}^2}{b}(y-x) \\
&= x + g_n(y-x)
\end{aligned}$$

where

$$g_n = \frac{\sigma_x^2}{\sigma_y^2}$$

The end result is multiplying the residual of the two measurements by a constant and adding to our previous value, which is the $g$ equation for the g-h filter. $g$ is the variance of the new estimate divided by the variance of the measurement. Of course in this case $g$ is not a constant as it varies with each time step as the variance changes. We can also derive the formula for $h$ in the same way. It is not a particularly illuminating derivation and I will skip it. The end result is

$$h_n = \frac{COV(x, \dot{x})}{\sigma_y^2}$$

The takeaway point is that $g$ and $h$ are specified fully by the variance and covariances of the measurement and predictions at time $n$. In other words, we are picking a point between the measurement and prediction by a scale factor determined by the quality of each of those two inputs.

## 7.9 References


- [1] C.B. Molwer and C.F. Van Loan "Nineteen Dubious Ways to Compute the Exponential of a Matrix, Twenty-Five Years Later,", *SIAM Review 45, 3-49.* 2003.
- [2] C.F. van Loan, "Computing Integrals Involving the Matrix Exponential," IEEE *Transactions Automatic Control*, June 1978.
- [3] Calvetti, D and Somersalo E, "Introduction to Bayesian Scientific Computing: Ten Lectures on Subjective Computing,", *Springer*, 2007.
- [4] Brown, R. G. and Hwang, P. Y.C., "Introduction to Random Signals and Applied Kalman Filtering", *Wiley and Sons*, Fourth Edition, p.143-147, 2012.

# Chapter 8

# Designing Kalman Filters

```
In [1]: from __future__ import division, print_function
        %matplotlib inline
```

```
In [2]: #format the book
        import book_format
        book_format.set_style()
```

```
Out[2]:
```

## 8.1 Introduction

In the last chapter we worked with 'textbook' problems. These are problems that are easy to state, program in a few lines of code, and teach. Real world problems are rarely this simple. In this chapter we will work with more realistic examples, and learn how to evaluate filter performance.

We will begin by tracking a robot in a 2D space, such as a field or warehouse. We will start with a simple noisy sensor that outputs noisy $(x, y)$ coordinates which we will need to filter to generate a 2D track. Once we have mastered this concept, we will extend the problem significantly with more sensors and then adding control inputs.

We will then move to a nonlinear problem. The world is nonlinear, but the Kalman filter is linear. Sometimes you can get away with using it for mildly nonlinear problems, sometimes you can't. I'll show you examples of both. This will set the stage for the remainder of the book, where we learn techniques for nonlinear problems.

## 8.2 Tracking a Robot

This first attempt at tracking a robot will closely resemble the 1D dog tracking problem of previous chapters. Instead of a sensor that outputs position in a hallway, we now have a sensor that supplies a noisy measurement of position in a 2D space. At each time $t$ it will provide an $(x, y)$ coordinate pair of the noisy measurement of the sensor's position in the field.

Implementation of code to interact with real sensors is beyond the scope of this book, so as before we will program simple simulations of the sensors. We will develop several of these sensors as we go, each with more complications, so as I program them I will just append a number to the function name.

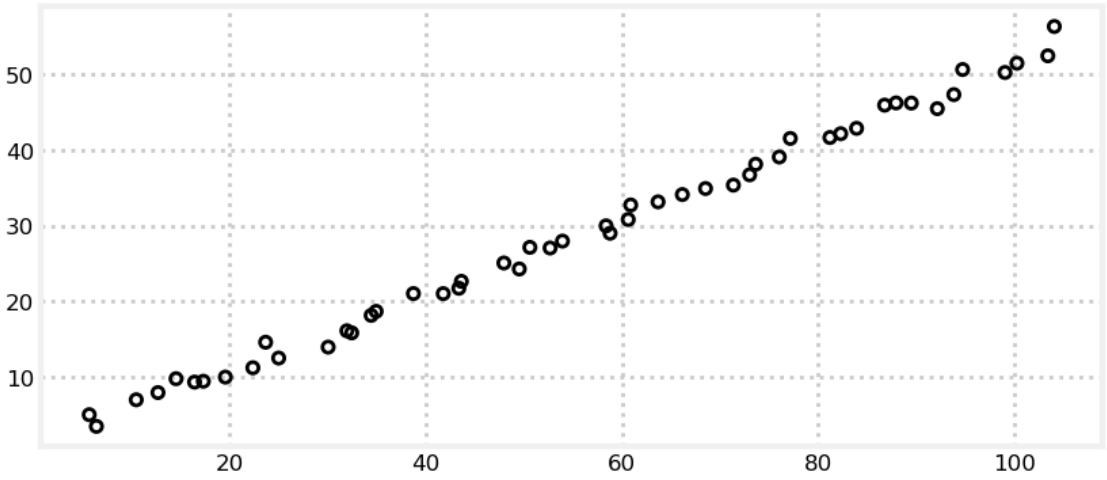
So let's start with a very simple sensor, one that simulates tracking an object traveling in a straight line. It is initialized with the initial position, velocity, and noise standard deviation. Each call to `read()` updates the position by one time step and returns the new measurement.

```
In [3]: from numpy.random import randn

        class PosSensor(object):
```

```
        def __init__(self, pos=(0, 0), vel=(0, 0), noise_std=1.):
            self.vel = vel
            self.noise_std = noise_std
            self.pos = [pos[0], pos[1]]

        def read(self):
            self.pos[0] += self.vel[0]
            self.pos[1] += self.vel[1]

            return [self.pos[0] + randn() * self.noise_std,
                    self.pos[1] + randn() * self.noise_std]
```

A quick test to verify that it works as we expect.

```
In [4]: import matplotlib.pyplot as plt
        import numpy as np
        from kf_book.book_plots import plot_measurements

        pos, vel = (4, 3), (2, 1)
        sensor = PosSensor(pos, vel, noise_std=1)
        ps = np.array([sensor.read() for _ in range(50)])
        plot_measurements(ps[:, 0], ps[:, 1]);
```




That looks correct. The slope is 1/2, as we would expect with a velocity of (2, 1), and the data seems to start at near (6, 4). It doesn't look realistic. This is still a 'textbook' representation. As we continue we will add complications that adds real world behavior.

### 8.2.1 Choose the State Variables

As always, the first step is to choose our state variables. We are tracking in two dimensions and have a sensor that gives us a reading in each of those two dimensions, so we know that we have the two *observed variables* $x$ and $y$. If we created our Kalman filter using only those two variables the performance would not be very good because we would be ignoring the information velocity can provide to us. We will want to incorporate velocity into our equations as well. I will represent this as

$$\mathbf{x} = \begin{bmatrix} x & \dot{x} & y & \dot{y} \end{bmatrix}^\mathsf{T}$$

There is nothing special about this organization. I could have used $\begin{bmatrix}x & y & \dot{x} & \dot{y}\end{bmatrix}^\mathsf{T}$ or something less logical. I just need to be consistent in the rest of the matrices. I like keeping positions and velocities next to each other because it keeps the covariances between positions and velocities in the same sub block of the covariance matrix. In my formulation `P[1,0]` contains the covariance of of $x$ and $\dot{x}$. In the alternative formulation that covariance is at `P[2, 0]`. This gets worse as the number of dimension increases.

Let's pause and address how you identify the hidden variables. This example is somewhat obvious because we've already worked through the 1D case, but other problems won't be obvious There is no easy answer to this question. The first thing to ask yourself is what is the interpretation of the first and second derivatives of the data from the sensors. We do that because obtaining the first and second derivatives is mathematically trivial if you are reading from the sensors using a fixed time step. The first derivative is just the difference between two successive readings. In our tracking case the first derivative has an obvious physical interpretation: the difference between two successive positions is velocity.

Beyond this you can start looking at how you might combine the data from two or more different sensors to produce more information. This opens up the field of *sensor fusion*, and we will be covering examples of this in later sections. For now, recognize that choosing the appropriate state variables is paramount to getting the best possible performance from your filter. Once you have chosen hidden variables, you must run many tests to ensure that you are generating real results for them. The Kalman filter runs whatever model you give it; if your model cannot generate good information for the hidden variables the Kalman filter output will be nonsensical.

### 8.2.2 Design State Transition Function

Our next step is to design the state transition function. Recall that the state transition function is implemented as a matrix $\mathbf{F}$ that we multiply with the previous state of our system to get the next state, like so.

$$\mathbf{x} = \mathbf{F}\mathbf{x}$$

I will not belabor this as it is very similar to the 1-D case we did in the previous chapter. The state transition equations are

$$\begin{aligned}
x &= 1x + \Delta t\dot{x} + 0y + 0\dot{y} \\
v_x &= 0x + 1\dot{x} + 0y + 0\dot{y} \\
y &= 0x + 0\dot{x} + 1y + \Delta t\dot{y} \\
v_y &= 0x + 0\dot{x} + 0y + 1\dot{y}
\end{aligned}$$

Laying it out that way shows us both the values and row-column organization required for $\mathbf{F}$. We convert this to matrix-vector form:

$$\begin{bmatrix}x \\ \dot{x} \\ y \\ \dot{y}\end{bmatrix} = \begin{bmatrix}1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x \\ \dot{x} \\ y \\ \dot{y}\end{bmatrix}$$

So, let's do this in Python. It is very simple; the only thing new here is setting `dim_z` to 2. We will see why it is set to 2 in step 4.

```
In [5]: from filterpy.kalman import KalmanFilter

        tracker = KalmanFilter(dim_x=4, dim_z=2)
        dt = 1.   # time step 1 second

        tracker.F = np.array([[1, dt, 0,  0],
                              [0,  1, 0,  0],
                              [0,  0, 1, dt],
                              [0,  0, 0,  1]])
```

### 8.2.3 Design the Process Noise Matrix

FilterPy can compute the **Q** matrix for us. For simplicity I will assume the noise is a discrete time Wiener process - that it is constant for each time period. This assumption allows me to use a variance to specify how much I think the model changes between steps. Revisit the Kalman Filter Math chapter if this is not clear.

```
In [6]: from scipy.linalg import block_diag
        from filterpy.common import Q_discrete_white_noise

        q = Q_discrete_white_noise(dim=2, dt=dt, var=0.001)
        tracker.Q = block_diag(q, q)
        print(tracker.Q)
```

```
[[0.    0.001 0.    0.   ]
 [0.001 0.001 0.    0.   ]
 [0.    0.    0.    0.001]
 [0.    0.    0.001 0.001]]
```

Here I assume the noise in x and y are independent, so the covariances between any x and y variable should be zero. This allows me to compute **Q** for one dimension, and then use `block_diag` to copy it for the x and y axis.

### 8.2.4 Design the Control Function

We haven't yet added controls to our robot, so there is nothing to be done for this step. The `KalmanFilter` class initializes `B` to zero under the assumption that there is no control input, so there is no code to write. If you like, you can be explicit and set `tracker.B` to 0, but as you can see it already has that value.

```
In [7]: tracker.B
```

### 8.2.5 Design the Measurement Function

The measurement function **H** defines how we go from the state variables to the measurements using the equation $\mathbf{z} = \mathbf{Hx}$. In this case we have measurements for (x,y), so we will design $\mathbf{z}$ as $\begin{bmatrix}x & y\end{bmatrix}^\mathsf{T}$ which is dimension 2x1. Our state variable is size 4x1. We can deduce the required size for **H** by recalling that multiplying a matrix of size MxN by NxP yields a matrix of size MxP. Thus,

$$(2 \times 1) = (a \times b)(4 \times 1) = (2 \times 4)(4 \times 1)$$

So, **H** is 2x4.

Filling in the values for **H** is easy because the measurement is the position of the robot, which is the $x$ and $y$ variables of the state $\mathbf{x}$. Let's make this slightly more interesting by deciding we want to change units. The measurements are returned in feet, and that we desire to work in meters. **H** changes from state to measurement, so the conversion is $\mathsf{feet} = \mathsf{meters}/0.3048$. This yields

$$\mathbf{H} = \begin{bmatrix} \frac{1}{0.3048} & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{0.3048} & 0 \end{bmatrix}$$

which corresponds to these linear equations

$$\begin{aligned} z_x &= (\frac{x}{0.3048}) + (0 * v_x) + (0 * y) + (0 * v_y) = \frac{x}{0.3048} \\ z_y &= (0 * x) + (0 * v_x) + (\frac{y}{0.3048}) + (0 * v_y) = \frac{y}{0.3048} \end{aligned}$$

This is a simple problem, and we could have found the equations directly without going through the dimensional analysis that I did above. But it is useful to remember that the equations of the Kalman filter

imply a specific dimensionality for all of the matrices, and when I start to get lost as to how to design something it is useful to look at the matrix dimensions.

Here is my implementation:

```
In [8]: tracker.H = np.array([[1/0.3048, 0, 0,        0],
                              [0,        0, 1/0.3048, 0]])
```

### 8.2.6 Design the Measurement Noise Matrix

We assume that the $x$ and $y$ variables are independent white Gaussian processes. That is, the noise in x is not in any way dependent on the noise in y, and the noise is normally distributed about the mean 0. For now let's set the variance for $x$ and $y$ to be 5 meters$^2$. They are independent, so there is no covariance, and our off diagonals will be 0. This gives us:

$$\mathbf{R} = \begin{bmatrix} \sigma_x^2 & \sigma_y\sigma_x \\ \sigma_x\sigma_y & \sigma_y^2 \end{bmatrix} = \begin{bmatrix} 5 & 0 \\ 0 & 5 \end{bmatrix}$$

It is a 2×2 matrix because we have 2 sensor inputs, and covariance matrices are always of size $n{\times}n$ for $n$ variables. In Python we write:

```
In [9]: tracker.R = np.array([[5., 0],
                              [0, 5]])
        tracker.R

Out[9]: array([[5., 0.],
               [0., 5.]])
```

### 8.2.7 Initial Conditions

For our simple problem we will set the initial position at (0,0) with a velocity of (0,0). Since that is a pure guess, we will set the covariance matrix $\mathbf{P}$ to a large value.

$$\mathbf{x} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \mathbf{P} = \begin{bmatrix} 500 & 0 & 0 & 0 \\ 0 & 500 & 0 & 0 \\ 0 & 0 & 500 & 0 \\ 0 & 0 & 0 & 500 \end{bmatrix}$$

The Python implementation is

```
In [10]: tracker.x = np.array([[0, 0, 0, 0]]).T
         tracker.P = np.eye(4) * 500.
```

### 8.2.8 Implement the Filter

Design is complete, now we just have to write the code to run the filter and output the data in the format of our choice. We will run the code for 30 iterations.

```
In [11]: from filterpy.stats import plot_covariance_ellipse
         from kf_book.book_plots import plot_filter

         R_std = 0.35
         Q_std = 0.04

         def tracker1():
             tracker = KalmanFilter(dim_x=4, dim_z=2)
             dt = 1.0   # time step
```

```
    tracker.F = np.array([[1, dt, 0,  0],
                          [0,  1, 0,  0],
                          [0,  0, 1, dt],
                          [0,  0, 0,  1]])
    tracker.u = 0.
    tracker.H = np.array([[1/0.3048, 0, 0, 0],
                          [0, 0, 1/0.3048, 0]])

    tracker.R = np.eye(2) * R_std**2
    q = Q_discrete_white_noise(dim=2, dt=dt, var=Q_std**2)
    tracker.Q = block_diag(q, q)
    tracker.x = np.array([[0, 0, 0, 0]]).T
    tracker.P = np.eye(4) * 500.
    return tracker

# simulate robot movement
N = 30
sensor = PosSensor((0, 0), (2, .2), noise_std=R_std)

zs = np.array([sensor.read() for _ in range(N)])

# run filter
robot_tracker = tracker1()
mu, cov, _, _ = robot_tracker.batch_filter(zs)

for x, P in zip(mu, cov):
    # covariance of x and y
    cov = np.array([[P[0, 0], P[2, 0]],
                    [P[0, 2], P[2, 2]]])
    mean = (x[0, 0], x[2, 0])
    plot_covariance_ellipse(mean, cov=cov, fc='g', std=3, alpha=0.5)

#plot results
zs *= .3048 # convert to meters
plot_filter(mu[:, 0], mu[:, 2])
plot_measurements(zs[:, 0], zs[:, 1])
plt.legend(loc=2)
plt.xlim(0, 20);
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-11-84e991e97aff> in <module>
 32 # run filter
 33 robot_tracker = tracker1()
---> 34 mu, cov, _, _ = robot_tracker.batch_filter(zs)
 35
 36 for x, P in zip(mu, cov):


C:\pcloud\filterpy\filterpy\kalman\kalman_filter.py in batch_filter(self, zs, Fs, Qs, Hs, Rs, Bs
918
```

```
    919                 self.update(z, R=R, H=H)
--> 920                 means[i, :] = self.x
    921                 covariances[i, :, :] = self.P
    922

ValueError: could not broadcast input array from shape (4,2) into shape (4,1)
```

I encourage you to play with this, setting $\mathbf{Q}$ and $\mathbf{R}$ to various values. However, we did a fair amount of that sort of thing in the last chapters, and we have a lot of material to cover, so I will move on to more complicated cases where we will also have a chance to experience changing these values.

I plotted the $3\sigma$ the covariance ellipse for $x$ and $y$ in green. Can you explain their shape? Perhaps you were expecting a tilted ellipse, as in the last chapters. If so, recall that in those chapters we were not plotting $x$ against $y$, but $x$ against $\dot{x}$. $x$ is correlated to $\dot{x}$, but $x$ is not correlated or dependent on $y$. Therefore our ellipses are not tilted. Furthermore, the noise for both $x$ and $y$ are modeled to have the same noise standard deviation. If we were to set R to, for example,

$$\mathbf{R} = \begin{bmatrix} 1 & 0 \\ 0 & .5 \end{bmatrix}$$

we would be telling the Kalman filter that there is more noise in $x$ than $y$, and our ellipses would be longer than they are tall.

The final value for $\mathbf{P}$ tells us everything we need to know about the correlation between the state variables. If we look at the diagonal alone we see the variance for each variable. In other words $\mathbf{P}_{0,0}$ is the variance for x, $\mathbf{P}_{1,1}$ is the variance for $\dot{x}$, $\mathbf{P}_{2,2}$ is the variance for y, and $\mathbf{P}_{3,3}$ is the variance for $\dot{y}$. We can extract the diagonal of a matrix using `numpy.diag()`.

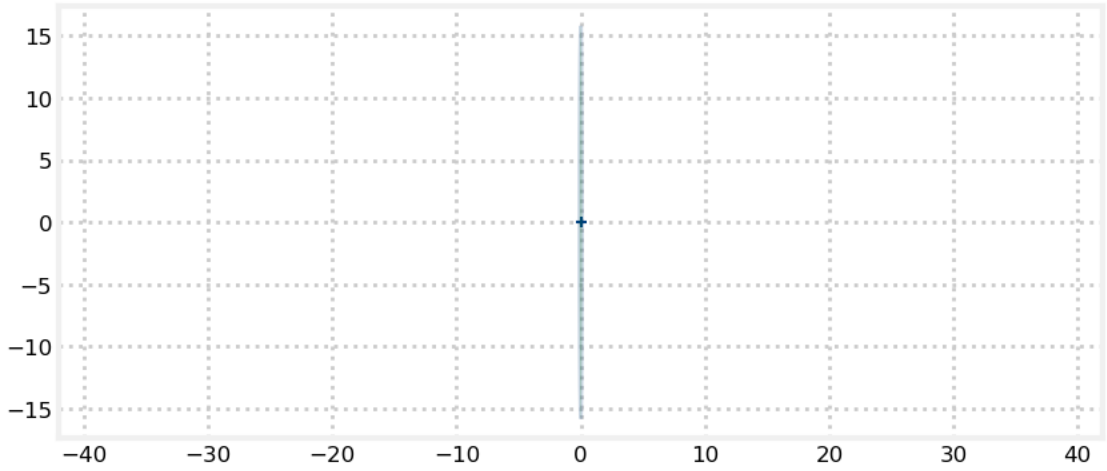
```
In [12]: print(np.diag(robot_tracker.P))
```

```
[  0.011 250.004   0.011 250.004]
```

The covariance matrix contains four 2×2 matrices that you should be able to easily pick out. This is due to the correlation of $x$ to $\dot{x}$, and of $y$ to $\dot{y}$. The upper left hand side shows the covariance of $x$ to $\dot{x}$.

```
In [13]: c = robot_tracker.P[0:2, 0:2]
         print(c)
         plot_covariance_ellipse((0, 0), cov=c, fc='g', alpha=0.2)
```

```
[[  0.011   0.006]
 [  0.006 250.004]]
```



The covariance contains the data for $x$ and $\dot{x}$ in the upper left because of how it is organized. Recall that entries $\mathbf{P}_{i,j}$ and $\mathbf{P}_{j,i}$ contain $\sigma_i\sigma_j$.

Finally, let's look at the lower left side of $\mathbf{P}$, which is all 0s. Why 0s? Consider $\mathbf{P}_{3,0}$. That stores the term $\sigma_3\sigma_0$, which is the covariance between $\dot{y}$ and $x$. These are independent, so the term will be 0. The rest of the terms are for similarly independent variables.

```
In [14]: robot_tracker.P[2:4, 0:2]
```

```
Out[14]: array([[0., 0.],
                [0., 0.]])
```

## 8.3 Filter Order

We have only studied tracking position and velocity. It has worked well, but only because I have been selecting problems for which this is an appropriate choice. You now have enough experience with the Kalman filter to consider this in more general terms.

What do I mean by order? In the context of these system models it is the number of derivatives required to accurately model a system. Consider a system that does not change, such as the height of a building. There is no change, so there is no need for a derivative, and the order of the system is zero. We could express this in an equation as $x = 312.5$.

A first order system has a first derivative. For example, change of position is velocity, and we can write this as

$$v = \frac{dx}{dt}$$

which we integrate into the Newtonian equation

$$x = vt + x_0.$$

This is also called a *constant velocity* model, because of the assumption of a constant velocity.

A second order system has a second derivative. The second derivative of position is acceleration, with the equation

$$a = \frac{d^2x}{dt^2}$$

which we integrate into

$$x = \frac{1}{2}at^2 + v_0t + x_0.$$

This is also known as a *constant acceleration* model.

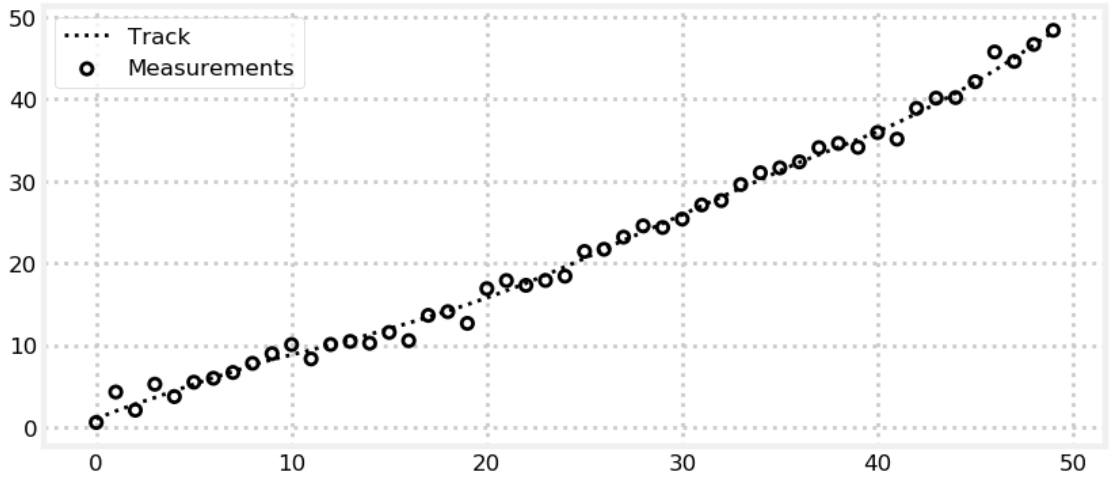
Another, equivalent way of looking at this is to consider the order of the polynomial. The constant acceleration model has a second derivative, so it is second order. Likewise, the polynomial $x = \frac{1}{2}at^2+v_0t+x_0$ is second order.

When we design the state variables and process model we must choose the order of the system we want to model. Let's say we are tracking something with a constant velocity. No real world process is perfect, and so there will be slight variations in the velocity over short time period. You might reason that the best approach is to use a second order filter, allowing the acceleration term to deal with the slight variations in velocity.

In practice that doesn't work well. To thoroughly understand this issue let's see the effects of using a process model that does not match the order of the system being filtered.

First we need a system to filter. I'll write a class to simulate an object with constant velocity. Essentially no physical system has a truly constant velocity, so on each update we alter the velocity by a small amount. I also write a sensor to simulate Gaussian noise in a sensor. The code is below, and I plot an example run to verify that it is working correctly.

```
In [15]: from kf_book.book_plots import plot_track

         class ConstantVelocityObject(object):
             def __init__(self, x0=0, vel=1., noise_scale=0.06):
                 self.x = x0
                 self.vel = vel
                 self.noise_scale = noise_scale

             def update(self):
                 self.vel += randn() * self.noise_scale
                 self.x += self.vel
                 return (self.x, self.vel)

         def sense(x, noise_scale=1.):
             return x[0] + randn()*noise_scale

         np.random.seed(124)
         obj = ConstantVelocityObject()

         xs, zs = [], []
         for i in range(50):
             x = obj.update()
             z = sense(x)
             xs.append(x)
             zs.append(z)

         xs = np.asarray(xs)

         plot_track(xs[:, 0])
         plot_measurements(range(len(zs)), zs)
         plt.legend(loc='best');
```




I am satisfied with this plot. The track is not perfectly straight due to the noise that we added to the system - this could be the track of a person walking down the street, or perhaps of an aircraft being buffeted by variable winds. There is no intentional acceleration here, so we call it a constant velocity system. Again, you may be asking yourself that since there is in fact a tiny bit of acceleration going on why would we not

use a second order Kalman filter to account for those changes? Let's find out.

How does one design a zero order, first order, or second order Kalman filter? We have been doing it all along, but just not using those terms. It might be slightly tedious, but I will elaborate fully on each - if the concept is clear to you feel free to skim a bit.

### 8.3.1 Zero Order Kalman Filter

A zero order Kalman filter is just a filter that tracks with no derivatives. We are tracking position, so that means we only have a state variable for position (no velocity or acceleration), and the state transition function also only accounts for position. Using the matrix formulation we would say that the state variable is

$$\mathbf{x} = \begin{bmatrix} x \end{bmatrix}$$

The state transition function is very simple. There is no change in position, so we need to model $x = x$; in other words, $x$ at time t+1 is the same as it was at time t. In matrix form, our state transition function is

$$\mathbf{F} = \begin{bmatrix} 1 \end{bmatrix}$$

The measurement function is very easy. Recall that we need to define how to convert the state variable $\mathbf{x}$ into a measurement. We will assume that our measurements are positions. The state variable only contains a position, so we get

$$\mathbf{H} = \begin{bmatrix} 1 \end{bmatrix}$$

Let's write a function that constructs and returns a zero order Kalman filter.

```
In [16]: def ZeroOrderKF(R, Q, P=20):
             """ Create zero order Kalman filter.
             Specify R and Q as floats."""

             kf = KalmanFilter(dim_x=1, dim_z=1)
             kf.x = np.array([0.])
             kf.R *= R
             kf.Q *= Q
             kf.P *= P
             kf.F = np.eye(1)
             kf.H = np.eye(1)
             return kf
```

### 8.3.2 First Order Kalman Filter

A first order Kalman filter tracks a first order system, such as position and velocity. We already did this for the dog tracking problem above, so this should be very clear. But let's do it again.

A first order system has position and velocity, so the state variable needs both of these. The matrix formulation could be

$$\mathbf{x} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

So now we have to design our state transition. The Newtonian equations for a time step are:

$$x_t = x_{t-1} + v\Delta t$$
$$v_t = v_{t-1}$$

Recall that we need to convert this into the linear equation

$$\begin{bmatrix}x\\ \dot{x}\end{bmatrix} = \mathbf{F}\begin{bmatrix}x\\ \dot{x}\end{bmatrix}$$

Setting

$$\mathbf{F} = \begin{bmatrix}1 & \Delta t\\ 0 & 1\end{bmatrix}$$

gives us the equations above.

Finally, we design the measurement function. The measurement function needs to implement

$$\mathbf{z} = \mathbf{Hx}$$

Our sensor still only reads position, so it should take the position from the state, and 0 out the velocity and acceleration, like so:

$$\mathbf{H} = \begin{bmatrix}1 & 0\end{bmatrix}$$

This function constructs and returns a first order Kalman filter.

```
In [17]: def FirstOrderKF(R, Q, dt):
             """ Create first order Kalman filter.
             Specify R and Q as floats."""

             kf = KalmanFilter(dim_x=2, dim_z=1)
             kf.x = np.zeros(2)
             kf.P *= np.array([[100, 0], [0, 1]])
             kf.R *= R
             kf.Q = Q_discrete_white_noise(2, dt, Q)
             kf.F = np.array([[1., dt],
                              [0., 1]])
             kf.H = np.array([[1., 0]])
             return kf
```

### 8.3.3 Second Order Kalman Filter

A second order Kalman filter tracks a second order system, such as position, velocity and acceleration. The state variable will be

$$\mathbf{x} = \begin{bmatrix}x\\ \dot{x}\\ \ddot{x}\end{bmatrix}$$

So now we have to design our state transition. The Newtonian equations for a time step are:

$$\begin{aligned}x_t &= x_{t-1} + v_{t-1}\Delta t + 0.5a_{t-1}\Delta t^2\\ v_t &= v_{t-1} + a_{t-1}\Delta t\\ a_t &= a_{t-1}\end{aligned}$$

Recall that we need to convert this into the linear equation

$$\begin{bmatrix}x\\ \dot{x}\\ \ddot{x}\end{bmatrix} = \mathbf{F}\begin{bmatrix}x\\ \dot{x}\\ \ddot{x}\end{bmatrix}$$

Setting

$$\mathbf{F} = \begin{bmatrix} 1 & \Delta t & .5\Delta t^2 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix}$$

gives us the equations above.

Finally, we design the measurement function. The measurement function needs to implement

$$z = \mathbf{Hx}$$

Our sensor still only reads position, so it should take the position from the state, and 0 out the velocity, like so:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

This function constructs and returns a second order Kalman filter.

```
In [18]: def SecondOrderKF(R_std, Q, dt, P=100):
             """ Create second order Kalman filter.
             Specify R and Q as floats."""

             kf = KalmanFilter(dim_x=3, dim_z=1)
             kf.x = np.zeros(3)
             kf.P[0, 0] = P
             kf.P[1, 1] = 1
             kf.P[2, 2] = 1
             kf.R *= R_std**2
             kf.Q = Q_discrete_white_noise(3, dt, Q)
             kf.F = np.array([[1., dt, .5*dt*dt],
                              [0., 1.,       dt],
                              [0., 0.,       1.]])
             kf.H = np.array([[1., 0., 0.]])
             return kf
```

## 8.4 Evaluating Filter Order

Now we can run each Kalman filter against the simulation and evaluate the results.

How do we evaluate the results? We can do this qualitatively by plotting the track and the Kalman filter output and eyeballing the results. However, a rigorous approach uses mathematics. Recall that the system covariance matrix $\mathbf{P}$ contains the computed variance and covariances for each of the state variables. The diagonal contains the variance. Remember that roughly 99% of all measurements fall within $3\sigma$ if the noise is Gaussian. If this is not clear please review the Gaussian chapter before continuing, as this is an important point.

So we can evaluate the filter by looking at the residuals between the estimated state and actual state and comparing them to the standard deviations which we derive from $\mathbf{P}$. If the filter is performing correctly 99% of the residuals will fall within $3\sigma$. This is true for all the state variables, not just for the position.

I must mention that this is only true for simulated systems. Real sensors are not perfectly Gaussian, and you may need to expand your criteria to, say, $5\sigma$ with real sensor data.

So let's run the first order Kalman filter against our first order system and access its performance. You can probably guess that it will do well, but let's look at it using the standard deviations.

First, let's write a routine to generate the noisy measurements for us.

```
In [19]: def simulate_system(Q, count):
             obj = ConstantVelocityObject(x0=.0, vel=0.5, noise_scale=Q)
             xs, zs = [], []
             for i in range(count):
```

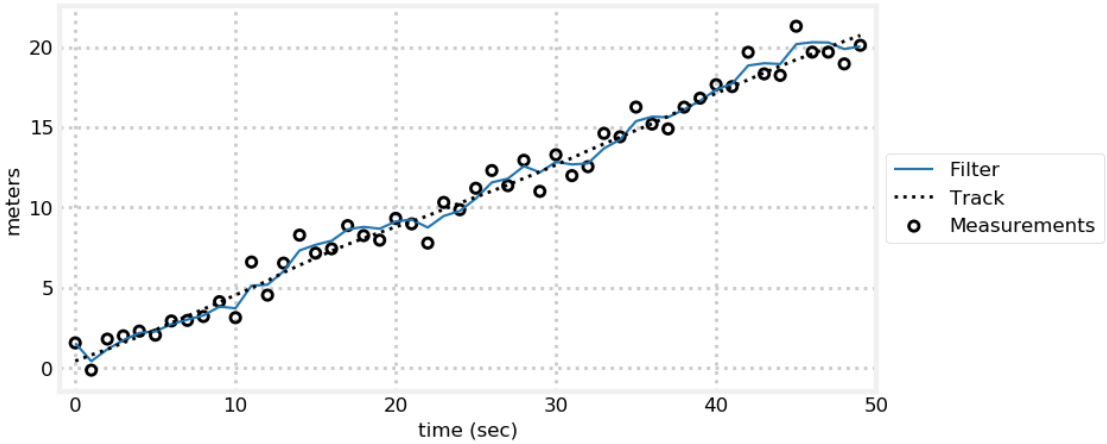
```
            x = obj.update()
            z = sense(x)
            xs.append(x)
            zs.append(z)
        return np.array(xs), np.array(zs)
```

And now a routine to perform the filtering and save the output in a `Saver` object.

```
In [20]: from filterpy.common import Saver

         def filter_data(kf, zs):
             s = Saver(kf)
             kf.batch_filter(zs, saver=s)
             s.to_array()
             return s
```

Now we are prepared to run the filter and look at the results.

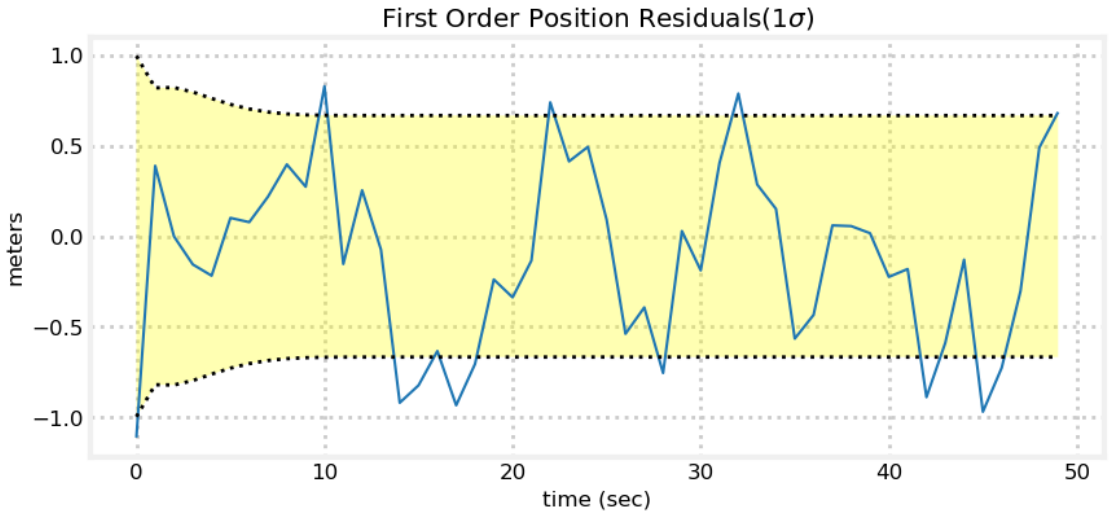
```
In [21]: from kf_book.book_plots import plot_kf_output

         R, Q = 1, 0.03
         xs, zs = simulate_system(Q=Q, count=50)

         kf = FirstOrderKF(R, Q, dt=1)
         data1 = filter_data(kf, zs)

         plot_kf_output(xs, data1.x, data1.z)
```




It looks like the filter is performing well, but it is hard to tell exactly how well. Let's look at the residuals and see if they help. We'll do this a lot, so I'll write a function to plot them.

```
In [22]: from kf_book.book_plots import plot_residual_limits, set_labels

         def plot_residuals(xs, data, col, title, y_label, stds=1):
             res = xs - data.x[:, col]
             plt.plot(res)
             plot_residual_limits(data.P[:, col, col], stds)
             set_labels(title, 'time (sec)', y_label)
```

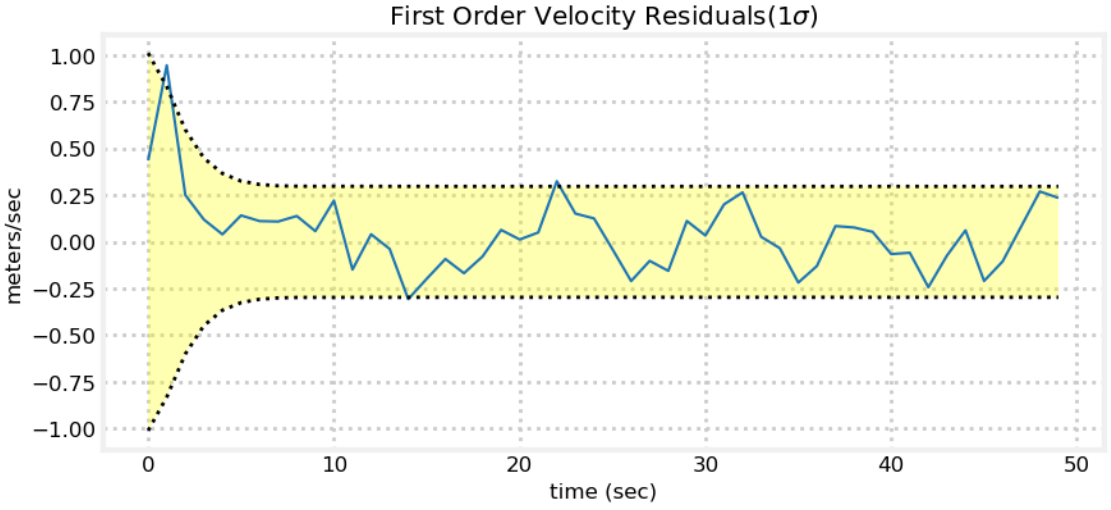
```
In [23]: plot_residuals(xs[:, 0], data1, 0,
                        title='First Order Position Residuals(1$\sigma$)',
                        y_label='meters')
```




How do we interpret this plot? The residual is drawn as the jagged line - the difference between the measurement and the predicted position. If there was no measurement noise and the Kalman filter prediction was always perfect the residual would always be zero. So the ideal output would be a horizontal line at 0. We can see that the residual is centered around 0, so this gives us confidence that the noise is Gaussian (because the errors fall equally above and below 0). The yellow area between dotted lines show the theoretical performance of the filter for 1 standard deviations. In other words, approximately 68% of the errors should fall within the dotted lines. The residual falls within this range, so we see that the filter is performing well, and that it is not diverging.

Let's look at the residuals for velocity.

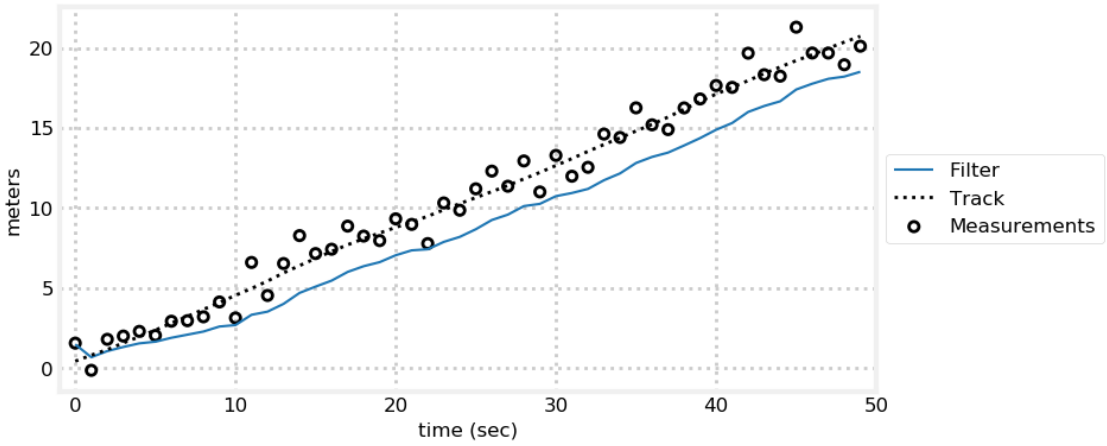
```
In [24]: plot_residuals(xs[:, 1], data1, 1,
                        title='First Order Velocity Residuals(1$\sigma$)',
                        y_label='meters/sec')
```



Again, as expected, the residual falls within the theoretical performance of the filter, so we feel confident that the filter is well designed for this system.

Now let's do the same thing using the zero order Kalman filter. All of the code and math is largely the same, so let's just look at the results without discussing the implementation much.

```
In [25]: kf0 = ZeroOrderKF(R, Q)
         data0 = filter_data(kf0, zs)
         plot_kf_output(xs, data0.x, data0.z)
```

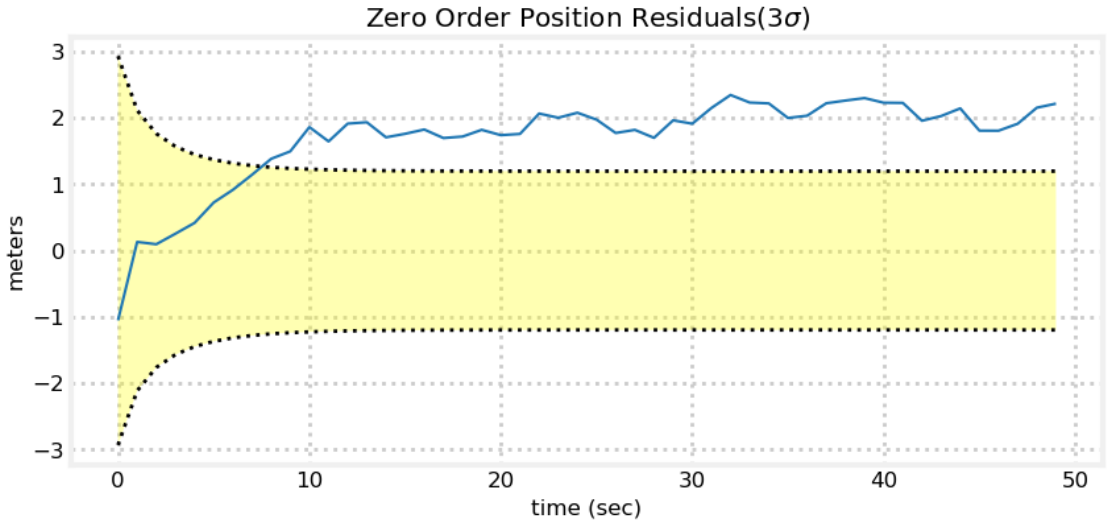



As we would expect, the filter has problems. Think back to the g-h filter, where we incorporated acceleration into the system. The g-h filter always lagged the input because there were not enough terms to allow the filter to adjust quickly enough to the changes in velocity. On every `predict()` step the Kalman filter assumes that there is no change in position - if the current position is 4.3 it will predict that the position at the next time period is 4.3. Of course, the actual position is closer to 5.3. The measurement, with noise, might be 5.4, so the filter chooses an estimate part way between 4.3 and 5.4, causing it to lag the actual value of 5.3 by a significant amount. This same thing happens in the next step, the next one, and so on. The filter never catches up.

This raises a very important point. The assumption of 'constant' is an assumption of constant-ness between discrete samples only. The filter's output can still change over time.

Now let's look at the residuals. We are not tracking velocity, so we can only look at the residual for position.

```
In [26]: plot_residuals(xs[:, 0], data0, 0,
                        title='Zero Order Position Residuals(3$\sigma$)',
                        y_label='meters',
                        stds=3)
```

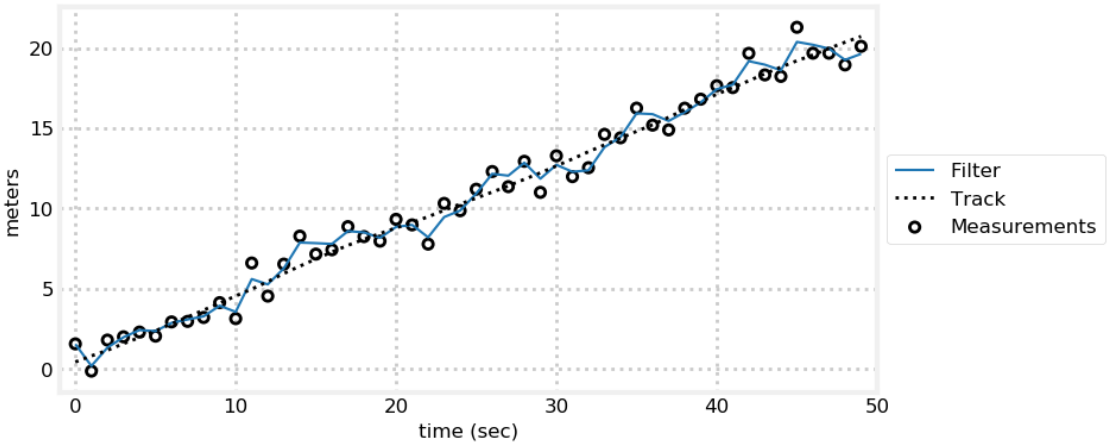



We can see that the filter diverges almost immediately. After a few seconds the residual exceeds the bounds of three standard deviations. It is important to understand that the covariance matrix $\mathbf{P}$ is only reporting the *theoretical* performance of the filter *assuming* all of the inputs are correct. In other words, this Kalman filter is diverging, but $\mathbf{P}$ implies that the Kalman filter's estimates are getting better and better with time because the variance is getting smaller. The filter has no way to know that you are lying to it about the system. This is sometimes referred to a *smug* filter - it is overconfident in its performance.

In this system the divergence is immediate and striking. In many systems it will only be gradual, and/or slight. It is important to look at charts like these for your systems to ensure that the performance of the filter is within the bounds of its theoretical performance.

Now let's try a second order system. This might strike you as a good thing to do. After all, we know there is a bit of noise in the movement of the simulated object, which implies there is some acceleration. Why not model the acceleration with a second order model? If there is no acceleration, the acceleration should just be estimated to be 0, right?. But is that what happens? Think about it before going on.

```
In [27]: kf2 = SecondOrderKF(R, Q, dt=1)
         data2 = filter_data(kf2, zs)
         plot_kf_output(xs, data2.x, data2.z)
```

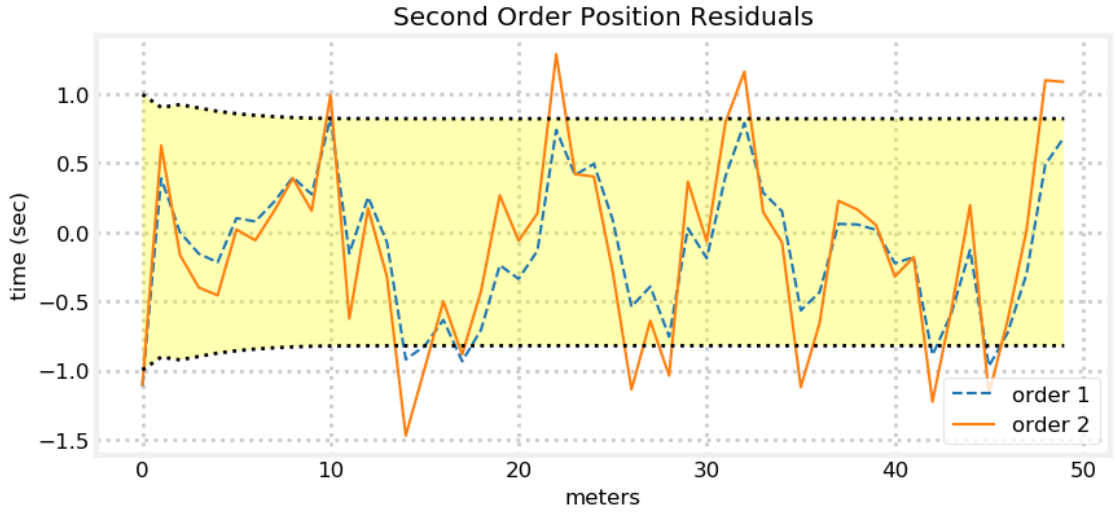


Did this perform as you expected?

We can see that second order filter performs poorly compared to the first order filter. Why? This filter models acceleration, and so the large changes in the measurement gets interpreted as acceleration instead of noise. Thus the filter closely tracks the noise. Not only that, but it *overshoots* the noise in places if the noise is consistently above or below the track because the filter incorrectly assumes an acceleration that does not exist, and so it's prediction goes further and further away from the track on each measurement. This is not a good state of affairs.

Still, the track doesn't look *horrible*. Let's see the story that the residuals tell. I will add a wrinkle here. The residuals for the second order system do not look terrible in that they do not diverge or exceed three standard deviations. However, it is very telling to look at the residuals for the first order vs the second order filter, so I have plotted both on the same graph.

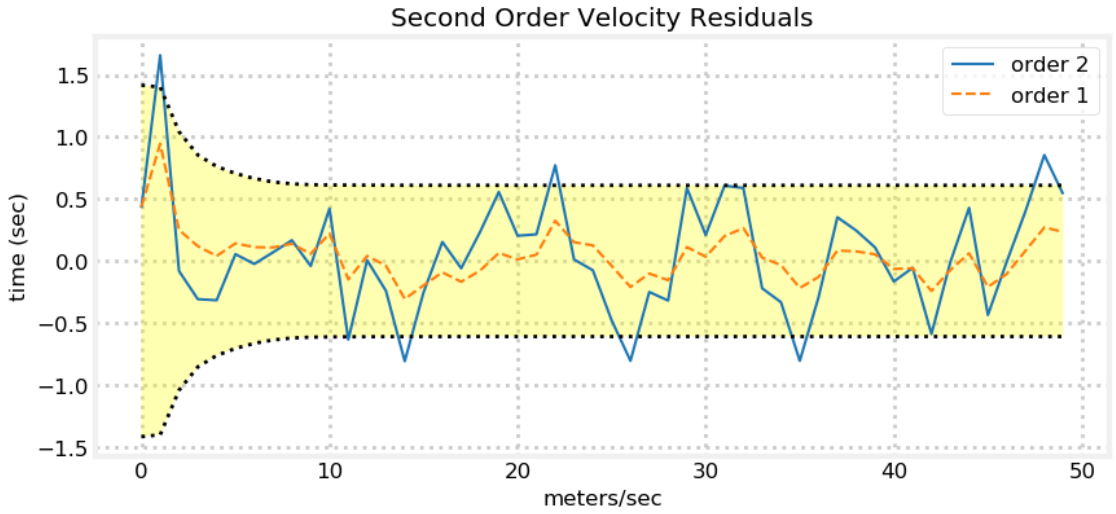
```
In [28]: res2 = xs[:, 0] - data2.x[:, 0]
         res1 = xs[:, 0] - data1.x[:, 0]

         plt.plot(res1, ls="--", label='order 1')
         plt.plot(res2, label='order 2')
         plot_residual_limits(data2.P[:, 0, 0])
         set_labels('Second Order Position Residuals',
                    'meters', 'time (sec)')
         plt.legend();
```




The second order position residuals are slightly worse than the residuals of the first order filter, but they still fall within the theoretical limits of the filter. There is nothing very alarming here.

Now let's look at the residuals for the velocity.

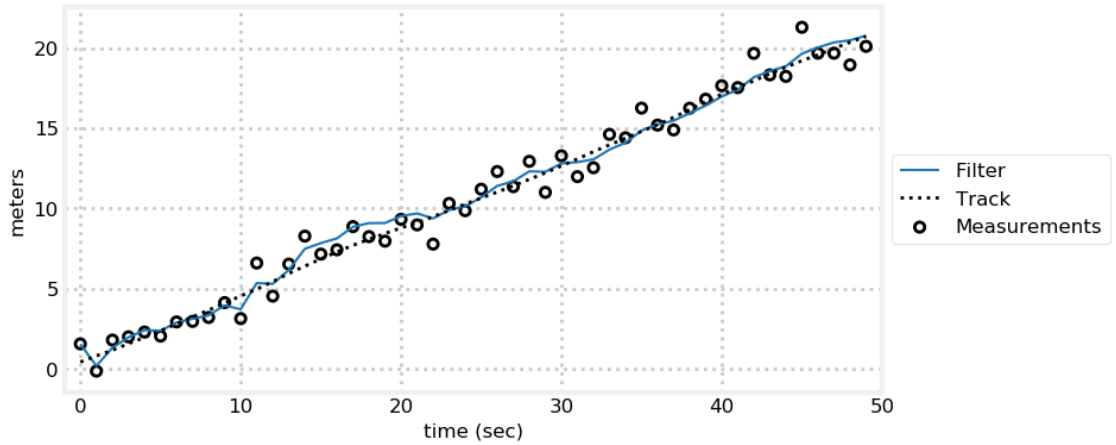
```
In [29]: res2 = xs[:, 1] - data2.x[:, 1]
         res1 = xs[:, 1] - data1.x[:, 1]

         plt.plot(res2, label='order 2')
         plt.plot(res1, ls='--', label='order 1')
         plot_residual_limits(data2.P[:, 1, 1])
         set_labels('Second Order Velocity Residuals',
                               'meters/sec', 'time (sec)')
         plt.legend();
```




Here the story is very different. While the residuals of the second order system fall within the theoretical bounds of the filter's performance, we can see that the residuals are *far* worse than for the first order filter. This is the usual result for this scenario. The filter is assuming that there is acceleration that does not exist. It mistakes noise in the measurement as acceleration and this gets added into the velocity estimate on every predict cycle. Of course the acceleration is not actually there and so the residual for the velocity is much larger than its optimum.

I have one more trick up my sleeve. We have a first order system; i.e. the velocity is more-or-less constant. Real world systems are never perfect, so of course the velocity is never exactly the same between time periods. When we use a first order filter we account for that slight variation in velocity with the *process noise*. The matrix $\mathbf{Q}$ is computed to account for this slight variation. If we move to a second order filter we are now accounting for the changes in velocity. Perhaps now we have no process noise, and we can set $\mathbf{Q}$ to zero!

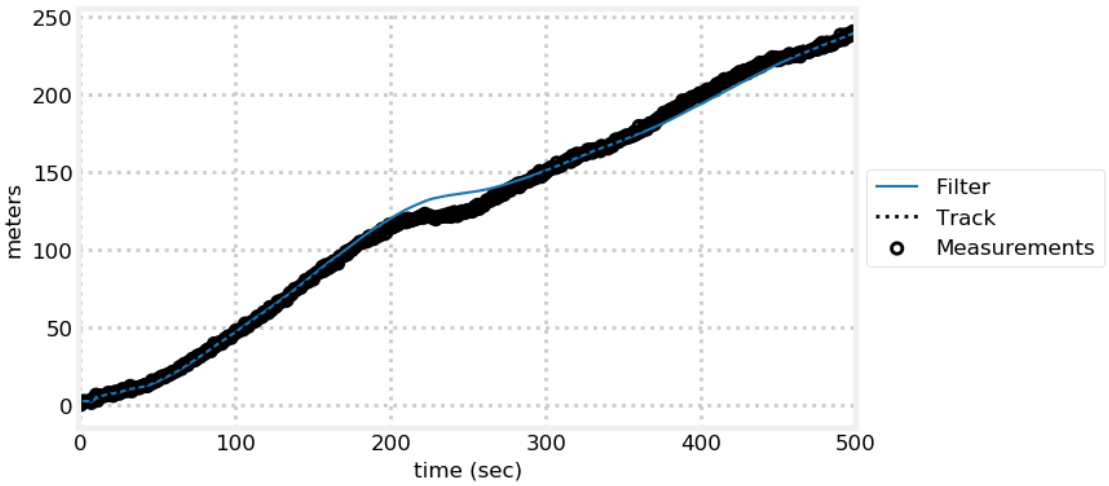
```
In [30]: kf2 = SecondOrderKF(R, 0, dt=1)
         data2 = filter_data(kf2, zs)
         plot_kf_output(xs, data2.x, data2.z)
```

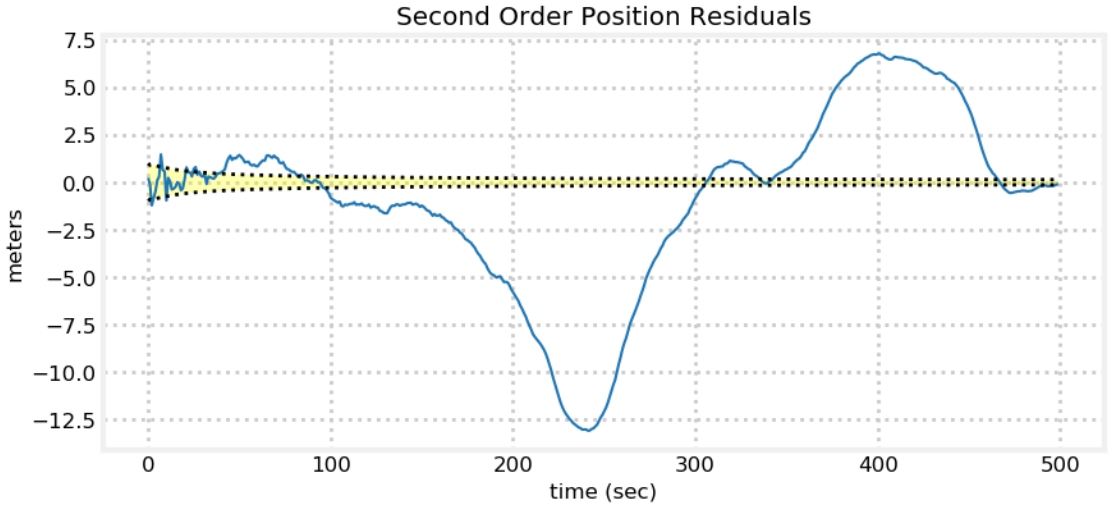



To my eye it looks like the filter quickly converges to the actual track. Success!

Or, maybe not. Setting the process noise to 0 tells the filter that the process model is perfect. Let's look at the performance of the filter over a longer period of time.

```
In [31]: np.random.seed(25944)
         xs500, zs500 = simulate_system(Q=Q, count=500)

         kf2 = SecondOrderKF(R, 0, dt=1)
         data500 = filter_data(kf2, zs500)

         plot_kf_output(xs500, data500.x, data500.z)
         plot_residuals(xs500[:, 0], data500, 0,
                        'Second Order Position Residuals',
                        'meters')
```



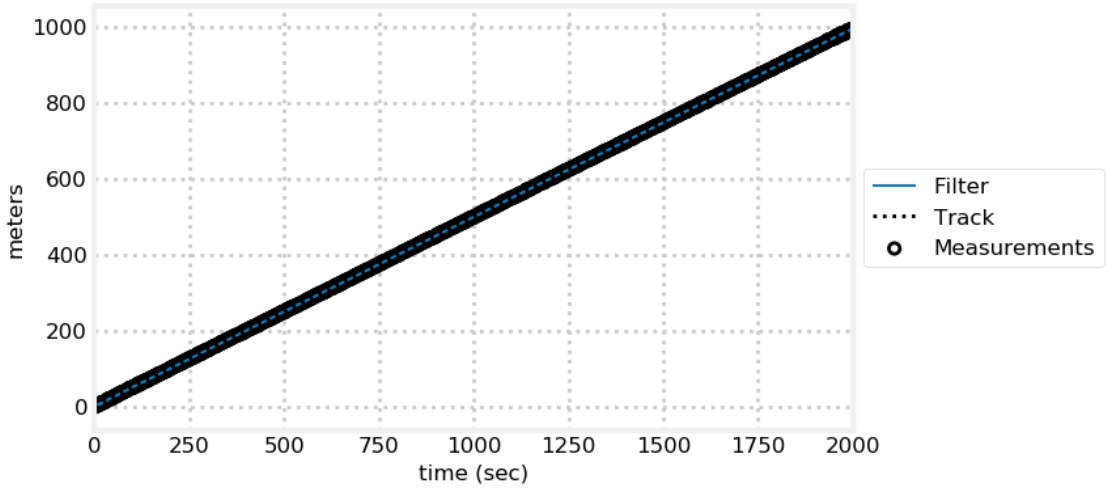



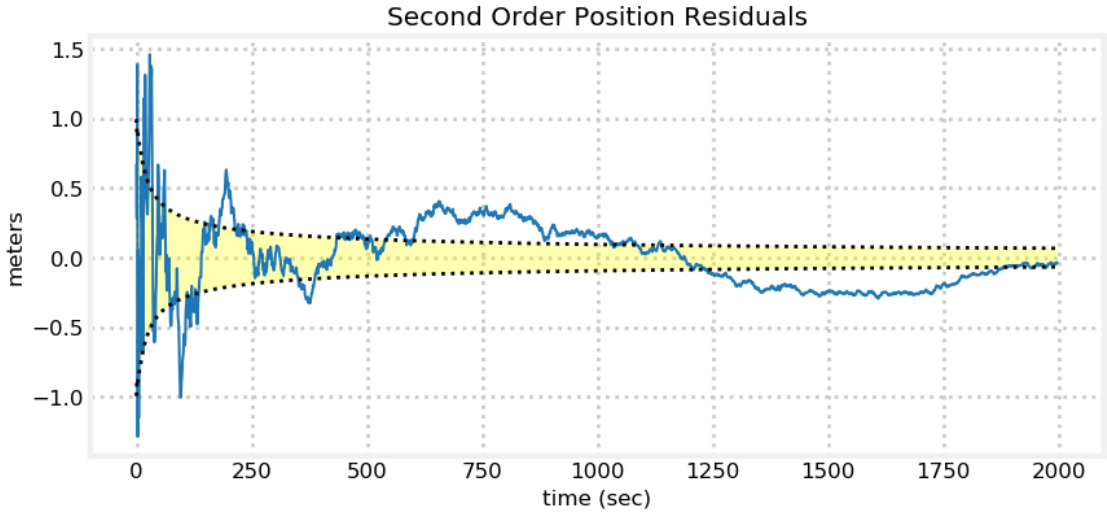

We can see that the performance of the filter is abysmal. We can see that in the track plot where the filter diverges from the track for an extended period of time.The residual plot makes the problem more apparent. Just before the 100th update the filter diverges sharply from the theoretical performance. It *might* be converging at the end, but I doubt it. The entire time, the filter is reporting smaller and smaller variances. **Do not trust the filter's covariance matrix to tell you if the filter is performing well!**

Why is this happening? Recall that if we set the process noise to zero we are telling the filter to use only the process model. The measurements end up getting ignored. The physical system is *not* perfect, and so the filter is unable to adapt to this imperfect behavior.

Maybe just a really low process noise? Let's try that.

```
In [32]: np.random.seed(32594)
         xs2000, zs2000 = simulate_system(Q=0.0001, count=2000)

         kf2 = SecondOrderKF(R, 0, dt=1)
         data2000 = filter_data(kf2, zs2000)

         plot_kf_output(xs2000, data2000.x, data2000.z)
         plot_residuals(xs2000[:, 0], data2000, 0,
                        'Second Order Position Residuals',
                        'meters')
```



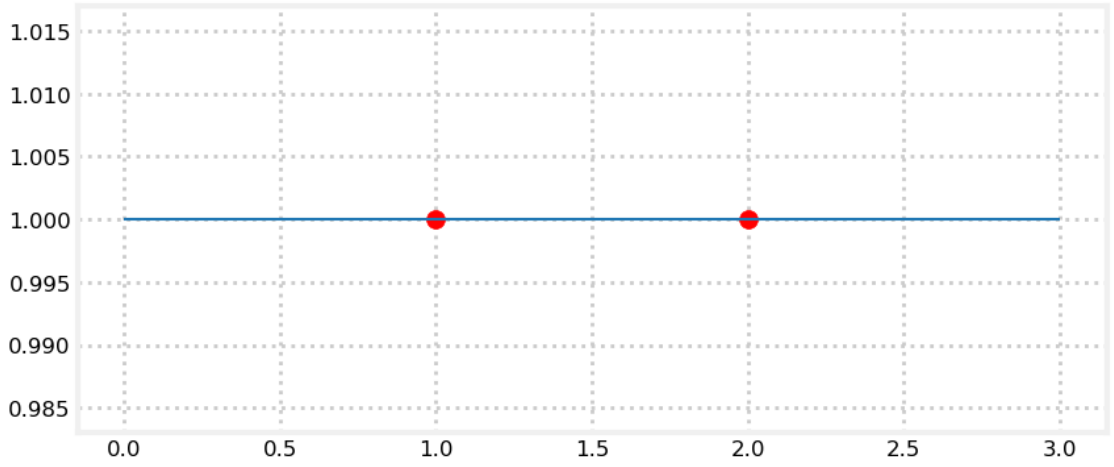



Again, the residual plot tells the story. The track looks very good, but the residual plot shows that the filter is diverging for significant periods of time.

How should you think about all of this? You might argue that the last plot is 'good enough' for your application, and perhaps it is. I warn you however that a diverging filter doesn't always converge. With a different data set, or a physical system that performs differently you can end up with a filter that veers further and further away from the measurements.

Also, let's think about this in a data fitting sense. Suppose I give you two points, and tell you to fit a straight line to the points.

```
In [33]: plt.scatter([1, 2], [1, 1], s=100, c='r')
         plt.plot([0, 3], [1, 1]);
```




A straight line is the only possible answer. Furthermore, the answer is optimal. If I gave you more points you could use a least squares fit to find the best line, and the answer would still be optimal in a least squares sense.

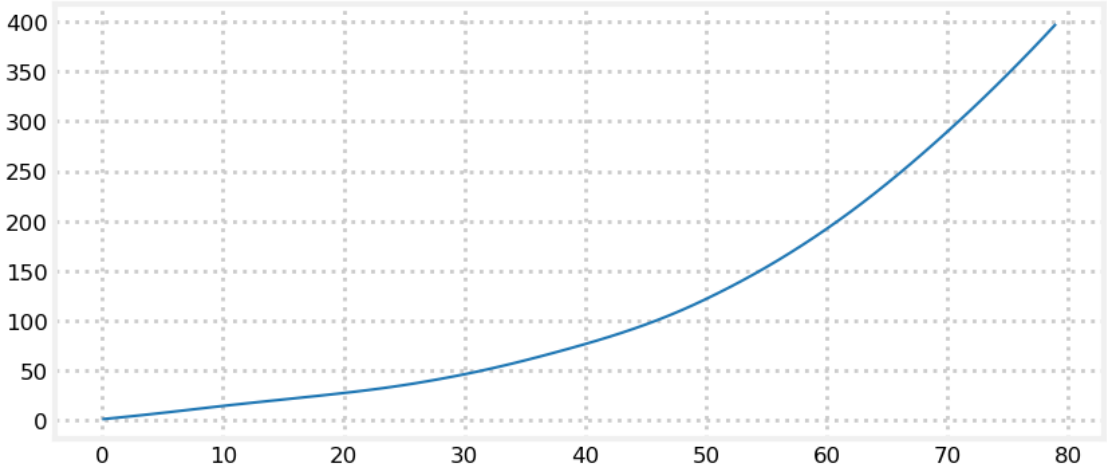
But suppose I told you to fit a higher order polynomial to those two points. There are an infinite number of answers to the problem. For example, an infinite number of second order parabolas pass through those points. When the Kalman filter is of higher order than your physical process it also has an infinite number of solutions to choose from. The answer is not just non-optimal, it often diverges and never recovers.

For best performance you need a filter whose order matches the system's order. In many cases that will be easy to do - if you are designing a Kalman filter to read the thermometer of a freezer it seems clear that a zero order filter is the right choice. But what order should we use if we are tracking a car? Order one will work well while the car is moving in a straight line at a constant speed, but cars turn, speed up, and slow down, in which case a second order filter will perform better. That is the problem addressed in the Adaptive Filters chapter. There we will learn how to design a filter that adapts to changing order in the tracked object's behavior.

With that said, a lower order filter can track a higher order process so long as you add enough process noise and you keep the discretization period small (100 samples a second are usually locally linear). The results will not be optimal, but they can still be very good, and I always reach for this tool first before trying an adaptive filter. Let's look at an example with acceleration. First, the simulation.

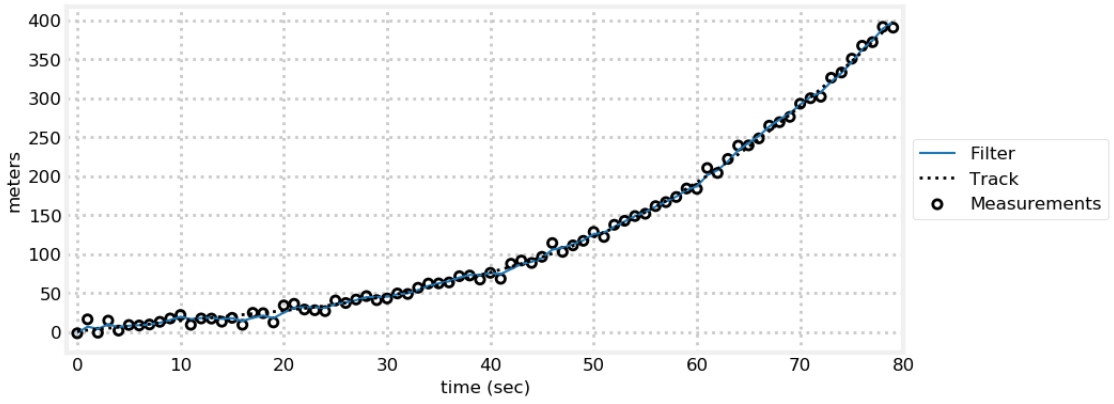
```
In [34]: class ConstantAccelerationObject(object):
             def __init__(self, x0=0, vel=1., acc=0.1, acc_noise=.1):
                 self.x = x0
                 self.vel = vel
                 self.acc = acc
                 self.acc_noise_scale = acc_noise
```

```
        def update(self):
            self.acc += randn() * self.acc_noise_scale
            self.vel += self.acc
            self.x += self.vel
            return (self.x, self.vel, self.acc)

    R, Q = 6., 0.02
    def simulate_acc_system(R, Q, count):
        obj = ConstantAccelerationObject(acc_noise=Q)
        zs = []
        xs = []
        for i in range(count):
            x = obj.update()
            z = sense(x, R)
            xs.append(x)
            zs.append(z)
        return np.asarray(xs), zs

    np.random.seed(124)
    xs, zs = simulate_acc_system(R=R, Q=Q, count=80)
    plt.plot(xs[:, 0]);
```




Now we will filter the data using a second order filter.

In [35]:
```
np.random.seed(124)
xs, zs = simulate_acc_system(R=R, Q=Q, count=80)

kf2 = SecondOrderKF(R, Q, dt=1)
data2 = filter_data(kf2, zs)

plot_kf_output(xs, data2.x, data2.z, aspect_equal=False)
plot_residuals(xs[:, 0], data2, 0,
               'Second Order Position Residuals',
               'meters')
```

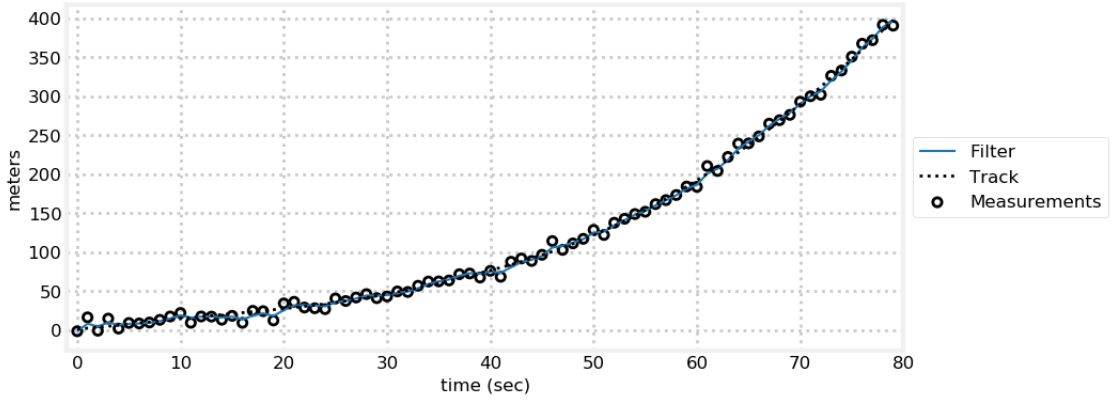


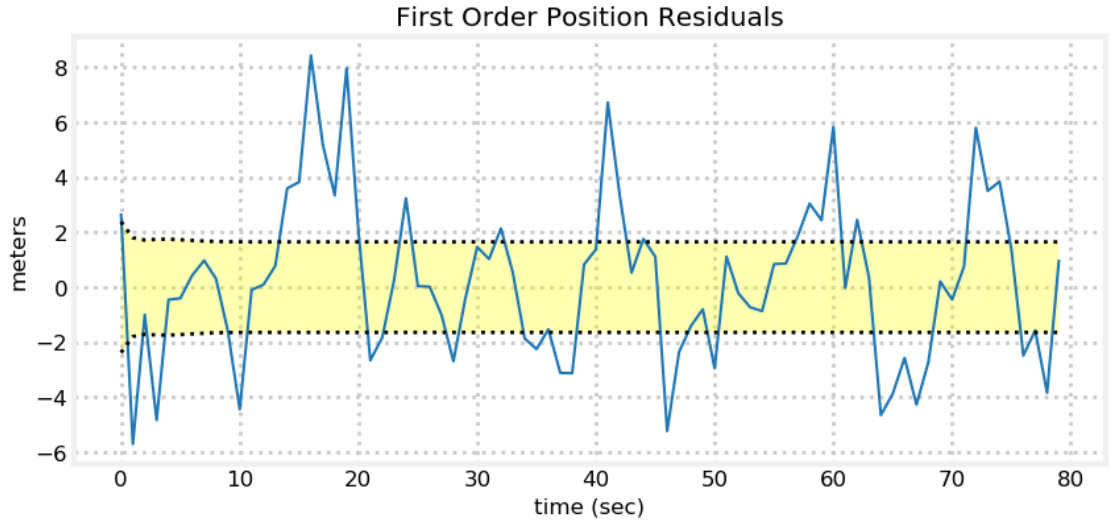


We can see that the filter is performing within the theoretical limits of the filter.

Now let's use a lower order filter. As already demonstrated the lower order filter will lag the signal because it is not modeling the acceleration. However, we can account for that (to an extent) by increasing the size of the process noise. The filter will treat the acceleration as noise in the process model. The result will be suboptimal, but if designed well it will not diverge. Choosing the amount of extra process noise is not an exact science. You will have to experiment with representative data. Here, I've multiplied it by 10, and am getting good results.

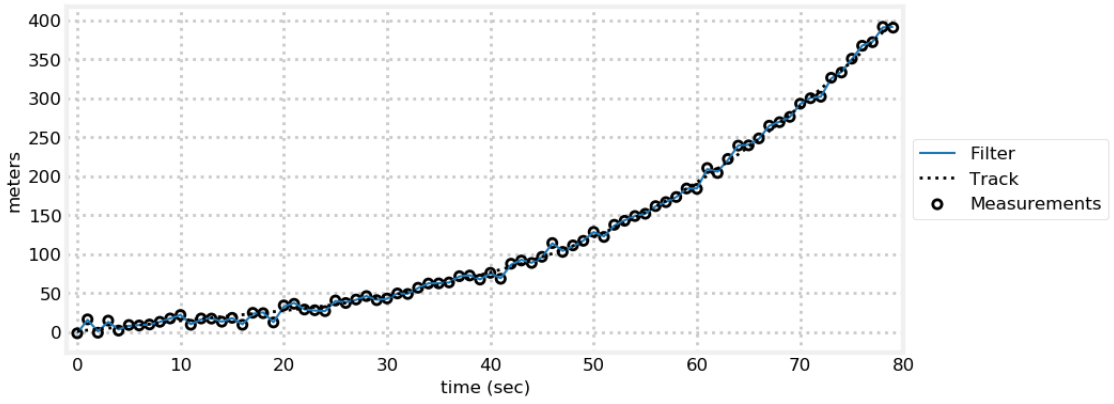
```
In [36]: kf3 = FirstOrderKF(R, Q * 10, dt=1)
         data3= filter_data(kf3, zs)

         plot_kf_output(xs, data3.x, data3.z, aspect_equal=False)
         plot_residuals(xs[:, 0], data3, 0,
                        'First Order Position Residuals',
                        'meters')
```




Think about what will happen if you make the process noise many times larger than it needs to be. A large process noise tells the filter to favor the measurements, so we would expect the filter to closely mimic the noise in the measurements. Let's find out.

```
In [37]: kf4 = FirstOrderKF(R, Q * 10000, dt=1)
         data4 = filter_data(kf4, zs)

         plot_kf_output(xs, data4.x, data4.z, aspect_equal=False)
         plot_residuals(xs[:, 0], data4, 0,
                        'First Order Position Residuals',
                        'meters')
```





## 8.5 Exercise: State Variable Design

As I've mentioned, you may place the variables in $\mathbf{x}$ in whatever order you choose. For example, you could define a 1D constant acceleration as $\mathbf{x} = \begin{bmatrix}\ddot{x} & x & \dot{x}\end{bmatrix}^\mathsf{T}$. I can't imagine why you would want that order, but it is possible.

Let's do something more reasonable. Design a second order filter for a robot moving in 2D, where $\mathbf{x} = \begin{bmatrix}x & y & \dot{x} & \dot{y}\end{bmatrix}^\mathsf{T}$. In this chapter we have been using $\mathbf{x} = \begin{bmatrix}x & \dot{x} & y & \dot{y}\end{bmatrix}^\mathsf{T}$.

Why would you choose a different ordering? As you'll see in a moment, changing the order of $\mathbf{x}$ changes the order of most of the rest of the filter's matrices. Depending on what data you want to inspect, such as the correlations in $\mathbf{P}$, various orderings of $\mathbf{x}$ can make that easier or more difficult.

Think about how to do this. What needs to change? Clearly, you just need to change the Kalman filter matrices to reflect this new design.

Give it a try below using this boilerplate code:

```
N = 30 # number of iterations
```

```
dt = 1.0 # time step
R_std = 0.35
Q_std = 0.04

sensor = PosSensor((0, 0), (2, .5), noise_std=R_std)
zs = np.array([sensor.read() for _ in range(N)])

tracker = KalmanFilter(dim_x=4, dim_z=2)
# assign state variables here

xs, ys = [], []
for z in zs:
    tracker.predict()
    tracker.update(z)
    xs.append(tracker.x[0])
    ys.append(tracker.x[1])
plt.plot(xs, ys);
```

### 8.5.1 Solution

Let's start with $\mathbf{F}$. With a bit of practice you should be able to just write out the matrix. If you found that difficult, then write out the set of equations for $\mathbf{F}$ ordering the variables as used by the state variable.

$$x = 1x + 0y + 1\dot{x}\Delta t + 0\dot{y}\Delta t \quad y = 0x + 1y + 0\dot{x}\Delta t + 1\dot{y}\Delta t \quad \dot{x} = 0x + 0y + 1\dot{x}\Delta t + 0\dot{y}\Delta t \quad \dot{y} = 0x + 0y + 0\dot{x}\Delta t + 1\dot{y}\Delta t$$

We can copy out the coefficients to get

$$\mathbf{F} = \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The state noise also requires shuffling things around. First, figure out how it should be arranged for the state variable ordering. This can be easier if you write the state variables vertically and horizontally across the matrix to see how they pair up. This is hard to do in Jupyter notebook, so I will forgo it here.

$$\mathbf{Q} = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\dot{x}} & \sigma_{x\dot{y}} \\ \sigma_{yx} & \sigma_y^2 & \sigma_{y\dot{x}} & \sigma_{y\dot{y}} \\ \sigma_{\dot{x}x} & \sigma_{\dot{x}y} & \sigma_{\dot{x}}^2 & \sigma_{\dot{x}\dot{y}} \\ \sigma_{\dot{y}x} & \sigma_{\dot{y}y} & \sigma_{\dot{y}\dot{x}} & \sigma_{\dot{y}}^2 \end{bmatrix}$$

There is no correlation between $x$ and $y$, so we can set terms with both to zero

$$\mathbf{Q} = \begin{bmatrix} \sigma_x^2 & 0 & \sigma_{x\dot{x}} & 0 \\ 0 & \sigma_y^2 & 0 & \sigma_{y\dot{y}} \\ \sigma_{\dot{x}x} & 0 & \sigma_{\dot{x}}^2 & 0 \\ 0 & \sigma_{\dot{y}y} & 0 & \sigma_{\dot{y}}^2 \end{bmatrix}$$

Now that you see this you can see the pattern and perhaps design $\mathbf{Q}$ more quickly.

`Q_discrete_white_noise` generates a matrix with a different ordering, but we can just copy terms from it, which we will see in the code below.

Now let's design $\mathbf{H}$. It converts the state $\begin{bmatrix} x & y & \dot{x} & \dot{y} \end{bmatrix}^\mathsf{T}$ into the measurement $\mathbf{z} = \begin{bmatrix} z_x & z_y \end{bmatrix}^\mathsf{T}$.

$$\mathbf{Hx} = \mathbf{z}$$

$$\begin{bmatrix} ? & ? & ? & ? \\ ? & ? & ? & ? \end{bmatrix} \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} z_x \\ z_y \end{bmatrix}$$

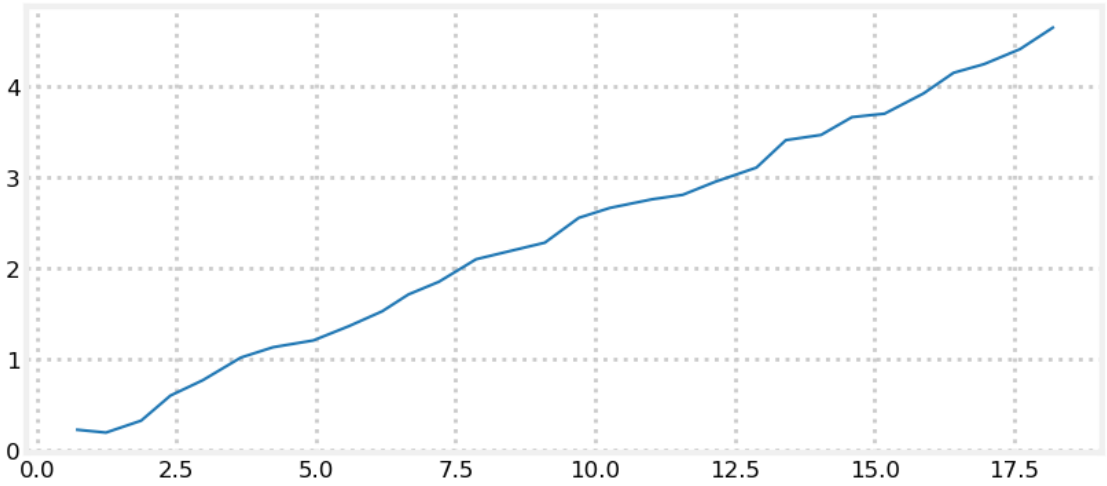
Now it should be easy to fill in the matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} z_x \\ z_y \end{bmatrix}$$

The measurement $\mathbf{z} = \begin{bmatrix} z_x & z_y \end{bmatrix}^\mathsf{T}$ has not changed so $\mathbf{R}$ does not change. Finally, $\mathbf{P}$. It uses the same ordering as $\mathbf{Q}$, so it is already designed for us.

In [38]:
```
N = 30 # number of iterations
dt = 1.0 # time step
R_std = 0.35
Q_std = 0.04

M_TO_FT = 1 / 0.3048

sensor = PosSensor((0, 0), (2, .5), noise_std=R_std)
zs = np.array([sensor.read() for _ in range(N)])

tracker = KalmanFilter(dim_x=4, dim_z=2)

tracker.F = np.array([[1, 0, dt,  0],
                      [0, 1,  0, dt],
                      [0, 0,  1,  0],
                      [0, 0,  0,  1]])

tracker.H = np.array([[M_TO_FT, 0, 0, 0],
                      [0, M_TO_FT, 0, 0]])

tracker.R = np.eye(2) * R_std**2
q = Q_discrete_white_noise(dim=2, dt=dt, var=Q_std**2)
tracker.Q[0,0] = q[0,0]
tracker.Q[1,1] = q[0,0]
tracker.Q[2,2] = q[1,1]
tracker.Q[3,3] = q[1,1]
tracker.Q[0,2] = q[0,1]
tracker.Q[2,0] = q[0,1]
tracker.Q[1,3] = q[0,1]
tracker.Q[3,1] = q[0,1]

tracker.x = np.array([[0, 0, 0, 0]]).T
tracker.P = np.eye(4) * 500.

xs, ys = [], []
for z in zs:
    tracker.predict()
    tracker.update(z)
```

```
        xs.append(tracker.x[0])
        ys.append(tracker.x[1])
    plt.plot(xs, ys);
```




## 8.6 Detecting and Rejecting Bad Measurement

The Kalman filter provides no way to detect and reject a bad measurement. Suppose you are tracking aircraft, and you receive a measurement 100km away from the aircraft's current position. If you call update with that value the new estimate will be skewed radically towards the measurement.

I'll run a simulation to give us a concrete example. After 100 epoch's I'll do an update with a measurement equal to twice the current position. `filterpy.common` provides `kinematic_kf` which creates a linear kinematic filter of arbitrary dimension and order. I'll use it here to keep the code tidy, I don't use it in the rest of the book because I want you to get lots of practice writing filters.

```
In [39]: from filterpy.common import kinematic_kf

        kf = kinematic_kf(dim=2, order=1, dt=1.0, order_by_dim=False)
        kf.Q = np.diag([0, 0, .003, .003])
        kf.x = np.array([[1., 1., 0., 0.]]).T
        kf.R = np.diag([0.03, 0.21]) # use different errors

        for i in range(101):
            kf.predict()
            kf.update(np.array([[i*.05, i*.05]])) # around 200 kph

        p0 = kf.x[0:2]

        kf.predict()
        prior = kf.x
        z = kf.x[0:2]*2
        kf.update(z)
        p1 = kf.x[0:2]

        # compute error of measurement from prior
```

```
y = np.abs(z - np.dot(kf.H, prior))
dist = np.linalg.norm(y)

np.set_printoptions(precision=2, suppress=True)

print('bad measurement       : {} km'.format(z.T))
print('before bad measurement: {} km'.format(p0.T))
print('after bad measurement : {} km'.format(p1.T))
print('estimate shift        : {:.1f} km'.format(np.linalg.norm(p1 - prior[:2])))
print('distance from prior   : {:.1f} km'.format(dist))
```

```
---------------------------------------------------------------------------

ValueError                                Traceback (most recent call last)

<ipython-input-39-7f019f1f50ad> in <module>
  8 for i in range(101):
  9     kf.predict()
---> 10     kf.update(np.array([[i*.05, i*.05]])) # around 200 kph
 11
 12 p0 = kf.x[0:2]


C:\pcloud\filterpy\filterpy\kalman\kalman_filter.py in update(self, z, R, H)
531
532         if H is None:
--> 533             z = reshape_z(z, self.dim_z, self.x.ndim)
534             H = self.H
535


C:\pcloud\filterpy\filterpy\common\helpers.py in reshape_z(z, dim_z, ndim)
309
310     if z.shape != (dim_z, 1):
--> 311         raise ValueError('z (shape {}) must be convertible to shape ({}, 1)'.format(z.shape
312
313     if ndim == 1:


ValueError: z (shape (1, 2)) must be convertible to shape (1, 1)
```

`kinematic_kf`? What's that? `filterpy.common` provides `kinematic_kf` to allow you to create a linear kinematic filter of arbitrary dimension and order. I don't use it in this book because I want you to gain a lot of experience creating Kalman filters. I used it here just to keep my example short and to expose you to this part of the library.

Back on topic. As you can see the estimate jumped 3.4 km, and the error between the prediction (prior) and the measurement is over 7 km.

What can we do to avoid this? Our first thought might be to just add a check if the prior is far from the measurement. Why the prior and not the current estimate? Because after the update the estimate could now be quite close to the bad measurement, although it isn't in this case.

Note that while I could have just written `prior[0:2] - z` to get the error, I used the mathematically correct $\mathbf{z} - \mathbf{Hx}$ to compute the error. This is just for illustration; the Kalman filter class stores the innovation in `KalmanFilter.y`. I use it instead of the value I computed above to illustrate this:

```
In [40]: print('error = {:.1f} km, at a speed of {:.0f} kph'.format(np.linalg.norm(kf.y), dist*3600))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)

<ipython-input-40-b7588efd6a7c> in <module>
----> 1 print('error = {:.1f} km, at a speed of {:.0f} kph'.format(np.linalg.norm(kf.y), dist*3600))


NameError: name 'dist' is not defined
```

In this example the measurement is nearly 7 km away from the predicted position. That sounds "far". Is it? It could be far if the units are kilometers and the update rate is 1 second; no aircraft can travel the over 25000 kph this error implies. It could be absurdly small if the units are centimeters and the epoch is 1 minute.

