# Kalman and Bayesian Filters in Python

Roger R Labbe Jr

# Contents

# Contents

Kalman and Bayesian Filters in Python
Table of Contents

**Version 0.0**

Not ready for public consumption. In development.

## 0.1 Motivation

This is a book for programmers that have a need or interest in Kalman filtering. The motivation for this book came out of my desire for a gentle introduction to Kalman filtering. I'm a software engineer that spent almost two decades in the avionics field, and so I have always been 'bumping elbows' with the Kalman filter, but never had the need to implement one myself. As I moved into solving tracking problems with computer vision I needed to start implementing them. There are classic textbooks in the field, such as Grewal and Andrew's excellent *Kalman Filtering.* But sitting down and trying to read these books is a dismal and trying experience if you do not have the background. Typcially the first few chapters fly through several years of undergraduate math, blithely referring you to textbooks on, for example, Itō calculus, and presenting an entire semester's worth of statistics in a few brief paragraphs. These books are good textbooks for an upper undergraduate course, and an invaluable reference to researchers and professionals, but the going is truly difficult for the more casual reader. Symbology is introduced without explanation, different texts use different words and variables 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 defition, 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.

However, as I began to finally understand the Kalman filter I realized the underlying concepts are quite straightforward. A few simple probability rules, some intuition about how we integrate disparate knowledge to explain events in our everyday life and the core 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 itself. A book or paper's author makes some statement of fact and presents a graph as proof. Unfortunately, why the statement is true is not clear to me, nor is the method by which you might make that plot obvious. Or maybe I wonder "is this true if R=0?" Or the author provide 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, but excercises 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 witholds information from me, presumably to avoid 'cheating'. None of this necessary, from my point of view. Certainly if you are designing a Kalman filter for a aircraft or missile you must be a thorough master of all of the mathematics and topics in a typical Kalman filter textbook. I just 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 chose different parameters than the author chose. I want to run simulations. I want to inject more noise in the signal and see how a filter performs.

I wrote this book to address all of those needs. This is not the book for you if you program avionics for Boeing or design radars for Ratheon. Go get a degree at Georgia Tech, UW, or the like, because you'll need it for that kind of work. This book is for the hobbiest, the curious, and the working engineer that needs to filter or smooth data.

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 IPython Notebook, which allows you to combine text, python, and python output in one place. Every plot, every piece of data in this book is generated from Python that is available to you right inside the notebook. Want to double the value of a parameter. Click on the Python cell, change the parameter's value, and click 'Run'. A new plot or printed output will appear in the book.

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.

This book has supporting libraries for computing statistics, plotting various things related to filters, and for the various filters that we cover. This does require a strong caveat; most code is written for didactic purposes. It is rare that I chose the most efficient solution (which often obscures the intent of the code), and I mostly did not concern myself with numerical stability. This is important to understand - Kalman filters in aircraft are carefully designed and implemented to be numerically stable; the naive implemention is not stable in many cases. If you are serious about Kalman filters this book will not be the last book you need. My intention is to introduce you to the concepts and mathematics, and to get you to the point where the textbooks are approachable.

Finally, this book is free. The cost for the books required to learn Kalman filtering is somewhat prohibitive even for a Silicon Valley engineer like myself; I cannot believe the 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 here. It's time to repay that. So, the book is free, it is hosted on free servers, and it uses free software for all of the code.

## 0.2 Reading the book

There are multiple ways to read this book. However, it is intended to be interactive and I recommend using it in that form. If you install IPython 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!

If you do not want to do that you can read this book online. the website http://nbviewer.org provides an IPython Notebook server that renders a notebook stored at github (or elsewhere). The rendering is done in real time when you load the book. If you read my book today, and then I make a change tomorrow, when you go back tomorrow you will see that change.

You may access this book via nbviewer at any by using this address: http://nbviewer.ipython.org/github/rlabbe/Kalman-Filters-and-Random-Signals-in-Python/blob/master/Introduction.ipynb

Finally, you may generate output in a variety of formats. I will not cover how to do that, other than to point you to IPython nbconvert. You can convert this book into static HTML pages, latex, or PDF. While I don't recommend it particularly, it is useful for those that don't want to program and/or are working offline.

## 0.3 Installation and Software Requirements

If you want to run the notebook on your computer, which is what I recommend, then you will have to have IPython installed. I do not cover how to do that in this book; requirements change based on what other python installations you may have, whether you use a third party package like Anaconda Python, what operating system you are using, and so on.

To use all features you will have to have Ipython 2.0 installed, which is released and stable as of April 2014. Most of the book does not require that, but I do make use of the interactive plotting widgets introduced in this release. A few cells will not run if you have an older version installed.

You will need Python 2.7 or later installed. Almost all of my work is done in Python 2.7, but I periodically test on 3.3. I do not promise any specific check in will work in 3.X, however. I do 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 3.14" into the book your script will fail; you must write "print (3.4)" as in Python 3.X.

You will need a recent version of NumPy, SciPy, and Matplotlib installed. I don't really know what the minimal might be. I have numpy 1.71, SciPy 0.13.0, and Matplotlib 1.3.1 installed on my machines.

Personally, I use the Anaconda Python distribution in all of my work, available here. I am not selecting them out of favoritism, I am merely documenting my environment. Should you have trouble running any of the code, perhaps knowing this will help you.

## 0.4 Provided Libraries

I've not structured anything nicely yet. For now just look for any .py files in the base directory. As I pull everything together I will turn this into a python library, and probably create a separate git project just for the python code.

There are python files with a name like *xxx*_internal.py. I use these to store functions that are useful for the book, but not of general interest. Often the Python is the point and focus of what I am talking about, but sometimes I just want to display a chart. IPython Notebook does not allow you to collapse the python code, and so it sometimes gets in the way. Some IPython books just incorporate .png files for the image, but I want to ensure that everything is open - if you want to look at the code you can.

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.

## 0.5 License

Kalman Filters and Random Signals in Python by Roger Labbe is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Based on a work at https://github.com/rlabbe/Kalman-Filters-and-Random-Signals-in-Python.

## 0.6 Contact

```
In [1]: #format the book
        import book_format
        book_format.load_style()

Out[1]: <IPython.core.display.HTML at 0x2c2e590>
```

# Chapter 1

# Signals and Noise

```
In [1]: #format the book
        %matplotlib inline
        from __future__ import division, print_function
        import matplotlib.pyplot as plt
        import book_format
        book_format.load_style()

Out[1]: <IPython.core.display.HTML at 0x10e5990>
```

# Chapter 2

# The g-h Filter

```
In [1]: #format the book
        %matplotlib inline
        from __future__ import division, print_function
        import matplotlib.pyplot as plt
        import book_format
        book_format.load_style()

Out[1]: <IPython.core.display.HTML at 0xdf8890>
```

## 2.1 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 coworker comes running up to you and announces her invention to you. After she explains, you eagerly stand on it and announce the results: "172 lbs". You are estatic - 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 coworker 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.

"What? It read 172 lbs just a few seconds ago" you complain to your coworker.

"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 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 does not inform 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 "160lb", and the second (B) reads "170lbs". 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 either A or B
- We could choose a number greater than either A or 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 more than 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 measure. 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 probably 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 youself what would be the 'usual' thing to happen if we made one million separate readings. Some of the times both scales will read too low, sometimes that will both 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 180lbs. 160lbs is a big error. But if we choose a weight between 160lbs and 170lbs our estimate will be better than 160lbs. 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 just the average of A and B. $\frac{160+170}{2} = 165$.

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 just use it instead of B? Can B possibly improve our knowledge over A alone?

The answer, perhaps counterintuitively, is yes, it can. Consider this case. We know scale A is accurate to 1 lb. In other words, if we weight 170 lbs, it could report 169,170, or 171 lbs. We know that scale B is accurate to 9 lbs. We do a reading, and A=160, and B=170. What should we estimate our weight to be?

Well, if we say 160 lbs we would be wrong, because B can only be 9 pounds off, and 170 lbs - 160 lbs is 10 lbs. 160 is not a possible measurement for B. In fact, the only number that satifies all of the constraints is 161 lbs. That is 1 lb within the reading of A, and 9 lbs within the reading of B.

This is an important result. With two relatively inaccurate sensors we were able to deduce an extremely accurate result. Now sure, that was a specially constructed case, but it generalizes. What if A is accurate to 3 lbs, B is accurate to 11 lbs, and we get the measurements of A=160 lbs and B=170 lbs? The result can only be from 159 lbs to 163 lbs, which is better than the range of 157 lbs to 163 lbs that is the range of values that A alone allows.

So two sensors, even if one is less accurate than the other, is better than one.

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.

So, 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 1,000,000 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. I will not prove it, but it can be proved that the average of a large number of weighings will be extremely close to my actual weight. Consider a simple case - the scale is accurate to within 1 lb. If I weigh 170, it will return one of 169, 170, and 171. The average of a bunch of 170 is 170, so we can exclude those. What is left is measurements of 169 and 171. But we know there will be as many 169s as there are 171s. The average of those will also be 170, and so the average of all must be 170, my true weight. It's not that hard to extend this to any arbitrary accuracy.

Okay, great! We have an answer, and phone the UB Thin chain to tell them the good news. As you might expect, their CEO does not share your enthusiam. First, no one has the patience to weigh themselves a million, or even a hundred times. Second, people are far less interested in what they weigh than whether they are actually losing weight or not.

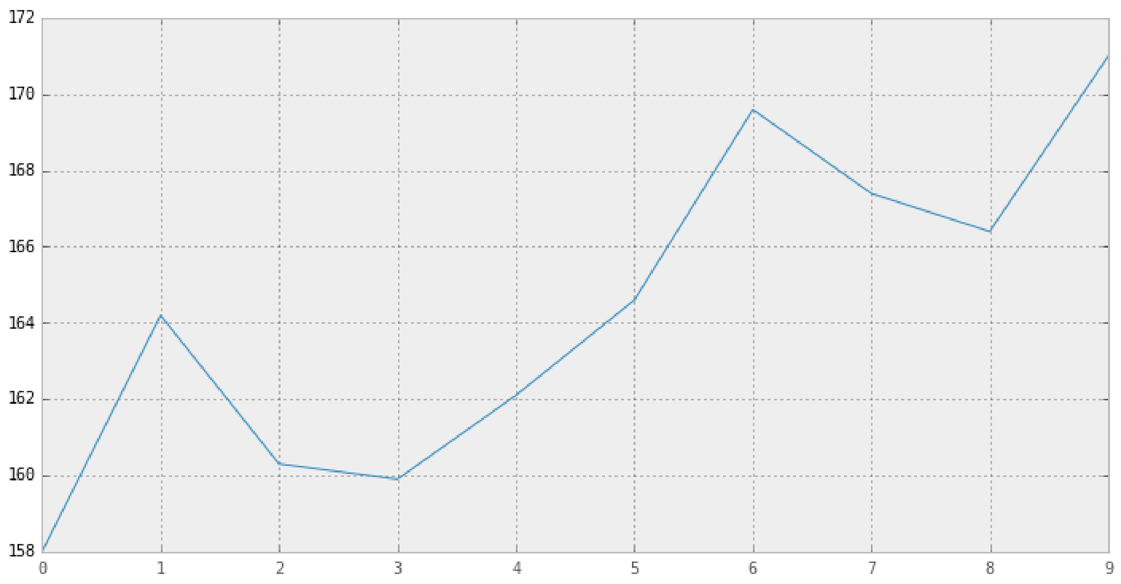
So, let's play 'what if' again. What if you measured your weight once a day, and got the readings 170, 161, and then 169. Did you gain weight, lose weight, or is this all just noisy measurements?

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 - bad measurements. 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! The CEO of UB Thin will not be interested in buying this.

Shall we give up? No, let's play 'what if'. 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, but it seems pretty likely that my weight held steady.

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.