We could add a check that either takes the performance limits of the aircraft into account:

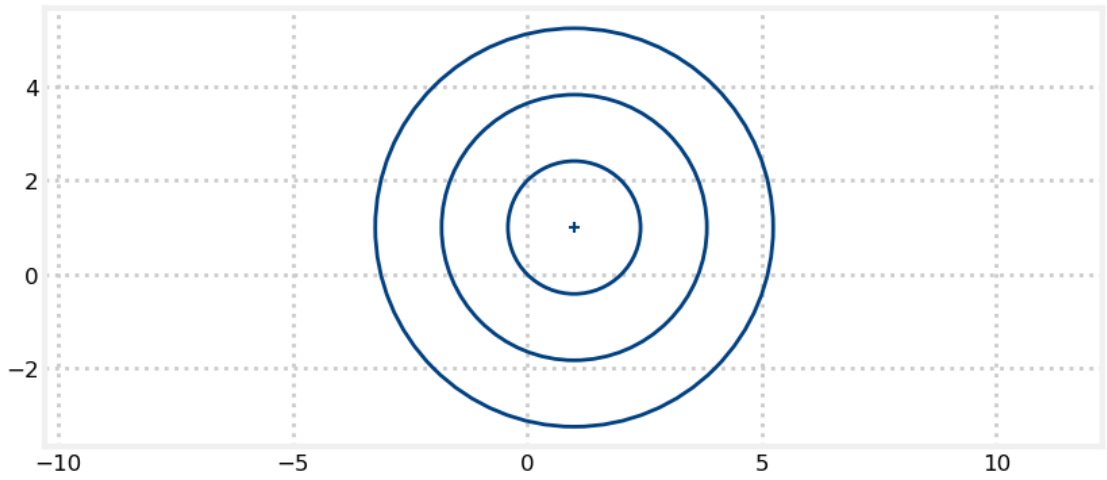
```
vel = y / dt
if vel >= MIN_AC_VELOCITY and vel <= MAX_AC_VELOCITY:
    kf.update()
```

Do you think this is a reasonable and robust solution? Come up with as many objections as you can before reading further.

This is not very satisfying to me. Suppose we had just initialized the filter with an guessed at position; we would discard good measurements and never start filtering. Second, this ignores the knowledge we have about the sensor and process errors. The Kalman filter maintains its current accuracy in $\mathbf{P}$. If $\mathbf{P}$ implies that $\sigma_x = 10$ meters, and the measurement is 1 km away, clearly the measurement is bad as it is 100 standard deviations away from the prior.

Let's plot $\mathbf{P}$. I'll plot the first, second, and third standard deviations.

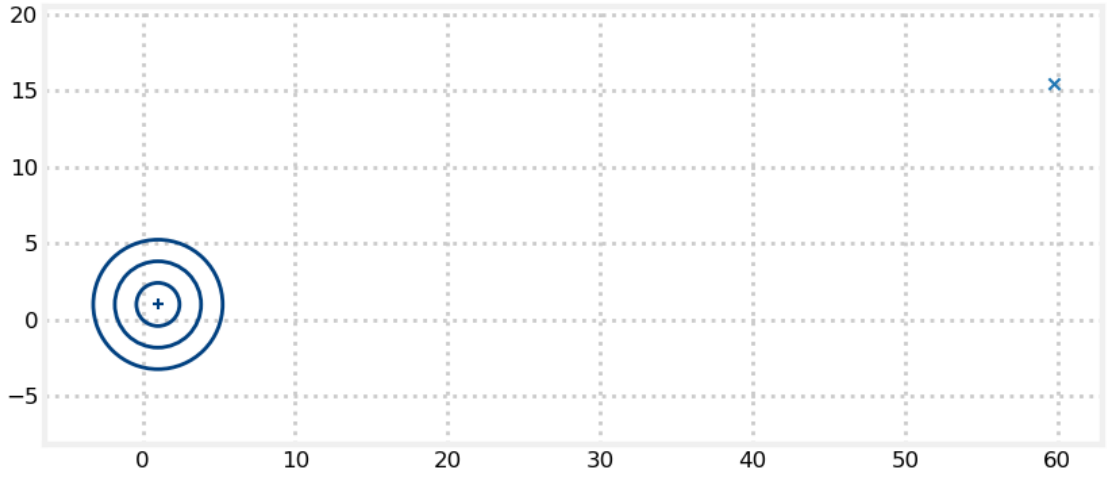
```
In [41]: x, P = kf.x[0:2], kf.P[0:2, 0:2]
         plot_covariance_ellipse(x, P, std=[1,2,3])
```



In previous sections $\mathbf{P}$ was a cicle, not an ellipse. In my code I set $\mathbf{R} = \begin{bmatrix} 0.03 & 0 \\ 0 & 0.15 \end{bmatrix}$, which gives the measurement of $y$ 5 times the error of $x$. That's somewhat artificial, but in subsequent chapters nearly all of the covariances will be elliptical for good reasons.

Think about what this means. Statistics tells us that 99% of all measurements fall within 3 standard deviations; that means 99% of all measurements should be within this ellipse. Let's plot the measurement with the ellipse.

```
In [42]: plot_covariance_ellipse(x, P, std=[1,2,3])
         plt.scatter(z[0], z[1], marker='x');
```




Clearly the measurement is far beyond the the prior's covariance; we probably want to consider this a bad measurement and not use it. How would we do that?

The first idea would to be extract the standard deviation for $x$ and $y$ and write a simple if statement. Here I will use another feature of the `KalmanFilter` class. The method `residual_of` computes the residual compared to the prior. I don't need to use it in this case since `kf.y` is already assigned by the call to `update()`, but if we are discarding measurements `update()` would not have been called yet and `kf.y` would contain the innovation from the previous epoch.

First, let's introduce two terms. We are discussing **gating**. A **gate** is a formula or algorithm that determines if a measurement is good or bad. Only good measurements get through the gate. This process is called gating.

In practice measurements are not purely Gaussian, so a gate of 3 standard deviations is likely to discard some good measurements. I will elaborate soon, for now we will use 4 standard deviations

```
In [43]: GATE_LIMIT = 4.
         std_x = np.sqrt(P[0,0])
         std_y = np.sqrt(P[1,1])
         y = kf.residual_of(z)[:,0]

         if y[0] > GATE_LIMIT * std_x or y[1] > GATE_LIMIT * std_y:
             print('discarding measurement, error is {:.0f} std, {:.0f} std'.format(y[0]/std_x, y[1]/st

         print('y   is', y)
         print('std is {:.2f} {:.2f}'.format(std_x, std_y))
```

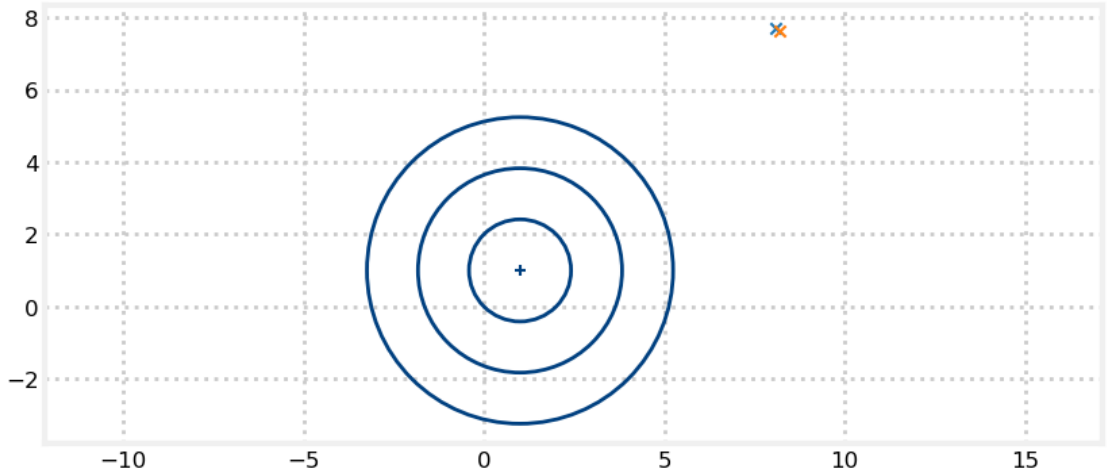
---

```
IndexError                                Traceback (most recent call last)

<ipython-input-43-bac38cacf2e4> in <module>
  5
  6 if y[0] > GATE_LIMIT * std_x or y[1] > GATE_LIMIT * std_y:
----> 7     print('discarding measurement, error is {:.0f} std, {:.0f} std'.format(y[0]/std_x, y[1]/
  8
  9 print('y   is', y)


IndexError: index 1 is out of bounds for axis 0 with size 1
```

We see the errors are roughly 39 and 18 standard deviations away. Is this good enough?

Maybe. However, notice that the if statement forms a rectangular region surrounding the ellipse. In the plot below I've drawn a measurement that is clearly outside of the 3 std ellipse, yet would be accepted by the gate, and another measurement that lies just on the 3 std boundary.

```
In [44]: plot_covariance_ellipse(x, P, std=[1,2,3])
         plt.scatter(8.08, 7.7, marker='x');
         plt.scatter(8.2, 7.65, marker='x');
```




There are alternative methods of defining this gate. The **mahalanobis distance** is a statistical measure of the distance of a point from a distribution. Before we get to the definition and math, let's compute the mahalanobis distance for some points. `filterpy.stats` implements `mahalanobis()`.

```
In [45]: from filterpy.stats import mahalanobis

         print('mahalanobis distance = {:.1f}'.format(mahalanobis(x=z, mean=x, cov=P)))
```

```
mahalanobis distance = 42.8
```

Without knowing the units we can compare it to the standard deviation errors computed in $x$ and $y$ separately, 39, and 18, and see that it is reasonably close. Let's see what we get for the point I plotted above.

```
In [46]: print('mahalanobis distance = {:.1f}'.format(mahalanobis(x=[8.08, 7.7], mean=x, cov=P)))
         print('mahalanobis distance = {:.1f}'.format(mahalanobis(x=[8.2, 7.65], mean=x, cov=P)))

mahalanobis distance = 6.9
mahalanobis distance = 6.9
```

As we will see the mahalanobis distance computes the scalar standard deviation *distance* point to a distribution, much like the Euclidian distance computes a scalar distance from a point to another point.

The cell above bears that out. The point that was sitting on the 3 std boundary has a mahalanobis distance of 3.0, and the one outside of the ellipse has a value of 3.6 std.

How do we compute the mahalanobis distance? It is defined as

$$D_m = \sqrt{(\mathbf{x} - \mu)^\mathsf{T}\mathbf{S}^{-1}(\mathbf{x} - \mu)}$$

Note how similar this is to the Euclidean distance, which we can write as:

$$D_e = \sqrt{(\mathbf{x} - \mathbf{y})^\mathsf{T}(\mathbf{x} - \mathbf{y})}$$

In fact, if the covariance $\mathbf{S}$ is the identity matrix the mahalanobis distance is the same as the Euclidean distance. You should be able to see how that works with the linear algebra: the inverse of the identity matrix is the identity matrix, so we are effectively multiplying the terms with 1. Think of it intuitively. If the standard deviation in each dimension is 1, then the a point anywhere on a circle of radius 1 around the mean will lie on the 1 std circle and also be 1 *unit* away in Euclidean measure.

This suggests another interpretation. If the covariance matrix is diagonal then we can think of the mahalanobis distance as being the *scaled* Euclidean distance, where each term is scaled by the covariance in the diagonal.

$$D_m = \sqrt{\sum_{i-1}^{N} \frac{(x_i - \mu_i)^2}{\sigma_i}}$$

In 2D that would be

$$D_m = \sqrt{\frac{1}{\sigma_x^2}(x_0 - x_1)^2 + \frac{1}{\sigma_y^2}(y_0 - y_1)^2}$$

This should give you insight into the equation for mahalanobis distance. You cannot divide by a matrix, but multiplying by the inverse is *effectively* the same, in a hand wavey sense. Multiplying by the difference $\mathbf{y} = \mathbf{x} - \mu$ on each side gives us the squared norm scaled by the covariance: $\mathbf{y}^\mathsf{T}\mathbf{S}^{-1}\mathbf{y}^\mathsf{T}$. The covariance terms are all squared, so taking the square root at the end brings us to a scalar distance which is the Euclidean distance scaled by the covariance.

### 8.6.1 Gating and Data Association Strategies

The two gates above are known as rectangular and ellipsoidal gates in some of the literature because of their shape. There are more alternatives I will not explore here. For example, a maneuver gate is used to define a region where it is possible for the object to maneuver, taking the object's current velocity and ability to maneuver into account. For example, a fighter jet would have a maneuver gate shaped much like a cone projecting in front of it's current direction of travel. An automobile would have a much smaller 2D pie slice shape projecting in front of it. A ship would have a much narrower slice since it's ability to turn or accelerate quickly is minimal.

Which gating technique should you use? There is no single answer. Much depends on the dimensionality of the problem and how much computational power you have. A rectangular gate is very cheap to compute, the maneuver gate isn't much worse, but the ellipsoidal gate can be expensive in higher dimensions. However,

as you increase dimensions the relative area difference between the ellipsoid and rectangular gate increases significantly.

This matters more than you might realize. Every measurement has noise. A spurious measurement can fall *within* our gate, causing us to accept it. The larger the area that extends past the ellipsoid, the larger the probability of the gate passing a bad measurement. I will not do the math here, but in 5 dimensions the rectangular gate is twice as likely to accept a bad measurement as the ellipsoid.

If computational time is a concern and you have many spurious measurements you can take a two gate approach. The first gate is large and rectangular, and used as a first pass to throw away obviously bad measurements. The few measurements that pass that gate are then subjected to the more expensive mahalanobis distance calculation. If you are running on a modern desktop processor the time of those matrix multiplies are not significant, but it may matter if you are running on an embedded chip with modest floating point performance.

Data association is a vast topic that requires its own book. The canonical example is tracking with radar. The radar detects multiple signals on each sweep. From that we need to form aircraft tracks, and reject noisy measurements. This is a very difficult problem. Suppose the first sweep gets 5 measurements. We would create 5 potential tracks. In the next sweep we get 6 measurements. We can combine any of the first measurements with any of the second measurements, giving us 30 potential tracks. But, it is also possible that these are all new aircraft that we didn't see in the last sweep, so that gives us 6 more tracks. After just a few epochs we reach millions, then billions of potential tracks.

Given this list of billions of tracks we can then compute a score for each track. I'll provide the math for that in the following section. But, visualize a track that forms a 'Z' shape over 3 epochs. No aircraft can maneuver like that, so we would give it a very low probability of being real. Another track forms a straight line, but imputes a velocity of 10,000 kph. That is also very improbable. Another track curves at 200 kph. That has a high probability.

So tracking becomes a matter of gating, data association, and pruning. For example, say the second radar sweep just occured. Do I combine all possible combinations into a tracks? I probably shouldn't. If point 1, sweep 1 imputes a velocity of 200kph with point 3, sweep 2 we should form a track from it. If the velocity is 5000 kph we shouldn't bother; we know that track is so unlikely as to be impossible. Then, as the tracks grow we will have well defined ellipsoidal or maneuver gates for them, and we can be far more selective about the measurements we associate with tracks.

There are schemes for associations. We can choose to associate a measurement to only one track. Or, we can choose to associate a measurement with multiple tracks, reflecting our lack of certainty with which track it belongs to. For example, aircraft tracks can cross from the point of view of the radar. As the aircraft approach associating a single measurement with one of the two aircraft can become uncertain. You could assign the measurement to both tracks for a short time. As you gather more measurements you could then go back and change assignment based on which is more probable given the new information.

'Billions' does not begin to capture the combinatorial explosion this implies. After just a few seconds you will be out of computer memory, a bit longer and you would require every atom in the universe to represent all of the potential tracks. So a practical algorithm needs to aggressively prune tracks. This pruning takes additional compute power.

A later chapter in the book provides the modern answer to this problem, the *particle filter*, which solves the combinatorial explosion with statistical sampling. It is my favored approach for this problem, so I will not write more about the techniques I am discussing in this section. I am not entirely up to date on current research in this area, so do your own research if you are trying to solve a problem requiring to either track multiple objects or deal with multiple spurious measurements. Particle filters have their own difficulties and limitations.

I will point you to a few books and researchers. Samuel S. Blackman's "Multiple-Target Tracking with Radar Application" is the clearest written book I have read on the topic, though it is dated (1986). Yaakov Bar-Shalom has written extensively on the subject. Subhash Challa et. al "Fundamentals of Object Tracking" is a fairly modern work that covers the various approaches. This book is fairly rigorous mathematically; filters are presented as a collection of integrals representing various Bayesian formulation, and it is up to you to translate that into a working algorithm. It is readable if you have taken on all the math in this book, but not easy. Lawrence D. Stone's "Bayesian Multiple Target Tracking" treats it as a Bayesian inference problem, as I have, but is also quite theoretical, where you are blithely told to find a maximum of a complicated integral,

which in practice you will probably solve using a particle filter.

Back to our simple problem - tracking a single object with an occasional bad measurement. How should that be implemented? It's fairly straightforward; if the measurement is bad, discard it and do not call update. This will cause you to call `predict()` twice in a row, which is fine. Your uncertainty will grow, but a few missed updates generally will not cause a problem.

What cutoff value should you use for your gate? I don't know. Theory says 3 std, but practice says otherwise. You will need to experiment. Gather data, run a filter on it using various gates, and see what value gives the best results. In the next section I give you some math to evaluate filter performance. Maybe you find out you need to accept all measurements $< 4.5$ std. I watched a NASA video where they stated they used a gate around 5-6 std. it depends on your problem and data.

## 8.7 Evaluating Filter Performance

It is easy to design a Kalman filter for a simulated situation. You know how much noise you are injecting in your process model, so you specify $\mathbf{Q}$ to have the same value. You also know how much noise is in the measurement simulation, so the measurement noise matrix $\mathbf{R}$ is equally trivial to define.

In practice design is more ad hoc. Real sensors rarely perform to spec, and they rarely perform in a Gaussian manner. They are also easily fooled by environmental noise. For example, circuit noise causes voltage fluctuations which can affect the output of a sensor. Creating a process model and noise is even more difficult. Modelling an automobile is very difficult. The steering causes nonlinear behavior, tires slip, people brake and accelerate hard enough to cause tire slips, winds push the car off course. The end result is the Kalman filter is an *inexact* model of the system. This inexactness causes suboptimal behavior, which in the worst case causes the filter to diverge completely.

Because of the unknowns you will be unable to analytically compute the correct values for the filter's matrices. You will start by making the best estimate possible, and then test your filter against a wide variety of simulated and real data. Your evaluation of the performance will guide you towards what changes you need to make to the matrices. We've done some of this already - I've shown you the effect of $\mathbf{Q}$ being too large or small.

Now let's consider more analytical ways of accessing performance. If the Kalman filter is performing optimally its estimation errors (the difference between the true state and the estimated state) will have these properties:

```
1. The mean of the estimation error is zero
2. Its covariance is described by the Kalman filter's covariance matrix
```

### 8.7.1 Normalized Estimated Error Squared (NEES)

The first method is the most powerful, but only possible in simulations. If you are simulating a system you know its true value, or 'ground truth'. It is then trival to compute the error of the system at any step as the difference between ground truth ($\mathbf{x}$) and the filter's state estimate ($\hat{\mathbf{x}}$):

$$\tilde{\mathbf{x}} = \mathbf{x} - \hat{\mathbf{x}}$$

We can then define the *normalized estimated error squared* (NEES) as

$$\epsilon = \tilde{\mathbf{x}}^\mathsf{T}\mathbf{P}^{-1}\tilde{\mathbf{x}}$$

To understand this equation let's look at it if the state's dimension is one. In that case both x and P are scalars, so

$$\epsilon = \frac{x^2}{P}$$

If this is not clear, recall that if $a$ is scalar, $a^\mathsf{T} = a$ and that $a^{-1} = \frac{1}{a}$.

So as the covariance matrix gets smaller NEES gets larger for the same error. The covariance matrix is the filter's estimate of it's error, so if it is small relative to the estimation error then it is performing worse than if it is large relative to the same estimation error.

This computation gives us a scalar result. If $\mathbf{x}$ is dimension $(n \times 1)$, then the dimension of the computation is $(1 \times n)(n \times n)(n \times 1) = (1 \times 1)$. What do we do with this number?

The math is outside the scope of this book, but a random variable in the form $\tilde{\mathbf{x}}^\mathsf{T}\mathbf{P}^{-1}\tilde{\mathbf{x}}$ is said to be *chi-squared distributed with n degrees of freedom*, and thus the expected value of the sequence should be $n$. Bar-Shalom [1] has an excellent discussion of this topic.

In plain terms, take the average of all the NEES values, and they should be less then the dimension of x. Let's see that using an example from earlier in the chapter:

```
In [47]: from scipy.linalg import inv

         def NEES(xs, est_xs, Ps):
             est_err = xs - est_xs
             err = []
             for x, p in zip(est_err, Ps):
                 err.append(np.dot(x.T, inv(p)).dot(x))
             return err
```

```
In [48]: R, Q = 6., 0.02
         xs, zs = simulate_acc_system(R=R, Q=Q, count=80)
         kf2 = SecondOrderKF(R, Q, dt=1)
         est_xs, ps, _, _ = kf2.batch_filter(zs)

         nees = NEES (xs, est_xs, ps)
         eps = np.mean(nees)

         print('mean NEES is: ', eps)
         if eps < kf2.dim_x:
             print('passed')
         else:
             print('failed')
```

```
mean NEES is:  0.8893319389009477
passed
```

`NEES` is implemented FilterPy; access it with

```
from filterpy.stats import NEES
```

This is an excellent measure of the filter's performance, and should be used whenever possible, especially in production code when you need to evaluate the filter while it is running. While designing a filter I still prefer plotting the residauls as it gives me a more intuitive understand what is happening.

However, if your simulation is of limited fidelity then you need to use another approach.

### 8.7.2 Likelihood Function

In statistics the likelihood is very similar to a probability, with a subtle difference that is important to us. A *probability* is the chance of something happening - as in what is the probability that a fair die rolls 6 three times in five rolls? The *likelihood* asks the reverse question - given that a die rolled 6 three times in five rolls, what is the likelihood that the die is fair?

We first discussed the likelihood function in the **Discrete Bayes** chapter. In the context of these filters the likelihood is a measure of how likely the measurement are given the current state.

This is important to us because we have the filter output and we want to know the likelihood that it is performing optimally given the assumptions of Gaussian noise and linear behavior. If the likelihood is low we know one of our assumptions is wrong. In the **Adaptive Filtering** chapter we will learn how to make use of this information to improve the filter; here we will only learn to make the measurement.

The residual and system uncertainty of the filter is defined as

$$\mathbf{y} = \mathbf{z} - \mathbf{Hx}$$
$$\mathbf{S} = \mathbf{HPH}^\mathsf{T} + \mathbf{R}$$

Given these we can compute the likelihood function with

$$\mathcal{L} = \frac{1}{\sqrt{2\pi S}} \exp[-\frac{1}{2}\mathbf{y}^\mathsf{T}\mathbf{S}^{-1}\mathbf{y}]$$

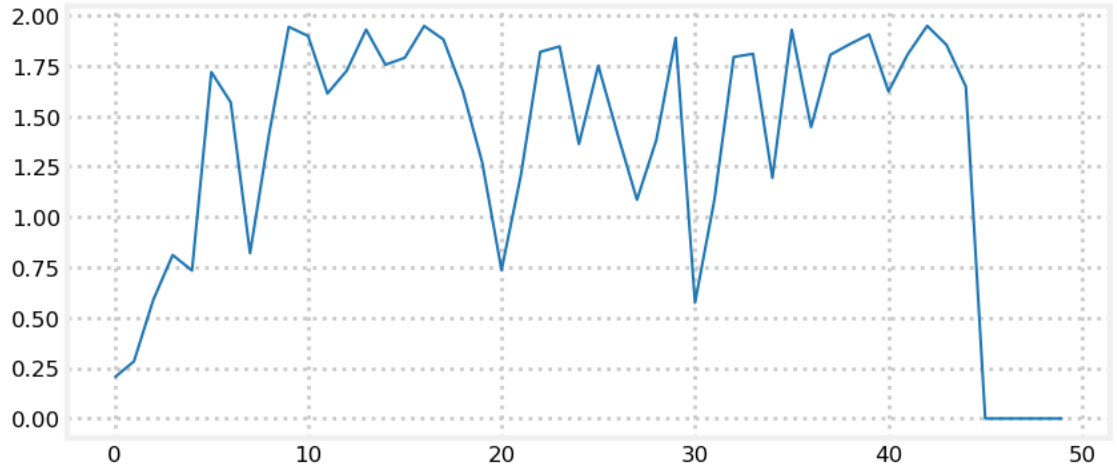
That may look complicated, but notice that the exponential is the equation for a Gaussian. This suggests an implementation of

```
from scipy.stats import multivariate_normal
hx = np.dot(H, x).flatten()
S = np.dot(H, P).dot(H.T) + R
likelihood = multivariate_normal.pdf(z.flatten(), mean=hx, cov=S)
```

In practice it happens a bit differently. Likelihoods can be difficult to deal with mathmatically. It is common to compute and use the *log-likelihood* instead, which is just the natural log of the likelihood. This has several benefits. First, the log is strictly increasing, and it reaches it's maximum value at the same point of the function it is applied to. If you want to find the maximum of a function you normally take the derivative of it; it can be difficult to find the derivative of some arbitrary function, but finding $\frac{d}{dx}log(f(x))$ is trivial, and the result is the same as $\frac{d}{dx}f(x)$. We don't use this property in this book, but it is essential when performing analysis on filters.

The likelihood and log-likelihood are computed for you when `update()` is called, and is accessible via the 'log_likelihood' and `likelihood` data attribute. Let's look at this: I'll run the filter with several measurements within expected range, and then inject measurements far from the expected values:

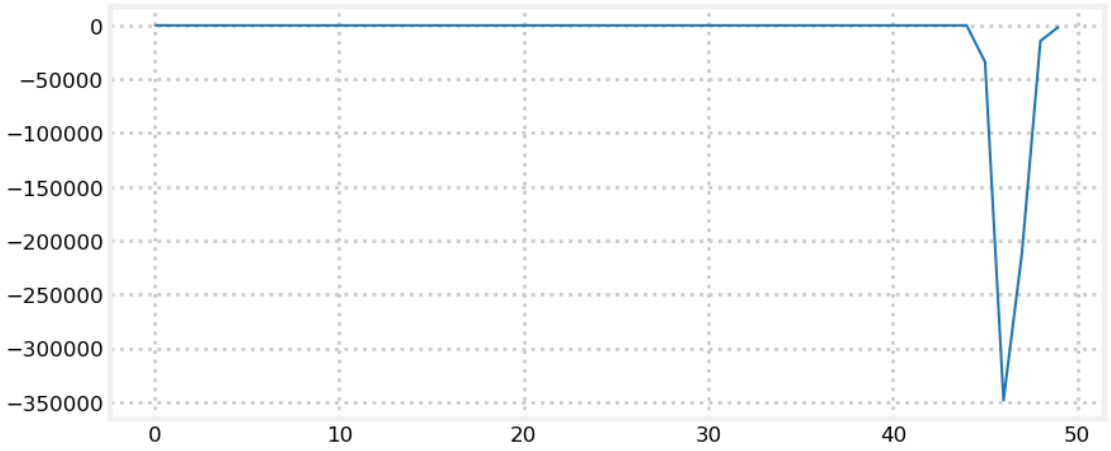
```
In [49]: R, Q = .05, 0.02
         xs, zs = simulate_acc_system(R=R, Q=Q, count=50)
         zs[-5:-1] = [100, 200, 200, 200] # bad measurements, bad!

         kf = SecondOrderKF(R, Q, dt=1, P=1)
         s = Saver(kf)
         kf.batch_filter(zs, saver=s)
         plt.plot(s.likelihood);
```



The likelihood gets larger as the filter converges during the first few iterations. After that the likelihood bounces around until it reaches the bad measurements, at which time it goes to zero, indicating that if the measurements are valid the filter is very unlikely to be optimal.

See how starkly the log-likelihood illustrates where the filter goes 'bad'.

```
In [50]: plt.plot(s.log_likelihood);
```




Why does it return to zero at the end? Think about that before reading the answer. The filter begins to adapt to the new measurements by moving the state close to the measurements. The residuals become small, and so the state and residuals agree.

## 8.8 Control Inputs

In the **Discrete Bayes** chapter I introduced the notion of using control signals to improve the filter's performance. Instead of assuming that the object continues to move as it has so far we use our knowledge of the control inputs to predict where the object is. In the **Univariate Kalman Filter** chapter we made use of the same idea. The predict method of the Kalman filter read

```
def predict(pos, movement):
    return (pos[0] + movement[0], pos[1] + movement[1])
```

In the last chapter we learned that the equation for the state prediction is:

$$\bar{\mathbf{x}} = \mathbf{F}\mathbf{x} + \mathbf{B}\mathbf{u}$$

Our state is a vector, so we need to represent the control input as a vector. Here $\mathbf{u}$ is the control input, and $\mathbf{B}$ is a matrix that transforms the control input into a change in $\mathbf{x}$. Let's consider a simple example. Suppose the state is $x = \begin{bmatrix} x & \dot{x} \end{bmatrix}$ for a robot we are controlling and the control input is commanded velocity. This gives us a control input of

$$\mathbf{u} = \begin{bmatrix} \dot{x}_{\mathtt{cmd}} \end{bmatrix}$$

For simplicity we will assume that the robot can respond instantly to changes to this input. That means that the new position and velocity after $\Delta t$ seconds will be

$$\begin{aligned} x &= x + \dot{x}_{\mathtt{cmd}} \Delta t \\ \dot{x} &= \dot{x}_{\mathtt{cmd}} \end{aligned}$$

We need to represent this set of equations in the form $\bar{\mathbf{x}} = \mathbf{Fx} + \mathbf{Bu}$.

I will use the $\mathbf{Fx}$ term to extract the $x$ for the top equation, and the $\mathbf{Bu}$ term for the rest, like so:

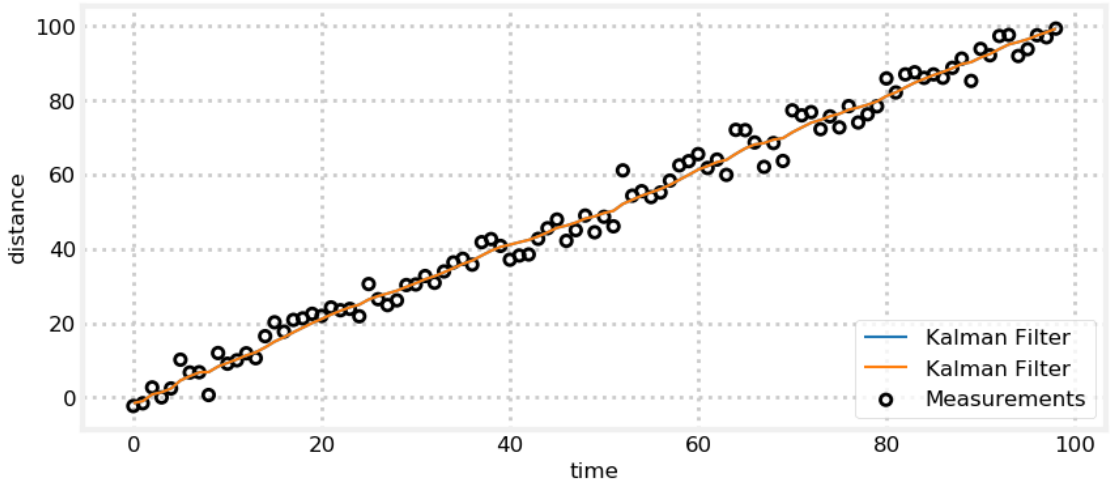
$$\begin{bmatrix} x \\ \dot{x} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix} + \begin{bmatrix} \Delta t \\ 1 \end{bmatrix} \begin{bmatrix} \dot{x}_{\mathtt{cmd}} \end{bmatrix}$$

This is a simplification; typical control inputs are changes to steering angle and changes in acceleration. This introduces nonlinearities which we will learn to deal with in a later chapter.

The rest of the Kalman filter will be designed as normal. You've seen this several times by now, so without further comment here is an example.

```
In [51]: dt = 1.
         R = 3.
         kf = KalmanFilter(dim_x=2, dim_z=1, dim_u = 1)
         kf.P *= 10
         kf.R *= R
         kf.Q = Q_discrete_white_noise(2, dt, 0.1)
         kf.F = np.array([[1., 0], [0., 0.]])
         kf.B = np.array([[dt], [ 1.]])
         kf.H = np.array([[1., 0]])
         print(kf.P)

         zs = [i + randn()*R for i in range(1, 100)]
         xs = []
         cmd_velocity = np.array([1.])
         for z in zs:
             kf.predict(u=cmd_velocity)
             kf.update(z)
             xs.append(kf.x[0])

         plt.plot(xs, label='Kalman Filter')
         plot_measurements(zs)
         plt.xlabel('time')
         plt.legend(loc=4)
         plt.ylabel('distance');
```

```
[[10.  0.]
 [ 0. 10.]]
```



## 8.9 Sensor Fusion

Early in the g-h filter chapter we discussed designing a filter for two scales, one accurate and one inaccurate. We determined that we should always include the information from the inaccurate filter - we should never discard any information. So consider a situation where we have two sensors measuring the system. How shall we incorporate that into our Kalman filter?

Suppose we have a train or cart on a railway. It has a sensor attached to the wheels counting revolutions, which can be converted to a distance along the track. Then, suppose we have a GPS-like sensor which I'll call a 'position sensor' mounted to the train which reports position. I'll explain why I don't just use a GPS in the next section. Thus, we have two measurements, both reporting position along the track. Suppose further that the accuracy of the wheel sensor is 1m, and the accuracy of the position sensor is 10m. How do we combine these two measurements into one filter? This may seem quite contrived, but aircraft use sensor fusion to fuse the measurements from sensors such as a GPS, INS, Doppler radar, VOR, the airspeed indicator, and more.

Kalman filters for inertial systems are very difficult, but fusing data from two or more sensors providing measurements of the same state variable (such as position) is quite easy. The relevant matrix is the measurement matrix $\mathbf{H}$. Recall that this matrix tells us how to convert from the Kalman filter's state $\mathbf{x}$ to a measurement $\mathbf{z}$. Suppose that we decide that our Kalman filter state should contain the position and velocity of the train, so that

$$\mathbf{x} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

We have two measurements for position, so we will define the measurement vector to be a vector of the measurements from the wheel and the position sensor.

$$\mathbf{z} = \begin{bmatrix} z_{wheel} \\ z_{ps} \end{bmatrix}$$

So we have to design the matrix $\mathbf{H}$ to convert $\mathbf{x}$ to $\mathbf{z}$ . They are both positions, so the conversion is nothing more than multiplying by one:

$$\begin{bmatrix} z_{wheel} \\ z_{ps} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

To make it clearer, suppose that the wheel reports not position but the number of rotations of the wheels, where 1 revolution yields 2 meters of travel. In that case we would write

$$\begin{bmatrix} z_{rot} \\ z_{ps} \end{bmatrix} = \begin{bmatrix} 0.5 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

Now we have to design the measurement noise matrix $\mathbf{R}$. Suppose that the measurement variance for the position is twice the variance of the wheel, and the standard deviation of the wheel is 1.5 meters. That gives us

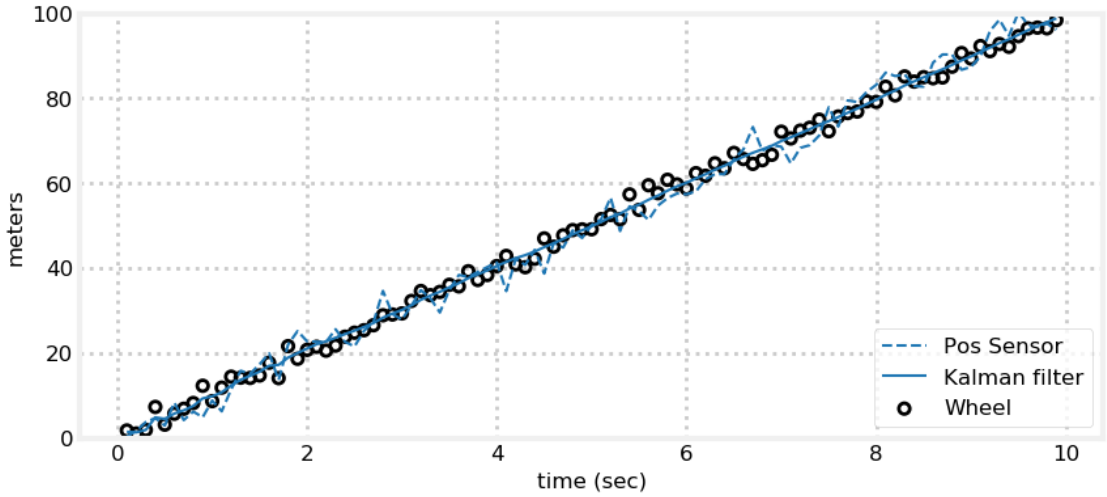
$$\begin{aligned} \sigma_{wheel} &= 1.5 \\ \sigma^2_{wheel} &= 2.25 \\ \sigma_{ps} &= 1.5 * 2 = 3 \\ \sigma^2_{ps} &= 9. \end{aligned}$$

That is pretty much our Kalman filter design. We need to design for $\mathbf{Q}$, but that is invariant to whether we are doing sensor fusion or not, so I will just choose some arbitrary value.

So let's run a simulation of this design. I will assume a velocity of 10 m/s with an update rate of 0.1 seconds.

```
In [52]: from numpy import array, asarray
         import numpy.random as random

         def fusion_test(wheel_sigma, ps_sigma, do_plot=True):
             dt = 0.1
             kf = KalmanFilter(dim_x=2, dim_z=2)

             kf.F = array([[1., dt], [0., 1.]])
             kf.H = array([[1., 0.], [1., 0.]])
             kf.x = array([[0.], [1.]])
             kf.Q *= array([[(dt**3)/3, (dt**2)/2],
                            [(dt**2)/2,  dt      ]]) * 0.02
             kf.P *= 100
             kf.R[0, 0] = wheel_sigma**2
             kf.R[1, 1] = ps_sigma**2
             s = Saver(kf)

             random.seed(1123)
             for i in range(1, 100):
                 m0 = i + randn()*wheel_sigma
                 m1 = i + randn()*ps_sigma
                 kf.predict()
                 kf.update(array([[m0], [m1]]))
                 s.save()
             s.to_array()
             print('fusion std: {:.3f}'.format(np.std(s.y[:, 0])))
             if do_plot:
                 ts = np.arange(0.1, 10, .1)
                 plot_measurements(ts, s.z[:, 0], label='Wheel')
                 plt.plot(ts, s.z[:, 1], ls='--', label='Pos Sensor')
                 plot_filter(ts, s.x[:, 0], label='Kalman filter')
                 plt.legend(loc=4)
                 plt.ylim(0, 100)
                 set_labels(x='time (sec)', y='meters')

         fusion_test(1.5, 3.0)
```

```
fusion std: 1.647
```




We can see the result of the Kalman filter in blue.

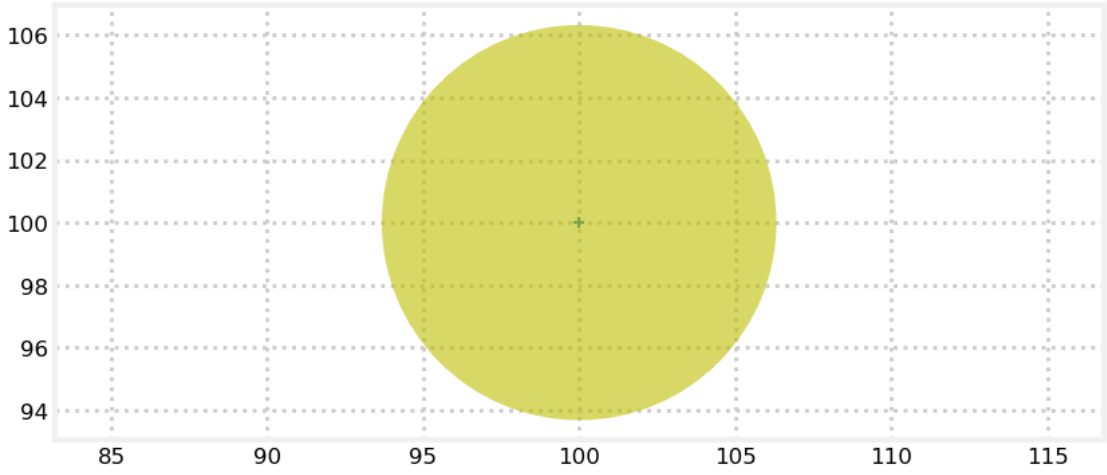
It may be somewhat difficult to understand the previous example at an intuitive level. Let's look at a different problem. Suppose we are tracking an object in 2D space, and have two radar systems at different positions. Each radar system gives us a range and bearing to the target. How do the readings from each data affect the results?

This is a nonlinear problem because we need to use trigonometry to compute coordinates from a range and bearing, and we have not yet learned how to solve nonlinear problems with Kalman filters. So for this problem ignore the code that I use and just concentrate on the charts that the code outputs. We will revisit this problem in subsequent chapters and learn how to write this code.

I will position the target at (100, 100). The first radar will be at (50, 50), and the second radar at (150, 50). This will cause the first radar to measure a bearing of 45 degrees, and the second will report 135 degrees.

I will create the Kalman filter first, and then plot its initial covariance matrix. I am using an **unscented Kalman filter**, which is covered in a later chapter.

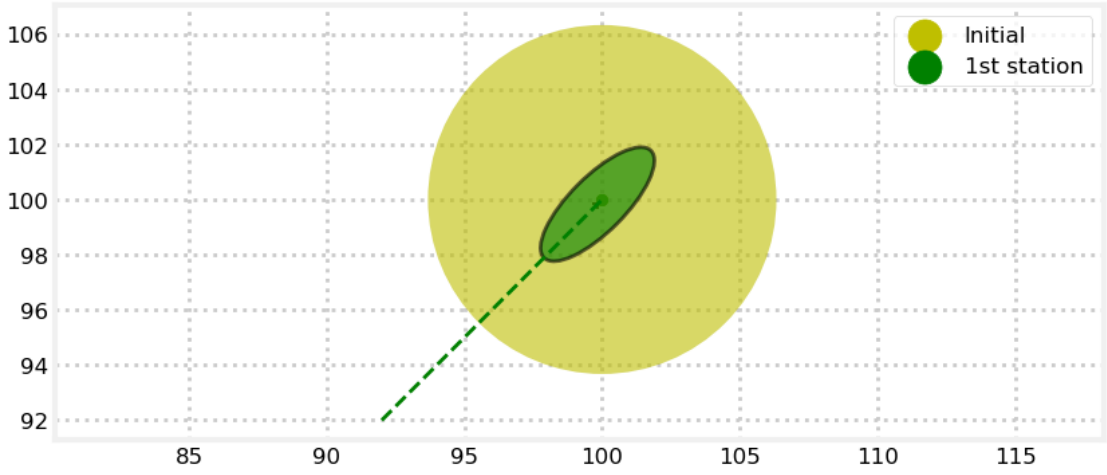
```
In [53]: from kf_book.kf_design_internal import sensor_fusion_kf

         kf = sensor_fusion_kf()
         x0, p0 = kf.x.copy(), kf.P.copy()
         plot_covariance_ellipse(x0, p0, fc='y', ec=None, alpha=0.6)
```



We are equally uncertain about the position in x and y, so the covariance is circular.

Now we will update the Kalman filter with a reading from the first radar. I will set the standard deviation of the bearing error at 0.5°, and the standard deviation of the distance error at 3.

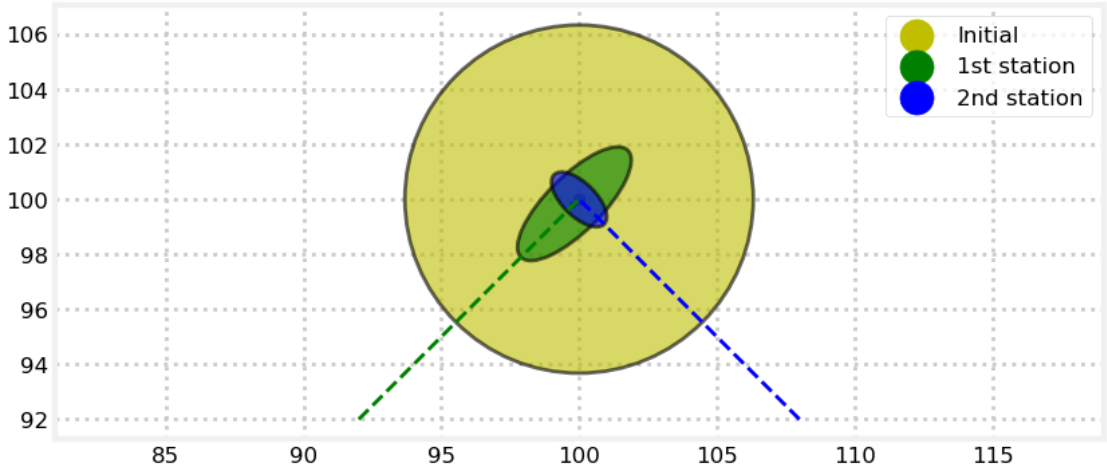
```
In [54]: from math import radians
         from kf_book.kf_design_internal import sensor_fusion_kf, set_radar_pos

         # set the error of the radar's bearing and distance
         kf.R[0, 0] = radians (.5)**2
         kf.R[1, 1] = 3.**2

         # compute position and covariance from first radar station
         set_radar_pos((50, 50))
         dist = (50**2 + 50**2) ** 0.5
         kf.predict()
         kf.update([radians(45), dist])

         # plot the results
         x1, p1 = kf.x.copy(), kf.P.copy()

         plot_covariance_ellipse(x0, p0, fc='y', ec=None, alpha=0.6)
         plot_covariance_ellipse(x1, p1, fc='g', ec='k', alpha=0.6)

         plt.scatter([100], [100], c='y', label='Initial')
         plt.scatter([100], [100], c='g', label='1st station')
         plt.legend(scatterpoints=1, markerscale=3)
         plt.plot([92, 100], [92, 100], c='g', lw=2, ls='--');
```




We can see the effect of the errors on the geometry of the problem. The radar station is to the lower left of the target. The bearing measurement is extremely accurate at $\sigma = 0.5°$, but the distance error is inaccurate at $\sigma = 3$. I've shown the radar reading with the dotted green line. We can easily see the effect of the accurate bearing and inaccurate distance in the shape of the covariance ellipse.

Now we can incorporate the second radar station's measurement. The second radar is at (150,50), which is below and to the right of the target. Before you go on, think about how you think the covariance will change when we incorporate this new reading.

```
In [55]: # compute position and covariance from second radar station
         set_radar_pos((150, 50))
         kf.predict()
         kf.update([radians(135), dist])

         plot_covariance_ellipse(x0, p0, fc='y', ec='k', alpha=0.6)
         plot_covariance_ellipse(x1, p1, fc='g', ec='k', alpha=0.6)
         plot_covariance_ellipse(kf.x, kf.P, fc='b', ec='k', alpha=0.6)

         plt.scatter([100], [100], c='y', label='Initial')
         plt.scatter([100], [100], c='g', label='1st station')
         plt.scatter([100], [100], c='b', label='2nd station')
         plt.legend(scatterpoints=1, markerscale=3)
         plt.plot([92, 100], [92, 100], c='g', lw=2, ls='--')
         plt.plot([108, 100], [92, 100], c='b', lw=2, ls='--');
```




We can see how the second radar measurement altered the covariance. The angle to the target is orthogonal to the first radar station, so the effects of the error in the bearing and range are swapped. So the angle of the covariance matrix switches to match the direction to the second station. It is important to note that the direction did not merely change; the size of the covariance matrix became much smaller as well.

The covariance will always incorporate all of the information available, including the effects of the geometry of the problem. This formulation makes it particularly easy to see what is happening, but the same thing occurs if one sensor gives you position and a second sensor gives you velocity, or if two sensors provide measurements of position.

One final thing before we move on: sensor fusion is a vast topic, and my coverage is simplistic to the point of being misleading. For example, GPS uses iterated least squares to determine the position from a set of pseudorange readings from the satellites without using a Kalman filter. I cover this topic in the supporting notebook **Iterative Least Squares for Sensor Fusion**

That is the usual but not exclusive way this computation is done in GPS receivers. If you are a hobbyist my coverage may get you started. A commercial grade filter requires very careful design of the fusion process. That is the topic of several books, and you will have to further your education by finding one that covers your domain.

### 8.9.1 Exercise: Can you Filter GPS outputs?

In the section above I have you apply a Kalman filter to a 'GPS-like' sensor. Can you apply a Kalman filter to the output of a commercial Kalman filter? In other words, will the output of your filter be better than, worse than, or equal to the GPS's output?

#### Solution

Commercial GPS's have a Kalman filter built into them, and their output is the filtered estimate created by that filter. So, suppose you have a steady stream of output from the GPS consisting of a position and position error. Can you not pass those two pieces of data into your own filter?

Well, what are the characteristics of that data stream, and more importantly, what are the fundamental requirements of the input to the Kalman filter?

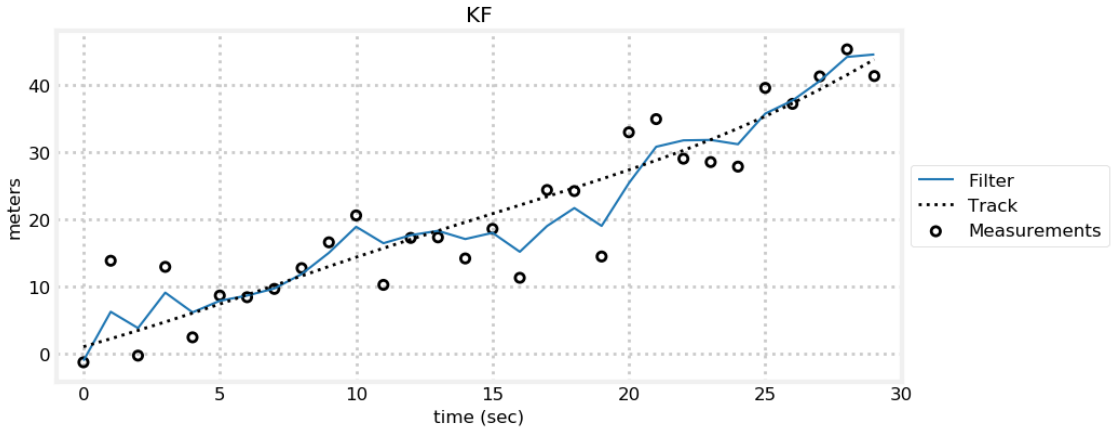
Inputs to the Kalman filter must be *Gaussian* and *time independent*. This is because we imposed the requirement of the Markov property: the current state is dependent only on the previous state and current inputs. This makes the recursive form of the filter possible. The output of the GPS is *time dependent* because the filter bases its current estimate on the recursive estimates of all previous measurements. Hence, the signal is not white, it is not time independent, and if you pass that data into a Kalman filter you have violated the mathematical requirements of the filter. So, the answer is no, you cannot get better estimates by running a KF on the output of a commercial GPS.

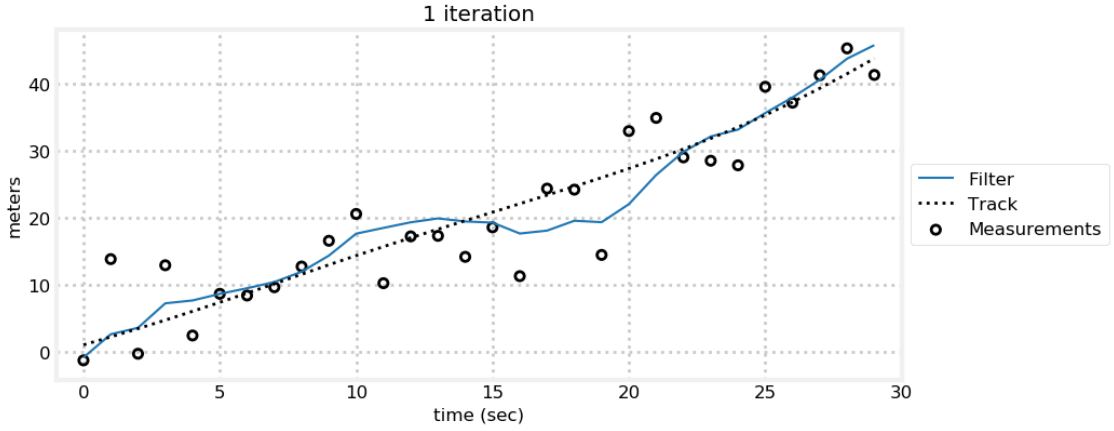
Another way to think of it is that Kalman filters are optimal in a least squares sense. There is no way to take an optimal solution, pass it through a filter, any filter, and get a 'more optimal' answer because it is a logical impossibility. At best the signal will be unchanged, in which case it will still be optimal, or it will be changed, and hence no longer optimal.

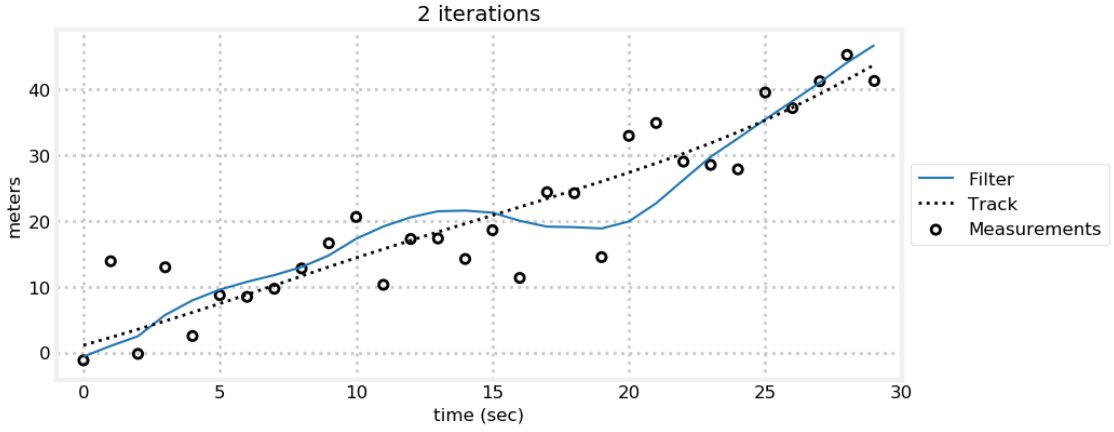
This is a difficult problem that hobbyists face when trying to integrate GPS, IMU's and other off the shelf sensors.

Let's look at the effect. A commercial GPS reports position, and an estimated error range. The estimated error just comes from the Kalman filter's $\mathbf{P}$ matrix. So let's filter some noisy data, take the filtered output as the new noisy input to the filter, and see what the result is. In other words, $\mathbf{x}$ will supply the $\mathbf{z}$ input, and $\mathbf{P}$ will supply the measurement covariance $\mathbf{R}$. To exaggerate the effects somewhat to make them more obvious I will plot the effects of doing this one, and then a second time. The second iteration doesn't make any 'sense' (no one would try that), it just helps me illustrate a point. First, the code and plots.

```
In [56]: np.random.seed(124)
         R = 5.
         xs, zs = simulate_acc_system(R=R, Q=Q, count=30)

         kf0 = SecondOrderKF(R, Q, dt=1)
         kf1 = SecondOrderKF(R, Q, dt=1)
         kf2 = SecondOrderKF(R, Q, dt=1)

         # Filter measurements
         fxs0, ps0, _, _ = kf0.batch_filter(zs)

         # filter twice more, using the state as the input
         fxs1, ps1, _, _ = kf1.batch_filter(fxs0[:, 0])
         fxs2, _, _, _ = kf2.batch_filter(fxs1[:, 0])

         plot_kf_output(xs, fxs0, zs, 'KF', False)
         plot_kf_output(xs, fxs1, zs, '1 iteration', False)
         plot_kf_output(xs, fxs2, zs, '2 iterations', False)
         R,Q
```


KF
meters
time (sec)
Filter
Track
Measurements


1 iteration
meters
time (sec)
Filter
Track
Measurements

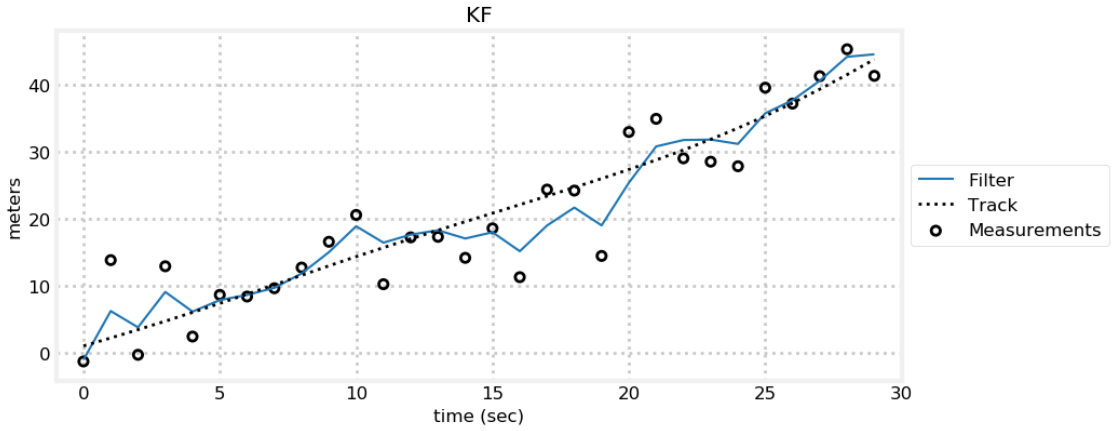

2 iterations
meters
time (sec)
Filter
Track
Measurements

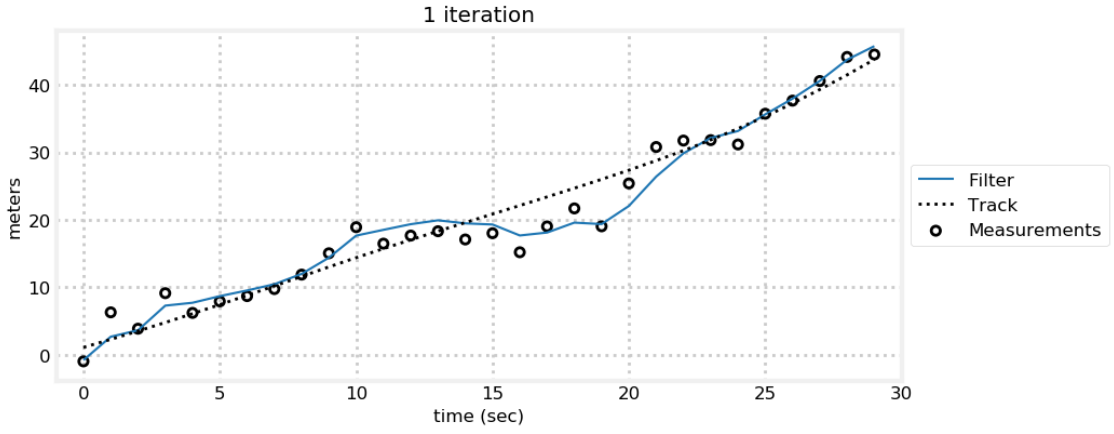
Out[56]: (5.0, 0.02)

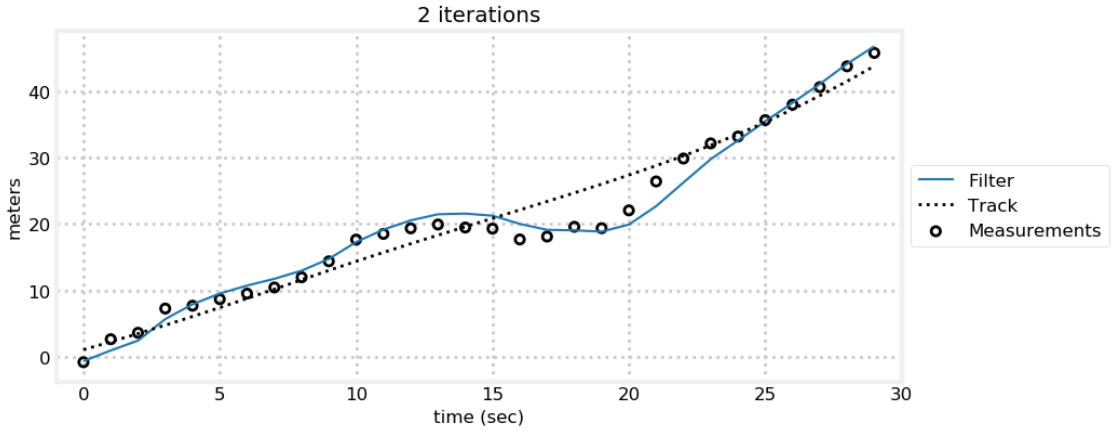
We see that the filtered output of the reprocessed signal is smoother, but it also diverges from the track. What is happening? Recall that the Kalman filter requires that the signal not be time correlated. However the output of the Kalman filter *is* time correlated because it incorporates all previous measurements into its estimate for this time period. So look at the last graph, for 2 iterations. The measurements start with several peaks that are larger than the track. This is 'remembered' (that is vague terminology, but I am trying to avoid the math) by the filter, and it has started to compute that the object is above the track. Later, at around 13 seconds we have a period where the measurements all happen to be below the track. This also gets incorporated into the memory of the filter, and the iterated output diverges far below the track.

Now let's look at this in a different way. The iterated output is *not* using $\mathbf{z}$ as the measurement, but the output of the previous Kalman filter estimate. So I will plot the output of the filter against the previous filter's output.

```
In [57]: plot_kf_output(xs, fxs0, zs, title='KF', aspect_equal=False)
         plot_kf_output(xs, fxs1, fxs0[:, 0], '1 iteration', False)
         plot_kf_output(xs, fxs2, fxs1[:, 0], '2 iterations', False)
```






I hope the problem with this approach is now apparent. In the bottom graph we can see that the KF is tracking the imperfect estimates of the previous filter, and incorporating delay into the signal as well due to the memory of the previous measurements being incorporated into the signal.

### 8.9.2 Exercise: Prove that the position sensor improves the filter

Devise a way to prove that fusing the position sensor and wheel measurements yields a better result than using the wheel alone.

#### Solution 1

Force the Kalman filter to disregard the position sensor measurement by setting the measurement noise for the position sensor to a near infinite value. Re-run the filter and observe the standard deviation of the residual.

```
In [58]: fusion_test(1.5, 3.0, do_plot=False)
         fusion_test(1.5, 1.e40, do_plot=False)

fusion std: 1.647
fusion std: 1.647
```

Here we can see the error in the filter where the position sensor measurement is almost entirely ignored is greater than that where it is used.

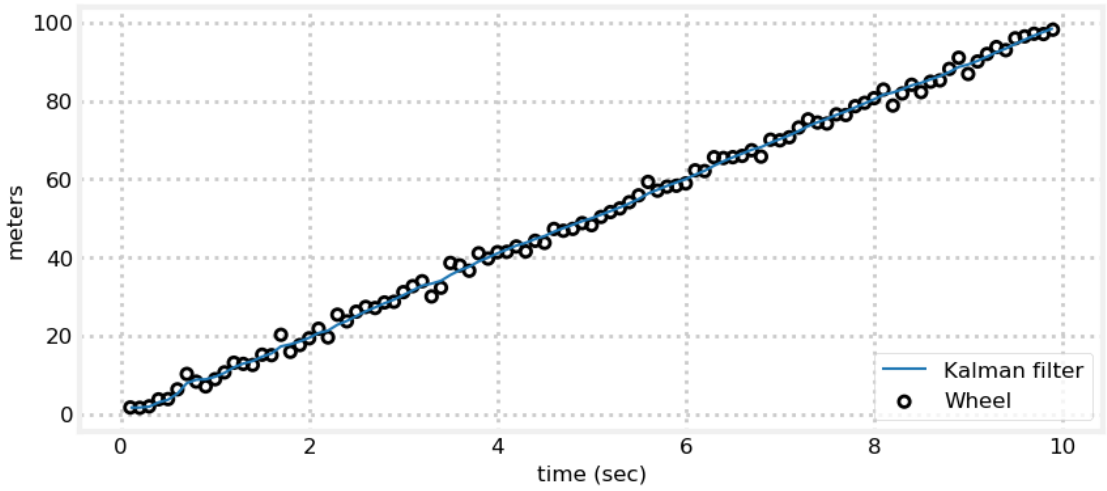
#### Solution 2

This is more work, but we can write a Kalman filter that only takes one measurement.

```
In [59]: dt = 0.1
         wheel_sigma = 1.5
         kf = KalmanFilter(dim_x=2, dim_z=1)
         kf.F = array([[1., dt], [0., 1.]])
         kf.H = array([[1., 0.]])
         kf.x = array([[0.], [1.]])
         kf.Q *= 0.01
         kf.P *= 100
         kf.R[0, 0] = wheel_sigma**2
```

```
random.seed(1123)
nom = range(1, 100)
zs = np.array([i + randn()*wheel_sigma for i in nom])
xs, _, _, _ = kf.batch_filter(zs)
ts = np.arange(0.1, 10, .1)

res = nom - xs[:, 0, 0]
print('std: {:.3f}'.format(np.std(res)))

plot_filter(ts, xs[:, 0], label='Kalman filter')
plot_measurements(ts, zs, label='Wheel')
set_labels(x='time (sec)', y='meters')
plt.legend(loc=4);
```

```
std: 0.523
```




On this run I got a standard deviation of 0.523 vs the value of 0.391 for the fused measurements.

## 8.10 Nonstationary Processes

So far we have assumed that the various matrices in the Kalman filter are *stationary* - nonchanging over time. For example, in the robot tracking section we assumed that $\Delta t = 1.0$ seconds, and designed the state transition matrix to be

$$\mathbf{F} = \begin{bmatrix} 1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

But what if our data rate changes in some unpredictable manner? Or what if we have two sensors, each running at a different rate? What if the error of the measurement changes?

Handling this is easy; you just alter the Kalman filter matrices to reflect the current situation. Let's go back to our dog tracking problem and assume that the data input is somewhat sporadic. For this problem we designed

$$\mathbf{x} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}^-$$

$$\mathbf{F} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix}$$

and set the Kalman filter variable `F` during initialization like so:

```
dt = 0.1
kf.F = np.array([[1, dt],
                 [0, 1]])
```

How would we handle $\Delta t$ changing for each measurement? It's easy - just modify the relevant matrices. In this case `F` is variant, so we will need to update this inside the update/predict loop. `Q` is also dependent on time, so it must be assigned during each loop as well. Here is an example of how we might code this:

```
In [60]: kf = KalmanFilter(dim_x=2, dim_z=1)
         kf.x = array([0., 1.])
         kf.H = array([[1, 0]])
         kf.P = np.eye(2) * 50
         kf.R = np.eye(1)
         q_var = 0.02

         # measurement tuple: (value, time)
         zs = [(1., 1.),  (2., 1.1), (3., 0.9), (4.1, 1.23), (5.01, 0.97)]
         for z, dt in zs:
             kf.F = array([[1, dt],
                           [0, 1]])
             kf.Q = Q_discrete_white_noise(dim=2, dt=dt, var=q_var)
             kf.predict()
             kf.update(z)
             print(kf.x)
```

```
[1. 1.]
[2.003 0.916]
[2.963 0.997]
[4.124 0.97 ]
[5.032 0.959]
```

### 8.10.1 Sensor fusion: Different Data Rates

It is rare that two different sensor classes output data at the same rate. Assume that the position sensor produces an update at 3 Hz, and the wheel updates at 7 Hz. Further assume that the timing is not precise - there is a bit of jitter such that a measurement can occur a bit before or after the predicted time. Let me further complicate the situation by having the wheel provide velocity estimates rather than position estimates.

We can do this by waiting for a data packet from either sensor. When we get it, we determine the amount of time that has passed since the last update. We then need to modify the affected matrices. Both $\mathbf{F}$ and $\mathbf{Q}$ contain a time term $\Delta t$ so we will need to adjust these on every innovation.

The measurement changes each time, so we will have to modify $\mathbf{H}$ and $\mathbf{R}$. The position sensor alters the position element of $\mathbf{x}$, so we assign:

$$\mathbf{H} = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

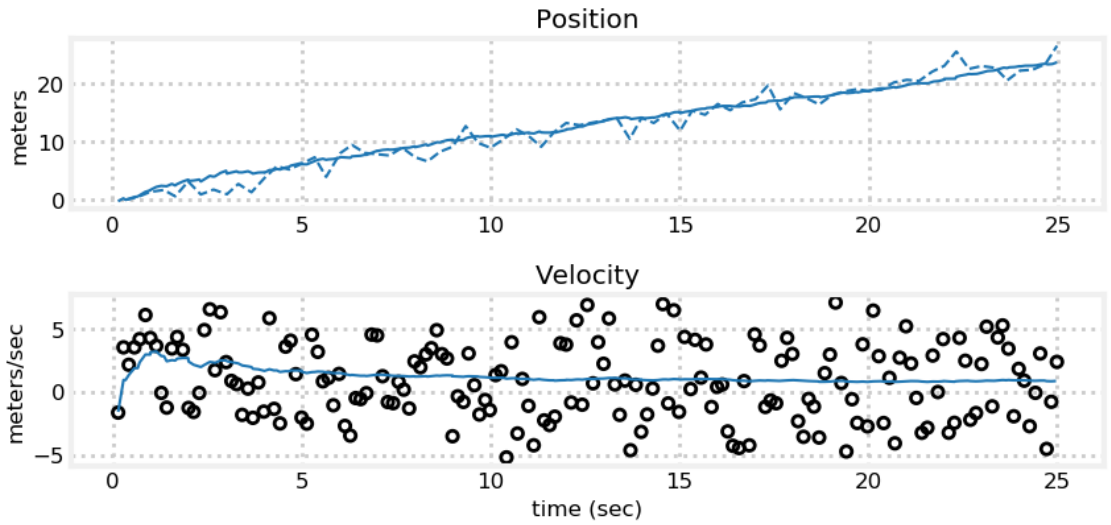
$$\mathbf{R} = \sigma_{ps}^2$$

The wheel sensor alters the velocity element of $\mathbf{x}$, so we assign:

$$\mathbf{H} = \begin{bmatrix} 0 & 1 \end{bmatrix}$$
$$\mathbf{R} = \sigma^2_{wheel}$$

```
In [61]: def gen_sensor_data(t, ps_std, wheel_std):
             # generate simulated sensor data
             pos_data, vel_data = [], []
             dt = 0.
             for i in range(t*3):
                 dt += 1/3.
                 t_i = dt + randn() * .01 # time jitter
                 pos_data.append([t_i, t_i + randn()*ps_std])

             dt = 0.
             for i in range(t*7):
                 dt += 1/7.
                 t_i = dt + randn() * .006 # time jitter
                 vel_data.append([t_i, 1. + randn()*wheel_std])
             return pos_data, vel_data


         def plot_fusion(xs, ts, zs_ps, zs_wheel):
             xs = np.array(xs)
             plt.subplot(211)
             plt.plot(zs_ps[:, 0], zs_ps[:, 1], ls='--', label='Pos Sensor')
             plot_filter(xs=ts, ys=xs[:, 0], label='Kalman filter')
             set_labels(title='Position', y='meters',)

             plt.subplot(212)
             plot_measurements(zs_wheel[:, 0], zs_wheel[:, 1],  label='Wheel')
             plot_filter(xs=ts, ys=xs[:, 1], label='Kalman filter')
             set_labels('Velocity', 'time (sec)', 'meters/sec')


         def fusion_test(pos_data, vel_data, wheel_std, ps_std):
             kf = KalmanFilter(dim_x=2, dim_z=1)
             kf.F = array([[1., 1.], [0., 1.]])
             kf.H = array([[1., 0.], [1., 0.]])
             kf.x = array([[0.], [1.]])
             kf.P *= 100

             xs, ts = [],  []

             # copy data for plotting
             zs_wheel = np.array(vel_data)
             zs_ps = np.array(pos_data)

             last_t = 0
             while len(pos_data) > 0 and len(vel_data) > 0:
                 if pos_data[0][0] < vel_data[0][0]:
                     t, z = pos_data.pop(0)
                     dt = t - last_t
                     last_t = t
```

```
            kf.H = np.array([[1., 0.]])
            kf.R[0,0] = ps_std**2
        else:
            t, z = vel_data.pop(0)
            dt = t - last_t
            last_t = t

            kf.H = np.array([[0., 1.]])
            kf.R[0,0] = wheel_std**2

        kf.F[0,1] = dt
        kf.Q = Q_discrete_white_noise(2, dt=dt, var=.02)
        kf.predict()
        kf.update(np.array([z]))

        xs.append(kf.x.T[0])
        ts.append(t)
    plot_fusion(xs, ts, zs_ps, zs_wheel)

random.seed(1123)
pos_data, vel_data = gen_sensor_data(25, 1.5, 3.0)
fusion_test(pos_data, vel_data, 1.5, 3.0);
```




## 8.11 Tracking a Ball

Now let's turn our attention to a situation where the physics of the object that we are tracking is constrained. A ball thrown in a vacuum must obey Newtonian laws. In a constant gravitational field it will travel in a parabola. I will assume you are familiar with the derivation of the formula:

$$y = \frac{g}{2}t^2 + v_{y0}t + y_0$$
$$x = v_{x0}t + x_0$$

where $g$ is the gravitational constant, $t$ is time, $v_{x0}$ and $v_{y0}$ are the initial velocities in the x and y plane. If the ball is thrown with an initial velocity of $v$ at angle $\theta$ above the horizon, we can compute $v_{x0}$ and $v_{y0}$ as

$$v_{x0} = v \cos\theta$$
$$v_{y0} = v \sin\theta$$

Because we don't have real data we will start by writing a simulator for a ball. As always, we add a noise term independent of time so we can simulate noisy sensors.

```
In [62]: from math import radians, sin, cos
         import math

         def rk4(y, x, dx, f):
             """computes 4th order Runge-Kutta for dy/dx.
             y is the initial value for y
             x is the initial value for x
             dx is the difference in x (e.g. the time step)
             f is a callable function (y, x) that you supply to
               compute dy/dx for the specified values.
             """

             k1 = dx * f(y, x)
             k2 = dx * f(y + 0.5*k1, x + 0.5*dx)
             k3 = dx * f(y + 0.5*k2, x + 0.5*dx)
             k4 = dx * f(y + k3, x + dx)

             return y + (k1 + 2*k2 + 2*k3 + k4) / 6.

         def fx(x,t):
             return fx.vel

         def fy(y,t):
             return fy.vel - 9.8*t


         class BallTrajectory2D(object):
             def __init__(self, x0, y0, velocity,
                          theta_deg=0.,
                          g=9.8,
                          noise=[0.0, 0.0]):
                 self.x = x0
                 self.y = y0
                 self.t = 0
                 theta = math.radians(theta_deg)
                 fx.vel = math.cos(theta) * velocity
                 fy.vel = math.sin(theta) * velocity
                 self.g = g
                 self.noise = noise


             def step(self, dt):
                 self.x = rk4(self.x, self.t, dt, fx)
                 self.y = rk4(self.y, self.t, dt, fy)
                 self.t += dt
```

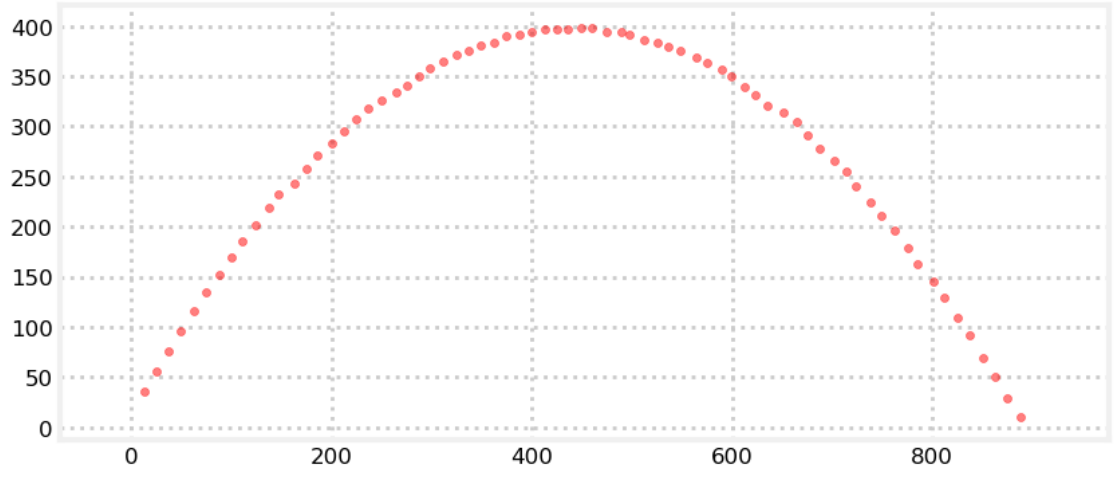
```
            return (self.x + randn()*self.noise[0],
                    self.y + randn()*self.noise[1])
```

So to create a trajectory starting at (0, 15) with a velocity of 100 m/s and an angle of 60° we would write:

```
traj = BallTrajectory2D(x0=0, y0=15, velocity=100, theta_deg=60)
```

and then call `traj.step(t)` for each time step. Let's test this

```
In [63]: def test_ball_vacuum(noise):
             y = 15
             x = 0
             ball = BallTrajectory2D(x0=x, y0=y,
                                     theta_deg=60., velocity=100.,
                                     noise=noise)
             t = 0
             dt = 0.25
             while y >= 0:
                 x, y = ball.step(dt)
                 t += dt
                 if y >= 0:
                     plt.scatter(x, y, color='r', marker='.', s=75, alpha=0.5)

             plt.axis('equal');

         #test_ball_vacuum([0, 0]) # plot ideal ball position
         test_ball_vacuum([1, 1]) # plot with noise
```




This looks reasonable, so let's continue (exercise for the reader: validate this simulation more robustly).

### 8.11.1 Choose the State Variables

We might think to use the same state variables as used for tracking the dog. However, this will not work. Recall that the Kalman filter state transition must be written as $\mathbf{x} = \mathbf{Fx} + \mathbf{Bu}$, which means we must calculate the current state from the previous state. Our assumption is that the ball is traveling in a vacuum,

so the velocity in x is a constant, and the acceleration in y is solely due to the gravitational constant $g$. We can discretize the Newtonian equations using the well known Euler method in terms of $\Delta t$ are:

$$\begin{aligned} x_t &= x_{t-1} + v_{x(t-1)}\Delta t \\ v_{xt} &= v_{x(t-1)} \\ y_t &= y_{t-1} + v_{y(t-1)}\Delta t \\ v_{yt} &= -g\Delta t + v_{y(t-1)} \end{aligned}$$

**sidebar**: *Euler's method integrates a differential equation stepwise by assuming the slope (derivative) is constant at time t. In this case the derivative of the position is velocity. At each time step $\Delta t$ we assume a constant velocity, compute the new position, and then update the velocity for the next time step. There are more accurate methods, such as Runge-Kutta available to us, but because we are updating the state with a measurement in each step Euler's method is very accurate. If you need to use Runge-Kutta you will have to write your own* `predict()` *function which computes the state transition for* $\mathbf{x}$*, and then uses the normal Kalman filter equation* $\mathbf{P} = \mathbf{FPF}^\mathsf{T} + \mathbf{Q}$ *to update the covariance matrix.*

This implies that we need to incorporate acceleration for $y$ into the Kalman filter, but not for $x$. This suggests the following state variable.

$$\mathbf{x} = \begin{bmatrix} x & \dot{x} & y & \dot{y} & \ddot{y} \end{bmatrix}^\mathsf{T}$$

However, the acceleration is due to gravity, which is a constant. Instead of asking the Kalman filter to track a constant we can treat gravity as what it really is - a control input. In other words, gravity is a force that alters the behavior of the system in a known way, and it is applied throughout the flight of the ball.

The equation for the state prediction is $\mathbf{x} = \mathbf{Fx} + \mathbf{Bu}$. $\mathbf{Fx}$ is the familiar state transition function which we will use to model the position and velocity of the ball. The vector $\mathbf{u}$ lets you specify a control input into the filter. For a car the control input will be things such as the amount the accelerator and brake are pressed, the position of the steering wheel, and so on. For our ball the control input will be gravity. The matrix $\mathbf{B}$ models how the control inputs affect the behavior of the system. Again, for a car $\mathbf{B}$ will convert the inputs of the brake and accelerator into changes of velocity, and the input of the steering wheel into a different position and heading. For our ball tracking problem it will compute the velocity change due to gravity. We will go into the details of that soon. For now, we design the state variable to be

$$\mathbf{x} = \begin{bmatrix} x & \dot{x} & y & \dot{y} \end{bmatrix}^\mathsf{T}$$

### 8.11.2 Design State Transition Function

Our next step is to design the state transition function. Recall that the state transition function is implemented as a matrix $\mathbf{F}$ that we multiply with the previous state of our system to get the next state, or prior $\bar{\mathbf{x}} = \mathbf{Fx}$.

I will not belabor this as it is very similar to the 1-D case we did in the previous chapter. Our state equations for position and velocity would be:

$$\begin{aligned} \bar{x} &= (1 * x) + (\Delta t * v_x) + (0 * y) + (0 * v_y) \\ \bar{v}_x &= (0 * x) + (1 * v_x) + (0 * y) + (0 * v_y) \\ \bar{y} &= (0 * x) + (0 * v_x) + (1 * y) + (\Delta t * v_y) \\ \bar{v}_y &= (0 * x) + (0 * v_x) + (0 * y) + (1 * v_y) \end{aligned}$$

Note that none of the terms include $g$, the gravitational constant. As I explained in the previous function we will account for gravity using the control input of the Kalman filter. In matrix form we write this as:

$$\mathbf{F} = \begin{bmatrix} 1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

### 8.11.3 Design the Control Input Function

We will use the control input to account for the force of gravity. The term $\mathbf{Bu}$ is added to $\mathbf{x}$ to account for how much $\mathbf{x}$ changes due to gravity. We can say that $\mathbf{Bu}$ contains $\begin{bmatrix}\Delta x_g & \Delta \dot{x_g} & \Delta y_g & \Delta \dot{y_g}\end{bmatrix}^\mathsf{T}$.

If we look at the discretized equations we see that gravity only affects the velocity for $y$.

$$\begin{aligned} x_t &= x_{t-1} + v_{x(t-1)}\Delta t \\ v_{xt} &= vx_{t-1} \\ y_t &= y_{t-1} + v_{y(t-1)}\Delta t \\ v_{yt} &= -g\Delta t + v_{y(t-1)} \end{aligned}$$

Therefore we want the product $\mathbf{Bu}$ to equal $\begin{bmatrix}0 & 0 & 0 & -g\Delta t\end{bmatrix}^\mathsf{T}$. In some sense it is arbitrary how we define $\mathbf{B}$ and $\mathbf{u}$ so long as multiplying them together yields this result. For example, we could define $\mathbf{B} = 1$ and $\mathbf{u} = \begin{bmatrix}0 & 0 & 0 & -g\Delta t\end{bmatrix}^\mathsf{T}$. But this doesn't really fit with our definitions for $\mathbf{B}$ and $\mathbf{u}$, where $\mathbf{u}$ is the control input, and $\mathbf{B}$ is the control function. The control input is $-g$ for the velocity of y. So this is one possible definition.

$$\mathbf{B} = \begin{bmatrix}0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \Delta t\end{bmatrix}, \mathbf{u} = \begin{bmatrix}0 \\ 0 \\ 0 \\ -g\end{bmatrix}$$

To me this seems a bit excessive; I would suggest we might want $\mathbf{u}$ to contain the control input for the two dimensions $x$ and $y$, which suggests

$$\mathbf{B} = \begin{bmatrix}0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & \Delta t\end{bmatrix}, \mathbf{u} = \begin{bmatrix}0 \\ -g\end{bmatrix}$$

.

You might prefer to only provide control inputs that actually exist, and there is no control input for $x$, so we arrive at

$$\mathbf{B} = \begin{bmatrix}0 \\ 0 \\ 0 \\ \Delta t\end{bmatrix}, \mathbf{u} = \begin{bmatrix}-g\end{bmatrix}$$

.

I've seen people use

$$\mathbf{B} = \begin{bmatrix}0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1\end{bmatrix}, \mathbf{u} = \begin{bmatrix}0 \\ 0 \\ 0 \\ -g\Delta t\end{bmatrix}$$

While this does produce the correct result, I am resistant to putting time into $\mathbf{u}$ as time is not a control input, it is what we use to convert the control input into a change in state, which is the job of $\mathbf{B}$.

### 8.11.4 Design the Measurement Function

The measurement function defines how we go from the state variables to the measurements using the equation $\mathbf{z} = \mathbf{Hx}$. We will assume that we have a sensor that provides us with the position of the ball in (x,y), but cannot measure velocities or accelerations. Therefore our function must be:

$$\begin{bmatrix}z_x \\ z_y\end{bmatrix} = \begin{bmatrix}1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0\end{bmatrix}\begin{bmatrix}x \\ \dot{x} \\ y \\ \dot{y}\end{bmatrix}$$

where

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

### 8.11.5 Design the Measurement Noise Matrix

As with the robot, we will assume that the error is independent in $x$ and $y$. In this case we will start by assuming that the measurement errors in x and y are 0.5 meters squared. Hence,

$$\mathbf{R} = \begin{bmatrix} 0.5 & 0 \\ 0 & 0.5 \end{bmatrix}$$

### 8.11.6 Design the Process Noise Matrix

We are assuming a ball moving in a vacuum, so there should be no process noise. We have 4 state variables, so we need a 4×4 covariance matrix:

$$\mathbf{Q} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

### 8.11.7 Design the Initial Conditions

We already performed this step when we tested the state transition function. Recall that we computed the initial velocity for $x$ and $y$ using trigonometry, and set the value of $\mathbf{x}$ with:

```
omega = radians(omega)
vx = cos(omega) * v0
vy = sin(omega) * v0

f1.x = np.array([[x, vx, y, vy]]).T
```

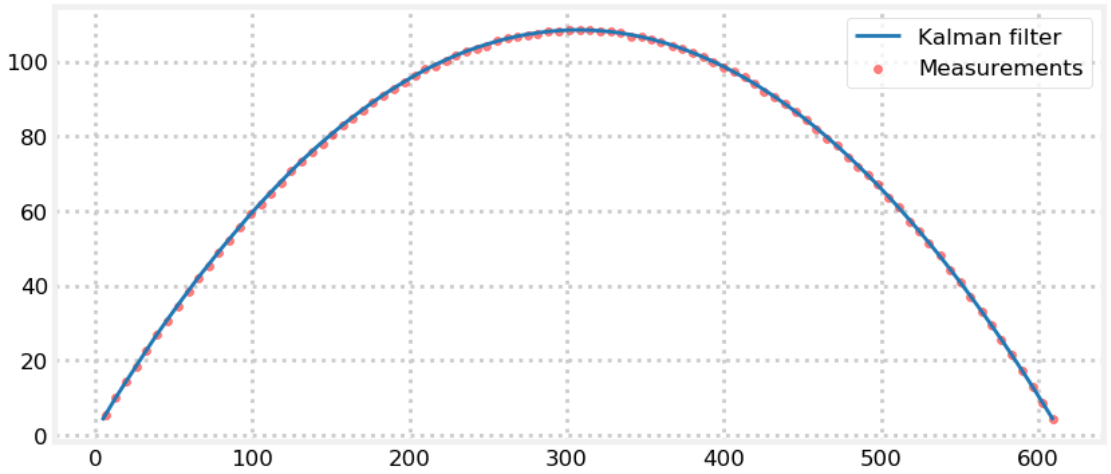
With all the steps done we are ready to implement our filter and test it. First, the implementation:

```
In [64]: from math import sin, cos, radians

         def ball_kf(x, y, omega, v0, dt, r=0.5, q=0.):
             kf = KalmanFilter(dim_x=4, dim_z=2, dim_u=1)

             kf.F = np.array([[1., dt, 0., 0.],    # x   = x0 + dx*dt
                              [0., 1., 0., 0.],    # dx  = dx0
                              [0., 0., 1., dt],    # y   = y0 + dy*dt
                              [0., 0., 0., 1.]])  # dy  = dy0

             kf.H = np.array([[1., 0., 0., 0.],
                              [0., 0., 1., 0.]])

             kf.B = np.array([[0., 0., 0., dt]]).T
             kf.R *= r
             kf.Q *= q

             omega = radians(omega)
             vx = cos(omega) * v0
             vy = sin(omega) * v0
```

```
            kf.x = np.array([[x, vx, y, vy]]).T
            return kf
```

Now we will test the filter by generating measurements for the ball using the ball simulation class.

```
In [65]: def track_ball_vacuum(dt):
            global kf
            x, y = 0., 1.
            theta = 35.  # launch angle
            v0 = 80.
            g = np.array([[-9.8]])  # gravitational constant
            ball = BallTrajectory2D(x0=x, y0=y, theta_deg=theta, velocity=v0,
                                    noise=[.2, .2])
            kf = ball_kf(x, y, theta, v0, dt)

            t = 0
            xs, ys = [], []
            while kf.x[2] > 0:
                t += dt
                x, y = ball.step(dt)
                z = np.array([[x, y]]).T

                kf.update(z)
                xs.append(kf.x[0])
                ys.append(kf.x[2])
                kf.predict(u=g)
                p1 = plt.scatter(x, y, color='r', marker='.', s=75, alpha=0.5)
            p2, = plt.plot(xs, ys, lw=2)
            plt.legend([p2, p1], ['Kalman filter', 'Measurements'],
                       scatterpoints=1)

        track_ball_vacuum(dt=1./10)
```




We see that the Kalman filter reasonably tracks the ball. However, as already explained, this is a trivial example because we have no process noise. We can predict trajectories in a vacuum with arbitrary precision; using a Kalman filter in this example is a needless complication. A least squares curve fit would give identical results.