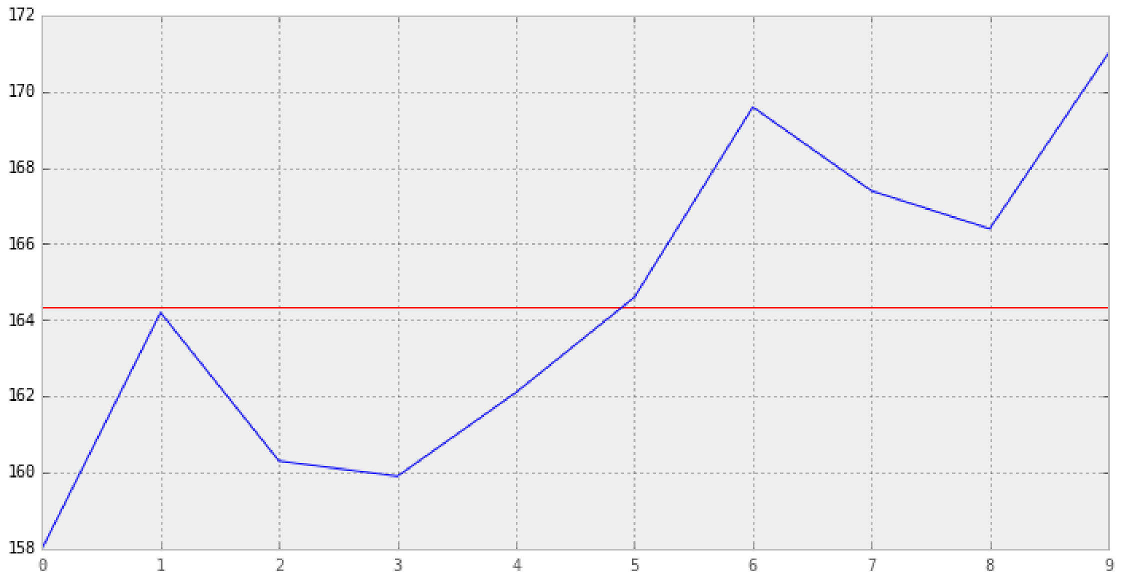
```
In [2]: %matplotlib inline
        from __future__ import division, print_function
        import matplotlib.pyplot as plt
        import matplotlib.pylab as pylab
        import numpy as np

        # make all plots a reasonable size
        pylab.rcParams['figure.figsize'] = 12,6

        weights = [158.0, 164.2, 160.3, 159.9, 162.1, 164.6, 169.6, 167.4, 166.4, 171.0]
        plt.plot(weights)
        plt.show()
```



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. Lets look at that in a chart. We can't be sure, but that surely looks like a weight gain, and a signifant weight gain at that. Let's test this assumption with some more plots. It is easier to 'eyeball' data sometimes.

So let's look at two hypothesis. First, let's assume we held the same weight. To get that number, we agreed that we should just average all the measurements. Let's look at that.

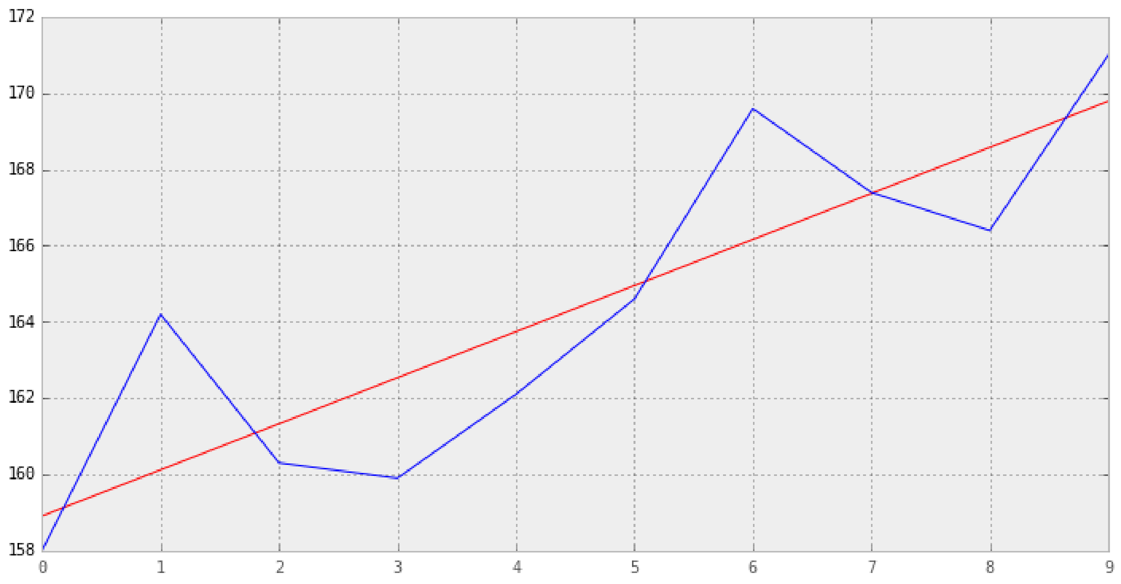
```
In [3]: import numpy as np
        ave = np.sum(weights) / len(weights)
        plt.plot([0,9], [ave,ave], c='r')
        plt.plot(weights, c='b')
        plt.show()
```



That doesn't look very convincing.

Now, let's assume we we gained weight. How much? I don't know, but numpy does! We just 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 squared fit". Let's not worry about the details of that computation, and just plot the results.

```
In [4]: xs = range(len(weights))
        line = np.poly1d(np.polyfit(xs, weights, 1))
        plt.plot (xs, line(xs), c='r')

        plt.plot(weights, c='b')
        plt.show()
```



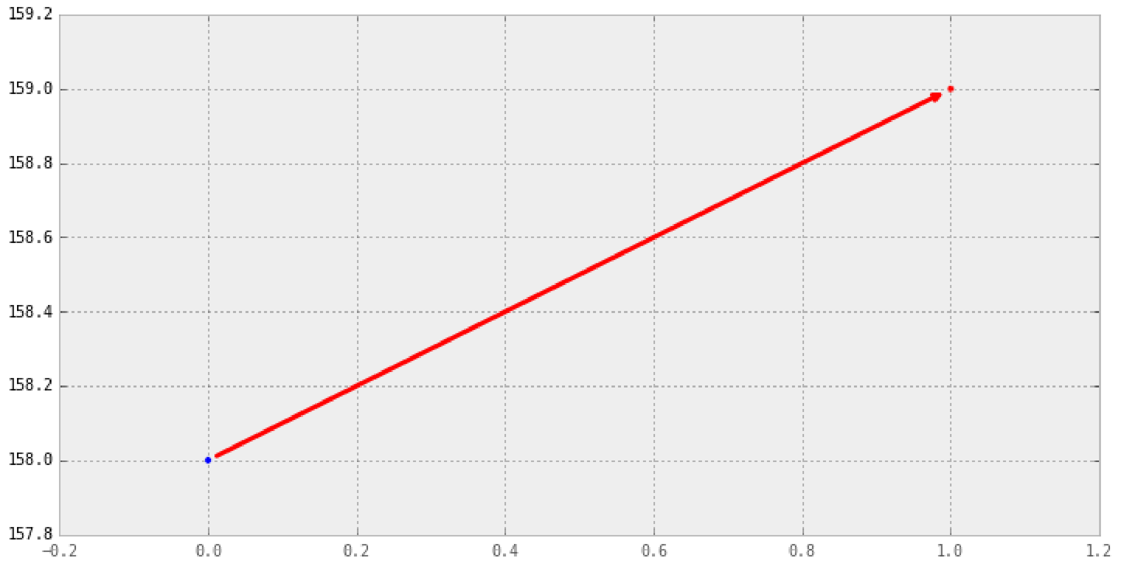
This looks much better, at least to my eyes. It seems far more likely to be true that I gained weight than I didn't gain any weight. Did I actually gain 13 lbs? Who can say? That seems impossible to answer.

"But is it impossible?" pipes up a coworker.

Let's try something crazy. Let's assume that somehow I know I am gaining about one lb a day. It doesn't matter how I know that, right now, just assume I know it somehow. Maybe I am eating a 6000 calorie a day diet, which would result in such a weight gain. Maybe there is another way. Let's just see if we can make use of such information if it was available.

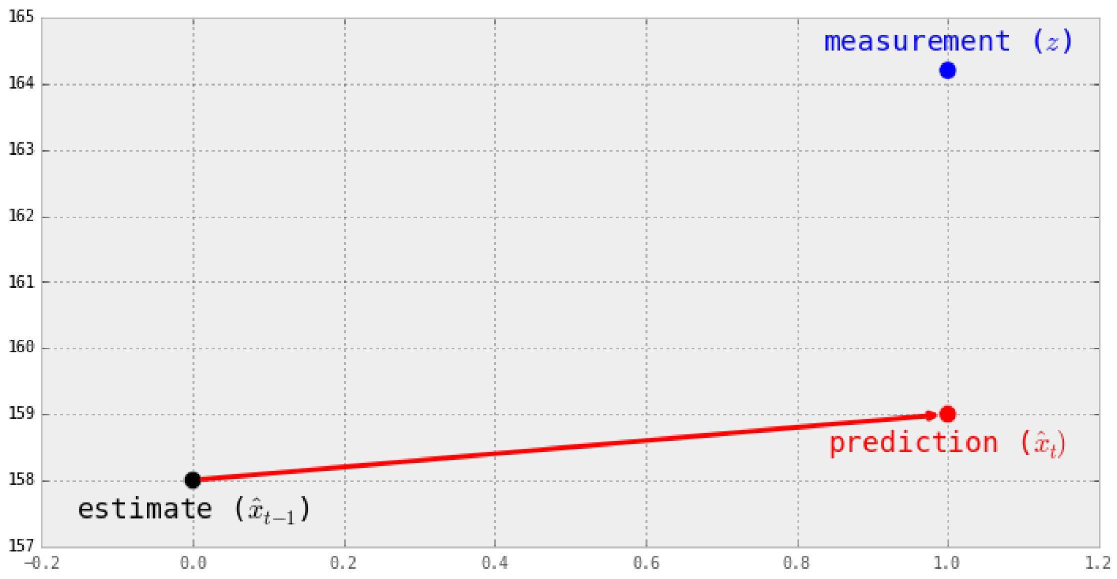
The first measurement was 158. We have no way of knowing any different, so let's just accept that as our estimate. If our weight today is 158, what will it be tomorrow? Well, we think we are gaining weight at 1 lb/day, so our prediction is 159, like so:

```
In [5]: import gh_internal
        gh_internal.plot_estimate_chart_1()
```



Okay, but what good is this? Sure, we could just assume the 1 lb/day is accurate, and just predict our weight for 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.

```
In [6]: gh_internal.plot_estimate_chart_2()
```



Here the measurement is in blue, the previous estimate (output of the filter) is black, and the estimate is red. So 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.

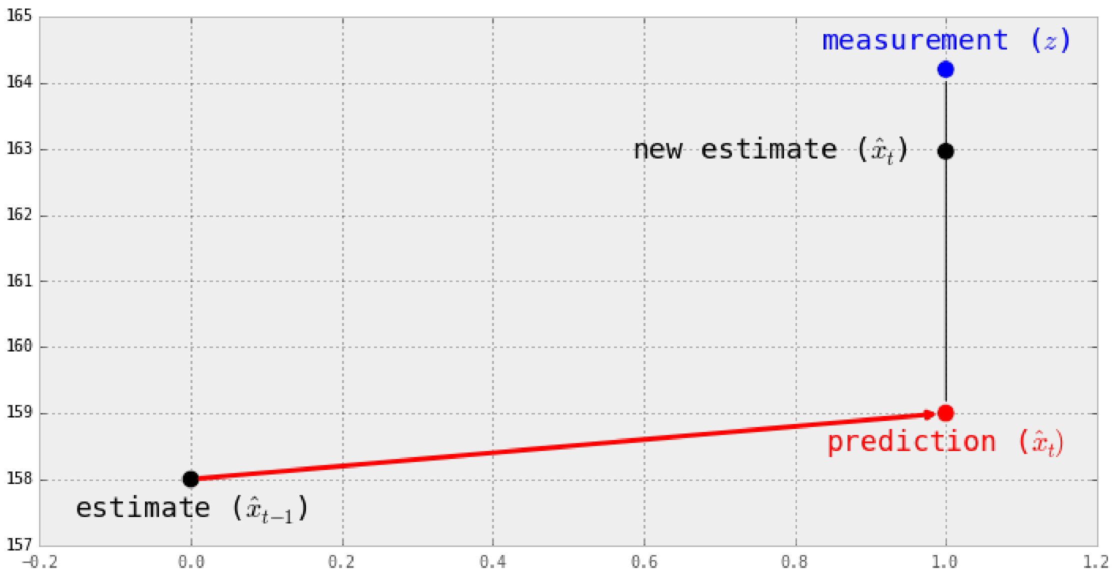
The key insight to this entire book follows. Read it carefully!

So what do we do? If we only take data from the measurement than the prediction will not affect the result. If we only take data 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.

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 like between 159 and 164.2.