## 8.12 Tracking a Ball in Air

For this problem we assume that we are tracking a ball traveling through the Earth's atmosphere. The path of the ball is influenced by wind, drag, and the rotation of the ball. We will assume that our sensor is a camera; code that we will not implement will perform some type of image processing to detect the position of the ball. This is typically called *blob detection* in computer vision. However, image processing code is not perfect; in any given frame it is possible to either detect no blob or to detect spurious blobs that do not correspond to the ball. Finally, we will not assume that we know the starting position, angle, or rotation of the ball; the tracking code will have to initiate tracking based on the measurements that are provided. The main simplification that we are making here is a 2D world; we assume that the ball is always traveling orthogonal to the plane of the camera's sensor. We have to make that simplification at this point because we have not discussed how we might extract 3D information from a camera, which provides only 2D data.

### 8.12.1 Implementing Air Drag

Our first step is to implement the math for a ball moving through air. There are several treatments available. A robust solution takes into account issues such as ball roughness (which affects drag non-linearly depending on velocity), the Magnus effect (spin causes one side of the ball to have higher velocity relative to the air vs the opposite side, so the coefficient of drag differs on opposite sides), the effect of lift, humidity, air density, and so on. I assume the reader is not interested in the details of ball physics, and so will restrict this treatment to the effect of air drag on a non-spinning baseball. I will use the math developed by Nicholas Giordano and Hisao Nakanishi in *Computational Physics* [1997]. This treatement does not take all the factors into account. The most detailed treatment is by Alan Nathan on his website at http://baseball.physics.illinois.edu/index.html. I use his math in my own work in computer vision, but I do not want to get distracted by a more complicated model.

**Important**: Before I continue, let me point out that you will not have to understand this next piece of physics to proceed with the Kalman filter. My goal is to create a reasonably accurate behavior of a baseball in the real world, so that we can test how our Kalman filter performs with real-world behavior. In real world applications it is usually impossible to completely model the physics of a real world system, and we make do with a process model that incorporates the large scale behaviors. We then tune the measurement noise and process noise until the filter works well with our data. There is a real risk to this; it is easy to finely tune a Kalman filter so it works perfectly with your test data, but performs badly when presented with slightly different data. This is perhaps the hardest part of designing a Kalman filter, and why it gets referred to with terms such as 'black art'.

I dislike books that implement things without explanation, so I will now develop the physics for a ball moving through air. Move on past the implementation of the simulation if you are not interested.

A ball moving through air encounters wind resistance. This imparts a force on the wall, called *drag*, which alters the flight of the ball. In Giordano this is denoted as

$$F_{drag} = -B_2 v^2$$

where $B_2$ is a coefficient derived experimentally, and $v$ is the velocity of the object. $F_{drag}$ can be factored into $x$ and $y$ components with

$$F_{drag,x} = -B_2 v v_x$$
$$F_{drag,y} = -B_2 v v_y$$

If $m$ is the mass of the ball, we can use $F = ma$ to compute the acceleration as

$$a_x = -\frac{B_2}{m} v v_x$$
$$a_y = -\frac{B_2}{m} v v_y$$

Giordano provides the following function for $\frac{B_2}{m}$, which takes air density, the cross section of a baseball, and its roughness into account. Understand that this is an approximation based on wind tunnel tests and several simplifying assumptions. It is in SI units: velocity is in meters/sec and time is in seconds.

$$\frac{B_2}{m} = 0.0039 + \frac{0.0058}{1 + \exp\left[(v-35)/5\right]}$$

Starting with this Euler discretization of the ball path in a vacuum:

$$\begin{aligned} x &= v_x \Delta t \\ y &= v_y \Delta t \\ v_x &= v_x \\ v_y &= v_y - 9.8\Delta t \end{aligned}$$

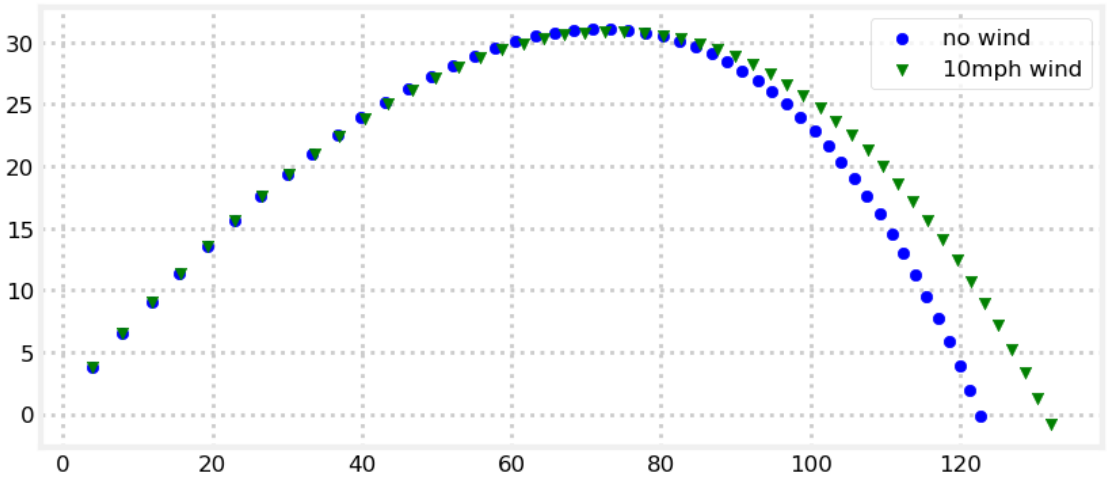
We can incorporate this force (acceleration) into our equations by incorporating $accel * \Delta t$ into the velocity update equations. We should subtract this component because drag will reduce the velocity. The code to do this is quite straightforward, we just need to break out the force into $x$ and $y$ components.

I will not belabor this issue further because computational physics is beyond the scope of this book. Recognize that a higher fidelity simulation would require incorporating things like altitude, temperature, ball spin, and several other factors. The aforementioned work by Alan Nathan covers this if you are interested. My intent here is to impart some real-world behavior into our simulation to test how our simpler prediction model used by the Kalman filter reacts to this behavior. Your process model will never exactly model what happens in the world, and a large factor in designing a good Kalman filter is carefully testing how it performs against real world data.

The code below computes the behavior of a baseball in air, at sea level, in the presence of wind. I plot the same initial hit with no wind, and then with a tail wind at 10 mph. Baseball statistics are universally done in US units, and we will follow suit here (http://en.wikipedia.org/wiki/United_States_customary_units). Note that the velocity of 110 mph is a typical exit speed for a baseball for a home run hit.

```
In [66]: from math import sqrt, exp

         def mph_to_mps(x):
             return x * .447

         def drag_force(velocity):
             """ Returns the force on a baseball due to air drag at
             the specified velocity. Units are SI"""

             return velocity * (0.0039 + 0.0058 /
                     (1. + exp((velocity-35.)/5.)))

         v = mph_to_mps(110.)
         x, y = 0., 1.
         dt = .1
         theta = radians(35)

         def solve(x, y, vel, v_wind, launch_angle):
             xs = []
             ys = []
             v_x = vel*cos(launch_angle)
             v_y = vel*sin(launch_angle)
             while y >= 0:
                 # Euler equations for x and y
                 x += v_x*dt
                 y += v_y*dt

                 # force due to air drag
                 velocity = sqrt((v_x-v_wind)**2 + v_y**2)
```

```
            F = drag_force(velocity)

            # euler's equations for vx and vy
            v_x = v_x - F*(v_x-v_wind)*dt
            v_y = v_y - 9.8*dt - F*v_y*dt

            xs.append(x)
            ys.append(y)

        return xs, ys

x, y = solve(x=0, y=1, vel=v, v_wind=0, launch_angle=theta)
p1 = plt.scatter(x, y, color='blue', label='no wind')

wind = mph_to_mps(10)
x, y = solve(x=0, y=1, vel=v, v_wind=wind, launch_angle=theta)
p2 = plt.scatter(x, y, color='green', marker="v",
                 label='10mph wind')
plt.legend(scatterpoints=1);
```




We can easily see the difference between the trajectory in a vacuum and in the air. I used the same initial velocity and launch angle in the ball in a vacuum section above. We computed that the ball in a vacuum would travel over 240 meters (nearly 800 ft). In the air, the distance is just over 120 meters, or roughly 400 ft. 400ft is a realistic distance for a well hit home run ball, so we can be confident that our simulation is reasonably accurate.

Without further ado we will create a ball simulation that uses the math above to create a more realistic ball trajectory. I will note that the nonlinear behavior of drag means that there is no analytic solution to the ball position at any point in time, so we need to compute the position step-wise. I use Euler's method to propagate the solution; use of a more accurate technique such as Runge-Kutta is left as an exercise for the reader. That modest complication is unnecessary for what we are doing because the accuracy difference between the techniques will be small for the time steps we will be using.

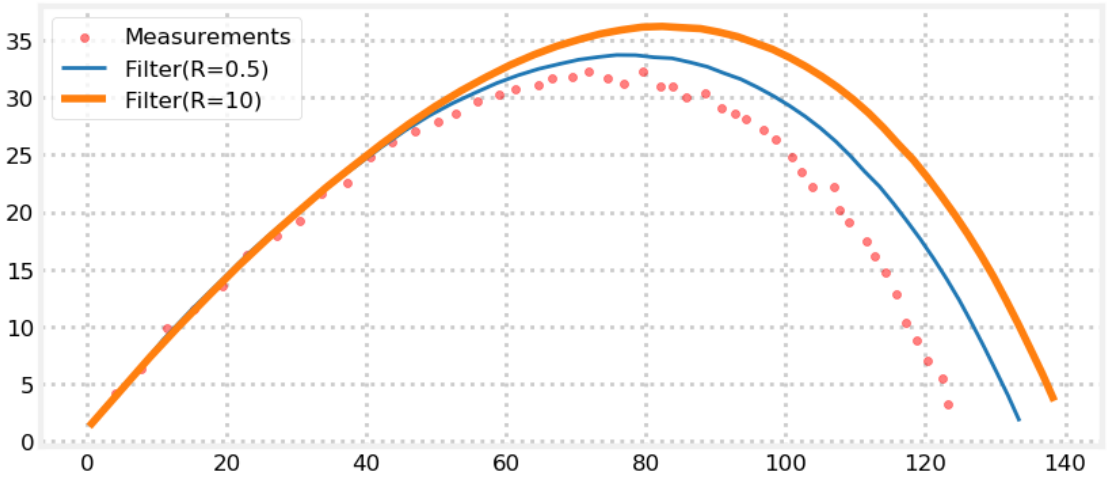
```
In [67]: class BaseballPath:
             def __init__(self, x0, y0, launch_angle_deg, velocity_ms,
                          noise=(1.0, 1.0)):
                 """ Create 2D baseball path object
```

```
            (x = distance from start point in ground plane,
             y=height above ground)

        x0,y0            initial position
        launch_angle_deg angle ball is travelling respective to
                         ground plane
        velocity_ms      speeed of ball in meters/second
        noise            amount of noise to add to each position
                         in (x, y)
        """

        omega = radians(launch_angle_deg)
        self.v_x = velocity_ms * cos(omega)
        self.v_y = velocity_ms * sin(omega)

        self.x = x0
        self.y = y0
        self.noise = noise


    def drag_force(self, velocity):
        """ Returns the force on a baseball due to air drag at
        the specified velocity. Units are SI
        """
        B_m = 0.0039 + 0.0058 / (1. + exp((velocity-35.)/5.))
        return B_m * velocity


    def update(self, dt, vel_wind=0.):
        """ compute the ball position based on the specified time
        step and wind velocity. Returns (x, y) position tuple.
        """

        # Euler equations for x and y
        self.x += self.v_x*dt
        self.y += self.v_y*dt

        # force due to air drag
        v_x_wind = self.v_x - vel_wind
        v = sqrt(v_x_wind**2 + self.v_y**2)
        F = self.drag_force(v)

        # Euler's equations for velocity
        self.v_x = self.v_x - F*v_x_wind*dt
        self.v_y = self.v_y - 9.81*dt - F*self.v_y*dt

        return (self.x + randn()*self.noise[0],
                self.y + randn()*self.noise[1])
```

Now we can test the Kalman filter against measurements created by this model.

```
In [68]: x, y = 0, 1.

         theta = 35. # launch angle
         v0 = 50.
```

```
dt = 1/10.   # time step
g = np.array([[-9.8]])

plt.figure()
ball = BaseballPath(x0=x, y0=y, launch_angle_deg=theta,
                    velocity_ms=v0, noise=[.3,.3])
f1 = ball_kf(x, y, theta, v0, dt, r=1.)
f2 = ball_kf(x, y, theta, v0, dt, r=10.)
t = 0
xs, ys = [], []
xs2, ys2 = [], []

while f1.x[2] > 0:
    t += dt
    x, y = ball.update(dt)
    z = np.array([[x, y]]).T

    f1.update(z)
    f2.update(z)
    xs.append(f1.x[0])
    ys.append(f1.x[2])
    xs2.append(f2.x[0])
    ys2.append(f2.x[2])
    f1.predict(u=g)
    f2.predict(u=g)

    p1 = plt.scatter(x, y, color='r', marker='.', s=75, alpha=0.5)

p2, = plt.plot(xs, ys, lw=2)
p3, = plt.plot(xs2, ys2, lw=4)
plt.legend([p1, p2, p3],
           ['Measurements', 'Filter(R=0.5)', 'Filter(R=10)'],
           loc='best', scatterpoints=1);
```




I have plotted the output of two different Kalman filter settings. The measurements are depicted as green circles, a Kalman filter with R=0.5 as a thin green line, and a Kalman filter with R=10 as a thick blue

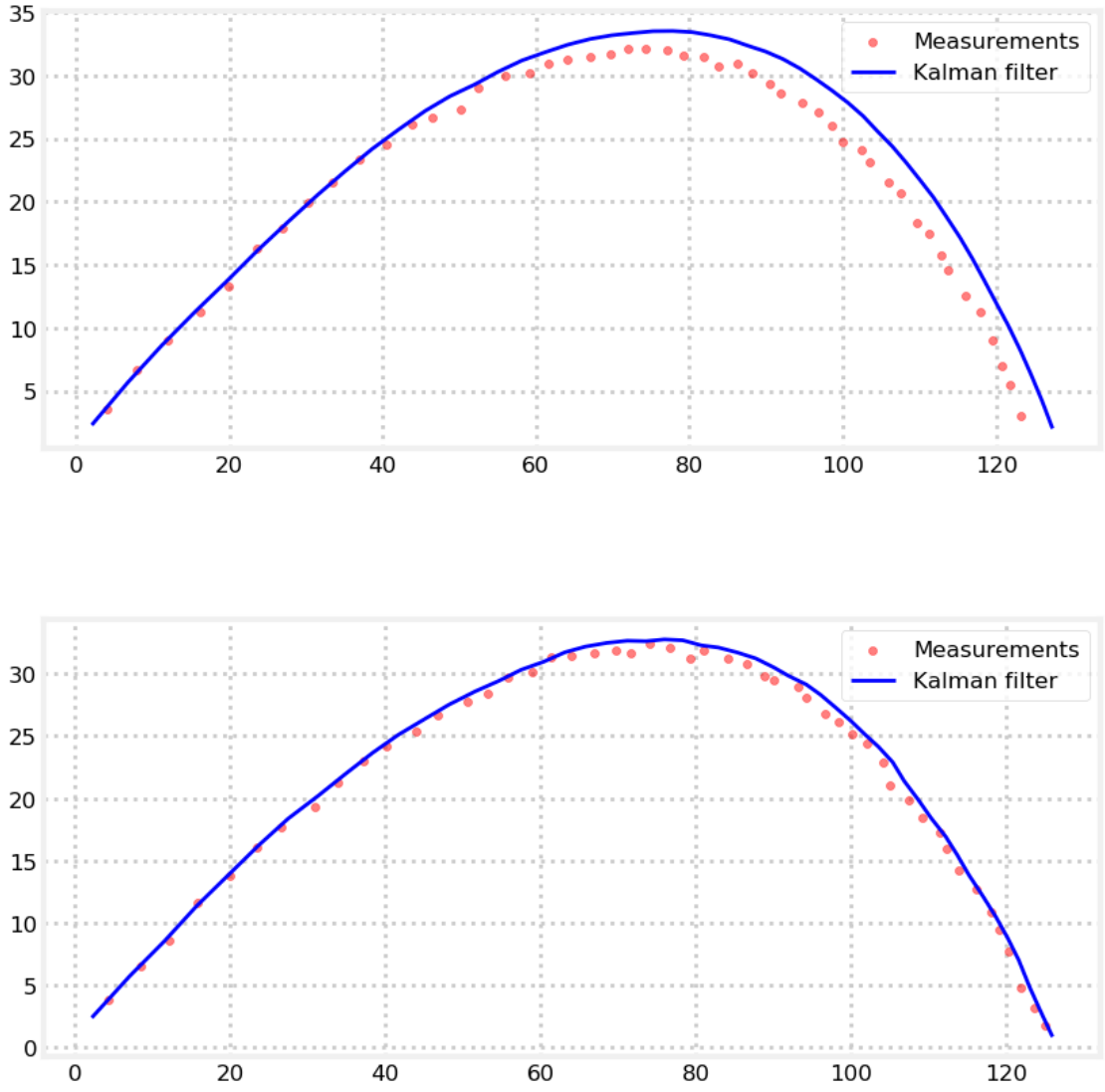
line. These R values are chosen merely to show the effect of measurement noise on the output, they are not intended to imply a correct design.

We can see that neither filter does very well. At first both track the measurements well, but as time continues they both diverge. This happens because the state model for air drag is nonlinear and the Kalman filter assumes that it is linear. If you recall our discussion about nonlinearity in the g-h filter chapter we showed why a g-h filter will always lag behind the acceleration of the system. We see the same thing here - the acceleration is negative, so the Kalman filter consistently overshoots the ball position. There is no way for the filter to catch up so long as the acceleration continues, so the filter will continue to diverge.

What can we do to improve this? The best approach is to perform the filtering with a nonlinear Kalman filter, and we will do this in subsequent chapters. However, there is also what I will call an 'engineering' solution to this problem as well. Our Kalman filter assumes that the ball is in a vacuum, and thus that there is no process noise. However, since the ball is in air the atmosphere imparts a force on the ball. We can think of this force as process noise. This is not a particularly rigorous thought; for one thing, this force is anything but Gaussian. Secondly, we can compute this force, so throwing our hands up and saying 'it's random' will not lead to an optimal solution. But let's see what happens if we follow this line of thought.

The following code implements the same Kalman filter as before, but with a non-zero process noise. I plot two examples, one with `Q=.1`, and one with `Q=0.01`.

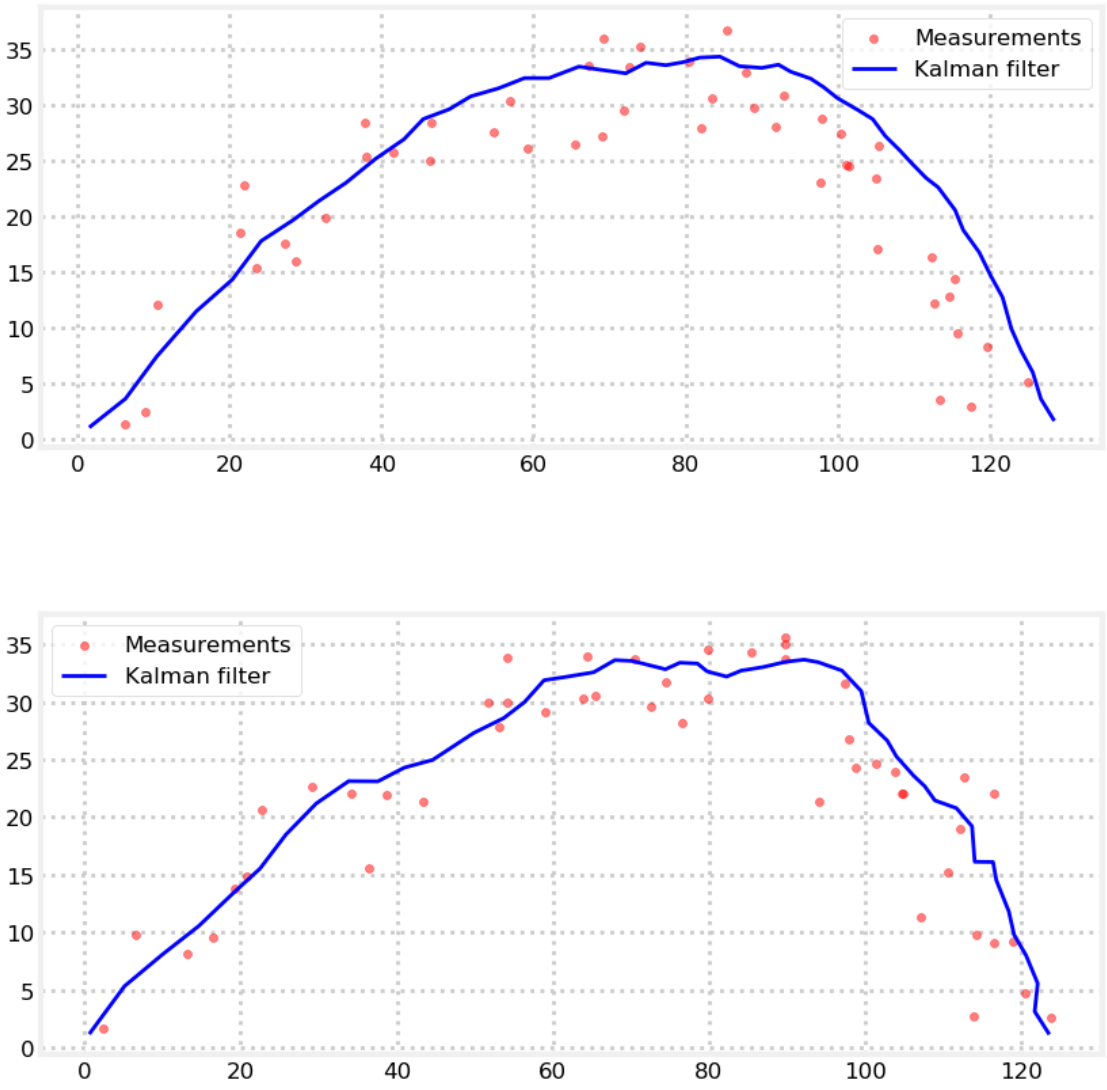
```
In [69]: def plot_ball_with_q(q, r=1., noise=0.3):
             x, y = 0., 1.
             theta = 35. # launch angle
             v0 = 50.
             dt = 1/10.   # time step
             g = np.array([[-9.8]])

             ball = BaseballPath(x0=x,
                                 y0=y,
                                 launch_angle_deg=theta,
                                 velocity_ms=v0,
                                 noise=[noise,noise])
             f1 = ball_kf(x, y, theta, v0, dt, r=r, q=q)
             t = 0
             xs, ys = [], []

             while f1.x[2] > 0:
                 t += dt
                 x, y = ball.update(dt)
                 z = np.array([[x, y]]).T

                 f1.update(z)
                 xs.append(f1.x[0])
                 ys.append(f1.x[2])
                 f1.predict(u=g)

                 p1 = plt.scatter(x, y, c='r', marker='.', s=75, alpha=0.5)

             p2, = plt.plot(xs, ys, lw=2, color='b')
             plt.legend([p1, p2], ['Measurements', 'Kalman filter'])
             plt.show()

         plot_ball_with_q(0.01)
         plot_ball_with_q(0.1)
```




The second filter tracks the measurements fairly well. There appears to be a bit of lag, but very little.

Is this a good technique? Usually not, but it depends. Here the nonlinearity of the force on the ball is fairly constant and regular. Assume we are trying to track an automobile - the accelerations will vary as the car changes speeds and turns. When we make the process noise higher than the actual noise in the system the filter will opt to weigh the measurements higher. If you don't have a lot of noise in your measurements this might work for you. However, consider this next plot where I have increased the noise in the measurements.

```
In [70]: plot_ball_with_q(0.01, r=3, noise=3.)
         plot_ball_with_q(0.1, r=3, noise=3.)
```




This output is terrible. The filter has no choice but to give more weight to the measurements than the process (prediction step), but when the measurements are noisy the filter output will just track the noise. This inherent limitation of the linear Kalman filter is what lead to the development of nonlinear versions of the filter.

With that said, it is certainly possible to use the process noise to deal with small nonlinearities in your system. This is part of the 'black art' of Kalman filters. Our model of the sensors and of the system are never perfect. Sensors are non-Gaussian and our process model is never perfect. You can mask some of this by setting the measurement errors and process errors higher than their theoretically correct values, but the trade off is a non-optimal solution. Certainly it is better to be non-optimal than to have your Kalman filter diverge. However, as we can see in the graphs above, it is easy for the output of the filter to be very bad. It is also very common to run many simulations and tests and to end up with a filter that performs very well under those conditions. Then, when you use the filter on real data the conditions are slightly different and the filter ends up performing terribly.

For now we will set this problem aside, as we are clearly misapplying the Kalman filter in this example. We will revisit this problem in subsequent chapters to see the effect of using various nonlinear techniques. In some domains you will be able to get away with using a linear Kalman filter for a nonlinear problem, but usually you will have to use one or more of the techniques you will learn in the rest of this book.

## 8.13 References

[1] Bar-Shalom, Yaakov, et al. *Estimation with Applications to Tracking and Navigation.* John Wiley & Sons, 2001.

# Chapter 9

# Nonlinear Filtering

```
In [1]: from __future__ import division, print_function
        %matplotlib inline

In [2]: #format the book
        import book_format
        book_format.set_style()
```

```
Out[2]:
```

## 9.1 Introduction

The Kalman filter that we have developed uses linear equations, and so the filter can only handle linear problems. But the world is nonlinear, and so the classic filter that we have been studying to this point can have very limited utility.

There can be nonlinearity in the process model. Suppose we want to track an object falling through the atmosphere. The acceleration of the object depends on the drag it encounters. Drag depends on air density, and the air density decreases with altitude. In one dimension this can be modelled with the nonlinear differential equation

$$\ddot{x} = \frac{0.0034ge^{-x/22000}\dot{x}^2}{2\beta} - g$$

A second source of nonlinearity comes from the measurements. For example, radars measure the slant range to an object, and we are typically interested in the aircraft's position over the ground. We invoke Pythagoras and get the nonlinear equation:

$$x = \sqrt{\mathtt{slant}^2 - \mathtt{altitude}^2}$$

These facts were not lost on the early adopters of the Kalman filter. Soon after Dr. Kalman published his paper people began working on how to extend the Kalman filter for nonlinear problems.

It is almost true to state that the only equation anyone knows how to solve is $\mathbf{Ax} = \mathbf{b}$. We only really know how to do linear algebra. I can give you any linear set of equations and you can either solve it or prove that it has no solution.

Anyone with formal education in math or physics has spent years learning various analytic ways to solve integrals, differential equations and so on. Yet even trivial physical systems produce equations that cannot be solved analytically. I can take an equation that you are able to integrate, insert a log term, and render it insolvable. This leads to jokes about physicists stating "assume a spherical cow on a frictionless surface in a vacuum…". Without making extreme simplifications most physical problems do not have analytic solutions.

How do we do things like model airflow over an aircraft in a computer, or predict weather, or track missiles with a Kalman filter? We retreat to what we know: $\mathbf{Ax} = \mathbf{b}$. We find some way to linearize the

problem, turning it into a set of linear equations, and then use linear algebra software packages to compute an approximate solution.

Linearizing a nonlinear problem gives us inexact answers, and in a recursive algorithm like a Kalman filter or weather tracking system these small errors can sometimes reinforce each other at each step, quickly causing the algorithm to spit out nonsense.

What we are about to embark upon is a difficult problem. There is not one obvious, correct, mathematically optimal solution anymore. We will be using approximations, we will be introducing errors into our computations, and we will forever be battling filters that *diverge*, that is, filters whose numerical errors overwhelm the solution.

In the remainder of this short chapter I will illustrate the specific problems the nonlinear Kalman filter faces. You can only design a filter after understanding the particular problems the nonlinearity in your problem causes. Subsequent chapters will then teach you how to design and implement different kinds of nonlinear filters.

## 9.2 The Problem with Nonlinearity

The mathematics of the Kalman filter is beautiful in part due to the Gaussian equation being so special. It is nonlinear, but when we add and multiply them we get another Gaussian as a result. That is very rare. $\sin x * \sin y$ does not yield a sin as an output.

What I mean by linearity may be obvious, but there are some subtleties. The mathematical requirements are twofold:

- additivity: $f(x+y) = f(x) + f(y)$
- homogeneity: $f(ax) = af(x)$

This leads us to say that a linear system is defined as a system whose output is linearly proportional to the sum of all its inputs. A consequence of this is that to be linear if the input is zero than the output must also be zero. Consider an audio amp - if I sing into a microphone, and you start talking, the output should be the sum of our voices (input) scaled by the amplifier gain. But if the amplifier outputs a nonzero signal such as a hum for a zero input the additive relationship no longer holds. This is because linearity requires that $amp(voice) = amp(voice + 0)$. This clearly should give the same output, but if amp(0) is nonzero, then

$$\begin{aligned} amp(voice) &= amp(voice + 0) \\ &= amp(voice) + amp(0) \\ &= amp(voice) + non\_zero\_value \end{aligned}$$

which is clearly nonsense. Hence, an apparently linear equation such as

$$L(f(t)) = f(t) + 1$$

is not linear because $L(0) = 1$. Be careful!

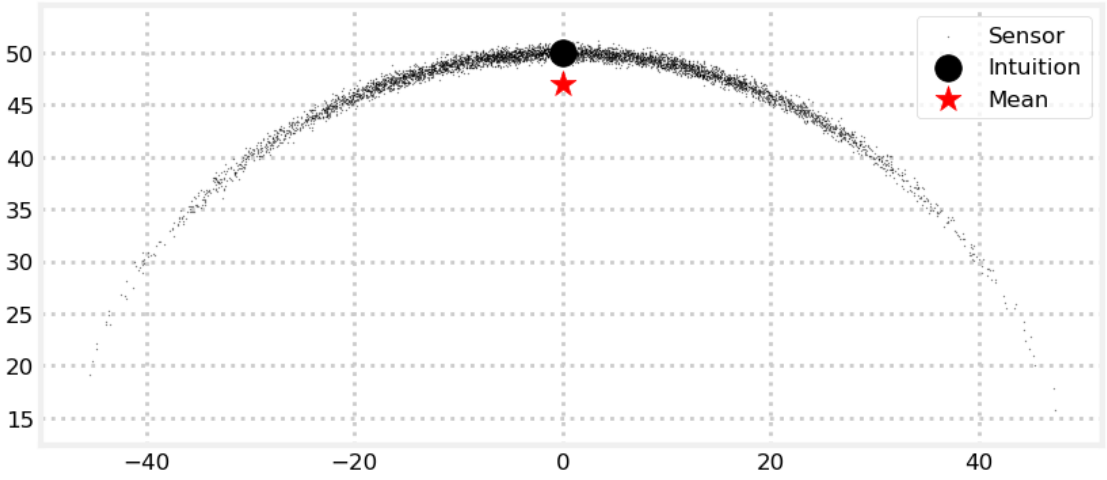
## 9.3 An Intuitive Look at the Problem

I particularly like the following way of looking at the problem, which I am borrowing from Dan Simon's *Optimal State Estimation* Section **??**. Consider a tracking problem where we get the range and bearing to a target, and we want to track its position. The reported distance is 50 km, and the reported angle is 90°. Assume that the errors in both range and angle are distributed in a Gaussian manner. Given an infinite number of measurements what is the expected value of the position?

I have been recommending using intuition to gain insight, so let's see how it fares for this problem. We might reason that since the mean of the range will be 50 km, and the mean of the angle will be 90°, that the answer will be x=0 km, y=50 km.

Let's plot that and find out. Here are 3000 points plotted with a normal distribution of the distance of 0.4 km, and the angle having a normal distribution of 0.35 radians. We compute the average of the all of the positions, and display it as a star. Our intuition is displayed with a large circle.

```
In [3]: import numpy as np
        from numpy.random import randn
        import matplotlib.pyplot as plt

        N = 5000
        a = np.pi/2. + (randn(N) * 0.35)
        r = 50.0     + (randn(N) * 0.4)
        xs = r * np.cos(a)
        ys = r * np.sin(a)

        plt.scatter(xs, ys, label='Sensor', color='k',
                    alpha=0.4, marker='.', s=1)
        xmean, ymean = sum(xs) / N, sum(ys) / N
        plt.scatter(0, 50, c='k', marker='o', s=200, label='Intuition')
        plt.scatter(xmean, ymean, c='r', marker='*', s=200, label='Mean')
        plt.axis('equal')
        plt.legend();
```



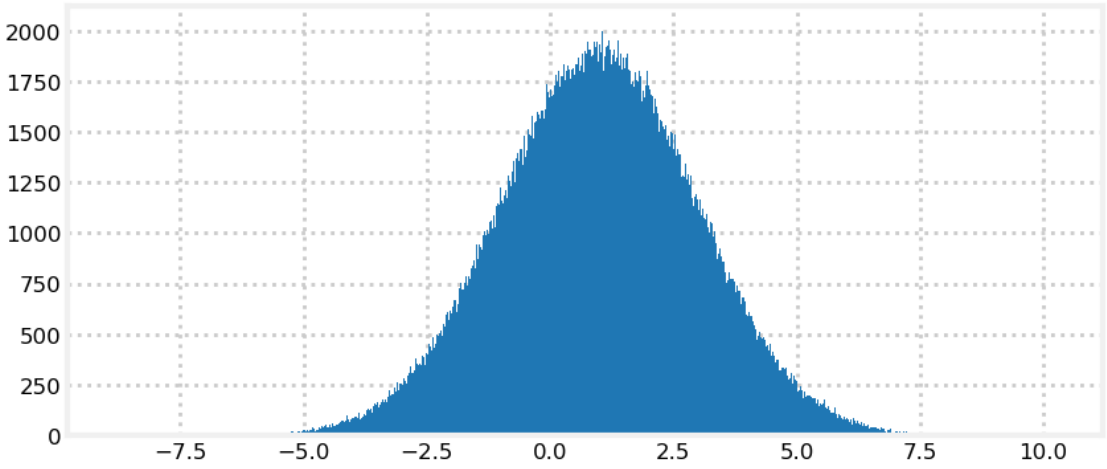

We can see that out intuition failed us because the nonlinearity of the problem forced all of the errors to be biased in one direction. This bias, over many iterations, can cause the Kalman filter to diverge. Even if it doesn't diverge the solution will not be optimal. Linear approximations applied to nonlinear problems yields inaccurate results.

## 9.4 The Effect of Nonlinear Functions on Gaussians

Gaussians are not closed under an arbitrary nonlinear function. Recall the equations of the Kalman filter - at each evolution we pass the Gaussian representing the state through the process function to get the Gaussian at time $k$. Our process function was always linear, so the output was always another Gaussian. Let's look at that on a graph. I will take an arbitrary Gaussian and pass it through the function $f(x) = 2x + 1$ and plot the result. We know how to do this analytically, but let's use sampling. I will generate 500,000 points with a normal distribution, pass them through $f(x)$, and plot the results. I do it this way because the next example will be nonlinear, and we will have no way to compute this analytically.

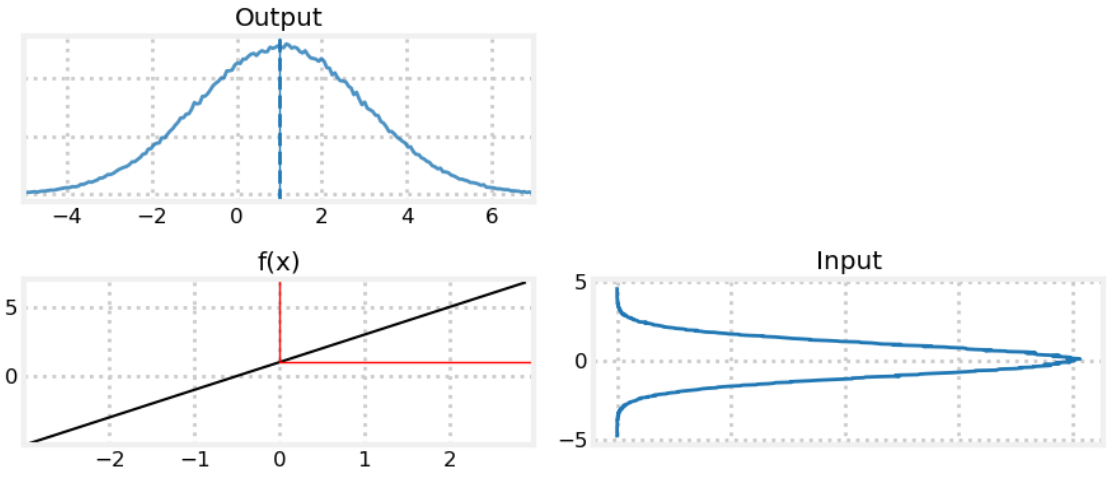
```
In [4]: from numpy.random import normal
```

```
data = normal(loc=0., scale=1., size=500000)
plt.hist(2*data + 1, 1000);
```




This is an unsurprising result. The result of passing the Gaussian through $f(x) = 2x + 1$ is another Gaussian centered around 1. Let's look at the input, nonlinear function, and output at once.

In [5]:
```
from kf_book.book_plots import set_figsize, figsize
from kf_book.nonlinear_plots import plot_nonlinear_func

def g1(x):
    return 2*x+1

plot_nonlinear_func(data, g1)
```

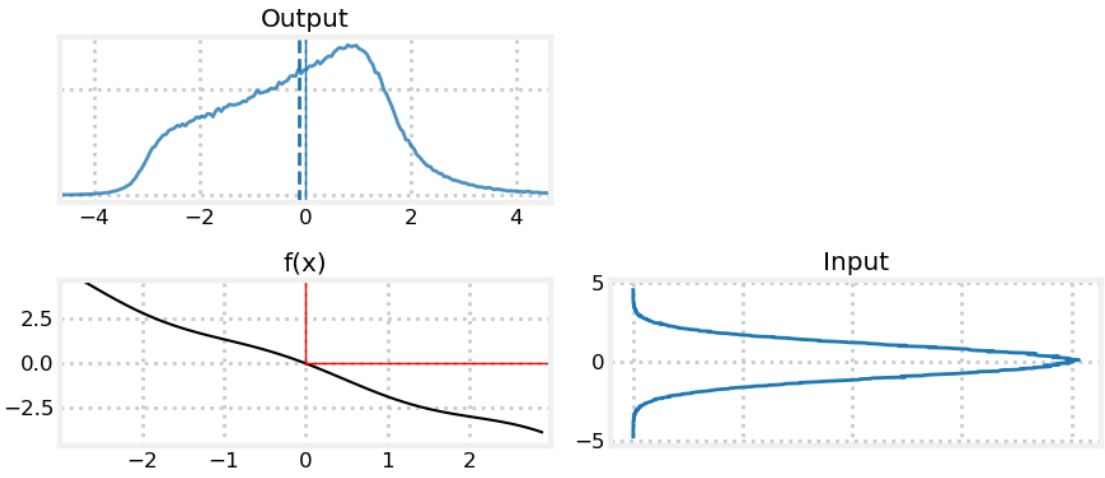



I explain how to plot Gaussians, and much more, in the Notebook *Computing_and_Plotting_PDFs* in the Supporting_Notebooks folder. You can also read it online here[1]

The plot labeled 'Input' is the histogram of the original data. This is passed through the function $f(x) = 2x + 1$ which is displayed in the chart on the bottom left. The red lines shows how one value, $x = 0$ is passed through the function. Each value from input is passed through in the same way to the output function on the right. For the output I computed the mean by taking the average of all the points, and drew the results with the dotted blue line. A solid blue line shows the actual mean for the point $x = 0$. The output looks like a Gaussian, and is in fact a Gaussian. We can see that the variance in the output is larger than the variance in the input, and the mean has been shifted from 0 to 1, which is what we would expect given the transfer function $f(x) = 2x + 1$ The $2x$ affects the variance, and the $+1$ shifts the mean The computed mean, represented by the dotted blue line, is nearly equal to the actual mean. If we used more points in our computation we could get arbitrarily close to the actual value.

Now let's look at a nonlinear function and see how it affects the probability distribution.

```
In [6]: def g2(x):
            return (np.cos(3*(x/2 + 0.7))) * np.sin(0.3*x) - 1.6*x

        plot_nonlinear_func(data, g2)
```



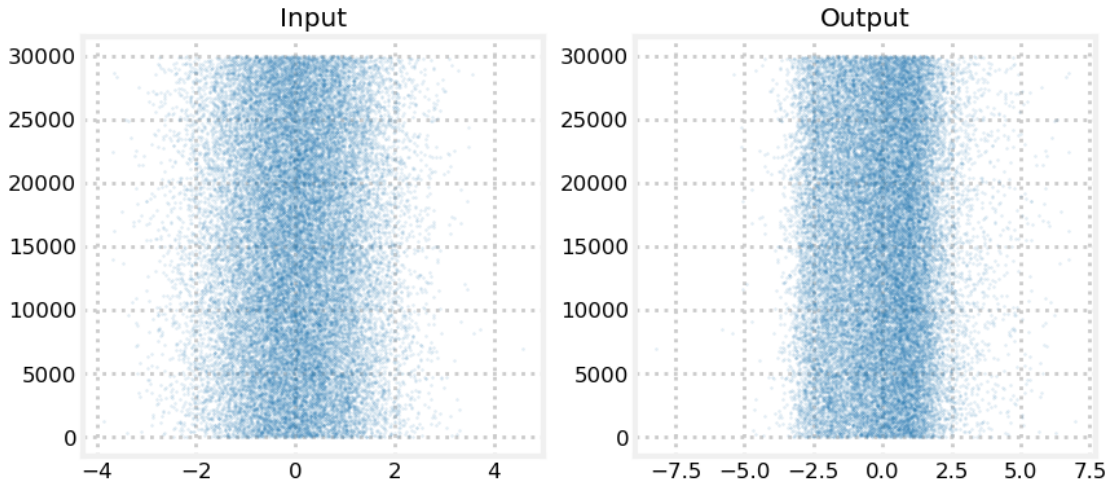

This result may be somewhat surprising to you. The function looks "fairly" linear, but the probability distribution of the output is completely different from a Gaussian. Recall the equations for multiplying two univariate Gaussians:

$$\mu = \frac{\sigma_1^2\mu_2 + \sigma_2^2\mu_1}{\sigma_1^2 + \sigma_2^2}$$
$$\sigma = \frac{1}{\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}}$$

These equations do not hold for non-Gaussians, and certainly do not hold for the probability distribution shown in the 'Output' chart above.

Here's another way to look at the same data as scatter plots.

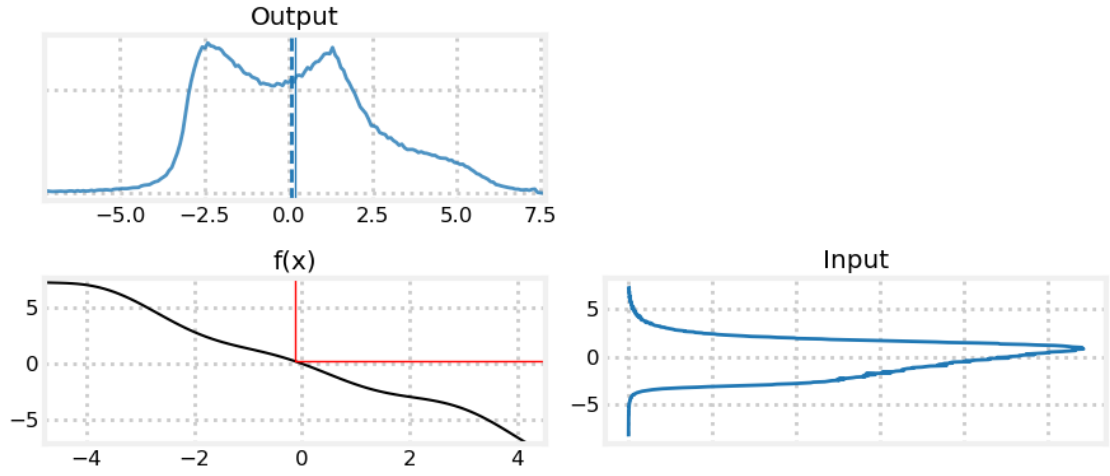
```
In [7]: N = 30000
        plt.subplot(121)
        plt.scatter(data[:N], range(N), alpha=.1, s=1.5)
        plt.title('Input')
        plt.subplot(122)
        plt.title('Output')
        plt.scatter(g2(data[:N]), range(N), alpha=.1, s=1.5);
```




The original data is clearly Gaussian, but the data passed through `g2(x)` is no longer normally distributed. There is a thick band near -3, and the points are unequally distributed on either side of the band. If you compare this to the pdf labelled 'output' in the previous chart you should be able to see how the pdf shape matches the distribution of `g(data)`.

Think of what this implies for the Kalman filter algorithm of the previous chapter. All of the equations assume that a Gaussian passed through the process function results in another Gaussian. If this is not true then all of the assumptions and guarantees of the Kalman filter do not hold. Let's look at what happens when we pass the output back through the function again, simulating the next step time step of the Kalman filter.

```
In [8]: y = g2(data)
        plot_nonlinear_func(y, g2)
```




As you can see the probability function is further distorted from the original Gaussian. However, the graph is still somewhat symmetric around x=0, let's see what the mean is.

```
In [9]: print('input  mean, variance: %.4f, %.4f' %
              (np.mean(data), np.var(data)))
```

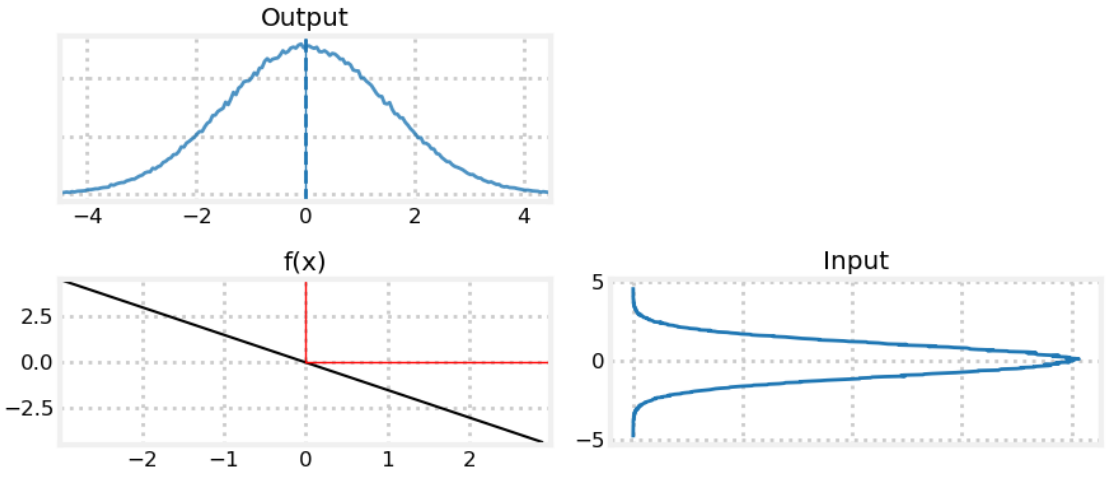
```
        print('output mean, variance: %.4f, %.4f' %
              (np.mean(y), np.var(y)))
```

```
input  mean, variance: -0.0028, 0.9974
output mean, variance: -0.1202, 2.4002
```

Let's compare that to the linear function that passes through (-2,3) and (2,-3), which is very close to the nonlinear function we have plotted. Using the equation of a line we have

$$m = \frac{-3-3}{2-(-2)} = -1.5$$

```
In [10]: def g3(x):
             return -1.5 * x

         plot_nonlinear_func(data, g3)
         out = g3(data)
         print('output mean, variance: %.4f, %.4f' %
               (np.mean(out), np.var(out)))
```




```
output mean, variance: 0.0043, 2.2441
```

Although the shapes of the output are very different, the mean and variance of each are almost the same. This may lead us to reasoning that perhaps we can ignore this problem if the nonlinear equation is 'close to' linear. To test that, we can iterate several times and then compare the results.

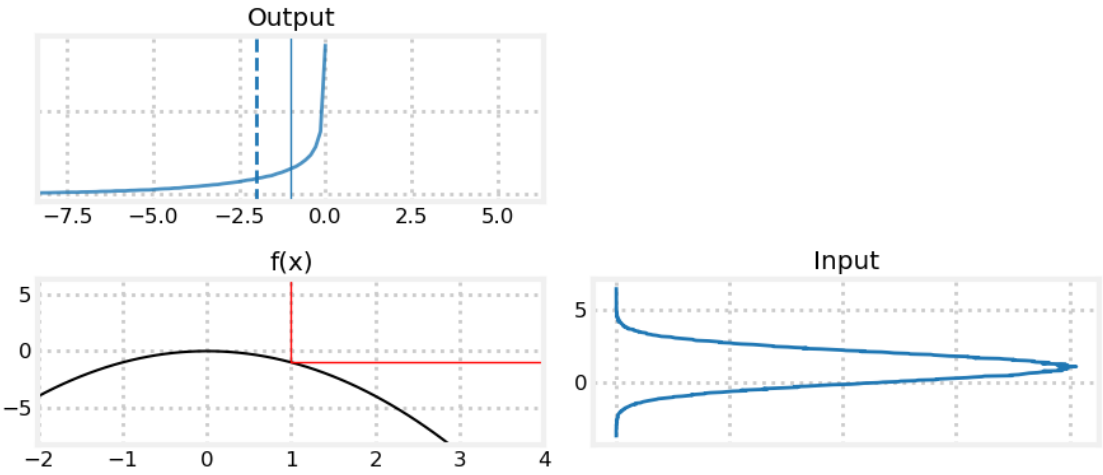
```
In [11]: out = g3(data)
         out2 = g2(data)

         for i in range(10):
             out = g3(out)
             out2 = g2(out2)
         print('linear    output mean, variance: %.4f, %.4f' %
               (np.average(out), np.std(out)**2))
         print('nonlinear output mean, variance: %.4f, %.4f' %
               (np.average(out2), np.std(out2)**2))
```

```
linear    output mean, variance: 0.2452, 7462.3247
nonlinear output mean, variance: -8.6253, 30439.3887
```

Unfortunately the nonlinear version is not stable. It drifted significantly from the mean of 0, and the variance is half an order of magnitude larger.

I minimized the issue by using a function that is quite close to a straight line. What happens if the function is $y(x) = -x^2$?

```
In [12]: def g3(x):
             return -x*x

         data = normal(loc=1, scale=1, size=500000)
         plot_nonlinear_func(data, g3)
```

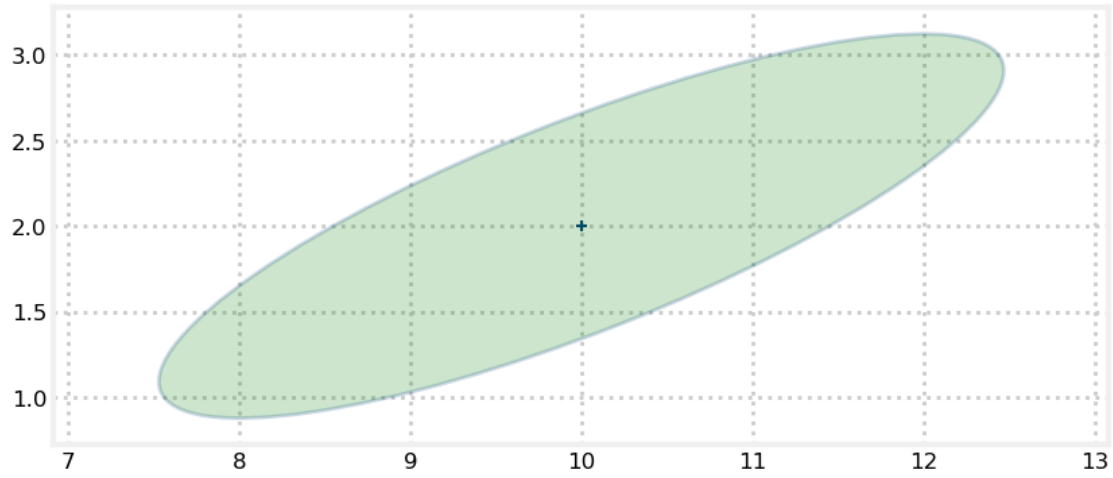



Despite the curve being smooth and reasonably straight at $x = 1$ the probability distribution of the output doesn't look anything like a Gaussian and the computed mean of the output is quite different than the value computed directly. This is not an unusual function - a ballistic object moves in a parabola, and this is the sort of nonlinearity your filter will need to handle. If you recall we've tried to track a ball and failed miserably. This graph should give you insight into why the filter performed so poorly.

## 9.5 A 2D Example

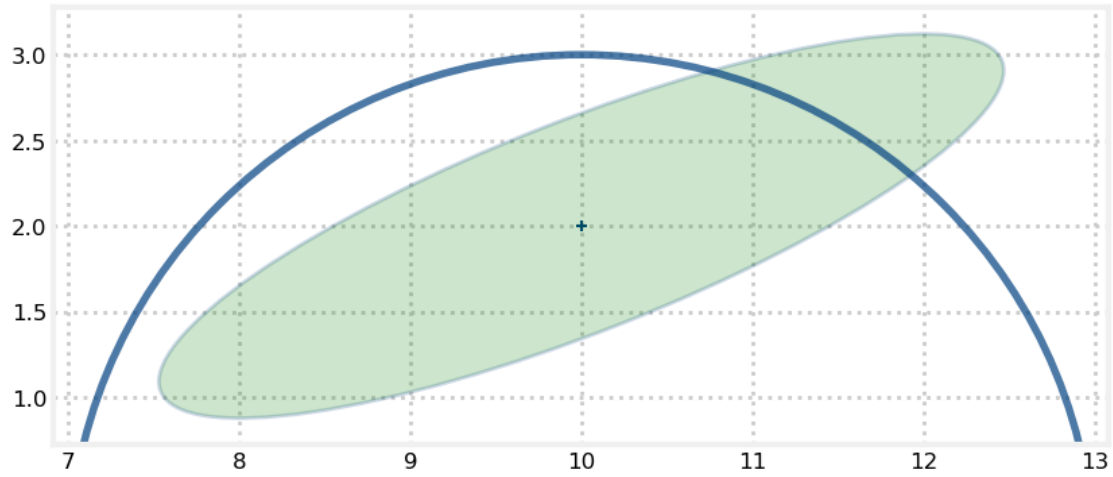
It is hard to look at probability distributions and reason about what will happen in a filter. So let's think about tracking an aircraft with radar. The estimate may have a covariance that looks like this:

```
In [13]: import kf_book.nonlinear_internal as nonlinear_internal

         nonlinear_internal.plot1()
```



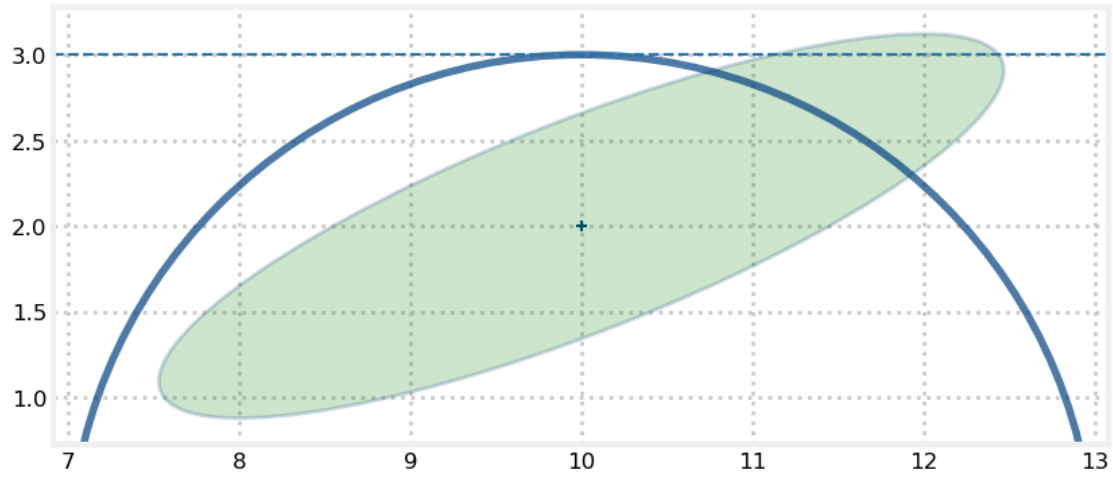

What happens when we try to linearize this problem? The radar gives us a range to the aircraft. Suppose the radar is directly under the aircraft (x=10) and the next measurement states that the aircraft is 3 miles away (y=3). The positions that could match that measurement form a circle with radius 3 miles, like so.

```
In [14]: nonlinear_internal.plot2()
```



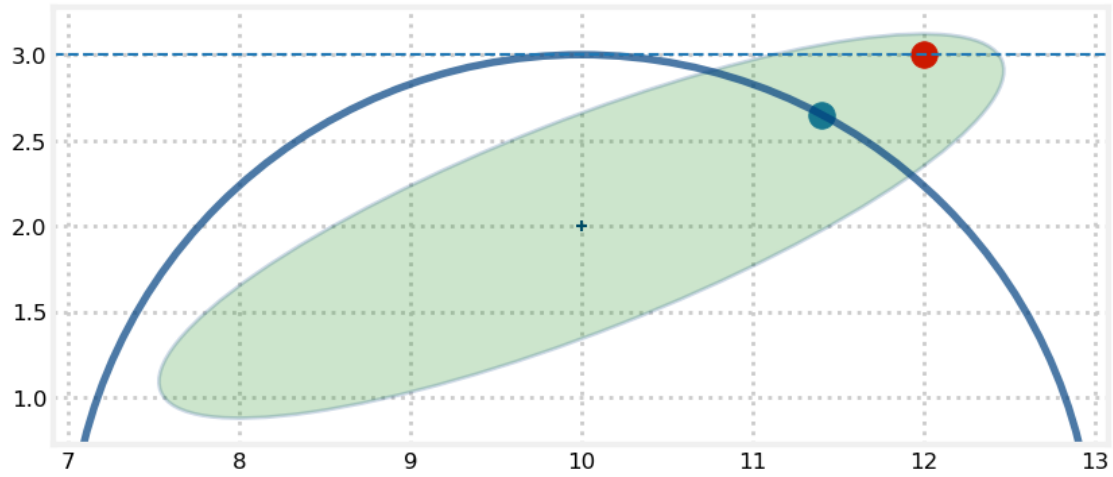

We can see by inspection that the probable position of the aircraft is somewhere near x=11.4, y=2.7 because that is where the covariance ellipse and range measurement overlap. But the range measurement is nonlinear so we have to linearize it. We haven't covered this material yet, but the Extended Kalman filter will linearize at the last position of the aircraft - (10,2). At x=10 the range measurement has y=3, and so we linearize at that point.

```
In [15]: nonlinear_internal.plot3()
```




Now we have a linear representation of the problem (literally a straight line) which we can solve. Unfortunately you can see that the intersection of the line and the covariance ellipse is a long way from the actual aircraft position.

```
In [16]: nonlinear_internal.plot4()
```




That sort of error often leads to disastrous results. The error in this estimate is large. But in the next innovation of the filter that very bad estimate will be used to linearize the next radar measurement, so the next estimate is likely to be markedly worse than this one. After only a few iterations the Kalman filter will diverge, and start producing results that have no correspondence to reality.

This covariance ellipse spans miles. I exaggerated the size to illustrate the difficulties of highly nonlinear systems. In real radar tracking problems the nonlinearity is usually not that bad, but the errors will still accumulate. Other systems you might work with could have this amount of nonlinearity - this was not an exaggeration only to make a point. You will always be battling divergence when working with nonlinear systems.

## 9.6 The Algorithms

You may be impatient to solve a specific problem, and wondering which filter to use. I will quickly survey the options. The subsequent chapters are somewhat independent of each other, and you can fruitfully skip around, though I recommend reading linearly if you truly want to master all of the material.

The workhorses of nonlinear filters are the *linearized Kalman filter* and *extended Kalman filter* (EKF). These two techniques were invented shortly after Kalman published his paper and they have been the main techniques used since then. The flight software in airplanes, the GPS in your car or phone almost certainly use one of these techniques.

However, these techniques are extremely demanding. The EKF linearizes the differential equations at one point, which requires you to find a solution to a matrix of partial derivatives (a Jacobian). This can be difficult or impossible to do analytically. If impossible, you have to use numerical techniques to find the Jacobian, but this is expensive computationally and introduces more error into the system. Finally, if the problem is quite nonlinear the linearization leads to a lot of error being introduced in each step, and the filters frequently diverge. You cannot throw some equations into some arbitrary solver and expect to to get good results. It's a difficult field for professionals. I note that most Kalman filtering textbooks merely gloss over the EKF despite it being the most frequently used technique in real world applications.

Recently the field has been changing in exciting ways. First, computing power has grown to the point that we can use techniques that were once beyond the ability of a supercomputer. These use *Monte Carlo* techniques - the computer generates thousands to tens of thousands of random points and tests all of them against the measurements. It then probabilistically kills or duplicates points based on how well they match the measurements. A point far away from the measurement is unlikely to be retained, whereas a point very close is quite likely to be retained. After a few iterations there is a clump of particles closely tracking your object, and a sparse cloud of points where there is no object.

This has two benefits. First, the algorithm is robust even for extremely nonlinear problems. Second, the algorithm can track arbitrarily many objects at once - some particles will match the behavior of one object, and other particles will match other objects. So this technique is often used to track automobile traffic, people in crowds, and so on.

The costs should be clear. It is computationally expensive to test tens of thousands of points for every step in the filter. But modern CPUs are very fast, and this is a good problem for GPUs because the part of the algorithm is parallelizable. Another cost is that the answer is not mathematical. With a Kalman filter my covariance matrix gives me important information about the amount of error in the estimate. The particle filter does not give me a rigorous way to compute this. Finally, the output of the filter is a cloud of points; I then have to figure out how to interpret it. Usually you will be doing something like taking the mean and standard deviations of the points, but this is a difficult problem. There are still many points that do not 'belong' to a tracked object, so you first have to run some sort of clustering algorithm to first find the points that seem to be tracking an object, and then you need another algorithm to produce a state estimate from those points. None of this is intractable, but it is all quite computationally expensive.

Finally, we have a new algorithm called the *unscented Kalman filter* (UKF). It does not require you to find analytic solutions to nonlinear equations, and yet almost always performs better than the EKF. It does well with nonlinear problems - problems where the EKF has significant difficulties. Designing the filter is extremely easy. Some will say the jury is still out on the UKF, but to my mind the UKF is superior in almost every way to the EKF. I suggest that the UKF should be the starting point for any implementation, especially if you are not a Kalman filter professional with a graduate degree in control theory. The main downside is that the UKF can be a few times slower than the EKF, but this really depends on whether the EKF solves the Jacobian analytically or numerically. If numerically the UKF is almost certainly faster. It has not been proven (and probably it cannot be proven) that the UKF always yields more accurate results than the EKF. In practice it almost always does, often significantly so. It is very easy to understand and implement, and I strongly suggest this filter as your starting point.

## 9.7 Summary

The world is nonlinear, but we only really know how to solve linear problems. This introduces significant difficulties for Kalman filters. We've looked at how nonlinearity affects filtering in 3 different but equivalent

ways, and I've given you a brief summary of the major appoaches: the linearized Kalman filter, the extended Kalman filter, the Unscented Kalman filter, and the particle filter.

Until recently the linearized Kalman filter and EKF have been the standard way to solve these problems. They are very difficult to understand and use, and they are also potentially very unstable.

Recent developments have offered what are to my mind superior approaches. The UKF dispenses with the need to find solutions to partial differential equations, yet it is also usually more accurate than the EKF. It is easy to use and understand. I can get a basic UKF going in a few minutes by using FilterPy. The particle filter dispenses with mathematical modeling completely in favor of a Monte Carlo technique of generating a random cloud of thousands of points. It runs slowly, but it can solve otherwise intractable problems with relative ease.

I get more email about the EKF than anything else; I suspect that this is because most treatments in books, papers, and on the internet use the EKF. If your interest is in mastering the field of course you will want to learn about the EKF. But if you are just trying to get good results I point you to the UKF and particle filter first. They are much easier to implement, understand, and use, and they are typically far more stable than the EKF.

Some will quibble with that advice. A lot of recent publications are devoted to a comparison of the EKF, UKF, and perhaps a few other choices for a given problem. Do you not need to perform a similar comparison for your problem? If you are sending a rocket to Mars then of course you do. You will be balancing issues such as accuracy, round off errors, divergence, mathematical proof of correctness, and the computational effort required. I can't imagine not knowing the EKF intimately.

On the other hand the UKF works spectacularly! I use it at work for real world applications. I mostly haven't even tried to implement an EKF for these applications because I can verify that the UKF is working fine. Is it possible that I might eke out another 0.2% of performance from the EKF in certain situations? Sure! Do I care? No! I completely understand the UKF implementation, it is easy to test and verify, I can pass the code to others and be confident that they can understand and modify it, and I am not a masochist that wants to battle difficult equations when I already have a working solution. If the UKF or particle filters start to perform poorly for some problem then I will turn to other techniques, but not before then. And realistically, the UKF usually provides substantially better performance than the EKF over a wide range of problems and conditions. If "really good" is good enough I'm going to spend my time working on other problems.

I'm belaboring this point because in most textbooks the EKF is given center stage, and the UKF is either not mentioned at all or just given a 2 page gloss that leaves you completely unprepared to use the filter. The UKF is still relatively new, and it takes time to write new editions of books. At the time many books were written the UKF was either not discovered yet, or it was just an unproven but promising curiosity. But as I am writing this now, the UKF has had enormous success, and it needs to be in your toolkit. That is what I will spend most of my effort trying to teach you.

## 9.8 References

[1] https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/Supporting_Notebooks/Computing_and_plotting_PDFs.ipynb

# Chapter 10

# The Unscented Kalman Filter

In [1]:
```
from __future__ import division, print_function
%matplotlib inline
```

In [2]:
```
#format the book
import book_format
book_format.set_style()
```

Out[2]:

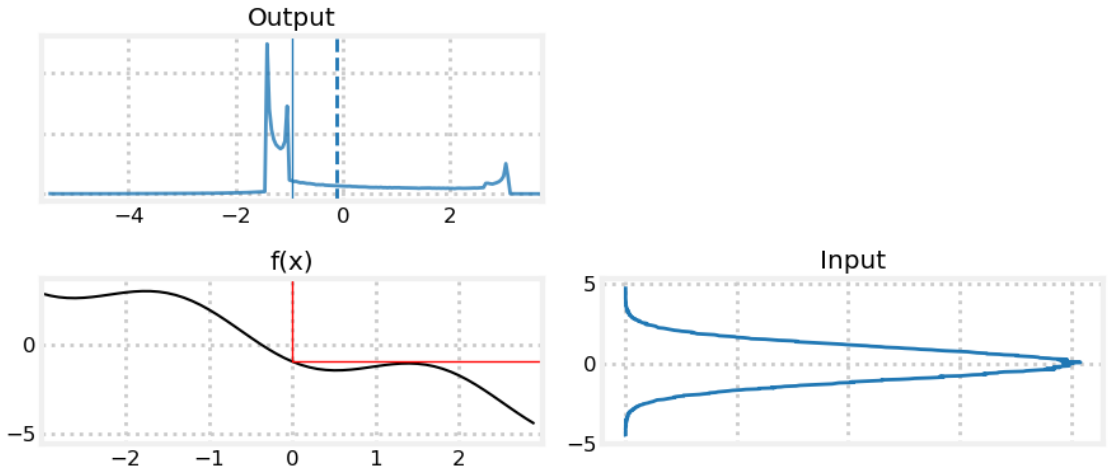
In the last chapter we discussed the difficulties that nonlinear systems pose. This nonlinearity can appear in two places. It can be in our measurements, such as a radar that is measuring the slant range to an object. Slant range requires you to take a square root to compute the x,y coordinates:

$$x = \sqrt{\text{slant}^2 - \text{altitude}^2}$$

The nonlinearity can also occur in the process model - we may be tracking a ball traveling through the air, where the effects of air drag lead to nonlinear behavior. The standard Kalman filter performs poorly or not at all with these sorts of problems.

In the last chapter I showed you a plot like this. I have altered the equation somewhat to emphasize the effects of nonlinearity.

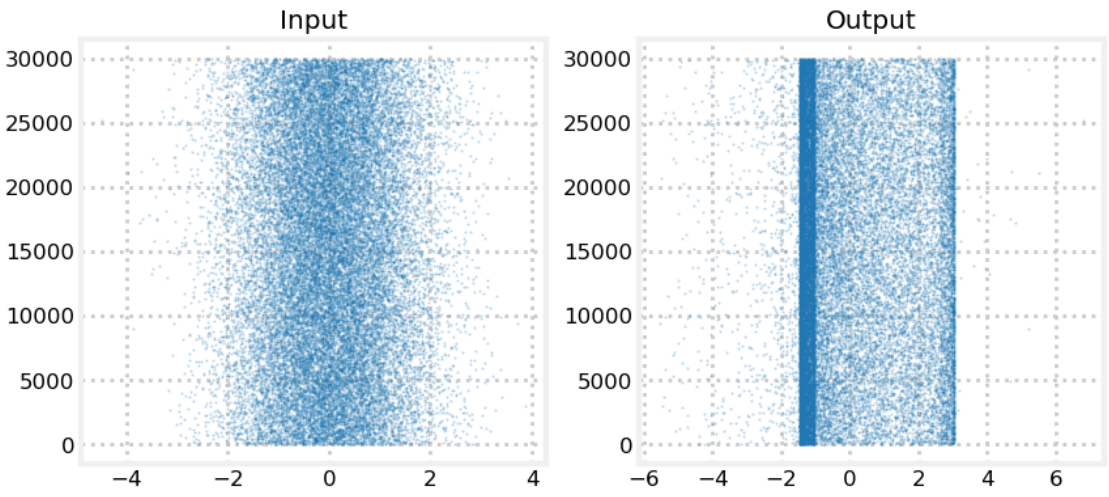
In [3]:
```
from kf_book.book_plots import set_figsize, figsize
import matplotlib.pyplot as plt
from kf_book.nonlinear_plots import plot_nonlinear_func
from numpy.random import normal
import numpy as np

# create 500,000 samples with mean 0, std 1
gaussian = (0., 1.)
data = normal(loc=gaussian[0], scale=gaussian[1], size=500000)

def f(x):
    return (np.cos(4*(x/2 + 0.7))) - 1.3*x

plot_nonlinear_func(data, f)
```




I generated this by taking 500,000 samples from the input, passing it through the nonlinear transform, and building a histogram of the result. We call these points *sigma points*. From the output histogram we can compute a mean and standard deviation which would give us an updated, albeit approximated Gaussian.

Let me show you a scatter plot of the data before and after being passed through `f(x)`.

```
In [4]: N = 30000
        plt.subplot(121)
        plt.scatter(data[:N], range(N), alpha=.2, s=1)
        plt.title('Input')
        plt.subplot(122)
        plt.title('Output')
        plt.scatter(f(data[:N]), range(N), alpha=.2, s=1);
```




The data itself appears to be Gaussian, which it is. By that I mean it looks like white noise scattered around the mean zero. In contrast `g(data)` has a defined structure. There are two bands, with a significant number of points in between. On the outside of the bands there are scattered points, but with many more on the negative side.

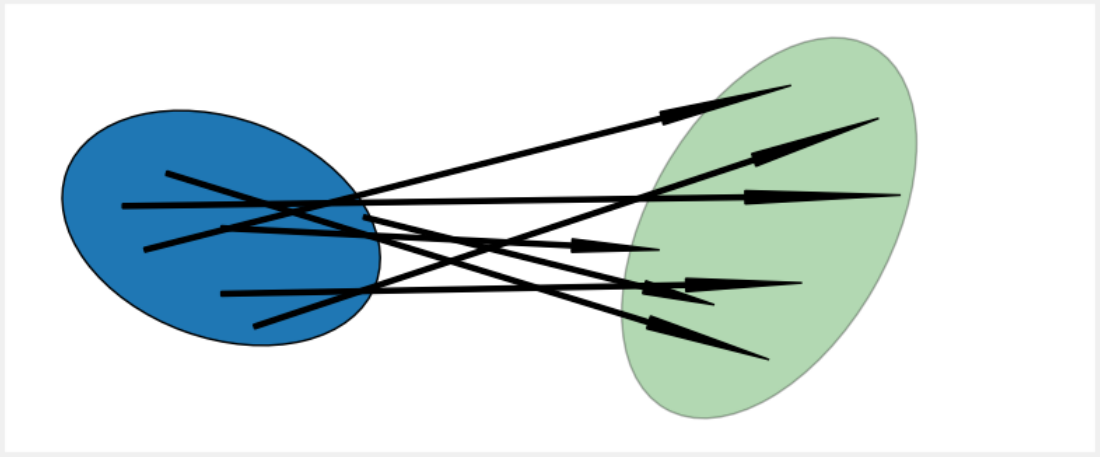
It has perhaps occurred to you that this sampling process constitutes a solution to our problem. Suppose for every update we generated 500,000 points, passed them through the function, and then computed the mean and variance of the result. This is called a *Monte Carlo* approach, and it used by some Kalman filter designs, such as the Ensemble filter and particle filter. Sampling requires no specialized knowledge, and does not require a closed form solution. No matter how nonlinear or poorly behaved the function is, as long as we sample with enough sigma points we will build an accurate output distribution.

"Enough points" is the rub. The graph above was created with 500,000 sigma points, and the output is still not smooth. What's worse, this is only for 1 dimension. The number of points required increases by the power of the number of dimensions. If you only needed 500 points for 1 dimension, you'd need 500 squared, or 250,000 points for two dimensions, 500 cubed, or 125,000,000 points for three dimensions, and so on. So while this approach does work, it is very computationally expensive. Ensemble filters and particle filters use clever techniques to significantly reduce this dimensionality, but the computational burdens are still very large. The unscented Kalman filter uses sigma points but drastically reduces the amount of computation by using a deterministic method to choose the points.

## 10.1 Sigma Points - Sampling from a Distribution

Let's look at the problem in terms of a 2D covariance ellipse. I choose 2D merely because it is easy to plot; this extends to any number of dimensions. Assuming some arbitrary nonlinear function, we will take random points from the first covariance ellipse, pass them through the nonlinear function, and plot their new position. Then we can compute the mean and covariance of the transformed points, and use that as our estimate of the mean and probability distribution.

```
In [5]: import kf_book.ukf_internal as ukf_internal
        ukf_internal.show_2d_transform()
```



On the left we show an ellipse depicting the $1\sigma$ distribution of two state variables. The arrows show how several randomly sampled points might be transformed by some arbitrary nonlinear function to a new distribution. The ellipse on the right is drawn semi-transparently to indicate that it is an *estimate* of the mean and variance of this collection of points.

Let's write a function which passes 10,000 points randomly drawn from the Gaussian

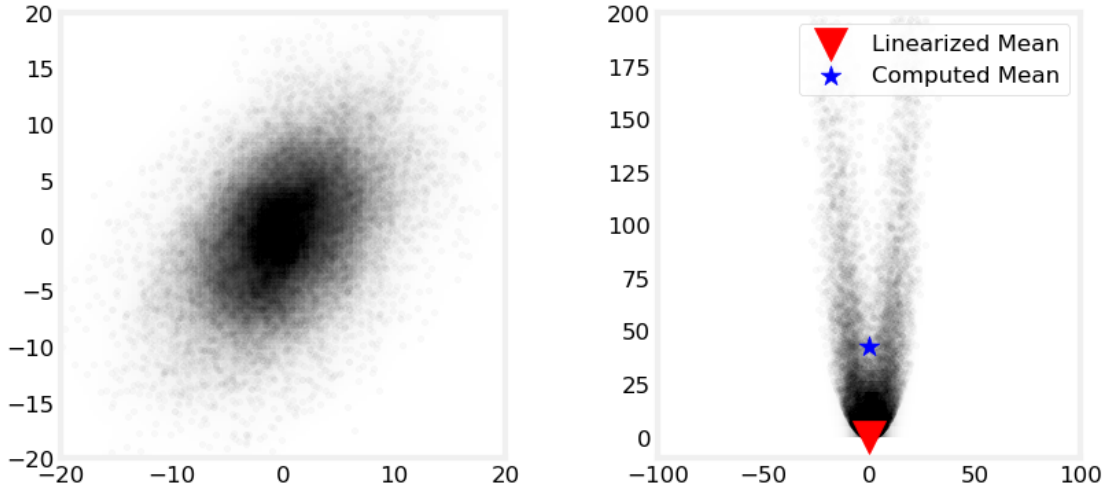
$$\mu = \begin{bmatrix}0\\0\end{bmatrix}, \Sigma = \begin{bmatrix}32 & 15\\15 & 40\end{bmatrix}$$

through the nonlinear system:

$$\begin{cases} \bar{x} = x + y \\ \bar{y} = 0.1x^2 + y^2 \end{cases}$$

```
In [6]: import numpy as np
        from numpy.random import multivariate_normal
        from kf_book.nonlinear_plots import plot_monte_carlo_mean

        def f_nonlinear_xy(x, y):
            return np.array([x + y, .1*x**2 + y*y])

        mean = (0., 0.)
        p = np.array([[32., 15.], [15., 40.]])
        # Compute linearized mean
        mean_fx = f_nonlinear_xy(*mean)

        #generate random points
        xs, ys = multivariate_normal(mean=mean, cov=p, size=10000).T
        plot_monte_carlo_mean(xs, ys, f_nonlinear_xy, mean_fx, 'Linearized Mean');
```

```
Difference in mean x=-0.055, y=42.895
```




This plot shows the strong nonlinearity that occurs with this function, and the large error that would result if we linearized in the way of the Extended Kalman filter (we will be learning this in the next chapter).

## 10.2 A Quick Example

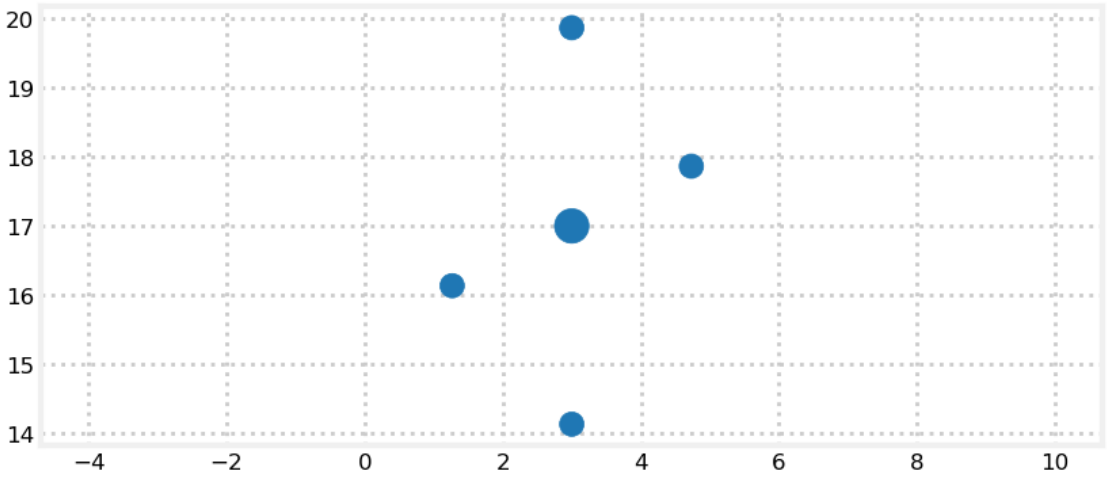
I'm soon going to proceed into the math the Unscented Kalman Filter (UKF) uses to choose sigma points and then perform the computation. But let's just start with an example so you can see the destination, so to speak.

We will learn that the UKF can use many different algorithms to generate the sigma points. Several algorithms are provided by FilterPy. Here's one possibility:

```
In [7]: from filterpy.kalman import JulierSigmaPoints

        sigmas = JulierSigmaPoints(n=2, kappa=1)
```

This will become clearer later, but the object will generate weighted sigma points for any given mean and covariance. Let's just look at an example, where the point's size indicate how much it is weighted:

```
In [8]: from kf_book.ukf_internal import plot_sigmas
        plot_sigmas(sigmas, x=[3, 17], cov=[[1, .5], [.5, 3]])
```




You can see we have 5 points centered around the mean (3, 17) in an odd pattern. It may seem absurd that this will do as well or better than 500,000 randomly generated points, but it will!

Okay, now let's implement the filter. We will implement a standard linear filter in 1D; we aren't quite ready to tackle a nonlinear filter yet. The design of the filter is not much different than what we have learned so far, with one difference. The KalmanFilter class uses the matrix **F** to compute the state transition function. Matrices mean **linear** algrebra, which work for linear problems, but not nonlinear ones. So, instead of a matrix we provide a function, just like we did above. The KalmanFilter class uses another matrix **H** to implement the measurement function, which converts a state into the equivalent measurement. Again, a matrix implies linearity, so instead of a matrix we provide a function. Perhaps it is clear why **H** is called the 'measurement function'; for the linear Kalman filter it is a matrix, but that is just a fast way to compute a function that happens to be linear.

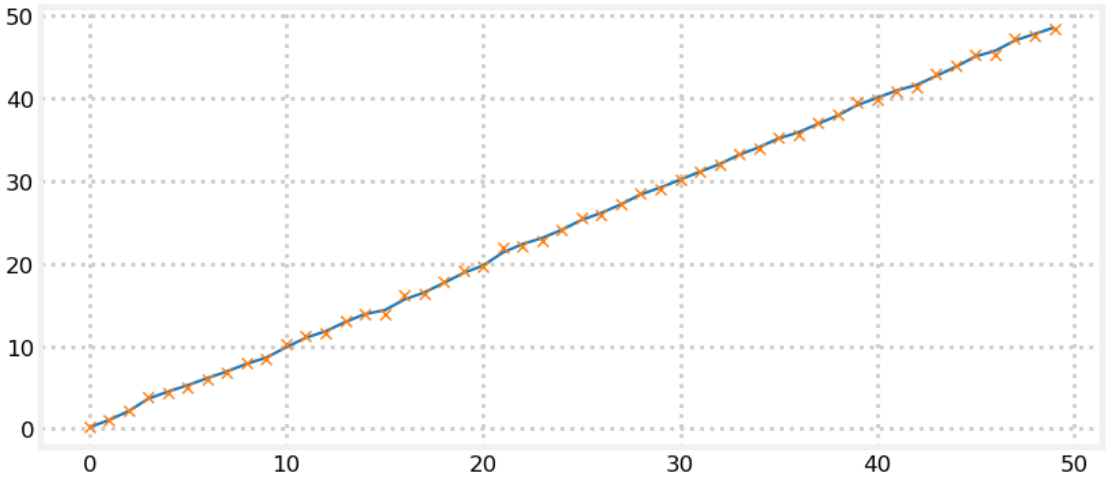
Without further ado, here are the state transistion function and measurement function for a 1D tracking problem, where the state is $\mathbf{x} = [x\ \dot{x}]^\mathsf{T}$:

```
In [9]: def fx(x, dt):
            xout = np.empty_like(x)
            xout[0] = x[1] * dt + x[0]
            xout[1] = x[1]
            return xout

        def hx(x):
            return x[:1] # return position [x]
```

Let's be clear, this is a linear example. There is no need to use a UKF for a linear problem, but I'm starting with the simplest possible example. But notice how I wrote `fx()` to compute $\mathbf{x}$ as a set of equations instead of a matrix multiplication. This is to illustrate that I could implement any arbitrary nonlinear function here; we are not constrained to linear equations.

The rest of the design is the same. Design **P**, **R**, and **Q**. You know how to do that, so let's just finish up the filter and run it.

```
In [10]: from numpy.random import randn
         from filterpy.kalman import UnscentedKalmanFilter
         from filterpy.common import Q_discrete_white_noise

         ukf = UnscentedKalmanFilter(dim_x=2, dim_z=1, dt=1., hx=hx, fx=fx, points=sigmas)
         ukf.P *= 10
         ukf.R *= .5
         ukf.Q = Q_discrete_white_noise(2, dt=1., var=0.03)

         zs, xs = [], []
         for i in range(50):
             z = i + randn()*.5
             ukf.predict()
             ukf.update(z)
             xs.append(ukf.x[0])
             zs.append(z)

         plt.plot(xs);
         plt.plot(zs, marker='x', ls='');
```




There really isn't a lot new here. You have to create an object that creates sigma points for you, and provide functions instead of matrices for **F** and **H**, but the rest is the same as before. This should give you enough confidence to plow through a bit of math and algorithms so you can understand what the UKF is doing.

## 10.3 Choosing Sigma Points

At the start of the chapter I used 500,000 randomly generated sigma points to compute the the probability distribution of a Gaussian passed through a nonlinear function. While the computed mean is quite accurate, computing 500,000 points for every update would cause our filter to be extremely slow. So, what would be fewest number of sampled points that we can use, and what kinds of constraints does this problem formulation put on the points? We will assume that we have no special knowledge about the nonlinear function as we want to find a generalized algorithm that works for any function.

Let's consider the simplest possible case and see if it offers any insight. The simplest possible system is the **identity function**: $f(x) = x$. If our algorithm does not work for the identity function then the filter

cannot converge. In other words, if the input is 1 (for a one dimensional system), the output must also be 1. If the output was different, such as 1.1, then when we fed 1.1 into the transform at the next time step, we'd get out yet another number, maybe 1.23. This filter diverges.

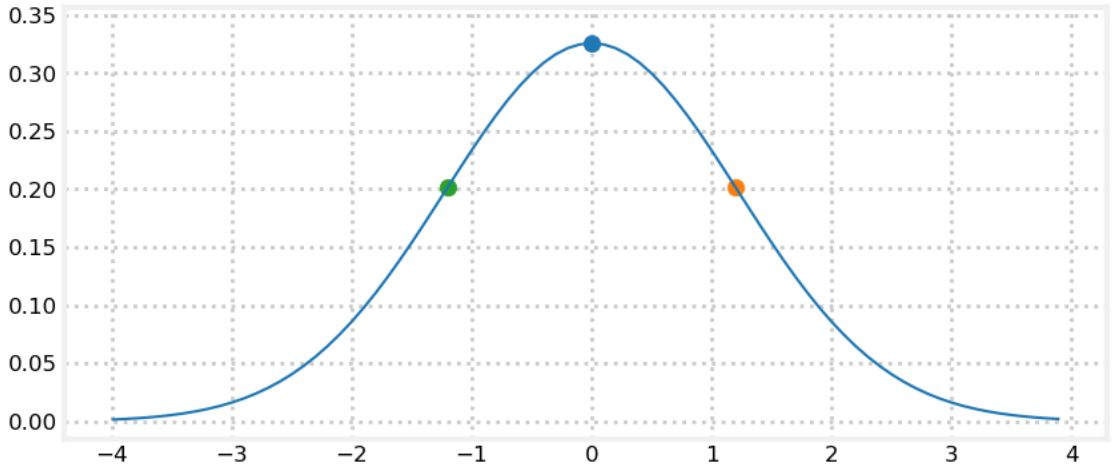
The fewest number of points that we can use is one per dimension. This is the number that the linear Kalman filter uses. The input to a Kalman filter for the distribution $\mathcal{N}(\mu, \sigma^2)$ is $\mu$ itself. So while this works for the linear case, it is not a good answer for the nonlinear case.

Perhaps we can use one point per dimension, but altered somehow. However, if we were to pass some value $\mu + \Delta$ into the identity function $f(x) = x$ it would not converge, so this will not work. If we didn't alter $\mu$ then this would be the standard Kalman filter. We must conclude that one sample will not work.

What is the next lowest number we can choose? Two. Consider the fact that Gaussians are symmetric, and that we probably want to always have one of our sample points be the mean of the input for the identity function to work. Two points would require us to select the mean, and then one other point. That one other point would introduce an asymmetry in our input that we probably don't want. It would be very difficult to make this work for the identity function $f(x) = x$.

The next lowest number is 3 points. 3 points allows us to select the mean, and then one point on each side of the mean, as depicted on the chart below.

```
In [11]: ukf_internal.show_3_sigma_points()
```




We can pass these points through a nonlinear function f(x) and compute the resulting mean and variance. The mean can be computed as the average of the 3 points, but that is not very general. For example, for a very nonlinear problem we might want to weight the center point much higher than the outside points, or we might want to weight the outside points higher.

A more general approach is to compute the weighted mean $\mu = \sum_i w_i f(\mathcal{X}_i)$, where the calligraphic $\mathcal{X}$ are the sigma points. We need the sums of the weights to equal one. Given that requirement, our task is to select $\mathcal{X}$ and their corresponding weights so that they compute to the mean and variance of the transformed sigma points.

If we weight the means it also makes sense to weight the covariances. It is possible to use different weights for the mean ($w^m$) and for the covariance ($w^c$). I use superscripts to allow space for indexes in the following equations. We can write

**Constraints** :

$$1 = \sum_i w_i^m$$

$$1 = \sum_i w_i^c$$

$$\mu = \sum_i w_i^m f(\mathcal{X}_i)$$

$$\Sigma = \sum_i w_i^c (f(\mathcal{X})_i - \mu)(f(\mathcal{X})_i - \mu)^\mathsf{T}$$

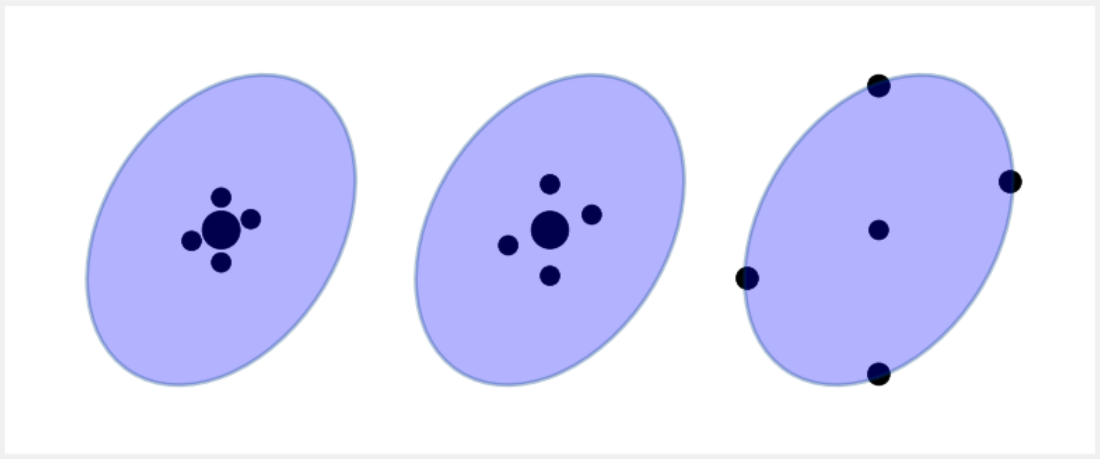
The first two equations are the constraint that the weights must sum to one. The third equation is how you compute a weight mean. The forth equation may be less familiar, but recall that the equation for the covariance of two random variables is:

$$COV(x, y) = \frac{\sum(x - \bar{x})(y - \bar{y})}{n}$$

These constraints do not form a unique solution. For example, if you make $w_0^m$ smaller you can compensate by making $w_1^m$ and $w_2^m$ larger. You can use different weights for the mean and covariances, or the same weights. Indeed, these equations do not require that any of the points be the mean of the input at all, though it seems 'nice' to do so, so to speak.

We want an algorithm that satisfies the constraints, preferably with only 3 points per dimension. Before we go on I want to make sure the idea is clear. Below are three different examples for the same covariance ellipse with different sigma points. The size of the sigma points is proportional to the weight given to each.

```
In [12]: ukf_internal.show_sigma_selections()
```



The points do not lie along the major and minor axis of the ellipse; nothing in the constraints require me to do that. I show the points evenly spaced, but the constraints do not require that.

The arrangement and weighting of the sigma points affect how we sample the distribution. Points that are close together will sample local effects, and thus probably work better for very nonlinear problems. Points that are far apart, or far off the axis of the ellipse will sample non-local effects and non Gaussian behavior. However, by varying the weights used for each point we can mitigate this. If the points are far from the mean but weighted very slightly we will incorporate some of the knowledge about the distribution without allowing the nonlinearity of the problem to create a bad estimate.

Please understand there are infinite ways to select sigma points. The constraints I choose are just one way to do it. For example, not all algorithms for creating the sigma points require the weights to sum to one. In fact, the algorithm I favor in this book does not have that property.

## 10.4 The Unscented Transform

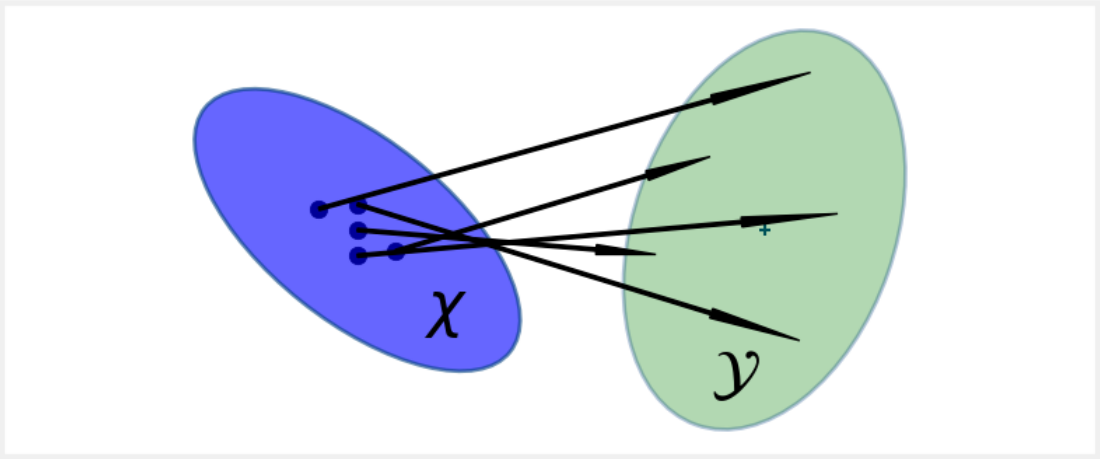
For the moment, assume an algorithm for selecting the sigma points and weights exists. How are the sigma points used to implement a filter?

The *unscented transform* is the core of the algorithm yet it is remarkably simple. It passes the sigma points $\boldsymbol{\chi}$ through a nonlinear function yielding a transformed set of points.

$$\boldsymbol{\mathcal{Y}} = f(\boldsymbol{\chi})$$

It then computes the mean and covariance of the transformed points. That mean and covariance becomes the new estimate. The figure below depicts the operation of the unscented transform. The green ellipse on the right represents the computed mean and covariance to the transformed sigma points.

```
In [13]: ukf_internal.show_sigma_transform(with_text=True)
```




The mean and covariance of the sigma points are computed as:

$$\mu = \sum_{i=0}^{2n} w_i^m \boldsymbol{\mathcal{Y}}_i$$

$$\Sigma = \sum_{i=0}^{2n} w_i^c (\boldsymbol{\mathcal{Y}}_i - \mu)(\boldsymbol{\mathcal{Y}}_i - \mu)^\mathsf{T}$$

These equations should be familar - they are the constraint equations we developed above.

In short, the unscented transform takes points sampled from some arbitary probability distribution, passes them through an arbitrary, nonlinear function and produces a Gaussian for each transformed points. I hope you can envision how we can use this to implement a nonlinear Kalman filter. Once we have Gaussians all of the mathematical apparatus we have already developed comes into play!

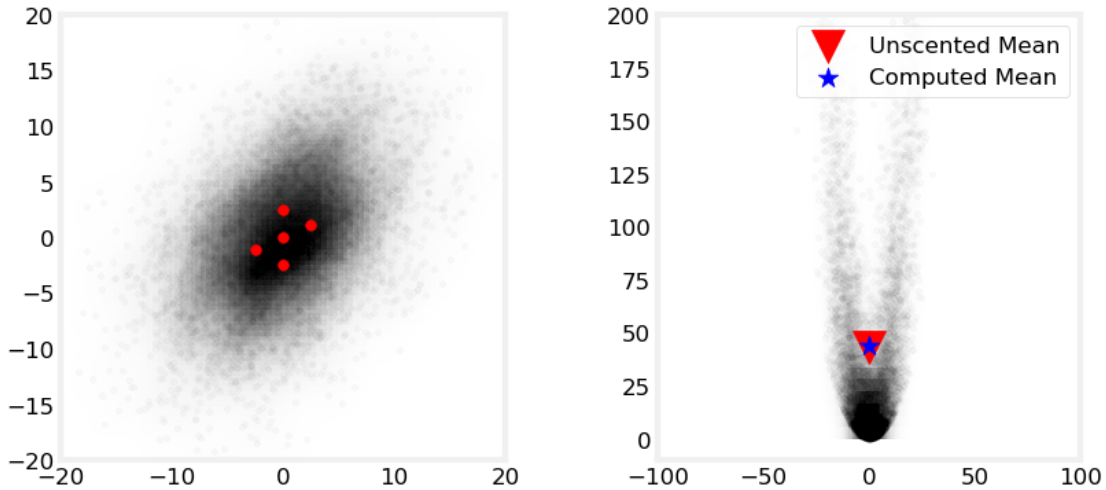
The name "unscented" might be confusing. It doesn't really mean much. It was a joke fostered by the inventor that his algorithm didn't "stink", and soon the name stuck. There is no mathematical meaning to the term.

### 10.4.1 Accuracy of the Unscented Transform

Earlier we wrote a function that found the mean of a distribution by passing 50,000 points through a nonlinear function. Let's now pass 5 sigma points through the same function, and compute their mean with the unscented transform. We will use the FilterPy function *MerweScaledSigmaPoints()* to create the sigma points and `unscented_transform` to perform the transform; we will learn about these functions later. In the first example in this chapter I used `JulierSigmaPoints`; they both choose sigma points, but in different ways which I will explain later.

```
In [14]: from filterpy.kalman import unscented_transform, MerweScaledSigmaPoints
         import scipy.stats as stats


         #initial mean and covariance
         mean = (0., 0.)
         p = np.array([[32., 15], [15., 40.]])

         # create sigma points and weights
         points = MerweScaledSigmaPoints(n=2, alpha=.3, beta=2., kappa=.1)
         sigmas = points.sigma_points(mean, p)

         ### pass through nonlinear function
         sigmas_f = np.empty((5, 2))
         for i in range(5):
             sigmas_f[i] = f_nonlinear_xy(sigmas[i, 0], sigmas[i ,1])

         ### use unscented transform to get new mean and covariance
         ukf_mean, ukf_cov = unscented_transform(sigmas_f, points.Wm, points.Wc)

         #generate random points
         np.random.seed(100)
         xs, ys = multivariate_normal(mean=mean, cov=p, size=5000).T

         plot_monte_carlo_mean(xs, ys, f_nonlinear_xy, ukf_mean, 'Unscented Mean')
         ax = plt.gcf().axes[0]
         ax.scatter(sigmas[:,0], sigmas[:,1], c='r', s=30);
```

```
Difference in mean x=-0.097, y=0.549
```



I find this result remarkable. Using only 5 points we were able to compute the mean with amazing accuracy. The error in x is only -0.097, and the error in y is 0.549. In contrast, a linearized approach (used by the EKF, which we will learn in the next chapter) gave an error of over 43 in y. If you look at the code that generates the sigma points you'll see that it has no knowledge of the nonlinear function, only of the mean and covariance of our initial distribution. The same 5 sigma points would be generated if we had a completely different nonlinear function.

I will admit to choosing a nonlinear function that makes the performance of the unscented tranform striking compared to the EKF. But the physical world is filled with very nonlinear behavior, and the UKF takes it in stride. I did not 'work' to find a function where the unscented transform happened to work well. You will see in the next chapter how more traditional techniques struggle with strong nonlinearities. This graph is the foundation of why I advise you to use the UKF or similar modern technique whenever possible.

## 10.5 The Unscented Kalman Filter

We can now present the UKF algorithm.

### 10.5.1 Predict Step

The UKF's predict step computes the prior using the process model $f()$. $f()$ is assumed to be nonlinear, so we generate sigma points $\mathcal{X}$ and their corresponding weights $W^m, W^c$ according to some function:

$$\boldsymbol{\chi} = \text{sigma-function}(\mathbf{x}, \mathbf{P})$$
$$W^m, W^c = \text{weight-function}(\mathtt{n}, \mathtt{parameters})$$

We pass each sigma point through $f(\mathbf{x}, \Delta t)$. This projects the sigma points forward in time according to the process model, forming the new prior, which is a set of sigma points we name $\boldsymbol{\mathcal{Y}}$:

$$\boldsymbol{\mathcal{Y}} = f(\boldsymbol{\chi}, \Delta t)$$

We compute the mean and covariance of the prior using the *unscented transform* on the transformed sigma points.

$$\mathbf{x}, \mathbf{P} = UT(\mathcal{Y}, w_m, w_c, \mathbf{Q})$$

These are the equations for the unscented transform:

$$\mathbf{x} = \sum_{i=0}^{2n} w_i^m \boldsymbol{\mathcal{Y}}_i$$
$$\mathbf{P} = \sum_{i=0}^{2n} w_i^c (\boldsymbol{\mathcal{Y}}_i - \mathbf{x})(\boldsymbol{\mathcal{Y}}_i - \mathbf{x})^\mathsf{T} + \mathbf{Q}$$

This table compares the linear Kalman filter with the Unscented Kalman Filter equations. I've dropped the subscript $i$ for readability.

| Kalman | Unscented |
|---|---|
| | $\boldsymbol{\mathcal{Y}} = f(\boldsymbol{\chi})$ |
| $\mathbf{x} = \mathbf{Fx}$ | $\mathbf{x} = \sum w^m \boldsymbol{\mathcal{Y}}$ |
| $\mathbf{P} = \mathbf{FPF}^\mathsf{T} + \mathbf{Q}$ | $\mathbf{P} = \sum w^c (\boldsymbol{\mathcal{Y}} - \mathbf{x})(\boldsymbol{\mathcal{Y}} - \mathbf{x})^\mathsf{T} + \mathbf{Q}$ |

### 10.5.2 Update Step

Kalman filters perform the update in measurement space. Thus we must convert the sigma points of the prior into measurements using a measurement function $h(x)$ that you define.

$$\boldsymbol{\mathcal{Z}} = h(\boldsymbol{\mathcal{Y}})$$

We compute the mean and covariance of these points using the unscented transform. The $z$ subscript denotes that these are the mean and covariance of the measurement sigma points.

$$\begin{aligned}
\boldsymbol{\mu}_z, \mathbf{P}_z &= UT(\boldsymbol{\mathcal{Z}}, w_m, w_c, \mathbf{R}) \\
\boldsymbol{\mu}_z &= \sum_{i=0}^{2n} w_i^m \boldsymbol{\mathcal{Z}}_i \\
\mathbf{P}_z &= \sum_{i=0}^{2n} w_i^c (\boldsymbol{\mathcal{Z}}_i - \boldsymbol{\mu}_z)(\boldsymbol{\mathcal{Z}}_i - \boldsymbol{\mu}_z)^\mathsf{T} + \mathbf{R}
\end{aligned}$$

Next we compute the residual and Kalman gain. The residual of the measurement $\mathbf{z}$ is trivial to compute:

$$\mathbf{y} = \mathbf{z} - \boldsymbol{\mu}_z$$

To compute the Kalman gain we first compute the cross covariance of the state and the measurements, which is defined as:

$$\mathbf{P}_{xz} = \sum_{i=0}^{2n} w_i^c (\boldsymbol{\mathcal{Y}}_i - \mathbf{x})(\boldsymbol{\mathcal{Z}}_i - \boldsymbol{\mu}_z)^\mathsf{T}$$

And then the Kalman gain is defined as

$$\mathbf{K} = \mathbf{P}_{xz}\mathbf{P}_z^{-1}$$

If you think of the inverse as a *kind of* matrix reciprocal, you can see that the Kalman gain is a simple ratio which computes:

$$\mathbf{K} \approx \frac{\mathbf{P}_{xz}}{\mathbf{P}_z} \approx \frac{\text{belief in state}}{\text{belief in measurement}}$$

Finally, we compute the new state estimate using the residual and Kalman gain:

$$\mathbf{x} = \bar{\mathbf{x}} + \mathbf{K}\mathbf{y}$$

and the new covariance is computed as:

$$\mathbf{P} = \mathbf{P} - \mathbf{K}\mathbf{P}_\mathbf{z}\mathbf{K}^\mathsf{T}$$

This step contains a few equations you have to take on faith, but you should be able to see how they relate to the linear Kalman filter equations. The linear algebra is slightly different from the linear Kalman filter, but the algorithm is the same Bayesian algorithm we have been implementing throughout the book.

This table compares the equations of the linear KF and UKF equations.

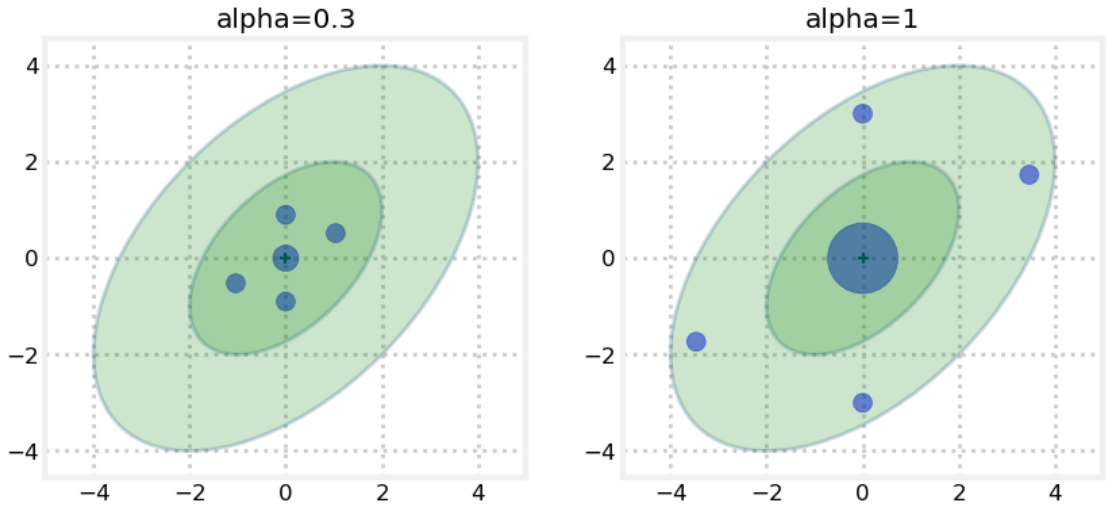
| Kalman Filter | Unscented Kalman Filter |
|---|---|
| | $\boldsymbol{\mathcal{Y}} = f(\boldsymbol{\chi})$ |
| $\mathbf{x} = \mathbf{Fx}$ | $\mathbf{x} = \sum w^m \boldsymbol{\mathcal{Y}}$ |
| $\mathbf{P} = \mathbf{FPF}^\mathsf{T} + \mathbf{Q}$ | $\mathbf{P} = \sum w^c (\boldsymbol{\mathcal{Y}} - \mathbf{x})(\boldsymbol{\mathcal{Y}} - \mathbf{x})^\mathsf{T} + \mathbf{Q}$ |
| | $\boldsymbol{\mathcal{Z}} = h(\boldsymbol{\mathcal{Y}})$ |
| | $\boldsymbol{\mu}_z = \sum w^m \boldsymbol{\mathcal{Z}}$ |
| $\mathbf{y} = \mathbf{z} - \mathbf{Hx}$ | $\mathbf{y} = \mathbf{z} - \boldsymbol{\mu}_z$ |
| $\mathbf{S} = \mathbf{HPH}^\mathsf{T} + \mathbf{R}$ | $\mathbf{P}_z = \sum w^c (\boldsymbol{\mathcal{Z}} - \boldsymbol{\mu}_z)(\boldsymbol{\mathcal{Z}} - \boldsymbol{\mu}_z)^\mathsf{T} + \mathbf{R}$ |
| $\mathbf{K} = \mathbf{PH}^\mathsf{T}\mathbf{S}^{-1}$ | $\mathbf{K} = \left[\sum w^c (\boldsymbol{\mathcal{Y}} - \bar{\mathbf{x}})(\boldsymbol{\mathcal{Z}} - \boldsymbol{\mu}_z)^\mathsf{T}\right] \mathbf{P}_z^{-1}$ |
| $\mathbf{x} = \mathbf{x} + \mathbf{Ky}$ | $\mathbf{x} = \mathbf{x} + \mathbf{Ky}$ |
| $\mathbf{P} = (\mathbf{I} - \mathbf{KH})\mathbf{P}$ | $\mathbf{P} = \bar{\mathbf{P}} - \mathbf{KP_z K}^\mathsf{T}$ |

## 10.6 Van der Merwe's Scaled Sigma Point Algorithm

There are many algorithms for selecting sigma points. Since 2005 or so research and industry have mostly settled on the version published by Rudolph Van der Merwe in his 2004 PhD dissertation [1]. It performs well with a variety of problems and it has a good tradeoff between performance and accuracy. It is a slight reformulation of the *Scaled Unscented Transform* published by Simon J. Julier [2].

This formulation uses 3 parameters to control how the sigma points are distributed and weighted: $\alpha$, $\beta$, and $\kappa$. Before we work through the equations, let's look at an example. I will plot the sigma points on top of a covariance ellipse showing the first and second standard deviations, and scale the points based on the mean weights.

```
In [15]: ukf_internal.plot_sigma_points()
```




We can see that the sigma points lie between the first and second standard deviation, and that the larger $\alpha$ spreads the points out. Furthermore, the larger $\alpha$ weights the mean (center point) higher than the smaller $\alpha$, and weights the rest less. This should fit our intuition - the further a point is from the mean the less we should weight it. We don't know how these weights and sigma points are selected yet, but the choices look reasonable.

### 10.6.1 Sigma Point Computation

The first sigma point is the mean of the input. This is the sigma point displayed in the center of the ellipses in the diagram above. We will call this $\boldsymbol{\chi}_0$.

$$\mathcal{X}_0 = \mu$$

For notational convenience we define $\lambda = \alpha^2(n + \kappa) - n$, where $n$ is the dimension of $\mathbf{x}$. The remaining sigma points are computed as

$$\boldsymbol{\chi}_i = \begin{cases} \mu + \left[\sqrt{(n+\lambda)\Sigma}\right]_i & \text{for i=1 .. n} \\ \mu - \left[\sqrt{(n+\lambda)\Sigma}\right]_{i-n} & \text{for i=(n+1) .. 2n} \end{cases}$$

The $i$ subscript chooses the i$^{th}$ roth vector of the matrix.

In other words, we scale the covariance matrix by a constant, take the square root of it, and ensure symmetry by both adding and subtracting it from the mean. We will discuss how you take the square root of a matrix later.

### 10.6.2 Weight Computation

This formulation uses one set of weights for the means, and another set for the covariance. The weights for the mean of $\mathcal{X}_0$ is computed as

$$W_0^m = \frac{\lambda}{n+\lambda}$$

The weight for the covariance of $\mathcal{X}_0$ is

$$W_0^c = \frac{\lambda}{n+\lambda} + 1 - \alpha^2 + \beta$$

The weights for the rest of the sigma points $\boldsymbol{\chi}_1 ... \boldsymbol{\chi}_{2n}$ are the same for the mean and covariance. They are

$$W_i^m = W_i^c = \frac{1}{2(n+\lambda)} \quad i = 1..2n$$

It may not be obvious why this is 'correct', and indeed, it cannot be proven that this is ideal for all nonlinear problems. But you can see that we are choosing the sigma points proportional to the square root of the covariance matrix, and the square root of variance is standard deviation. So, the sigma points are spread roughly according to $\pm 1\sigma$ times some scaling factor. There is an $n$ term in the denominator, so with more dimensions the points will be spread out and weighed less.

**Important note:** Ordinarily these weights do not sum to one. I get many questions about this. Getting weights that sum to greater than one, or even negative values is expected. I cover this in more detail below.

### 10.6.3 Reasonable Choices for the Parameters

$\beta = 2$ is a good choice for Gaussian problems, $\kappa = 3 - n$ where $n$ is the dimension of $\mathbf{x}$ is a good choice for $\kappa$, and $0 \leq \alpha \leq 1$ is an appropriate choice for $\alpha$, where a larger value for $\alpha$ spreads the sigma points further from the mean.

## 10.7 Using the UKF

Let's solve some problems so you can gain confidence in how easy the UKF is to use. We will start with a linear problem you already know how to solve with the linear Kalman filter. Although the UKF was designed for nonlinear problems, it finds the same optimal result as the linear Kalman filter for linear problems. We will write a filter to track an object in 2D using a constant velocity model. This will allow us to focus on what is the same (and most is the same!) and what is different with the UKF.

Designing a Kalman filter requires you to specify the $\mathbf{x}$, $\mathbf{F}$, $\mathbf{H}$, $\mathbf{R}$, and $\mathbf{Q}$ matrices. We have done this many times so I will give you the matrices without a lot of discussion. We want a constant velocity model, so we define $\mathbf{x}$ to be

$$\mathbf{x} = \begin{bmatrix} x & \dot{x} & y & \dot{y} \end{bmatrix}^\mathsf{T}$$

With this ordering of state variables the state transition matrix is

$$\mathbf{F} = \begin{bmatrix} 1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

which implement the Newtonian equations

$$x_k = x_{k-1} + \dot{x}_{k-1}\Delta t$$
$$y_k = y_{k-1} + \dot{y}_{k-1}\Delta t$$

Our sensors provide position but not velocity, so the measurement function is

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

The sensor readings are in meters with an error of $\sigma = 0.3$ meters in both $x$ and $y$. This gives us a measurement noise matrix of

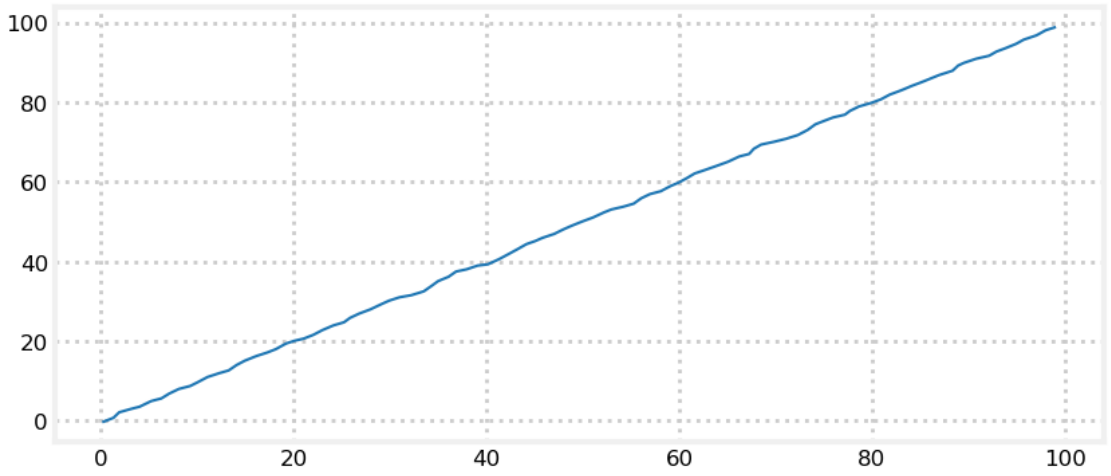
$$\mathbf{R} = \begin{bmatrix} 0.3^2 & 0 \\ 0 & 0.3^2 \end{bmatrix}$$

Finally, let's assume that the process noise can be represented by the discrete white noise model - that is, that over each time period the acceleration is constant. We can use `FilterPy`'s `Q_discrete_white_noise()` to create this matrix for us, but for review the matrix is

$$\mathbf{Q} = \begin{bmatrix} \frac{1}{4}\Delta t^4 & \frac{1}{2}\Delta t^3 \\ \frac{1}{2}\Delta t^3 & \Delta t^2 \end{bmatrix} \sigma^2$$

My implementation of this filter is:

```
In [16]: from filterpy.kalman import KalmanFilter
         from filterpy.common import Q_discrete_white_noise
         from numpy.random import randn

         std_x, std_y = .3, .3
         dt = 1.0

         np.random.seed(1234)
         kf = KalmanFilter(4, 2)
         kf.x = np.array([0., 0., 0., 0.])
         kf.R = np.diag([std_x**2, std_y**2])
         kf.F = np.array([[1, dt, 0, 0],
                          [0, 1, 0, 0],
                          [0, 0, 1, dt],
                          [0, 0, 0, 1]])
         kf.H = np.array([[1, 0, 0, 0],
                          [0, 0, 1, 0]])

         kf.Q[0:2, 0:2] = Q_discrete_white_noise(2, dt=1, var=0.02)
         kf.Q[2:4, 2:4] = Q_discrete_white_noise(2, dt=1, var=0.02)

         zs = [np.array([i + randn()*std_x,
                         i + randn()*std_y]) for i in range(100)]
         xs, _, _, _ = kf.batch_filter(zs)
         plt.plot(xs[:, 0], xs[:, 2]);
```




This should hold no surprises for you. Now let's implement a UKF. Again, this is purely for educational purposes; using a UKF for a linear problem confers no benefit. `FilterPy` implements the UKF with the class `UnscentedKalmanFilter`.

The first thing to do is implement the functions `f(x, dt)` and `h(x)`. `f(x, dt)` implements the state transition function, and `h(x)` implements the measurement function. These correspond to the matrices **F** and **H** in the linear filter.

Below is a reasonable implementation of these two functions. Each is expected to return a 1D NumPy array or list containing the result. You may give them more readable names than `f` and `h`.

```
In [17]: def f_cv(x, dt):
             """ state transition function for a
             constant velocity aircraft"""

             F = np.array([[1, dt, 0,  0],
                           [0,  1, 0,  0],
                           [0,  0, 1, dt],
                           [0,  0, 0,  1]])
             return np.dot(F, x)

         def h_cv(x):
             return x[[0, 2]]
```

Next you specify how to compute the sigma points and weights. We gave Van der Merwe's version above, but there are many different choices. FilterPy uses a class named `SigmaPoints` which must implement a method:

```
def sigma_points(self, x, P)
```

and contain attributes `Wm` and `Wc`, which hold the weights for computing the mean and covariance, respectively.

FilterPy derives the class `MerweScaledSigmaPoints` from `SigmaPoints` and implements the aforementioned methods.

When you create the UKF you will pass in the $f()$ and $h()$ functions and the sigma point object, as in this example:

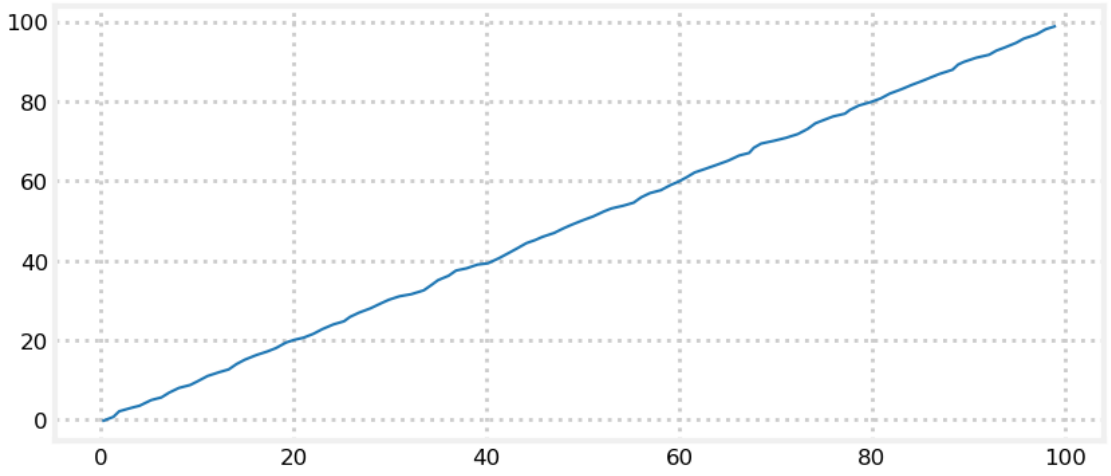
```
from filterpy.kalman import MerweScaledSigmaPoints
from filterpy.kalman import UnscentedKalmanFilter as UKF
```

```
points = MerweScaledSigmaPoints(n=4, alpha=.1, beta=2., kappa=-1)
ukf = UKF(dim_x=4, dim_z=2, fx=f_cv, hx=h_cv, dt=dt, points=points)
```

The rest of the code is the same as for the linear kalman filter. I'll use the same measurements and compute the standard deviation of the difference between the two solutions.

```
In [18]: from filterpy.kalman import UnscentedKalmanFilter as UKF

         import numpy as np

         sigmas = MerweScaledSigmaPoints(4, alpha=.1, beta=2., kappa=1.)
         ukf = UKF(dim_x=4, dim_z=2, fx=f_cv,
                   hx=h_cv, dt=dt, points=sigmas)
         ukf.x = np.array([0., 0., 0., 0.])
         ukf.R = np.diag([0.09, 0.09])
         ukf.Q[0:2, 0:2] = Q_discrete_white_noise(2, dt=1, var=0.02)
         ukf.Q[2:4, 2:4] = Q_discrete_white_noise(2, dt=1, var=0.02)

         uxs = []
         for z in zs:
             ukf.predict()
             ukf.update(z)
             uxs.append(ukf.x.copy())
         uxs = np.array(uxs)

         plt.plot(uxs[:, 0], uxs[:, 2])
         print('UKF standard deviation {:.3f} meters'.format(np.std(uxs - xs)))
```

```
UKF standard deviation 0.000 meters
```




This gave me a standard deviation of 0.013 meters, which is quite small.

The implementation of the UKF is not that different from the linear Kalman filter. Instead of implementing the state transition and measurement functions as the matrices $\mathbf{F}$ and $\mathbf{H}$ you supply nonlinear functions `f()` and `h()`. The rest of the theory and implementation remains the same. The code implementing `predict()` and `update()` differs, but from a designer's point of view the problem formulation and filter design is very similar.

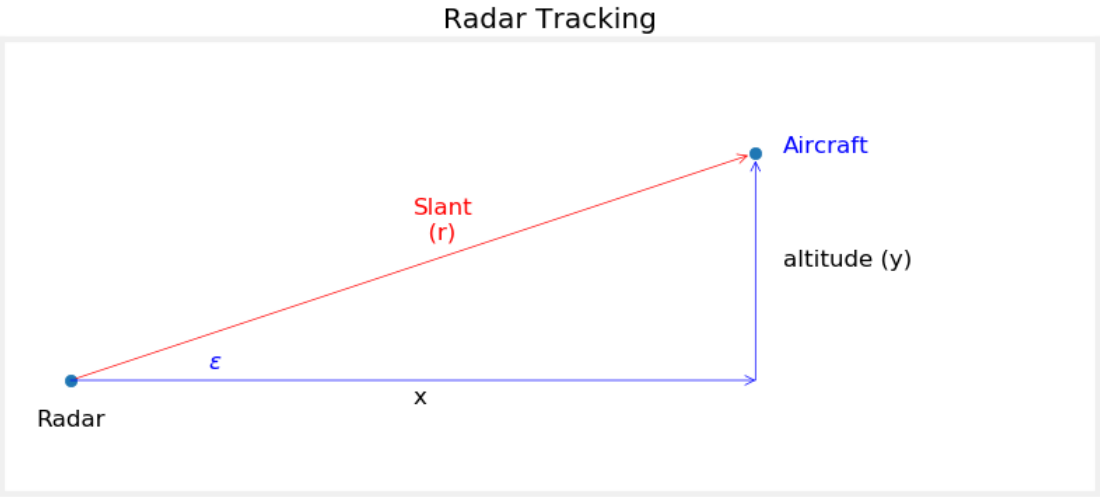
## 10.8 Tracking an Airplane

Let's tackle our first nonlinear problem. We will write a filter to track an airplane using radar as the sensor. To keep the problem as similar to the previous one as possible we will track in two dimensions. We will track one dimension on the ground and the altitude of the aircraft. Each dimension is independent so we can do this with no loss of generality.

Radars work by emitting radio waves or microwaves. Anything in the beam's path will reflect some of the signal back to the radar. By timing how long it takes for the reflected signal to return it can compute the *slant distance* to the target. Slant distance is the straight line distance from the radar to the object. Bearing is computed using the *directive gain* of the antenna.

We compute the (x,y) position of the aircraft from the slant distance and elevation angle as illustrated by this diagram:

```
In [19]: import kf_book.ekf_internal as ekf_internal
         ekf_internal.show_radar_chart()
```




The *elevation angle* $\epsilon$ is the angle above the line of sight formed by the ground.

We will assume that the aircraft is flying at a constant altitude. Thus we have a three variable state vector:

$$\mathbf{x} = \begin{bmatrix}\texttt{distance}\\ \texttt{velocity}\\ \texttt{altitude}\end{bmatrix} = \begin{bmatrix}x\\ \dot{x}\\ y\end{bmatrix}$$

The state transition function is linear

$$\mathbf{x} = \begin{bmatrix}1 & \Delta t & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{bmatrix}x\\ \dot{x}\\ y\end{bmatrix}$$

and can be computed with:

```
In [20]: def f_radar(x, dt):
             """ state transition function for a constant velocity
             aircraft with state vector [x, velocity, altitude]'"""

             F = np.array([[1, dt, 0],
```

```
                    [0,  1, 0],
                    [0,  0, 1]], dtype=float)
        return np.dot(F, x)
```

Next we design the measurement function. As in the linear Kalman filter the measurement function converts the filter's prior into a measurement. We need to convert the position and velocity of the aircraft into the elevation angle and range from the radar station.

Range is computed with the Pythagorean theorem:

$$\text{range} = \sqrt{(x_{\text{ac}} - x_{\text{radar}})^2 + (y_{\text{ac}} - y_{\text{radar}})^2}$$

The elevation angle $\epsilon$ is the arctangent of $y/x$:

$$\epsilon = \tan^{-1} \frac{y_{\text{ac}} - y_{\text{radar}}}{x_{\text{ac}} - x_{\text{radar}}}$$

We need to define a Python function to compute this. I'll take advantage of the fact that a function can own a variable to store the radar's position. While this isn't necessary for this problem (we could hard code the value, or use a global), this gives the function more flexibility.

```
In [21]: def h_radar(x):
             dx = x[0] - h_radar.radar_pos[0]
             dy = x[2] - h_radar.radar_pos[1]
             slant_range = math.sqrt(dx**2 + dy**2)
             elevation_angle = math.atan2(dy, dx)
             return [slant_range, elevation_angle]

         h_radar.radar_pos = (0, 0)
```

There is a nonlinearity that we are not considering, the fact that angles are modular. The residual is the difference between the measurement and the prior projected into measurement space. The angular difference between 359° and 1° is 2°, but 359° - 1° = 358°. This is exacerbated by the UKF which computes sums of weighted values in the unscented transform. For now we will place our sensors and targets in positions that avoid these nonlinear regions. Later I will show you how to handle this problem.

We need to simulate the radar and the aircraft. By now this should be second nature for you, so I offer the code without discussion.

```
In [22]: from numpy.linalg import norm
         from math import atan2

         class RadarStation:

             def __init__(self, pos, range_std, elev_angle_std):
                 self.pos = np.asarray(pos)
                 self.range_std = range_std
                 self.elev_angle_std = elev_angle_std


             def reading_of(self, ac_pos):
                 """ Returns (range, elevation angle) to aircraft.
                 Elevation angle is in radians.
                 """

                 diff = np.subtract(ac_pos, self.pos)
                 rng = norm(diff)
                 brg = atan2(diff[1], diff[0])
                 return rng, brg
```

```
        def noisy_reading(self, ac_pos):
            """ Compute range and elevation angle to aircraft with
            simulated noise"""

            rng, brg = self.reading_of(ac_pos)
            rng += randn() * self.range_std
            brg += randn() * self.elev_angle_std
            return rng, brg

    class ACSim:
        def __init__(self, pos, vel, vel_std):
            self.pos = np.asarray(pos, dtype=float)
            self.vel = np.asarray(vel, dtype=float)
            self.vel_std = vel_std

        def update(self, dt):
            """ Compute and returns next position. Incorporates
            random variation in velocity. """

            dx = self.vel*dt + (randn() * self.vel_std) * dt
            self.pos += dx
            return self.pos
```

A military grade radar achieves 1 meter RMS range accuracy, and 1 mrad RMS for elevation angle [1]. We will assume a more modest 5 meter range accuracy, and 0.5° angular accuracy as this provides a more challenging data set for the filter.

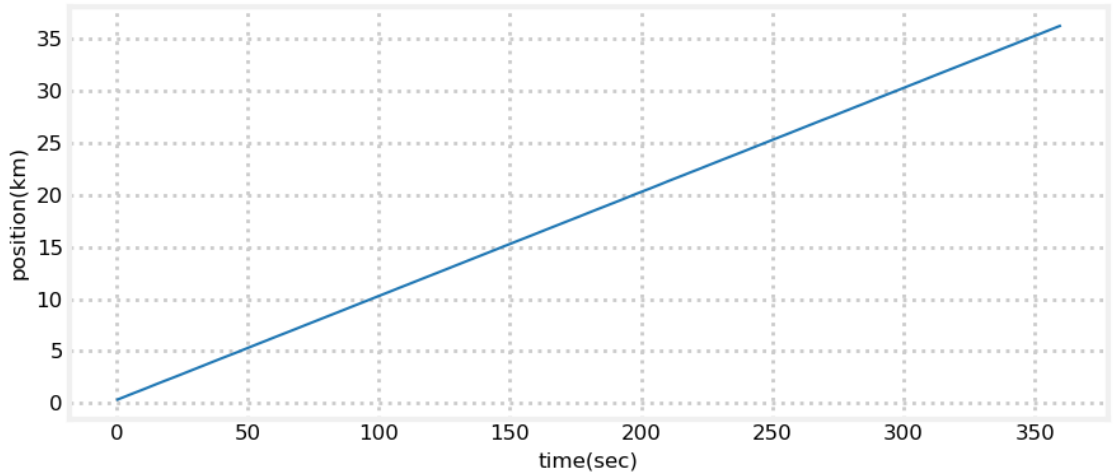
The design of $\mathbf{Q}$ requires some discussion. The state is $\begin{bmatrix}x & \dot{x} & y\end{bmatrix}^{\mathrm{T}}$. The first two elements are down range distance and velocity, so we can use `Q_discrete_white_noise` noise to compute the values for the upper left hand side of Q. The third element is altitude, which we assume is independent of $x$. That results in a block design for $\mathbf{Q}$:

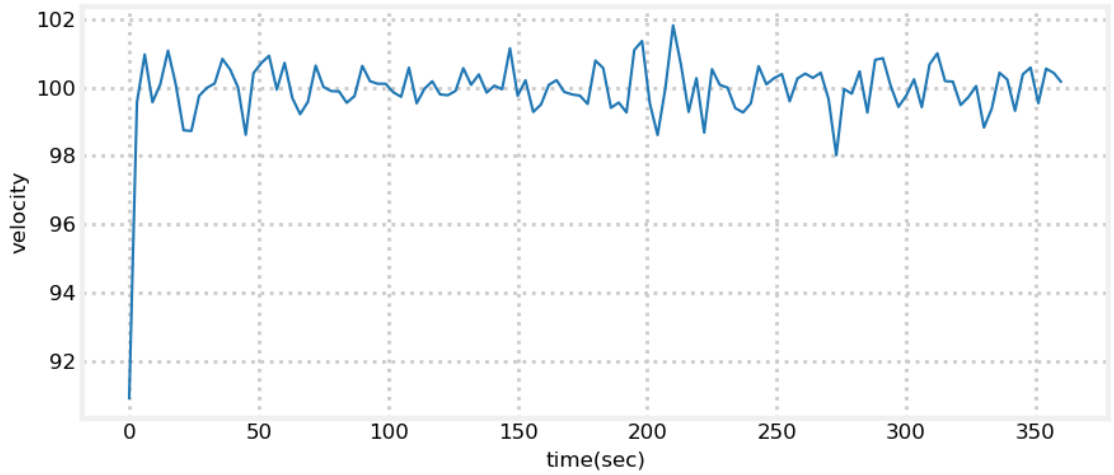
$$\mathbf{Q} = \begin{bmatrix}\mathbf{Q}_{\mathrm{x}} & \mathbf{0} \\ \mathbf{0} & Q_{\mathrm{y}}\end{bmatrix}$$

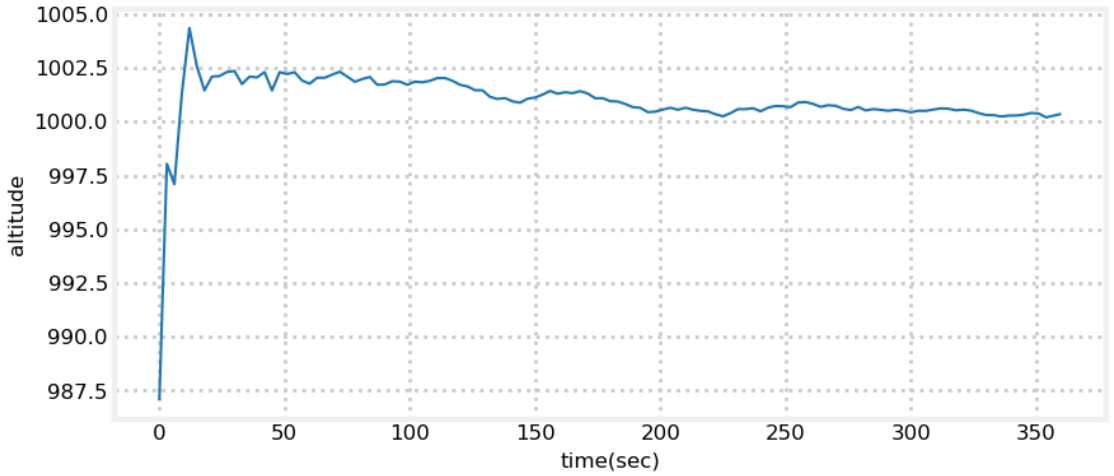
I'll start with the aircraft positioned directly over the radar station, flying at 100 m/s. A typical height finder radar might update only once every 3 seconds so we will use that for our epoch period.

In [23]:
```
import math
from kf_book.ukf_internal import plot_radar

dt = 3. # 12 seconds between readings
range_std = 5 # meters
elevation_angle_std = math.radians(0.5)
ac_pos = (0., 1000.)
ac_vel = (100., 0.)
radar_pos = (0., 0.)
h_radar.radar_pos = radar_pos

points = MerweScaledSigmaPoints(n=3, alpha=.1, beta=2., kappa=0.)
kf = UKF(3, 2, dt, fx=f_radar, hx=h_radar, points=points)

kf.Q[0:2, 0:2] = Q_discrete_white_noise(2, dt=dt, var=0.1)
kf.Q[2,2] = 0.1
```

```
kf.R = np.diag([range_std**2, elevation_angle_std**2])
kf.x = np.array([0., 90., 1100.])
kf.P = np.diag([300**2, 30**2, 150**2])

np.random.seed(200)
pos = (0, 0)
radar = RadarStation(pos, range_std, elevation_angle_std)
ac = ACSim(ac_pos, (100, 0), 0.02)

time = np.arange(0, 360 + dt, dt)
xs = []
for _ in time:
    ac.update(dt)
    r = radar.noisy_reading(ac.pos)
    kf.predict()
    kf.update([r[0], r[1]])
    xs.append(kf.x)
plot_radar(xs, time)
```





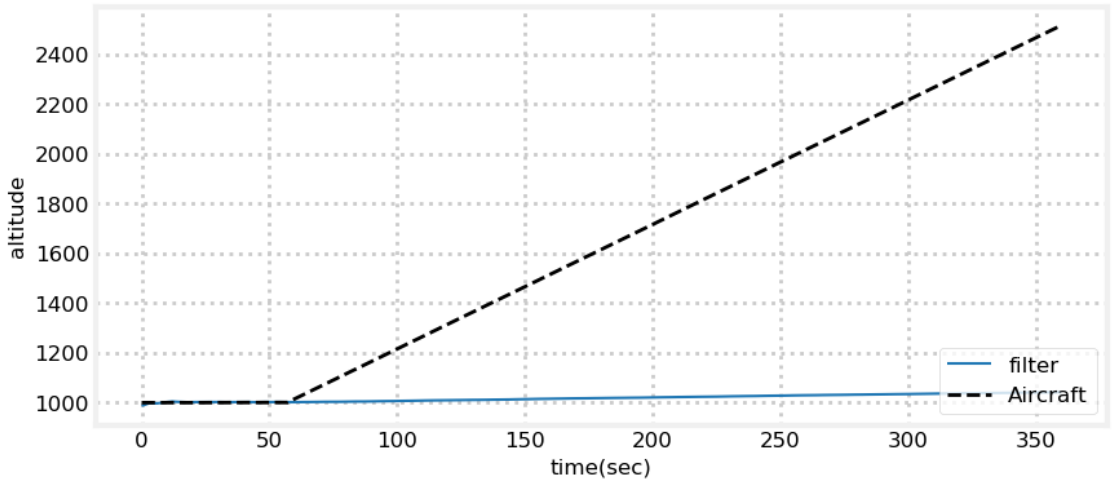

This may or may not impress you, but it impresses me! In the Extended Kalman filter chapter we will solve the same problem, but it will take a significant amount of mathematics.

### 10.8.1 Tracking Maneuevering Aircraft

The previous example produced good results, but it assumed the aircraft did not change altitude. Here are the filter results if the aircraft starts climbing after one minute.

In [24]:
```
from kf_book.ukf_internal import plot_altitude

# reset aircraft position
kf.x = np.array([0., 90., 1100.])
kf.P = np.diag([300**2, 30**2, 150**2])
ac = ACSim(ac_pos, (100, 0), 0.02)

np.random.seed(200)
time = np.arange(0, 360 + dt, dt)
xs, ys = [], []
for t in time:
    if t >= 60:
        ac.vel[1] = 300/60 # 300 meters/minute climb
    ac.update(dt)
    r = radar.noisy_reading(ac.pos)
    ys.append(ac.pos[1])
    kf.predict()
    kf.update([r[0], r[1]])
    xs.append(kf.x)

plot_altitude(xs, time, ys)
print('Actual altitude: {:.1f}'.format(ac.pos[1]))
print('UKF altitude   : {:.1f}'.format(xs[-1][2]))
```

```
Actual altitude: 2515.6
UKF altitude   : 1042.0
```




The filter is unable to track the changing altitude. What do we have to change in our design? I hope you answered "add climb rate to the state", like so:

$$\mathbf{x} = \begin{bmatrix} \texttt{distance} \\ \texttt{velocity} \\ \texttt{altitude} \\ \texttt{climb rate} \end{bmatrix} = \begin{bmatrix} x \\ \dot{x} \\ y \\ \dot{y} \end{bmatrix}$$

This requires the following change to the state transition function, which is still linear.

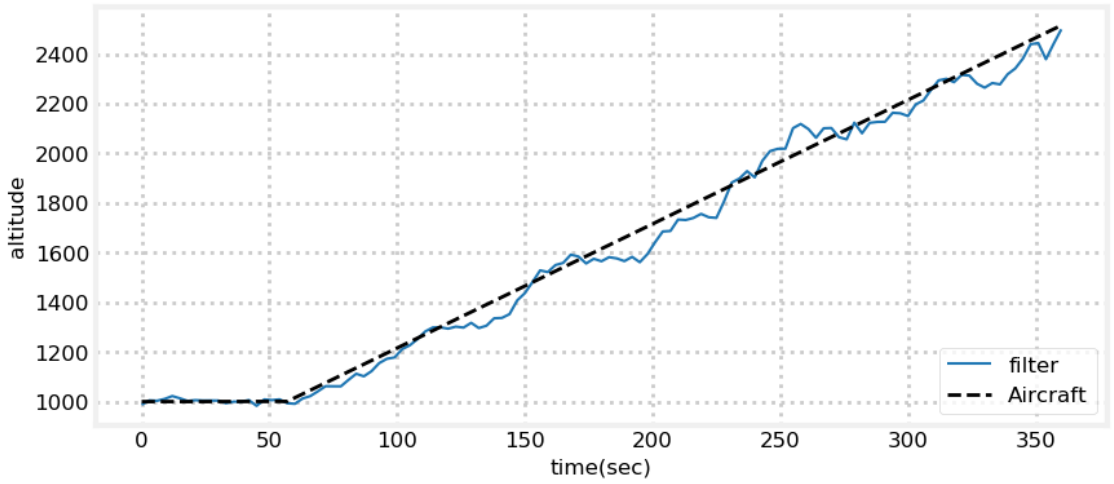
$$\mathbf{F} = \begin{bmatrix} 1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \\ y \\ \dot{y} \end{bmatrix}$$

The measurement function stays the same, but we must alter $\mathbf{Q}$ to account for the dimensionality change of $\mathbf{x}$.

```
In [25]: def f_cv_radar(x, dt):
             """ state transition function for a constant velocity
             aircraft"""
             F = np.array([[1, dt, 0, 0],
                           [0,  1, 0, 0],
                           [0,  0, 1, dt],
                           [0,  0, 0, 1]], dtype=float)
             return np.dot(F, x)

         def cv_UKF(fx, hx, R_std):
             points = MerweScaledSigmaPoints(n=4, alpha=.1, beta=2., kappa=-1.)
             kf = UKF(4, len(R_std), dt, fx=fx, hx=hx, points=points)

             kf.Q[0:2, 0:2] = Q_discrete_white_noise(2, dt=dt, var=0.1)
             kf.Q[2:4, 2:4] = Q_discrete_white_noise(2, dt=dt, var=0.1)
             kf.R = np.diag(R_std)
             kf.R = np.dot(kf.R, kf.R)  # square to get variance
             kf.x = np.array([0., 90., 1100., 0.])
             kf.P = np.diag([300**2, 3**2, 150**2, 3**2])
             return kf
```

```
In [26]: np.random.seed(200)
         ac = ACSim(ac_pos, (100, 0), 0.02)

         kf_cv = cv_UKF(f_cv_radar, h_radar, R_std=[range_std, elevation_angle_std])
         time = np.arange(0, 360 + dt, dt)
         xs, ys = [], []
         for t in time:
             if t >= 60:
                 ac.vel[1] = 300/60 # 300 meters/minute climb
             ac.update(dt)
             r = radar.noisy_reading(ac.pos)
             ys.append(ac.pos[1])
             kf_cv.predict()
             kf_cv.update([r[0], r[1]])
             xs.append(kf_cv.x)

         plot_altitude(xs, time, ys)
         print('Actual altitude: {:.1f}'.format(ac.pos[1]))
         print('UKF altitude   : {:.1f}'.format(xs[-1][2]))
```

```
Actual altitude: 2515.6
UKF altitude   : 2499.7
```




A significant amount of noise has been introduced into the altitude estimate, but we are now accurately tracking altitude.

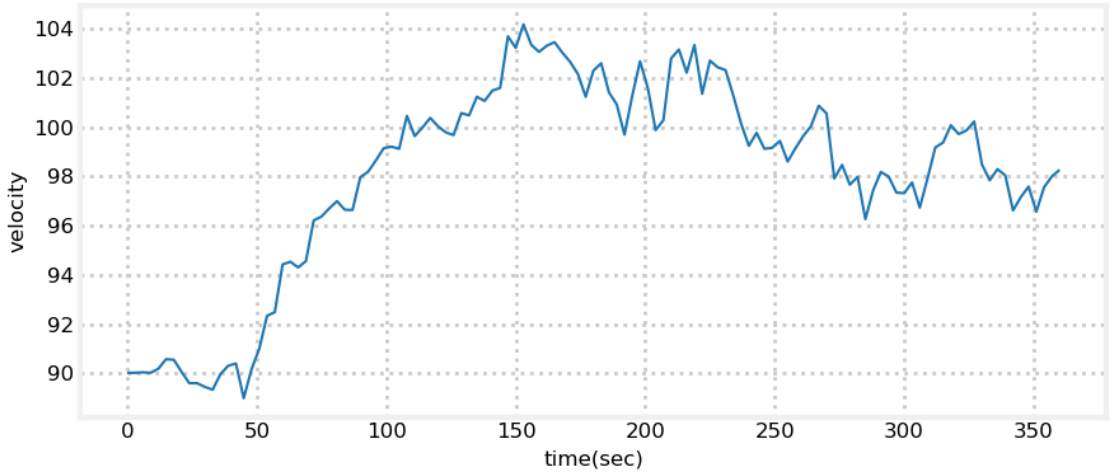
### 10.8.2 Sensor Fusion

Now let's consider an example of sensor fusion. We have some type of Doppler system that produces a velocity estimate with 2 m/s RMS accuracy. I say "some type" because as with the radar I am not trying to teach you how to create an accurate filter for a Doppler system. A full implementation must account for the signal to noise ratio, atmospheric effects, the geometry of the system, and so on.

The radar's accuracy in the last examples allowed us to estimate velocities to within one m/s or so, I will degrade that accuracy to illustrate the effect of sensor fusion. Let's change the range error to $\sigma = 500$ meters and then compute the standard deviation of the estimated velocity. I'll skip the first several measurements because the filter is converging during that time, causing artificially large deviations.

The standard deviation without using Doppler is:

```
In [27]: range_std = 500.
         elevation_angle_std = math.degrees(0.5)
         np.random.seed(200)
         pos = (0, 0)
         radar = RadarStation(pos, range_std, elevation_angle_std)
         ac = ACSim(ac_pos, (100, 0), 0.02)

         kf_sf = cv_UKF(f_cv_radar, h_radar, R_std=[range_std, elevation_angle_std])
         time = np.arange(0, 360 + dt, dt)
         xs = []
         for _ in time:
             ac.update(dt)
             r = radar.noisy_reading(ac.pos)
             kf_sf.predict()
             kf_sf.update([r[0], r[1]])
             xs.append(kf_sf.x)

         xs = np.asarray(xs)
         plot_radar(xs, time, plot_x=False, plot_vel=True, plot_alt=False)
         print('Velocity std {:.1f} m/s'.format(np.std(xs[10:, 1])))
```




```
Velocity std 3.4 m/s
```

For Doppler we need to include the velocity in $x$ and $y$ into the measurement. The `ACSim` class stores velocity in the data member `vel`. To perform the Kalman filter update we just need to call `update` with a list containing the slant distance, elevation angle, and velocity in $x$ and $y$:

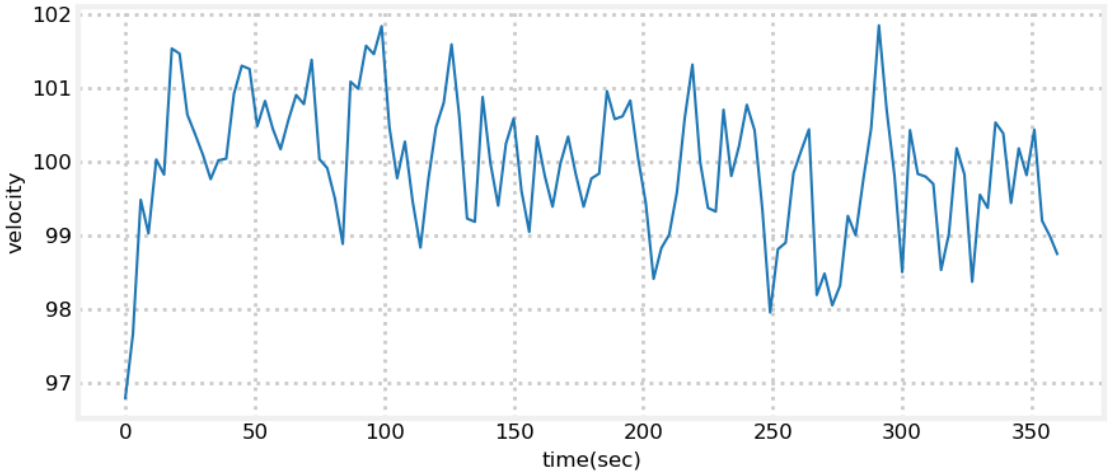
$$z = [\texttt{slant\_range}, \text{elevation angle}, \dot{x}, \dot{y}]$$

The measurement contains four values so the measurement function also needs to return four values. The slant range and elevation angle will be computed as before, and we do not need to compute the velocity in $x$ and $y$ as they are provided by the state estimate.

```
In [28]: def h_vel(x):
             dx = x[0] - h_vel.radar_pos[0]
             dz = x[2] - h_vel.radar_pos[1]
             slant_range = math.sqrt(dx**2 + dz**2)
             elevation_angle = math.atan2(dz, dx)
             return slant_range, elevation_angle, x[1], x[3]
```

Now we can implement our filter.

```
In [29]: h_radar.radar_pos = (0, 0)
         h_vel.radar_pos = (0, 0)

         range_std = 500.
         elevation_angle_std = math.degrees(0.5)
         vel_std = 2.

         np.random.seed(200)
         ac = ACSim(ac_pos, (100, 0), 0.02)
         radar = RadarStation((0, 0), range_std, elevation_angle_std)

         kf_sf2 = cv_UKF(f_cv_radar, h_vel,
                         R_std=[range_std, elevation_angle_std, vel_std, vel_std])

         time = np.arange(0, 360 + dt, dt)
         xs = []
         for t in time:
             ac.update(dt)
             r = radar.noisy_reading(ac.pos)
             # simulate the doppler velocity reading
             vx = ac.vel[0] + randn()*vel_std
             vz = ac.vel[1] + randn()*vel_std
             kf_sf2.predict()
             kf_sf2.update([r[0], r[1], vx, vz])
             xs.append(kf_sf2.x)
         xs = np.asarray(xs)
         plot_radar(xs, time, plot_x=False, plot_vel=True, plot_alt=False)
         print('Velocity std {:.1f} m/s'.format(np.std(xs[10:,1])))
```



```
Velocity std 0.9 m/s
```

By incorporating the velocity sensor we were able to reduce the standard deviation from 3.5 m/s to 1.3 m/s.

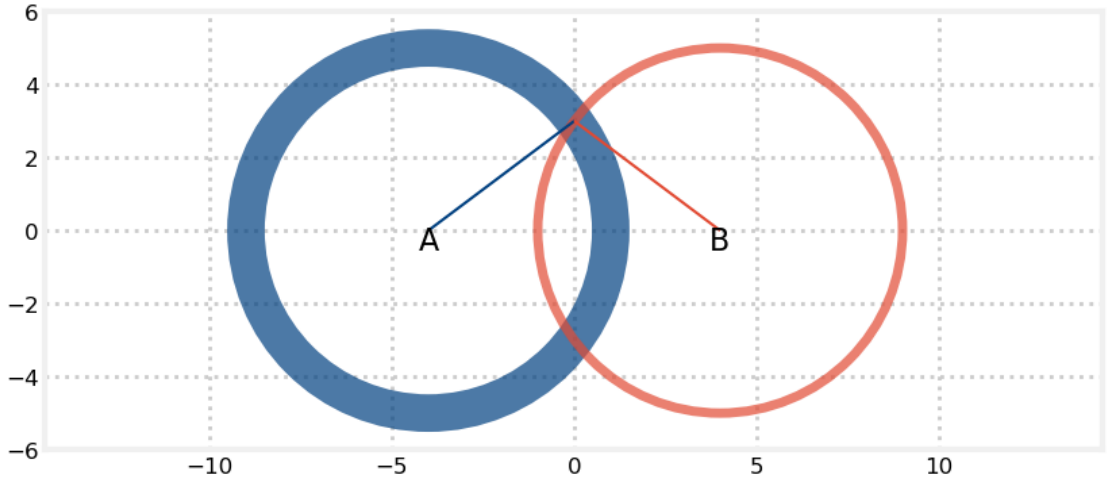
Sensor fusion is a large topic, and this is a rather simplistic implementation. In a typical navigation problem we have sensors that provide *complementary* information. For example, a GPS might provide somewhat accurate position updates once a second with poor velocity estimation while an inertial system might provide very accurate velocity updates at 50Hz but terrible position estimates. The strengths and weaknesses of each sensor are orthogonal to each other. This leads to the *Complementary filter*, which blends the high update rate inertial velocity measurements with the accurate but slowly updated position estimates of the GPS to produce high rate and accurate position and velocity estimates. The high rate velocity estimates are integrated between the GPS updates to produce accurate and high rate position estimates.

### 10.8.3 Multiple Position Sensors

The last sensor fusion problem was a toy example. Let's tackle a problem that is not so toy-like. Before GPS ships and aircraft navigated via various range and bearing systems such as VOR, LORAN, TACAN, DME, and so on. These systems emit beacons in the form of radio waves. The sensor extracts the range and/or bearing to the beacon from the signal. For example, an aircraft might have two VOR receivers. The pilot tunes each receiver to a different VOR station. Each VOR receiver displays the *radial* - the direction from the VOR station on the ground to the aircraft. The pilot uses a chart to find the intersection point of the radials, which identifies the location of the aircraft.

That is a manual approach with low accuracy. A Kalman filter will produce far more accurate position estimates. Assume we have two sensors, each of which provides a bearing only measurement to the target, as in the chart below. The width of the perimeters are proportional to the $3\sigma$ of the sensor noise. The aircraft must be positioned somewhere within the intersection of the two perimeters with a high degree of probability.

```
In [30]: ukf_internal.show_two_sensor_bearing()
```




We compute the bearing between a sensor and the target as:

```
In [31]: def bearing(sensor, target_pos):
             return math.atan2(target_pos[1] - sensor[1],
                               target_pos[0] - sensor[0])
```

The filter receives the measurement from the two sensors in a vector. The code will accept any iterable container, so I use a Python list for efficiency. We can implement that as:

```
In [32]: def measurement(A_pos, B_pos, pos):
             angle_a = bearing(A_pos, pos)
             angle_b = bearing(B_pos, pos)
             return [angle_a, angle_b]
```

Assume a constant velocity model for the aircraft. For a change of pace I compute the new positions explicitly rather than using matrix-vector multiplication:

```
In [33]: def fx_VOR(x, dt):
             x[0] += x[1] * dt
             x[2] += x[3] * dt
             return x
```

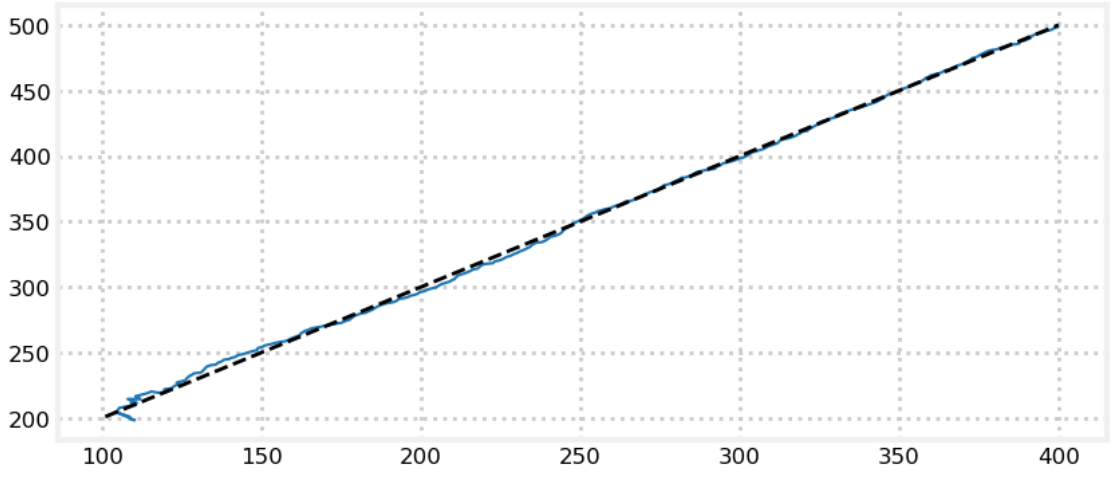
Next we implement the measurement function. It converts the prior to an array containing the measurement to both stations. I'm not a fan of global variables, but I put the position of the stations in the global variables `sa_pos` and `sb_pos` to demonstrate this method of sharing data with $h()$:

```
In [34]: sa_pos = [-400, 0]
         sb_pos = [400, 0]

         def hx_VOR(x):
             # measurement to A
             pos = (x[0], x[2])
             return measurement(sa_pos, sb_pos, pos)
```

Now we write boilerplate which constructs the filter, runs it, and plots the results:

```
In [35]: def moving_target_filter(pos, std_noise, Q, dt=0.1, kappa=0.0):
             points = MerweScaledSigmaPoints(n=4, alpha=.1, beta=2., kappa=kappa)
             f = UKF(dim_x=4, dim_z=2, dt=dt,
                     hx=hx_VOR, fx=fx_VOR, points=points)
             f.x = np.array([pos[0], 1., pos[1], 1.])

             q = Q_discrete_white_noise(2, dt, Q)
             f.Q[0:2, 0:2] = q
             f.Q[2:4, 2:4] = q
             f.R *= std_noise**2
             f.P *= 1000
             return f

         def plot_straight_line_target(f, std_noise):
             xs, txs = [], []
             for i in range(300):
                 target_pos[0] += 1 + randn()*0.0001
                 target_pos[1] += 1 + randn()*0.0001
                 txs.append((target_pos[0], target_pos[1]))

                 z = measurement(sa_pos, sb_pos, target_pos)
                 z[0] += randn() * std_noise
                 z[1] += randn() * std_noise

                 f.predict()
                 f.update(z)
```

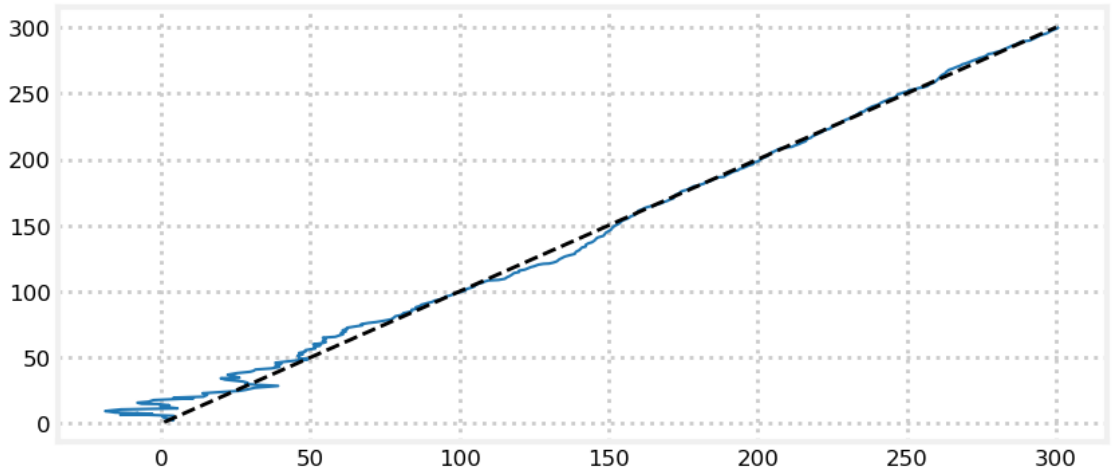
```
            xs.append(f.x)

        xs = np.asarray(xs)
        txs = np.asarray(txs)

        plt.plot(xs[:, 0], xs[:, 2])
        plt.plot(txs[:, 0], txs[:, 1], ls='--', lw=2, c='k')
        plt.show()

    np.random.seed(123)
    target_pos = [100, 200]

    std_noise = math.radians(0.5)
    f = moving_target_filter(target_pos, std_noise, Q=1.0)
    plot_straight_line_target(f, std_noise)
```




This looks quite good to me. The beginning of the track exhibits large errors, but the filter settles down and produces good estimates.

Let's revisit the nonlinearity of the angles. I will position the target between the two sensors at (0,0). This will cause a nonlinearity in the computation of the residuals because the mean angle will be near zero. As the angle goes below 0 the measurement function will compute a large positive angle of nearly $2\pi$. The residual between the prediction and measurement will thus be very large, nearly $2\pi$ instead of nearly 0. This makes it impossible for the filter to perform accurately, as seen in the example below.

```
In [36]: target_pos = [0, 0]
         f = moving_target_filter(target_pos, std_noise, Q=1.0)
         plot_straight_line_target(f, std_noise)
```

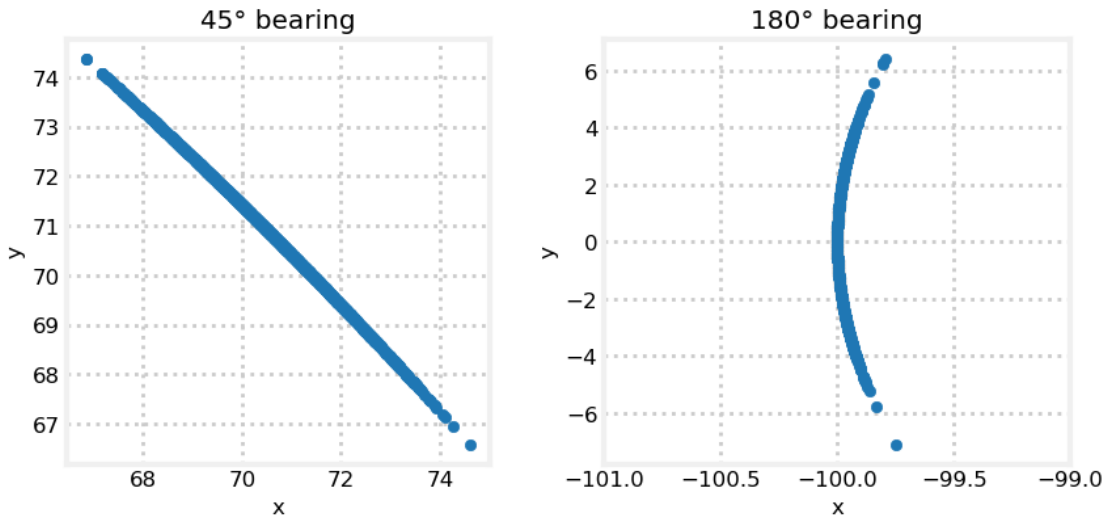



This performance is unacceptable. `FilterPy`'s UKF code allows you to specify a function which computes the residuals in cases of nonlinear behavior like this,. The final example in this chapter demonstrates its use.

## 10.9 Effects of Sensor Error and Geometry

The geometry of the sensors relative to the tracked object imposes a physical limitation that can be extremely difficult to deal with when designing filters. If the radials of the VOR stations are nearly parallel to each other then a very small angular error translates into a very large distance error. What is worse, this behavior is nonlinear - the error in the *x-axis* vs the *y-axis* will vary depending on the actual bearing. These scatter plots show the error distribution for a $1°\sigma$ error for two different bearings.

```
In [37]: ukf_internal.plot_scatter_of_bearing_error()
```



## 10.10 Exercise: Explain Filter Performance

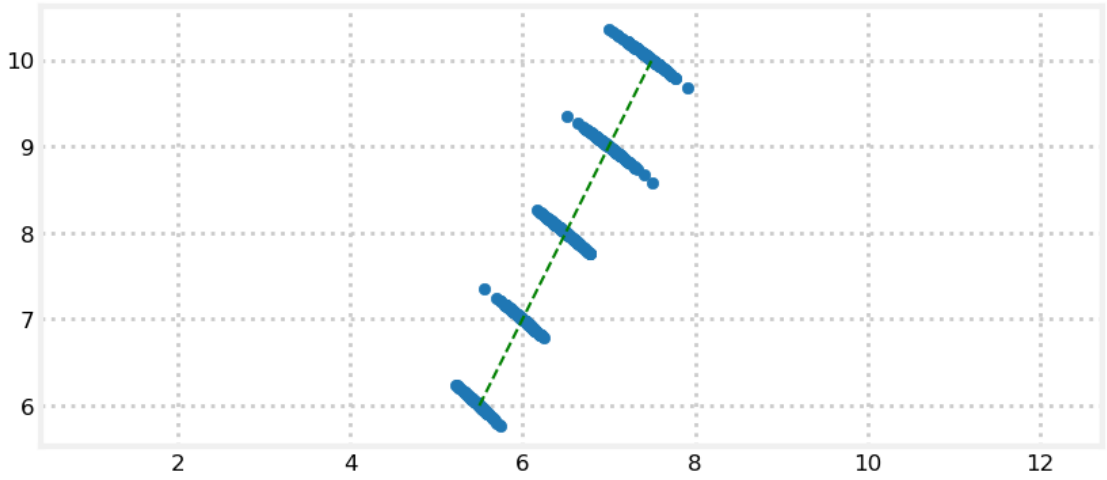
We can see that for small angular errors the positional errors are very large. Explain how we got such relatively good performance out of the UKF in the target tracking problems above. Answer for both the one sensor and multiple sensor problem.

### 10.10.1 Solution

This is very important to understand. Try very hard to answer this before reading the answer below. If you cannot answer this you may need to revisit some of the earlier material in the **Multidimensional Kalman Filter** chapter.

There are several factors contributing to our success. First, let's consider the case of having only one sensor. Any single measurement has an extreme range of possible positions. But, our target is moving, and the UKF is taking that into account. Let's plot the results of several measurements taken in a row for a moving target.

```
In [38]: ukf_internal.plot_scatter_moving_target()
```



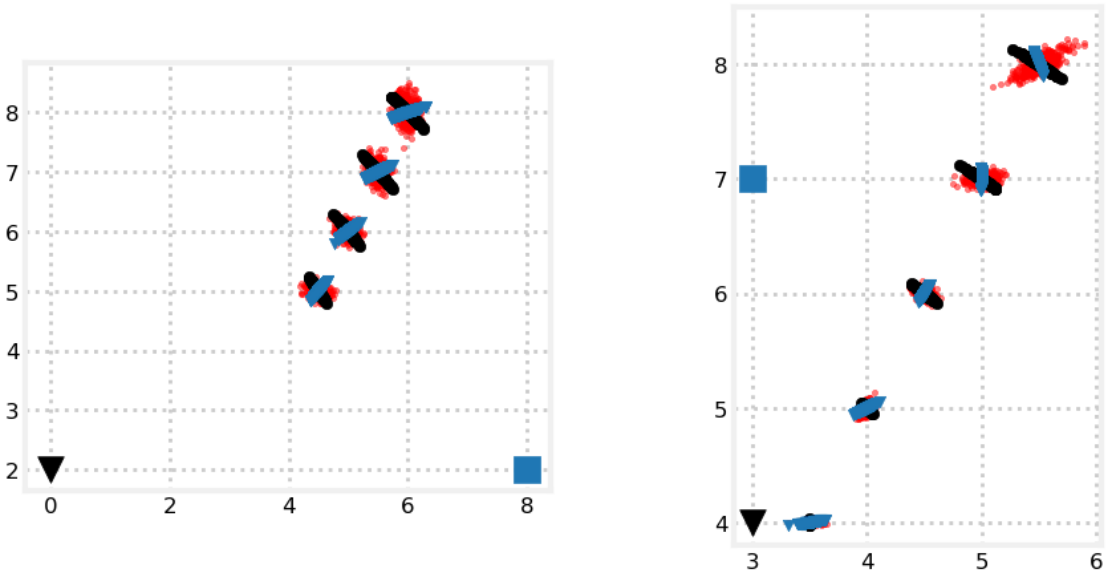

Each individual measurement has a very large position error. However, a plot of successive measurements shows a clear trend - the target is obviously moving towards the upper right. When a Kalman filter computes the Kalman gain it takes the distribution of errors into account by using the measurement function. In this example the error lies on an approximately 45° line, so the filter will discount errors in that direction. On the other hand, there is almost no error in measurement orthogonal to that, and again the Kalman gain will take that into account.

This graph makes it look easy because we have plotted 100 measurements for each position update. The movement of the aircraft is obvious. In contrast, the Kalman filter only gets one measurement per update. Therefore the filter will not be able to generate as good a fit as the dotted green line implies.

Now consider that the bearing gives us no distance information. Suppose we set the initial estimate to 1,000 kilometers away from the sensor (vs the actual distance of 7.07 km) and make $\mathbf{P}$ very small. At that distance a 1° error translates into a positional error of 17.5 km. The KF would never be able to converge onto the actual target position because the filter is incorrectly very certain about its position estimates and because there is no distance information provided in the measurements.

Now let's consider the effect of adding a second sensor. Here are two plots showing the effects of different sensor placements. I used a square and a triangle as a symbol for the two sensors, and drew a distribution of the errors due to each sensor using the same symbol shape and color. I then computed the $(x, y)$ coordinate corresponding to the two noisy bearing measurements and plotted them with red dots to show the distribution of the noisy measurements in $x$ and $y$.

```
In [39]: with figsize(10,5):
             ukf_internal.plot_iscts_two_sensors()
```




In the first plot I placed the sensors nearly orthogonal to the target's initial position so we get these lovely 'x' shape intersections. We can see how the errors in $x$ and $y$ change as the target moves by the shape the scattered red dots make - as the target gets further away from the sensors, but nearer the $y$ coordinate of sensor B the shape becomes strongly elliptical.

In the second plot the airplane starts very near one sensor, and then flies past the second sensor. The intersections of the errors are very non-orthogonal, and the resulting position errors become very spread out.

## 10.11 Implementation of the UKF

FilterPy implements the UKF, but it is instructive to learn how to translate equations into code. Implementing the UKF is quite straightforward. First, let's write code to compute the mean and covariance of the sigma points.

We will store the sigma points and weights in matrices, like so:

$$\text{weights} = \begin{bmatrix} w_0 & w_1 & \dots & w_{2n} \end{bmatrix}$$

$$\text{sigmas} = \begin{bmatrix} \mathcal{X}_{0,0} & \mathcal{X}_{0,1} & \dots & \mathcal{X}_{0,n-1} \\ \mathcal{X}_{1,0} & \mathcal{X}_{1,1} & \dots & \mathcal{X}_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{X}_{2n,0} & \mathcal{X}_{2n,1} & \dots & \mathcal{X}_{2n,n-1} \end{bmatrix}$$

That's a lot of subscripts to describe something very simple, so here's an example for a two dimensional problem ($n$=2):

```
In [40]: points = MerweScaledSigmaPoints(n=2, alpha=.1, beta=2., kappa=1.)
         points.sigma_points(x=[0.,0], P=[[1.,.1],[.1, 1]])
```

```
Out[40]: array([[ 0.   ,  0.   ],
                [ 0.173,  0.017],
                [ 0.   ,  0.172],
                [-0.173, -0.017],
                [ 0.   , -0.172]])
```

The sigma point for the mean is on the first row. Its position is (0, 0), which is equal to the mean (0,0). The second sigma point is at position (0.173, 0.017), and so on. There are are $2n + 1 = 5$ rows, one row per sigma point. If $n = 3$, then there would be 3 columns and 7 rows.

The choice to store the sigmas in row-column vs column row format is somewhat arbitrary; my choice makes the rest of the code clearer as I can refer to the i$^{th}$ sigma point as `sigmas[i]` instead of `sigmas[:, i]`.

### 10.11.1 Weights

Computing the weights with NumPy is easy. Recall that the Van der Merwe scaled sigma point implementation states:

$$\begin{aligned}
\lambda &= \alpha^2(n + \kappa) - n \\
W_0^m &= \frac{\lambda}{n + \lambda} \\
W_0^c &= \frac{\lambda}{n + \lambda} + 1 - \alpha^2 + \beta \\
W_i^m = W_i^c &= \frac{1}{2(n + \lambda)} \quad i = 1..2n
\end{aligned}$$

Code for these is:

```
lambda_ = alpha**2 * (n + kappa) - n
Wc = np.full(2*n + 1,  1. / (2*(n + lambda_))
Wm = np.full(2*n + 1,  1. / (2*(n + lambda_))
Wc[0] = lambda_ / (n + lambda_) + (1. - alpha**2 + beta)
Wm[0] = lambda_ / (n + lambda_)
```

I use the underscore in `lambda_` because `lambda` is a reserved word in Python. A trailing underscore is the Pythonic workaround.

### 10.11.2 Sigma Points

The equations for the sigma points are:

$$\begin{cases}
\mathcal{X}_0 = \mu \\
\mathcal{X}_i = \mu + \left[\sqrt{(n + \lambda)\Sigma}\right]_i, & \texttt{for i=1..n} \\
\mathcal{X}_i = \mu - \left[\sqrt{(n + \lambda)\Sigma}\right]_{i-n} & \texttt{for i=(n+1)..2n}
\end{cases}$$

The Python is not difficult once we understand the $\left[\sqrt{(n + \lambda)\Sigma}\right]_i$ term.

The term $\sqrt{(n + \lambda)\Sigma}$ is a matrix because $\Sigma$ is a matrix. The subscript $i$ in $[\sqrt{(n + \lambda)\Sigma}]_i$ is choosing the i-th row vector of the matrix. What is the square root of a matrix? There is no unique definition. One definition is that the square root of a matrix $\Sigma$ is the matrix $S$ that, when multiplied by itself, yields $\Sigma$: if $\Sigma = SS$ then $S = \sqrt{\Sigma}$.

We will choose an alternative definition that has numerical properties which make it easier to compute. We can define the square root as the matrix S, which when multiplied by its transpose, returns $\Sigma$:

$$\Sigma = \mathbf{S}\mathbf{S}^\mathsf{T}$$

This definition is favored because **S** is computed using the *Cholesky decomposition* [3]. It decomposes a Hermitian, positive-definite matrix into a triangular matrix and its conjugate transpose. The matrix can be either upper or lower triangular, like so:

$$A = LL^*A = U^*U$$

The asterick denotes the conjugate transpose; we have only real numbers so for us we can write:

$$A = LL^\mathsf{T}A = U^\mathsf{T}U$$

$\mathbf{P}$ has these properties, so we can treat $\mathbf{S} = \text{cholesky}(\mathbf{P})$ as the square root of $\mathbf{P}$.

SciPy provides `cholesky()` method in `scipy.linalg`. If your language of choice is Fortran, C, or C++, libraries such as LAPACK provide this routine. Matlab provides `chol()`.

By default `scipy.linalg.cholesky()` returns a upper triangular matrix, so I elected to write the code to expect an upper triangular matrix. For this reason I access the result by row, so that the first sigma point, which is the center point, is affected by a full row of non-zero values. If you provide your own square root implementation you will need to take this into account. You will find UKF algorithms in the literature that take the values column first. This is fine if the cholesky is lower-triangular, or if you are using a different algorithm that computes a symmetric matrix so row vs column order doesn't matter.

```
In [41]: import scipy
         a = np.array([[2., .1], [.1, 3]])
         s = scipy.linalg.cholesky(a)
         print("cholesky:")
         print(s)
         np.dot(s, s.T)
```

```
cholesky:
[[1.414 0.071]
 [0.    1.731]]
```

```
Out[41]: array([[2.005, 0.122],
                [0.122, 2.995]])
```

So we can implement the sigma points with this code:

```
sigmas = np.zeros((2*n+1, n))
U = scipy.linalg.cholesky((n+lambda_)*P) # sqrt

sigmas[0] = X
for k in range (n):
    sigmas[k+1]   = X + U[k]
    sigmas[n+k+1] = X - U[k]
```

Now let's implement the unscented transform. Recall the equations

$$\mu = \sum_i w_i^m \mathcal{X}_i$$

$$\Sigma = \sum_i w_i^c (\mathcal{X}_i - \mu)(\mathcal{X}_i - \mu)^\mathsf{T}$$

We implement the sum of the means with

```
x = np.dot(Wm, sigmas)
```

If you are not a heavy user of NumPy this may look foreign to you. NumPy is not just a library that make linear algebra possible; under the hood it is written in C and Fortran to achieve much faster speeds than Python can reach. A typical speedup is 20x to 100x. To get that speedup we must avoid using for loops, and instead use NumPy's built in functions to perform calculations. So, instead of writing a for loop to compute the sum of products, we call the built in `numpy.dot(x, y)` method. The dot product of two vectors is the sum of the element-wise multiplications of each element. If passed a 1D array and a 2D array it will compute the sum of inner products:

```
In [42]: a = np.array([10, 100])
         b = np.array([[1, 2, 3],
                       [4, 5, 6]])
         np.dot(a, b)
```

```
Out[42]: array([410, 520, 630])
```

All that is left is to compute $\mathbf{P} = \sum_i w_i(\mathcal{X}_i - \mu)(\mathcal{X}_i - \mu)^\mathsf{T} + \mathbf{Q}$:

```
kmax, n = sigmas.shape
P = zeros((n, n))
for k in range(kmax):
    y = sigmas[k] - x
    P += Wc[k] * np.outer(y, y)
P += Q
```

This introduces another feature of NumPy. The state variable `x` is one dimensional, as is `sigmas[k]`, so the difference `sigmas[k]-X` is also one dimensional. NumPy will not compute the transpose of a 1-D array; it considers the transpose of `[1,2,3]` to be `[1,2,3]`. So we call the function `np.outer(y,y)` which computes the value of $\mathbf{yy}^\mathsf{T}$ for the 1D array $\mathbf{y}$. An alternative implementation could be:

```
y = (sigmas[k] - x).reshape(kmax, 1) # convert into 2D array
P += Wc[K] * np.dot(y, y.T)
```

This code is slower and not idiomatic, so we will not use it.