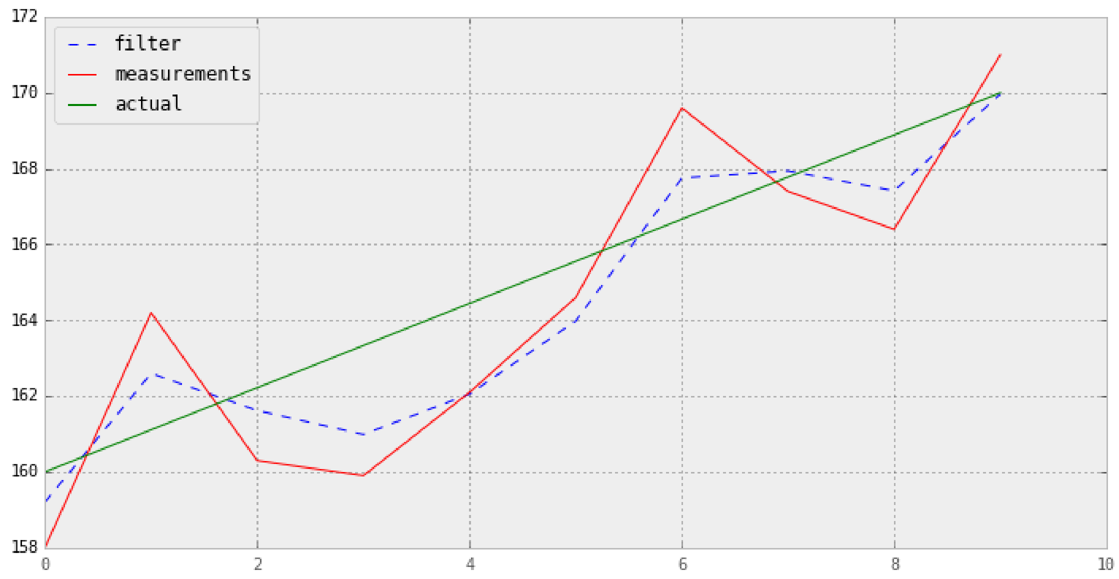
Should it be half way? Maybe, but in general it seems like we might know that our prediction is very accurate or very inaccurate. Probably the accuracy of our prediction differs from the accuracy of the scale. Recall what we did when A was much more accurate than B - we scaled the answer to be closer to A than B. Let's look at that in a chart. **ESTIMATE AND PREDICTION SHOULD NOT USE THE SAME SYMBOLOGY HAT**

```
In [7]: gh_internal.plot_estimate_chart_3()
```



Now tet's try a randomly chosen number: $\frac{6}{10}$. Our estimate will be six tenth the measurement and the rest will be from the prediction. Let's just code that up and see the result. 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).

```
In [8]: w = 160
        gain_rate = 1
        time_step = 1
        scale_factor = 6/10

        # store the filtered results
        estimates = []

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

            # update filter
            w = w * (1-scale_factor) + z * scale_factor
            estimates.append(w)


        plt.xlim([0,10])

        p1, = plt.plot (estimates, 'b--')
        p2, = plt.plot(weights, c='r')
        p3, = plt.plot([0,9],[160,170],c='g')
        plt.legend([p1,p2,p3], ['filter', 'measurements', 'actual'], 2)
        plt.show()
```

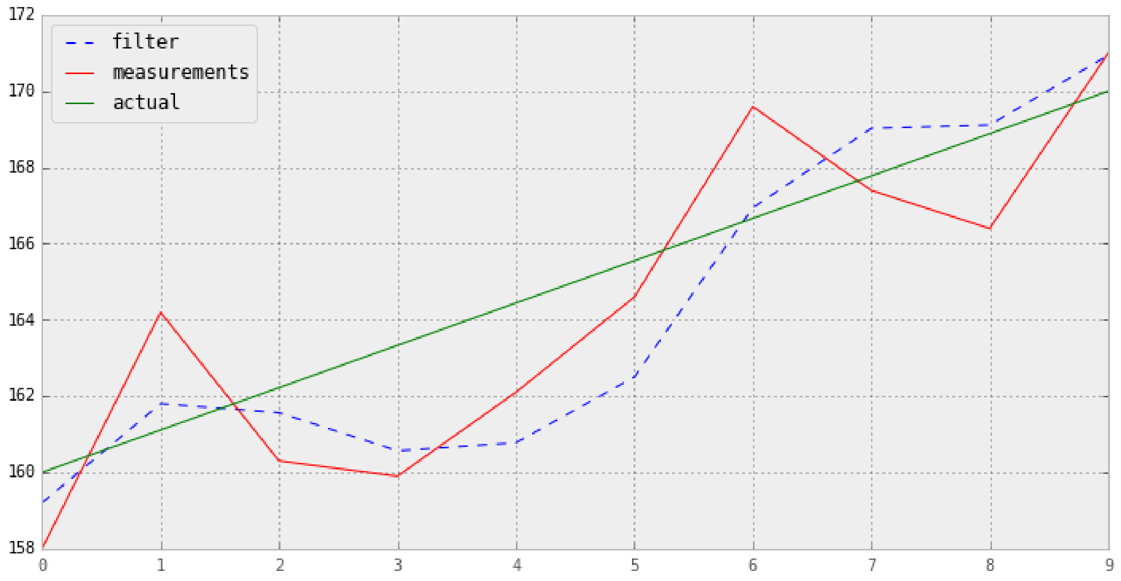


That is pretty good! The blue dots, showing our estimates, are not a straight line, but they are straighter than the measurements and somewhat close to the trend line we created. Also, it seems to get better over time. This may strike you as silly; of course the data will look good if we assume the conclusion, that our weight gain is around 1 lb.

But, what if? What if instead of just leaving the weight gain at 1 lb, we compute it from our estimate. In other words, our first estimate is 162.28. We started at 158; this implies a weight gain not of 1, but 4.28 lbs. Can we use this information somehow? It kind of 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.

So, should we just set the new gain/day to 4.28 lbs? Hmm, sounds like our same problem again. We have two numbers (the gain/day for yesterday and for today), and want to combine them somehow to get tomorrow's gain. Let's use our same tool, and the only tool we have so far - take a combination of the two. This time I will use another arbitrarily chosen number, $\frac{2}{3}$.

```
In [9]: import numpy.random as random
        w = 160
        gain_rate = 1
        time_step = 1
        weight_scale = 6/10
        gain_scale = 2/3

        estimates = []

        for z in weights:
            # prediction step
            w_prediction = w + gain_rate*time_step
            gain_rate = gain_rate

            # update step
            gain_rate = gain_rate * 1-gain_scale + ((z - w) * gain_scale/time_step)
            w = w_prediction * (1-weight_scale) +  (z * weight_scale)
            estimates.append(w)

        p1, = plt.plot (estimates, 'b--')
        p2, = plt.plot(weights, c='r')
        p3, = plt.plot([0,9],[160,170],c='g')
        plt.legend([p1,p2,p3], ['filter', 'measurements', 'actual'], 2)
        plt.show()
```



I think this is starting to look really good. We used no methodology for choosing our scaling factors of $\frac{6}{10}$ and $\frac{2}{3}$ (actually, they are not particularly good choices for this problem), and we 'luckily' choose 1 lb/day as our initial guess for the weight gain, but otherwise all of the reasoning followed from very reasonable assumptions.

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 next value for **all** state variables, both *weight* and *gain_rate*. In this case we are assuming that the the gain does not vary, but when we generalize this algorithm we will remove that assumption.

## 2.2 The g-h Filter

This algorithm is known as the g-h 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).

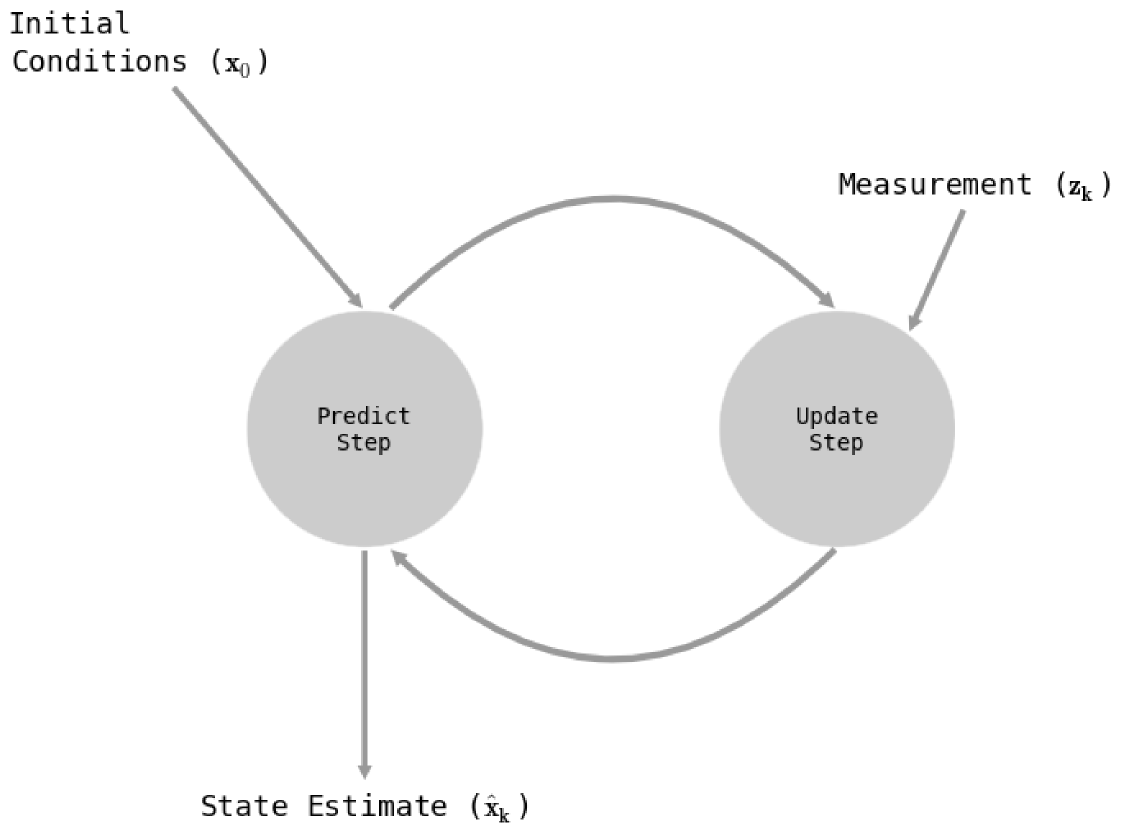
This filter is the basis for a huge number of filters, including the Kalman filter. In other words, the Kalman filter is a form of the g-h filter. So is the Least Squares filter, which you may have heard of, and so is the Benedict-Bordner filter, which you probably have not. Each filter has a different way of assigning values to $g$ and $h$, but otherwise the algorithms are identical. For example, the $\alpha$-$\beta$ filter just assigns a constant to $g$ and $h$, constained to a certain range of values. Other filters will vary $g$ and $h$ dynamically, and filters like the Kalman filter will vary them based on the number of dimensions in the problem.

**Let me repeat the key insights 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 mathematically elaborations to various 'what if' questions we will ask about $g$ and $h$.

- Multiple measurements are more accurate than one measurement
- Always choose a number part way between two measurements to create a more accurate estimate
- Predict the next measurement 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

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

```
In [10]: gh_internal.create_predict_update_chart()
```


Initial
Conditions ($\mathbf{x}_0$)
Measurement ($\mathbf{z_k}$)
Predict
Step
Update
Step
State Estimate ($\hat{\mathbf{x}}_\mathbf{k}$)

I'll begin to introduce the nomenclature and variable names used in the literature. 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 $\mathbf{z_k}$ is the data for this time step. A bold font denotes a vector. So far we have only consided having one sensor, and hence one sensor measurement, but in general we may have $n$ sensors and $n$ measurements. $\mathbf{x}$ denotes our data, and is bold to denote that it is a vector. For example, for our scale example, it represents both the initial weight and inital weight gain rate, like so:

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

Finally, a hat '^' indicates an *estimate*. So the output of the predict step time $k$ at is the *estimate* of our state $\hat{\mathbf{x}}_\mathbf{k}$, and again this is a vector of all the state variables. So for our scale example it containes the estimate of both the weight and the gain rate.

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

## 2.3 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, this is specialized code. Rewrite it to work with any data. Use this function signature:

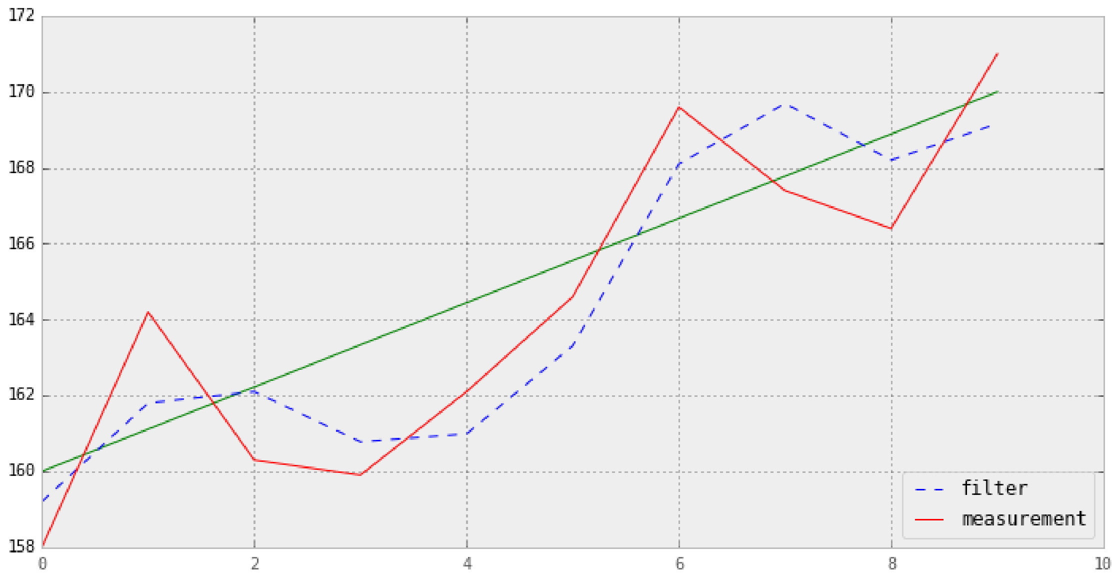
```
def g_h_filter (data, x0, dx, g, h)
    """
    Performs g-h filter on 1 state variable with a fixed g and h.

    '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
    """
```

Test it by passing in the same weight data as before, plot the results, and visually determine that it works.