### 10.11.3 Predict Step

For the predict step, we will generate the weights and sigma points as specified above. We pass each sigma point through the function f.

$$\boldsymbol{\mathcal{Y}} = f(\boldsymbol{\chi})$$

Then we compute the predicted mean and covariance using the unscented transform. In the code below you can see that I am assuming that this is a method in a class that stores the various matrices and vectors needed by the filter.

```
def predict(self, sigma_points_fn):
    """ Performs the predict step of the UKF. On return,
    self.xp and self.Pp contain the predicted state (xp)
    and covariance (Pp). 'p' stands for prediction.
    """

    # calculate sigma points for given mean and covariance
    sigmas = sigma_points_fn(self.x, self.P)

    for i in range(self._num_sigmas):
        self.sigmas_f[i] = self.fx(sigmas[i], self._dt)

    self.xp, self.Pp = unscented_transform(
                       self.sigmas_f, self.Wm, self.Wc, self.Q)
```

### 10.11.4 Update Step

The update step converts the sigmas into measurement space via the function `h(x)`.

$$\mathcal{Z} = h(\mathcal{Y})$$

The mean and covariance of those points is computed with the unscented transform. The residual and Kalman gain is then computed. The cross variance is computed as:

$$\mathbf{P}_{xz} = \sum_{i=0}^{2n} w_i^c(\boldsymbol{\mathcal{Y}}_i - \mu)(\boldsymbol{\mathcal{Z}}_i - \mu_z)^\mathsf{T}$$

Finally, we compute the new state estimate using the residual and Kalman gain:

$$K = \mathbf{P}_{xz}\mathbf{P}_z^{-1}$$
$$\mathbf{x} = \mathbf{x} + \mathbf{K}\mathbf{y}$$

and the new covariance is computed as:

$$\mathbf{P} = \mathbf{P} - \mathbf{K}\mathbf{P}_z\mathbf{K}^\mathsf{T}$$

This function can be implemented as follows, assuming it is a method of a class that stores the necessary matrices and data.

```
def update(self, z):
    # rename for readability
    sigmas_f = self.sigmas_f
    sigmas_h = self.sigmas_h

    # transform sigma points into measurement space
    for i in range(self._num_sigmas):
        sigmas_h[i] = self.hx(sigmas_f[i])

    # mean and covariance of prediction passed through UT
    zp, Pz = unscented_transform(sigmas_h, self.Wm, self.Wc, self.R)

    # compute cross variance of the state and the measurements
    Pxz = np.zeros((self._dim_x, self._dim_z))
    for i in range(self._num_sigmas):
        Pxz += self.Wc[i] * np.outer(sigmas_f[i] - self.xp,
                                     sigmas_h[i] - zp)

    K = np.dot(Pxz, inv(Pz)) # Kalman gain

    self.x = self.xp + np.dot(K, z - zp)
    self.P = self.Pp - np.dot(K, Pz).dot(K.T)
```

### 10.11.5 FilterPy's Implementation

FilterPy has generalized the code somewhat. You can specify different sigma point algorithms, specify how to compute the residual of the state variables (you can not subtract angles because they are modular), provide a matrix square root function, and more. See the help for details.

https://filterpy.readthedocs.org/#unscented-kalman-filter

## 10.12 Batch Processing

The Kalman filter is recursive - estimates are based on the currernt measurement and prior estimate. But it is very common to have a set of data that have been already collected which we want to filter. In this case the filter can be run in a *batch* mode, where all of the measurements are filtered at once.

Collect your measurements into an array or list.

```
zs = read_altitude_from_csv()
```

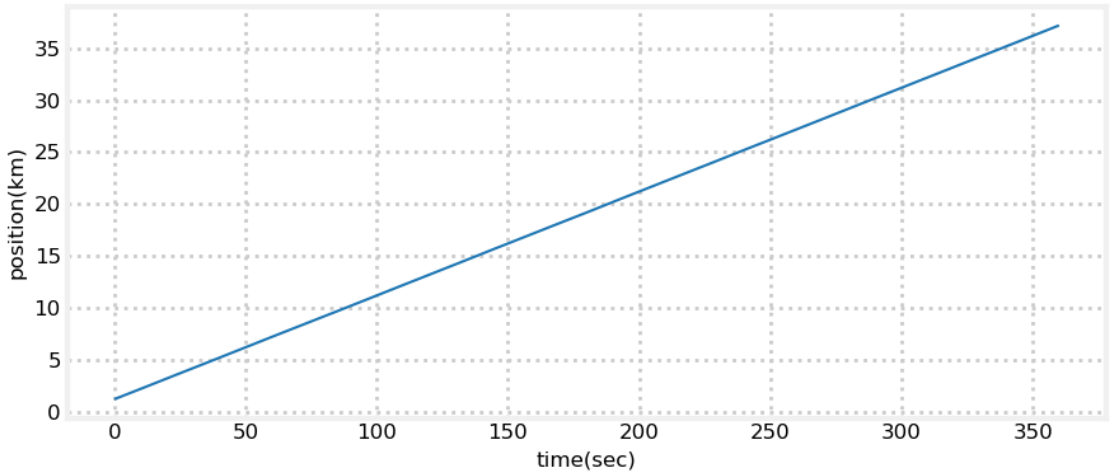
Then call the `batch_filter()` method.

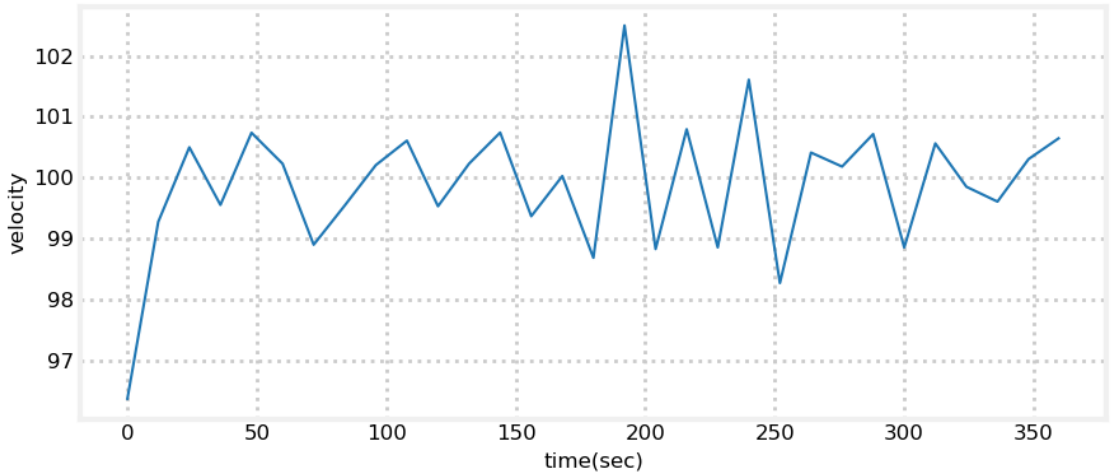
```
Xs, Ps = ukf.batch_filter(zs)
```

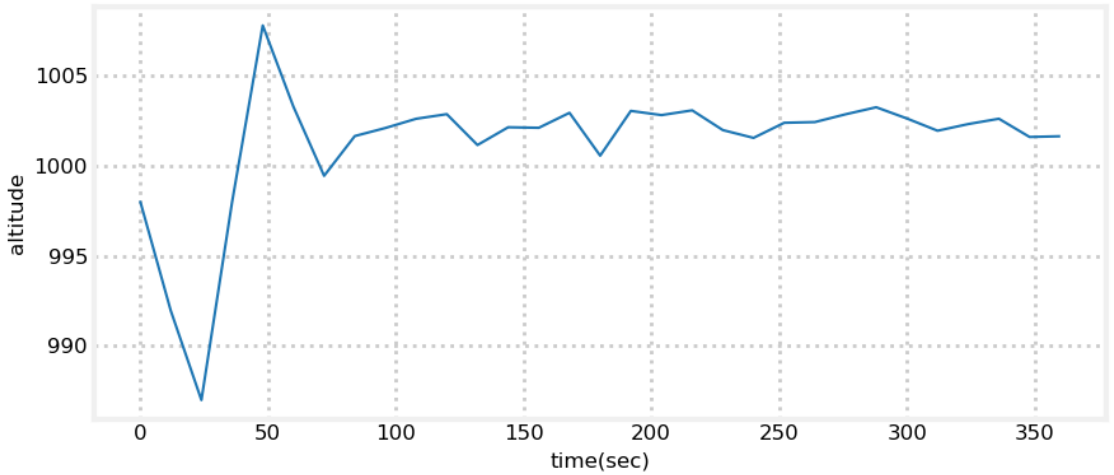
The function takes the list/array of measurements, filters it, and returns an array of state estimates (`Xs`) and covariance matrices (`Ps`) for the entire data set.

Here is a complete example drawing from the radar tracking problem above.

```
In [43]: dt = 12. # 12 seconds between readings
         range_std = 5 # meters
         bearing_std = math.radians(0.5)

         ac_pos = (0., 1000.)
         ac_vel = (100., 0.)
         radar_pos = (0., 0.)
         h_radar.radar_pos = radar_pos

         points = MerweScaledSigmaPoints(n=3, alpha=.1, beta=2., kappa=0.)
         kf = UKF(3, 2, dt, fx=f_radar, hx=h_radar, points=points)

         kf.Q[0:2 ,0:2] = Q_discrete_white_noise(2, dt=dt, var=0.1)
         kf.Q[2, 2] = 0.1

         kf.R = np.diag([range_std**2, bearing_std**2])
         kf.x = np.array([0., 90., 1100.])
         kf.P = np.diag([300**2, 30**2, 150**2])

         radar = RadarStation((0, 0), range_std, bearing_std)
         ac = ACSim(ac_pos, (100, 0), 0.02)

         np.random.seed(200)

         t = np.arange(0, 360 + dt, dt)
         n = len(t)

         zs = []
         for i in range(len(t)):
             ac.update(dt)
             r = radar.noisy_reading(ac.pos)
             zs.append([r[0], r[1]])

         xs, covs = kf.batch_filter(zs)
         ukf_internal.plot_radar(xs, t)
```


position(km)
0
5
10
15
20
25
30
35
0
50
100
150
200
250
300
350
time(sec)


velocity
97
98
99
100
101
102
0
50
100
150
200
250
300
350
time(sec)


altitude
990
995
1000
1005
0
50
100
150
200
250
300
350
time(sec)

## 10.13 Smoothing the Results

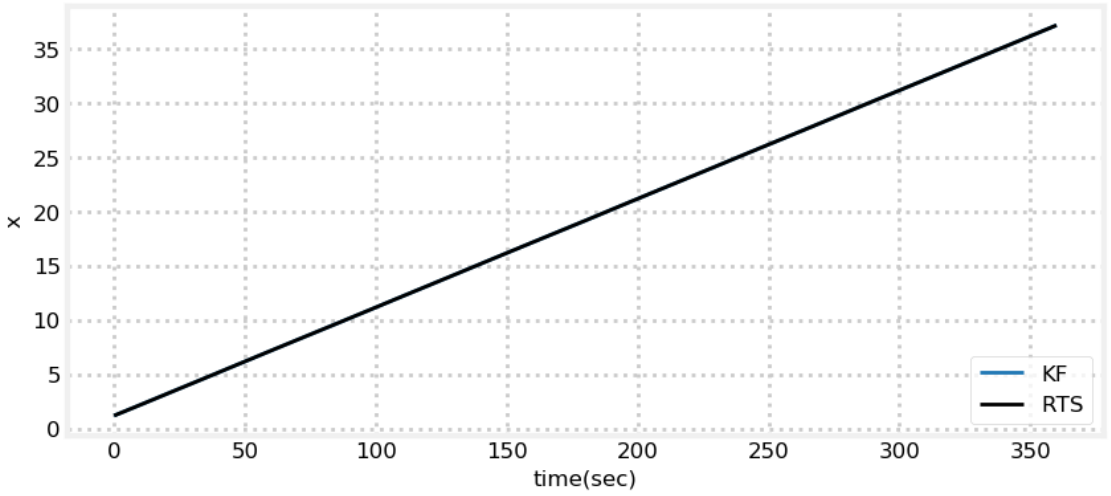
Assume that we are tracking a car. Suppose we get a noisy measurement that implies that the car is starting to turn to the left, but the state function has predicted that the car is moving straight. The Kalman filter has no choice but to move the state estimate somewhat towards the noisy measurement, as it cannot judge whether this is just a particularly noisy measurement or the true start of a turn.

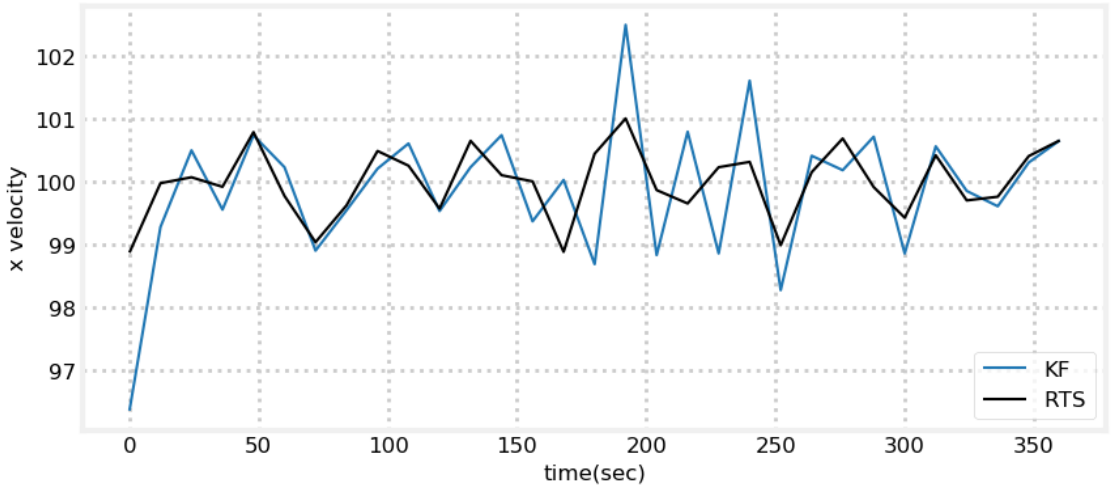
If we are collecting data and post-processing it we have measurements after the questionable one that informs us if a turn was made or not. Suppose the subsequent measurements all continue turning left. We can then be sure that the measurement was not very noisy, but instead a turn was initiated.

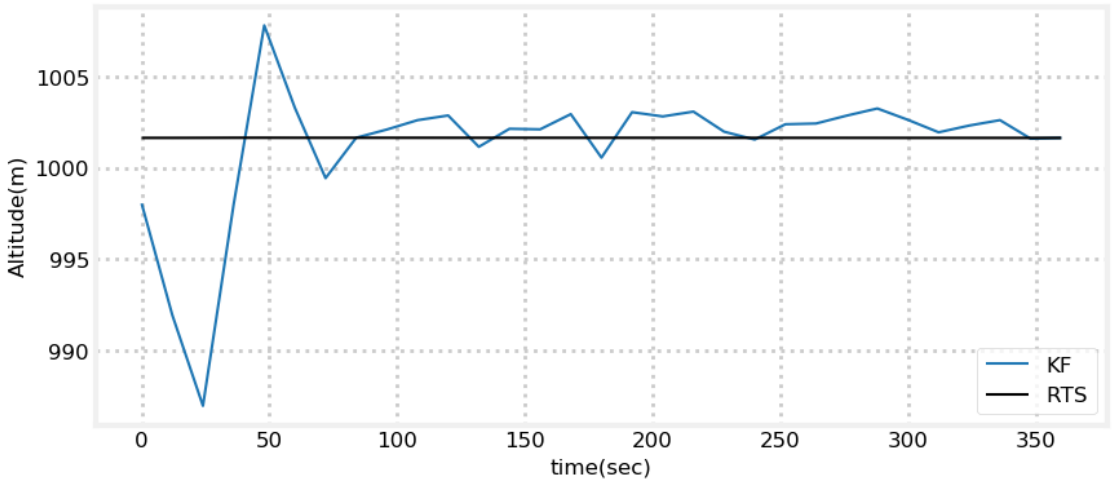
We will not develop the math or algorithm here, I will just show you how to call the algorithm in `FilterPy`. The algorithm that we have implemented is called an *RTS smoother*, after the three inventors of the algorithm: Rauch, Tung, and Striebel.

The routine is `UnscentedKalmanFilter.rts_smoother()`. Using it is trivial; we pass in the means and covariances computed from the `batch_filter` step, and receive back the smoothed means, covariances, and Kalman gain.

```
In [44]: Ms, P, K = kf.rts_smoother(xs, covs)
         ukf_internal.plot_rts_output(xs, Ms, t)

Difference in position in meters:
         [-1.0928  0.1593  0.3259 -0.2323 -0.1727]
```






From these charts we can see that the improvement in the position is small, but the improvement in the velocity is good, and spectacular for the altitude. The difference in the position are very small, so I printed the difference between the UKF and the smoothed results for the last 5 points. I recommend always using the RTS smoother if you can post-process your data.

## 10.14 Choosing the Sigma Parameters

I have found the literature on choosing values for $\alpha$, $\beta$, and $\kappa$ to be rather lacking. Van der Merwe's dissertation contains the most information, but it is not exhaustive. So let's explore what they do.

Van der Merwe suggests using $\beta = 2$ for Gaussian problems, and $\kappa = 3 - n$. So let's start there and vary $\alpha$. I will let $n = 1$ to minimize the size of the arrays we need to look at and to avoid having to compute the square root of matrices.

```
In [45]: from kf_book.ukf_internal import print_sigmas
         print_sigmas(mean=0, cov=3, alpha=1)
```

```
sigmas:  [ 0.  3. -3.]
mean weights: [0.6667 0.1667 0.1667]
cov weights: [2.6667 0.1667 0.1667]
lambda: 2
sum cov 2.9999999999999996
```

So what is going on here? We can see that for a mean of 0 the algorithm chooses sigma points of 0, 3, and -3, but why? Recall the equation for computing the sigma points:

$$\begin{aligned}\mathcal{X}_0 &= \mu \\ \mathcal{X}_i &= \mu \pm \sqrt{(n+\lambda)\Sigma}\end{aligned}$$

My choice of $n = 1$ reduces everything to scalars, allowing us to avoid computing the square root of matrices. So, for our values the equation is

$$\begin{aligned}\mathcal{X}_0 &= 0 \\ \mathcal{X}_i &= 0 \pm \sqrt{(1+2) \times 3} \\ &= \pm 3\end{aligned}$$

So as $\alpha$ gets larger the sigma points get more spread out. Let's set it to an absurd value.

```
In [46]: print_sigmas(mean=0, cov=3, alpha=200)
```

```
sigmas:  [   0.  600. -600.]
mean weights: [1. 0. 0.]
cov weights: [-39996.      0.      0.]
lambda: 119999
sum cov -39996.00000000001
```

We can see that the sigma points spread over 100 standard deviations. If our data was Gaussian we'd be incorporating data many standard deviations away from the mean; for nonlinear problems this is unlikely to produce good results. But suppose our distribution was not Gaussian, but instead had very fat tails? We might need to sample from those tails to get a good estimate, and hence it would make sense to make $\kappa$ larger (not 200, which was absurdly large to make the change in the sigma points stark).

With a similar line of reasoning, suppose that our distribution has nearly no tails - the probability distribution looks more like an inverted parabola. In such a case we'd probably want to pull the sigma points in closer to the mean to avoid sampling in regions where there will never be real data.

Now let's look at the change in the weights. When we have $k + n = 3$ the weights were 0.6667 for the mean, and 0.1667 for the two outlying sigma points. On the other hand, when $\alpha = 200$ the mean weight shot up to 0.99999 and the outlier weights were set to 0.000004. Recall the equations for the weights:

$$\begin{aligned}W_0 &= \frac{\lambda}{n+\lambda} \\ W_i &= \frac{1}{2(n+\lambda)}\end{aligned}$$

We can see that as $\lambda$ gets larger the fraction for the weight of the mean $(\lambda/(n+\lambda))$ approaches 1, and the fraction for the weights of the rest of the sigma points approaches 0. This is invariant on the size of your covariance. So as we sample further and further away from the mean we end up giving less weight to those samples, and if we sampled very close to the mean we'd give very similar weights to all.

However, the advice that Van der Merwe gives is to constrain $\alpha$ in the range $0 > \alpha \geq 1$. He suggests $10^{-3}$ as a good value. Let's try that.

```
In [47]: print_sigmas(mean=0, cov=13, alpha=.001, kappa=0)
```

```
sigmas:  [ 0.      0.0036 -0.0036]
mean weights: [-999999.  500000.  500000.]
cov weights: [-999996.  500000.  500000.]
lambda: -0.999999
sum cov 3.9999989999923855
```

## 10.15 Robot Localization - A Fully Worked Example

It is time to undertake a significant problem. Most books choose simple, textbook problems with simple answers, and you are left wondering how to implement a real world problem. This example will not teach you how to tackle any problem, but illustrates the type of things you will have to consider as you design and implement a filter.

We will consider the problem of robot localization. In this scenario we have a robot that is moving through a landscape using a sensor to detect landmarks. This could be a self driving car using computer vision to identify trees, buildings, and other landmarks. It might be one of those small robots that vacuum your house, or a robot in a warehouse.

The robot has 4 wheels in the same configuration used by automobiles. It maneuvers by pivoting the front wheels. This causes the robot to pivot around the rear axle while moving forward. This is nonlinear behavior which we will have to model.

The robot has a sensor that gives it approximate range and bearing to known targets in the landscape. This is nonlinear because computing a position from a range and bearing requires square roots and trigonometry.

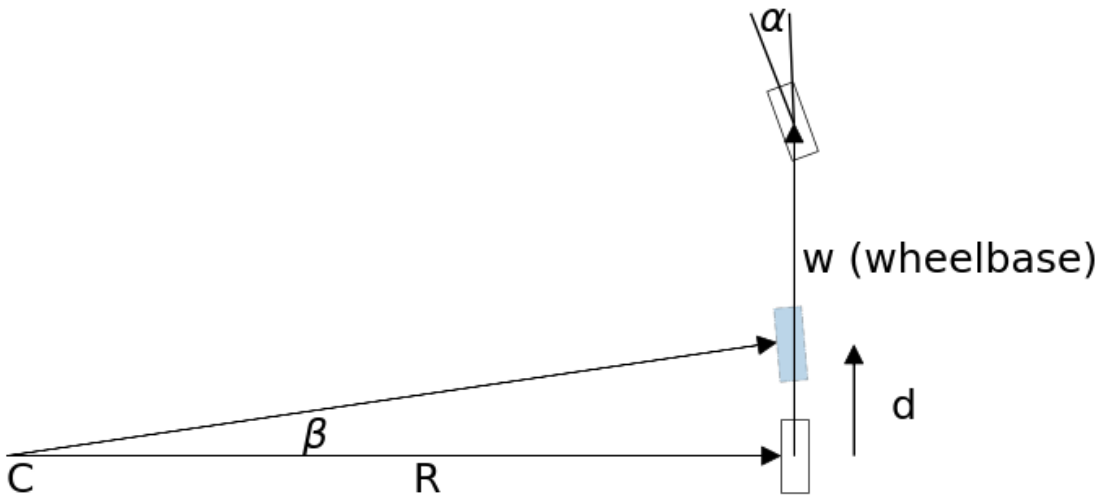
Both the process model and measurement models are nonlinear. The UKF accommodates both, so we provisionally conclude that the UKF is a viable choice for this problem.

### 10.15.1 Robot Motion Model

At a first approximation an automobile steers by pivoting the front tires while moving forward. The front of the car moves in the direction that the wheels are pointing while pivoting around the rear tires. This simple description is complicated by issues such as slippage due to friction, the differing behavior of the rubber tires at different speeds, and the need for the outside tire to travel a different radius than the inner tire. Accurately modeling steering requires a complicated set of differential equations.

For Kalman filtering, especially for lower speed robotic applications a simpler *bicycle model* has been found to perform well. This is a depiction of the model:

```
In [48]: ekf_internal.plot_bicycle()
```




Here we see the front tire is pointing in direction $\alpha$ relative to the wheelbase. Over a short time period the car moves forward and the rear wheel ends up further ahead and slightly turned inward, as depicted with the blue shaded tire. Over such a short time frame we can approximate this as a turn around a radius $R$. We can compute the turn angle $\beta$ with

$$\beta = \frac{d}{w}\tan(\alpha)$$

and the turning radius R is given by

$$R = \frac{d}{\beta}$$

where the distance the rear wheel travels given a forward velocity $v$ is $d = v\Delta t$.
With $\theta$ being the robot's orientation we compute the position $C$ before the turn starts as

$$\begin{aligned} C_x &= x - R\sin(\theta) \\ C_y &= y + R\cos(\theta) \end{aligned}$$

After the move forward for time $\Delta t$ the new position and orientation of the robot is

$$\begin{aligned} \bar{x} &= C_x + R\sin(\theta + \beta) \\ \bar{y} &= C_y - R\cos(\theta + \beta) \\ \bar{\theta} &= \theta + \beta \end{aligned}$$

Once we substitute in for $C$ we get

$$\begin{aligned} \bar{x} &= x - R\sin(\theta) + R\sin(\theta + \beta) \\ \bar{y} &= y + R\cos(\theta) - R\cos(\theta + \beta) \\ \bar{\theta} &= \theta + \beta \end{aligned}$$

You do not need to understand this math in detail if you are not interested in steering models. The important thing to recognize is that our motion model is nonlinear, and we will need to deal with that with our Kalman filter.

### 10.15.2 Design the State Variables

For our robot we will maintain the position and orientation:

$$\mathbf{x} = \begin{bmatrix} x & y & \theta \end{bmatrix}^\mathsf{T}$$

I could include velocities into this model, but as you will see the math will already be quite challenging. The control input $\mathbf{u}$ is the commanded velocity and steering angle

$$\mathbf{u} = \begin{bmatrix} v & \alpha \end{bmatrix}^\mathsf{T}$$

### 10.15.3 Design the System Model

We model our system as a nonlinear motion model plus white noise.

$$\bar{x} = x + f(x, u) + \mathcal{N}(0, Q)$$

Using the motion model for a robot that we created above, we can write:

```
In [49]: from math import tan, sin, cos, sqrt

         def move(x, dt, u, wheelbase):
             hdg = x[2]
             vel = u[0]
             steering_angle = u[1]
             dist = vel * dt

             if abs(steering_angle) > 0.001: # is robot turning?
                 beta = (dist / wheelbase) * tan(steering_angle)
                 r = wheelbase / tan(steering_angle) # radius

                 sinh, sinhb = sin(hdg), sin(hdg + beta)
                 cosh, coshb = cos(hdg), cos(hdg + beta)
                 return x + np.array([-r*sinh + r*sinhb,
                                      r*cosh - r*coshb, beta])
             else: # moving in straight line
                 return x + np.array([dist*cos(hdg), dist*sin(hdg), 0])
```

We will use this function to implement the state transition function `f(x)`.

I will design the UKF so that $\Delta t$ is small. If the robot is moving slowly then this function should give a reasonably accurate prediction. If $\Delta t$ is large or your system's dynamics are very nonlinear this method will fail. In those cases you will need to implement it using a more sophisticated numerical integration technique such as Runge Kutta. Numerical integration is covered briefly in the **Kalman Filter Math** chapter.

### 10.15.4 Design the Measurement Model

The sensor provides a noisy bearing and range to multiple known locations in the landscape. The measurement model must convert the state $\begin{bmatrix} x & y & \theta \end{bmatrix}^\mathsf{T}$ into a range and bearing to the landmark. If $p$ is the position of a landmark, the range $r$ is

$$r = \sqrt{(p_x - x)^2 + (p_y - y)^2}$$

We assume that the sensor provides bearing relative to the orientation of the robot, so we must subtract the robot's orientation from the bearing to get the sensor reading, like so:

$$\phi = \tan^{-1}(\frac{p_y - y}{p_x - x}) - \theta$$

Thus our measurement function is

$$\begin{aligned}\mathbf{z} &= h(\mathbf{x}, \mathbf{P}) &&+\mathcal{N}(0, R)\\ &= \begin{bmatrix}\sqrt{(p_x - x)^2 + (p_y - y)^2} \\ \tan^{-1}(\frac{p_y - y}{p_x - x}) - \theta\end{bmatrix} &&+\mathcal{N}(0, R)\end{aligned}$$

I will not implement this yet as there is a difficulty that will be discussed in the *Implementation* section below.

### 10.15.5 Design Measurement Noise

It is reasonable to assume that the range and bearing measurement noise is independent, hence

$$\mathbf{R} = \begin{bmatrix}\sigma^2_{range} & 0 \\ 0 & \sigma^2_{bearing}\end{bmatrix}$$

### 10.15.6 Implementation

Before we begin coding we have another issue to handle. The residual is $y = z - h(x)$. Suppose z has a bearing of 1° and $h(x)$ is 359°. Subtracting them gives −358°. This will throw off the computation of the Kalman gain because the correct angular difference is 2°. So we will have to write code to correctly compute the bearing residual.

```
In [50]: def normalize_angle(x):
             x = x % (2 * np.pi)    # force in range [0, 2 pi)
             if x > np.pi:          # move to [-pi, pi)
                 x -= 2 * np.pi
             return x
```

```
In [51]: print(np.degrees(normalize_angle(np.radians(1-359))))
```

```
1.9999999999999774
```

The state vector has the bearing at index 2, but the measurement vector has it at index 1, so we need to write functions to handle each. Another issue we face is that as the robot maneuvers different landmarks will be visible, so we need to handle a variable number of measurements. The function for the residual in the measurement will be passed an array of several measurements, one per landmark.

```
In [52]: def residual_h(a, b):
             y = a - b
             # data in format [dist_1, bearing_1, dist_2, bearing_2,...]
             for i in range(0, len(y), 2):
                 y[i + 1] = normalize_angle(y[i + 1])
             return y

         def residual_x(a, b):
             y = a - b
             y[2] = normalize_angle(y[2])
             return y
```

We can now implement the measurement model. The equation is

$$h(\mathbf{x}, \mathbf{P}) = \begin{bmatrix}\sqrt{(p_x - x)^2 + (p_y - y)^2} \\ \tan^{-1}(\frac{p_y - y}{p_x - x}) - \theta\end{bmatrix}$$

The expression $\tan^{-1}(\frac{p_y-y}{p_x-x}) - \theta$ can produce a result outside the range $[-\pi, \pi)$, so we should normalize the angle to that range.

The function will be passed an array of landmarks and needs to produce an array of measurements in the form `[dist_to_1, bearing_to_1, dist_to_2, bearing_to_2, ...]`.

```
In [53]: def Hx(x, landmarks):
             """ takes a state variable and returns the measurement
             that would correspond to that state. """
             hx = []
             for lmark in landmarks:
                 px, py = lmark
                 dist = sqrt((px - x[0])**2 + (py - x[1])**2)
                 angle = atan2(py - x[1], px - x[0])
                 hx.extend([dist, normalize_angle(angle - x[2])])
             return np.array(hx)
```

Our difficulties are not over. The unscented transform computes the average of the state and measurement vectors, but each contains a bearing. There is no unique way to compute the average of a set of angles. For example, what is the average of 359° and 3°? Intuition suggests the answer should be 1°, but a naive $\frac{1}{n}\sum x$ approach yields 181°.

One common approach is to take the arctan of the sum of the sins and cosines.

$$\bar{\theta} = atan2\left(\frac{\sum_{i=1}^{n}\sin\theta_i}{n}, \frac{\sum_{i=1}^{n}\cos\theta_i}{n}\right)$$

`UnscentedKalmanFilter.__init__()` has an argument `x_mean_fn` for a function which computes the mean of the state, and `z_mean_fn` for a function which computes the mean of the measurement. We will code these function as:

```
In [54]: def state_mean(sigmas, Wm):
             x = np.zeros(3)

             sum_sin = np.sum(np.dot(np.sin(sigmas[:, 2]), Wm))
             sum_cos = np.sum(np.dot(np.cos(sigmas[:, 2]), Wm))
             x[0] = np.sum(np.dot(sigmas[:, 0], Wm))
             x[1] = np.sum(np.dot(sigmas[:, 1], Wm))
             x[2] = atan2(sum_sin, sum_cos)
             return x

         def z_mean(sigmas, Wm):
             z_count = sigmas.shape[1]
             x = np.zeros(z_count)

             for z in range(0, z_count, 2):
                 sum_sin = np.sum(np.dot(np.sin(sigmas[:, z+1]), Wm))
                 sum_cos = np.sum(np.dot(np.cos(sigmas[:, z+1]), Wm))

                 x[z] = np.sum(np.dot(sigmas[:,z], Wm))
                 x[z+1] = atan2(sum_sin, sum_cos)
             return x
```

These functions take advantage of the fact that NumPy's trigometric functions operate on arrays, and `dot` performs element-wise multiplication. NumPy is implemented in C and Fortran, so `sum(dot(sin(x), w))` is much faster than writing the equivalent loop in Python.

With that done we are now ready to implement the UKF. I want to point out that when I designed this filter I did not just design all of functions above in one sitting, from scratch. I put together a basic UKF

with predefined landmarks, verified it worked, then started filling in the pieces. "What if I see different landmarks?" That lead me to change the measurement function to accept an array of landmarks. "How do I deal with computing the residual of angles?" This led me to write the angle normalization code. "What is the *mean* of a set of angles?" I searched on the internet, found an article on Wikipedia, and implemented that algorithm. Do not be daunted. Design what you can, then ask questions and solve them, one by one.

You've seen the UKF implemention already, so I will not describe it in detail. There are two new thing here. When we construct the sigma points and filter we have to provide it the functions that we have written to compute the residuals and means.

```
points = SigmaPoints(n=3, alpha=.00001, beta=2, kappa=0,
                     subtract=residual_x)

ukf = UKF(dim_x=3, dim_z=2, fx=fx, hx=Hx, dt=dt, points=points,
          x_mean_fn=state_mean, z_mean_fn=z_mean,
          residual_x=residual_x, residual_z=residual_h)
```

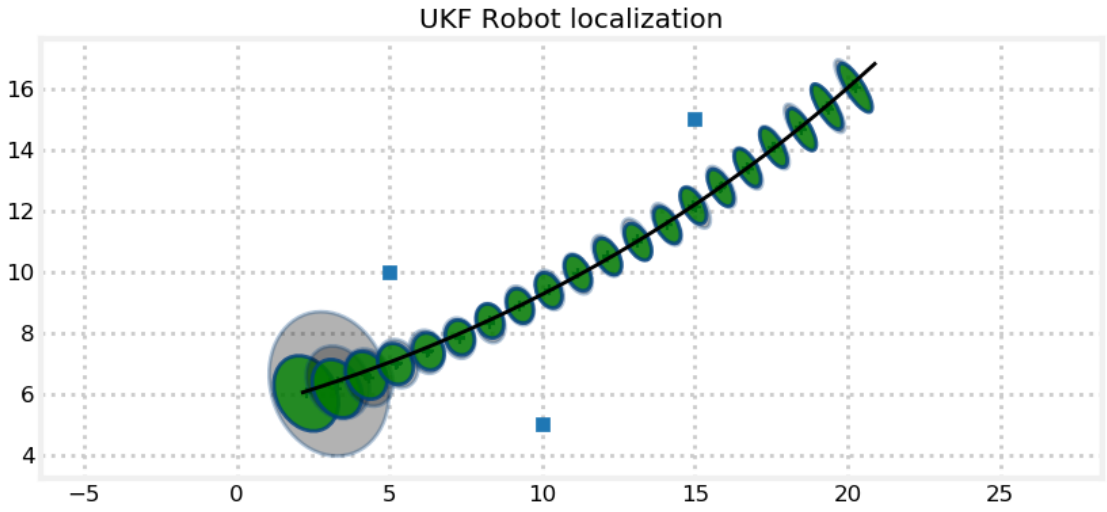
Next, we need to pass extra data into our `f(x, dt)` and `h(x)` functions. We want to use `move(x, dt, u, wheelbase)` for `f(x, dt)`, and `Hx(x, landmarks)` for `h(x)`. We can do this, we just have to pass the extra parameters into `predict()` and `update()` as keyword argument,s like so:

```
ukf.predict(u=u, wheelbase=wheelbase)
ukf.update(z, landmarks=landmarks)
```

The rest of the code runs the simulation and plots the results. I create a variable `landmarks` that contains the coordinates of the landmarks. I update the simulated robot position 10 times a second, but run the UKF only once per second. We are not using Runge Kutta to integrate the differential equations of motion, so a small time step makes the simulation more accurate.

```
In [55]: from filterpy.stats import plot_covariance_ellipse

         dt = 1.0
         wheelbase = 0.5

         def run_localization(
             cmds, landmarks, sigma_vel, sigma_steer, sigma_range,
             sigma_bearing, ellipse_step=1, step=10):

             plt.figure()
             points = MerweScaledSigmaPoints(n=3, alpha=.00001, beta=2, kappa=0,
                                             subtract=residual_x)
             ukf = UKF(dim_x=3, dim_z=2*len(landmarks), fx=move, hx=Hx,
                       dt=dt, points=points, x_mean_fn=state_mean,
                       z_mean_fn=z_mean, residual_x=residual_x,
                       residual_z=residual_h)

             ukf.x = np.array([2, 6, .3])
             ukf.P = np.diag([.1, .1, .05])
             ukf.R = np.diag([sigma_range**2,
                              sigma_bearing**2]*len(landmarks))
             ukf.Q = np.eye(3)*0.0001

             sim_pos = ukf.x.copy()

             # plot landmarks
             if len(landmarks) > 0:
```

```
            plt.scatter(landmarks[:, 0], landmarks[:, 1],
                        marker='s', s=60)

        track = []
        for i, u in enumerate(cmds):
            sim_pos = move(sim_pos, dt/step, u, wheelbase)
            track.append(sim_pos)

            if i % step == 0:
                ukf.predict(u=u, wheelbase=wheelbase)

                if i % ellipse_step == 0:
                    plot_covariance_ellipse(
                        (ukf.x[0], ukf.x[1]), ukf.P[0:2, 0:2], std=6,
                         facecolor='k', alpha=0.3)

                x, y = sim_pos[0], sim_pos[1]
                z = []
                for lmark in landmarks:
                    dx, dy = lmark[0] - x, lmark[1] - y
                    d = sqrt(dx**2 + dy**2) + randn()*sigma_range
                    bearing = atan2(lmark[1] - y, lmark[0] - x)
                    a = (normalize_angle(bearing - sim_pos[2] +
                         randn()*sigma_bearing))
                    z.extend([d, a])
                ukf.update(z, landmarks=landmarks)

                if i % ellipse_step == 0:
                    plot_covariance_ellipse(
                        (ukf.x[0], ukf.x[1]), ukf.P[0:2, 0:2], std=6,
                         facecolor='g', alpha=0.8)
        track = np.array(track)
        plt.plot(track[:, 0], track[:,1], color='k', lw=2)
        plt.axis('equal')
        plt.title("UKF Robot localization")
        plt.show()
        return ukf

In [56]: landmarks = np.array([[5, 10], [10, 5], [15, 15]])
         cmds = [np.array([1.1, .01])] * 200
         ukf = run_localization(
             cmds, landmarks, sigma_vel=0.1, sigma_steer=np.radians(1),
             sigma_range=0.3, sigma_bearing=0.1)
         print('Final P:', ukf.P.diagonal())
```




```
Final P: [0.0085 0.0177 0.0006]
```

The rest of the code runs the simulation and plots the results. I create a variable `landmarks` that contains the coordinates of the landmarks. I update the simulated robot position 10 times a second, but run the UKF only once. This is for two reasons. First, we are not using Runge Kutta to integrate the differental equations of motion, so a narrow time step allows our simulation to be more accurate. Second, it is fairly normal in embedded systems to have limited processing speed. This forces you to run your Kalman filter only as frequently as absolutely needed.

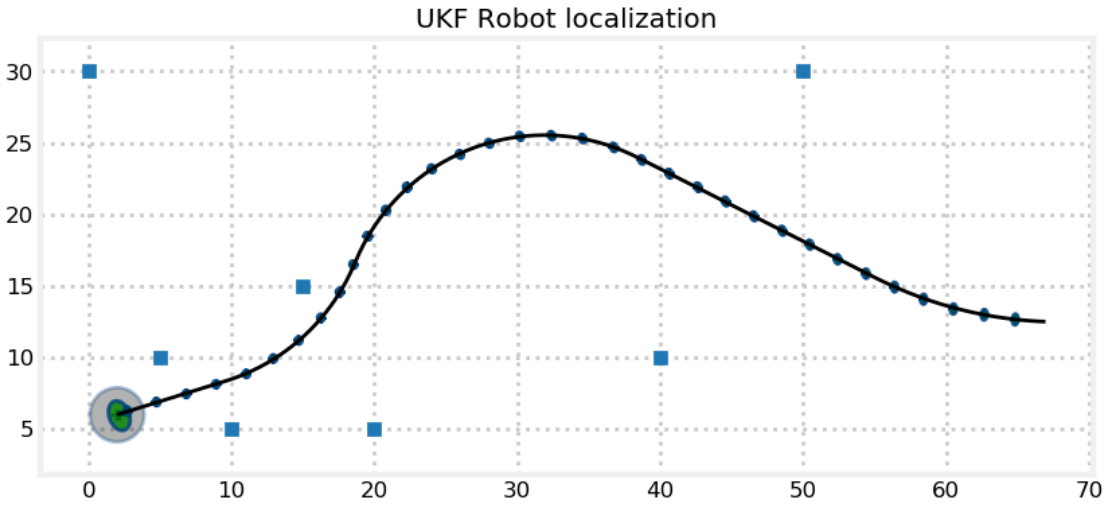
### 10.15.7 Steering the Robot

The steering simulation in the run above is not realistic. The velocity and steering angles never changed, which doesn't pose much of a problem for the Kalman filter. We could implement a complicated PID controlled robot simulation, but I will just generate varying steering commands using NumPy's `linspace` method. I'll also add more landmarks as the robot will be traveling much further than in the first example.

```
In [57]: landmarks = np.array([[5, 10], [10, 5], [15, 15], [20, 5],
                               [0, 30], [50, 30], [40, 10]])
         dt = 0.1
         wheelbase = 0.5
         sigma_range=0.3
         sigma_bearing=0.1

         def turn(v, t0, t1, steps):
           return [[v, a] for a in np.linspace(
                          np.radians(t0), np.radians(t1), steps)]

         # accelerate from a stop
         cmds = [[v, .0] for v in np.linspace(0.001, 1.1, 30)]
         cmds.extend([cmds[-1]]*50)

         # turn left
         v = cmds[-1][0]
         cmds.extend(turn(v, 0, 2, 15))
```

```
        cmds.extend([cmds[-1]]*100)

        #turn right
        cmds.extend(turn(v, 2, -2, 15))
        cmds.extend([cmds[-1]]*200)

        cmds.extend(turn(v, -2, 0, 15))
        cmds.extend([cmds[-1]]*150)

        cmds.extend(turn(v, 0, 1, 25))
        cmds.extend([cmds[-1]]*100)
```

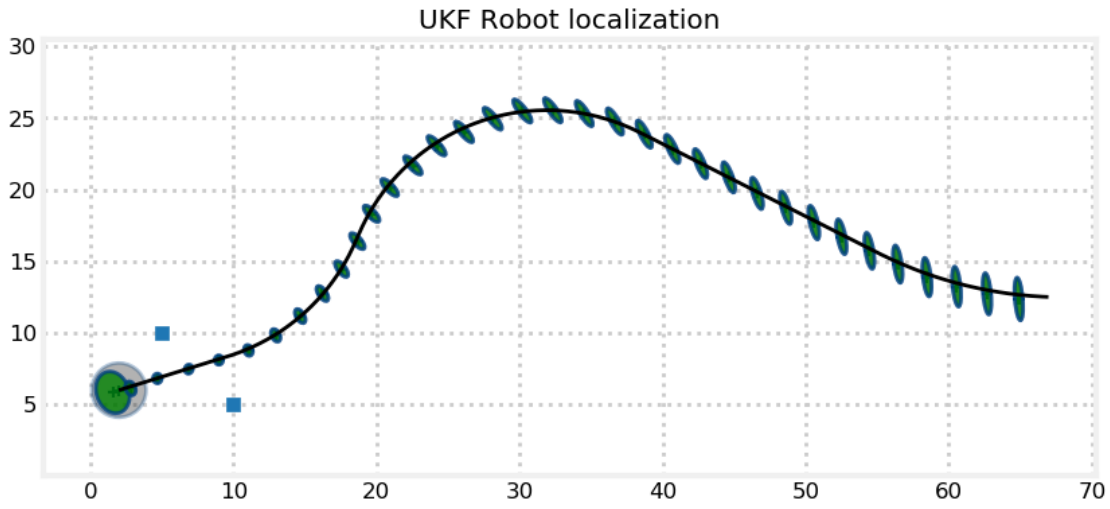
```
In [58]: ukf = run_localization(
             cmds, landmarks, sigma_vel=0.1, sigma_steer=np.radians(1),
             sigma_range=0.3, sigma_bearing=0.1, step=1,
             ellipse_step=20)
         print('final covariance', ukf.P.diagonal())
```




```
final covariance [0.0012 0.0042 0.0003]
```

The uncertainty becomes very small very quickly. The covariance ellipses are displaying the $6\sigma$ covariance, yet the ellipses are so small they are hard to see. We can incorporate more error into the answer by only supplying two landmarks near the start point. When we run this filter the errors increase as the robot gets further away from these landmarks.

```
In [59]: ukf = run_localization(
             cmds, landmarks[0:2], sigma_vel=0.1, sigma_steer=np.radians(1),
             sigma_range=0.3, sigma_bearing=0.1, step=1,
             ellipse_step=20)
         print('final covariance', ukf.P.diagonal())
```




```
final covariance [0.0025 0.0654 0.0007]
```

## 10.16 Discussion

Your impression of this chapter probably depends on how many nonlinear Kalman filters you have implemented in the past. If this is your first exposure perhaps the computation of $2n+1$ sigma points and the subsequent writing of the $f(x)$ and $h(x)$ function struck you as a bit finicky. Indeed, I spent more time than I'd care to admit getting everything working because of the need to handle the modular math of angles. On the other hand, if you have implemented an extended Kalman filter (EKF) perhaps you are bouncing gleefully in your seat. There is a small amount of tedium in writing the functions for the UKF, but the concepts are very basic. The EKF for the same problem requires some fairly difficult mathematics. For many problems we cannot find a closed form solution for the equations of the EKF, and we must retreat to some sort of iterated solution.

The advantage of the UKF over the EKF is not only the relative ease of implementation. It is somewhat premature to discuss this because you haven't learned the EKF yet, but the EKF linearizes the problem at one point and passes that point through a linear Kalman filter. In contrast, the UKF takes $2n+1$ samples. Therefore the UKF is often more accurate than the EKF, especially when the problem is highly nonlinear. While it is not true that the UKF is guaranteed to always outperform the EKF, in practice it has been shown to perform at least as well, and usually much better than the EKF.

Hence my recommendation is to always start by implementing the UKF. If your filter has real world consequences if it diverges (people die, lots of money lost, power plant blows up) of course you will have to engage in sophisticated analysis and experimentation to choose the best filter. That is beyond the scope of this book, and you should be going to graduate school to learn this theory.

Finally, I have spoken of the UKF as *the* way to perform sigma point filters. This is not true. The specific version I chose is Julier's scaled unscented filter as parameterized by Van der Merwe in his 2004 dissertation. If you search for Julier, Van der Merwe, Uhlmann, and Wan you will find a family of similar sigma point filters that they developed. Each technique uses a different way of choosing and weighting the sigma points. But the choices don't stop there. For example, the SVD Kalman filter uses singular value decomposition (SVD) to find the approximate mean and covariance of the probability distribution. Think of this chapter as an introduction to the sigma point filters, rather than a definitive treatment of how they work.

## 10.17 References


- [1] Rudolph Van der Merwe. "Sigma-Point Kalman Filters for Probabilistic Inference in Dynamic State-Space Models" dissertation (2004).
- [2] Simon J. Julier. "The Scaled Unscented Transformation". Proceedings of the American Control Conference 6. IEEE. (2002)
- [1] http://www.esdradar.com/brochures/Compact%20Tracking%2037250X.pdf
- [2] Julier, Simon J.; Uhlmann, Jeffrey "A New Extension of the Kalman Filter to Nonlinear Systems". Proc. SPIE 3068, Signal Processing, Sensor Fusion, and Target Recognition VI, 182 (July 28, 1997)
- [3] Cholesky decomposition. Wikipedia. http://en.wikipedia.org/wiki/Cholesky_decomposition

# Chapter 11

# The Extended Kalman Filter

```
In [1]: from __future__ import division, print_function
        %matplotlib inline
```

```
In [2]: #format the book
        import book_format
        book_format.set_style()
```

```
Out[2]:
```

We have developed the theory for the linear Kalman filter. Then, in the last two chapters we broached the topic of using Kalman filters for nonlinear problems. In this chapter we will learn the Extended Kalman filter (EKF). The EKF handles nonlinearity by linearizing the system at the point of the current estimate, and then the linear Kalman filter is used to filter this linearized system. It was one of the very first techniques used for nonlinear problems, and it remains the most common technique.

The EKF provides significant mathematical challenges to the designer of the filter; this is the most challenging chapter of the book. I do everything I can to avoid the EKF in favor of other techniques that have been developed to filter nonlinear problems. However, the topic is unavoidable; all classic papers and a majority of current papers in the field use the EKF. Even if you do not use the EKF in your own work you will need to be familiar with the topic to be able to read the literature.

## 11.1 Linearizing the Kalman Filter

The Kalman filter uses linear equations, so it does not work with nonlinear problems. Problems can be nonlinear in two ways. First, the process model might be nonlinear. An object falling through the atmosphere encounters drag which reduces its acceleration. The drag coefficient varies based on the velocity the object. The resulting behavior is nonlinear - it cannot be modeled with linear equations. Second, the measurements could be nonlinear. For example, a radar gives a range and bearing to a target. We use trigonometry, which is nonlinear, to compute the position of the target.

For the linear filter we have these equations for the process and measurement models:

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + w_x$$
$$\mathbf{z} = \mathbf{H}\mathbf{x} + w_z$$

Where $\mathbf{A}$ is the systems dynamic matrix. Using the state space methods covered in the **Kalman Filter Math** chapter these equations can be tranformed into

$$\bar{\mathbf{x}} = \mathbf{F}\mathbf{x}$$
$$\mathbf{z} = \mathbf{H}\mathbf{x}$$

where $\mathbf{F}$ is the *fundamental matrix*. The noise $w_x$ and $w_z$ terms are incorporated into the matrices $\mathbf{R}$ and $\mathbf{Q}$. This form of the equations allow us to compute the state at step $k$ given a measurement at step $k$ and

the state estimate at step $k-1$. In earlier chapters I built your intuition and minimized the math by using problems describable with Newton's equations. We know how to design $\mathbf{F}$ based on high school physics.

For the nonlinear model the linear expression $\mathbf{Fx} + \mathbf{Bu}$ is replaced by a nonlinear function $f(\mathbf{x}, \mathbf{u})$, and the linear expression $\mathbf{Hx}$ is replaced by a nonlinear function $h(\mathbf{x})$:

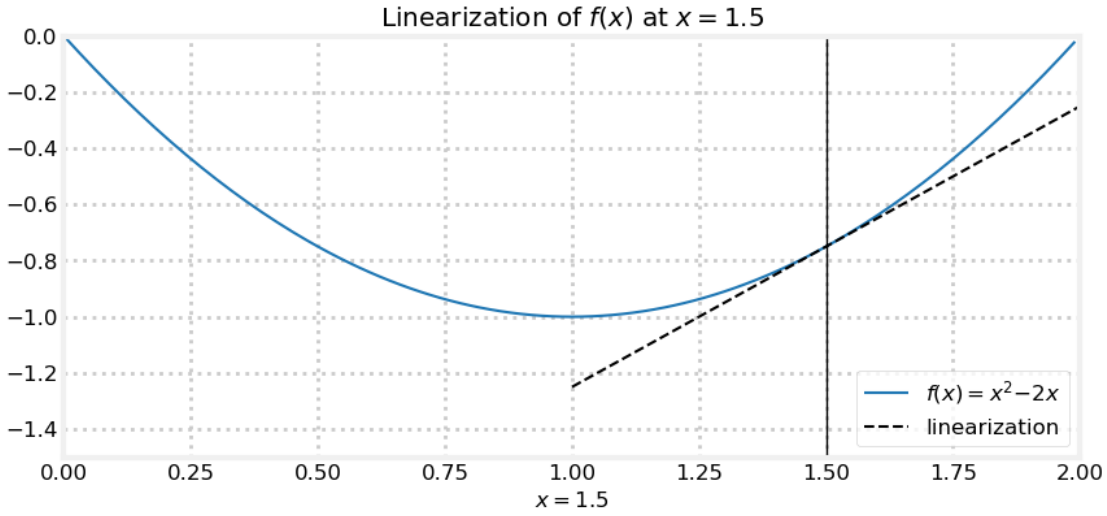
$$\dot{\mathbf{x}} = f(\mathbf{x}, \mathbf{u}) + w_x$$
$$\mathbf{z} = h(\mathbf{x}) + w_z$$

You might imagine that we could proceed by finding a new set of Kalman filter equations that optimally solve these equations. But if you remember the charts in the **Nonlinear Filtering** chapter you'll recall that passing a Gaussian through a nonlinear function results in a probability distribution that is no longer Gaussian. So this will not work.

The EKF does not alter the Kalman filter's linear equations. Instead, it *linearizes* the nonlinear equations at the point of the current estimate, and uses this linearization in the linear Kalman filter.

*Linearize* means what it sounds like. We find a line that most closely matches the curve at a defined point. The graph below linearizes the parabola $f(x) = x^2 - 2x$ at $x = 1.5$.

```
In [3]: import kf_book.ekf_internal as ekf_internal
        ekf_internal.show_linearization()
```




If the curve above is the process model, then the dotted lines shows the linearization of that curve for the estimate $x = 1.5$.

We linearize systems by taking the derivative, which finds the slope of a curve:

$$f(x) = x^2 - 2x$$
$$\frac{df}{dx} = 2x - 2$$

and then evaluating it at $x$:

$$\begin{aligned} m &= f'(x = 1.5) \\ &= 2(1.5) - 2 \\ &= 1 \end{aligned}$$

Linearizing systems of differential equations is similar. We linearize $f(\mathbf{x}, \mathbf{u})$, and $h(\mathbf{x})$ by taking the partial derivatives of each to evaluate $\mathbf{F}$ and $\mathbf{H}$ at the point $\mathbf{x}_t$ and $\mathbf{u}_t$. We call the partial derivative of a matrix the *Jacobian*. This gives us the the discrete state transition matrix and measurement model matrix:

$$\mathbf{F} = \left.\frac{\partial f(\mathbf{x}_t, \mathbf{u}_t)}{\partial \mathbf{x}}\right|_{\mathbf{x}_t, \mathbf{u}_t}$$
$$\mathbf{H} = \left.\frac{\partial h(\bar{\mathbf{x}}_t)}{\partial \bar{\mathbf{x}}}\right|_{\bar{\mathbf{x}}_t}$$

This leads to the following equations for the EKF. I put boxes around the differences from the linear filter:

| linear Kalman filter | EKF |
|---|---|
| | $\boxed{\mathbf{F} = \left.\frac{\partial f(\mathbf{x}_t, \mathbf{u}_t)}{\partial \mathbf{x}}\right\|_{\mathbf{x}_t, \mathbf{u}_t}}$ |
| $\mathbf{x} = \mathbf{Fx} + \mathbf{Bu}$ | $\boxed{\mathbf{x} = f(\mathbf{x}, \mathbf{u})}$ |
| $\mathbf{P} = \mathbf{FPF}^\mathsf{T} + \mathbf{Q}$ | $\mathbf{P} = \mathbf{FPF}^\mathsf{T} + \mathbf{Q}$ |
| | $\boxed{\mathbf{H} = \left.\frac{\partial h(\bar{\mathbf{x}}_t)}{\partial \bar{\mathbf{x}}}\right\|_{\bar{\mathbf{x}}_t}}$ |
| $\mathbf{y} = \mathbf{z} - \mathbf{Hx}$ | $\mathbf{y} = \mathbf{z} - \boxed{h(\bar{x})}$ |
| $\mathbf{K} = \mathbf{PH}^\mathsf{T}(\mathbf{HPH}^\mathsf{T} + \mathbf{R})^{-1}$ | $\mathbf{K} = \mathbf{PH}^\mathsf{T}(\mathbf{HPH}^\mathsf{T} + \mathbf{R})^{-1}$ |
| $\mathbf{x} = \mathbf{x} + \mathbf{Ky}$ | $\mathbf{x} = \mathbf{x} + \mathbf{Ky}$ |
| $\mathbf{P} = (\mathbf{I} - \mathbf{KH})\mathbf{P}$ | $\mathbf{P} = (\mathbf{I} - \mathbf{KH})\mathbf{P}$ |

We don't normally use $\mathbf{Fx}$ to propagate the state for the EKF as the linearization causes inaccuracies. It is typical to compute $\bar{\mathbf{x}}$ using a suitable numerical integration technique such as Euler or Runge Kutta. Thus I wrote $\mathbf{x} = f(\mathbf{x}, \mathbf{u})$. For the same reasons we don't use $\mathbf{Hx}$ in the computation for the residual, opting for the more accurate $h(\bar{\mathbf{x}})$.

I think the easiest way to understand the EKF is to start off with an example. Later you may want to come back and reread this section.

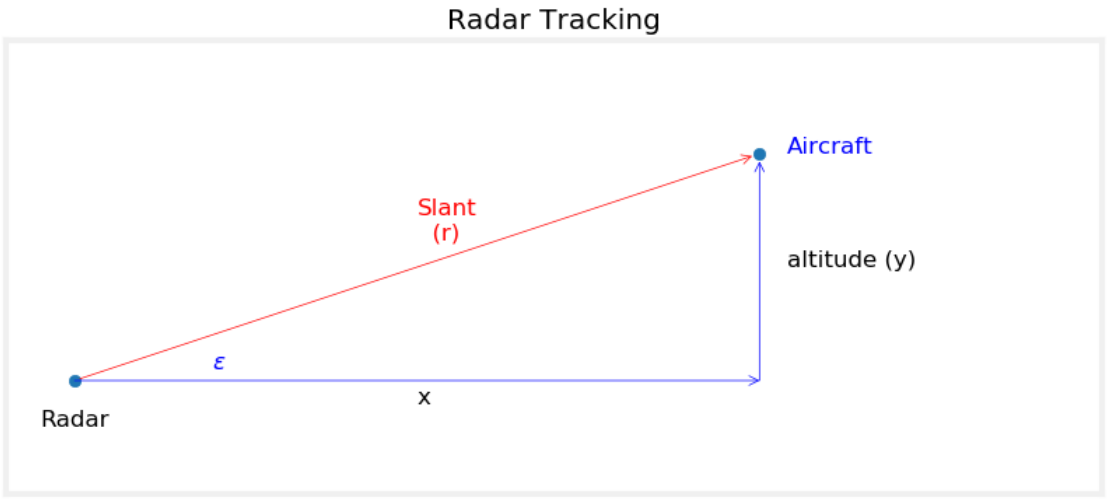
## 11.2 Example: Tracking a Airplane

This example tracks an airplane using ground based radar. We implemented a UKF for this problem in the last chapter. Now we will implement an EKF for the same problem so we can compare both the filter performance and the level of effort required to implement the filter.

Radars work by emitting a beam of radio waves and scanning for a return bounce. Anything in the beam's path will reflects some of the signal back to the radar. By timing how long it takes for the reflected signal to get back to the radar the system can compute the *slant distance* - the straight line distance from the radar installation to the object.

The relationship between the radar's slant range distance $r$ and elevation angle $\epsilon$ with the horizontal position $x$ and altitude $y$ of the aircraft is illustrated in the figure below:

```
In [4]: ekf_internal.show_radar_chart()
```




This gives us the equalities:

$$\epsilon = \tan^{-1}\frac{y}{x}$$
$$r^2 = x^2 + y^2$$

### 11.2.1 Design the State Variables

We want to track the position of an aircraft assuming a constant velocity and altitude, and measurements of the slant distance to the aircraft. That means we need 3 state variables - horizontal distance, horizonal velocity, and altitude:

$$\mathbf{x} = \begin{bmatrix}\texttt{distance}\\ \texttt{velocity}\\ \texttt{altitude}\end{bmatrix} = \begin{bmatrix}x\\ \dot{x}\\ y\end{bmatrix}$$

### 11.2.2 Design the Process Model

We assume a Newtonian, kinematic system for the aircraft. We've used this model in previous chapters, so by inspection you may recognize that we want

$$\mathbf{F} = \left[\begin{array}{cc|c} 1 & \Delta t & 0 \\ 0 & 1 & 0 \\ \hline 0 & 0 & 1 \end{array}\right]$$

I've partioned the matrix into blocks to show the upper left block is a constant velocity model for $x$, and the lower right block is a constant position model for $y$.

However, let's practice finding these matrices. We model systems with a set of differential equations. We need an equation in the form

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{w}$$

where $\mathbf{w}$ is the system noise.

The variables $x$ and $y$ are independent so we can compute them separately. The differential equations for motion in one dimension are:

$$v = \dot{x}$$
$$a = \ddot{x} = 0$$

Now we put the differential equations into state-space form. If this was a second or greater order differential system we would have to first reduce them to an equivalent set of first degree equations. The equations are first order, so we put them in state space matrix form as

$$\begin{bmatrix}\dot{x}\\ \ddot{x}\end{bmatrix} = \begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}\begin{bmatrix}x\\ \dot{x}\end{bmatrix}$$
$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x}$$

where $\mathbf{A} = \begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}$.

Recall that $\mathbf{A}$ is the *system dynamics matrix*. It describes a set of linear differential equations. From it we must compute the state transition matrix $\mathbf{F}$. $\mathbf{F}$ describes a discrete set of linear equations which compute $\mathbf{x}$ for a discrete time step $\Delta t$.

A common way to compute $\mathbf{F}$ is to use the power series expansion of the matrix exponential:

$$\mathbf{F}(\Delta t) = e^{\mathbf{A}\Delta t} = \mathbf{I} + \mathbf{A}\Delta t + \frac{(\mathbf{A}\Delta t)^2}{2!} + \frac{(\mathbf{A}\Delta t)^3}{3!} + ...$$

$\mathbf{A}^2 = \begin{bmatrix}0 & 0\\ 0 & 0\end{bmatrix}$, so all higher powers of $\mathbf{A}$ are also $\mathbf{0}$. Thus the power series expansion is:

$$\begin{aligned}\mathbf{F} &= \mathbf{I} + \mathbf{A}t + \mathbf{0}\\ &= \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} + \begin{bmatrix}0 & 1\\ 0 & 0\end{bmatrix}\Delta t\\ \mathbf{F} &= \begin{bmatrix}1 & \Delta t\\ 0 & 1\end{bmatrix}\end{aligned}$$

This is the same result used by the kinematic equations! This exercise was unnecessary other than to illustrate finding the state transition matrix from linear differential equations. We will conclude the chapter with an example that will require the use of this technique.

### 11.2.3 Design the Measurement Model

The measurement function takes the state estimate of the prior $\bar{\mathbf{x}}$ and turn it into a measurement of the slant range distance. We use the Pythagorean theorem to derive:

$$h(\bar{\mathbf{x}}) = \sqrt{x^2 + y^2}$$

The relationship between the slant distance and the position on the ground is nonlinear due to the square root. We linearize it by evaluating its partial derivative at $\mathbf{x}_t$:

$$\mathbf{H} = \left.\frac{\partial h(\bar{\mathbf{x}})}{\partial \bar{\mathbf{x}}}\right|_{\bar{\mathbf{x}}_t}$$

The partial derivative of a matrix is called a Jacobian, and takes the form

$$\frac{\partial \mathbf{H}}{\partial \bar{\mathbf{x}}} = \begin{bmatrix}\frac{\partial h_1}{\partial x_1} & \frac{\partial h_1}{\partial x_2} & \cdots\\ \frac{\partial h_2}{\partial x_1} & \frac{\partial h_2}{\partial x_2} & \cdots\\ \vdots & \vdots & \end{bmatrix}$$

In other words, each element in the matrix is the partial derivative of the function $h$ with respect to the $x$ variables. For our problem we have

$$\mathbf{H} = \begin{bmatrix}\partial h/\partial x & \partial h/\partial \dot{x} & \partial h/\partial y\end{bmatrix}$$

Solving each in turn:

$$\begin{aligned}\frac{\partial h}{\partial x} &= \frac{\partial}{\partial x}\sqrt{x^2+y^2} \\ &= \frac{x}{\sqrt{x^2+y^2}}\end{aligned}$$

and

$$\begin{aligned}\frac{\partial h}{\partial \dot{x}} &= \frac{\partial}{\partial \dot{x}}\sqrt{x^2+y^2} \\ &= 0\end{aligned}$$

and

$$\begin{aligned}\frac{\partial h}{\partial y} &= \frac{\partial}{\partial y}\sqrt{x^2+y^2} \\ &= \frac{y}{\sqrt{x^2+y^2}}\end{aligned}$$

giving us

$$\mathbf{H} = \begin{bmatrix} \frac{x}{\sqrt{x^2+y^2}} & 0 & \frac{y}{\sqrt{x^2+y^2}} \end{bmatrix}$$

This may seem daunting, so step back and recognize that all of this math is doing something very simple. We have an equation for the slant range to the airplane which is nonlinear. The Kalman filter only works with linear equations, so we need to find a linear equation that approximates $\mathbf{H}$. As we discussed above, finding the slope of a nonlinear equation at a given point is a good approximation. For the Kalman filter, the 'given point' is the state variable $\mathbf{x}$ so we need to take the derivative of the slant range with respect to $\mathbf{x}$. For the linear Kalman filter $\mathbf{H}$ was a constant that we computed prior to running the filter. For the EKF $\mathbf{H}$ is updated at each step as the evaluation point $\bar{\mathbf{x}}$ changes at each epoch.

To make this more concrete, let's now write a Python function that computes the Jacobian of $h$ for this problem.

```
In [5]: from math import sqrt
        def HJacobian_at(x):
            """ compute Jacobian of H matrix at x """

            horiz_dist = x[0]
            altitude   = x[2]
            denom = sqrt(horiz_dist**2 + altitude**2)
            return array ([[horiz_dist/denom, 0., altitude/denom]])
```

Finally, let's provide the code for $h(\bar{\mathbf{x}})$:

```
In [6]: def hx(x):
            """ compute measurement for slant range that
            would correspond to state x.
            """

            return (x[0]**2 + x[2]**2) ** 0.5
```

Now let's write a simulation for our radar.

```
In [7]: from numpy.random import randn
        import math

        class RadarSim:
            """ Simulates the radar signal returns from an object
```

```
    flying at a constant altityude and velocity in 1D.
    """

    def __init__(self, dt, pos, vel, alt):
        self.pos = pos
        self.vel = vel
        self.alt = alt
        self.dt = dt

    def get_range(self):
        """ Returns slant range to the object. Call once
        for each new measurement at dt time from last call.
        """

        # add some process noise to the system
        self.vel = self.vel  + .1*randn()
        self.alt = self.alt + .1*randn()
        self.pos = self.pos + self.vel*self.dt

        # add measurement noise
        err = self.pos * 0.05*randn()
        slant_dist = math.sqrt(self.pos**2 + self.alt**2)

        return slant_dist + err
```

### 11.2.4 Design Process and Measurement Noise

The radar measures the range to a target. We will use $\sigma_{range} = 5$ meters for the noise. This gives us

$$\mathbf{R} = \begin{bmatrix}\sigma^2_{range}\end{bmatrix} = \begin{bmatrix}25\end{bmatrix}$$

The design of $\mathbf{Q}$ requires some discussion. The state $\mathbf{x} = \begin{bmatrix}x & \dot{x} & y\end{bmatrix}^\mathsf{T}$. The first two elements are position (down range distance) and velocity, so we can use `Q_discrete_white_noise` noise to compute the values for the upper left hand side of $\mathbf{Q}$. The third element of $\mathbf{x}$ is altitude, which we are assuming is independent of the down range distance. That leads us to a block design of $\mathbf{Q}$ of:

$$\mathbf{Q} = \begin{bmatrix}\mathbf{Q}_\mathbf{x} & 0 \\ 0 & \mathbf{Q}_\mathbf{y}\end{bmatrix}$$

### 11.2.5 Implementation

`FilterPy` provides the class `ExtendedKalmanFilter`. It works similarly to the `KalmanFilter` class we have been using, except that it allows you to provide a function that computes the Jacobian of $\mathbf{H}$ and the function $h(\mathbf{x})$.

We start by importing the filter and creating it. The dimension of `x` is 3 and `z` has dimension 1.

```
from filterpy.kalman import ExtendedKalmanFilter

rk = ExtendedKalmanFilter(dim_x=3, dim_z=1)
```

We create the radar simulator:

```
radar = RadarSim(dt, pos=0., vel=100., alt=1000.)
```

We will initialize the filter near the airplane's actual position:

```
rk.x = array([radar.pos, radar.vel-10, radar.alt+100])
```

We assign the system matrix using the first term of the Taylor series expansion we computed above:

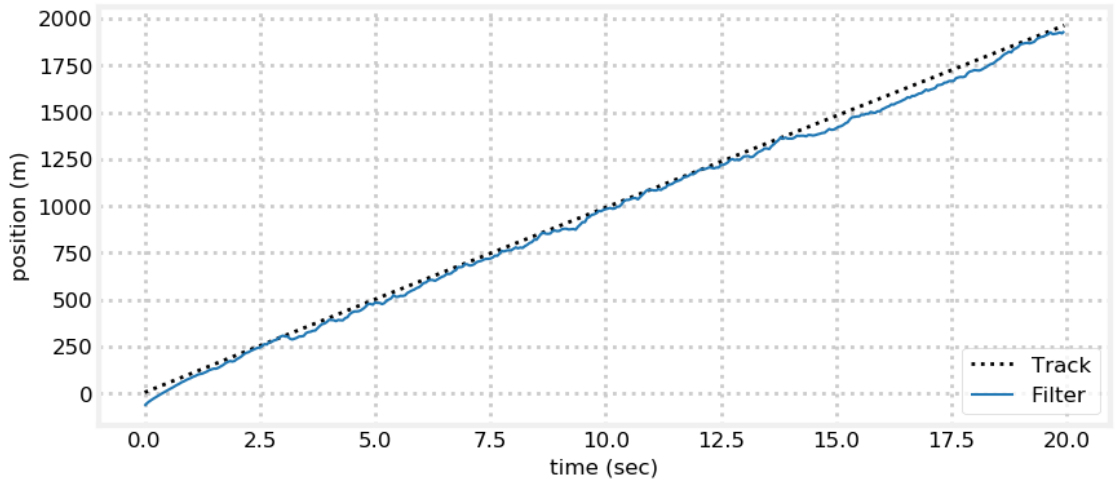
```
dt = 0.05
rk.F = eye(3) + array([[0, 1, 0],
                       [0, 0, 0],
                       [0, 0, 0]])*dt
```

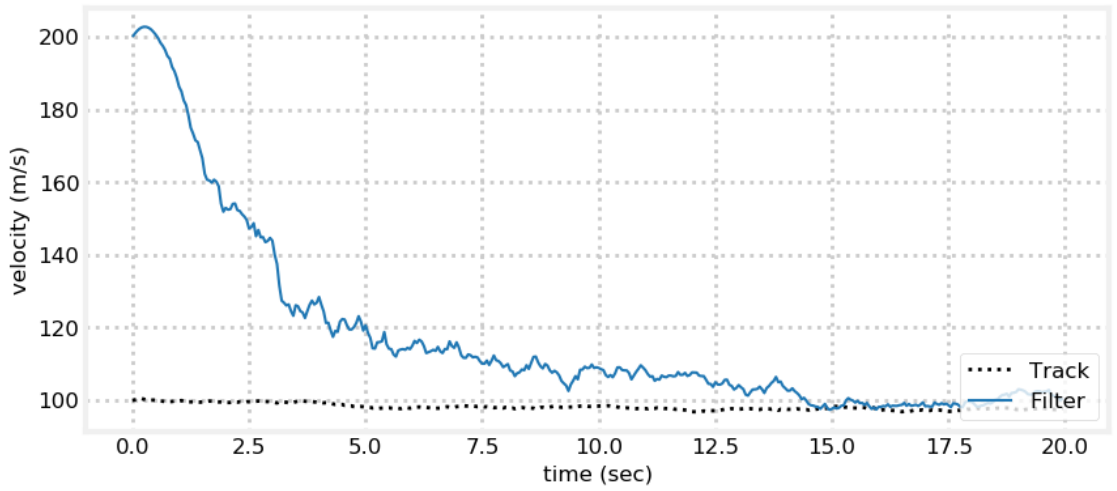
After assigning reasonable values to $\mathbf{R}$, $\mathbf{Q}$, and $\mathbf{P}$ we can run the filter with a simple loop. We pass the functions for computing the Jacobian of $\mathbf{H}$ and $h(x)$ into the `update` method.

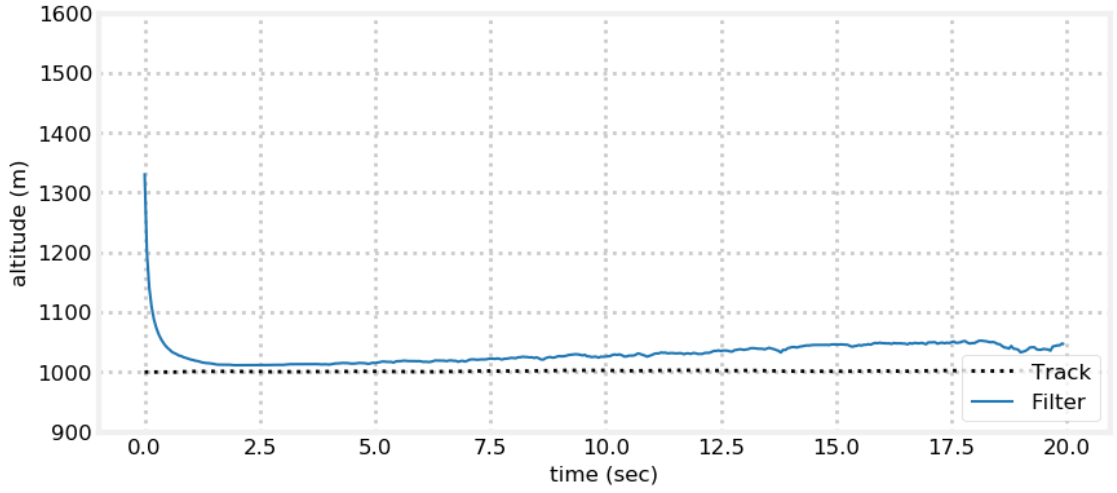
```
for i in range(int(20/dt)):
    z = radar.get_range()
    rk.update(array([z]), HJacobian_at, hx)
    rk.predict()
```

Adding some boilerplate code to save and plot the results we get:

```
In [8]: from filterpy.common import Q_discrete_white_noise
        from filterpy.kalman import ExtendedKalmanFilter
        from numpy import eye, array, asarray
        import numpy as np

        dt = 0.05
        rk = ExtendedKalmanFilter(dim_x=3, dim_z=1)
        radar = RadarSim(dt, pos=0., vel=100., alt=1000.)

        # make an imperfect starting guess
        rk.x = array([radar.pos-100, radar.vel+100, radar.alt+1000])

        rk.F = eye(3) + array([[0, 1, 0],
                               [0, 0, 0],
                               [0, 0, 0]]) * dt

        range_std = 5. # meters
        rk.R = np.diag([range_std**2])
        rk.Q[0:2, 0:2] = Q_discrete_white_noise(2, dt=dt, var=0.1)
        rk.Q[2,2] = 0.1
        rk.P *= 50

        xs, track = [], []
        for i in range(int(20/dt)):
            z = radar.get_range()
            track.append((radar.pos, radar.vel, radar.alt))

            rk.update(array([z]), HJacobian_at, hx)
            xs.append(rk.x)
            rk.predict()

        xs = asarray(xs)
        track = asarray(track)
        time = np.arange(0, len(xs)*dt, dt)
        ekf_internal.plot_radar(xs, track, time)
```


position (m)
time (sec)
Track
Filter


velocity (m/s)
time (sec)
Track
Filter


altitude (m)
time (sec)
Track
Filter

## 11.3 Using SymPy to compute Jacobians

Depending on your experience with derivatives you may have found the computation of the Jacobian difficult. Even if you found it easy, a slightly more difficult problem easily leads to very difficult computations.

As explained in Appendix A, we can use the SymPy package to compute the Jacobian for us.

```
In [9]: import sympy
        from IPython.display import display
        sympy.init_printing(use_latex='mathjax')

        x, x_vel, y = sympy.symbols('x, x_vel y')

        H = sympy.Matrix([sympy.sqrt(x**2 + y**2)])

        state = sympy.Matrix([x, x_vel, y])
        J = H.jacobian(state)

        display(state)
        display(J)
```

$$\begin{bmatrix} x \\ x_{vel} \\ y \end{bmatrix}$$

$$\begin{bmatrix} \frac{x}{\sqrt{x^2+y^2}} & 0 & \frac{y}{\sqrt{x^2+y^2}} \end{bmatrix}$$

This result is the same as the result we computed above, and with much less effort on our part!

## 11.4 Robot Localization

It's time to try a real problem. I warn you that this section is difficult. However, most books choose simple, textbook problems with simple answers, and you are left wondering how to solve a real world problem.

We will consider the problem of robot localization. We already implemented this in the **Unscented Kalman Filter** chapter, and I recommend you read it now if you haven't already. In this scenario we have a robot that is moving through a landscape using a sensor to detect landmarks. This could be a self driving car using computer vision to identify trees, buildings, and other landmarks. It might be one of those small robots that vacuum your house, or a robot in a warehouse.

The robot has 4 wheels in the same configuration used by automobiles. It maneuvers by pivoting the front wheels. This causes the robot to pivot around the rear axle while moving forward. This is nonlinear behavior which we will have to model.

The robot has a sensor that measures the range and bearing to known targets in the landscape. This is nonlinear because computing a position from a range and bearing requires square roots and trigonometry.

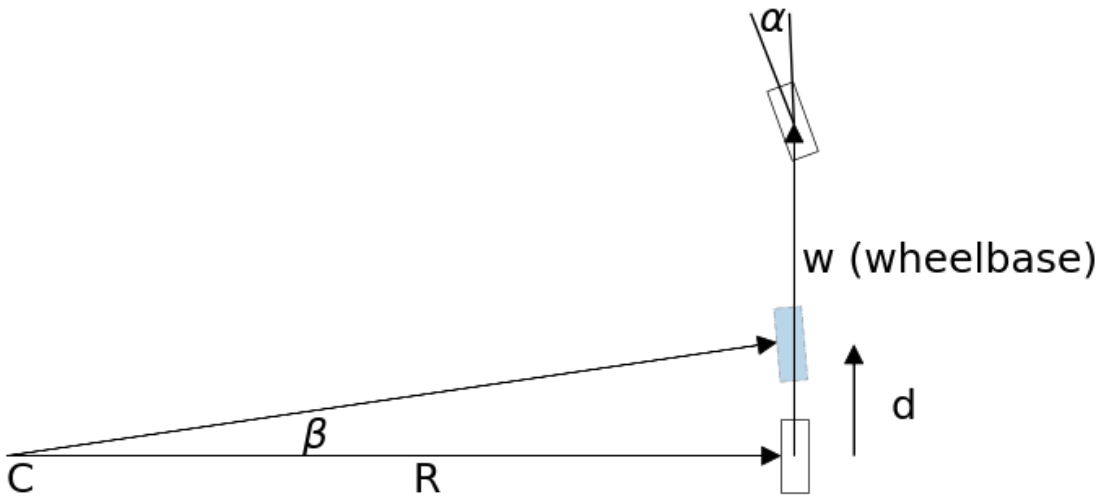
Both the process model and measurement models are nonlinear. The EKF accommodates both, so we provisionally conclude that the EKF is a viable choice for this problem.

### 11.4.1 Robot Motion Model

At a first approximation an automobile steers by pivoting the front tires while moving forward. The front of the car moves in the direction that the wheels are pointing while pivoting around the rear tires. This simple description is complicated by issues such as slippage due to friction, the differing behavior of the rubber tires at different speeds, and the need for the outside tire to travel a different radius than the inner tire. Accurately modeling steering requires a complicated set of differential equations.

For lower speed robotic applications a simpler *bicycle model* has been found to perform well. This is a depiction of the model:

```
In [10]: ekf_internal.plot_bicycle()
```




In the **Unscented Kalman Filter** chapter we derived these equations:

$$\begin{aligned}
\beta &= \frac{d}{w}\tan(\alpha) \\
x &= x - R\sin(\theta) + R\sin(\theta+\beta) \\
y &= y + R\cos(\theta) - R\cos(\theta+\beta) \\
\theta &= \theta + \beta
\end{aligned}$$

where $\theta$ is the robot's heading.

You do not need to understand this model in detail if you are not interested in steering models. The important thing to recognize is that our motion model is nonlinear, and we will need to deal with that with our Kalman filter.

### 11.4.2 Design the State Variables

For our filter we will maintain the position $x, y$ and orientation $\theta$ of the robot:

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix}$$

Our control input $\mathbf{u}$ is the velocity $v$ and steering angle $\alpha$:

$$\mathbf{u} = \begin{bmatrix} v \\ \alpha \end{bmatrix}$$

### 11.4.3 Design the System Model

We model our system as a nonlinear motion model plus noise.

$$\bar{x} = f(x, u) + \mathcal{N}(0, Q)$$

Using the motion model for a robot that we created above, we can expand this to

$$\begin{bmatrix}\bar{x}\\ y\\ \theta\end{bmatrix} = \begin{bmatrix}x\\ y\\ \theta\end{bmatrix} + \begin{bmatrix}-R\sin(\theta) + R\sin(\theta+\beta)\\ R\cos(\theta) - R\cos(\theta+\beta)\\ \beta\end{bmatrix}$$

We find The $\mathbf{F}$ by taking the Jacobian of $f(x,u)$.

$$\mathbf{F} = \frac{\partial f(x,u)}{\partial x} = \begin{bmatrix}\frac{\partial f_1}{\partial x} & \frac{\partial f_1}{\partial y} & \frac{\partial f_1}{\partial \theta}\\ \frac{\partial f_2}{\partial x} & \frac{\partial f_2}{\partial y} & \frac{\partial f_2}{\partial \theta}\\ \frac{\partial f_3}{\partial x} & \frac{\partial f_3}{\partial y} & \frac{\partial f_3}{\partial \theta}\end{bmatrix}$$

When we calculate these we get

$$\mathbf{F} = \begin{bmatrix}1 & 0 & -R\cos(\theta) + R\cos(\theta+\beta)\\ 0 & 1 & -R\sin(\theta) + R\sin(\theta+\beta)\\ 0 & 0 & 1\end{bmatrix}$$

We can double check our work with SymPy.

```
In [11]: import sympy
         from sympy.abc import alpha, x, y, v, w, R, theta
         from sympy import symbols, Matrix
         sympy.init_printing(use_latex="mathjax", fontsize='16pt')
         time = symbols('t')
         d = v*time
         beta = (d/w)*sympy.tan(alpha)
         r = w/sympy.tan(alpha)

         fxu = Matrix([[x-r*sympy.sin(theta) + r*sympy.sin(theta+beta)],
                       [y+r*sympy.cos(theta)- r*sympy.cos(theta+beta)],
                       [theta+beta]])
         F = fxu.jacobian(Matrix([x, y, theta]))
         F
```

Out[11]:

$$\begin{bmatrix}1 & 0 & -\frac{w\cos(\theta)}{\tan(\alpha)} + \frac{w\cos\left(\frac{tv\tan(\alpha)}{w}+\theta\right)}{\tan(\alpha)}\\ 0 & 1 & -\frac{w\sin(\theta)}{\tan(\alpha)} + \frac{w\sin\left(\frac{tv\tan(\alpha)}{w}+\theta\right)}{\tan(\alpha)}\\ 0 & 0 & 1\end{bmatrix}$$

That looks a bit complicated. We can use SymPy to substitute terms:

```
In [12]: # reduce common expressions
         B, R = symbols('beta, R')
         F = F.subs((d/w)*sympy.tan(alpha), B)
         F.subs(w/sympy.tan(alpha), R)
```

Out[12]:

$$\begin{bmatrix}1 & 0 & -R\cos(\theta) + R\cos(\beta+\theta)\\ 0 & 1 & -R\sin(\theta) + R\sin(\beta+\theta)\\ 0 & 0 & 1\end{bmatrix}$$

This form verifies that the computation of the Jacobian is correct.

Now we can turn our attention to the noise. Here, the noise is in our control input, so it is in *control space*. In other words, we command a specific velocity and steering angle, but we need to convert that into errors in $x, y, \theta$. In a real system this might vary depending on velocity, so it will need to be recomputed for every prediction. I will choose this as the noise model; for a real robot you will need to choose a model that accurately depicts the error in your system.

$$\mathbf{M} = \begin{bmatrix}\sigma_{vel}^2 & 0 \\ 0 & \sigma_\alpha^2\end{bmatrix}$$

If this was a linear problem we would convert from control space to state space using the by now familiar $\mathbf{FMF}^\mathsf{T}$ form. Since our motion model is nonlinear we do not try to find a closed form solution to this, but instead linearize it with a Jacobian which we will name $\mathbf{V}$.

$$\mathbf{V} = \frac{\partial f(x, u)}{\partial u}\begin{bmatrix}\frac{\partial f_1}{\partial v} & \frac{\partial f_1}{\partial \alpha} \\ \frac{\partial f_2}{\partial v} & \frac{\partial f_2}{\partial \alpha} \\ \frac{\partial f_3}{\partial v} & \frac{\partial f_3}{\partial \alpha}\end{bmatrix}$$

These partial derivatives become very difficult to work with. Let's compute them with SymPy.

```
In [13]: V = fxu.jacobian(Matrix([v, alpha]))
         V = V.subs(sympy.tan(alpha)/w, 1/R)
         V = V.subs(time*v/R, B)
         V = V.subs(time*v, 'd')
         V
```

Out[13]:

$$\begin{bmatrix}t\cos(\beta+\theta) & \frac{d(\tan^2(\alpha)+1)\cos(\beta+\theta)}{\tan(\alpha)} - \frac{w(-\tan^2(\alpha)-1)\sin(\theta)}{\tan^2(\alpha)} + \frac{w(-\tan^2(\alpha)-1)\sin(\beta+\theta)}{\tan^2(\alpha)} \\ t\sin(\beta+\theta) & \frac{d(\tan^2(\alpha)+1)\sin(\beta+\theta)}{\tan(\alpha)} + \frac{w(-\tan^2(\alpha)-1)\cos(\theta)}{\tan^2(\alpha)} - \frac{w(-\tan^2(\alpha)-1)\cos(\beta+\theta)}{\tan^2(\alpha)} \\ \frac{t}{R} & \frac{d(\tan^2(\alpha)+1)}{w}\end{bmatrix}$$

This should give you an appreciation of how quickly the EKF become mathematically intractable.

This gives us the final form of our prediction equations:

$$\mathbf{x} = \mathbf{x} + \begin{bmatrix}-R\sin(\theta) + R\sin(\theta+\beta) \\ R\cos(\theta) - R\cos(\theta+\beta) \\ \beta\end{bmatrix}$$

$$\mathbf{P} = \mathbf{FPF}^\mathsf{T} + \mathbf{VMV}^\mathsf{T}$$

This form of linearization is not the only way to predict $\mathbf{x}$. For example, we could use a numerical integration technique such as *Runge Kutta* to compute the movement of the robot. This will be required if the time step is relatively large. Things are not as cut and dried with the EKF as for the Kalman filter. For a real problem you have to carefully model your system with differential equations and then determine the most appropriate way to solve that system. The correct approach depends on the accuracy you require, how nonlinear the equations are, your processor budget, and numerical stability concerns.

### 11.4.4 Design the Measurement Model

The robot's sensor provides a noisy bearing and range measurement to multiple known locations in the landscape. The measurement model must convert the state $\begin{bmatrix}x & y & \theta\end{bmatrix}^\mathsf{T}$ into a range and bearing to the landmark. If $\mathbf{p}$ is the position of a landmark, the range $r$ is

$$r = \sqrt{(p_x - x)^2 + (p_y - y)^2}$$

The sensor provides bearing relative to the orientation of the robot, so we must subtract the robot's orientation from the bearing to get the sensor reading, like so:

$$\phi = \arctan(\frac{p_y - y}{p_x - x}) - \theta$$

Thus our measurement model $h$ is

$$\begin{aligned}\mathbf{z} &= h(\bar{\mathbf{x}}, \mathbf{p}) &&+\mathcal{N}(0, R)\\ &= \begin{bmatrix}\sqrt{(p_x - x)^2 + (p_y - y)^2} \\ \arctan(\frac{p_y - y}{p_x - x}) - \theta\end{bmatrix} &&+\mathcal{N}(0, R)\end{aligned}$$

This is clearly nonlinear, so we need linearize $h$ at $\mathbf{x}$ by taking its Jacobian. We compute that with SymPy below.

```
In [14]: px, py = symbols('p_x, p_y')
         z = Matrix([[sympy.sqrt((px-x)**2 + (py-y)**2)],
                     [sympy.atan2(py-y, px-x) - theta]])
         z.jacobian(Matrix([x, y, theta]))
```

Out[14]:

$$\begin{bmatrix}\frac{-p_x+x}{\sqrt{(p_x-x)^2+(p_y-y)^2}} & \frac{-p_y+y}{\sqrt{(p_x-x)^2+(p_y-y)^2}} & 0 \\ -\frac{-p_y+y}{(p_x-x)^2+(p_y-y)^2} & -\frac{p_x-x}{(p_x-x)^2+(p_y-y)^2} & -1\end{bmatrix}$$

Now we need to write that as a Python function. For example we might write:

```
In [15]: from math import sqrt

         def H_of(x, landmark_pos):
             """ compute Jacobian of H matrix where h(x) computes
             the range and bearing to a landmark for state x """

             px = landmark_pos[0]
             py = landmark_pos[1]
             hyp = (px - x[0, 0])**2 + (py - x[1, 0])**2
             dist = sqrt(hyp)

             H = array(
                 [[-(px - x[0, 0]) / dist, -(py - x[1, 0]) / dist, 0],
                  [ (py - x[1, 0]) / hyp,  -(px - x[0, 0]) / hyp, -1]])
             return H
```

We also need to define a function that converts the system state into a measurement.

```
In [16]: from math import atan2

         def Hx(x, landmark_pos):
             """ takes a state variable and returns the measurement
             that would correspond to that state.
             """
             px = landmark_pos[0]
             py = landmark_pos[1]
             dist = sqrt((px - x[0, 0])**2 + (py - x[1, 0])**2)

             Hx = array([[dist],
                         [atan2(py - x[1, 0], px - x[0, 0]) - x[2, 0]]])
             return Hx
```

### 11.4.5 Design Measurement Noise

It is reasonable to assume that the noise of the range and bearing measurements are independent, hence

$$\mathbf{R} = \begin{bmatrix}\sigma^2_{range} & 0 \\ 0 & \sigma^2_{bearing}\end{bmatrix}$$

### 11.4.6 Implementation

We will use `FilterPy`'s `ExtendedKalmanFilter` class to implement the filter. Its `predict()` method uses the standard linear equations for the process model. Ours is nonlinear, so we will have to override `predict()` with our own implementation. I'll want to also use this class to simulate the robot, so I'll add a method `move()` that computes the position of the robot which both `predict()` and my simulation can call.