### 2.3.1 Solution and Discussion

```
In [11]: def g_h_filter (data, x0, dx, g, h, dt=1., pred=None):
             """
             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
             'pred' is an optional list. If provided, each prediction will
             be stored in it
             """

             x = x0
             results = []
             for z in data:
                 #prediction step
                 x_est = x + (dx*dt)
                 dx = dx

                 if pred is not None:
                     pred.append(x_est)

                 # update step
                 residual = z - x_est
                 dx = dx    + h * (residual) / dt
                 x  = x_est + g * residual

                 results.append(x)
```

```
    return results

def plot_g_h_results (measurements, filtered_data, title=''):
    p1, = plt.plot (filtered_data, 'b--')
    p2, = plt.plot(measurements,c='r')
    plt.legend((p1,p2), ('filter', 'measurement'), 4)
    plt.title(title)
    plt.show()


plt.xlim([0,10])
plt.plot([0,9],[160,170],c='g')
data = g_h_filter (data=weights, x0=160, dx=1, g=6./10, h = 2./3, dt=1.)
plot_g_h_results (weights, data)
```



Note that I rewrote the equations somewhat:

$$\begin{aligned}\hat{x}_t &= gz + \hat{x}_{t-1}(1-g)\\ &= gz + \hat{x}_{t-1} - g\hat{x}_{t-1}\\ &= \hat{x}_{t-1} + g*(z-\hat{x}_{t-1})\\ &= \hat{x}_{t-1} + g*residual\end{aligned}$$

I'll let you decide which form of the equation is more expressive. One form explicitly uses $g$ and $(1-g)$ to compute the point between the two values, the other finds the difference between the points, and adds a fraction of that to the first value. Both are computing the same thing. You'll see both forms in the literature, so I have used both to expose you to them.

## 2.4 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 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 $k$ and bases it on the number of state variables there are in $\mathbf{x}$. 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. Eli Brookner's book mentioned in the previous paragraph is an excellent resource for that topic, if it interests you. 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 will be making a number of carefully considered choices to optimize it's performance, and those choices indirectly affect g and h. Don't worry if this is not too clear right now, it will be much clearer later after we develop the Kalman filter theory.

However, 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.

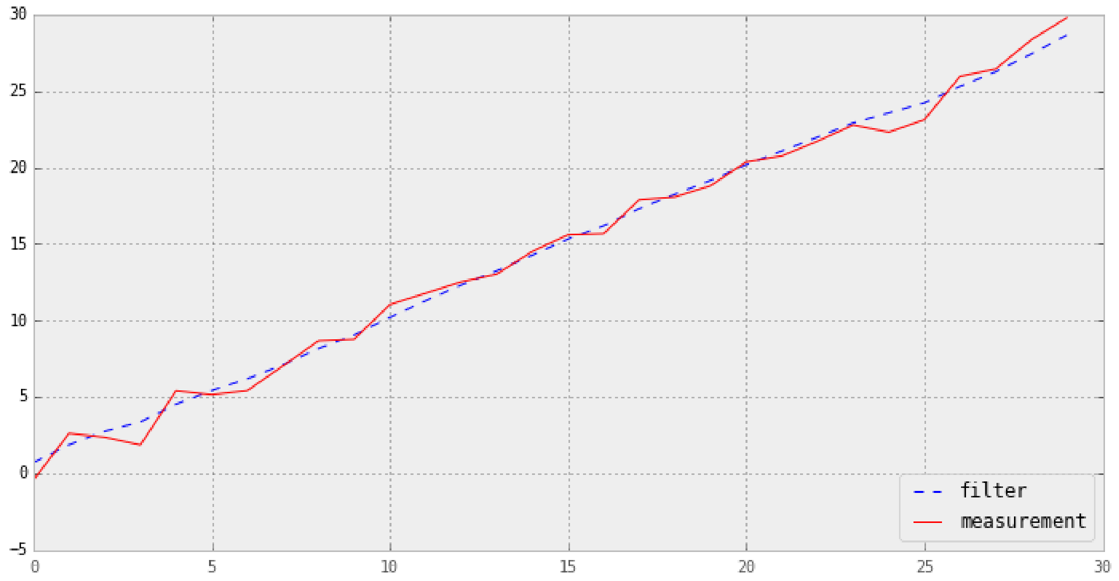
## 2.5 Exercise: create measurement function

Now let's write a function that generates noisy data for us. Recall from chapter 0 that we model a noisy signal as the signal plus white noise 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 [12]: # your code here
```

### 2.5.1 Solution

```
In [13]: def gen_data (x0, dx, count, noise_factor):
             return [x0 + dx*i + random.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)
```



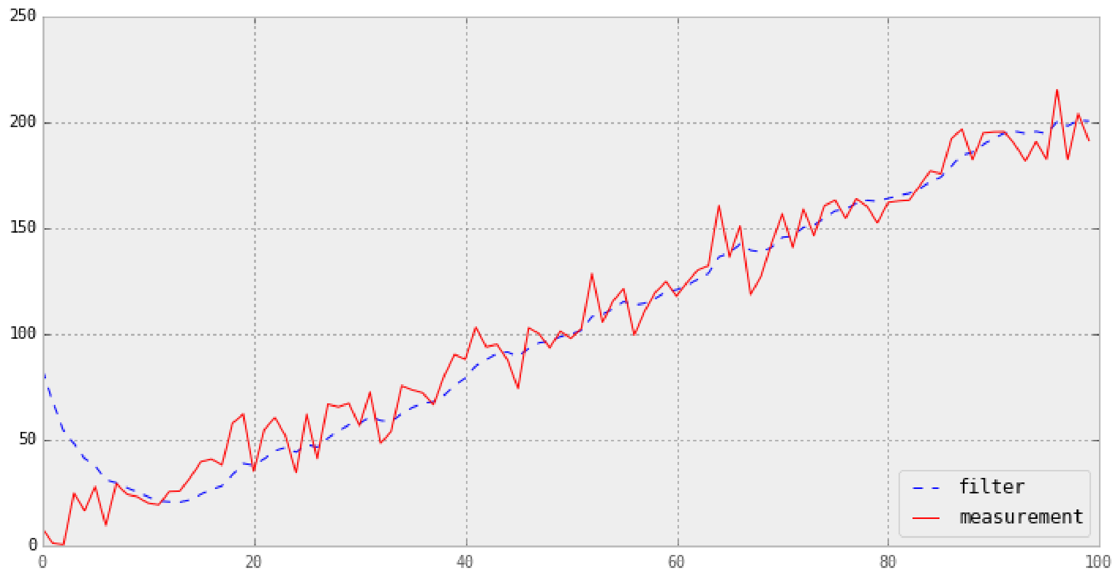
## 2.6 Exercise: Bad Initial Conditions

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 you initial guess for x to be 100.

```
In [14]: # your code here
```

### 2.6.1 Solution and Discussion

```
In [15]: 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=0.01)
         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 sinusodial 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.

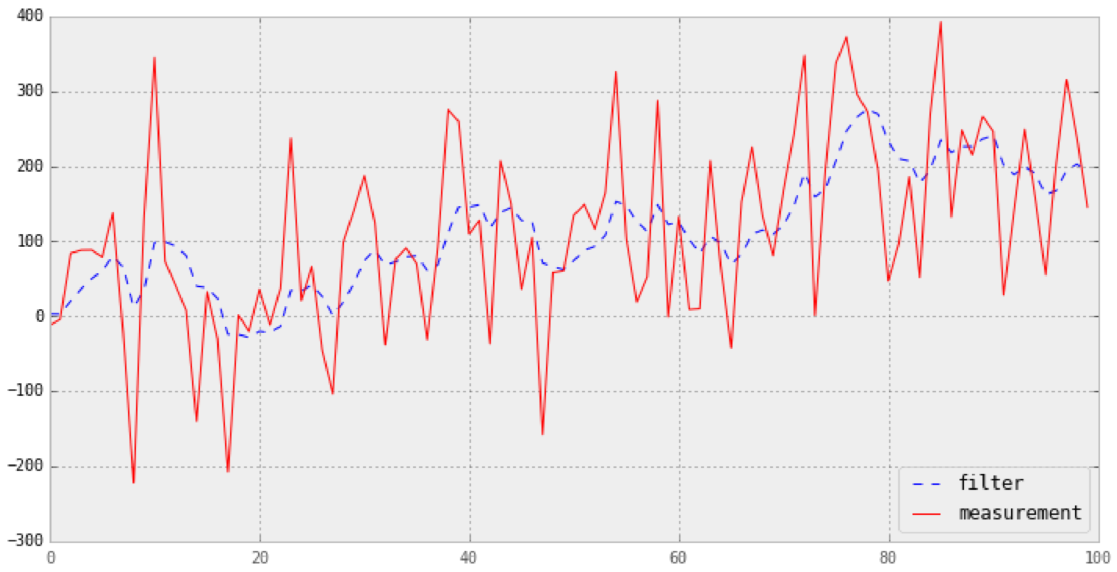
## 2.7 Exercise: Extreme Noise

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 [16]: # your code here
```

### 2.7.1 Solution and Discussion

```
In [17]: 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 noisey 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 with no noise 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 existance of the multitude of chapters beyond this one should suggest that we can do much better than this.

## 2.8 Excercise: The Effect of Acceleration

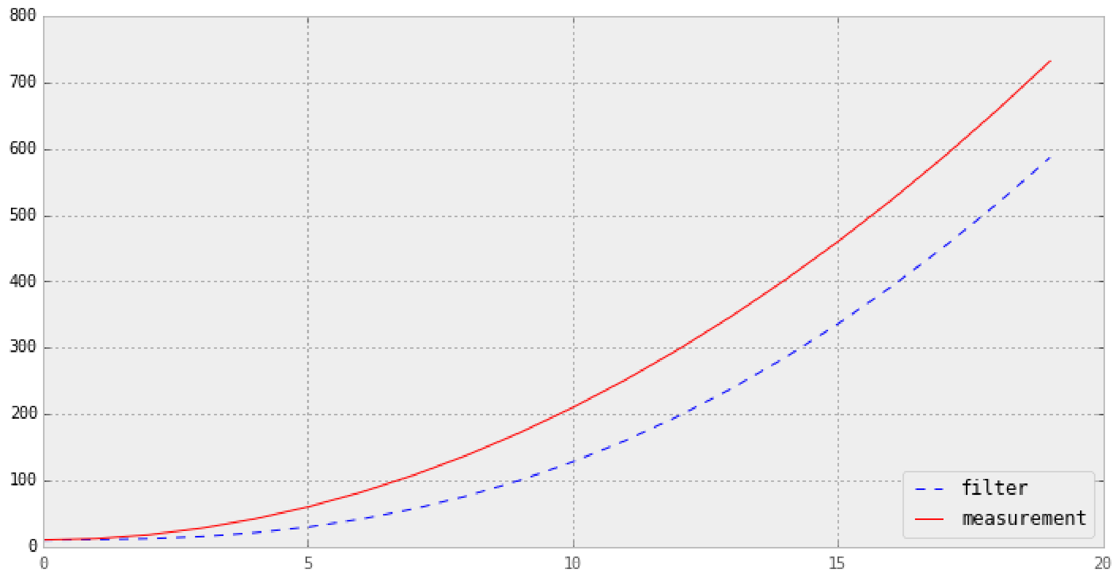
Write a new data generation function that adds in a constant acceleration factor to each data point. In other words, increment dx as you compute each data point so that the velocity (dx) is ever increasing. Set the noise to 0, $g = 0.2$ and $h = 0.02$ and plot the results. Explain what you see.

```
In [18]: # your code here
```