The matrices for the prediction step are quite large. While writing this code I made several errors before I finally got it working. I only found my errors by using SymPy's `evalf` function. `evalf` evaluates a SymPy `Matrix` with specific values for the variables. I decided to demonstrate this technique to you, and used `evalf` in the Kalman filter code. You'll need to understand a couple of points.

First, `evalf` uses a dictionary to specify the values. For example, if your matrix contains an `x` and `y`, you can write

```
M.evalf(subs={x:3, y:17})
```

to evaluate the matrix for `x=3` and `y=17`.

Second, `evalf` returns a `sympy.Matrix` object. Use `numpy.array(M).astype(float)` to convert it to a NumPy array. `numpy.array(M)` creates an array of type `object`, which is not what you want.

Here is the code for the EKF:

```
In [17]: from filterpy.kalman import ExtendedKalmanFilter as EKF
         from numpy import array, sqrt
         class RobotEKF(EKF):
             def __init__(self, dt, wheelbase, std_vel, std_steer):
                 EKF.__init__(self, 3, 2, 2)
                 self.dt = dt
                 self.wheelbase = wheelbase
                 self.std_vel = std_vel
                 self.std_steer = std_steer

                 a, x, y, v, w, theta, time = symbols(
                     'a, x, y, v, w, theta, t')
                 d = v*time
                 beta = (d/w)*sympy.tan(a)
                 r = w/sympy.tan(a)

                 self.fxu = Matrix(
                     [[x-r*sympy.sin(theta)+r*sympy.sin(theta+beta)],
                      [y+r*sympy.cos(theta)-r*sympy.cos(theta+beta)],
                      [theta+beta]])

                 self.F_j = self.fxu.jacobian(Matrix([x, y, theta]))
                 self.V_j = self.fxu.jacobian(Matrix([v, a]))

                 # save dictionary and it's variables for later use
                 self.subs = {x: 0, y: 0, v:0, a:0,
                              time:dt, w:wheelbase, theta:0}
                 self.x_x, self.x_y, = x, y
                 self.v, self.a, self.theta = v, a, theta

             def predict(self, u):
                 self.x = self.move(self.x, u, self.dt)

                 self.subs[self.theta] = self.x[2, 0]
                 self.subs[self.v] = u[0]
```

```
            self.subs[self.a] = u[1]

            F = array(self.F_j.evalf(subs=self.subs)).astype(float)
            V = array(self.V_j.evalf(subs=self.subs)).astype(float)

            # covariance of motion noise in control space
            M = array([[self.std_vel*u[0]**2, 0],
                       [0, self.std_steer**2]])

            self.P = np.dot(F, self.P).dot(F.T) + np.dot(V, M).dot(V.T)

        def move(self, x, u, dt):
            hdg = x[2, 0]
            vel = u[0]
            steering_angle = u[1]
            dist = vel * dt

            if abs(steering_angle) > 0.001: # is robot turning?
                beta = (dist / self.wheelbase) * tan(steering_angle)
                r = self.wheelbase / tan(steering_angle) # radius

                dx = np.array([[-r*sin(hdg) + r*sin(hdg + beta)],
                               [r*cos(hdg) - r*cos(hdg + beta)],
                               [beta]])
            else: # moving in straight line
                dx = np.array([[dist*cos(hdg)],
                               [dist*sin(hdg)],
                               [0]])
            return x + dx
```

Now we have another issue to handle. The residual is notionally computed as $y = z - h(x)$ but this will not work because our measurement contains an angle in it. Suppose z has a bearing of 1° and $h(x)$ has a bearing of 359°. Naively subtracting them would yield a angular difference of −358°, whereas the correct value is 2°. We have to write code to correctly compute the bearing residual.

```
In [18]: def residual(a, b):
             """ compute residual (a-b) between measurements containing
             [range, bearing]. Bearing is normalized to [-pi, pi)"""
             y = a - b
             y[1] = y[1] % (2 * np.pi)    # force in range [0, 2 pi)
             if y[1] > np.pi:             # move to [-pi, pi)
                 y[1] -= 2 * np.pi
             return y
```

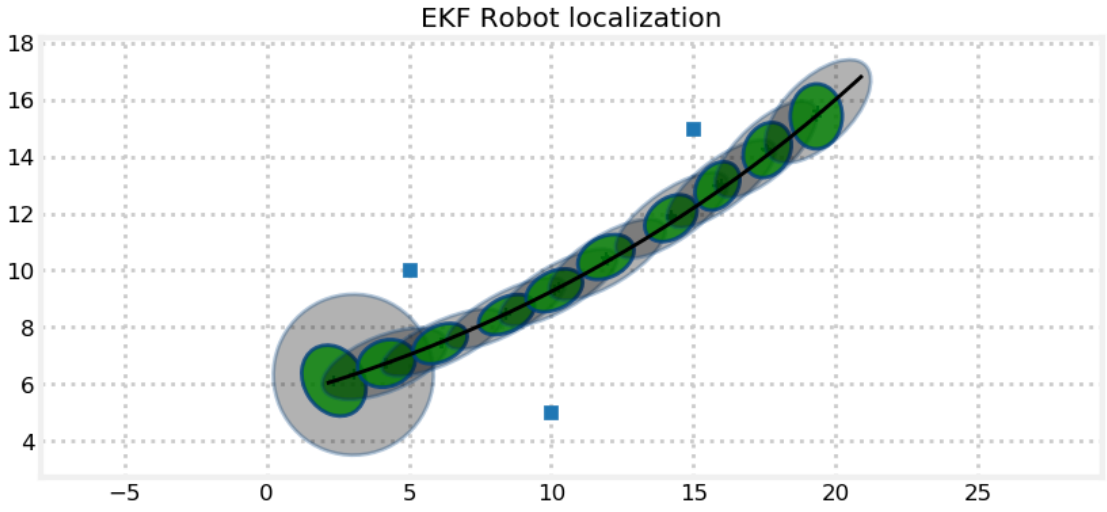
The rest of the code runs the simulation and plots the results, and shouldn't need too much comment by now. I create a variable `landmarks` that contains the landmark coordinates. I update the simulated robot position 10 times a second, but run the EKF only once per second. This is for two reasons. First, we are not using Runge Kutta to integrate the differental equations of motion, so a narrow time step allows our simulation to be more accurate. Second, it is fairly normal in embedded systems to have limited processing speed. This forces you to run your Kalman filter only as frequently as absolutely needed.

```
In [19]: from filterpy.stats import plot_covariance_ellipse
         from math import sqrt, tan, cos, sin, atan2
         import matplotlib.pyplot as plt
```

```
dt = 1.0

def z_landmark(lmark, sim_pos, std_rng, std_brg):
    x, y = sim_pos[0, 0], sim_pos[1, 0]
    d = np.sqrt((lmark[0] - x)**2 + (lmark[1] - y)**2)
    a = atan2(lmark[1] - y, lmark[0] - x) - sim_pos[2, 0]
    z = np.array([[d + randn()*std_rng],
                  [a + randn()*std_brg]])
    return z

def ekf_update(ekf, z, landmark):
    ekf.update(z, HJacobian=H_of, Hx=Hx,
               residual=residual,
               args=(landmark), hx_args=(landmark))


def run_localization(landmarks, std_vel, std_steer,
                     std_range, std_bearing,
                     step=10, ellipse_step=20, ylim=None):
    ekf = RobotEKF(dt, wheelbase=0.5, std_vel=std_vel,
                   std_steer=std_steer)
    ekf.x = array([[2, 6, .3]]).T # x, y, steer angle
    ekf.P = np.diag([.1, .1, .1])
    ekf.R = np.diag([std_range**2, std_bearing**2])

    sim_pos = ekf.x.copy() # simulated position
    # steering command (vel, steering angle radians)
    u = array([1.1, .01])

    plt.figure()
    plt.scatter(landmarks[:, 0], landmarks[:, 1],
                marker='s', s=60)

    track = []
    for i in range(200):
        sim_pos = ekf.move(sim_pos, u, dt/10.) # simulate robot
        track.append(sim_pos)

        if i % step == 0:
            ekf.predict(u=u)

            if i % ellipse_step == 0:
                plot_covariance_ellipse(
                    (ekf.x[0,0], ekf.x[1,0]), ekf.P[0:2, 0:2],
                     std=6, facecolor='k', alpha=0.3)

            x, y = sim_pos[0, 0], sim_pos[1, 0]
            for lmark in landmarks:
                z = z_landmark(lmark, sim_pos,
                               std_range, std_bearing)
                ekf_update(ekf, z, lmark)

            if i % ellipse_step == 0:
                plot_covariance_ellipse(
```

```
                        (ekf.x[0,0], ekf.x[1,0]), ekf.P[0:2, 0:2],
                        std=6, facecolor='g', alpha=0.8)
        track = np.array(track)
        plt.plot(track[:, 0], track[:,1], color='k', lw=2)
        plt.axis('equal')
        plt.title("EKF Robot localization")
        if ylim is not None: plt.ylim(*ylim)
        plt.show()
        return ekf
```

```
In [20]: landmarks = array([[5, 10], [10, 5], [15, 15]])

         ekf = run_localization(
             landmarks, std_vel=0.1, std_steer=np.radians(1),
             std_range=0.3, std_bearing=0.1)
         print('Final P:', ekf.P.diagonal())
```




```
Final P: [0.025 0.041 0.002]
```

I have plotted the landmarks as solid squares. The path of the robot is drawn with a black line. The covariance ellipses for the predict step are light gray, and the covariances of the update are shown in green. To make them visible at this scale I have set the ellipse boundary at $6\sigma$.

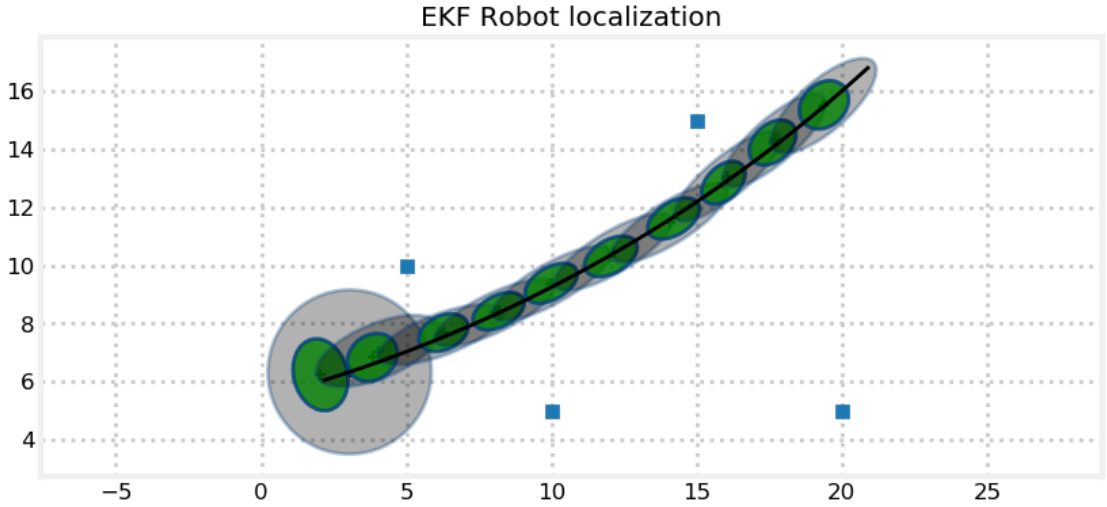
We can see that there is a lot of uncertainty added by our motion model, and that most of the error in in the direction of motion. We determine that from the shape of the blue ellipses. After a few steps we can see that the filter incorporates the landmark measurements and the errors improve.

I used the same initial conditions and landmark locations in the UKF chapter. The UKF achieves much better accuracy in terms of the error ellipse. Both perform roughly as well as far as their estimate for $\mathbf{x}$ is concerned.

Now let's add another landmark.

```
In [21]: landmarks = array([[5, 10], [10, 5], [15, 15], [20, 5]])

         ekf = run_localization(
             landmarks, std_vel=0.1, std_steer=np.radians(1),
```

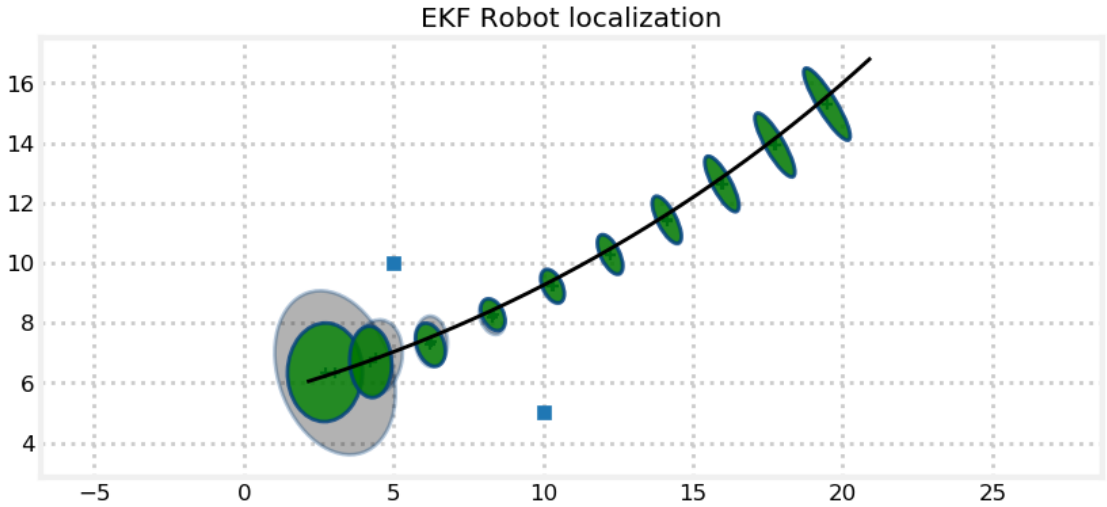
```
        std_range=0.3, std_bearing=0.1)
plt.show()
print('Final P:', ekf.P.diagonal())
```




```
Final P: [0.021 0.02  0.002]
```

The uncertainly in the estimates near the end of the track are smaller. We can see the effect that multiple landmarks have on our uncertainty by only using the first two landmarks.

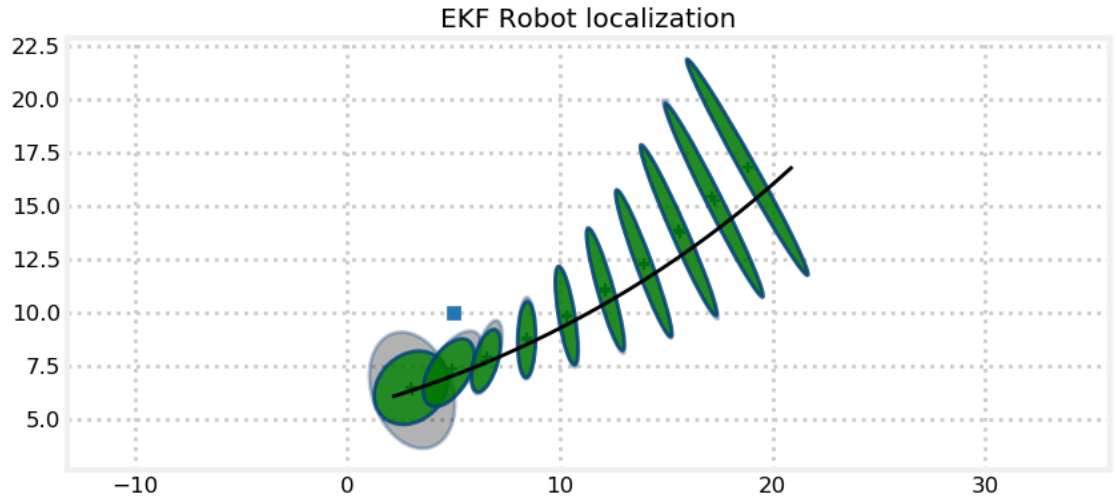
```
In [22]: ekf = run_localization(
             landmarks[0:2], std_vel=1.e-10, std_steer=1.e-10,
             std_range=1.4, std_bearing=.05)
         print('Final P:', ekf.P.diagonal())
```



```
Final P: [0.019 0.045 0.   ]
```

The estimate quickly diverges from the robot's path after passing the landmarks. The covariance also grows quickly. Let's see what happens with only one landmark:

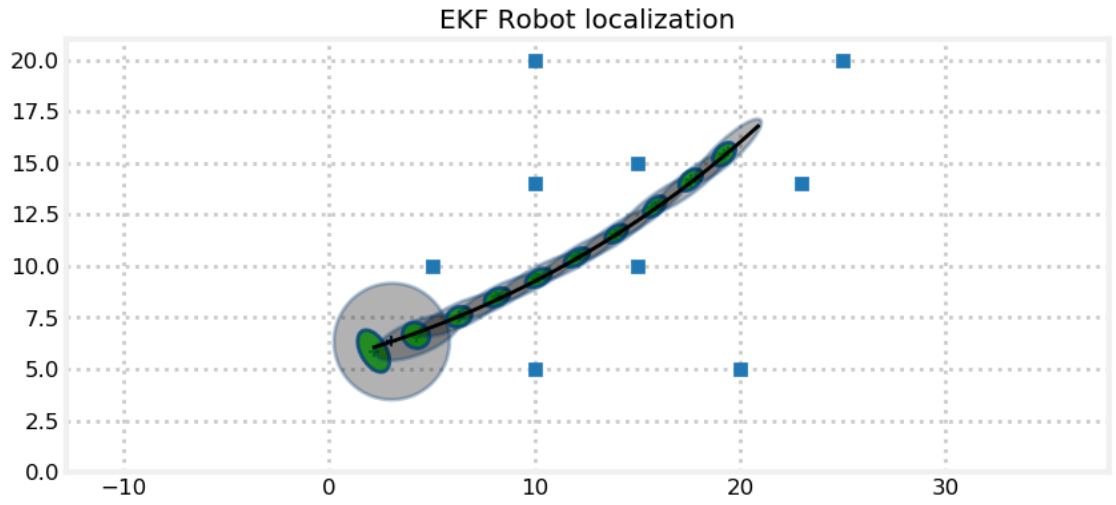
```
In [23]: ekf = run_localization(
             landmarks[0:1], std_vel=1.e-10, std_steer=1.e-10,
             std_range=1.4, std_bearing=.05)
         print('Final P:', ekf.P.diagonal())
```




```
Final P: [0.272 0.787 0.004]
```

As you probably suspected, one landmark produces a very bad result. Conversely, a large number of landmarks allows us to make very accurate estimates.

```
In [24]: landmarks = array([[5, 10], [10,  5], [15, 15], [20,  5], [15, 10],
                            [10,14], [23, 14], [25, 20], [10, 20]])

         ekf = run_localization(
             landmarks, std_vel=0.1, std_steer=np.radians(1),
             std_range=0.3, std_bearing=0.1, ylim=(0, 21))
         print('Final P:', ekf.P.diagonal())
```




```
Final P: [0.009 0.009 0.001]
```

### 11.4.7 Discussion

I said that this was a real problem, and in some ways it is. I've seen alternative presentations that used robot motion models that led to simpler Jacobians. On the other hand, my model of the movement is also simplistic in several ways. First, it uses a bicycle model. A real car has two sets of tires, and each travels on a different radius. The wheels do not grip the surface perfectly. I also assumed that the robot responds instantaneously to the control input. Sebastian Thrun writes in *Probabilistic Robots* that this simplified model is justified because the filters perform well when used to track real vehicles. The lesson here is that while you have to have a reasonably accurate nonlinear model, it does not need to be perfect to operate well. As a designer you will need to balance the fidelity of your model with the difficulty of the math and the CPU time required to perform the linear algebra.

Another way in which this problem was simplistic is that we assumed that we knew the correspondance between the landmarks and measurements. But suppose we are using radar - how would we know that a specific signal return corresponded to a specific building in the local scene? This question hints at SLAM algorithms - simultaneous localization and mapping. SLAM is not the point of this book, so I will not elaborate on this topic.

## 11.5 UKF vs EKF

In the last chapter I used the UKF to solve this problem. The difference in implementation should be very clear. Computing the Jacobians for the state and measurement models was not trivial despite a rudimentary motion model. A different problem could result in a Jacobian which is difficult or impossible to derive analytically. In contrast, the UKF only requires you to provide a function that computes the system motion model and another for the measurement model.

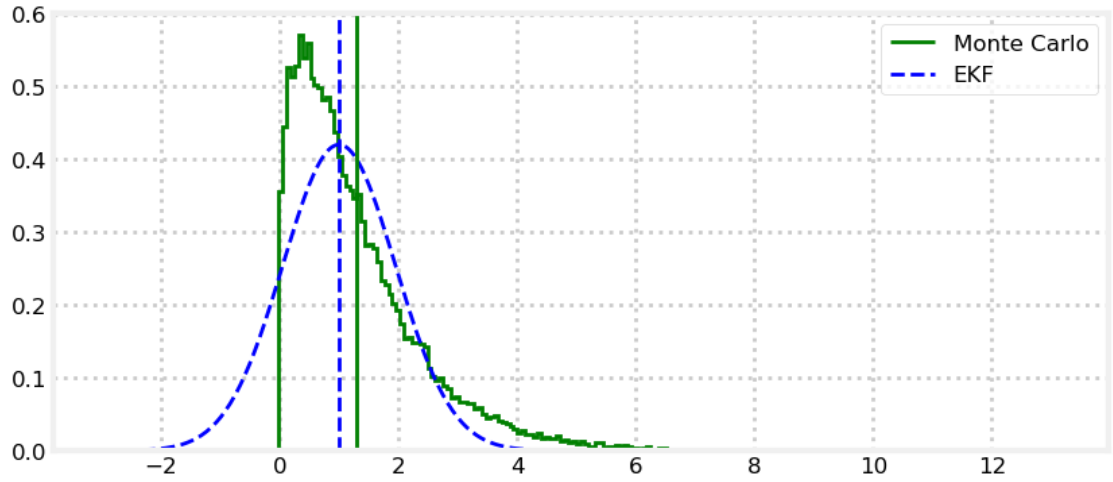
There are many cases where the Jacobian cannot be found analytically. The details are beyond the scope of this book, but you will have to use numerical methods to compute the Jacobian. That undertaking is not trivial, and you will spend a significant portion of a master's degree at a STEM school learning techniques to handle such situations. Even then you'll likely only be able to solve problems related to your field - an aeronautical engineer learns a lot about Navier Stokes equations, but not much about modelling chemical reaction rates.

So, UKFs are easy. Are they accurate? In practice they often perform better than the EKF. You can find plenty of research papers that prove that the UKF outperforms the EKF in various problem domains. It's not hard to understand why this would be true. The EKF works by linearizing the system model and measurement model at a single point, and the UKF uses $2n+1$ points.

Let's look at a specific example. Take $f(x) = x^3$ and pass a Gaussian distribution through it. I will compute an accurate answer using a monte carlo simulation. I generate 50,000 points randomly distributed according to the Gaussian, pass each through $f(x)$, then compute the mean and variance of the result.

The EKF linearizes the function by taking the derivative to find the slope at the evaluation point $x$. This slope becomes the linear function that we use to transform the Gaussian. Here is a plot of that.

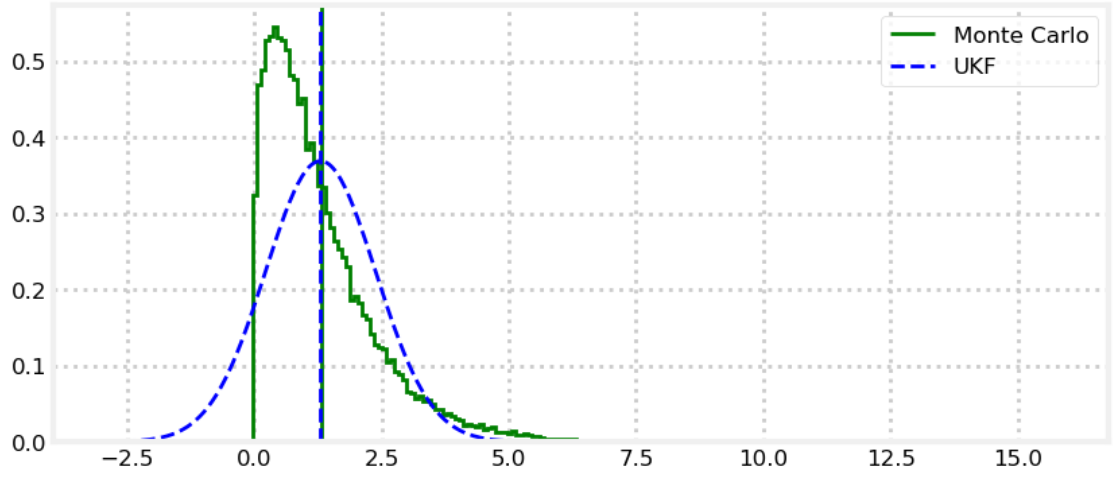
```
In [25]: import kf_book.nonlinear_plots as nonlinear_plots
         nonlinear_plots.plot_ekf_vs_mc()
```




```
actual mean=1.30, std=1.14
EKF    mean=1.00, std=0.95
```

The EKF computation is rather inaccurate. In contrast, here is the performance of the UKF:

```
In [26]: nonlinear_plots.plot_ukf_vs_mc(alpha=0.001, beta=3., kappa=1.)
```



```
actual mean=1.31, std=1.13
UKF    mean=1.30, std=1.08
```

Here we can see that the computation of the UKF's mean is accurate to 2 decimal places. The standard deviation is slightly off, but you can also fine tune how the UKF computes the distribution by using the $\alpha$, $\beta$, and $\gamma$ parameters for generating the sigma points. Here I used $\alpha = 0.001$, $\beta = 3$, and $\gamma = 1$. Feel free to modify them to see the result. You should be able to get better results than I did. However, avoid over-tuning the UKF for a specific test. It may perform better for your test case, but worse in general.

# Chapter 12

# Particle Filters

```
In [1]: from __future__ import division, print_function
        %matplotlib inline
```

```
In [2]: #format the book
        import book_format
        book_format.set_style()
```

```
Out[2]:
```

## 12.1 Motivation

Here is our problem. We have moving objects that we want to track. Maybe the objects are fighter jets and missiles, or maybe we are tracking people playing cricket in a field. It doesn't really matter. Which of the filters that we have learned can handle this problem? Unfortunately, none of them are ideal. Let's think about the characteristics of this problem.

- **multimodal**: We want to track zero, one, or more than one object simultaneously.
- **occlusions**: One object can hide another, resulting in one measurement for multiple objects.
- **nonlinear behavior**: Aircraft are buffeted by winds, balls move in parabolas, and people collide into each other.
- **nonlinear measurements**: Radar gives us the distance to an object. Converting that to an (x,y,z) coordinate requires a square root, which is nonlinear.
- **non-Gaussian noise:** as objects move across a background the computer vision can mistake part of the background for the object.
- **continuous:** the object's position and velocity (i.e. the state space) can smoothly vary over time.
- **multivariate**: we want to track several attributes, such as position, velocity, turn rates, etc.
- **unknown process model**: we may not know the process model of the system

None of the filters we have learned work well with all of these constraints.

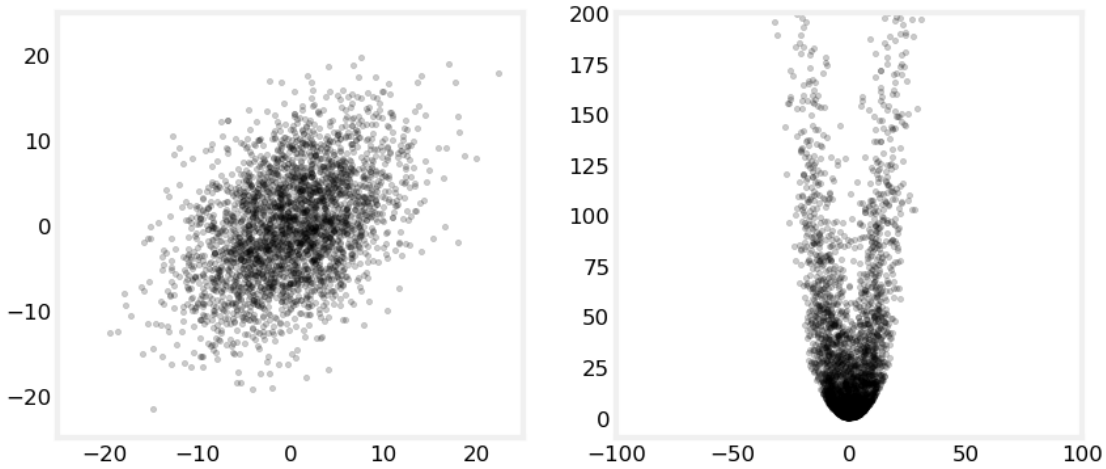
- **Discrete Bayes filter**: This has most of the attributes. It is multimodal, can handle nonlinear measurements, and can be extended to work with nonlinear behavior. However, it is discrete and univariate.
- **Kalman filter**: The Kalman filter produces optimal estimates for unimodal linear systems with Gaussian noise. None of these are true for our problem.

- **Unscented Kalman filter**: The UKF handles nonlinear, continuous, multivariate problems. However, it is not multimodal nor does it handle occlusions. It can handle noise that is modestly non-Gaussian, but does not do well with distributions that are very non-Gaussian or problems that are very nonlinear.
- **Extended Kalman filter**: The EKF has the same strengths and limitations as the UKF, except that is it even more sensitive to strong nonlinearities and non-Gaussian noise.

## 12.2 Monte Carlo Sampling

In the UKF chapter I generated a plot similar to this to illustrate the effects of nonlinear systems on Gaussians:

```
In [3]: import kf_book.pf_internal as pf_internal
        pf_internal.plot_monte_carlo_ukf()
```




The left plot shows 3,000 points normally distributed based on the Gaussian

$$\mu = \begin{bmatrix}0\\0\end{bmatrix},\ \Sigma = \begin{bmatrix}32 & 15\\15 & 40\end{bmatrix}$$

The right plots shows these points passed through this set of equations:

$$x = x + y$$
$$y = 0.1x^2 + y^2$$

Using a finite number of randomly sampled points to compute a result is called a *Monte Carlo* (MC) method. The idea is simple. Generate enough points to get a representative sample of the problem, run the points through the system you are modeling, and then compute the results on the transformed points.

In a nutshell this is what particle filtering does. The Bayesian filter algorithm we have been using throughout the book is applied to thousands of particles, where each particle represents a *possible* state for the system. We extract the estimated state from the thousands of particles using weighted statistics of the particles.

## 12.3 Generic Particle Filter Algorithm

1. **Randomly generate a bunch of particles**

Particles can have position, heading, and/or whatever other state variable you need to estimate. Each has a weight (probability) indicating how likely it matches the actual state of the system. Initialize each with the same weight.

2. **Predict next state of the particles**

Move the particles based on how you predict the real system is behaving.

3. **Update**

Update the weighting of the particles based on the measurement. Particles that closely match the measurements are weighted higher than particles which don't match the measurements very well.

4. **Resample**

Discard highly improbable particle and replace them with copies of the more probable particles.

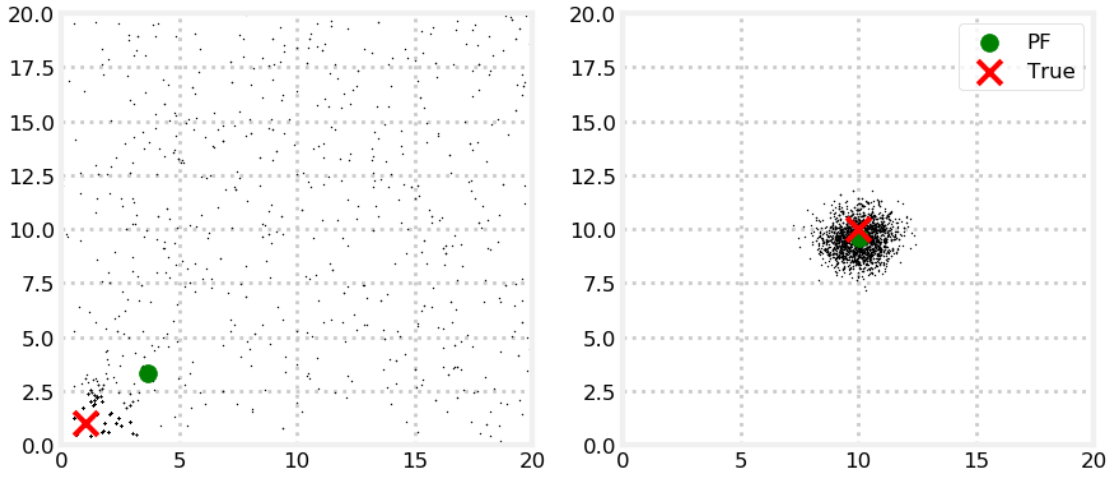
5. **Compute Estimate**

Optionally, compute weighted mean and covariance of the set of particles to get a state estimate.

This naive algorithm has practical difficulties which we will need to overcome, but this is the general idea. Let's see an example. I wrote a particle filter for the robot localization problem from the UKF and EKF chapters. The robot has steering and velocity control inputs. It has sensors that measures distance to visible landmarks. Both the sensors and control mechanism have noise in them, and we need to track the robot's position.

Here I run a particle filter and plotted the positions of the particles. The plot on the left is after one iteration, and on the right is after 10. The red 'X' shows the actual position of the robot, and the large circle is the computed weighted mean position.

```
In [4]: pf_internal.show_two_pf_plots()
```




If you are viewing this in a browser, this animation shows the entire sequence:

After the first iteration the particles are still largely randomly scattered around the map, but you can see that some have already collected near the robot's position. The computed mean is quite close to the robot's

position. This is because each particle is weighted based on how closely it matches the measurement. The robot is near (1,1), so particles that are near (1, 1) will have a high weight because they closely match the measurements. Particles that are far from the robot will not match the measurements, and thus have a very low weight. The estimated position is computed as the weighted mean of positions of the particles. Particles near the robot contribute more to the computation so the estimate is quite accurate.

Several iterations later you can see that all the particles have clustered around the robot. This is due to the *resampling* step. Resampling discards particles that are very improbable (very low weight) and replaces them with particles with higher probability.

I haven't fully shown *why* this works nor fully explained the algorithms for particle weighting and resampling, but it should make intuitive sense. Make a bunch of random particles, move them so they 'kind of' follow the robot, weight them according to how well they match the measurements, only let the likely ones live. It seems like it should work, and it does.

## 12.4 Probability distributions via Monte Carlo

Suppose we want to know the area under the curve $y = e^{\sin(x)}$ in the interval $[0, \pi]$. The area is computed with the definite integral $\int_0^\pi e^{\sin(x)}\,dx$. As an exercise, go ahead and find the answer; I'll wait.

If you are wise you did not take that challenge; $e^{\sin(x)}$ cannot be integrated analytically. The world is filled with equations which we cannot integrate. For example, consider calculating the luminosity of an object. An object reflects some of the light that strike it. Some of the reflected light bounces off of other objects and restrikes the original object, increasing the luminosity. This creates a *recursive integral*. Good luck with that one.

However, integrals are trivial to compute using a Monte Carlo technique. To find the area under a curve create a bounding box that contains the curve in the desired interval. Generate randomly positioned point within the box, and compute the ratio of points that fall under the curve vs the total number of points. For example, if 40% of the points are under the curve and the area of the bounding box is 1, then the area under the curve is approximately 0.4. As you tend towards infinite points you can achieve any arbitrary precision. In practice, a few thousand points will give you a fairly accurate result.

You can use this technique to numerically integrate a function of any arbitrary difficulty. this includes non-integrable and noncontinuous functions. This technique was invented by Stanley Ulam at Los Alamos National Laboratory to allow him to perform computations for nuclear reactions which were unsolvable on paper.

Let's compute $\pi$ by finding the area of a circle. We will define a circle with a radius of 1, and bound it in a square. The side of the square has length 2, so the area is 4. We generate a set of uniformly distributed random points within the box, and count how many fall inside the circle. The area of the circle is computed as the area of the box times the ratio of points inside the circle vs. the total number of points. Finally, we know that $A = \pi r^2$, so we compute $\pi = A/r^2$.

We start by creating the points.

```
N = 20000
pts = uniform(-1, 1, (N, 2))
```

A point is inside a circle if its distance from the center of the circle is less than or equal to the radius. We compute the distance with `numpy.linalg.norm`, which computes the magnitude of a vector. Since vectors start at (0, 0) calling norm will compute the point's distance from the origin.

```
dist = np.linalg.norm(pts, axis=1)
```

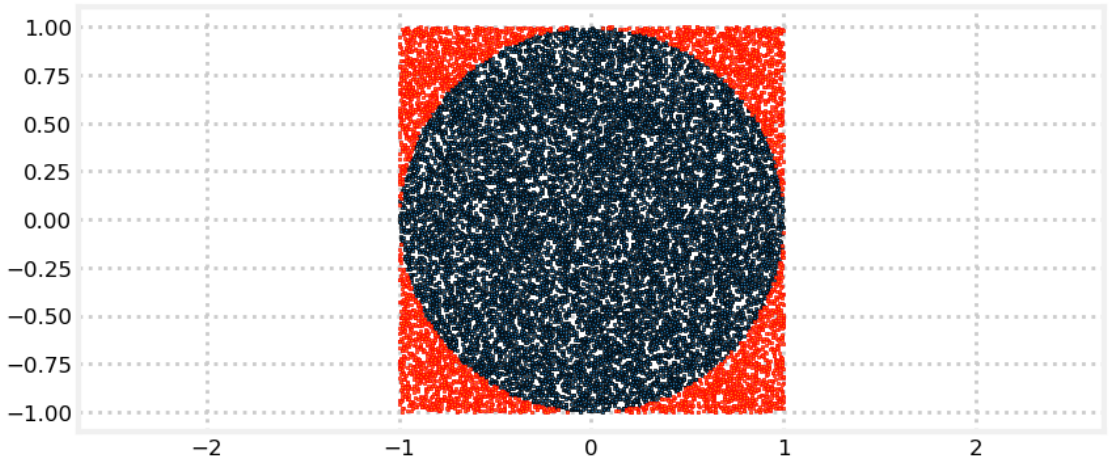
Next we compute which of this distances fit the criteria. This code returns a bool array that contains `True` if it meets the condition `dist <= 1`:

```
in_circle = dist <= 1
```

All that is left is to count the points inside the circle, compute pi, and plot the results. I've put it all in one cell so you can experiment with alternative values for `N`, the number of points.

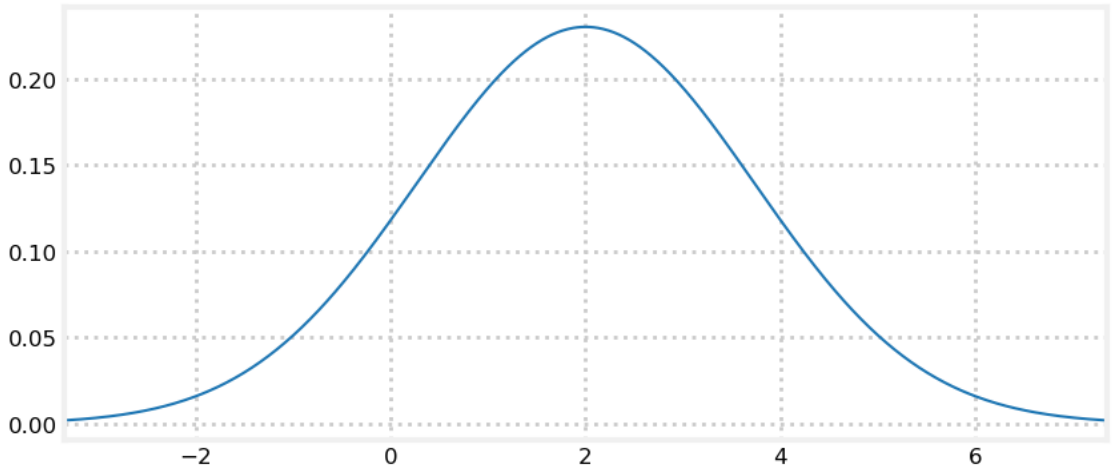
```
In [5]: import matplotlib.pyplot as plt
        import numpy as np
        from numpy.random import uniform

        N = 20000  # number of points
        radius = 1.
        area = (2*radius)**2

        pts = uniform(-1, 1, (N, 2))

        # distance from (0,0)
        dist = np.linalg.norm(pts, axis=1)
        in_circle = dist <= 1

        pts_in_circle = np.count_nonzero(in_circle)
        pi = 4 * (pts_in_circle / N)

        # plot results
        plt.scatter(pts[in_circle,0], pts[in_circle,1],
                    marker=',', edgecolor='k', s=1)
        plt.scatter(pts[~in_circle,0], pts[~in_circle,1],
                    marker=',', edgecolor='r', s=1)
        plt.axis('equal')

        print('mean pi(N={})= {:.4f}'.format(N, pi))
        print('err  pi(N={})= {:.4f}'.format(N, np.pi-pi))
```

```
mean pi(N=20000)= 3.1684
err  pi(N=20000)= -0.0268
```




This insight leads us to the realization that we can use Monte Carlo to compute the probability density of any probability distribution. For example, suppose we have this Gaussian:

```
In [6]: from filterpy.stats import plot_gaussian_pdf
        plot_gaussian_pdf(mean=2, variance=3);
```

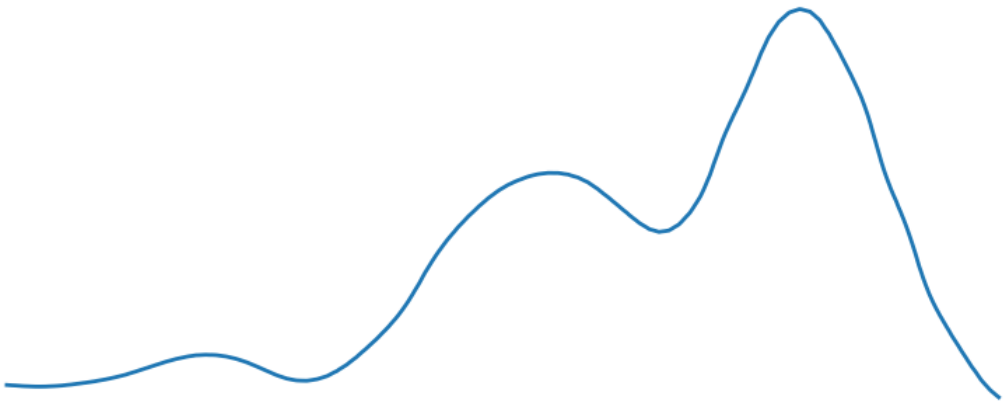



The probability density function (PDF) gives the probability that the random value falls between 2 values. For example, we may want to know the probability of x being between 0 and 2 in the graph above. This is a continuous function, so we need to take the integral to find the area under the curve, as the area is equal to the probability for that range of values to occur.

$$P[a \leq X \leq b] = \int_a^b f_X(x)\, dx$$

It is easy to compute this integral for a Gaussian. But real life is not so easy. For example, the plot below shows a probability distribution. There is no way to analytically describe an arbitrary curve, let alone integrate it.

```
In [7]: pf_internal.plot_random_pd()
```



We can use Monte Carlo methods to compute any integral. The PDF is computed with an integral, hence we can compute the PDF of this curve using Monte Carlo.

## 12.5 The Particle Filter

All of this brings us to the particle filter. Consider tracking a robot or a car in an urban environment. For consistency I will use the robot localization problem from the EKF and UKF chapters. In this problem we tracked a robot that has a sensor which measures the range and bearing to known landmarks.

Particle filters are a family of algorithms. I'm presenting a specific form of a particle filter that is intuitive to grasp and relates to the problems we have studied in this book. This will leave a few of the steps seeming a bit 'magical' since I haven't offered a full explanation. That will follow later in the chapter.

Taking insight from the discussion in the previous section we start by creating several thousand *particles*. Each particle has a position that represents a possible belief of where the robot is in the scene, and perhaps a heading and velocity. Suppose that we have no knowledge of the location of the robot. We would want to scatter the particles uniformly over the entire scene. If you think of all of the particles representing a probability distribution, locations where there are more particles represent a higher belief, and locations with fewer particles represents a lower belief. If there was a large clump of particles near a specific location that would imply that we were more certain that the robot is there.

Each particle needs a weight - ideally the probability that it represents the true position of the robot. This probability is rarely computable, so we only require it be *proportional* to that probability, which is computable. At initialization we have no reason to favor one particle over another, so we assign a weight of $1/N$, for $N$ particles. We use $1/N$ so that the sum of all probabilities equals one.

The combination of particles and weights forms the *probability distribution* for our problem. Think back to the *Discrete Bayes* chapter. In that chapter we modeled positions in a hallway as discrete and uniformly spaced. This is very similar except the particles are randomly distributed in a continuous space rather than constrained to discrete locations. In this problem the robot can move on a plane of some arbitrary dimension, with the lower right corner at (0,0).

To track our robot we need to maintain states for x, y, and heading. We will store `N` particles in a `(N, 3)` shaped array. The three columns contain x, y, and heading, in that order.

If you are passively tracking something (no control input), then you would need to include velocity in the state and use that estimate to make the prediction. More dimensions requires exponentially more particles to form a good estimate, so we always try to minimize the number of random variables in the state.

This code creates a uniform and Gaussian distribution of particles over a region:

```
In [8]: from numpy.random import uniform

        def create_uniform_particles(x_range, y_range, hdg_range, N):
            particles = np.empty((N, 3))
            particles[:, 0] = uniform(x_range[0], x_range[1], size=N)
            particles[:, 1] = uniform(y_range[0], y_range[1], size=N)
            particles[:, 2] = uniform(hdg_range[0], hdg_range[1], size=N)
            particles[:, 2] %= 2 * np.pi
            return particles

        def create_gaussian_particles(mean, std, N):
            particles = np.empty((N, 3))
            particles[:, 0] = mean[0] + (randn(N) * std[0])
            particles[:, 1] = mean[1] + (randn(N) * std[1])
            particles[:, 2] = mean[2] + (randn(N) * std[2])
            particles[:, 2] %= 2 * np.pi
            return particles
```

For example:

```
In [9]: create_uniform_particles((0,1), (0,1), (0, np.pi*2), 4)

Out[9]: array([[0.772, 0.336, 4.171],
               [0.333, 0.34 , 4.319],
```

```
[0.6  , 0.274, 5.02 ],
[0.054, 0.022, 5.034]])
```

### 12.5.1 Predict Step

The predict step in the Bayes algorithm uses the process model to update the belief in the system state. How would we do that with particles? Each particle represents a possible position for the robot. Suppose we send a command to the robot to move 0.1 meters while turning by 0.007 radians. We could move each particle by this amount. If we did that we would soon run into a problem. The robot's controls are not perfect so it will not move exactly as commanded. Therefore we need to add noise to the particle's movements to have a reasonable chance of capturing the actual movement of the robot. If you do not model the uncertainty in the system the particle filter will not correctly model the probability distribution of our belief in the robot's position.

```
In [10]: def predict(particles, u, std, dt=1.):
             """ move according to control input u (heading change, velocity)
             with noise Q (std heading change, std velocity)`"""

             N = len(particles)
             # update heading
             particles[:, 2] += u[0] + (randn(N) * std[0])
             particles[:, 2] %= 2 * np.pi

             # move in the (noisy) commanded direction
             dist = (u[1] * dt) + (randn(N) * std[1])
             particles[:, 0] += np.cos(particles[:, 2]) * dist
             particles[:, 1] += np.sin(particles[:, 2]) * dist
```

### 12.5.2 Update Step

Next we get a set of measurements - one for each landmark currently in view. How should these measurements be used to alter our probability distribution as modeled by the particles?

Think back to the **Discrete Bayes** chapter. In that chapter we modeled positions in a hallway as discrete and uniformly spaced. We assigned a probability to each position which we called the *prior*. When a new measurement came in we multiplied the current probability of that position (the *prior*) by the *likelihood* that the measurement matched that location:

```
def update(likelihood, prior):
    posterior = prior * likelihood
    return normalize(posterior)
```

which is an implementation of the equation

$$x = \|\mathcal{L}\bar{x}\|$$

which is a realization of Bayes theorem:

$$\begin{aligned} P(x \mid z) &= \frac{P(z \mid x)\, P(x)}{P(z)} \\ &= \frac{\texttt{likelihood} \times \texttt{prior}}{\texttt{normalization}} \end{aligned}$$

We do the same with our particles. Each particle has a position and a weight which estimates how well it matches the measurement. Normalizing the weights so they sum to one turns them into a probability distribution. The particles those that are closest to the robot will generally have a higher weight than ones far from the robot.

```
In [11]: def update(particles, weights, z, R, landmarks):
             for i, landmark in enumerate(landmarks):
                 distance = np.linalg.norm(particles[:, 0:2] - landmark, axis=1)
                 weights *= scipy.stats.norm(distance, R).pdf(z[i])

             weights += 1.e-300      # avoid round-off to zero
             weights /= sum(weights) # normalize
```

In the literature this part of the algorithm is called *Sequential Importance Sampling*, or SIS. The equation for the weights is called the *importance density*. I will give these theoretical underpinnings in a following section. For now I hope that this makes intuitive sense. If we weight the particles according to how well they match the measurements they are probably a good sample for the probability distribution of the system after incorporating the measurements. Theory proves this is so. The weights are the *likelihood* in Bayes theorem. Different problems will need to tackle this step in slightly different ways but this is the general idea.

### 12.5.3 Computing the State Estimate

In most applications you will want to know the estimated state after each update, but the filter consists of nothing but a collection of particles. Assuming that we are tracking one object (i.e. it is unimodal) we can compute the mean of the estimate as the sum of the weighted values of the particles.

$$\mu = \frac{1}{N}\sum_{i=1}^{N} w^i x^i$$

Here I adopt the notation $x^i$ to indicate the $\mathtt{i}^{th}$ particle. A superscript is used because we often need to use subscripts to denote time steps the $\mathtt{k}^{th}$ or $\mathtt{k+1}^{th}$ particle, yielding the unwieldy $x^i_{k+1}$.

This function computes both the mean and variance of the particles:

```
In [12]: def estimate(particles, weights):
             """returns mean and variance of the weighted particles"""

             pos = particles[:, 0:2]
             mean = np.average(pos, weights=weights, axis=0)
             var  = np.average((pos - mean)**2, weights=weights, axis=0)
             return mean, var
```

If we create a uniform distribution of points in a 1x1 square with equal weights we get a mean position very near the center of the square at (0.5, 0.5) and a small variance.

```
In [13]: particles = create_uniform_particles((0,1), (0,1), (0, 5), 1000)
         weights = np.array([.25]*1000)
         estimate(particles, weights)
```

```
Out[13]: (array([0.494, 0.514]), array([0.083, 0.085]))
```

### 12.5.4 Particle Resampling

The SIS algorithm suffers from the *degeneracy problem*. It starts with uniformly distributed particles with equal weights. There may only be a handful of particles near the robot. As the algorithm runs any particle that does not match the measurements will acquire an extremely low weight. Only the particles which are near the robot will have an appreciable weight. We could have 5,000 particles with only 3 contributing meaningfully to the state estimate! We say the filter has *degenerated*.This problem is usually solved by some form of *resampling* of the particles.

Particles with very small weights do not meaningfully describe the probability distribution of the robot. The resampling algorithm discards particles with very low probability and replaces them with new particles

with higher probability. It does that by duplicating particles with relatively high probability. The duplicates are slightly dispersed by the noise added in the predict step. This results in a set of points in which a large majority of the particles accurately represent the probability distribution.

There are many resampling algorithms. For now let's look at one of the simplest, *simple random resampling*, also called *multinomial resampling*. It samples from the current particle set $N$ times, making a new set of particles from the sample. The probability of selecting any given particle should be proportional to its weight.

We accomplish this with NumPy's `cumsum` function. `cumsum` computes the cumulative sum of an array. That is, element one is the sum of elements zero and one, element two is the sum of elements zero, one and two, etc. Then we generate random numbers in the range of 0.0 to 1.0 and do a binary search to find the weight that most closely matches that number:

```
In [14]: def simple_resample(particles, weights):
             N = len(particles)
             cumulative_sum = np.cumsum(weights)
             cumulative_sum[-1] = 1. # avoid round-off error
             indexes = np.searchsorted(cumulative_sum, random(N))

             # resample according to indexes
             particles[:] = particles[indexes]
             weights.fill(1.0 / N)
```

We don't resample at every epoch. For example, if you received no new measurements you have not received any information from which the resample can benefit. We can determine when to resample by using something called the *effective* $N$, which approximately measures the number of particles which meaningfully contribute to the probability distribution. The equation for this is

$$\hat{N}_{\text{eff}} = \frac{1}{\sum w^2}$$

and we can implement this in Python with

```
In [15]: def neff(weights):
             return 1. / np.sum(np.square(weights))
```

If $\hat{N}_{\text{eff}}$ falls below some threshold it is time to resample. A useful starting point is $N/2$, but this varies by problem. It is also possible for $\hat{N}_{\text{eff}} = N$, which means the particle set has collapsed to one point (each has equal weight). It may not be theoretically pure, but if that happens I create a new distribution of particles in the hopes of generating particles with more diversity. If this happens to you often, you may need to increase the number of particles, or otherwise adjust your filter. We will talk more of this later.

## 12.6 SIR Filter - A Complete Example

There is more to learn, but we know enough to implement a full particle filter. We will implement the *Sampling Importance Resampling filter*, or SIR.

I need to introduce a more sophisticated resampling method than I gave above. FilterPy provides several resampling methods. I will describe them later. They take an array of weights and returns indexes to the particles that have been chosen for the resampling. We just need to write a function that performs the resampling from these indexes:

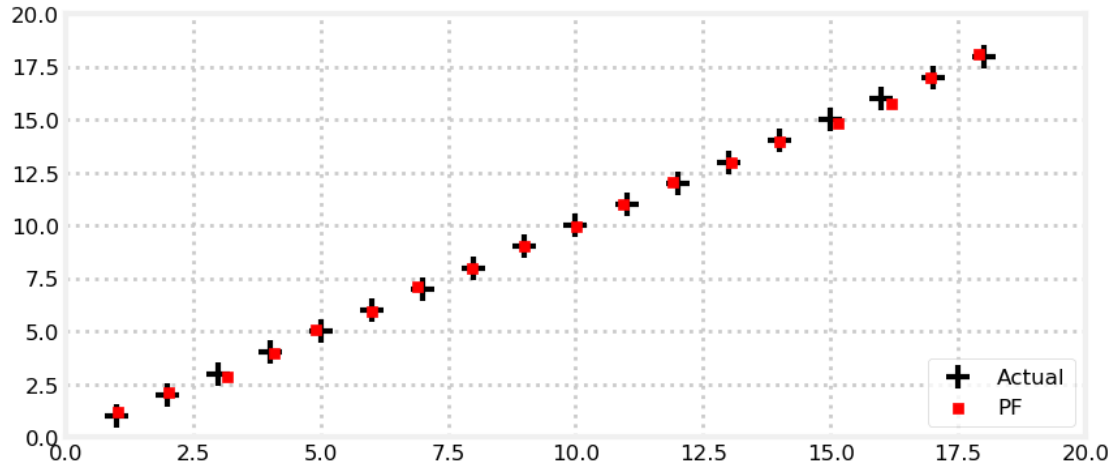
```
In [16]: def resample_from_index(particles, weights, indexes):
             particles[:] = particles[indexes]
             weights.resize(len(particles))
             weights.fill (1.0 / len(weights))
```

To implement the filter we need to create the particles and the landmarks. We then execute a loop, successively calling `predict`, `update`, resampling, and then computing the new state estimate with `estimate`.

```
In [17]: from filterpy.monte_carlo import systematic_resample
         from numpy.linalg import norm
         from numpy.random import randn
         import scipy.stats

         def run_pf1(N, iters=18, sensor_std_err=.1,
                     do_plot=True, plot_particles=False,
                     xlim=(0, 20), ylim=(0, 20),
                     initial_x=None):
             landmarks = np.array([[-1, 2], [5, 10], [12,14], [18,21]])
             NL = len(landmarks)

             plt.figure()

             # create particles and weights
             if initial_x is not None:
                 particles = create_gaussian_particles(
                     mean=initial_x, std=(5, 5, np.pi/4), N=N)
             else:
                 particles = create_uniform_particles((0,20), (0,20), (0, 6.28), N)
             weights = np.ones(N) / N

             if plot_particles:
                 alpha = .20
                 if N > 5000:
                     alpha *= np.sqrt(5000)/np.sqrt(N)
                 plt.scatter(particles[:, 0], particles[:, 1],
                             alpha=alpha, color='g')

             xs = []
             robot_pos = np.array([0., 0.])
             for x in range(iters):
                 robot_pos += (1, 1)

                 # distance from robot to each landmark
                 zs = (norm(landmarks - robot_pos, axis=1) +
                       (randn(NL) * sensor_std_err))

                 # move diagonally forward to (x+1, x+1)
                 predict(particles, u=(0.00, 1.414), std=(.2, .05))

                 # incorporate measurements
                 update(particles, weights, z=zs, R=sensor_std_err,
                        landmarks=landmarks)

                 # resample if too few effective particles
                 if neff(weights) < N/2:
                     indexes = systematic_resample(weights)
                     resample_from_index(particles, weights, indexes)
                     assert np.allclose(weights, 1/N)
                 mu, var = estimate(particles, weights)
```

```
            xs.append(mu)

            if plot_particles:
                plt.scatter(particles[:, 0], particles[:, 1],
                            color='k', marker=',', s=1)
            p1 = plt.scatter(robot_pos[0], robot_pos[1], marker='+',
                             color='k', s=180, lw=3)
            p2 = plt.scatter(mu[0], mu[1], marker='s', color='r')

        xs = np.array(xs)
        #plt.plot(xs[:, 0], xs[:, 1])
        plt.legend([p1, p2], ['Actual', 'PF'], loc=4, numpoints=1)
        plt.xlim(*xlim)
        plt.ylim(*ylim)
        print('final position error, variance:\n\t', mu - np.array([iters, iters]), var)
        plt.show()

    from numpy.random import seed
    seed(2)
    run_pf1(N=5000, plot_particles=False)
```

```
final position error, variance:
    [-0.106  0.106] [0.009 0.008]
```




Most of this code is devoted to initialization and plotting. The entirety of the particle filter processing consists of these lines:

```
# move diagonally forward to (x+1, x+1)
predict(particles, u=(0.00, 1.414), std=(.2, .05))

 # incorporate measurements
update(particles, weights, z=zs, R=sensor_std_err,
       landmarks=landmarks)

# resample if too few effective particles
if neff(weights) < N/2:
```

```
    indexes = systematic_resample(weights)
    resample_from_index(particles, weights, indexes)

mu, var = estimate(particles, weights)
```

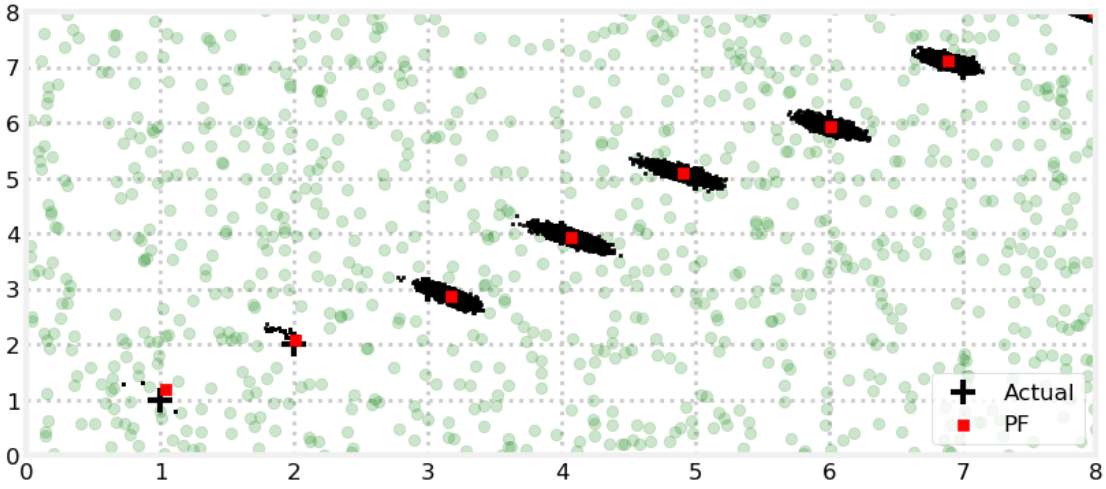
The first line predicts the position of the particles with the assumption that the robot is moving in a straight line (`u[0] == 0`) and moving 1 unit in both the x and y axis (`u[1]==1.414`). The standard deviation for the error in the turn is 0.2, and the standard deviation for the distance is 0.05. When this call returns the particles will all have been moved forward, but the weights are no longer correct as they have not been updated.

The next line incorporates the measurement into the filter. This does not alter the particle positions, it only alters the weights. If you recall the weight of the particle is computed as the probability that it matches the Gaussian of the sensor error model. The further the particle from the measured distance the less likely it is to be a good representation.

The final two lines example the effective particle count ($\hat{N}_{\text{eff}}$). If it falls below $N/2$ we perform resampling to try to ensure our particles form a good representation of the actual probability distribution.

Now let's look at this with all the particles plotted. Seeing this happen interactively is much more instructive, but this format still gives us useful information. I plotted the original random distribution of points in a very pale green and large circles to help distinguish them from the subsequent iterations where the particles are plotted with black pixels. The number of particles makes it hard to see the details, so I limited the number of iterations to 8 so we can zoom in and look more closely.

```
In [18]: seed(2)
         run_pf1(N=5000, iters=8, plot_particles=True,
                 xlim=(0,8), ylim=(0,8))

final position error, variance:
         [-0.019 -0.005] [0.005 0.006]
```




From the plot it looks like there are only a few particles at the first two robot positions. This is not true; there are 5,000 particles, but due to resampling most are duplicates of each other. The reason for this is the Gaussian for the sensor is very narrow. This is called *sample impoverishment* and can lead to filter divergence. I'll address this in detail below. For now, looking at the second step at x=2 we can see that the particles have dispersed a bit. This dispersion is due to the motion model noise. All particles are projected forward according to the control input `u`, but noise is added to each particle proportional to the error in the

control mechanism in the robot. By the third step the particles have dispersed enough to make a convincing cloud of particles around the robot.

The shape of the particle cloud is an ellipse. This is not a coincidence. The sensors and robot control are both modeled as Gaussian, so the probability distribution of the system is also a Gaussian. The particle filter is a sampling of the probability distribution, so the cloud should be an ellipse.

It is important to recognize that the particle filter algorithm *does not require* the sensors or system to be Gaussian or linear. Because we represent the probability distribution with a cloud of particles we can handle any probability distribution and strongly nonlinear problems. There can be discontinuities and hard limits in the probability model.

### 12.6.1 Effect of Sensor Errors on the Filter

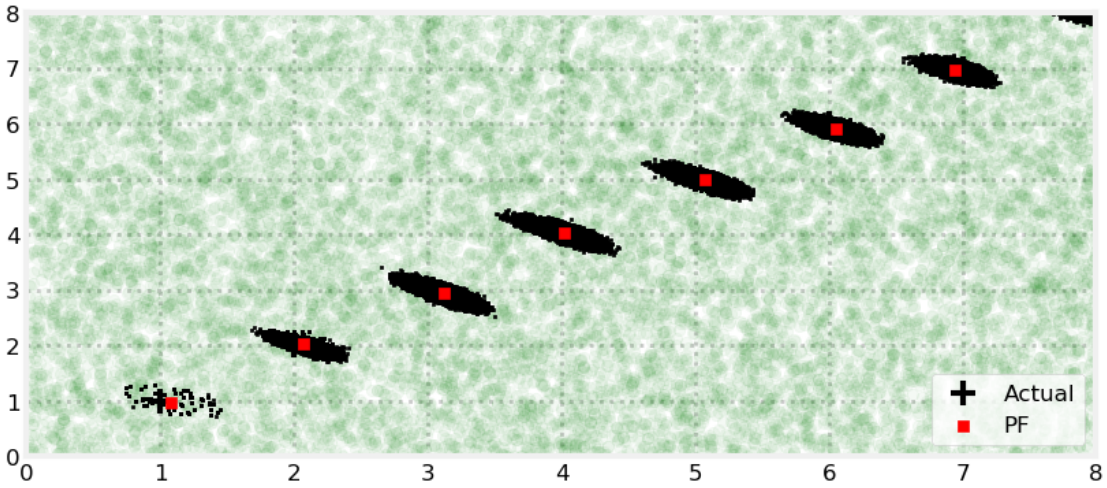
The first few iterations of the filter resulted in many duplicate particles. This happens because the model for the sensors is Gaussian, and we gave it a small standard deviation of $\sigma = 0.1$. This is counterintuitive at first. The Kalman filter performs better when the noise is smaller, yet the particle filter can perform worse.

We can reason about why this is true. If $\sigma = 0.1$, the robot is at (1, 1) and a particle is at (2, 2) the particle is 14 standard deviations away from the robot. This gives it a near zero probability. It contributes nothing to the estimate of the mean, and it is extremely unlikely to survive after the resampling. If $\sigma = 1.4$ then the particle is only $1\sigma$ away and thus it will contribute to the estimate of the mean. During resampling it is likely to be copied one or more times.

This is *very important* to understand - a very accurate sensor can lead to poor performance of the filter because few of the particles will be a good sample of the probability distribution. There are a few fixes available to us. First, we can artificially increase the sensor noise standard deviation so the particle filter will accept more points as matching the robots probability distribution. This is non-optimal because some of those points will be a poor match. The real problem is that there aren't enough points being generated such that enough are near the robot. Increasing `N` usually fixes this problem. This decision is not cost free as increasing the number of particles significantly increase the computation time. Still, let's look at the result of using 100,000 particles.

```
In [19]: seed(2)
         run_pf1(N=100000, iters=8, plot_particles=True,
                 xlim=(0,8), ylim=(0,8))
```

```
final position error, variance:
         [-0.17   0.084] [0.005 0.005]
```



There are many more particles at x=1, and we have a convincing cloud at x=2. Clearly the filter is performing better, but at the cost of large memory usage and long run times.

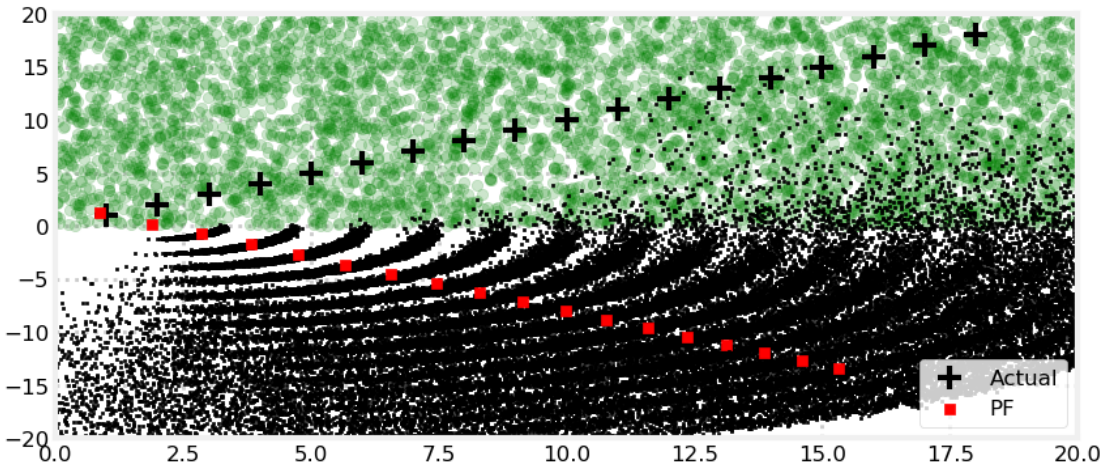
Another approach is to be smarter about generating the initial particle cloud. Suppose we guess that the robot is near (0, 0). This is not exact, as the simulation actually places the robot at (1, 1), but it is close. If we create a normally distributed cloud near (0, 0) there is a much greater chance of the particles matching the robot's position.

`run_pf1()` has an optional parameter `initial_x`. Use this to specify the initial position guess for the robot. The code then uses `create_gaussian_particles(mean, std, N)` to create particles distributed normally around the initial guess. We will use this in the next section.

### 12.6.2 Filter Degeneracy From Inadequate Samples

The filter as written is far from perfect. Here is how it performs with a different random seed.

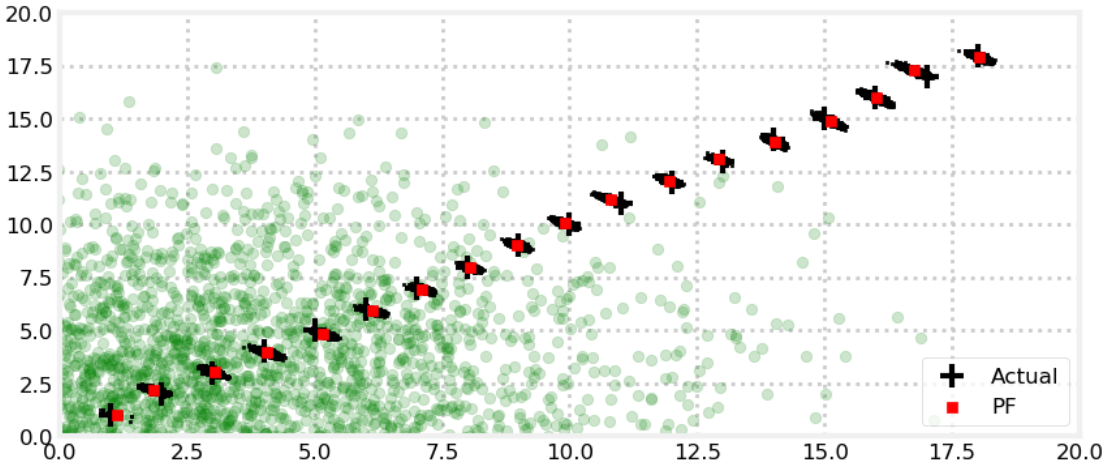
```
In [20]: seed(6)
         run_pf1(N=5000, plot_particles=True, ylim=(-20, 20))

final position error, variance:
         [ -2.688 -31.47 ] [47.065 47.03 ]
```




Here the initial sample of points did not generate any points near the robot. The particle filter does not create new points during the resample operation, so it ends up duplicating points which are not a representative sample of the probability distribution. As mentioned earlier this is called *sample impoverishment*. The problem quickly spirals out of control. The particles are not a good match for the landscape measurement so they become dispersed in a highly nonlinear, curved distribution, and the particle filter diverges from reality. No particles are available near the robot, so it cannot ever converge.

Let's make use of the `create_gaussian_particles()` method to try to generate more points near the robot. We can do this by using the `initial_x` parameter to specify a location to create the particles.

```
In [21]: seed(6)
         run_pf1(N=5000, plot_particles=True, initial_x=(1,1, np.pi/4))

final position error, variance:
         [ 0.035 -0.077] [0.007 0.009]
```




This works great. You should always try to create particles near the initial position if you have any way to roughly estimate it. Do not be *too* careful - if you generate all the points very near a single position the particles may not be dispersed enough to capture the nonlinearities in the system. This is a fairly linear system, so we could get away with a smaller variance in the distribution. Clearly this depends on your problem. Increasing the number of particles is always a good way to get a better sample, but the processing cost may be a higher price than you are willing to pay.

## 12.7 Importance Sampling

I've hand waved a difficulty away which we must now confront. There is some probability distribution that describes the position and movement of our robot. We want to draw a sample of particles from that distribution and compute the integral using MC methods.

Our difficulty is that in many problems we don't know the distribution. For example, the tracked object might move very differently than we predicted with our state model. How can we draw a sample from a probability distribution that is unknown?

There is a theorem from statistics called *importance sampling*[1]. Remarkably, it gives us a way to draw samples from a different and known probability distribution and use those to compute the properties of the unknown one. It's a fantastic theorem that brings joy to my heart.

The idea is simple, and we already used it. We draw samples from the known probability distribution, but *weight the samples* according to the distribution we are interested in. We can then compute properties such as the mean and variance by computing the weighted mean and weighted variance of the samples.

For the robot localization problem we drew samples from the probability distribution that we computed from our state model prediction step. In other words, we reasoned 'the robot was there, it is perhaps moving at this direction and speed, hence it might be here'. Yet the robot might have done something completely different. It may have fell off a cliff or been hit by a mortar round. In each case the probability distribution is not correct. It seems like we are stymied, but we are not because we can use importance sampling. We drew particles from that likely incorrect probability distribution, then weighted them according to how well the particles match the measurements. That weighting is based on the true probability distribution, so according to the theory the resulting mean, variance, etc, will be correct!

How can that be true? I'll give you the math; you can safely skip this if you don't plan to go beyond the robot localization problem. However, other particle filter problems require different approaches to importance sampling, and a bit of math helps. Also, the literature and much of the content on the web uses the mathematical formulation in favor of my rather imprecise "imagine that..." exposition. If you want to understand the literature you will need to know the following equations.

We have some probability distribution $\pi(x)$ which we want to take samples from. However, we don't know what $\pi(x)$ is; instead we only know an alternative probability distribution $q(x)$. In the context of

robot localization, $\pi(x)$ is the probability distribution for the robot, but we don't know it, and $q(x)$ is the probability distribution of our measurements, which we do know.

The expected value of a function $f(x)$ with probability distribution $\pi(x)$ is

$$\mathbb{E}\big[f(x)\big] = \int f(x)\pi(x)\,dx$$

We don't know $\pi(x)$ so we cannot compute this integral. We do know an alternative distribution $q(x)$ so we can add it into the integral without changing the value with

$$\mathbb{E}\big[f(x)\big] = \int f(x)\pi(x)\frac{q(x)}{q(x)}\,dx$$

Now we rearrange and group terms

$$\mathbb{E}\big[f(x)\big] = \int f(x)q(x)\ \cdot \frac{\pi(x)}{q(x)}\,dx$$

$q(x)$ is known to us, so we can compute $\int f(x)q(x)$ using MC integration. That leaves us with $\pi(x)/q(x)$. That is a ratio, and we define it as a *weight*. This gives us

$$\mathbb{E}\big[f(x)\big] = \sum_{i=1}^{N} f(x^i)w(x^i)$$

Maybe that seems a little abstract. If we want to compute the mean of the particles we would compute

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x^i w^i$$

which is the equation I gave you earlier in the chapter.

It is required that the weights be proportional to the ratio $\pi(x)/q(x)$. We normally do not know the exact value, so in practice we normalize the weights by dividing them by $\sum w(x^i)$.

When you formulate a particle filter algorithm you will have to implement this step depending on the particulars of your situation. For robot localization the best distribution to use for $q(x)$ is the particle distribution from the `predict()` step of the filter. Let's look at the code again:

```
def update(particles, weights, z, R, landmarks):
    for i, landmark in enumerate(landmarks):
        dist = np.linalg.norm(particles[:, 0:2] - landmark, axis=1)
        weights *= scipy.stats.norm(dist, R).pdf(z[i])

    weights += 1.e-300      # avoid round-off to zero
    weights /= sum(weights) # normalize
```

Here we compute the weight as the based on the Bayesian computation ‖likelihood × prior‖

Of course if you can compute the posterior probability distribution from the prior you should do so. If you cannot, then importance sampling gives you a way to solve this problem. In practice, computing the posterior is incredibly difficult. The Kalman filter became a spectacular success because it took advantage of the properties of Gaussians to find an analytic solution. Once we relax the conditions required by the Kalman filter (Markov property, Gaussian measurements and process) importance sampling and monte carlo methods make the problem tractable.

## 12.8 Resampling Methods

The resampling algorithm affects the performance of the filter. For example, suppose we resampled particles by picking particles at random. This would lead us to choosing many particles with a very low weight, and the resulting set of particles would be a terrible representation of the problem's probability distribution.

Research on the topic continues, but a handful of algorithms work well in practice across a wide variety of situations. We desire an algorithm that has several properties. It should preferentially select particles that have a higher probability. It should select a representative population of the higher probability particles to avoid sample impoverishment. It should include enough lower probability particles to give the filter a chance of detecting strongly nonlinear behavior.

FilterPy implements several of the popular algorithms. FilterPy doesn't know how your particle filter is implemented, so it cannot generate the new samples. Instead, the algorithms create a `numpy.array` containing the indexes of the particles that are chosen. Your code needs to perform the resampling step. For example, I used this for the robot:

```
In [22]: def resample_from_index(particles, weights, indexes):
             particles[:] = particles[indexes]
             weights.resize(len(particles))
             weights.fill(1.0 / len(weights))
```

### 12.8.1 Multinomial Resampling

Multinomial resampling is the algorithm that I used while developing the robot localization example. The idea is simple. Compute the cumulative sum of the normalized weights. This gives you an array of increasing values from 0 to 1. Here is a plot which illustrates how this spaces out the weights. The colors are meaningless, they just make the divisions easier to see.

```
In [23]: from kf_book.pf_internal import plot_cumsum
         print('cumulative sume is', np.cumsum([.1, .2, .1, .6]))
         plot_cumsum([.1, .2, .1, .6])
```

```
cumulative sume is [0.1 0.3 0.4 1. ]
```

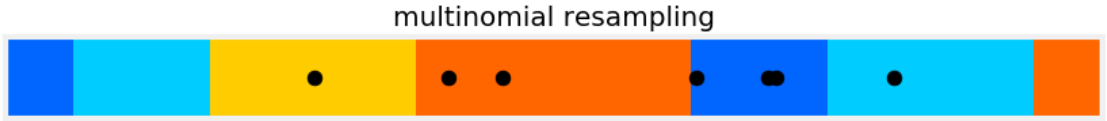

To select a weight we generate a random number uniformly selected between 0 and 1 and use binary search to find its position inside the cumulative sum array. Large weights occupy more space than low weights, so they are more likely to be selected.

This is very easy to code using NumPy's ufunc support. Ufuncs apply functions to every element of an array, returning an array of the results. `searchsorted` is NumPy's binary search algorithm. If you provide it with an array of search values it will return an array of answers: a single answer for each search value.

```
In [24]: def multinomal_resample(weights):
             cumulative_sum = np.cumsum(weights)
             cumulative_sum[-1] = 1.  # avoid round-off errors
             return np.searchsorted(cumulative_sum, random(len(weights)))
```

Here is an example:

```
In [25]: from kf_book.pf_internal import plot_multinomial_resample
         plot_multinomial_resample([.1, .2, .3, .4, .2, .3, .1])
```



This is an $O(n\log(n))$ algorithm. That is not terrible, but there are $O(n)$ resampling algorithms with better properties with respect to the uniformity of the samples. I'm showing it because you can understand the other algorithms as variations on this one. There is a faster implementation of this multinomial resampling that uses the inverse of the CDF of the distribution. You can search on the internet if you are interested.

Import the function from FilterPy using

```
from filterpy.monte_carlo import multinomal_resample
```

### 12.8.2 Residual Resampling

Residual resampling both improves the run time of multinomial resampling, and ensures that the sampling is uniform across the population of particles. It's fairly ingenious: the normalized weights are multiplied by $N$, and then the integer value of each weight is used to define how many samples of that particle will be taken. For example, if the weight of a particle is 0.0012 and $N$=3000, the scaled weight is 3.6, so 3 samples will be taken of that particle. This ensures that all higher weight particles are chosen at least once. The running time is $O(N)$, making it faster than multinomial resampling.

However, this does not generate all $N$ selections. To select the rest, we take the *residual*: the weights minus the integer part, which leaves the fractional part of the number. We then use a simpler sampling scheme such as multinomial, to select the rest of the particles based on the residual. In the example above the scaled weight was 3.6, so the residual will be 0.6 (3.6 - int(3.6)). This residual is very large so the particle will be likely to be sampled again. This is reasonable because the larger the residual the larger the error in the round off, and thus the particle was relatively under sampled in the integer step.

```
In [26]: def residual_resample(weights):
             N = len(weights)
             indexes = np.zeros(N, 'i')

             # take int(N*w) copies of each weight
             num_copies = (N*np.asarray(weights)).astype(int)
             k = 0
             for i in range(N):
                 for _ in range(num_copies[i]): # make n copies
                     indexes[k] = i
                     k += 1

             # use multinormial resample on the residual to fill up the rest.
             residual = w - num_copies     # get fractional part
             residual /= sum(residual)     # normalize
             cumulative_sum = np.cumsum(residual)
             cumulative_sum[-1] = 1. # ensures sum is exactly one
             indexes[k:N] = np.searchsorted(cumulative_sum, random(N-k))

             return indexes
```

You may be tempted to replace the inner for loop with a slice `indexes[k:k + num_copies[i]] = i`, but very short slices are comparatively slow, and the for loop usually runs faster.

Let's look at an example:

```
In [27]: from kf_book.pf_internal import plot_residual_resample
         plot_residual_resample([.1, .2, .3, .4, .2, .3, .1])
```


You may import this from FilterPy using

```
from filterpy.monte_carlo import residual_resample
```

### 12.8.3 Stratified Resampling

This scheme aims to make selections relatively uniformly across the particles. It works by dividing the cumulative sum into $N$ equal sections, and then selects one particle randomly from each section. This guarantees that each sample is between 0 and $\frac{2}{N}$ apart.

The plot below illustrates this. The colored bars show the cumulative sum of the array, and the black lines show the $N$ equal subdivisions. Particles, shown as black circles, are randomly placed in each subdivision.

```
In [28]: from kf_book.pf_internal import plot_stratified_resample
         plot_stratified_resample([.1, .2, .3, .4, .2, .3, .1])
```


The code to perform the stratification is quite straightforward.

```
In [29]: def stratified_resample(weights):
             N = len(weights)
             # make N subdivisions, chose a random position within each one
             positions = (random(N) + range(N)) / N

             indexes = np.zeros(N, 'i')
             cumulative_sum = np.cumsum(weights)
             i, j = 0, 0
             while i < N:
                 if positions[i] < cumulative_sum[j]:
                     indexes[i] = j
                     i += 1
                 else:
                     j += 1
             return indexes
```

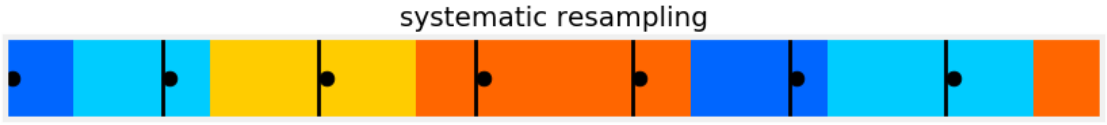
Import it from FilterPy with

```
from filterpy.monte_carlo import stratified_resample
```

### 12.8.4 Systematic Resampling

The last algorithm we will look at is systemic resampling. As with stratified resampling the space is divided into $N$ divisions. We then choose a random offset to use for all of the divisions, ensuring that each sample is exactly $\frac{1}{N}$ apart. It looks like this.

```
In [30]: from kf_book.pf_internal import plot_systematic_resample
         plot_systematic_resample([.1, .2, .3, .4, .2, .3, .1])
```




Having seen the earlier examples the code couldn't be simpler.

```
In [31]: def systematic_resample(weights):
             N = len(weights)

             # make N subdivisions, choose positions
             # with a consistent random offset
             positions = (np.arange(N) + random()) / N

             indexes = np.zeros(N, 'i')
             cumulative_sum = np.cumsum(weights)
             i, j = 0, 0
             while i < N:
                 if positions[i] < cumulative_sum[j]:
                     indexes[i] = j
                     i += 1
                 else:
                     j += 1
             return indexes
```

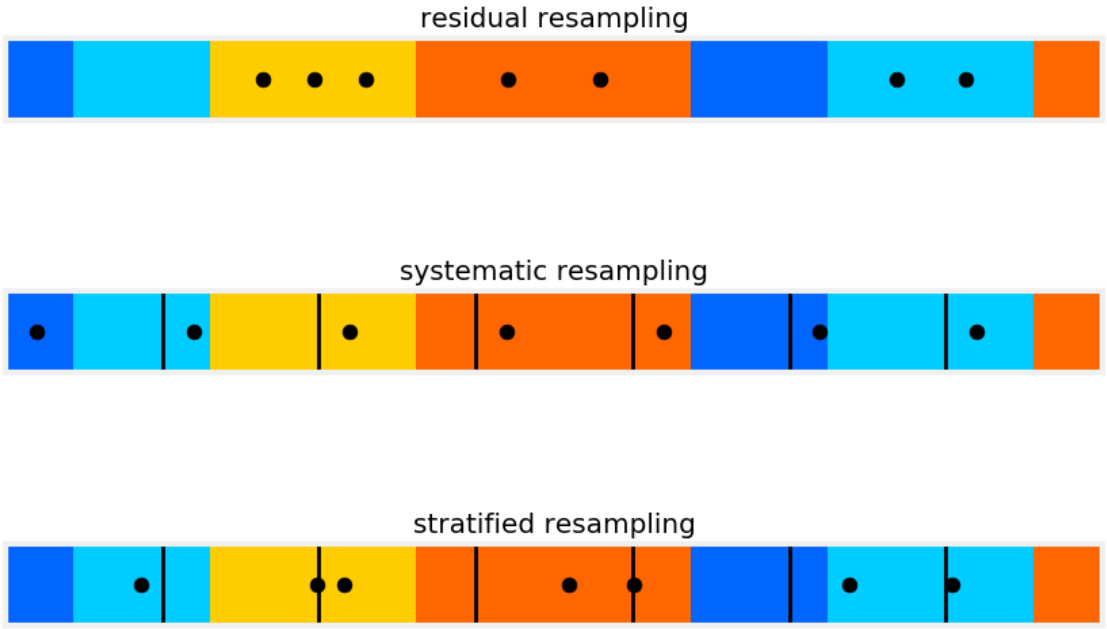
Import from FilterPy with

```
from filterpy.monte_carlo import systematic_resample
```

### 12.8.5 Choosing a Resampling Algorithm

Let's look at the four algorithms at once so they are easier to compare.

```
In [32]: a = [.1, .2, .3, .4, .2, .3, .1]
         np.random.seed(4)
         plot_multinomial_resample(a)
         plot_residual_resample(a)
         plot_systematic_resample(a)
         plot_stratified_resample(a)
```




The performance of the multinomial resampling is quite bad. There is a very large weight that was not sampled at all. The largest weight only got one resample, yet the smallest weight was sample was sampled twice. Most tutorials on the net that I have read use multinomial resampling, and I am not sure why. Multinomial resampling is rarely used in the literature or for real problems. I recommend not using it unless you have a very good reason to do so.

The residual resampling algorithm does excellently at what it tries to do: ensure all the largest weights are resampled multiple times. It doesn't evenly distribute the samples across the particles - many reasonably large weights are not resampled at all.

Both systematic and stratified perform very well. Systematic sampling does an excellent job of ensuring we sample from all parts of the particle space while ensuring larger weights are proportionality resampled more often. Stratified resampling is not quite as uniform as systematic resampling, but it is a bit better at ensuring the higher weights get resampled more.

Plenty has been written on the theoretical performance of these algorithms, and feel free to read it. In practice I apply particle filters to problems that resist analytic efforts, and so I am a bit dubious about the validity of a specific analysis to these problems. In practice both the stratified and systematic algorithms perform well and similarly across a variety of problems. I say try one, and if it works stick with it. If performance of the filter is critical try both, and perhaps see if there is literature published on your specific problem that will give you better guidance.

## 12.9 Summary

This chapter only touches the surface of what is a vast topic. My goal was not to teach you the field, but to expose you to practical Bayesian Monte Carlo techniques for filtering.

Particle filters are a type of *ensemble* filtering. Kalman filters represents state with a Gaussian. Measurements are applied to the Gaussian using Bayes Theorem, and the prediction is done using state-space methods. These techniques are applied to the Gaussian - the probability distribution.

In contrast, ensemble techniques represent a probability distribution using a discrete collection of points and associated probabilities. Measurements are applied to these points, not the Gaussian distribution.

Likewise, the system model is applied to the points, not a Gaussian. We then compute the statistical properties of the resulting ensemble of points.

These choices have many trade-offs. The Kalman filter is very efficient, and is an optimal estimator if the assumptions of linearity and Gaussian noise are true. If the problem is nonlinear than we must linearize the problem. If the problem is multimodal (more than one object being tracked) then the Kalman filter cannot represent it. The Kalman filter requires that you know the state model. If you do not know how your system behaves the performance is poor.