### 2.8.1 Solution and Discussion

```
In [19]: def gen_data (x0, dx, count, noise_factor, accel=0):
             zs = []
             for i in range (count):
                 zs.append (x0 + dx*i + random.randn()*noise_factor)
                 dx += accel
             return zs

         predictions = []
         zs = gen_data (x0=10, dx=0, count=20, noise_factor=0, accel = 2)
```

```
data = g_h_filter (data=zs, x0=10, dx=0, g=0.2, h=0.02, pred=predictions)
plt.xlim([0,20])
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.

## 2.9 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 of a large value of g will be? A small value?

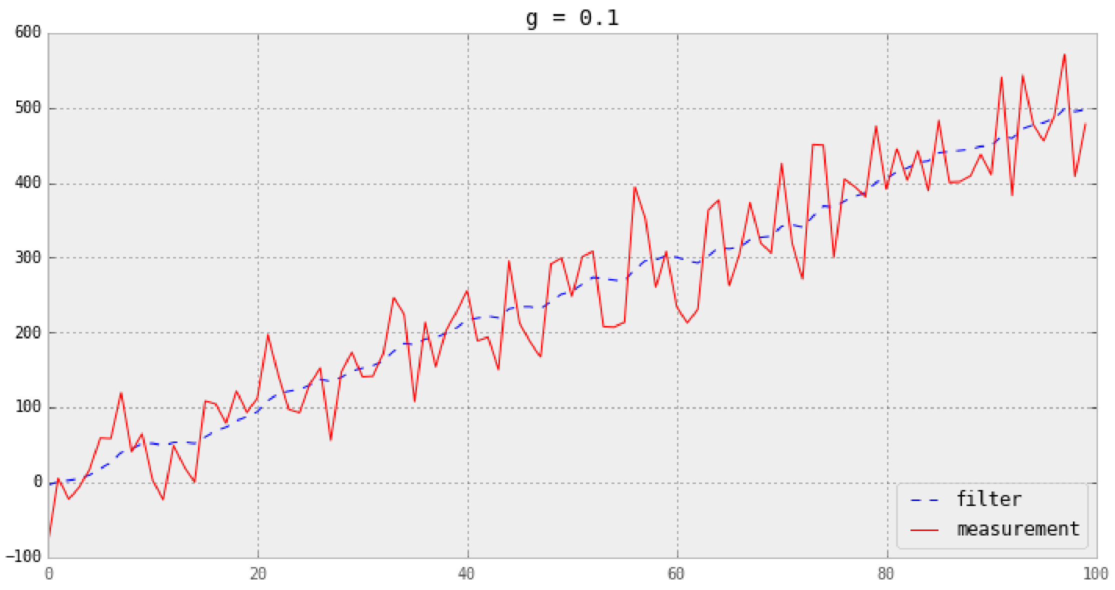
Now, let the $noise\_factor = 50$ and $dx = 5$. Plot the results of $g = 0.1,\ 0.5,$ and $0.9$.

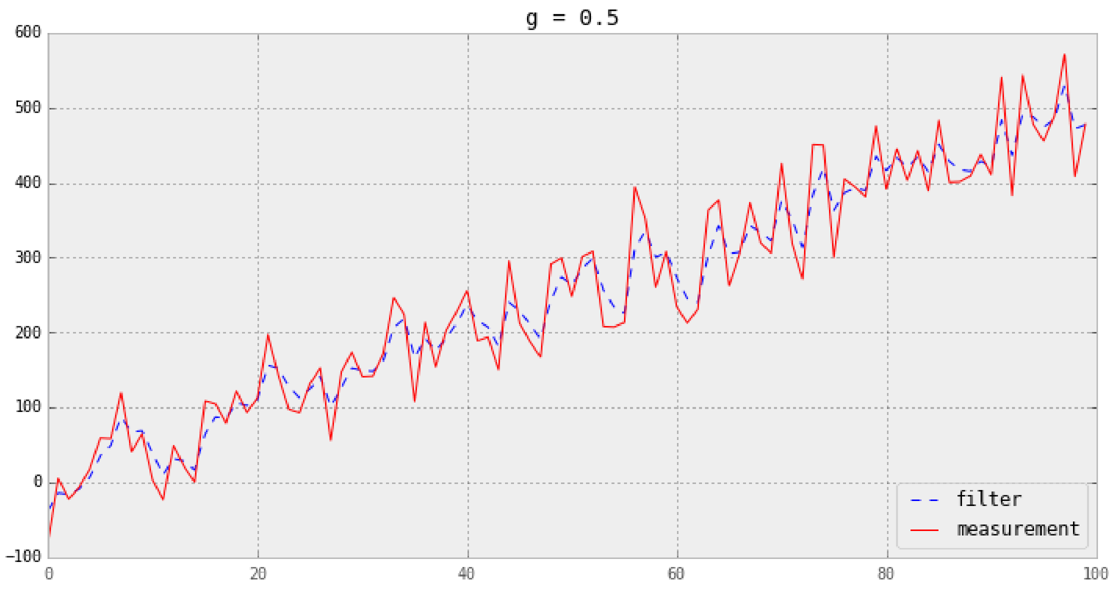
```
In [20]: # your code here
```

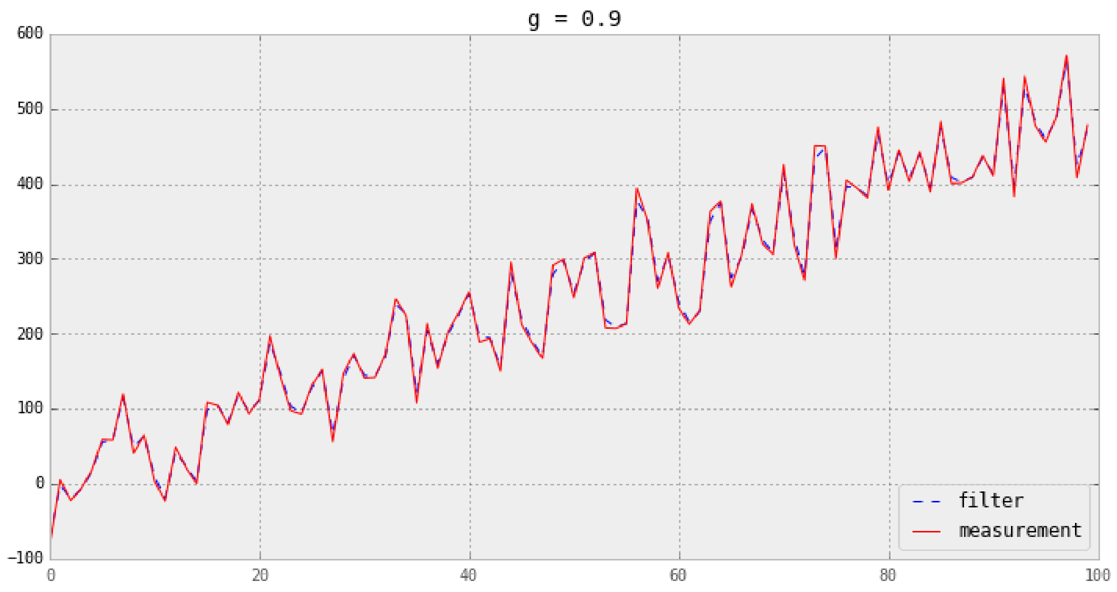
### 2.9.1 Solution and Discussion

```
In [21]: zs = gen_data (x0=5, dx=5, count=100, noise_factor=50)

         data = g_h_filter (data=zs, x0=0., dx=5., dt=1.,g=0.1, h=0.01)
         plot_g_h_results  (zs, data, 'g = 0.1')

         data = g_h_filter (data=zs, x0=0., dx=5., dt=1.,g=0.5, h=0.01)
```

```
plot_g_h_results  (zs, data, 'g = 0.5')

data = g_h_filter (data=zs, x0=0., dx=5., dt=1.,g=0.9, h=0.01)
plot_g_h_results  (zs, data, 'g = 0.9')
```

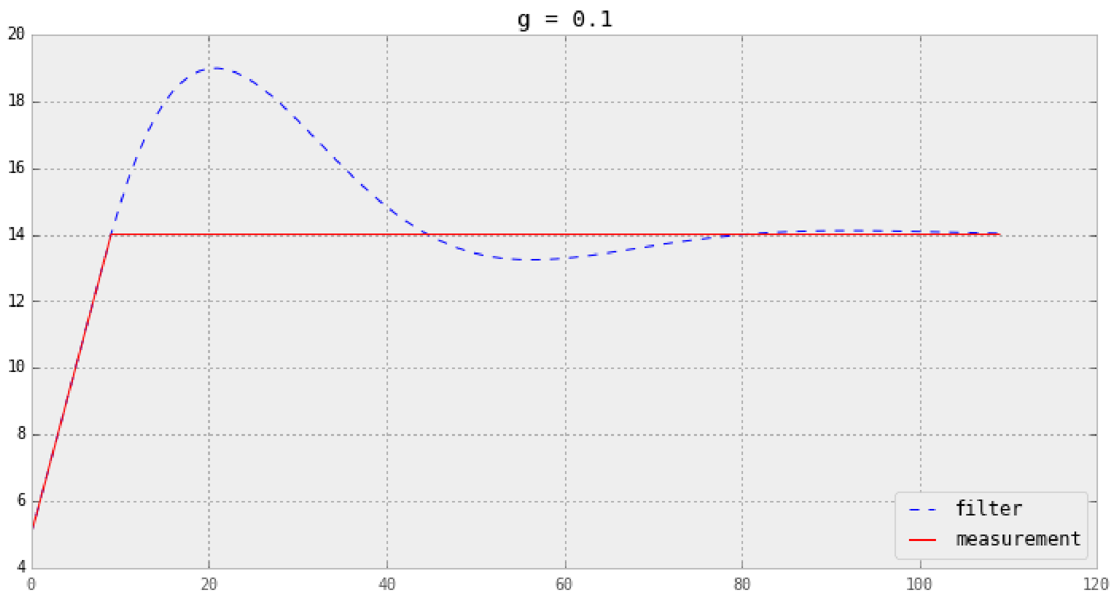


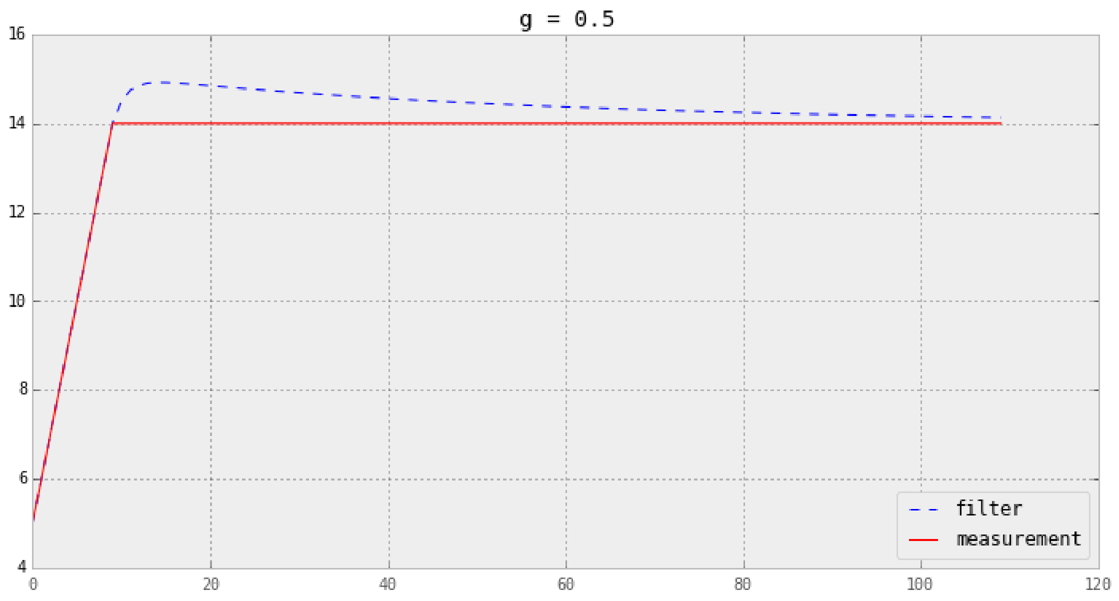


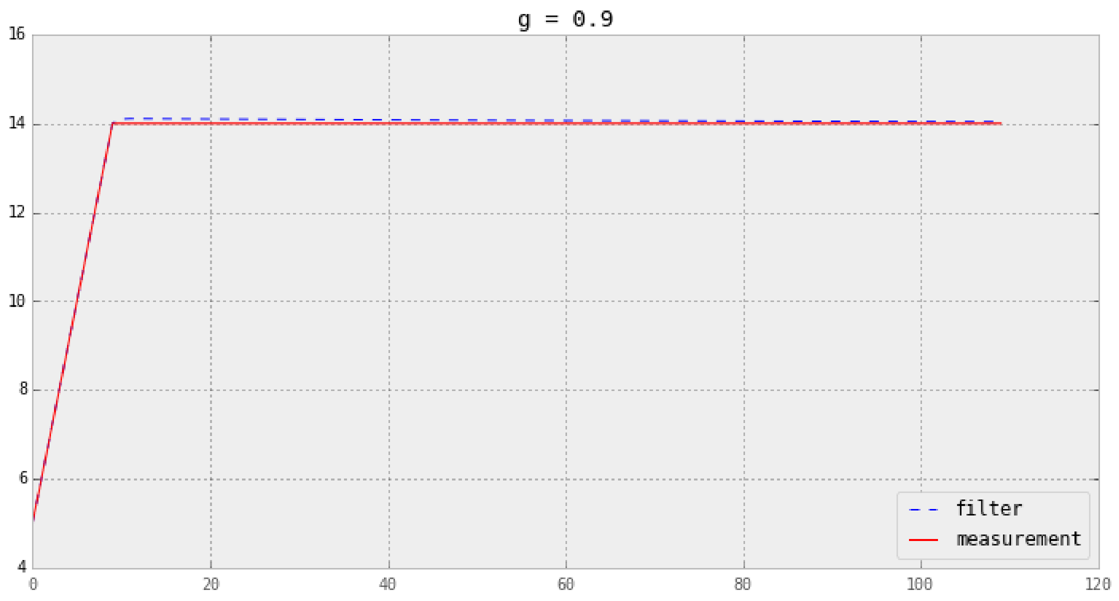


It is clear that as $g$ is larger we more closely follow the measurement instead of the prediction. When $g = 0.9$ we follow the signal almost exactly, and reject almost none of the noise. One might naively conclude that $g$ should always be very small to maximize noise rejection. However, that means 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 look. I will create data that has $\dot{x} = 1$ for 9 steps before changing to $\dot{x} = 0$.

```
In [22]: zs = [5,6,7,8,9,10,11,12,13,14]

         for i in range(100):
             zs.append(14)

         data = g_h_filter (data=zs, x0=4., dx=1., dt=1.,g=0.1, h=0.01)
         plot_g_h_results  (zs, data, 'g = 0.1')

         data = g_h_filter (data=zs, x0=4., dx=1., dt=1.,g=0.5, h=0.01)
         plot_g_h_results  (zs, data, 'g = 0.5')

         data = g_h_filter (data=zs, x0=4., dx=1., dt=1.,g=0.9, h=0.01)
         plot_g_h_results  (zs, data, 'g = 0.9')
```


g = 0.1
filter
measurement
0
20
40
60
80
100
120
4
6
8
10
12
14
16
18
20

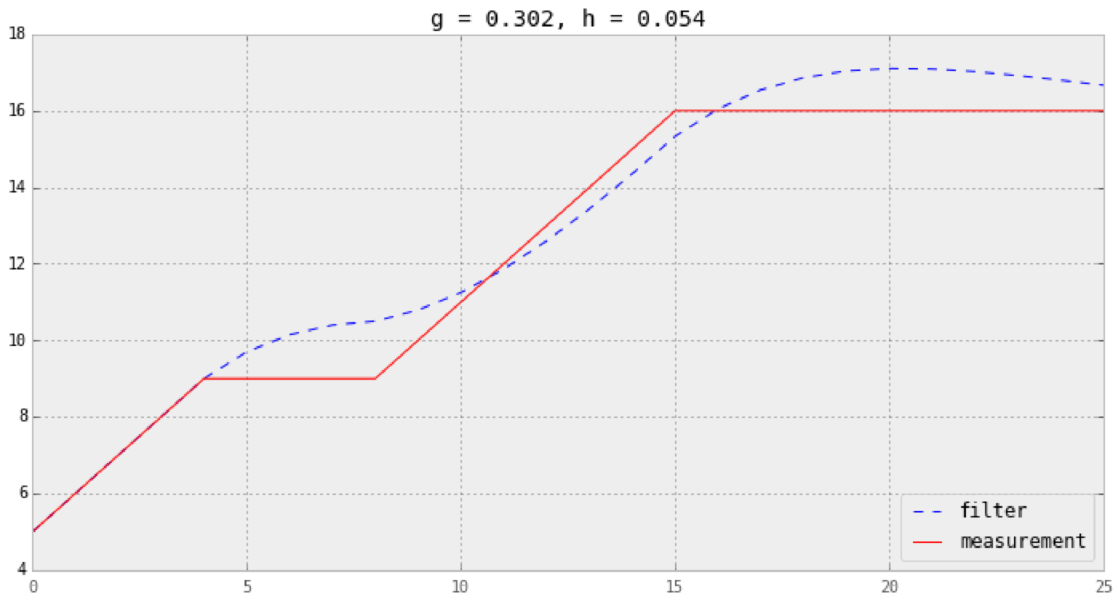

g = 0.5
filter
measurement
0
20
40
60
80
100
120
4
6
8
10
12
14
16

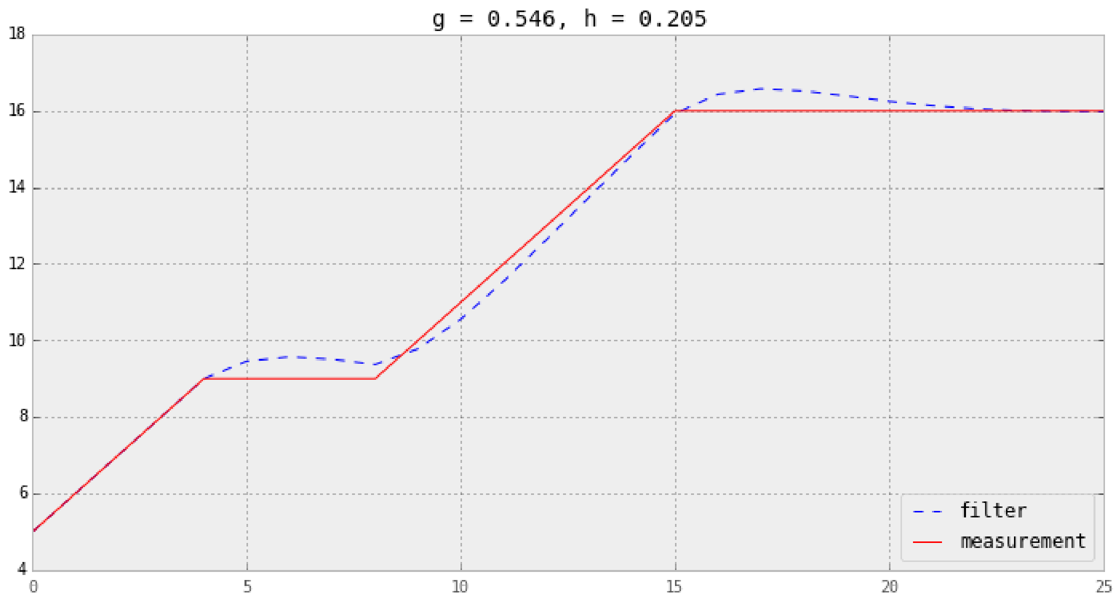


Here we can see the effects of ignoring the signal. We not only filter out noise, but legitimate changes in the signal as well.

Maybe we need a 'Godilocks' filter, where 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 [23]: 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]

         data = g_h_filter (data=zs, x0=4., dx=1., dt=1.,g=.302, h=0.054)
         plot_g_h_results  (zs, data, 'g = 0.302, h = 0.054')

         data = g_h_filter (data=zs, x0=4., dx=1., dt=1.,g=.546, h=0.205)
         plot_g_h_results  (zs, data, 'g = 0.546, h = 0.205')
```



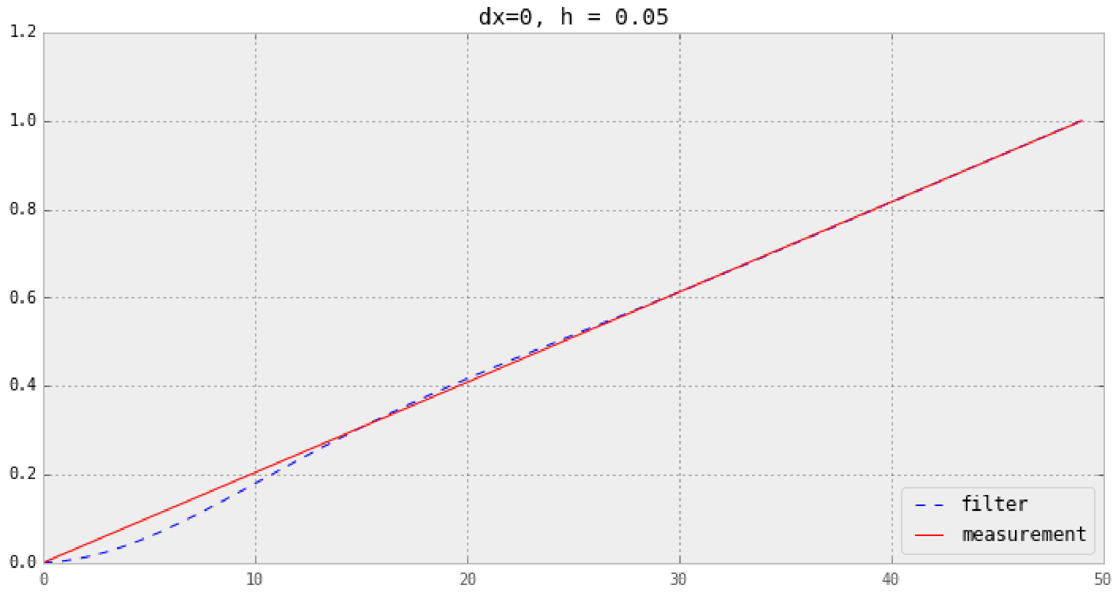


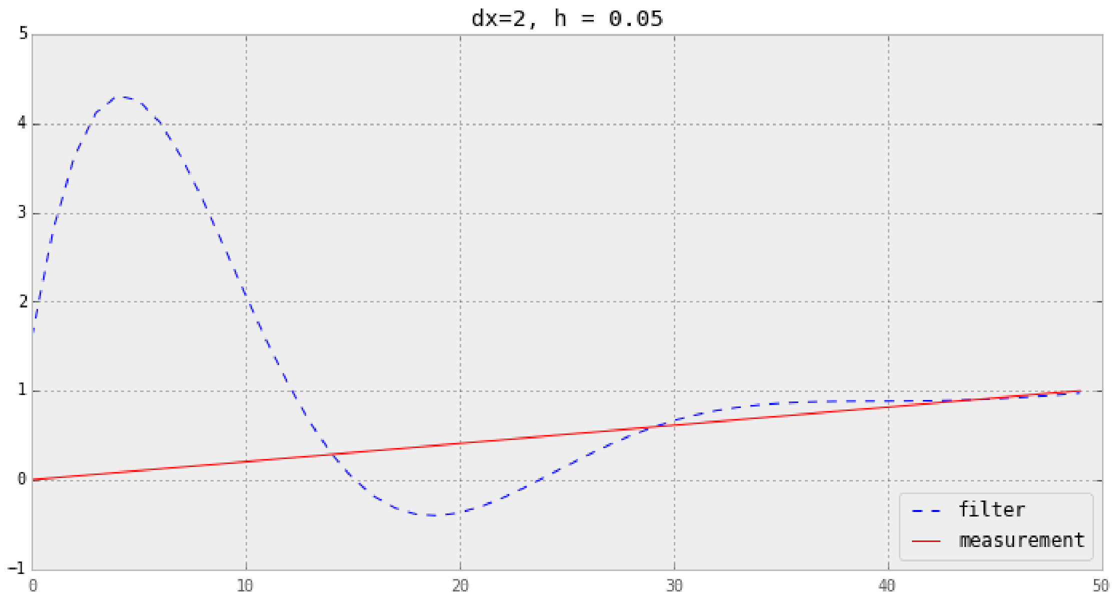
## 2.10 Varying h

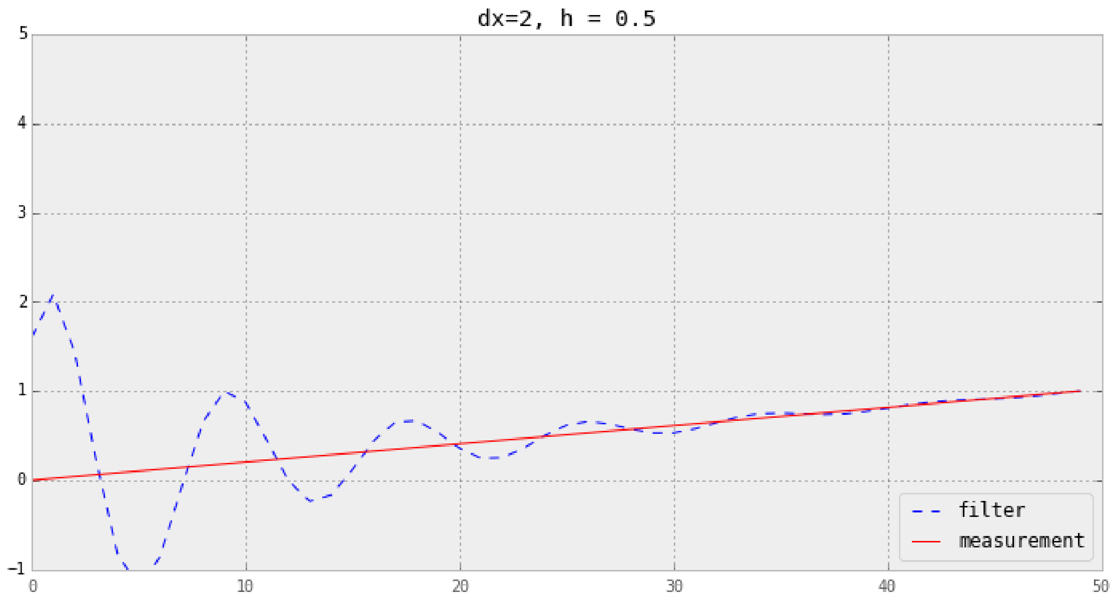
Now let's leave g unchanged and investigate the effect of modifying h. We know that h affects how much of we favor the measurement of $\dot{x}$ vs our predictiton. But what does this *mean*? If our signal is changing a lot (quickly relative to the time step of our filter), then a large $h$ will cause 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 [24]: zs = np.linspace(0,1,50)

         data = g_h_filter (data=zs, x0=0, dx=0., dt=1.,g=.2, h=0.05)
         plot_g_h_results  (zs, data, 'dx=0, h = 0.05')

         data = g_h_filter (data=zs, x0=0, dx=2., dt=1.,g=.2, h=0.05)
         plt.ylim([-1,5])
         plot_g_h_results  (zs, data, 'dx=2, h = 0.05')

         data = g_h_filter (data=zs, x0=0, dx=2., dt=1.,g=.2, h=0.5)
         plt.ylim([-1,5])
         plot_g_h_results  (zs, data, 'dx=2, h = 0.5')
```







## 2.11 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 direcly when designing Kalman filters. Furthermore, due to your choices $g$ 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.

missing: properties - recursive - markov - random variable should we fit bayes into here, or just leave to the next chapter chart showing recursive loop

# Chapter 3

# Discrete Bayes Filter