In contrast, particle filters work with any arbitrary, non-analytic probability distribution. The ensemble of particles, if large enough, form an accurate approximation of the distribution. It performs wonderfully even in the presence of severe nonlinearities. Importance sampling allows us to compute probabilities even if we do not know the underlying probability distribution. Monte Carlo techniques replace the analytic integrals required by the other filters.

This power comes with a cost. The most obvious costs are the high computational and memory burdens the filter places on the computer. Less obvious is the fact that they are fickle. You have to be careful to avoid particle degeneracy and divergence. It can be very difficult to prove the correctness of your filter. If you are working with multimodal distributions you have further work to cluster the particles to determine the paths of the multiple objects. This can be very difficult when the objects are close to each other.

There are many different classes of particle filter; I only described the naive SIS algorithm, and followed that with a SIR algorithm that performs well. There are many classes of filters, and many examples of filters in each class. It would take a small book to describe them all.

When you read the literature on particle filters you will find that it is strewn with integrals. We perform computations on probability distributions using integrals, so using integrals gives the authors a powerful and compact notation. You must recognize that when you reduce these equations to code you will be representing the distributions with particles, and integrations are replaced with sums over the particles. If you keep in mind the core ideas in this chapter the material shouldn't be daunting.

## 12.10 References

[1] *Importance Sampling*, Wikipedia. https://en.wikipedia.org/wiki/Importance_sampling

# Chapter 13

# Smoothing

```
In [1]: from __future__ import division, print_function
        %matplotlib inline
```

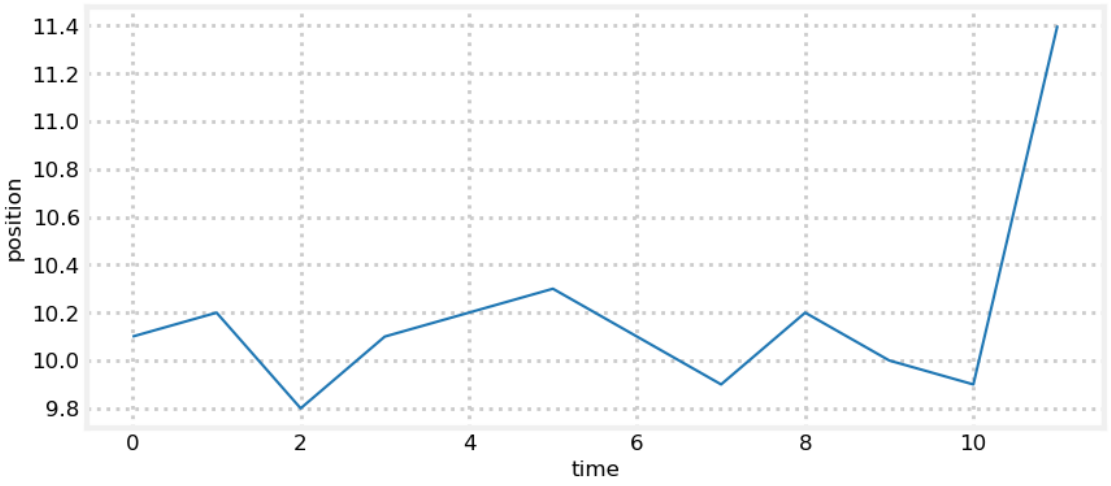
```
In [2]: #format the book
        import book_format
        book_format.set_style()
```

```
Out[2]:
```

## 13.1 Introduction

The performance of the Kalman filter is not optimal when you consider future data. For example, suppose we are tracking an aircraft, and the latest measurement deviates far from the current track, like so (I'll only consider 1 dimension for simplicity):

```
In [3]: import matplotlib.pyplot as plt

        data = [10.1, 10.2, 9.8, 10.1, 10.2, 10.3,
                10.1, 9.9, 10.2, 10.0, 9.9, 11.4]
        plt.plot(data)
        plt.xlabel('time')
        plt.ylabel('position');
```



After a period of near steady state, we have a very large change. Assume the change is past the limit of the aircraft's flight envelope. Nonetheless the Kalman filter incorporates that new measurement into the filter based on the current Kalman gain. It cannot reject the noise because the measurement could reflect the initiation of a turn. Granted it is unlikely that we are turning so abruptly, but it is impossible to say whether

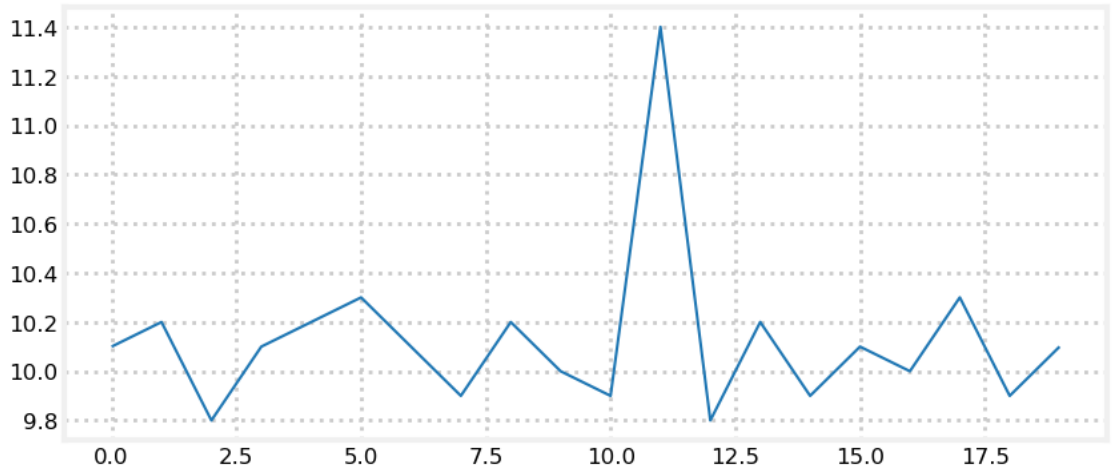
- The aircraft started a turn awhile ago, but the previous measurements were noisy and didn't show the change.
- The aircraft is turning, and this measurement is very noisy
- The measurement is very noisy and the aircraft has not turned
- The aircraft is turning in the opposite direction, and the measurement is extremely noisy

Now, suppose the following measurements are:
11.3 12.1 13.3 13.9 14.5 15.2

```
In [4]: data2 = [11.3, 12.1, 13.3, 13.9, 14.5, 15.2]
        plt.plot(data + data2);
```


Given these future measurements we can infer that yes, the aircraft initiated a turn.
On the other hand, suppose these are the following measurements.

```
In [5]: data3 = [9.8, 10.2, 9.9, 10.1, 10.0, 10.3, 9.9, 10.1]
        plt.plot(data + data3);
```




In this case we are led to conclude that the aircraft did not turn and that the outlying measurement was merely very noisy.

## 13.2 An Overview of How Smoothers Work

The Kalman filter is a *recursive* filter with the Markov property - it's estimate at step `k` is based only on the estimate from step `k-1` and the measurement at step `k`. But this means that the estimate from step `k-1` is based on step `k-2`, and so on back to the first epoch. Hence, the estimate at step `k` depends on all of the previous measurements, though to varying degrees. `k-1` has the most influence, `k-2` has the next most, and so on.

Smoothing filters incorporate future measurements into the estimate for step `k`. The measurement from `k+1` will have the most effect, `k+2` will have less effect, `k+3` less yet, and so on.

This topic is called *smoothing*, but I think that is a misleading name. I could smooth the data above by passing it through a low pass filter. The result would be smooth, but not necessarily accurate because a low pass filter will remove real variations just as much as it removes noise. In contrast, Kalman smoothers are *optimal* - they incorporate all available information to make the best estimate that is mathematically achievable.

## 13.3 Types of Smoothers

There are three classes of Kalman smoothers that produce better tracking in these situations.

- Fixed-Interval Smoothing

This is a batch processing based filter. This filter waits for all of the data to be collected before making any estimates. For example, you may be a scientist collecting data for an experiment, and don't need to know the result until the experiment is complete. A fixed-interval smoother will collect all the data, then estimate the state at each measurement using all available previous and future measurements. If it is possible for you to run your Kalman filter in batch mode it is always recommended to use one of these filters a it will provide much better results than the recursive forms of the filter from the previous chapters.

- Fixed-Lag Smoothing

Fixed-lag smoothers introduce latency into the output. Suppose we choose a lag of 4 steps. The filter will ingest the first 3 measurements but not output a filtered result. Then, when the 4th measurement comes in the filter will produce the output for measurement 1, taking measurements 1 through 4 into account. When

the 5th measurement comes in, the filter will produce the result for measurement 2, taking measurements 2 through 5 into account. This is useful when you need recent data but can afford a bit of lag. For example, perhaps you are using machine vision to monitor a manufacturing process. If you can afford a few seconds delay in the estimate a fixed-lag smoother will allow you to produce very accurate and smooth results.

- Fixed-Point Smoothing

A fixed-point filter operates as a normal Kalman filter, but also produces an estimate for the state at some fixed time $j$. Before the time $k$ reaches $j$ the filter operates as a normal filter. Once $k > j$ the filter estimates $x_k$ and then also updates its estimate for $x_j$ using all of the measurements between $j \dots k$. This can be useful to estimate initial paramters for a system, or for producing the best estimate for an event that happened at a specific time. For example, you may have a robot that took a photograph at time $j$. You can use a fixed-point smoother to get the best possible pose information for the camera at time $j$ as the robot continues moving.

## 13.4 Choice of Filters

The choice of these filters depends on your needs and how much memory and processing time you can spare. Fixed-point smoothing requires storage of all measurements, and is very costly to compute because the output is for every time step is recomputed for every measurement. On the other hand, the filter does produce a decent output for the current measurement, so this filter can be used for real time applications.

Fixed-lag smoothing only requires you to store a window of data, and processing requirements are modest because only that window is processed for each new measurement. The drawback is that the filter's output always lags the input, and the smoothing is not as pronounced as is possible with fixed-interval smoothing.

Fixed-interval smoothing produces the most smoothed output at the cost of having to be batch processed. Most algorithms use some sort of forwards/backwards algorithm that is only twice as slow as a recursive Kalman filter.

## 13.5 Fixed-Interval Smoothing

There are many fixed-lag smoothers available in the literature. I have chosen to implement the smoother invented by Rauch, Tung, and Striebel because of its ease of implementation and efficiency of computation. It is also the smoother I have seen used most often in real applications. This smoother is commonly known as an RTS smoother.

Derivation of the RTS smoother runs to several pages of densely packed math. I'm not going to inflict it on you. Instead I will briefly present the algorithm, equations, and then move directly to implementation and demonstration of the smoother.

The RTS smoother works by first running the Kalman filter in a batch mode, computing the filter output for each step. Given the filter output for each measurement along with the covariance matrix corresponding to each output the RTS runs over the data backwards, incorporating its knowledge of the future into the past measurements. When it reaches the first measurement it is done, and the filtered output incorporates all of the information in a maximally optimal form.

The equations for the RTS smoother are very straightforward and easy to implement. This derivation is for the linear Kalman filter. Similar derivations exist for the EKF and UKF. These steps are performed on the output of the batch processing, going backwards from the most recent in time back to the first estimate. Each iteration incorporates the knowledge of the future into the state estimate. Since the state estimate already incorporates all of the past measurements the result will be that each estimate will contain knowledge of all measurements in the past and future. Here is it very important to distinguish between past, present, and future so I have used subscripts to denote whether the data is from the future or not.

`Predict Step`

$$\mathbf{P} = \mathbf{F}\mathbf{P}_k\mathbf{F}^\mathsf{T} + \mathbf{Q}$$

Update Step

$$\begin{aligned}\mathbf{K}_k &= \mathbf{P}_k\mathbf{F}^\mathsf{T}\mathbf{P}^{-1} \\ \mathbf{x}_k &= \mathbf{x}_k + \mathbf{K}_k(\mathbf{x}_{k+1} - \mathbf{F}\mathbf{x}_k) \\ \mathbf{P}_k &= \mathbf{P}_k + \mathbf{K}_k(\mathbf{P}_{k+1} - \mathbf{P})\mathbf{K}_k^\mathsf{T}\end{aligned}$$

As always, the hardest part of the implementation is correctly accounting for the subscripts. A basic implementation without comments or error checking would be:

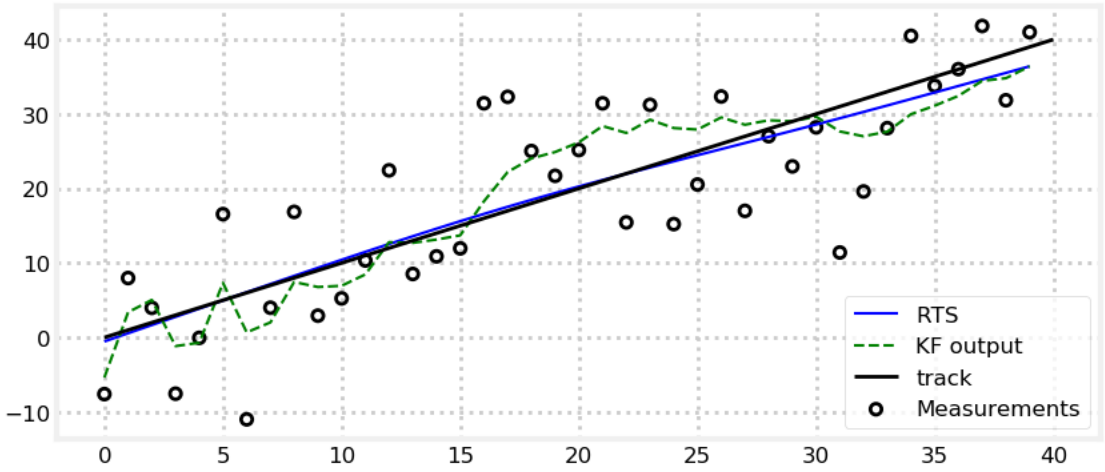
```
def rts_smoother(Xs, Ps, F, Q):
    n, dim_x, _ = Xs.shape

    # smoother gain
    K = zeros((n,dim_x, dim_x))
    x, P, Pp = Xs.copy(), Ps.copy(), Ps.copy

    for k in range(n-2,-1,-1):
        Pp[k] = dot(F, P[k]).dot(F.T) + Q # predicted covariance

        K[k]  = dot(P[k], F.T).dot(inv(Pp[k]))
        x[k] += dot(K[k], x[k+1] - dot(F, x[k]))
        P[k] += dot(K[k], P[k+1] - Pp[k]).dot(K[k].T)
    return (x, P, K, Pp)
```

This implementation mirrors the implementation provided in FilterPy. It assumes that the Kalman filter is being run externally in batch mode, and the results of the state and covariances are passed in via the `Xs` and `Ps` variable.

Here is an example.

```
In [6]: import numpy as np
        from numpy import random
        from numpy.random import randn
        import matplotlib.pyplot as plt
        from filterpy.kalman import KalmanFilter
        import kf_book.book_plots as bp

        def plot_rts(noise, Q=0.001, show_velocity=False):
            random.seed(123)
            fk = KalmanFilter(dim_x=2, dim_z=1)

            fk.x = np.array([0., 1.])      # state (x and dx)

            fk.F = np.array([[1., 1.],
                             [0., 1.]])    # state transition matrix

            fk.H = np.array([[1., 0.]])    # Measurement function
            fk.P = 10.                     # covariance matrix
            fk.R = noise                   # state uncertainty
            fk.Q = Q                       # process uncertainty

            # create noisy data
            zs = np.asarray([t + randn()*noise for t in range (40)])

            # filter data with Kalman filter, than run smoother on it
```

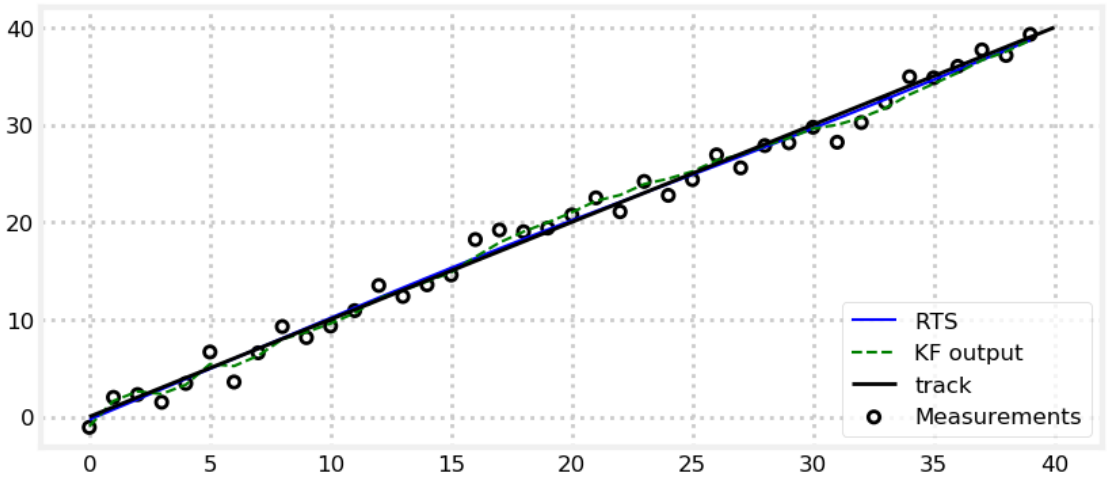
```
    mu, cov, _, _ = fk.batch_filter(zs)
    M, P, C, _ = fk.rts_smoother(mu, cov)

    # plot data
    if show_velocity:
        index = 1
        print('gu')
    else:
        index = 0
    if not show_velocity:
        bp.plot_measurements(zs, lw=1)
    plt.plot(M[:, index], c='b', label='RTS')
    plt.plot(mu[:, index], c='g', ls='--', label='KF output')
    if not show_velocity:
        N = len(zs)
        plt.plot([0, N], [0, N], 'k', lw=2, label='track')
    plt.legend(loc=4)
    plt.show()

plot_rts(7.)
```




I've injected a lot of noise into the signal to allow you to visually distinguish the RTS output from the ideal output. In the graph above we can see that the Kalman filter, drawn as the green dotted line, is reasonably smooth compared to the input, but it still wanders from from the ideal line when several measurements in a row are biased towards one side of the line. In contrast, the RTS output is both extremely smooth and very close to the ideal output.

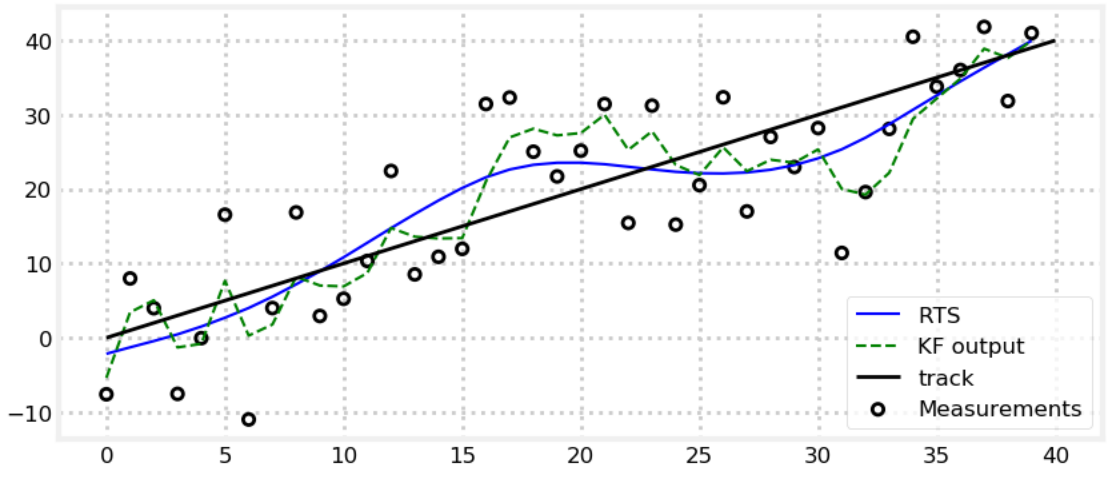
With a perhaps more reasonable amount of noise we can see that the RTS output nearly lies on the ideal output. The Kalman filter output, while much better, still varies by a far greater amount.

```
In [7]: plot_rts(noise=1.)
```




However, we must understand that this smoothing is predicated on the system model. We have told the filter that what we are tracking follows a constant velocity model with very low process error. When the filter *looks ahead* it sees that the future behavior closely matches a constant velocity so it is able to reject most of the noise in the signal. Suppose instead our system has a lot of process noise. For example, if we are tracking a light aircraft in gusty winds its velocity will change often, and the filter will be less able to distinguish between noise and erratic movement due to the wind. We can see this in the next graph.

```
In [8]: plot_rts(noise=7., Q=.1)
```

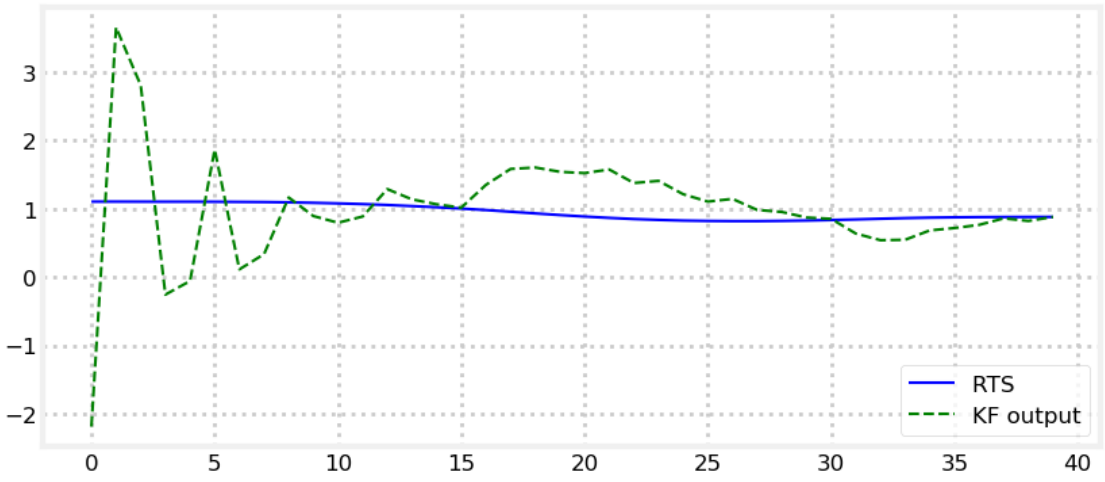



This underscores the fact that these filters are not *smoothing* the data in colloquial sense of the term. The filter is making an optimal estimate based on previous measurements, future measurements, and what you tell it about the behavior of the system and the noise in the system and measurements.

Let's wrap this up by looking at the velocity estimates of Kalman filter vs the RTS smoother.

```
In [9]: plot_rts(7.,show_velocity=True)
```

gu

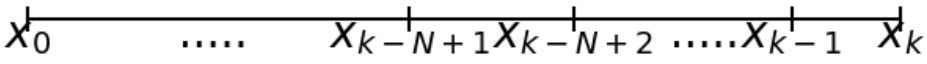



The improvement in the velocity, which is an hidden variable, is even more dramatic.

## 13.6 Fixed-Lag Smoothing

The RTS smoother presented above should always be your choice of algorithm if you can run in batch mode because it incorporates all available data into each estimate. Not all problems allow you to do that, but you may still be interested in receiving smoothed values for previous estimates. The number line below illustrates this concept.

```
In [10]: from kf_book.book_plots import figsize
         from kf_book.smoothing_internal import *

         with figsize(y=2):
             show_fixed_lag_numberline()
```




At step $k$ we can estimate $x_k$ using the normal Kalman filter equations. However, we can make a better estimate for $x_{k-1}$ by using the measurement received for $x_k$. Likewise, we can make a better estimate for $x_{k-2}$ by using the measurements recevied for $x_{k-1}$ and $x_k$. We can extend this computation back for an arbitrary $N$ steps.

Derivation for this math is beyond the scope of this book; Dan Simon's *Optimal State Estimation* [2] has a very good exposition if you are interested. The essense of the idea is that instead of having a state vector $\mathbf{x}$ we make an augmented state containing

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_k \\ \mathbf{x}_{k-1} \\ \vdots \\ \mathbf{x}_{k-N+1} \end{bmatrix}$$

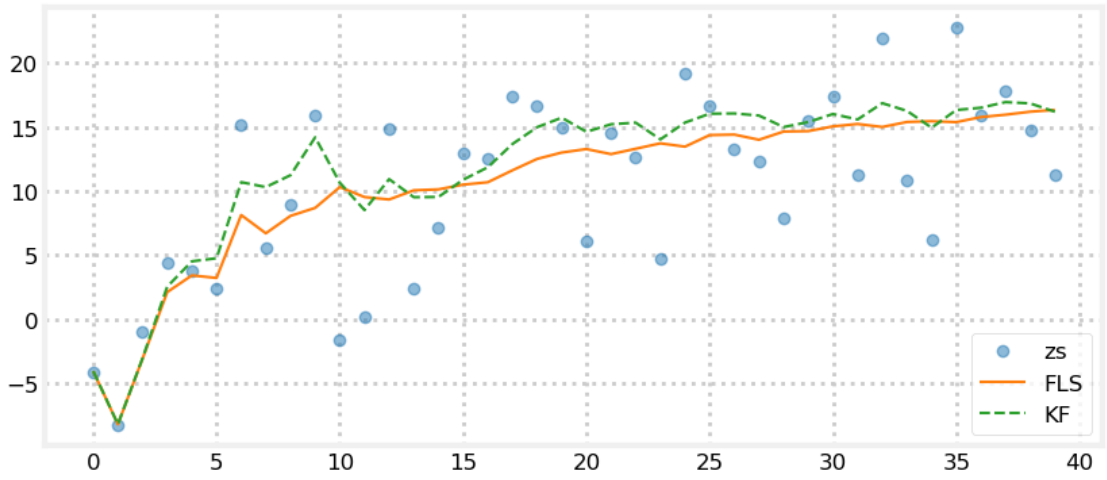
This yields a very large covariance matrix that contains the covariance between states at different steps. FilterPy's class `FixedLagSmoother` takes care of all of this computation for you, including creation of the augmented matrices. All you need to do is compose it as if you are using the `KalmanFilter` class and then call `smooth()`, which implements the predict and update steps of the algorithm.

Each call of `smooth` computes the estimate for the current measurement, but it also goes back and adjusts the previous `N-1` points as well. The smoothed values are contained in the list `FixedLagSmoother.xSmooth`. If you use `FixedLagSmoother.x` you will get the most recent estimate, but it is not smoothed and is no different from a standard Kalman filter output.

```
In [11]: from filterpy.kalman import FixedLagSmoother, KalmanFilter
         import numpy.random as random

         fls = FixedLagSmoother(dim_x=2, dim_z=1, N=8)

         fls.x = np.array([0., .5])
         fls.F = np.array([[1.,1.],
                           [0.,1.]])

         fls.H = np.array([[1.,0.]])
         fls.P *= 200
         fls.R *= 5.
         fls.Q *= 0.001

         kf = KalmanFilter(dim_x=2, dim_z=1)
         kf.x = np.array([0., .5])
         kf.F = np.array([[1.,1.],
                          [0.,1.]])
         kf.H = np.array([[1.,0.]])
         kf.P *= 200
         kf.R *= 5.
         kf.Q *= 0.001

         N = 4 # size of lag

         nom =  np.array([t/2. for t in range (0, 40)])
         zs = np.array([t + random.randn()*5.1 for t in nom])

         for z in zs:
             fls.smooth(z)

         kf_x, _, _, _ = kf.batch_filter(zs)
         x_smooth = np.array(fls.xSmooth)[:, 0]


         fls_res = abs(x_smooth - nom)
         kf_res = abs(kf_x[:, 0] - nom)

         plt.plot(zs,'o', alpha=0.5, marker='o', label='zs')
         plt.plot(x_smooth, label='FLS')
```

```
        plt.plot(kf_x[:, 0], label='KF', ls='--')
        plt.legend(loc=4)

        print('standard deviation fixed-lag: {:.3f}'.format(np.mean(fls_res)))
        print('standard deviation kalman: {:.3f}'.format(np.mean(kf_res)))
```

```
standard deviation fixed-lag: 2.616
standard deviation kalman: 3.562
```




Here I have set `N=8` which means that we will incorporate 8 future measurements into our estimates. This provides us with a very smooth estimate once the filter converges, at the cost of roughly 8x the amount of computation of the standard Kalman filter. Feel free to experiment with larger and smaller values of `N`. I chose 8 somewhat at random, not due to any theoretical concerns.

## 13.7 References

[1] H. Rauch, F. Tung, and C. Striebel. "Maximum likelihood estimates of linear dynamic systems," *AIAA Journal*, **3**(8), pp. 1445-1450 (August 1965).

[2] Dan Simon. "Optimal State Estimation," John Wiley & Sons, 2006.
http://arc.aiaa.org/doi/abs/10.2514/3.3166

# Chapter 14

# Adaptive Filtering

```
In [1]: from __future__ import division, print_function
        %matplotlib inline
```

```
In [2]: #format the book
        import book_format
        book_format.set_style()
```

```
Out[2]:
```

## 14.1 Introduction

So far we have considered the problem of tracking objects that are well behaved in relation to our process model. For example, we can use a constant velocity filter to track an object moving in a straight line. So long as the object moves in a straight line at a reasonably constant speed, or varies its track and/or velocity very slowly this filter will perform very well. Suppose instead that we are trying to track a maneuvering target, such as a car along a road, an aircraft in flight, and so on. In these situations the filters perform quite poorly. Alternatively, consider a situation such as tracking a sailboat in the ocean. Even if we model the control inputs we have no way to model the wind or the ocean currents.

A first order approach to this problem is to make the process noise $\mathbf{Q}$ larger to account for the unpredictability of the system dynamics. While this can work in the sense of providing a non-diverging filter, the result is typically far from optimal. The larger $\mathbf{Q}$ results in the filter giving more emphasis to the noise in the measurements. We will see an example of this shortly.

In this chapter we will discuss the concept of an *adaptive filter*. The filter will *adapt* itself when it detects dynamics that the process model cannot account for. I will start with an example of the problem, and then discuss and implement various adaptive filters.

## 14.2 Maneuvering Targets

We need a simulation of a maneuvering target. I will implement a simple 2D model with steering inputs. You provide a new speed and/or direction, and it will modify its state to match.

```
In [3]: from math import sin, cos, radians

        def angle_between(x, y):
          return min(y-x, y-x+360, y-x-360, key=abs)

        class ManeuveringTarget(object):
            def __init__(self, x0, y0, v0, heading):
                self.x = x0
```

```
        self.y = y0
        self.vel = v0
        self.hdg = heading

        self.cmd_vel = v0
        self.cmd_hdg = heading
        self.vel_step = 0
        self.hdg_step = 0
        self.vel_delta = 0
        self.hdg_delta = 0


    def update(self):
        vx = self.vel * cos(radians(90-self.hdg))
        vy = self.vel * sin(radians(90-self.hdg))
        self.x += vx
        self.y += vy

        if self.hdg_step > 0:
            self.hdg_step -= 1
            self.hdg += self.hdg_delta

        if self.vel_step > 0:
            self.vel_step -= 1
            self.vel += self.vel_delta
        return (self.x, self.y)


    def set_commanded_heading(self, hdg_degrees, steps):
        self.cmd_hdg = hdg_degrees
        self.hdg_delta = angle_between(self.cmd_hdg,
                                       self.hdg) / steps
        if abs(self.hdg_delta) > 0:
            self.hdg_step = steps
        else:
            self.hdg_step = 0


    def set_commanded_speed(self, speed, steps):
        self.cmd_vel = speed
        self.vel_delta = (self.cmd_vel - self.vel) / steps
        if abs(self.vel_delta) > 0:
            self.vel_step = steps
        else:
            self.vel_step = 0
```

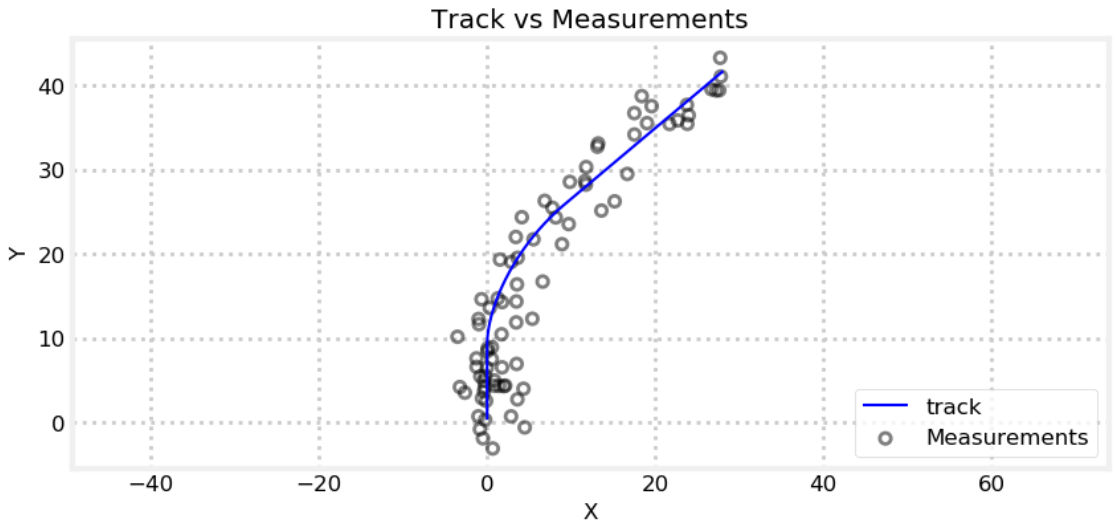
Now let's implement a simulated sensor with noise.

```
In [4]: from numpy.random import randn

    class NoisySensor(object):
        def __init__(self, std_noise=1.):
            self.std = std_noise

        def sense(self, pos):
```

```
            """Pass in actual position as tuple (x, y).
            Returns position with noise added (x,y)"""

            return (pos[0] + (randn() * self.std),
                    pos[1] + (randn() * self.std))
```

Now let's generate a track and plot it to test that everything is working. I'll put the data generation in a function so we can create paths of different lengths (why will be clear soon).

```
In [5]: import kf_book.book_plots as bp
        import numpy as np
        import matplotlib.pyplot as plt

        def generate_data(steady_count, std):
            t = ManeuveringTarget(x0=0, y0=0, v0=0.3, heading=0)
            xs, ys = [], []

            for i in range(30):
                x, y = t.update()
                xs.append(x)
                ys.append(y)

            t.set_commanded_heading(310, 25)
            t.set_commanded_speed(1, 15)

            for i in range(steady_count):
                x, y = t.update()
                xs.append(x)
                ys.append(y)

            ns = NoisySensor(std)
            pos = np.array(list(zip(xs, ys)))
            zs = np.array([ns.sense(p) for p in pos])
            return pos, zs

        sensor_std = 2.
        track, zs = generate_data(50, sensor_std)
        plt.figure()
        bp.plot_measurements(*zip(*zs), alpha=0.5)
        plt.plot(*zip(*track), color='b', label='track')
        plt.axis('equal')
        plt.legend(loc=4)
        bp.set_labels(title='Track vs Measurements', x='X', y='Y')
```




This large amount of noise allows us to see the effect of various design choices more easily.

Now we can implement a Kalman filter to track this object. But let's make a simplification. The $x$ and $y$ coordinates are independent, so we can track each independently. In the remainder of this chapter we will only track the $x$ coordinate to keep the code and matrices as small as possible.

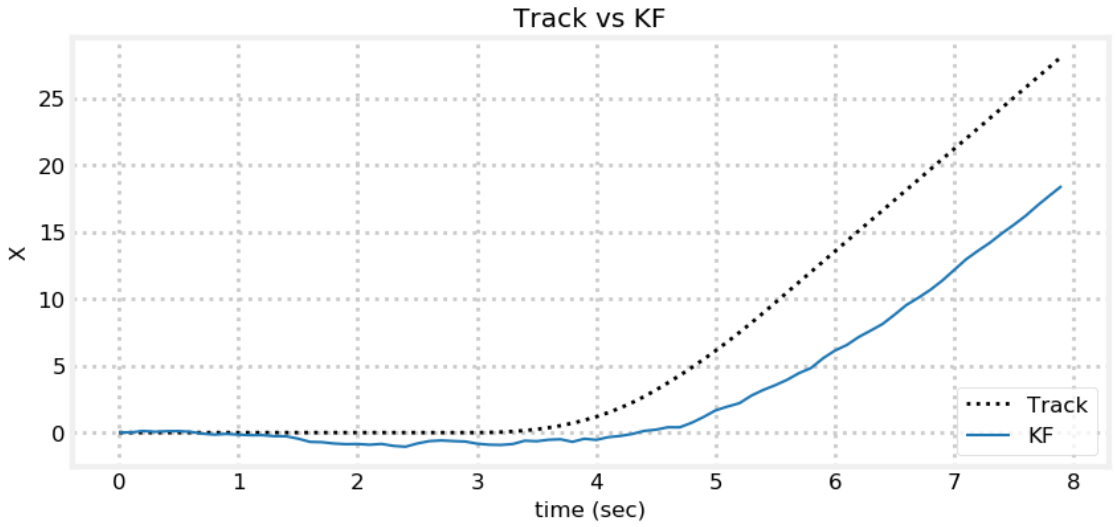
We start with a constant velocity filter.

```
In [6]: from filterpy.kalman import KalmanFilter
        from filterpy.common import Q_discrete_white_noise

        def make_cv_filter(dt, std):
            cvfilter = KalmanFilter(dim_x = 2, dim_z=1)
            cvfilter.x = np.array([0., 0.])
            cvfilter.P *= 3
            cvfilter.R *= std**2
            cvfilter.F = np.array([[1, dt],
                                   [0,  1]], dtype=float)
            cvfilter.H = np.array([[1, 0]], dtype=float)
            cvfilter.Q = Q_discrete_white_noise(dim=2, dt=dt, var=0.02)
            return cvfilter

        def initialize_filter(kf, std_R=None):
            """ helper function - we will be reinitialing the filter
            many times.
            """
            kf.x.fill(0.)
            kf.P = np.eye(kf.dim_x) * .1
            if std_R is not None:
                kf.R = np.eye(kf.dim_z) * std_R
```

Now we run it:

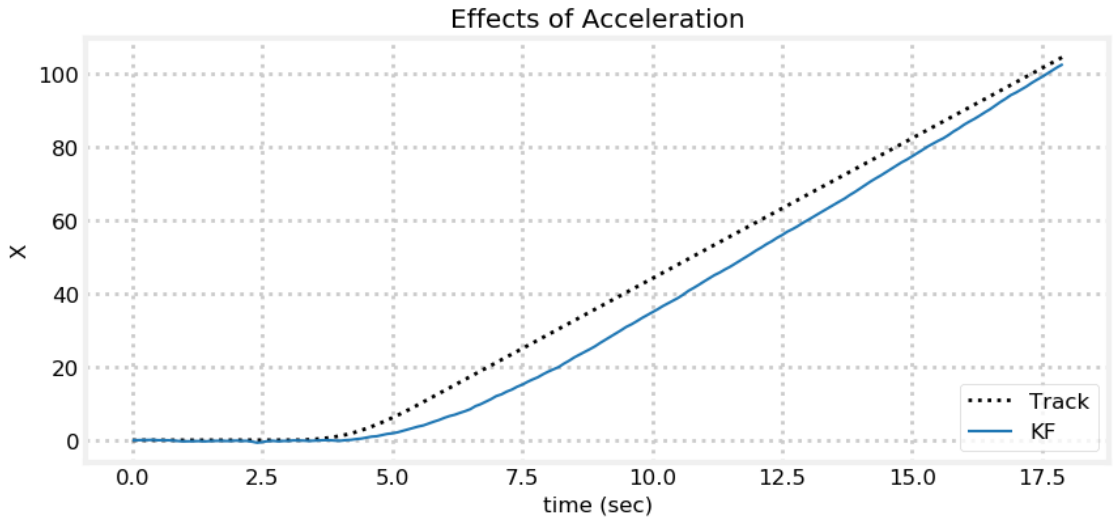
```
In [7]: sensor_std = 2.
        dt = 0.1

        # initialize filter
```

```
cvfilter = make_cv_filter(dt, sensor_std)
initialize_filter(cvfilter)

track, zs = generate_data(50, sensor_std)

# run it
z_xs = zs[:, 0]
kxs, _, _, _ = cvfilter.batch_filter(z_xs)

# plot results
bp.plot_track(track[:, 0], dt=dt)
bp.plot_filter(kxs[:, 0], dt=dt, label='KF')
bp.set_labels(title='Track vs KF', x='time (sec)', y='X');
plt.legend(loc=4);
```



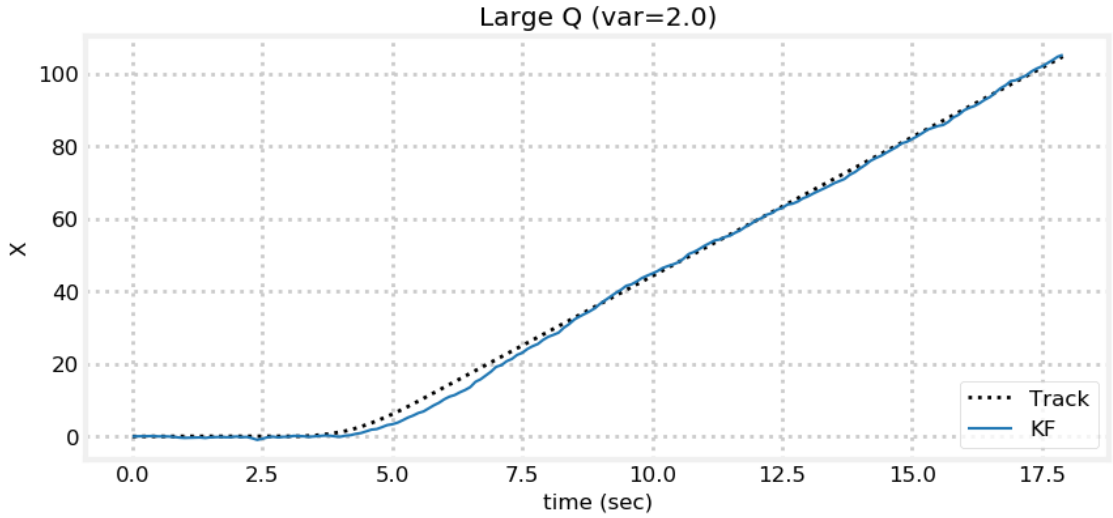

We can see from the plot that the Kalman filter was unable to track the change in heading. Recall from the **g-h Filter** chapter that this is because the filter is not modeling acceleration, hence it will always lag the input. The filter will eventually catch up with the signal if the signal enters a steady state. Let's look at that.

In [8]:
```
# reinitialize filter
dt = 0.1
initialize_filter(cvfilter)

track2, zs2 = generate_data(150, sensor_std)
xs2 = track2[:, 0]
z_xs2 = zs2[:, 0]

kxs2, _, _, _ = cvfilter.batch_filter(z_xs2)

bp.plot_track(xs2, dt=dt)
bp.plot_filter(kxs2[:, 0], dt=dt, label='KF')
plt.legend(loc=4)
bp.set_labels(title='Effects of Acceleration',
              x='time (sec)', y='X')
```



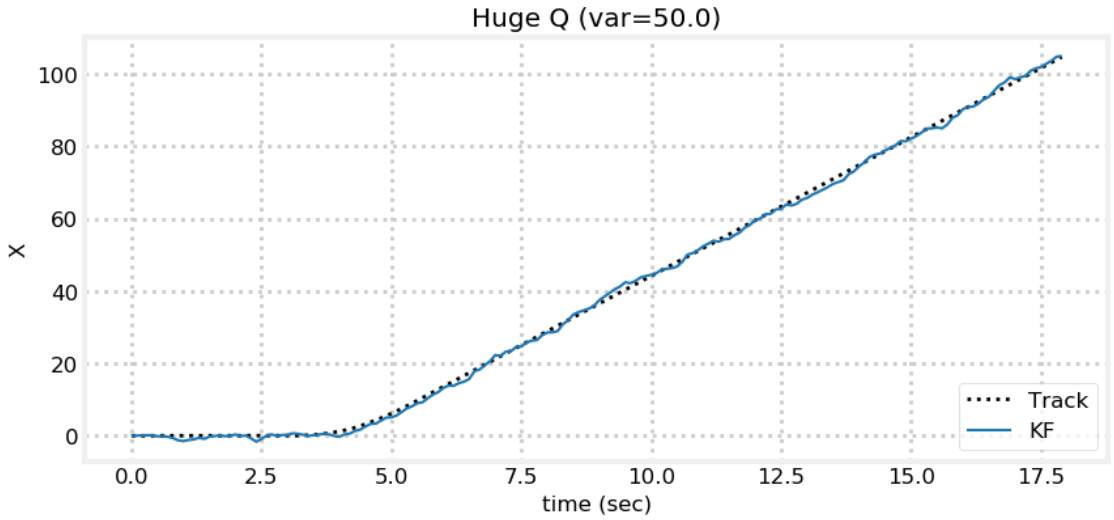

The underlying problem is that our process model is correct for the steady state sections, but incorrect for when the object is maneuvering. We can try to account for this by increasing the size of Q, like so.

```
In [9]: # reinitialize filter
        dt = 0.1
        initialize_filter(cvfilter)
        cvfilter.Q = Q_discrete_white_noise(dim=2, dt=dt, var=2.0)
        track, zs = generate_data(50, sensor_std)

        # recompute track
        kxs2, _, _, _ = cvfilter.batch_filter(z_xs2)
        bp.plot_track(xs2, dt=dt)
        bp.plot_filter(kxs2[:, 0], dt=dt, label='KF')
        plt.legend(loc=4)
        bp.set_labels(title='Large Q (var=2.0)', x='time (sec)', y='X')
```



We can see that the filter reacquired the track more quickly, but at the cost of a lot of noise in the output. Furthermore, many tracking situations could not tolerate the amount of lag shown between seconds 4 and 8. We could reduce it further at the cost of very noisy output, like so:

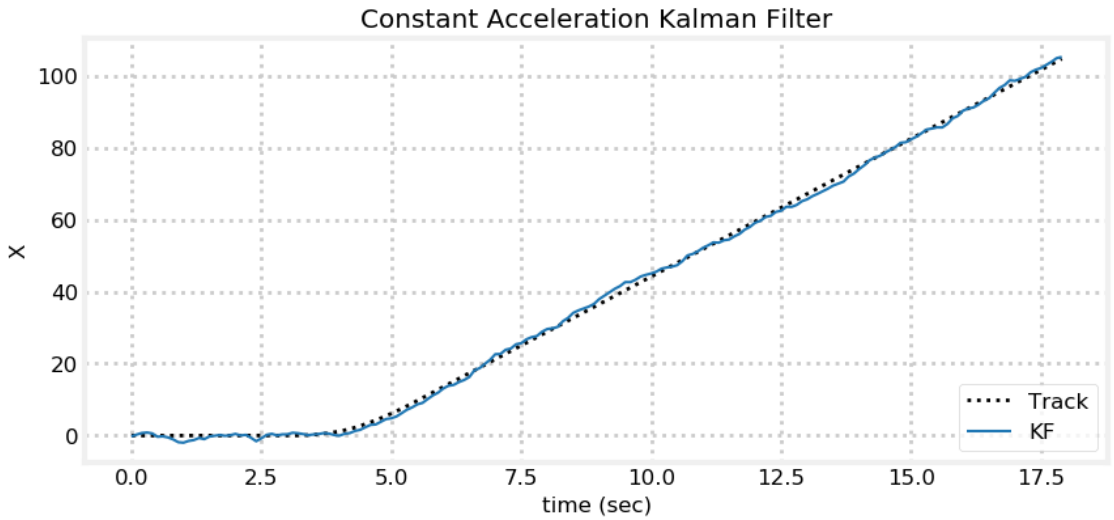
```
In [10]: # reinitialize filter
         dt = 0.1
         initialize_filter(cvfilter)
         cvfilter.Q = Q_discrete_white_noise(dim=2, dt=dt, var=50.0)
         track, zs = generate_data(50, sensor_std)

         # recompute track
         cvfilter.x.fill(0.)
         kxs2, _, _, _ = cvfilter.batch_filter(z_xs2)

         bp.plot_track(xs2, dt=dt)
         bp.plot_filter(kxs2[:, 0], dt=dt, label='KF')
         plt.legend(loc=4)
         bp.set_labels(title='Huge Q (var=50.0)', x='time (sec)', y='X')
```




Maneuvers imply acceleration, so let's implement a constant acceleration Kalman filter and see how it fairs with the same data.

```
In [11]: def make_ca_filter(dt, std):
             cafilter = KalmanFilter(dim_x=3, dim_z=1)
             cafilter.x = np.array([0., 0., 0.])
             cafilter.P *= 3
             cafilter.R *= std
             cafilter.Q = Q_discrete_white_noise(dim=3, dt=dt, var=0.02)
             cafilter.F = np.array([[1, dt, 0.5*dt*dt],
                                    [0, 1,          dt],
                                    [0, 0,           1]])
             cafilter.H = np.array([[1., 0, 0]])
             return cafilter

         def initialize_const_accel(f):
```

```
            f.x = np.array([0., 0., 0.])
            f.P = np.eye(3) * 3

In [12]: dt = 0.1
         cafilter = make_ca_filter(dt, sensor_std)
         initialize_const_accel(cafilter)

         kxs2, _, _, _ = cafilter.batch_filter(z_xs2)

         bp.plot_track(xs2, dt=dt)
         bp.plot_filter(kxs2[:, 0], dt=dt, label='KF')
         plt.legend(loc=4)
         bp.set_labels(title='Constant Acceleration Kalman Filter',
                       x='time (sec)', y='X')
```




The constant acceleration model is able to track the maneuver with no lag, but at the cost of very noisy output during the steady state behavior. The noisy output is due to the filter being unable to distinguish between the beginning of an maneuver and noise in the signal. Noise in the signal implies an acceleration, and so the acceleration term of the filter tracks it.

It seems we cannot win. A constant velocity filter cannot react quickly when the target is accelerating, but a constant acceleration filter misinterprets noise during zero acceleration regimes as acceleration instead of nosie.

Yet there is an important insight here that will lead us to a solution. When the target is not maneuvering (the acceleration is zero) the constant velocity filter performs optimally. When the target is maneuvering the constant acceleration filter performs well, as does the constant velocity filter with an artificially large process noise $\mathbf{Q}$. If we make a filter that adapts itself to the behavior of the tracked object we could have the best of both worlds.

## 14.3 Detecting a Maneuver

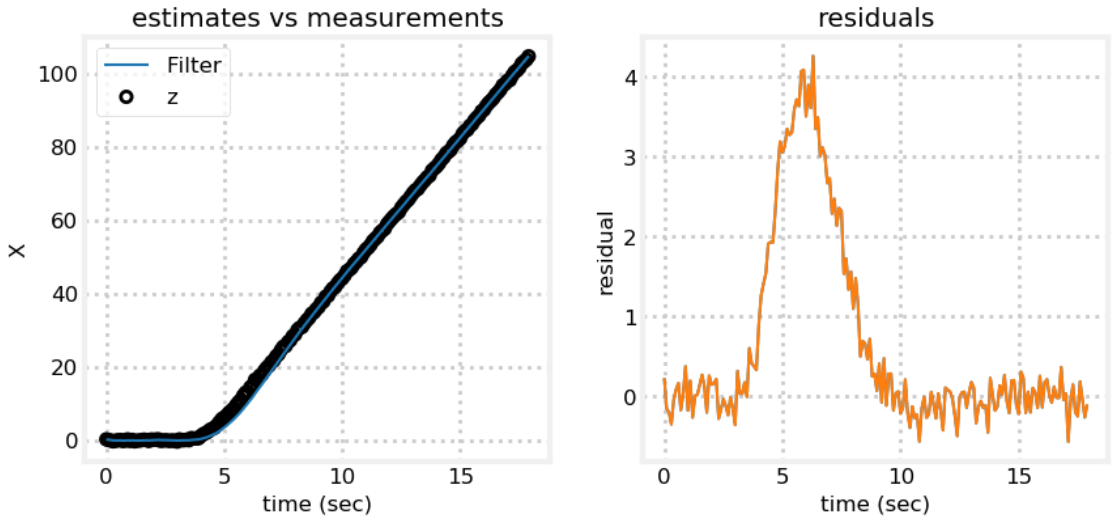
Before we discuss how to create an adaptive filter we have to ask "how do we detect a maneuver?" We cannot reasonably adapt a filter to respond to maneuvers if we do not know when a maneuver is happening.

We have been defining *maneuver* as the time when the tracked object is accelerating, but in general we can say that the object is maneuvering with respect to the Kalman filter if its behavior is different than the process model being used by the filter.

What is the mathematical consequence of a maneuvering object for the filter? The object will be behaving differently than predicted by the filter, so the residual will be large. Recall that the residual is the difference between the current prediction of the filter and the measurement.

To confirm this, let's plot the residual for the filter during the maneuver. I will reduce the amount of noise in the data to make it easier to see the residual.

In [13]:
```
from kf_book.adaptive_internal import plot_track_and_residuals

def show_residual_chart():
    dt = 0.1
    sensor_std = 0.2

    # initialize filter
    cvfilter = make_cv_filter(dt, sensor_std)
    initialize_filter(cvfilter)
    pos2, zs2 = generate_data(150, sensor_std)
    xs2 = pos2[:, 0]
    z_xs2 = zs2[:, 0]

    cvfilter.Q = Q_discrete_white_noise(dim=2, dt=dt, var=0.02)
    xs, res = [], []
    for z in z_xs2:
        cvfilter.predict()
        cvfilter.update([z])
        xs.append(cvfilter.x[0])
        res.append(cvfilter.y[0])

    xs = np.asarray(xs)
    plot_track_and_residuals(dt, xs, z_xs2, res)

show_residual_chart();
```




On the left I have plotted the noisy measurements against the Kalman filter output. On the right I display the residuals computed by the filter - the difference between the measurement and the predictions made by the Kalman filter. Let me emphasize this to make this clear. The plot on the right is not merely the

difference between the two lines in the left plot. The left plot shows the difference between the measurements and the final Kalman filter output, whereas the right plot shows us the difference between the measurements and the *predictions of the process model*.

That may seem like a subtle distinction, but from the plots you see it is not. The amount of deviation in the left plot when the maneuver starts is small, but the deviation in the right plot tells a different story. If the tracked object was moving according to the process model the residual plot should bounce around 0.0. This is because the measurements will be obeying the equation

$$\texttt{measurement} = \texttt{process\_model}(t) + \texttt{noise}(t)$$

Once the target starts maneuvering the predictions of the target behavior will not match the behavior as the equation will be

$$\texttt{measurement} = \texttt{process\_model}(t) + \texttt{maneuver\_delta}(t) + \texttt{noise}(t)$$

Therefore if the residuals diverge from a mean of 0.0 we know that a maneuver has commenced.

We can see from the residual plot that we have our work cut out for us. We can clearly see the result of the maneuver in the residual plot, but the amount of noise in the signal obscures the start of the maneuver. This is our age old problem of extracting the signal from the noise.

## 14.4 Adjustable Process Noise

The first approach we will consider will use a lower order model and adjust the process noise based on whether a maneuver is occurring or not. When the residual gets "large" (for some reasonable definition of large) we will increase the process noise. This will cause the filter to favor the measurement over the process prediction and the filter will track the signal closely. When the residual is small we will then scale back the process noise.

There are many ways of doing this in the literature, I will consider a couple of choices.

### 14.4.1 Continuous Adjustment

The first method (from Bar-Shalom [1]) normalizes the square of the residual using the following equation:

$$\epsilon = \mathbf{y}^\mathsf{T}\mathbf{S}^{-1}\mathbf{y}$$

where $\mathbf{y}$ is the residual and $\mathbf{S}$ is the system uncertainty (covariance), which has the equation

$$\mathbf{S} = \mathbf{H}\mathbf{P}\mathbf{H}^\mathsf{T} + \mathbf{R}$$

If the linear algebra used to compute this confuses you, recall that we can think of matrix inverses in terms of division, so $\epsilon = \mathbf{y}^\mathsf{T}\mathbf{S}^{-1}\mathbf{y}$ can be thought of as computing

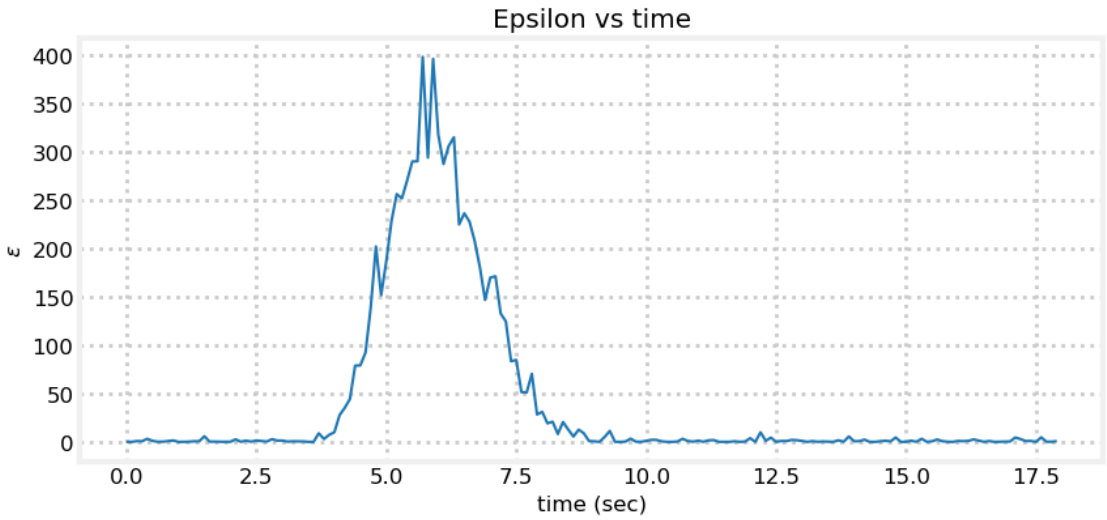
$$\epsilon \approx \frac{\mathbf{y}^2}{\mathbf{S}}$$

Both $\mathbf{y}$ and $\mathbf{S}$ are attributes of `filterpy.KalmanFilter` so implementation will be straightforward.

Let's look at a plot of $\epsilon$ against time.

```
In [14]: from numpy.linalg import inv
         dt = 0.1
         sensor_std = 0.2
         cvfilter= make_cv_filter(dt, sensor_std)
         _, zs2 = generate_data(150, sensor_std)

         epss = []
         for z in zs2[:, 0]:
             cvfilter.predict()
```

```
        cvfilter.update([z])
        y, S = cvfilter.y, cvfilter.S
        eps = np.dot(y.T, inv(S)).dot(y)
        epss.append(eps)

    t = np.arange(0, len(epss) * dt, dt)
    plt.plot(t, epss)
    bp.set_labels(title='Epsilon vs time',
                  x='time (sec)', y='$\epsilon$')
```



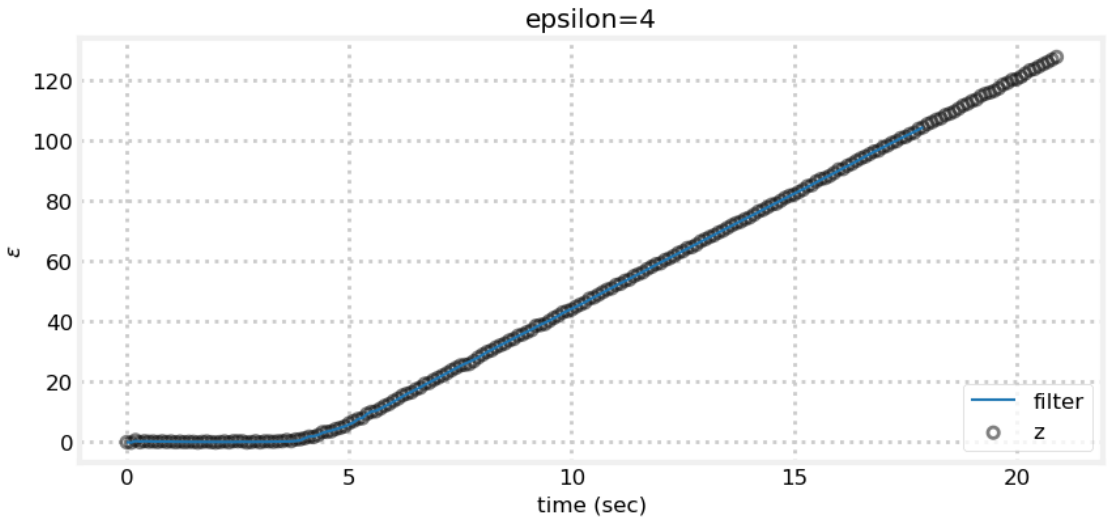

This plot should make clear the effect of normalizing the residual. Squaring the residual ensures that the signal is always greater than zero, and normalizing by the measurement covariance scales the signal so that we can distinguish when the residual is markedly changed relative to the measurement noise. The maneuver starts at t=3 seconds, and we can see that $\epsilon$ starts to increase rapidly not long after that.

We will want to start scaling $\mathbf{Q}$ up once $\epsilon$ exceeds some limit, and back down once it again falls below that limit. We multiply $\mathbf{Q}$ by a scaling factor. Perhaps there is literature on choosing this factor analytically; I derive it experimentally. We can be somewhat more analytical about choosing the limit for $\epsilon$ (named $\epsilon_{max}$) - generally speaking once the residual is greater than 3 standard deviations or so we can assume the difference is due to a real change and not to noise. However, sensors are rarely truly Gaussian and so a larger number, such as 5-6 standard deviations is used in practice.

I have implemented this algorithm using reasonable values for $\epsilon_{max}$ and the $\mathbf{Q}$ scaling factor. To make inspection of the result easier I have limited the plot to the first 10 seconds of simulation.

In [15]:
```
# reinitialize filter
dt = 0.1
sensor_std = 0.2
cvfilter = make_cv_filter(dt, sensor_std)
_, zs2 = generate_data(180, sensor_std)

Q_scale_factor = 1000.
eps_max = 4.

xs, epss = [], []

count = 0
```

```
for i, z in zip(t, zs2[:, 0]):
    cvfilter.predict()
    cvfilter.update([z])
    y, S = cvfilter.y, cvfilter.S
    eps = np.dot(y.T, inv(S)).dot(y)
    epss.append(eps)
    xs.append(cvfilter.x[0])

    if eps > eps_max:
        cvfilter.Q *= Q_scale_factor
        count += 1
    elif count > 0:
        cvfilter.Q /= Q_scale_factor
        count -= 1

bp.plot_measurements(zs2[:,0], dt=dt, label='z', alpha=0.5)
bp.plot_filter(t, xs, label='filter')
plt.legend(loc=4)
bp.set_labels(title='epsilon=4', x='time (sec)', y='$\epsilon$')
```




The performance of this filter is markedly better than the constant velocity filter. The constant velocity filter took roughly 10 seconds to reacquire the signal after the start of the maneuver. The adaptive filter takes under a second to do the same.

### 14.4.2 Continuous Adjustment - Standard Deviation Version

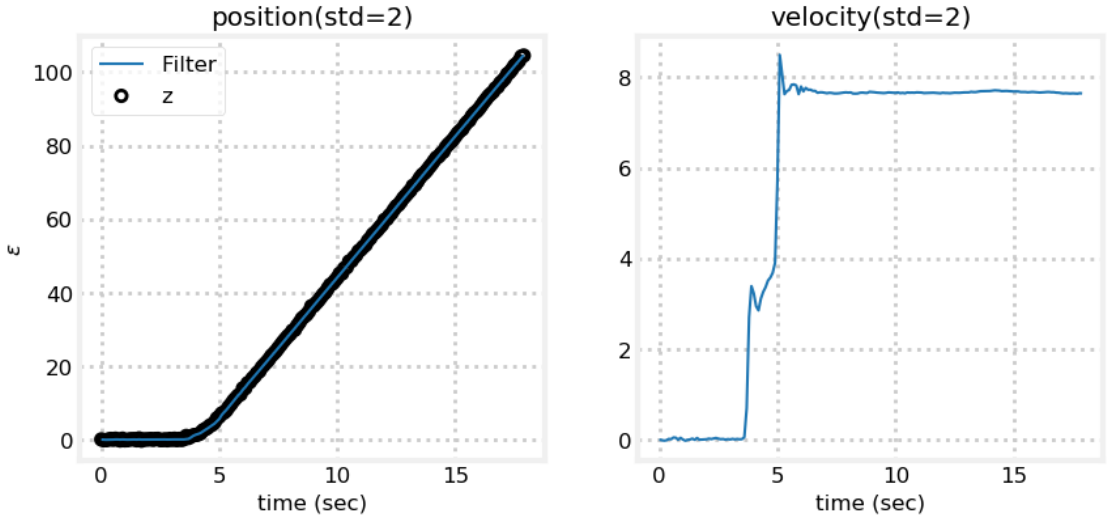
Another, very similar method from Zarchan [2] sets the limit based on the standard deviation of the measurement error covariance. Here the equations are:

$$\begin{aligned} std &= \sqrt{\mathbf{HPH}^\mathsf{T} + \mathbf{R}} \\ &= \sqrt{\mathbf{S}} \end{aligned}$$

If the absolute value of the residual is more than some multiple of the standard deviation computed above we increase the process noise by a fixed amount, recompute Q, and continue.

In [16]:
```
from math import sqrt

def zarchan_adaptive_filter(Q_scale_factor, std_scale,
                            std_title=False,
                            Q_title=False):
    cvfilter = make_cv_filter(dt, std=0.2)
    pos2, zs2 = generate_data(180-30, std=0.2)
    xs2 = pos2[:,0]
    z_xs2 = zs2[:,0]

    # reinitialize filter
    initialize_filter(cvfilter)
    cvfilter.R = np.eye(1)*0.2

    phi = 0.02
    cvfilter.Q = Q_discrete_white_noise(dim=2, dt=dt, var=phi)
    xs, ys = [], []
    count = 0
    for z in z_xs2:
        cvfilter.predict()
        cvfilter.update([z])
        y = cvfilter.y
        S = cvfilter.S
        std = sqrt(S)

        xs.append(cvfilter.x)
        ys.append(y)

        if abs(y[0]) > std_scale*std:
            phi += Q_scale_factor
            cvfilter.Q = Q_discrete_white_noise(2, dt, phi)
            count += 1
        elif count > 0:
            phi -= Q_scale_factor
            cvfilter.Q = Q_discrete_white_noise(2, dt, phi)
            count -= 1

    xs = np.asarray(xs)
    plt.subplot(121)
    bp.plot_measurements(z_xs2, dt=dt, label='z')
    bp.plot_filter(xs[:, 0], dt=dt, lw=1.5)
    bp.set_labels(x='time (sec)', y='$\epsilon$')
    plt.legend(loc=2)
    if std_title:
        plt.title('position(std={})'.format(std_scale))
    elif Q_title:
        plt.title('position(Q scale={})'.format(Q_scale_factor))
    else:
        plt.title('position')

    plt.subplot(122)
    plt.plot(np.arange(0, len(xs)*dt, dt), xs[:, 1], lw=1.5)
    plt.xlabel('time (sec)')
    if std_title:
```

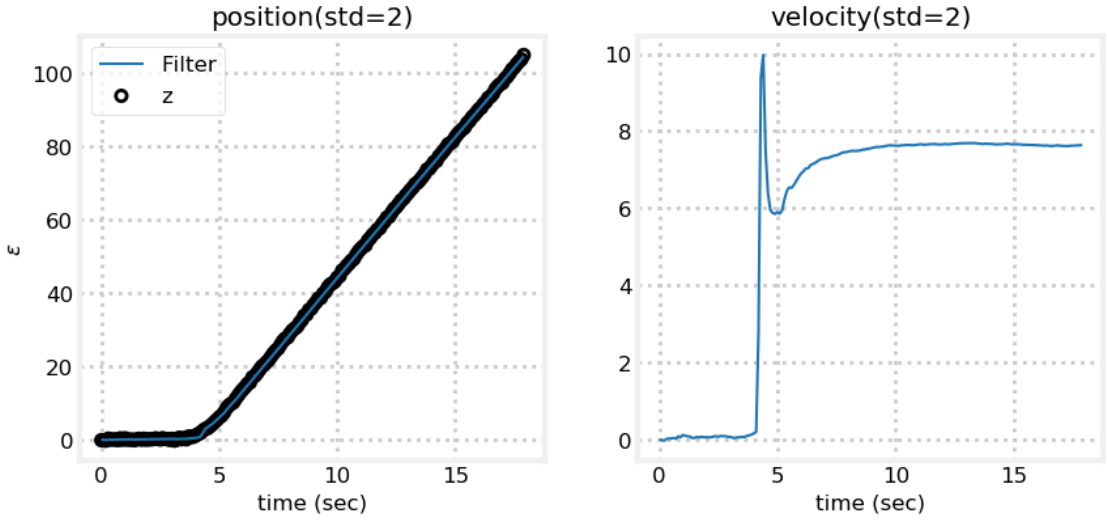
```
        plt.title('velocity(std={})'.format(std_scale))
    elif Q_title:
        plt.title('velocity(Q scale={})'.format(Q_scale_factor))
    else:
        plt.title('velocity')
    plt.show()

zarchan_adaptive_filter(1000, 2, std_title=True)
```




So I chose to use 1000 as the scaling factor for the noise, and 2 as the standard deviation limit. Why these numbers? Well, first, let's look at the difference between 2 and 3 standard deviations.

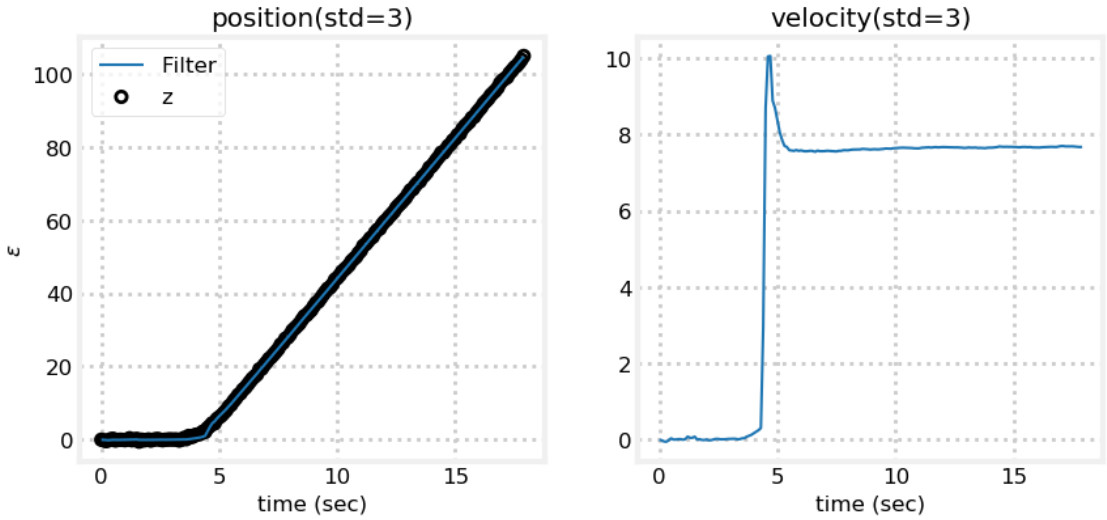
**Two Standard Deviations**

```
In [17]: zarchan_adaptive_filter(1000, 2, std_title=True)
```



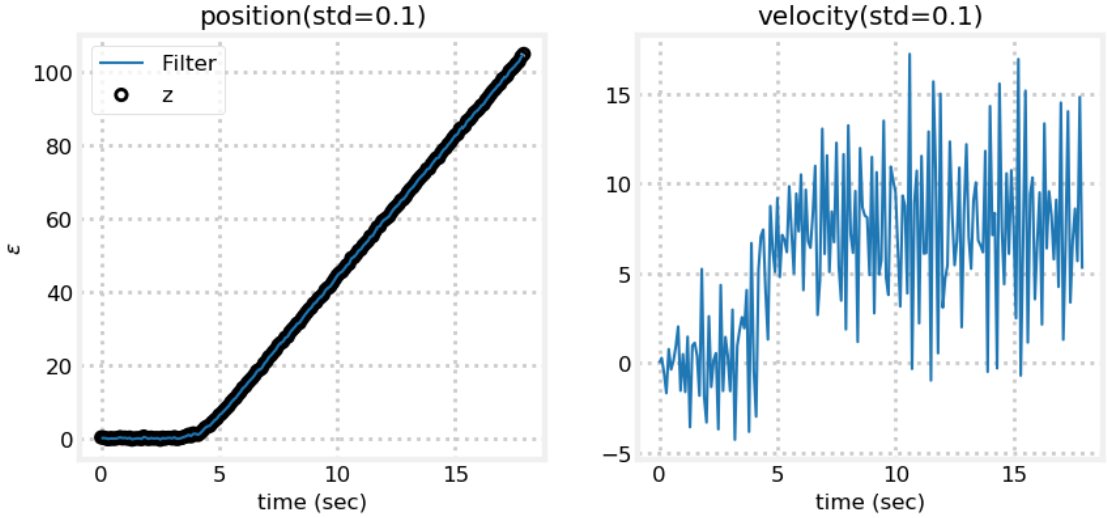

**Three Standard Deviations**

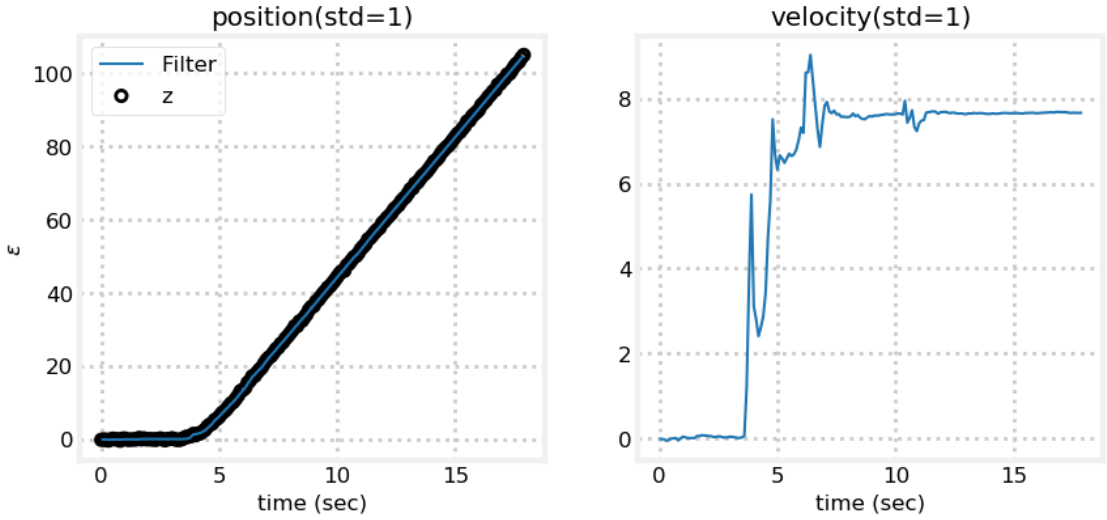
```
In [18]: zarchan_adaptive_filter(1000, 3, std_title=True)
```




We can see from the charts that the filter output for the position is very similar regardless of weather we use 2 standard deviations or three. But the computation of the velocity is a different matter. Let's explore this further. First, let's make the standard deviation very small.

```
In [19]: zarchan_adaptive_filter(1000, .1, std_title=True)
         zarchan_adaptive_filter(1000, 1, std_title=True)
```

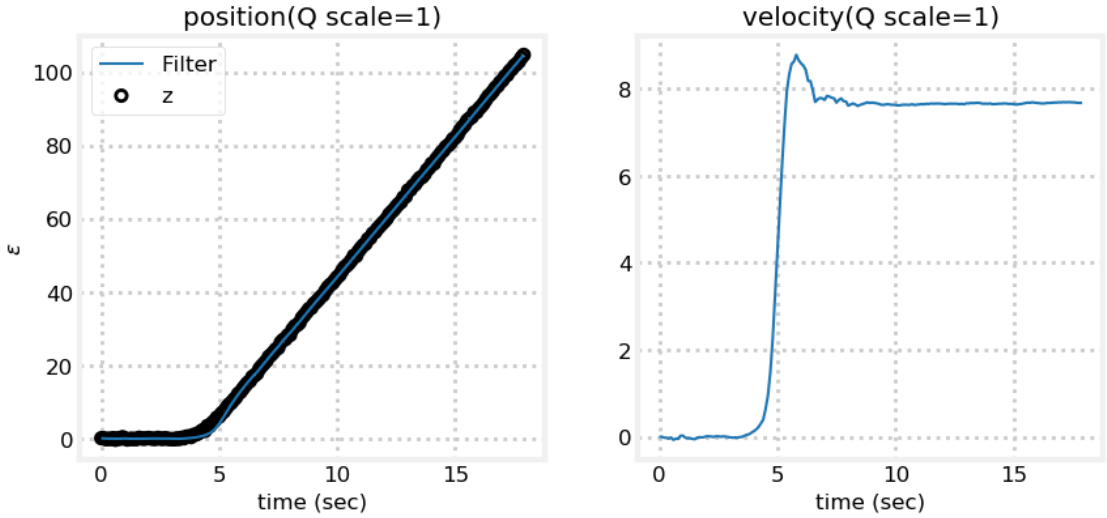


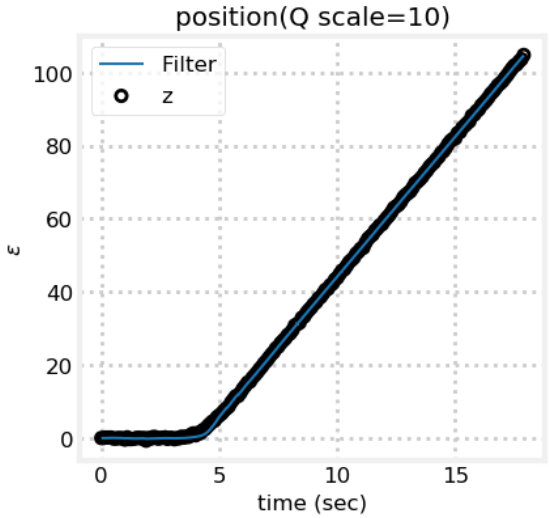


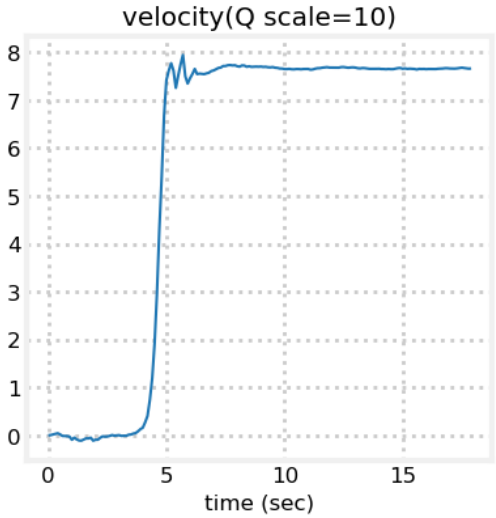

As the standard deviation limit gets smaller the computation of the velocity gets worse. Think about why this is so. If we start varying the filter so that it prefers the measurement over the prediction as soon as the residual deviates even slightly from the prediction we very quickly be giving almost all the weight towards the measurement. With no weight for the prediction we have no information from which to create the hidden variables. So, when the limit is 0.1 std you can see that the velocity is swamped by the noise in the measurement. On the other hand, because we are favoring the measurements so much the position follows the maneuver almost perfectly.

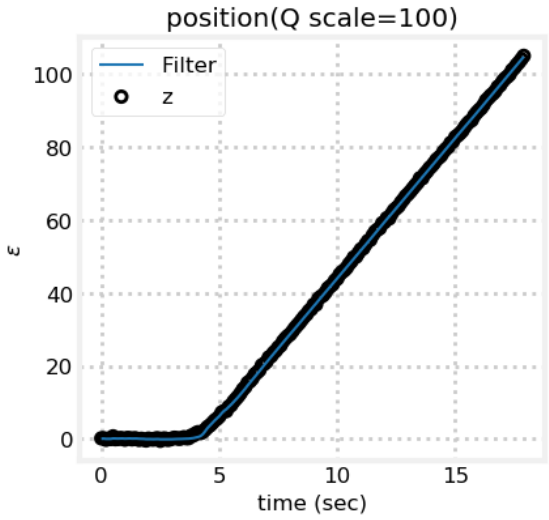
Now let's look at the effect of various increments for the process noise. Here I have held the standard deviation limit to 2 std, and varied the increment from 1 to 10,000.

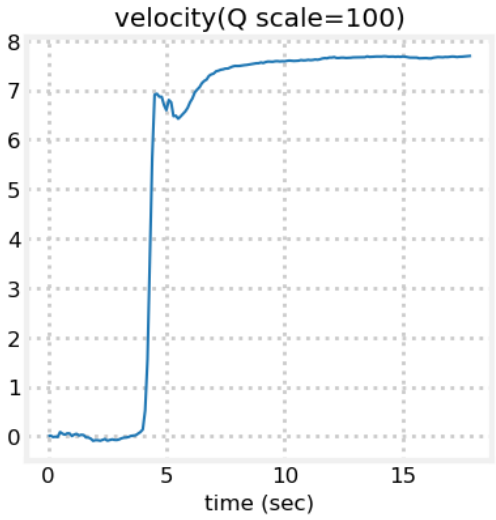
```
In [20]: zarchan_adaptive_filter(1, 2, Q_title=True)
         zarchan_adaptive_filter(10, 2, Q_title=True)
         zarchan_adaptive_filter(100, 2, Q_title=True)
         zarchan_adaptive_filter(1000, 2, Q_title=True)
         zarchan_adaptive_filter(10000, 2, Q_title=True)
```

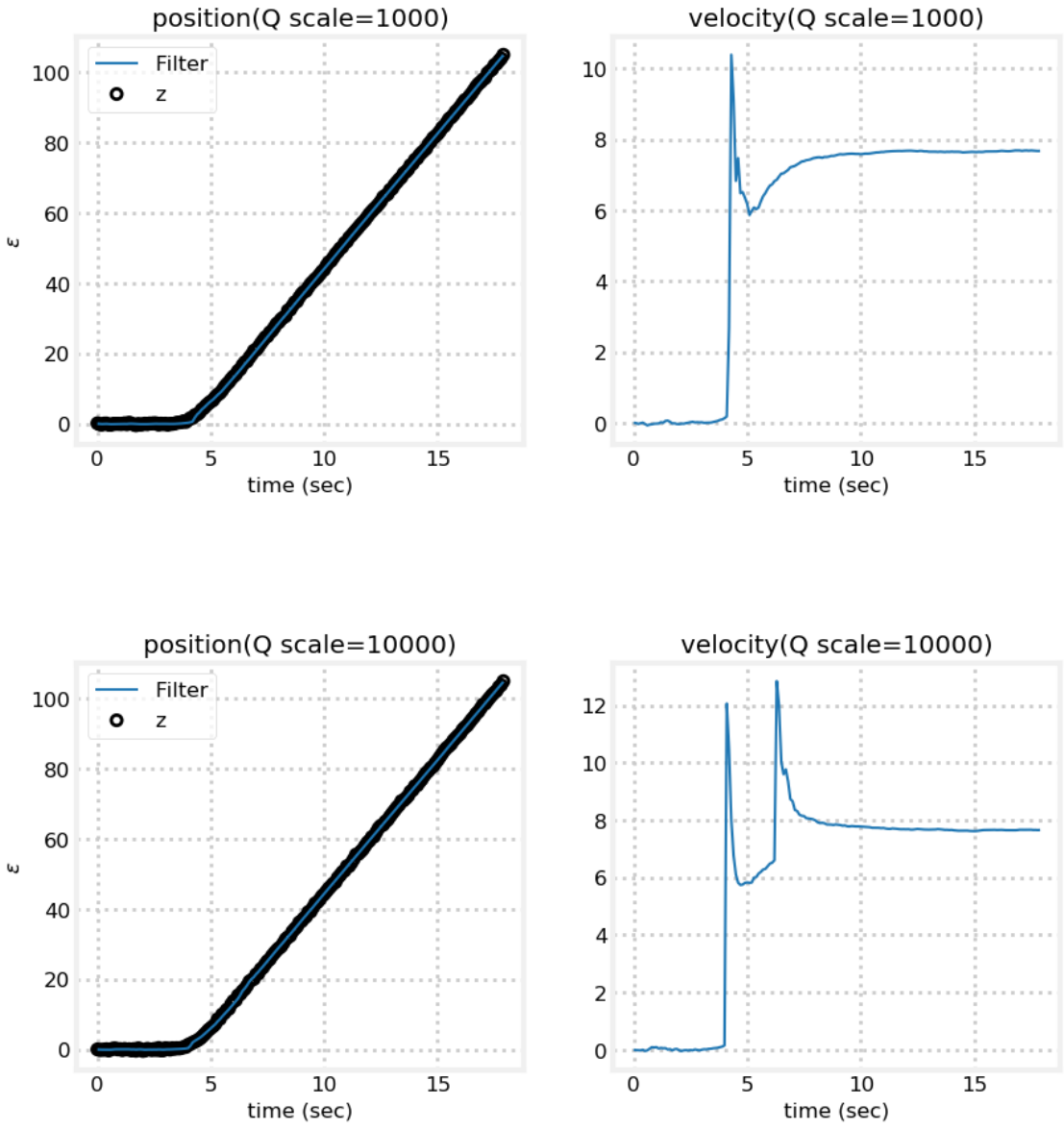



position(Q scale=10)
Filter
z
100
80
60
40
20
0
ε
0
5
10
15
time (sec)


velocity(Q scale=10)
8
7
6
5
4
3
2
1
0
0
5
10
15
time (sec)


position(Q scale=100)
Filter
z
100
80
60
40
20
0
ε
0
5
10
15
time (sec)


velocity(Q scale=100)
8
7
6
5
4
3
2
1
0
0
5
10
15
time (sec)




Here we can see that the position estimate gets marginally better as the increment factor increases, but that the velocity estimate starts to create a large overshoot.

It isn't possible for me to tell you which of these is 'correct'. You will need to test your filter's performance against real and simulated data, and choose the design that best matches the performance you need for each of the state variables.

## 14.5 Fading Memory Filter

Fading memory filters are not normally classified as an adaptive filter since they do not adapt to the the input, but they do provide good performance with maneuvering targets. They also have the benefit of having a very simple computational form for first, second, and third order kinematic filters (e.g. the filters we are using in this chapter). This simple form does not require the Ricatti equations to compute the gain of the Kalman filter, which drastically reduces the amount of computation. However, there is also a form that

works with the standard Kalman filter. I will focus on the latter in this chapter since our focus is more on adaptive filters. Both forms of the fading memory filter are implemented in `FilterPy`.

The Kalman filter is recursive, but it incorporates all of the previous measurements into the current computation of the filter gain. If the target behavior is consistent with the process model than this allows the Kalman filter to find the optimal estimate for every measurement. Consider a ball in flight - we can clearly estimate the position of the ball at time t better if we take into account all the previous measurement. If we only used some of the measurements we would be less certain about the current position, and thus more influenced by the noise in the measurement. If this is still not clear, consider the worst case. Suppose we forget all but the last measurement and estimates. We would then have no confidence in the position and trajectory of the ball, and would have little choice but to weight the current measurement heavily. If the measurement is noisy, the estimate is noisy. We see this effect every time a Kalman filter is initialized. The early estimates are noisy, but then they settle down as more measurements are acquired.

However, if the target is maneuvering it is not always behaving like the process model predicts. In this case remembering all of the past measurements and estimates is a liability. We can see this in all of the charts above. The target initiates a turn, and the Kalman filter continues to project movement in a straight line. This is because the filter has built a history of the target's movement, and incorrectly 'feels' confident that the target is moving in a straight line at a given heading and velocity.

The fading memory filter accounts for this problem by giving less weight to older measurements, and greater weight to the more recent measurements.

There are many formulations for the fading memory filter; I use the one provided by Dan Simon in *Optimal State Estimation* [3]. I will not go through his derivation, but only provide the results.

The Kalman filter equation for the covariances of the estimation error is

$$\bar{\mathbf{P}} = \mathbf{FPF}^\mathsf{T} + \mathbf{Q}$$

We can force the filter to forget past measurements by multiplying a term $\alpha$

$$\tilde{\mathbf{P}} = \alpha^2\mathbf{FPF}^\mathsf{T} + \mathbf{Q}$$

where $\alpha > 1.0$. If $\alpha == 1$ then we get the normal Kalman filter performance. $\alpha$ is an attribute of the `KalmanFilter` class; its value defaults to 1 so the filter acts like a Kalman filter unless $\alpha$ is assigned a value other than 1. There is no hard and fast rule for choosing $\alpha$, but it is typically very close to 1, such as 1.01. You will need to make many runs with either simulated or real data to determine a value that responds to maneuvers without causing the estimate to become too noisy due to overly weighting the noisy measurement.

Why does this work? If we increase the estimate error covariance the filter becomes more uncertain about it's estimate, hence it gives more weight to the measurement.

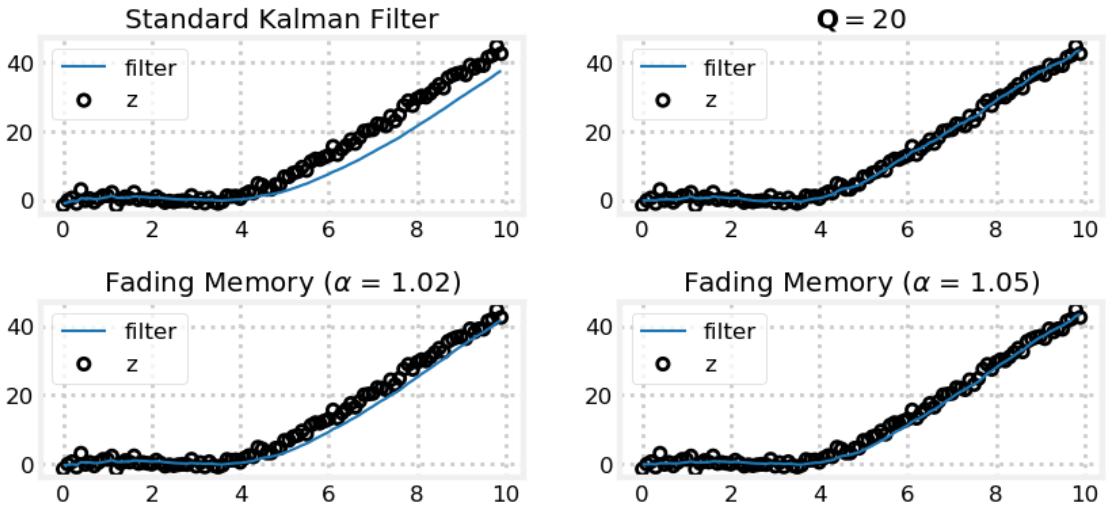
One caveat - if we use $\alpha$ than we are computing $\tilde{\mathbf{P}}$, not $\bar{\mathbf{P}}$. In other words, `KalmanFilter.P` *is not* equal to the covariance of the prior, so do not treat it as if it is.

Let's filter our data using the fading memory filter and see the result. I will inject a lot of error into the system so that we can compare various approaches.

```
In [21]: pos2, zs2 = generate_data(70, std=1.2)
         xs2 = pos2[:, 0]
         z_xs2 = zs2[:, 0]

         cvfilter = make_cv_filter(dt, std=1.2)
         cvfilter.x.fill(0.)
         cvfilter.Q = Q_discrete_white_noise(dim=2, dt=dt, var=0.02)
         cvfilter.alpha = 1.00

         xs, res = [], []
         for z in z_xs2:
             cvfilter.predict()
             cvfilter.update([z])
             xs.append(cvfilter.x[0])
```

```
    res.append(cvfilter.y[0])
xs = np.asarray(xs)
plt.subplot(221)
bp.plot_measurements(z_xs2, dt=dt, label='z')
plt.plot(t[0:100], xs, label='filter')
plt.legend(loc=2)
plt.title('Standard Kalman Filter')

cvfilter.x.fill(0.)
cvfilter.Q = Q_discrete_white_noise(dim=2, dt=dt, var=20.)
cvfilter.alpha = 1.00

xs, res = [], []
for z in z_xs2:
    cvfilter.predict()
    cvfilter.update([z])
    xs.append(cvfilter.x[0])
    res.append(cvfilter.y[0])

xs = np.asarray(xs)

plt.subplot(222)
bp.plot_measurements(z_xs2, dt=dt, label='z')
plt.plot(t[0:100], xs, label='filter')
plt.legend(loc=2)
plt.title('$\mathbf{Q}=20$')

cvfilter.x.fill(0.)
cvfilter.Q = Q_discrete_white_noise(dim=2, dt=dt, var=0.02)
cvfilter.alpha = 1.02

xs, res = [], []
for z in z_xs2:
    cvfilter.predict()
    cvfilter.update([z])
    xs.append(cvfilter.x[0])
    res.append(cvfilter.y[0])
xs = np.asarray(xs)
plt.subplot(223)
bp.plot_measurements(z_xs2, dt=dt, label='z')
plt.plot(t[0:100], xs, label='filter')
plt.legend(loc=2)
plt.title('Fading Memory ($\\alpha$ = 1.02)')

cvfilter.x.fill(0.)
cvfilter.Q = Q_discrete_white_noise(dim=2, dt=dt, var=0.02)
cvfilter.alpha = 1.05

xs, res = [], []
for z in z_xs2:
    cvfilter.predict()
    cvfilter.update([z])
    xs.append(cvfilter.x[0])
    res.append(cvfilter.y[0])
```

```
xs = np.asarray(xs)
plt.subplot(224)
bp.plot_measurements(z_xs2, dt=dt, label='z')
plt.plot(t[0:100], xs, label='filter')
plt.legend(loc=2)
plt.title('Fading Memory ($\\alpha$ = 1.05)');
```




The first plot shows the performance of the Kalman filter. The filter diverges when the maneuver starts and does not reacquire the signal until about 10 seconds. I then made the filter track the maneuver very quickly by making the process noise large, but this has the cost of making the filter estimate very noisy due to unduly weighting the noisy measurements. I then implemented a fading memory filter with $\alpha = 1.02$. The filtered estimate is very smooth, but it does take a few seconds to converge when the target regains steady state behavior. However, the time to do so is considerably smaller than for the Kalman filter, and the amount of lag is much smaller - the estimate for the fading memory is much closer to the actual track than the Kalman filter's track is. Finally, I bumped up $\alpha$ to 1.05. Here we can see that the filter responds almost instantly to the maneuver, but that the estimate is not as straight during the steady state operation because the filter is forgetting the past measurements.

This is quite good performance for such a small change in code! Note that there is no 'correct' choice here. You will need to design your filter based on your needs and the characteristics of the measurement noise, process noise, and maneuvering behavior of the target.

## 14.6 Multiple Model Estimation

The example I have been using in this chapter entails a target moving in a steady state, performing a maneuver, and then returning to a steady state. We have been thinking of this as two models - a constant velocity model, and a constant acceleration model. Whenever you can describe the system as obeying one of a finite set of models you can use *Multiple Model (MM) Estimation*. We use a bank of multiple filters, each using a different process to describe the system, and either switch between them or blend them based on the dynamics of the tracked object.

As you might imagine this is a broad topic, and there are many ways of designing and implementing MM estimators. But consider a simple approach for the target we have been tracking in this chapter. One idea would be to simultaneously run a constant velocity and a constant acceleration filter, and to switch between their outputs when we detect a maneuver by inspecting the residuals. Even this choice gives us many options. Consider the dynamics of a turning object. For example, an automobile turns on a wheelbase

- the front wheels turn, and the car pivots around the rear wheels. This is a nonlinear process, so for best results we would want to use some type of nonlinear filter (EKF, UKF, etc) to model the turns. On the other hand, a linear constant velocity filter would perform fine for the steady state portions of the travel. So our bank of filters might consist of a linear KF and an EKF filter for the turns. However, neither is particularly well suited for modeling behaviors such as accelerating and braking. So a highly performing MM estimator might contain a bank of many filters, each designed to perform best for a certain performance envelope of the tracked object.

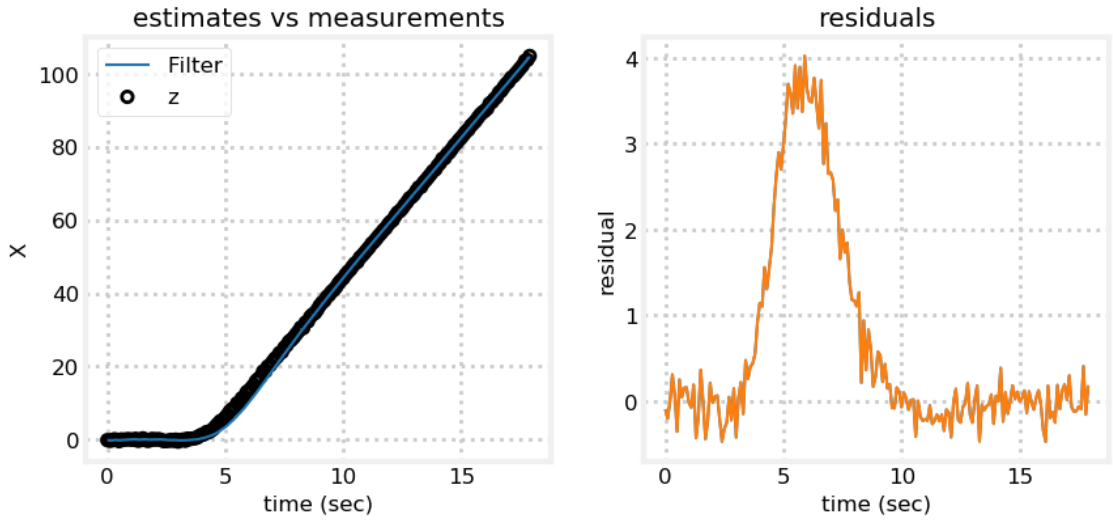
Of course, you do not need to base your filters on the order of the model. You can use different noise models, different adapters in each. For example, in the section above I showed many plots depicting the effects of changing parameters on the estimate of the velocity and position. Perhaps one setting works better for position, and a different setting for velocity. Put both in your bank of filters. You could then take the best estimates for the position from one filter, and the best estimate for the velocity from a different filter.

### 14.6.1 A Two Filter Adaptive Filter

I trust the idea of switching between filters to get the best performance is clear, but what mathematical foundations should we use to implement it? The problem that we face is trying to detect via noisy measurements when a change in regime should result in a change in model. What aspect of the Kalman filter measures how far the measurement deviates from the prediction? Yes, the *residual*.

Let's say we have a first order (constant velocity) Kalman filter. As long as the target is not maneuvering the filter will track it's behavior closely, and roughly 68% of the measurements should fall within $1\sigma$. Furthermore the residual should fluctuate around 0 because as many if the sensor is Gaussian an equal number of measurement should have positive error as have negative errors. If the residual grows and stays beyond predicted bounds then the target must not be performing as predicted by the state model. We saw this earlier in this chart where the residual switched from bouncing around 0 to suddenly jumping and staying above zero once the tracked object began maneuvering.

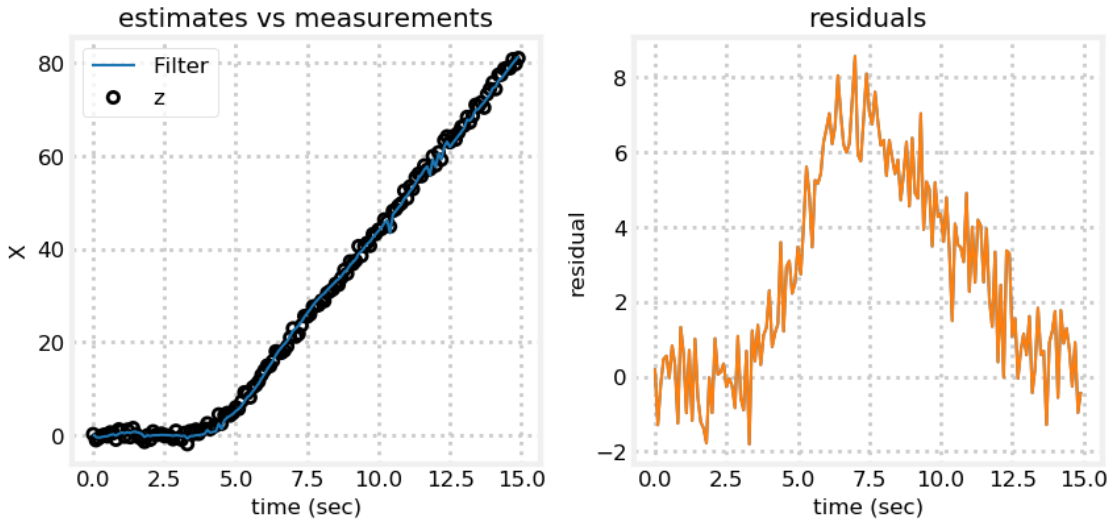
```
In [22]: show_residual_chart()
```




For this problem we saw that the constant velocity filter performed better the constant acceleration filter while the object was in steady state, and the opposite was true when the object is maneuvering. In the chart above that transition occurs at 4 seconds.

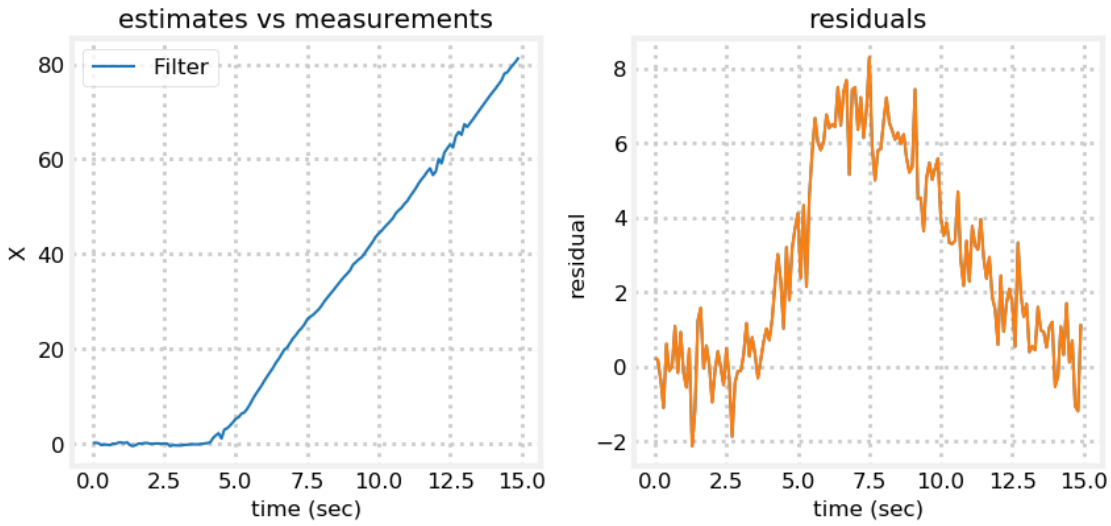
So the algorithm is easy. Initialize both a constant velocity and constant acceleration filter and run them together in a predict/update loop. After every update examine the residual of the constant velocity filter. If it falls within theoretical bounds use the estimate from the constant velocity filter as the estimate, otherwise use the estimate from the constant acceleration filter.

```
In [23]: def run_filter_bank(threshold, show_zs=True):
             dt = 0.1
             cvfilter= make_cv_filter(dt, std=0.8)
             cafilter = make_ca_filter(dt, std=0.8)
             pos, zs = generate_data(120, std=0.8)
             z_xs = zs[:, 0]
             xs, res = [], []

             for z in z_xs:
                 cvfilter.predict()
                 cafilter.predict()
                 cvfilter.update([z])
                 cafilter.update([z])

                 std = np.sqrt(cvfilter.R[0,0])
                 if abs(cvfilter.y[0]) < 2 * std:
                     xs.append(cvfilter.x[0])
                 else:
                     xs.append(cafilter.x[0])
                 res.append(cvfilter.y[0])
             xs = np.asarray(xs)
             if show_zs:
                 plot_track_and_residuals(dt, xs, z_xs, res)
             else:
                 plot_track_and_residuals(dt, xs, None, res)

         run_filter_bank(threshold=1.4)
```




Here the filter tracks the maneuver closely. While the target is not maneuvering our estimate is nearly noise free, and then once it does maneuver we quickly detect that and switch to the constant acceleration filter. However, it is not ideal. Here is the filter output plotted alone:

```
In [24]: run_filter_bank(threshold=1.4, show_zs=False)
```




You can see that the estimate jumps when the filter bank switches from one filter to the other. I would not use this algorithm in a production system. The next section gives a state of the art implementation of a filter bank that eliminates this problem.

## 14.7 MMAE

The core idea of using several filters to detect a maneuver is sound, but the estimate is jagged when we abruptly transition between the filters. Choosing one filter over the other flies in the face of this entire book, which uses probability to determine the *likelihood* of measurements and models. We don't choose *either* the measurement or prediction, depending on which is more likely, we choose a *blend* of the two in proportion to their likelihoods. We should do the same here. This approach is called the *Multiple Model Adaptive Estimator*, or MMAE.

In the **Designing Kalman Filters** chapter we learned the *likelihood function*

$$\mathcal{L} = \frac{1}{\sqrt{2\pi S}} \exp[-\frac{1}{2}\mathbf{y}^\mathsf{T}\mathbf{S}^{-1}\mathbf{y}]$$

which tells us how likely a filter is to be performing optimally given the inputs. $\mathbf{y}$ is the residual and $\mathbf{S}$ is the system uncertainty (covariance in measurement space). This is just a Gaussian of the residual and the system uncertainty. A large residual will give a large uncertainty, and thus low likelihood that the measurement matches the filter's current state. We can use this to compute the probability that each filter is the best fit to the data. If we have N filters, we can compute the probability that filter i is correct in relation to the rest of the filters with

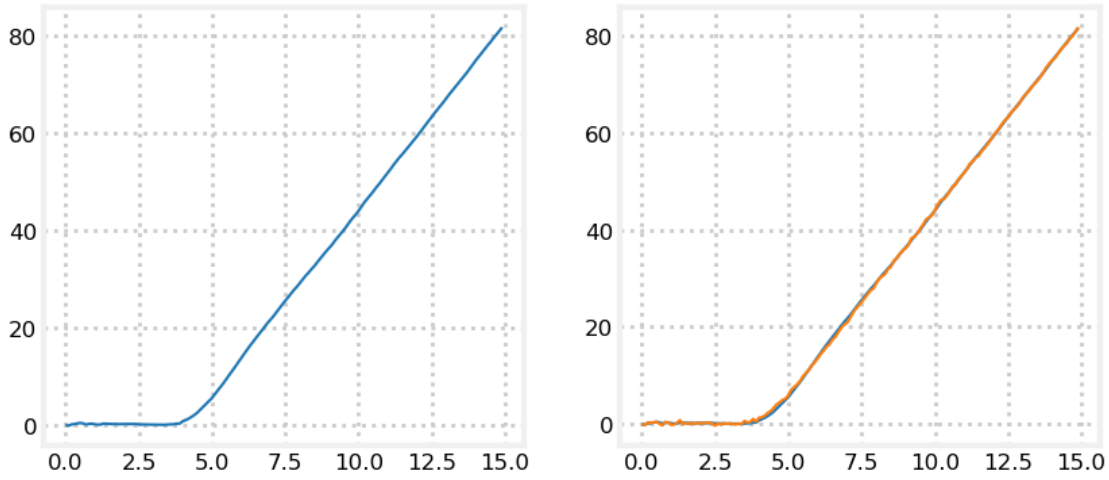
$$p_k^i = \frac{\mathcal{L}_k^i p_{k-1}^i}{\sum_{j=1}^{N} \mathcal{L}_k^j p_{k-1}^j}$$

That looks messy, but it is straightforward. The numerator is just the likelihood from this time step multiplied by the probability that this filter was correct at the last time frame. We need all of the probabilities for the filter to sum to one, so we normalize by the probabilities for all of the other filters with the term in the denominator.

That is a recursive definition, so we need to assign some initial probability for each filter. In the absence of better information, use $\frac{1}{N}$ for each. Then we can compute the estimated state as the sum of the state from each filter multiplied the the probability of that filter being correct.

Here is a complete implementation:

```
In [25]: def run_filter_bank():
             dt = 0.1
             cvfilter = make_cv_filter(dt, std=0.2)
             cafilter = make_ca_filter(dt, std=0.2)

             _, zs = generate_data(120, std=0.2)
             z_xs = zs[:, 0]
             xs, probs = [], []

             pv, pa = 0.8, 0.2
             pvsum, pasum = 0., 0.

             for z in z_xs:
                 cvfilter.predict()
                 cafilter.predict()
                 cvfilter.update([z])
                 cafilter.update([z])

                 cv_likelihood = cvfilter.likelihood * pv
                 ca_likelihood = cafilter.likelihood * pa

                 pv = (cv_likelihood) / (cv_likelihood + ca_likelihood)
                 pa = (ca_likelihood) / (cv_likelihood + ca_likelihood)

                 x = (pv * cvfilter.x[0]) + (pa*cafilter.x[0])
                 xs.append(x)
                 probs.append(pv / pa)

             xs = np.asarray(xs)
             t = np.arange(0, len(xs) * dt, dt)
             plt.subplot(121)
             plt.plot(t, xs)
             plt.subplot(122)
             plt.plot(t, xs)
             plt.plot(t, z_xs)
             return xs, probs

         xs, probs = run_filter_bank()
```




I plot the filter's estimates alone on the left so you can see how smooth the result is. On the right I plot both the estimate and the measurements to prove that the filter is tracking the maneuver.

Again I want to emphasize that this is nothing more than the Bayesian algorithm we have been using throughout the book. We have two (or more) measurements or estimate, each with an associated probability. We choose are estimate as a weighted combination of each of those values, where the weights are proportional to the probability of correctness. The computation of the probability at each step is

$$\frac{\texttt{Prob(meas | state)} \times \texttt{prior}}{\texttt{normalization}}$$

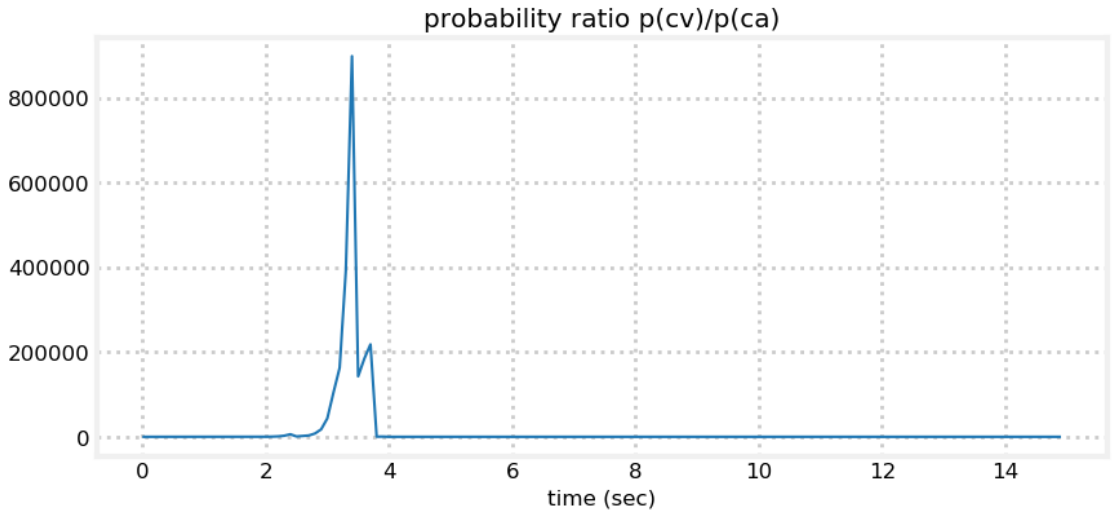
which is Bayes therom.

For real world problems you are likely to need more than two filters in your bank. In my job I track objects using computer vision. I track hockey pucks. Pucks slide, they bounce and skitter, they roll, they ricochet, they are picked up and carried, and they are 'dribbled' quickly by the players. I track humans who are athletes, and their capacity for nonlinear behavior is nearly limitless. A two filter bank doesn't get very far in those circumstances. I need to model multiple process models, different assumptions for noise due to the computer vision detection, and so on. But you have the main idea.

### 14.7.1 Limitations of the MMAE Filter

The MMAE as I have presented it has a significant problem. Look at this chart of the ratio of the probability for the constant velocity vs constant acceleration filter.

```
In [26]: plt.plot(t[0:len(probs)], probs)
         plt.title('probability ratio p(cv)/p(ca)')
         plt.xlabel('time (sec)');
```




For the first three seconds, while the tracked object travels in a straight direction, the constant velocity filter become much more probable than the constant acceleration filter. Once the maneuver starts the probability quickly changes to to favor the constant acceleration model. However, the maneuver is completed by second six. You might expect that the probability for the constant velocity filter would once again become large, but instead it remains at zero.

This happens because of the recursive computation of the probability:

$$p_k = \frac{\mathcal{L}p_{k-1}}{\sum \text{probabilities}}$$

Once the probability becomes very small it can never recover. The result is that the filter bank quickly converges on only the most probable filters. A robust scheme needs to monitor the probability of each filter and kill off the filters with very low probability and replace them with filters with greater likelihood of performing well. You can subdivide the existing filters into new filters that try to span the characteristics that make them perform well. In the worst case, if a filter has diverged you can reinitialize a filter's state so that it is closer to the current measurements.

## 14.8 Interacting Multiple Models (IMM)

Let's think about multiple models in another way. The scenario is as before - we wish to track a maneuvering target. We can design a set of Kalman filters which make different modeling assumptions. They can differ in terms of the filter order, or in the amount of noise in the process model. As each new measurement comes in each filter has a probability of being the correct model.

This naive approach leads to combinatorial explosion. At step 1 we generate $N$ hypotheses, or 1 per filter. At step 2 we generate another $N$ hypotheses which then need to be combined with the prior $N$ hypotheses, which yields $N^2$ hypothesis. Many different schemes have been tried which either cull unlikely hypotheses or merge similar ones, but the algorithms still suffered from computational expense and/or poor performance. I will not cover these in this book, but prominent examples in the literature are the generalized pseudo Bayes (GPB) algorithms.

The *Interacting Multiple Models* (IMM) algorithm was invented by Blom[5] to solve the combinatorial explosion problem of multiple models. A subsequent paper by Blom and Bar-Shalom is the most cited paper [6]. The idea is to have 1 filter for each possible mode of behavior of the system. At each epoch we we let the filters *interact* with each other. The more likely filters modify the estimates of the less likely filters so they more nearly represent the current state of the sytem. This blending is done probabilistically, so the unlikely filters also modify the likely filters, but by a much smaller amount.

For example, suppose we have two modes: going straight, or turning. Each mode is represented by a Kalman filter, maybe a first order and second order filter. Now say the target it turning. The second order filter will produce a good estimate, and the first order filter will lag the signal. The likelihood function of each tells us which of the filters is most probable. The first order filter will have low likelihood, so we adjust its estimate greatly with the second order filter. The the second order filter is very likely, so its estimate will only be changed slightly by the first order Kalman filter.

Now suppose the target stops turning. Because we have been revising the first order filter's estimate with the second order estimate it will not have been lagging the signal by very much. within just a few epochs it will be producing very good (high likelihood) estimates and be the most probable filter. It will then start contributing heavily to the estimate of the second order filter. Recall that a second order filter mistakes measurement noise for acceleration. This adjustment insures reduces this effect greatly.

### 14.8.1 Mode Probabilities

We define a set of modes for our system, $m$, and assume that the target is always in one of these modes. In the discussion above we have the modes straight and turn, so $m = \{\text{straight},\ \text{turn}\}$.

We assign a probability that the target is in any given mode. This gives us a vector of *mode probabilities* with one probability for each possible mode. $m$ has two modes, so we will have a vector of two probabilities. If we think that there is a 70% chance that the target is going straight we can say

$$\mu = \begin{bmatrix}0.7 & 0.3\end{bmatrix}$$

We get 0.3 for the turn because the probabilities must sum to one. $\mu$ is typically but not universally used as the symbol for the mode probabilities, so I will use it. Do not confuse it with the mean.

In Python we can implement this as

```
In [27]: mu = np.array([0.7, 0.3])
         mu

Out[27]: array([0.7, 0.3])
```

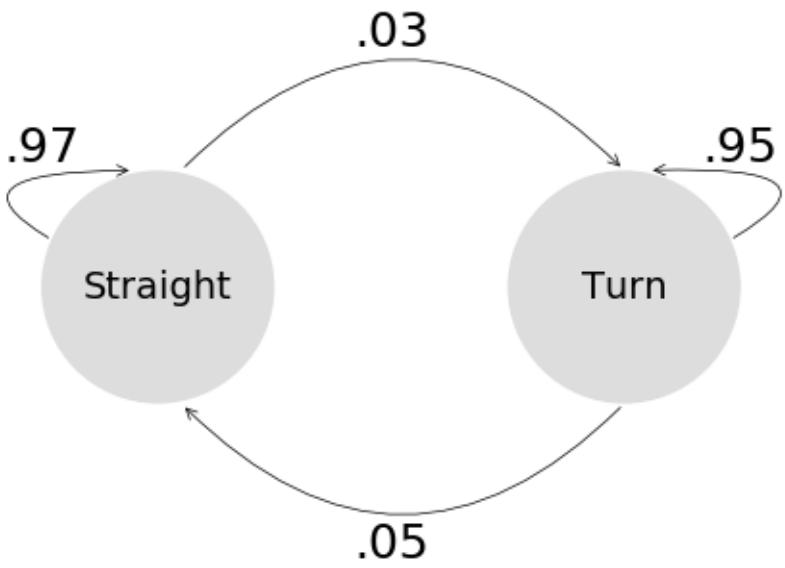
We can formalize it by saying that the prior probability that $m_i$ is correct (the maneuvering object is in mode $i$), given the prior measurements $Z$, is

$$\mu_i = P(m_i|Z)$$

### 14.8.2 Mode Transitions

Next we have to consider that this is a maneuvering target. It will go straight, then turn, then go straight again. We can model the transition between these modes as a *Markov chain*, as in this illustration:

```
In [28]: import kf_book.adaptive_internal as adaptive_internal
         adaptive_internal.plot_markov_chain()
```




This shows an example of two modes for a target, going straight and performing a turn. If the current mode of the target is straight, then we predict that there is a 97% chance of the target continuing straight, and a 3% chance of starting a turn. Once the target is turning, we then predict that there is a 95% chance of staying in the turn, and a 5% of returning to a straight path.

The algorithm is not sensitive to the exact numbers, and you will typically use simulation or trials to choose appropriate values. However, these values are quite representative.

We represent Markov chains with a *transition probability matrix*, which we will call $\mathbf{M}$. For the Markov chain in the illustration we would write

$$\mathbf{M} = \begin{bmatrix} .97 & .03 \\ .05 & .95 \end{bmatrix}$$

In other words $\mathbf{M}[i, j]$ is the probability of mode being $j$ given that the last mode was $i$. In this example the probability of the mode currently being straight ($j = 0$) given that the last mode was a turn ($i = 1$) is $\mathbf{M}[1,\ 0] = 0.05$. In Python we'd write:

```
In [29]: M = np.array([[.97, .03], [.05, .95]])
         print(M)
         print('From turn to straight probablility is', M[1, 0], 'percent')
```

```
[[0.97 0.03]
 [0.05 0.95]]
From turn to straight probablility is 0.05 percent
```

This allows us to compute the new mode probabilities based on the probability of a transition. Let's compute the probability of the mode being straight after a transition. There are two ways for us to be

moving straight. We could have been moving straight, and then continued straight, or we could have been turning, but then went straight. The former probability is calculated with $(0.7 \times 0.97)$ and the latter with $(0.3 \times 0.05)$. We are multiplying the mode probability with the relevant probability from the Markov Chain. The *total probability* is the sum of the two, or $(0.7)(0.97) + (0.3)(0.05) = 0.694$.

Recall the *total probability theorem* from the second chapter. It states that the probability of several distinct events is

$$P(A) = \sum P(A \mid B)\, P(B)$$

Here $P(A \mid B)$ is the transition matrix $\mathbf{M}$ and $P(B)$ is $\mu$. We are using arrays and matrices, and so we take advantage of the fact that a vector times a matrix computes the sum of products:

$$\begin{bmatrix}\mu_1 & \mu_2\end{bmatrix} \begin{bmatrix}m_{11} & m_{12} \\ m_{21} & m_{22}\end{bmatrix} = \begin{bmatrix}\mu_1 m_{11} + \mu_2 m_{21} & \mu_1 m_{12} + \mu_2 m_{22}\end{bmatrix}$$

The IMM literature expresses this as

$$\bar{c}_j = \sum_{i=1}^{N} \mu_i M_{ij}$$

We use NumPy's `dot` function to compute this for us:

```
In [30]: cbar = np.dot(mu, M)
         cbar
```

```
Out[30]: array([0.694, 0.306])
```

### 14.8.3 Computing the Mode Probabilities

We will compute the new mode probabilities using Bayes theorem. Recall that Bayes theorem states

$$\text{posterior} = \frac{\text{prior} \cdot \text{likelihood}}{\text{normalization factor}}$$

Here the prior is the total probability computation we performed in the last section. The Kalman filter computes the *likelihood*, which is the likelihood of the measurements given the current state of the filter. For review the equation is:

$$\mathcal{L} = \frac{1}{\sqrt{2\pi \mathbf{S}}} \exp[-\frac{1}{2}\mathbf{y}^\mathsf{T}\mathbf{S}^{-1}\mathbf{y}]$$

In mathematical notation the updated mode probability is:

$$\mu_i = \|\mathcal{L}_i \bar{c}_i\|$$

In words, for each Kalman filter (mode) we compute the mode probability as the probability of the current mode taking the possible transition into account times the likelihood that this is the correct mode. Then we normalize all of the probabilities so they sum to one.

This is trivial to compute in Python. I'll introduce the variable `L` to store the likelihoods. Likelihoods are computed by the `KalmanFilter.update()` step, and in the code snippet below I just hard coded values for `L` since we haven't created the Kalman filters yet:

```
In [31]: # L = [kf0.L, kf1.L]  # get likelihoods from Kalman filters
         L = [0.000134, 0.0000748]
         mu = cbar * L
         mu /= sum(mu) # normalize
         mu
```

```
Out[31]: array([0.802, 0.198])
```

Here you can see that the relatively strong likelihood for the straight filter pushed the probability for the straight mode from 70% to 80.2%.

## 14.9 Mixing Probabilities

At this point we could use the mode transitions to compute the probabilities for all possible choices. If $\mu = \begin{bmatrix}0.63 & 0.27\end{bmatrix}$, then we can use the transition probability matrix to compute all possible outcomes. In other words, if the current mode is straight ($\mu = 0.63$), we can compute two new probabilities based on whether the target keeps moving straight, or turns. We do the same for the turning mode ($\mu = 0.27$). We will have gone from 2 mode probabilities to 4. At the next step 4 will turn into 8, and so on. It's computationally exact, but infeasible in practice. After only 30 epochs you'd require 8GB of memory to store the mode probabilities in double precision.

We need a better, albeit approximate way. IMMs resolve this by computing *mixing probabilities*. The idea is simple. Let's say the first mode (straight) is currently very likely, and the second mode (turn) is unlikely. Instead of having the Kalman filter for the straight mode compute its state as the weighted average of all of the filters in the filter bank. Filters with high probability of matching the target's mode get weighted more than filters with lower probability. The result is that the information from the probable filters improve the accuracy of the filters that are improbable. This is the crux of the algorithm.

What we need to do is very simple. Each Kalman filter performs the update step, computing a new mean and covariance. But then we compute a new mean and covariance for each filter as a weighted sum of these means and covariances according to *mixing probabilities* which we call $\omega$. Likely filters will be slightly adjusted by the unlikely filters, and the unlikely filters will be strongly adjusted by the likely ones. The literature calls these adjusted means and covariances either the *mixed conditions* or *mixed initial conditions*. I use the notation $\mathbf{x}_j^m$ for the mixed state, and $\mathbf{P}_j^m$ for the mixed covariance. The equations are:

$$\mathbf{x}_j^m = \sum_{i=1}^N \omega_{ij}\mathbf{x}_i$$
$$\mathbf{P}_j^m = \sum_{i=1}^N \omega_{ij}\left[(\mathbf{x}^i - \mathbf{x}_i^m)(\mathbf{x}^i - \mathbf{x}_i^m)^\mathsf{T} + \mathbf{P}_i\right]$$

Just think of the subscripts as indexes into arrays. Putting it in pseudo-Python we can write this as:

```
for j in N:
    x0[j] = sum_over_i(w[i,j] * x[i])
    P0[j] = sum_over_i(w[i, j] * (P[i] + np.outer(x[i] - x0[j])))
```

Don't let the notation confuse what is a simple idea: incorporate estimates from the probable filters into the estimates of the improbable filters, ensuring all have a good estimate.

How do we compute the mixing probabilities? Think about it, and try to give a reasonable answer before reading on. We have mode probabilities which describe the current probability of each mode, and then transition probabilities describing how likely we are to change modes. How do we compute the new probability?

Bayes theorem, of course! Prior times the likelihood, normalized. The prior is the mode probability, and the likelihood comes from the Markov chain, which we store in the matrix $\mathbf{M}$.

$$\boldsymbol{\omega}_{ij} = \|\mu_i \cdot \mathbf{M}_{ij}\|$$

We can compute this as follows. I computed the update of $\mu$ and $\bar{c}$ out of order above (ou must compute $\bar{c}$ incorporating the transition probability matrix into $\mu$), so I'll need to correct that here:

```
In [32]: cbar = np.dot(mu, M) #compute total probability that target is in mode j

         omega = np.zeros((2, 2))
         for i in range(2):
             for j in range(2):
                 omega[i, j] = (M[i, j] * mu[i]) / cbar[j]
         omega
```

```
Out[32]: array([[0.987, 0.114],
                [0.013, 0.886]])
```

The Kalman filters need to perform the prediction step to compute the new prior. They use the mixed estimates:

$$\bar{\mathbf{x}}_j = \mathbf{F}_j\mathbf{x}_j^m$$
$$\bar{\mathbf{P}}_j = \mathbf{F}_j\mathbf{P}_j^m\mathbf{F}_j^\mathsf{T} + \mathbf{Q}_j$$

### 14.9.1 IMM Estimate

Now we need a final state estimate from the bank of filters. How do we do that? Just weight the mixed estimate from each Kalman filter:

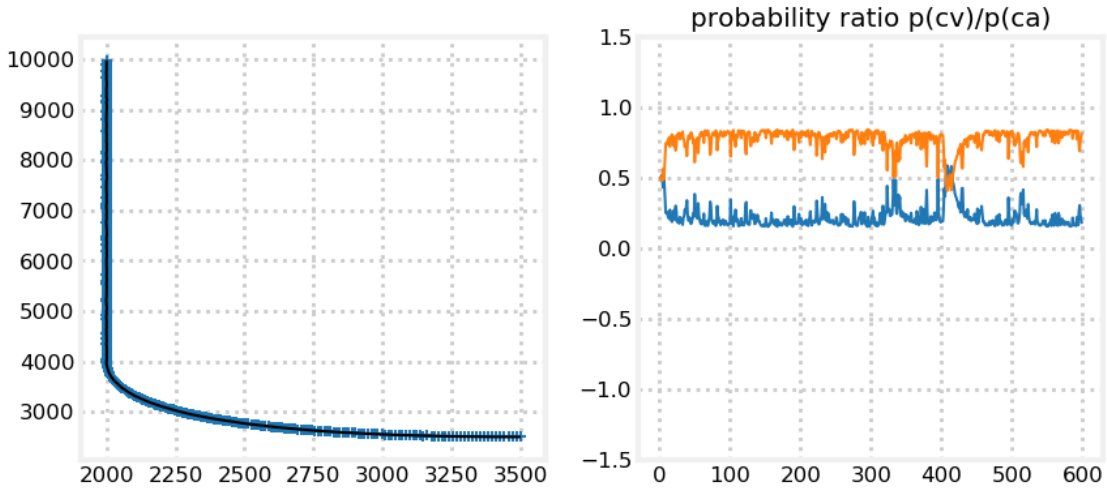
$$\mathbf{x} = \sum_{j=1}^{N} \mu_j\bar{\mathbf{x}}_j$$
$$\mathbf{P} = \sum_{j=1}^{N} \mu_j \left[(\bar{\mathbf{x}}_j - \bar{\mathbf{x}})(\bar{\mathbf{x}}_j - \bar{\mathbf{x}})^\mathsf{T} + \bar{\mathbf{P}}_j\right]$$

### 14.9.2 Tracking Maneuvering Target with the IMM

Let's work an example. Crassidis[4] is one of the few texts with a worked example, so I have chosen his example. He tracks a moving target for 600 seconds. The target starts off moving straight, and then a control input is injected starting at 400 seconds, causing the target to make a 90 degree turn. He uses two constant acceleration Kalman filters. One filter assumes no process noise, and the other assumes process noise with spectral density $10^{-3}\mathbf{I}$. He assumes very good initialization of the filters, setting $\mathbf{P} = 10^{-12}$ for both filters. My implementation follows:

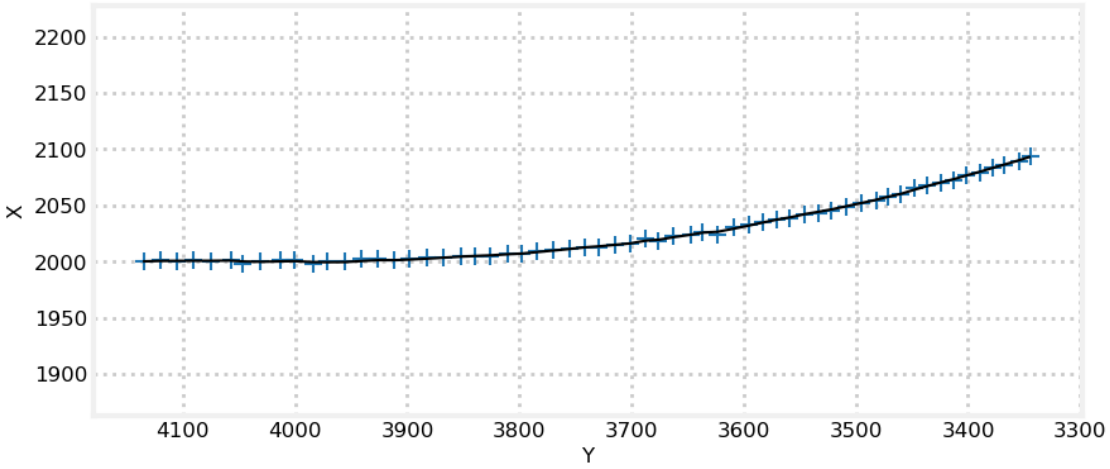
```
In [33]: import copy
         from scipy.linalg import block_diag
         from filterpy.kalman import IMMEstimator
         from filterpy.common import Saver

         N = 600
         dt = 1.
         imm_track = adaptive_internal.turning_target(N)

         # create noisy measurements
         zs = np.zeros((N, 2))
         r = 1
         for i in range(N):
             zs[i, 0] = imm_track[i, 0] + randn()*r
             zs[i, 1] = imm_track[i, 2] + randn()*r

         ca = KalmanFilter(6, 2)
         dt2 = (dt**2)/2
         F = np.array([[1, dt, dt2],
                       [0,  1,  dt],
                       [0,  0,   1]])

         ca.F = block_diag(F, F)
         ca.x = np.array([[2000., 0, 0, 10000, -15, 0]]).T
         ca.P *= 1.e-12
```

```
ca.R *= r**2
q = np.array([[.05, .125, 1/6],
              [.125, 1/3, .5],
              [1/6, .5, 1]])*1.e-3
ca.Q = block_diag(q, q)
ca.H = np.array([[1, 0, 0, 0, 0, 0],
                 [0, 0, 0, 1, 0, 0]])

# create identical filter, but with no process error
cano = copy.deepcopy(ca)
cano.Q *= 0

filters = [ca, cano]

M = np.array([[0.97, 0.03],
              [0.03, 0.97]])
mu = np.array([0.5, 0.5])
bank = IMMEstimator(filters, mu, M)

xs, probs = [], []
for i, z in enumerate(zs):
    z = np.array([z]).T
    bank.predict()
    bank.update(z)

    xs.append(bank.x.copy())
    probs.append(bank.mu.copy())

xs = np.array(xs)
probs = np.array(probs)
plt.subplot(121)
plt.plot(xs[:, 0], xs[:, 3], 'k')
plt.scatter(zs[:, 0], zs[:, 1], marker='+')

plt.subplot(122)
plt.plot(probs[:, 0])
plt.plot(probs[:, 1])
plt.ylim(-1.5, 1.5)
plt.title('probability ratio p(cv)/p(ca)');
```




It is rather hard to see the performance of the filter, so let's look at the performance just as the turn starts. I've swapped the $x$ and $y$ axis to let us zoom in closely. In the chart below the turn starts at $Y = 4000$. If you look very closely you can see that the estimate wavers slightly after the turn is initiated, but the filter tracks the measurement without lag and soon tracks smoothly.

```
In [34]: plt.plot(xs[390:450, 3], xs[390:450, 0], 'k')
         plt.scatter(zs[390:450, 1], zs[390:450, 0], marker='+', s=100);
         plt.xlabel('Y'); plt.ylabel('X')
         plt.gca().invert_xaxis()
         plt.axis('equal');
```




### 14.9.3 Limitations of the IMM

I haven't extensively worked with the IMM, so I am unable to address this as well as I would like. However, the IMM was invented to track maneuvering aircraft for air traffic control, and by all reports it performs superbly in that role.

That use case assumes a few things. Foremost among them is the requirement that all of the filters in the bank have the same dimensional design. A review of the math should show why. To create the mixed estimate the IMM performs this computation:

$$\mathbf{x} = \sum_{j=1}^{N} \mu_j \bar{\mathbf{x}}_j$$

This is computable if and only if the state $x$ in each filter is of the same dimension. Furthermore, the interpretation of $x[i]$ must be the same for each filter.

For example, suppose we tried to filter that uses a constant velocity model and another with a constant acceleration model. This doesn't work because the dimension of $x$ are different. `FilterPy` will raise a `ValueError` if you try to use filters with different dimensions.

```
In [35]: ca = KalmanFilter(3, 1)
         cv = KalmanFilter(2, 1)

         trans = np.array([[0.97, 0.03],
                           [0.03, 0.97]])

         imm = IMMEstimator([ca, cv], (0.5, 0.5), trans)


        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-35-e96cb02ea05b> in <module>
          5                   [0.03, 0.97]])
          6
    ----> 7 imm = IMMEstimator([ca, cv], (0.5, 0.5), trans)


        C:\pcloud\filterpy\filterpy\kalman\IMM.py in __init__(self, filters, mu, M)
        142             if x_shape != f.x.shape:
        143                 raise ValueError(
    --> 144                     'All filters must have the same state dimension')
        145
        146         self.x = zeros(filters[0].x.shape)


        ValueError: All filters must have the same state dimension
```

I occasionally get emails or bug reports about this. In the past I have recommended designing the filter with constant velocity to be of dimension 3, and then implement `F` to ignore acceleration

```
F = np.array([[1, dt, 0],
              [0, 1,  0],
              [0, 0,  0]])
```

In retrospect I am not sure if this is a sound recommendation. It allows the IMM to work, but clearly the estimate for acceleration will be incorrect, since one filter will have an accurate estimate of acceleration, and the other filter will have an estimate of 0. This inaccurate acceleration will then be used to perform the next predict cycle.

Consider a more extreme case. Suppose one of your filters interprets `x[2]` to be acceleration, and another filter interprets it as angular rotation rate. Clearly `x[2]` of the mixed estimate will be meaningless since you cannot sum (linear) acceleration with a rotation rate.

As I said I am not particularly well versed in the IMM. Perhaps the literature explains how to handle these situations. All I can say is that the IMM implemented by FilterPy will not work for these use cases.

The IMM as designed for air traffic control uses filters with different process assumptions. An aircraft can be flying level, it can be descending/ascending, it can be performing a coordinated turn, or an uncoordinated turn. You can design a filter for each case with a different `F` and `Q` matrix, but the state estimate `x` will be the same for all.

## 14.10 Summary

This chapter contains some of the more challenging material in this book. However, it is the gateway to implementing realistic Kalman filters. If we are controlling a robot we know its process model, and it is easy to construct a Kalman filter for it. Far more commonly we are given a set of time series data and asked to make sense of it. The process model is largely unknown to us. We use the techniques in this chapter to *learn* (in a machine learning sense) how to parameterize our models. The models change over time as the target maneuver, so our filters must be adaptive.

Finding an optimal answer involves combinatorial explosion, and is impractical in practice. The IMM algorithm has become the standard algorithm because of its good performance and computational tractability.

A real filter bank usually involves more than two filters. It is common to have many filters. As the target's regime changes some filters become infinitesimally likely. Most adaptive filters implement an algorithm that kills off extremely unlikely filters and replaces them with filters that more closely match the current regime. This is highly specific to your problem space, and is usually very ad-hoc. You will need to devise schemes for killing and creating filters and test them against simulated or real data.

Despite the complexity of the algorithms, I hope you recognize the underlying ideas are very simple. We use the same two tools that we learned in the second chapter: Bayes theorem and the total probability theorem. We incorporate new information using Bayes theorem, and compute the effect of the process models using the total probability theorem.

For me, this chapter underscores the beauty of the Bayesian formulation of Kalman filters. I don't much care if you learn the details of the IMM algorithm. I do hope that you see that very simple probabilistic reasoning led to these results. The linear algebra equations of the Kalman filter that Dr. Kalman derived came from a different form of reasoning called *orthogonal projection*. It is beautiful in its own way, and I urge you to read his paper. But I'm not sure I find them intuitive to use, and it is not at all clear how to devise new, non-optimal filters such as the IMM using those techniques. In contrast, Bayes theorem lets us handle these problems with ease.

## 14.11 References


- [1] Bar-Shalom, Y., Xiao-Rong L., and Thiagalingam Kirubarajan. *Estimation with Applications to Tracking and Navigation*. New York: Wiley, p. 424, 2001.
- [2] Zarchan, P., and Musoff, H., *Fundamentals of Kalman Filtering: A Practical Approach*. Reston, VA: American Institute of Aeronautics and Astronautics, 2000. Print.
- [3] Simon, D., *Optimal State Estimation: Kalman, H and Nonlinear Approaches*. Hoboken, NJ: Wiley-Interscience, p. 208-212, 2006
- [4] Crassidis, John L., and John L. Junkins. *Optimal estimation of dynamic systems*. CRC press, 2011.
- [5] Blom, H.A.P., "An Efficient Filter for Abruptly Changing Systems", *Proceedings of 23rd Conference on Decision and Control*, Las Vegas, NV, Dec 1984.

- [6] Blom, H.A.P and Bar-Shalom, Y., "The Interacting Multiple Model Algorithm for Systems with Markovian Switching Coefficients", *IEEE Transactions on Automatic Control*, Vol. AC-8, No. 8, Aug. 1998, pp. 780-783.

# Appendix A

# Installation

```
In [1]: from __future__ import division, print_function
```

This book is written in Jupyter Notebook, a browser based interactive Python environment that mixes Python, text, and math. I choose it because of the interactive features - I found Kalman filtering nearly impossible to learn until I started working in an interactive environment. It is difficult to form an intuition about many of the parameters until you can change them and immediately see the output. An interactive environment also allows you to play 'what if' scenarios. "What if I set **Q** to zero?" It is trivial to find out with Jupyter Notebook.

Another reason I choose it is because most textbooks leaves many things opaque. For example, there might be a beautiful plot next to some pseudocode. That plot was produced by software, but software that is not available to the reader. I want everything that went into producing this book to be available to you. How do you plot a covariance ellipse? You won't know if you read most books. With Jupyter Notebook all you have to do is look at the source code.

Even if you choose to read the book online you will want Python and the SciPy stack installed so that you can write your own Kalman filters. There are many different ways to install these libraries, and I cannot cover them all, but I will cover a few typical scenarios.

## A.1 Installing the SciPy Stack

This book requires IPython, Jupyter, NumPy, SciPy, SymPy, and Matplotlib. The SciPy stack of NumPy, SciPy, and Matplotlib depends on third party Fortran and C code, and is not trivial to install from source code. The SciPy website strongly urges using a pre-built installation, and I concur with this advice.

Jupyter notebook is the software that allows you to run Python inside of the browser - the book is a collection of Jupyter notebooks. IPython provides the infrastructure for Jupyter and data visualization. NumPy and Scipy are packages which provide the linear algebra implementation that the filters use. Sympy performs symbolic math - I use it to find derivatives of algebraic equations. Finally, Matplotlib provides plotting capability.

I use the Anaconda distribution from Continuum Analytics. This is an excellent distribution that combines all of the packages listed above, plus many others. IPython recommends this package to install Ipython. Installation is very straightforward, and it can be done alongside other Python installations you might already have on your machine. It is free to use. You may download it from here: http://continuum.io/downloads I strongly recommend using the latest Python 3 version that they provide. For now I support Python 2.7, but perhaps not much longer.

There are other choices for installing the SciPy stack. You can find instructions here: http://scipy.org/install.html It can be very cumbersome, and I do not support it or provide any instructions on how to do it.

Many Linux distributions come with these packages pre-installed. However, they are often somewhat dated and they will need to be updated as the book depends on recent versions of all. Updating a specific Linux installation is beyond the scope of this book. An advantage of the Anaconda distribution is that it does not modify your local Python installation, so you can install it and not break your linux distribution.

Some people have been tripped up by this. They install Anaconda, but the installed Python remains the default version and then the book's software doesn't run correctly.

I do not run regression tests on old versions of these libraries. In fact, I know the code will not run on older versions (say, from 2014-2015). I do not want to spend my life doing tech support for a book, thus I put the burden on you to install a recent version of Python and the SciPy stack.

You will need Python 2.7 or later installed. Almost all of my work is done in Python 3.6, but I periodically test on 2.7. I do not promise any specific check in will work in 2.7 however. I use Python's `from __future__ import ...` statement to help with compatibility. For example, all prints need to use parenthesis. If you try to add, say, `print x` into the book your script will fail; you must write `print(x)` as in Python 3.X.

Please submit a bug report at the book's github repository if you have installed the latest Anaconda and something does not work - I will continue to ensure the book will run with the latest Anaconda release. I'm rather indifferent if the book will not run on an older installation. I'm sorry, but I just don't have time to provide support for everyone's different setups. Packages like `jupyter notebook` are evolving rapidly, and I cannot keep up with all the changes *and* remain backwards compatible as well.

If you need older versions of the software for other projects, note that Anaconda allows you to install multiple versions side-by-side. Documentation for this is here:

https://conda.io/docs/user-guide/tasks/manage-python.html

## A.2 Installing FilterPy

FilterPy is a Python library that implements all of the filters used in this book, and quite a few others. Installation is easy using `pip`. Issue the following from the command prompt:

```
pip install filterpy
```

FilterPy is written by me, and the latest development version is always available at https://github.com/rlabbe/filterpy.

## A.3 Downloading and Running the Book

The book is stored in a github repository. From the command line type the following:

```
git clone --depth=1 https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python.git
```

This will create a directory named Kalman-and-Bayesian-Filters-in-Python. The `depth` parameter just gets you the latest version. Unless you need to see my entire commit history this is a lot faster and saves space.

If you do not have git installed, browse to https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python where you can download the book via your browser.

Now, from the command prompt change to the directory that was just created, and then run Jupyter notebook:

```
cd Kalman-and-Bayesian-Filters-in-Python
jupyter notebook
```

A browser window should launch showing you all of the chapters in the book. Browse to the first chapter by clicking on it, then open the notebook in that subdirectory by clicking on the link.

More information about running the notebook can be found here:

http://jupyter-notebook-beginner-guide.readthedocs.org/en/latest/execute.html

## A.4 Companion Software

Code that is specific to the book is stored with the book in the subdirectory *./kf_book*. This code is in a state of flux; I do not wish to document it here yet. I do mention in the book when I use code from this directory, so it should not be a mystery.

In the *kf_book* subdirectory there are Python files with a name like *xxx*_internal.py. I use these to store functions that are useful for a specific chapter. This allows me to hide away Python code that is not particularly interesting to read - I may be generating a plot or chart, and I want you to focus on the contents of the chart, not the mechanics of how I generate that chart with Python. If you are curious as to the mechanics of that, just go and browse the source.

Some chapters introduce functions that are useful for the rest of the book. Those functions are initially defined within the Notebook itself, but the code is also stored in a Python file that is imported if needed in later chapters. I do document when I do this where the function is first defined, but this is still a work in progress. I try to avoid this because then I always face the issue of code in the directory becoming out of sync with the code in the book. However, IPython Notebook does not give us a way to refer to code cells in other notebooks, so this is the only mechanism I know of to share functionality across notebooks.

There is an undocumented directory called **experiments**. This is where I write and test code prior to putting it in the book. There is some interesting stuff in there, and feel free to look at it. As the book evolves I plan to create examples and projects, and a lot of this material will end up there. Small experiments will eventually just be deleted. If you are just interested in reading the book you can safely ignore this directory.

The subdirectory *./kf_book* contains a css file containing the style guide for the book. The default look and feel of IPython Notebook is rather plain. Work is being done on this. I have followed the examples set by books such as Probabilistic Programming and Bayesian Methods for Hackers. I have also been very influenced by Professor Lorena Barba's fantastic work, available here. I owe all of my look and feel to the work of these projects.

## A.5 Using Jupyter Notebook

A complete tutorial on Jupyter Notebook is beyond the scope of this book. Many are available online. In short, Python code is placed in cells. These are prefaced with text like `In [1]:`, and the code itself is in a boxed area. If you press CTRL-ENTER while focus is inside the box the code will run and the results will be displayed below the box. Like this:

```
In [2]: print(3+7.2)
```

```
10.2
```

If you have this open in Jupyter Notebook now, go ahead and modify that code by changing the expression inside the print statement and pressing CTRL+ENTER. The output should be changed to reflect what you typed in the code cell.

## A.6 SymPy

SymPy is a Python package for performing symbolic mathematics. The full scope of its abilities are beyond this book, but it can perform algebra, integrate and differentiate equations, find solutions to differential equations, and much more. For example, we use use it to compute the Jacobian of matrices and expected value integral computations.

First, a simple example. We will import SymPy, initialize its pretty print functionality (which will print equations using LaTeX). We will then declare a symbol for SymPy to use.

```
In [3]: import sympy
        sympy.init_printing(use_latex='mathjax')
```

```
        phi, x = sympy.symbols('\phi, x')
        phi
```

Out[3]:

$\phi$

Notice how it prints the symbol `phi` using LaTeX. Now let's do some math. What is the derivative of $\sqrt{\phi}$?

```
In [4]: sympy.diff('sqrt(phi)')
```

Out[4]:

$$\frac{1}{2\sqrt{\phi}}$$

We can factor equations

```
In [5]: sympy.factor(phi**3 -phi**2 + phi - 1)
```

Out[5]:

$(\phi - 1)\left(\phi^2 + 1\right)$

and we can expand them.

```
In [6]: ((phi+1)*(phi-4)).expand()
```

Out[6]:

$\phi^2 - 3\phi - 4$

You can evauate an equation for specific values of its variables:

```
In [7]: w =x**2 -3*x +4
        print(w.subs(x, 4))
        print(w.subs(x, 12))
```

```
8
112
```

You can also use strings for equations that use symbols that you have not defined:

```
In [8]: x = sympy.expand('(t+1)*2')
        x
```

Out[8]:

$2t + 2$

Now let's use SymPy to compute the Jacobian of a matrix. Given the function

$$h = \sqrt{(x^2 + z^2)}$$

find the Jacobian with respect to x, y, and z.

```
In [9]: x, y, z = sympy.symbols('x y z')

        H = sympy.Matrix([sympy.sqrt(x**2 + z**2)])

        state = sympy.Matrix([x, y, z])
        H.jacobian(state)
```

Out[9]:

$$\begin{bmatrix} \frac{x}{\sqrt{x^2+z^2}} & 0 & \frac{z}{\sqrt{x^2+z^2}} \end{bmatrix}$$

Now let's compute the discrete process noise matrix $\mathbf{Q}$ given the continuous process noise matrix

$$\mathbf{Q} = \Phi_s \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The integral is

$$\mathbf{Q} = \int_0^{\Delta t} \mathbf{F}(t)\mathbf{Q}\mathbf{F}^T(t)\, dt$$

where

$$\mathbf{F}(\Delta t) = \begin{bmatrix} 1 & \Delta t & \Delta t^2/2 \\ 0 & 1 & \Delta t \\ 0 & 0 & 1 \end{bmatrix}$$

In [10]:
```
dt = sympy.symbols('\Delta{t}')
F_k = sympy.Matrix([[1, dt, dt**2/2],
                    [0,  1,      dt],
                    [0,  0,       1]])
Q = sympy.Matrix([[0,0,0],
                  [0,0,0],
                  [0,0,1]])

sympy.integrate(F_k*Q*F_k.T,(dt, 0, dt))
```

Out[10]:

$$\begin{bmatrix} \frac{\Delta t^5}{20} & \frac{\Delta t^4}{8} & \frac{\Delta t^3}{6} \\ \frac{\Delta t^4}{8} & \frac{\Delta t^3}{3} & \frac{\Delta t^2}{2} \\ \frac{\Delta t^3}{6} & \frac{\Delta t^2}{2} & \Delta t \end{bmatrix}$$

## A.7 Various Links

https://ipython.org/

https://jupyter.org/

https://www.scipy.org/

# Appendix B

# Symbols and Notations

Here is a collection of the notation used by various authors for the linear Kalman filter equations.

## B.1 Labbe

$$\begin{aligned}
\overline{\mathbf{x}} &= \mathbf{F}\mathbf{x} + \mathbf{B}\mathbf{u} \\
\overline{\mathbf{P}} &= \mathbf{F}\mathbf{P}\mathbf{F}^{\mathsf{T}} + \mathbf{Q}
\end{aligned}$$

$$\begin{aligned}
\mathbf{y} &= \mathbf{z} - \mathbf{H}\overline{\mathbf{x}} \\
\mathbf{S} &= \mathbf{H}\overline{\mathbf{P}}\mathbf{H}^{\mathsf{T}} + \mathbf{R} \\
\mathbf{K} &= \overline{\mathbf{P}}\mathbf{H}^{\mathsf{T}}\mathbf{S}^{-1} \\
\mathbf{x} &= \overline{\mathbf{x}} + \mathbf{K}\mathbf{y} \\
\mathbf{P} &= (\mathbf{I} - \mathbf{K}\mathbf{H})\overline{\mathbf{P}}
\end{aligned}$$

## B.2 Wikipedia

$$\begin{aligned}
\hat{\mathbf{x}}_{k|k-1} &= \mathbf{F}_k\hat{\mathbf{x}}_{k-1|k-1} + \mathbf{B}_k\mathbf{u}_k \\
\mathbf{P}_{k|k-1} &= \mathbf{F}_k\mathbf{P}_{k-1|k-1}\mathbf{F}_k^{\mathsf{T}} + \mathbf{Q}_k \\
\tilde{\mathbf{y}}_k &= \mathbf{z}_k - \mathbf{H}_k\hat{\mathbf{x}}_{k|k-1} \\
\mathbf{S}_k &= \mathbf{H}_k\mathbf{P}_{k|k-1}\mathbf{H}_k^{\mathsf{T}} + \mathbf{R}_k \\
\mathbf{K}_k &= \mathbf{P}_{k|k-1}\mathbf{H}_k^{\mathsf{T}}\mathbf{S}_k^{-1} \\
\hat{\mathbf{x}}_{k|k} &= \hat{\mathbf{x}}_{k|k-1} + \mathbf{K}_k\tilde{\mathbf{y}}_k \\
\mathbf{P}_{k|k} &= (I - \mathbf{K}_k\mathbf{H}_k)\mathbf{P}_{k|k-1}
\end{aligned}$$

## B.3 Brookner

$$\begin{aligned}
X^*_{n+1,n} &= \Phi X^*_{n,n} \\
X^*_{n,n} &= X^*_{n,n-1} + H_n(Y_n - MX^*_{n,n-1}) \\
H_n &= S^*_{n,n-1}M^{\mathsf{T}}[R_n + MS^*_{n,n-1}M^{\mathsf{T}}]^{-1} \\
S^*_{n,n-1} &= \Phi S^*_{n-1,n-1}\Phi^{\mathsf{T}} + Q_n \\
S^*_{n-1,n-1} &= (I - H_{n-1}M)S^*_{n-1,n-2}
\end{aligned}$$

## B.4 Gelb

$$
\begin{aligned}
\underline{\hat{x}}_k(-) &= \Phi_{k-1}\underline{\hat{x}}_{k-1}(+) \\
\underline{\hat{x}}_k(+) &= \underline{\hat{x}}_k(-) + K_k[Z_k - H_k\underline{\hat{x}}_k(-)] \\
K_k &= P_k(-)H_k^{\mathsf{T}}[H_kP_k(-)H_k^{\mathsf{T}} + R_k]^{-1} \\
P_k(+) &= \Phi_{k-1}P_{k-1}(+)\Phi_{k-1}^{\mathsf{T}} + Q_{k-1} \\
P_k(-) &= (I - K_kH_k)P_k(-)
\end{aligned}
$$

## B.5 Brown

$$
\begin{aligned}
\hat{\mathbf{x}}_{k+1}^- &= \phi_k\hat{\mathbf{x}}_k \\
\hat{\mathbf{x}}_k &= \hat{\mathbf{x}}_k^- + \mathbf{K}_k[\mathbf{z}_k - \mathbf{H}_k\hat{x}_k^-] \\
\mathbf{K}_k &= \mathbf{P}_k^-\mathbf{H}_k^{\mathsf{T}}[\mathbf{H}_k\mathbf{P}_k^-\mathbf{H}_k^T + \mathbf{R}_k]^{-1} \\
\mathbf{P}_{k+1}^- &= \phi_k\mathbf{P}_k\phi_k^{\mathsf{T}} + \mathbf{Q}_k \\
\mathbf{P}_k &= (\mathbf{I} - \mathbf{K}_k\mathbf{H}_k)\mathbf{P}_k^-
\end{aligned}
$$

## B.6 Zarchan

$$
\begin{aligned}
\hat{x}_k &= \Phi_k\hat{x}_{k-1} + G_ku_{k-1} + K_k[z_k - H\Phi_k\hat{x}_{k-1} - HG_ku_{k-1}] \\
M_k &= \Phi_kP_{k-1}\phi_k^{\mathsf{T}} + Q_k \\
K_k &= M_kH^{\mathsf{T}}[HM_kH^{\mathsf{T}} + R_k]^{-1} \\
P_k &= (I - K_kH)M_k
\end{aligned}
$$

# Appendix C

# H Infinity filter

In [1]:
```
from __future__ import division, print_function
%matplotlib inline
```

In [2]:
```
#format the book
import book_format
book_format.set_style()
```

Out[2]:

I am still mulling over how to write this chapter. In the meantime, Professor Dan Simon at Cleveland State University has an accessible introduction here:

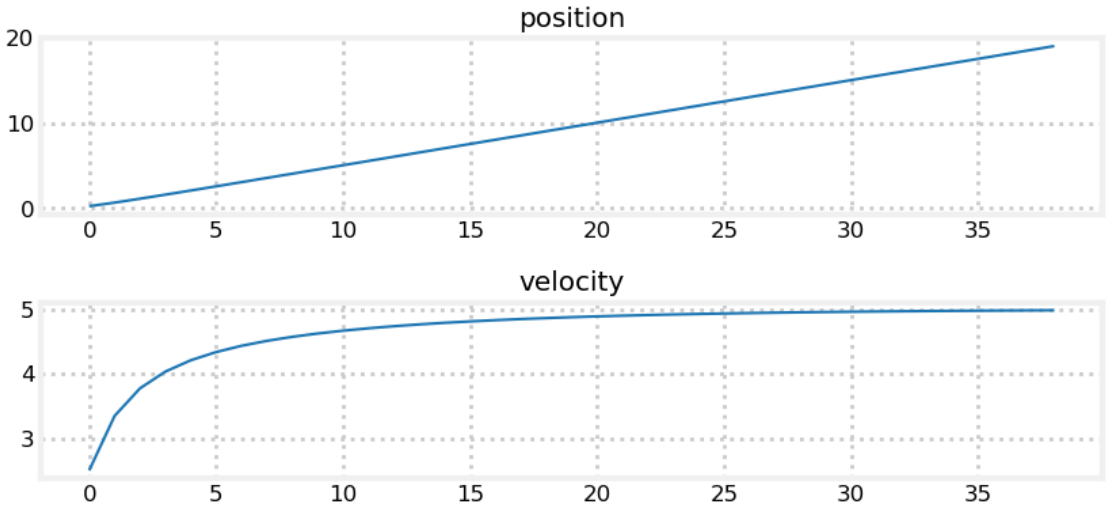
http://academic.csuohio.edu/simond/courses/eec641/hinfinity.pdf

In one sentence the $H_\infty$ (H infinity) filter is like a Kalman filter, but it is robust in the face of non-Gaussian, non-predictable inputs.

My FilterPy library contains an H-Infinity filter. I've pasted some test code below which implements the filter designed by Simon in the article above. Hope it helps.

In [3]:
```
from __future__ import (absolute_import, division, print_function,
                        unicode_literals)

from numpy import array
import matplotlib.pyplot as plt

from filterpy.hinfinity import HInfinityFilter

dt = 0.1
f = HInfinityFilter(2, 1, dim_u=1, gamma=.01)

f.F = array([[1., dt],
             [0., 1.]])

f.H = array([[0., 1.]])
f.G = array([[dt**2 / 2, dt]]).T

f.P = 0.01
f.W = array([[0.0003, 0.005],
             [0.0050, 0.100]])/ 1000  #process noise

f.V = 0.01
f.Q = 0.01
```

```
u = 1. #acceleration of 1 f/sec**2

xs = []
vs = []

for i in range(1,40):
    f.update (5)
    #print(f.x.T)
    xs.append(f.x[0,0])
    vs.append(f.x[1,0])
    f.predict(u=u)

plt.subplot(211)
plt.plot(xs)
plt.title('position')
plt.subplot(212)
plt.plot(vs)
plt.title('velocity');
```



# Appendix D

# Ensemble Kalman Filters

```
In [1]: from __future__ import division, print_function
        %matplotlib inline
```

```
In [2]: #format the book
        import book_format
        book_format.set_style()
```

Out[2]:

> I am not well versed with Ensemble filters. I have implemented one for this book, and made it work, but I have not used one in real life. Different sources use slightly different forms of these equations. If I implement the equations given in the sources the filter does not work. It is possible that I am doing something wrong. However, in various places on the web I have seen comments by people stating that they do the kinds of things I have done in my filter to make it work. In short, I do not understand this topic well, but choose to present my lack of knowledge rather than to gloss over the subject. I hope to master this topic in the future and to author a more definitive chapter. At the end of the chapter I document my current confusion and questions. In any case if I got confused by the sources perhaps you also will, so documenting my confusion can help you avoid the same.

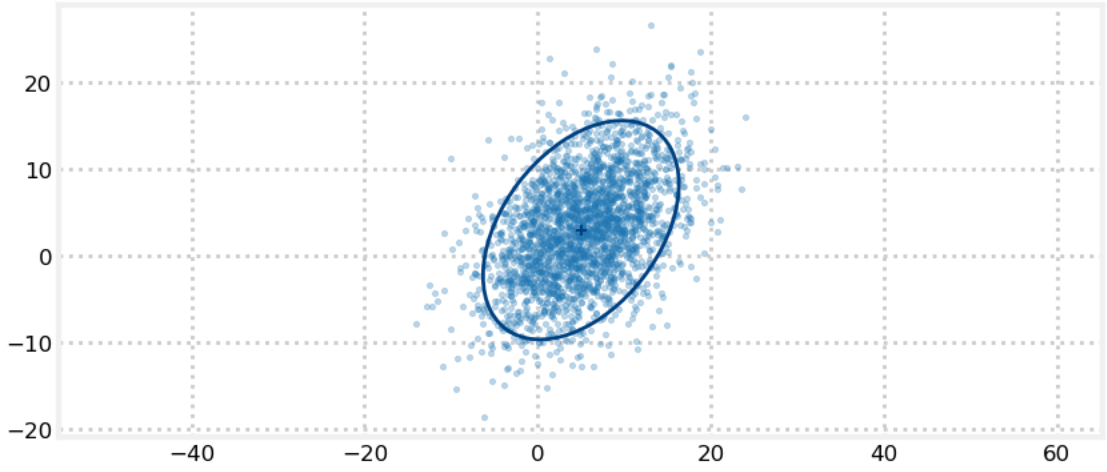
The ensemble Kalman filter (EnKF) is very similar to the unscented Kalman filter (UKF) of the last chapter. If you recall, the UKF uses a set of deterministically chosen weighted sigma points passed through nonlinear state and measurement functions. After the sigma points are passed through the function, we find the mean and covariance of the points and use this as the filter's new mean and covariance. It is only an approximation of the true value, and thus suboptimal, but in practice the filter is highly accurate. It has the advantage of often producing more accurate estimates than the EKF does, and also does not require you to analytically derive the linearization of the state and measurement equations.

The ensemble Kalman filter works in a similar way, except it uses a *Monte Carlo* method to choose a large numbers of sigma points. It came about from the geophysical sciences as an answer for the very large states and systems needed to model things such as the ocean and atmosphere. There is an interesting article on it's development in weather modeling in *SIAM News* [1]. The filter starts by randomly generating a large number of points distributed about the filter's initial state. This distribution is proportional to the filter's covariance $\mathbf{P}$. In other words 68% of the points will be within one standard deviation of the mean, 95% percent within two standard deviations, and so on. Let's look at this in two dimensions. We will use `numpy.random.multivariate_normal()` function to randomly create points from a multivariate normal distribution drawn from the mean (5, 3) with the covariance

$$\begin{bmatrix}32 & 15\\15 & 40\end{bmatrix}$$

I've drawn the covariance ellipse representing two standard deviations to illustrate how the points are distributed.

```
In [3]: import matplotlib.pyplot as plt
        import numpy as np
        from numpy.random import multivariate_normal
        from filterpy.stats import (covariance_ellipse,
                                    plot_covariance_ellipse)

        mean = (5, 3)
        P = np.array([[32, 15],
                      [15., 40.]])

        x,y = multivariate_normal(mean=mean, cov=P, size=2500).T
        plt.scatter(x, y, alpha=0.3, marker='.')
        plt.axis('equal')

        plot_covariance_ellipse(mean=mean, cov=P,
                                variance=2.**2,
                                facecolor='none')
```




## D.1 The Algorithm

As I already stated, when the filter is initialized a large number of sigma points are drawn from the initial state ($\mathbf{x}$) and covariance ($\mathbf{P}$). From there the algorithm proceeds very similarly to the UKF. During the prediction step the sigma points are passed through the state transition function, and then perturbed by adding a bit of noise to account for the process noise. During the update step the sigma points are translated into measurement space by passing them through the measurement function, they are perturbed by a small amount to account for the measurement noise. The Kalman gain is computed from the

We already mentioned the main difference between the UKF and EnKF - the UKF choses the sigma points deterministically. There is another difference, implied by the algorithm above. With the UKF we generate new sigma points during each predict step, and after passing the points through the nonlinear function we reconstitute them into a mean and covariance by using the *unscented transform*. The EnKF keeps propagating the originally created sigma points; we only need to compute a mean and covariance as outputs for the filter!

Let's look at the equations for the filter. As usual, I will leave out the typical subscripts and superscripts; I am expressing an algorithm, not mathematical functions. Here $N$ is the number of sigma points, $\chi$ is the set of sigma points.

### D.1.1 Initialize Step

$$\boldsymbol{\chi} \sim \mathcal{N}(\mathbf{x}_0, \mathbf{P}_0)$$

This says to select the sigma points from the filter's initial mean and covariance. In code this might look like

```
N = 1000
sigmas = multivariate_normal(mean=x, cov=P, size=N)
```

### D.1.2 Predict Step

$$\boldsymbol{\chi} = f(\boldsymbol{\chi}, \mathbf{u}) + v_Q$$
$$\mathbf{x} = \frac{1}{N}\sum_1^N \boldsymbol{\chi}$$

That is short and sweet, but perhaps not entirely clear. The first line passes all of the sigma points through a use supplied state transition function and then adds some noise distributed according to the **Q** matrix. In Python we might write

```
for i, s in enumerate(sigmas):
    sigmas[i] = fx(x=s, dt=0.1, u=0.)

sigmas += multivariate_normal(x, Q, N)
```

The second line computes the mean from the sigmas. In Python we will take advantage of `numpy.mean` to do this very concisely and quickly.

```
x = np.mean(sigmas, axis=0)
```

We can now optionally compute the covariance of the mean. The algorithm does not need to compute this value, but it is often useful for analysis. The equation is

$$\mathbf{P} = \frac{1}{N-1}\sum_1^N [\boldsymbol{\chi} - \mathbf{x}^-][\boldsymbol{\chi} - \mathbf{x}^-]^\mathsf{T}$$

$\boldsymbol{\chi} - \mathbf{x}^-$ is a one dimensional vector, so we will use `numpy.outer` to compute the $[\boldsymbol{\chi} - \mathbf{x}^-][\boldsymbol{\chi} - \mathbf{x}^-]^\mathsf{T}$ term. In Python we might write

```
P = 0
for s in sigmas:
    P += outer(s-x, s-x)
P = P / (N-1)
```

### D.1.3 Update Step

In the update step we pass the sigma points through the measurement function, compute the mean and covariance of the sigma points, compute the Kalman gain from the covariance, and then update the Kalman state by scaling the residual by the Kalman gain. The equations are

$$\begin{aligned}
\boldsymbol{\chi}_h &= h(\boldsymbol{\chi}, u) \\
\mathbf{z}_{mean} &= \frac{1}{N}\sum_1^N \boldsymbol{\chi}_h
\end{aligned}$$

$$\begin{aligned}
\mathbf{P}_{zz} &= \frac{1}{N-1}\sum_1^N [\boldsymbol{\chi}_h - \mathbf{z}_{mean}][\boldsymbol{\chi}_h - \mathbf{z}_{mean}]^\mathsf{T} + \mathbf{R} \\
\mathbf{P}_{xz} &= \frac{1}{N-1}\sum_1^N [\boldsymbol{\chi} - \mathbf{x}^-][\boldsymbol{\chi}_h - \mathbf{z}_{mean}]^\mathsf{T}
\end{aligned}$$

$$\begin{aligned}
\mathbf{K} &= \mathbf{P}_{xz}\mathbf{P}_{zz}^{-1} \\
\boldsymbol{\chi} &= \boldsymbol{\chi} + \mathbf{K}[\mathbf{z} - \boldsymbol{\chi}_h + \mathbf{v}_R]
\end{aligned}$$

$$\begin{aligned}
\mathbf{x} &= \frac{1}{N}\sum_1^N \boldsymbol{\chi} \\
\mathbf{P} &= \mathbf{P} - \mathbf{K}\mathbf{P}_{zz}\mathbf{K}^\mathsf{T}
\end{aligned}$$

This is very similar to the linear KF and the UKF. Let's just go line by line.
The first line,

$$\boldsymbol{\chi}_h = h(\boldsymbol{\chi}, u),$$

just passes the sigma points through the measurement function $h$. We name the resulting points $\chi_h$ to distinguish them from the sigma points. In Python we could write this as

```
sigmas_h = h(sigmas, u)
```

The next line computes the mean of the measurement sigmas.

$$\mathbf{z}_{mean} = \frac{1}{N}\sum_1^N \boldsymbol{\chi}_h$$

In Python we write

```
z_mean = np.mean(sigmas_h, axis=0)
```

Now that we have the mean of the measurement sigmas we can compute the covariance for every measurement sigma point, and the *cross variance* for the measurement sigma points vs the sigma points. That is expressed by these two equations

$$\begin{aligned}
\mathbf{P}_{zz} &= \frac{1}{N-1}\sum_1^N [\boldsymbol{\chi}_h - \mathbf{z}_{mean}][\boldsymbol{\chi}_h - \mathbf{z}_{mean}]^\mathsf{T} + \mathbf{R} \\
\mathbf{P}_{xz} &= \frac{1}{N-1}\sum_1^N [\boldsymbol{\chi} - \mathbf{x}^-][\boldsymbol{\chi}_h - \mathbf{z}_{mean}]^\mathsf{T}
\end{aligned}$$

We can express this in Python with

```
P_zz = 0
for sigma in sigmas_h:
    s = sigma - z_mean
```

```
    P_zz += outer(s, s)
P_zz = P_zz / (N-1) + R

P_xz = 0
for i in range(N):
    P_xz += outer(self.sigmas[i] - self.x, sigmas_h[i] - z_mean)
P_xz /= N-1
```

Computation of the Kalman gain is straightforward $\mathbf{K} = \mathbf{P}_{xz}\mathbf{P}_{zz}^{-1}$.
In Python this is
`python K = np.dot(P_xz, inv(P_zz))`
Next, we update the sigma points with

$$\boldsymbol{\chi} = \boldsymbol{\chi} + \mathbf{K}[\mathbf{z} - \boldsymbol{\chi}_h + \mathbf{v}_R]$$

Here $\mathbf{v}_R$ is the perturbation that we add to the sigmas. In Python we can implement this with

```
v_r = multivariate_normal([0]*dim_z, R, N)
for i in range(N):
    sigmas[i] += dot(K, z + v_r[i] - sigmas_h[i])
```

Our final step is recompute the filter's mean and covariance.

```
    x = np.mean(sigmas, axis=0)
    P = self.P - dot(K, P_zz).dot(K.T)
```

## D.2 Implementation and Example

I have implemented an EnKF in the `FilterPy` library. It is in many ways a toy. Filtering with a large number of sigma points gives us very slow performance. Furthermore, there are many minor variations on the algorithm in the literature. I wrote this mostly because I was interested in learning a bit about the filter. I have not used it for a real world problem, and I can give no advice on using the filter for the large problems for which it is suited. Therefore I will refine my comments to implementing a very simple filter. I will use it to track an object in one dimension, and compare the output to a linear Kalman filter. This is a filter we have designed many times already in this book, so I will design it with little comment. Our state vector will be

$$\mathbf{x} = \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

The state transition function is

$$\mathbf{F} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$$

and the measurement function is

$$\mathbf{H} = \begin{bmatrix} 1 & 0 \end{bmatrix}$$

The EnKF is designed for nonlinear problems, so instead of using matrices to implement the state transition and measurement functions you will need to supply Python functions. For this problem they can be written as:

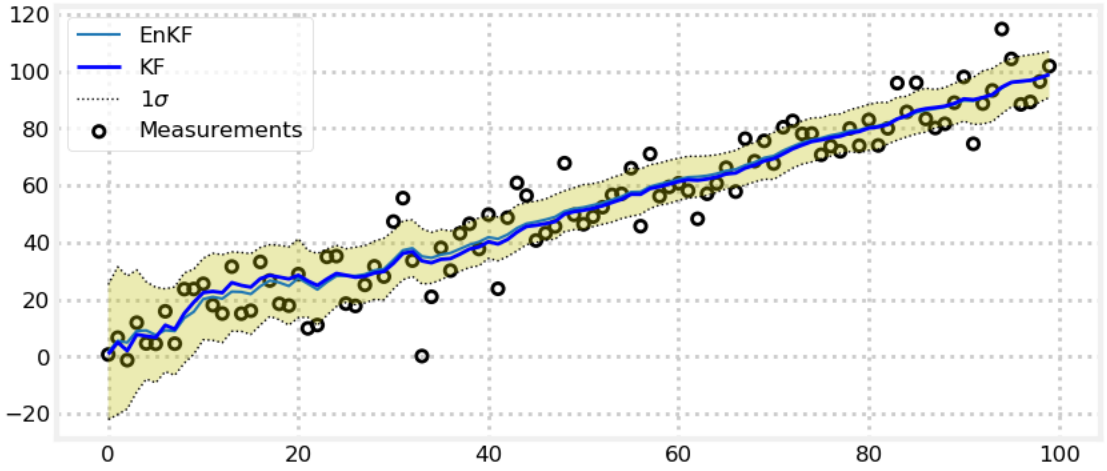
```
def hx(x):
    return np.array([x[0]])

def fx(x, dt):
    return np.dot(F, x)
```

One final thing: the EnKF code, like the UKF code, uses a single dimension for $\mathbf{x}$, not a two dimensional column matrix as used by the linear kalman filter code.

Without further ado, here is the code.

```
In [4]: import kf_book.book_plots as bp
        from numpy.random import randn
        from filterpy.kalman import EnsembleKalmanFilter as EnKF
        from filterpy.kalman import KalmanFilter
        from filterpy.common import Q_discrete_white_noise

        np.random.seed(1234)

        def hx(x):
            return np.array([x[0]])

        def fx(x, dt):
            return np.dot(F, x)

        F = np.array([[1., 1.],[0., 1.]])

        x = np.array([0., 1.])
        P = np.eye(2) * 100.
        enf = EnKF(x=x, P=P, dim_z=1, dt=1., N=20, hx=hx, fx=fx)

        std_noise = 10.
        enf.R *= std_noise**2
        enf.Q = Q_discrete_white_noise(2, 1., .001)

        kf = KalmanFilter(dim_x=2, dim_z=1)
        kf.x = np.array([x]).T
        kf.F = F.copy()
        kf.P = P.copy()
        kf.R = enf.R.copy()
        kf.Q = enf.Q.copy()
        kf.H = np.array([[1., 0.]])

        measurements = []
        results = []
        ps = []
        kf_results = []

        zs = []
        for t in range (0,100):
            # create measurement = t plus white noise
            z = t + randn()*std_noise
            zs.append(z)

            enf.predict()
            enf.update(np.asarray([z]))

            kf.predict()
            kf.update(np.asarray([[z]]))

            # save data
```

```
        results.append (enf.x[0])
        kf_results.append (kf.x[0,0])
        measurements.append(z)
        ps.append(3*(enf.P[0,0]**.5))

    results = np.asarray(results)
    ps = np.asarray(ps)

    plt.plot(results, label='EnKF')
    plt.plot(kf_results, label='KF', c='b', lw=2)
    bp.plot_measurements(measurements)
    plt.plot (results - ps, c='k',linestyle=':', lw=1, label='1$\sigma$')
    plt.plot(results + ps, c='k', linestyle=':', lw=1)
    plt.fill_between(range(100), results - ps, results + ps, facecolor='y', alpha=.3)
    plt.legend(loc='best');
```




It can be a bit difficult to see, but the KF and EnKF start off slightly different, but soon converge to producing nearly the same values. The EnKF is a suboptimal filter, so it will not produce the optimal solution that the KF produces. However, I deliberately chose $N$ to be quite small (20) to guarantee that the EnKF output is quite suboptimal. If I chose a more reasonable number such as 2000 you would be unable to see the difference between the two filter outputs on this graph.

## D.3 Outstanding Questions

All of this should be considered as *my* questions, not lingering questions in the literature. However, I am copying equations directly from well known sources in the field, and they do not address the discrepancies.

First, in Brown [2] we have all sums multiplied by $\frac{1}{N}$, as in

$$\hat{x} = \frac{1}{N}\sum_{i=1}^{N} \chi_k^{(i)}$$

The same equation in Crassidis [3] reads (I'll use the same notation as in Brown, although Crassidis' is different)

$$\hat{x} = \frac{1}{N-1}\sum_{i=1}^{N} \chi_k^{(i)}$$

The same is true in both sources for the sums in the computation for the covariances. Crassidis, in the context of talking about the filter's covariance, states that $N-1$ is used to ensure an unbiased estimate. Given the following standard equations for the mean and standard deviation (p.2 of Crassidis), this makes sense for the covariance.

$$\mu = \frac{1}{N}\sum_{i=1}^{N}[\tilde{z}(t_i) - \hat{z}(t_i)]$$

$$\sigma^2 = \frac{1}{N-1}\sum_{i=1}^{N}\{[\tilde{z}(t_i) - \hat{z}(t_i)] - \mu\}^2$$

However, I see no justification or reason to use $N-1$ to compute the mean. If I use $N-1$ in the filter for the mean the filter does not converge and the state essentially follows the measurements without any filtering. However, I do see a reason to use it for the covariance as in Crassidis, in contrast to Brown. Again, I support my decision empirically - $N-1$ works in the implementation of the filter, $N$ does not.

My second question relates to the use of the $\mathbf{R}$ matrix. In Brown $\mathbf{R}$ is added to $\mathbf{P}_{zz}$ whereas it isn't in Crassidis and other sources. I have read on the web notes by other implementers that adding R helps the filter, and it certainly seems reasonable and necessary to me, so this is what I do.

My third question relates to the computation of the covariance $\mathbf{P}$. Again, we have different equations in Crassidis and Brown. I have chosen the implementation given in Brown as it seems to give me the behavior that I expect (convergence of $\mathbf{P}$ over time) and it closely compares to the form in the linear KF. In contrast I find the equations in Crassidis do not seem to converge much.

My fourth question relates to the state estimate update. In Brown we have

$$\boldsymbol{\chi} = \boldsymbol{\chi} + \mathbf{K}[\mathbf{z} - \mathbf{z}_{mean} + \mathbf{v}_R]$$

whereas in Crassidis we have

$$\boldsymbol{\chi} = \boldsymbol{\chi} + \mathbf{K}[\mathbf{z} - \boldsymbol{\chi}_h + \mathbf{v}_R]$$

To me the Crassidis equation seems logical, and it produces a filter that performs like the linear KF for linear problems, so that is the formulation that I have chosen.

I am not comfortable saying either book is wrong; it is quite possible that I missed some point that makes each set of equations work. I can say that when I implemented them as written I did not get a filter that worked. I define "work" as performs essentially the same as the linear KF for linear problems. Between reading implementation notes on the web and reasoning about various issues I have chosen the implementation in this chapter, which does in fact seem to work correctly. I have yet to explore the significant amount of original literature that will likely definitively explain the discrepancies. I would like to leave this here in some form even if I do find an explanation that reconciles the various differences, as if I got confused by these books than probably others will as well.

## D.4 References


- [1] Mackenzie, Dana. *Ensemble Kalman Filters Bring Weather Models Up to Date* Siam News, Volume 36, Number 8, October 2003. http://www.siam.org/pdf/news/362.pdf
- [2] Brown, Robert Grover, and Patrick Y.C. Hwang. *Introduction to Random Signals and Applied Kalman Filtering, With MATLAB® excercises and solutions.* Wiley, 2012.
- [3] Crassidis, John L., and John L. Junkins. *Optimal estimation of dynamic systems.* CRC press, 2011.