```
In [1]: #format the book
        %matplotlib inline
        from __future__ import division, print_function
        import matplotlib.pyplot as plt
        import book_format
        book_format.load_style()
```

```
Out[1]: <IPython.core.display.HTML at 0xf9c990>
```

The Kalman filter belongs to a family of filters called *bayesian filters*. Without going into blah blah

## 3.1 Tracking a Dog

Let us begin with a simple problem. We have a dog friendly workspace, and so people bring their dogs to work. However, occasionally the dogs wander out of your office and down the halls. We want to be able to track them. So during a hackathon somebody created a little 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 our 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 try 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, we will assume that there are only 10 positions in a single hallway to consider, which we will number 0 to 9, where 1 is to the right of 0, 2 is to the right of 1, and so on. 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. He is equally likely to be in any position. The probability that he is in each position is therefore 1/10

Let us represent our belief of his position at any time in a numpy array.

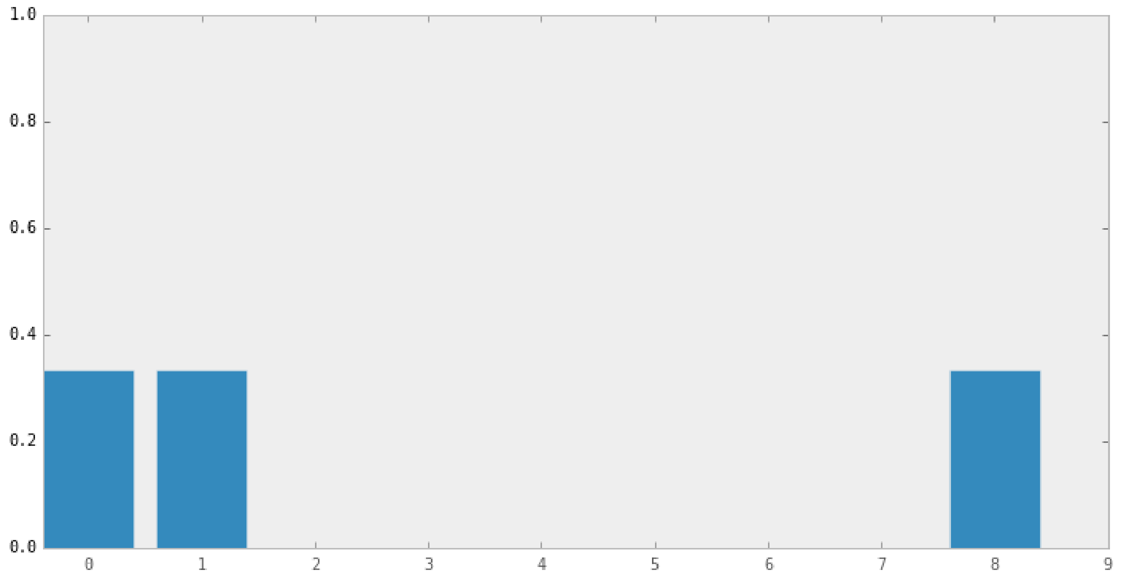
```
In [2]: import numpy as np

        pos = np.array([.1, .1, .1, .1, .1, .1, .1, .1, .1, .1])
```

Now let's create a map of the hallway in another list. Suppose there are first two doors close together, and then another door quite a bit further down the hallway. We will use 1 to denote a door, and 0 to denote a wall:

```
In [3]: hallway = np.array([1, 1, 0, 0, 0, 0, 0, 0, 1, 0])
```

So I start listening to Simon's transmissions on the network, and the first data I get from the sesnor is "door". From this I conclude that he is in front of a door, but which one? I have no idea. I have no reason to believe is in front of the first, second, or third door. But what I can do is assign a probability to each door. All doors are equally likely, so I assign a probability of 1/3 to each door.

```
In [4]: from __future__ import print_function, division
        import matplotlib.pyplot as plt
        import bar_plot
        import numpy as np

        pos = np.array([0.333, 0.333, 0., 0., 0., 0., 0., 0., 0.333, 0.])
        bar_plot.plot (pos)
```



We call this 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 so far we have narrowed down our knowledge in his position to these locations.

I hand coded the pos array in the code above. How would we implement this in code? Well, hallway represents each door as a 1, and wall as 0, so we will multiply the hallway variable by the percentage, like so;

```
In [5]: pos = hallway * 0.3
        print('pos =', pos)
```

```
pos = [ 0.3  0.3  0.   0.   0.   0.   0.   0.   0.3  0. ]
```

## 3.2 Extracting Information from Multiple 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 where Simon is at the end of that sequence? Of course! Given the hallway's layout there is only one place where you can be in front of a door, move once to the right, and be in front of another door, 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'. Therefore the only possibility is that he is now in front of the second door. We denote this in Python with:

```
In [6]: pos = np.array([0,1,0,0,0,0,0,0,0,0])
        print(pos)
```

```
[0 1 0 0 0 0 0 0 0 0]
```

Obviously I carefully constructed the hallway layout and sensor readings to give us an exact answer quickly. Real problems will not be so clear cut. But this should trigger your intuition - the first sensor reading only gave us very low probabilities (0.333) for Simon's location, but after a position update and another sensor reading we knew much 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. For example, suppose we had a long sequence of "door, right, door, right, wall, right, wall, right, door, right, door, right, wall, right, wall, right, wall, right, wall, right, door". Simon could only be located where we had a sequence of [1,1,0,0,1,1,0,0,0,0,1] in the hallway. There might be only one match for that, or at most a few. Either way we will be far more certain about his position then when we started.

We could work through the code to implement this solution, but instead let us consider a real world complication to the problem.

## 3.3 Noisy Sensors

Unfortunately I have yet to come across a perfect sensor. Perhaps the sensor would not detect a door if Simon sat in front of it while scratching himself, or it might report there is a door if he is facing towards the wall, not down the hallway. So in practice when I get a report 'door' I cannot assign 1/3 as the probability for each door. I have to assign something less than 1/3 to each door, and then assign a small probability to each blank wall position. At this point it doesn't matter exactly what numbers we assign; let us say that the probably of 'door' being correct is 0.6, and the probability of being incorrect is 0.2, which is another way of saying it is about 3 times more likely to be right than wrong. How would we do this?

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

The key, as with the problem above, is probabilities. We are already comfortable with assigning a probabilistic belief about the location of the dog; now we just have to incorporate the additional uncertainty caused by the sensor noise. Say we think there is a 50% chance that our dog is in front of a specific door and we get a reading of 'door'. Well, we think that is only likely to be true 0.6 of the time, so we multiply: $0.5 * 0.6 = 0.3$. Likewise, if we think the chances that our dog is in front of a wall is 0.1, and the reading is 'door', we would multiply the probability by the chances of a miss: $0.1 * 0.2 = 0.02$.

However, we more or less chose 0.6 and 0.2 at random; if we multiply the pos array by these values the end result will no longer represent a true probability distribution.

```
In [7]: def measure (pos, measure, p_hit, p_miss):
            q = np.array(pos, dtype=float)
            for i in range(len(hallway)):
                if hallway[i] == measure:
                    q[i] = pos[i] * p_hit
                else:
                    q[i] = pos[i] * p_miss
            return q

        pos = np.array([0.2]*10)
        reading = 1 # 1 is 'door'
        pos = measure (pos, 1, .6, .2)

        print(pos)
        print('sum =', sum(pos))
```

```
[ 0.12  0.12  0.04  0.04  0.04  0.04  0.04  0.04  0.12  0.04]
sum = 0.64
```

We can see that this is not a probability distribution because it does not sum to 1.0. But we can see that the code is doing mostly the right thing - the doors are assigned a number (0.12) that is 3 times higher than the walls (0.04). So we can write a bit of code to normalize the result so that the probabilities correctly sum to 1.0.

```
In [8]: def normalize (p):
            s = sum(p)
            for i in range (len(p)):
                p[i] = p[i] / s

        def sense (pos, measure, p_hit, p_miss):
            q = np.array(pos, dtype=float)
            for i in range(len(hallway)):
                if hallway[i] == measure:
                    q[i] = pos[i] * p_hit
                else:
                    q[i] = pos[i] * p_miss
            normalize(q)
            return q


        pos = np.array([0.2]*10)
        reading = 1 # 1 is 'door'
        pos = sense (pos, 1, .6, .2)

        print('sum =', sum(pos))
        print('probability of door =', pos[0])
        print('probability of wall =', pos[2])
```

```
sum = 1.0
probability of door = 0.1875
probability of wall = 0.0625
```

Normalization is done by dividing each element by the sum of all elements in the list. If this is not clear you should spend a few minutes proving it to yourself algebraically. 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 intuitiion 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.

## 3.4 Incorporating Movement Data

Recall how quickly we were able to find an exact solution to our dog's position when we incorporated a series of measurements and movement updates. However, that occured in a fictional world of perfect sensors. Might we be able to find an exact solution even in the presense of noisy sensors?

Unfortunately, the answer is no. Even if the sensor readings perfectly match an extremely complicated hallway map we could not say that we are 100% sure that the dog is in a specific position - there is, after all, the 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 head-spinningly complicated, but lets just go ahead and program the math, which as we have seen is quite simple.

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 pos array?

I hope after a moment's thought it is clear that we should just 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 the move to the right we should believe that there is a 50% chance he is at position 4. So let's implement that. Recall that the hallway is circular, so we will use modulo arithmetic to perform the shift correctly

```
In [9]: import numpy
        def perfect_update(pos, move):
            """ move the position by 'move' spaces, where positive is to the right, and negative
            is to the left
            """
            n = len(pos)
            result = np.array(pos, dtype=float)
            for i in range(n):
                result[i] = pos[(i-move) % n]
            return result

        pos = np.array([.4, .1, .2, .3])
        print('pos before update =', pos)
        pos = perfect_update(pos, 1)
        print('pos after update =', pos)
```

```
pos before update = [ 0.4  0.1  0.2  0.3]
pos after update = [ 0.3  0.4  0.1  0.2]
```

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.

## 3.5 Adding Noise to the Update

We want to solve real world problems, and we have already stated that all sensors have noise. Therefore the code above must be wrong. What if the sensor reported that our dog moved one space, but he actually moved two spaces, or zero? Once again this may initially sound like an insummountable problem, but let's just model it in math. Since this is just an example, we will create a pretty simple noise model for the sensor - later in the book we will handle far more sophisticated errors.

We will say that when the sensor sends a movement update, it is 80% likely to be right, and it is 10% likely to overshoot one position to the right, and 10% likely to undershoot to the left. That is, if we say the movement was 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.

This is slightly harder than the math we have done so far, but it is still tractable. Each result in the array now needs to incorporate probabilities for 3 different situations. For example, consider position 9 for the case where the reported movement is 2. It should be clear that after the move we need to incorporate the probability that was at position 7 (9-2). However, there is a small chance that our dog actually moved from either 1 or 3 spaces away due to the sensor noise, so we also need to use positions 6 and 8. How much? Well, we have the probabilities, so we can just multiply and add. It would be 80% of position 7 plus 10% of position 6 and 10% of position 8! Let's try coding that:

```
In [10]: def update (pos, move, p_correct, p_under, p_over):
             n = len(pos)
             result = np.array(pos, dtype=float)
             for i in range(n):
                         result[i] =  \
                     pos[(i-move) % n]   * p_correct + \
                     pos[(i-move-1) % n] * p_over + \
                     pos[(i-move+1) % n] * p_under
             return result

         p = np.array([0,0,0,1,0,0,0,0])
         res = update (p, 2, .8, .1, .1)
         print(res)
```

```
[ 0.   0.   0.   0.   0.1  0.8  0.1  0. ]
```

The simple test case that we ran appears to work correctly. We initially believed that the dog was in position 3 with 100% certainty; after the movement update we now give an 80% probability to the dog being in position 5, and a 10% chance to undershooting to position 4, and a 10% chance of overshooting to position 6. Let us look at a case where we have multiple beliefs:

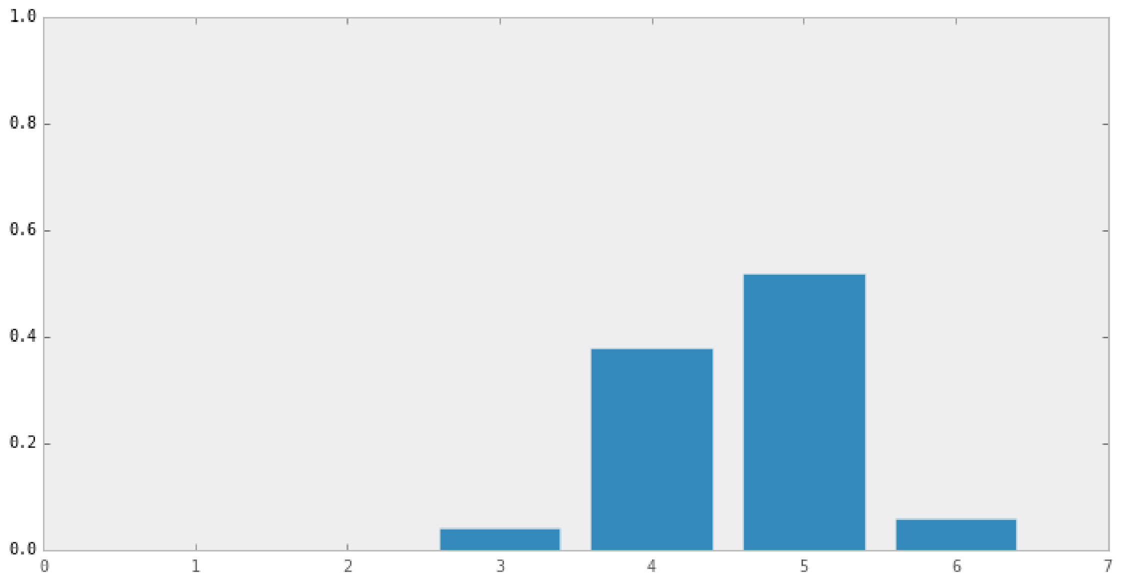
```
In [11]: p = np.array([0, 0, .4, .6, 0, 0, 0, 0])
         res = update (p, 2, .8, .1, .1)
         print(res)
```

```
[ 0.    0.    0.    0.04  0.38  0.52  0.06  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 ($.4 * .8$) and the 10% chance that we undershot ($.6 * .1$). Overshooting plays no role here because if we overshot both .4 and .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 probably feel dismay. In the example above we started with probabilitys of .4 and .6 in two fields; after performing the update the probabilities are not only lowered, but they are strewn out across the map.

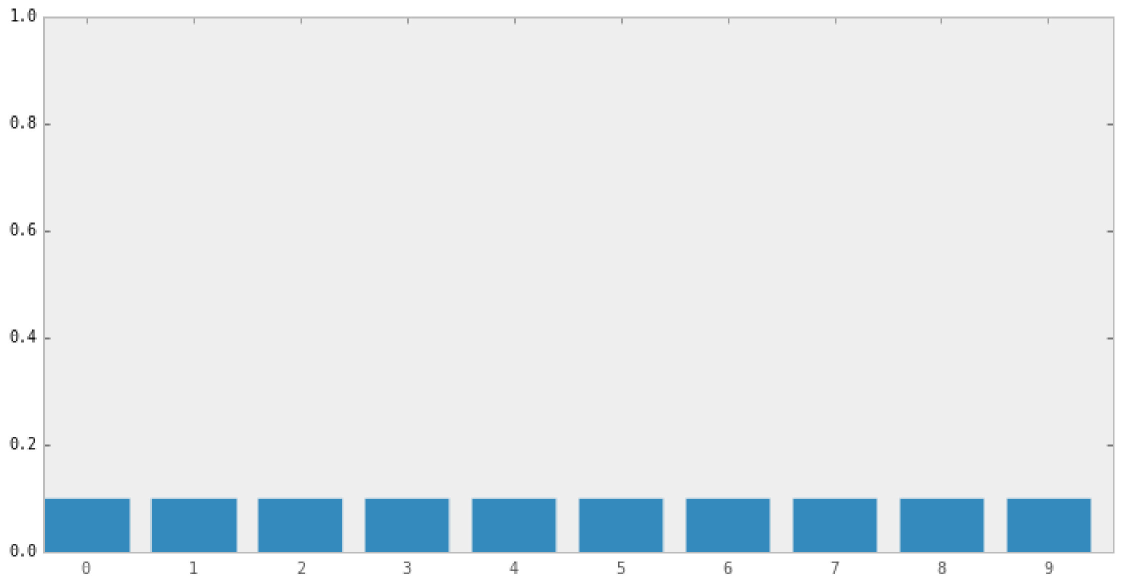
```
In [12]: bar_plot.plot (res)
```



This is not a coincidence, or the result of a carefully chosen example - it is always true of the update step. This is inevitable; if our sensor is noisy we will lose a bit of information on every update. Suppose we were to perform the update 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 pos array. Let's try this with say 500 iterations.

```
In [13]: pos = [1.0,0,0,0,0,0,0,0,0,0]
         for i in range (500):
             pos = update(pos, 1, .8, .1, .1)
         print(pos)
         bar_plot.plot(pos)
```

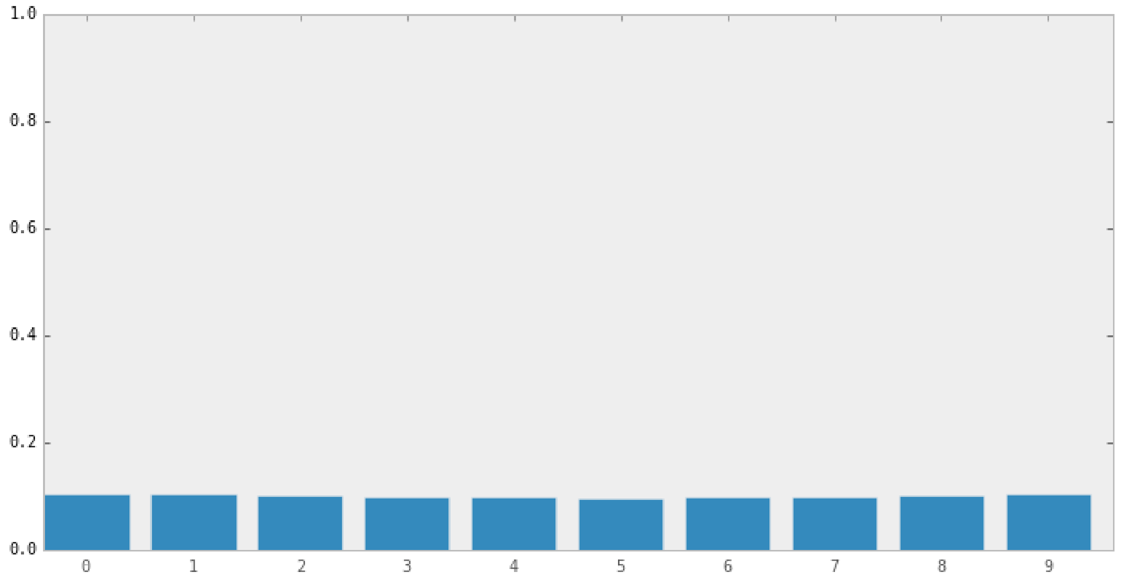
```
[ 0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1  0.1]
```



After 500 iterations we have lost all information, even though we were 100% sure that we started in position 1. Feel free to play with the numbers to see the effect of different number of updates. For example, after 100 updates we have a small amount of information left.

```
In [14]: pos = [1.0,0,0,0,0,0,0,0,0,0]
         for i in range (100):
             pos = update(pos, 1, .8, .1, .1)
         print(pos)
         bar_plot.plot(pos)
```

```
[ 0.10407069  0.10329322  0.10125784  0.09874205  0.09670682  0.09592945
  0.09670682  0.09874205  0.10125784  0.10329322]
```



## 3.6 Integrating Measurements and Movement Updates

The problem of loosing information during an update may make it seem as if our system would quickly devolve into no knowledge. However, recall that our process is not an endless series of updates, but of *measure->update->measure->update->measure->update...* The output of the measure step is fed into the update. The update step, with a degraded certainty, is then fed into the measure step.

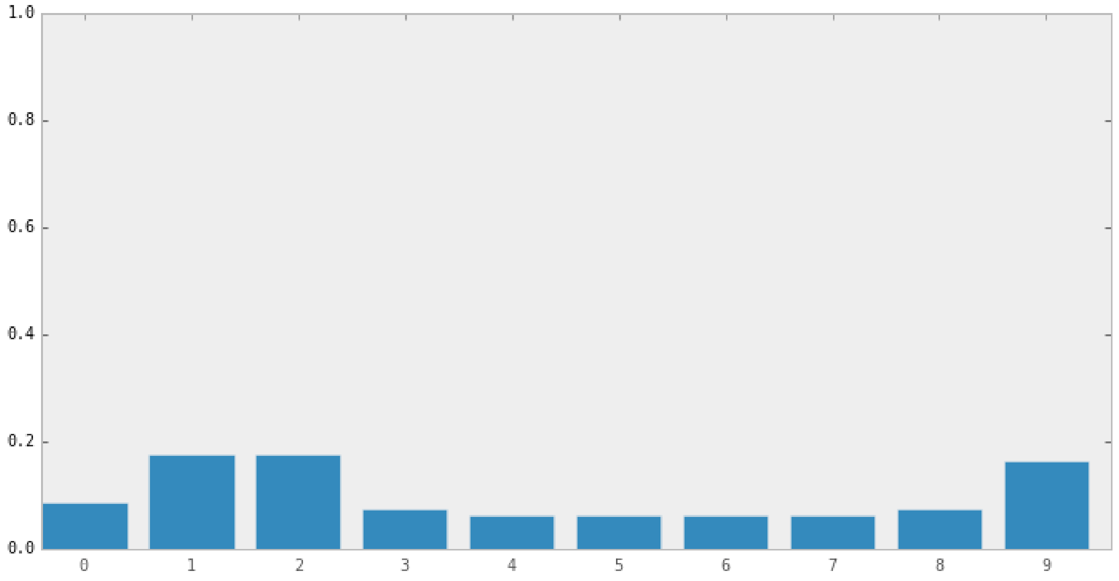
Let's think about this intuitively. After the first measure->update round we have degraded the knowledge we gained by the measurement by a small amount. But now we take another measurement. When we try to incorporate that new measurement into our belief, do we become more certain, less certain, or equally certain. Consider a simple case - you are sitting in your office. A co-worker asks another co-worker where you are, and they report "in his office". You keep sitting there while they ask and answer "has he moved"? "No" "Where is he" "In his office". Eventually you get up and move, and lets say the person didn't see you move. At that time the questions will go "Has he moved" "no" (but you have!) "Where is he" "In the kitchen". Wow! At that moment the statement that you haven't moved conflicts strongly with the next measurement that you are in the kitchen. If we were modelling these with probabilities the probability that you are in your office would lowever, and the probability that you are in the kitchen would go up a little bit. But now imagine the subsequent conversation: "has he moved" "no" "where is he" "in the kitchen". Pretty quickly the belief that you are in your office would fade away, and the belief that you are in the kitchen would increase to near certainty. The belief that you are in the office will never go to zero, nor will the belief that you are in the kitchen ever go to 1.0 because of the chances of error, but in practice your co-workers would be correct to be quite confident in their system.

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

Well, we have already programmed the measure step, and we have programmed the update step. All we need to do is feed the result of one into the other, and we will have programmed our dog tracker!!! Let's see how it performs. We will input data as if the dog started at position 0 and moved right at each update. However, as in a real world application, we will start with no knowledge and assign equal probability to all positions.

```
In [15]: p = np.array([.1]*10)
         p = sense(p, 1, .6, .2)
         print(p)
         p = update(p, 1, .8, .1, .1)
         print(p)
         bar_plot.plot(p)
```

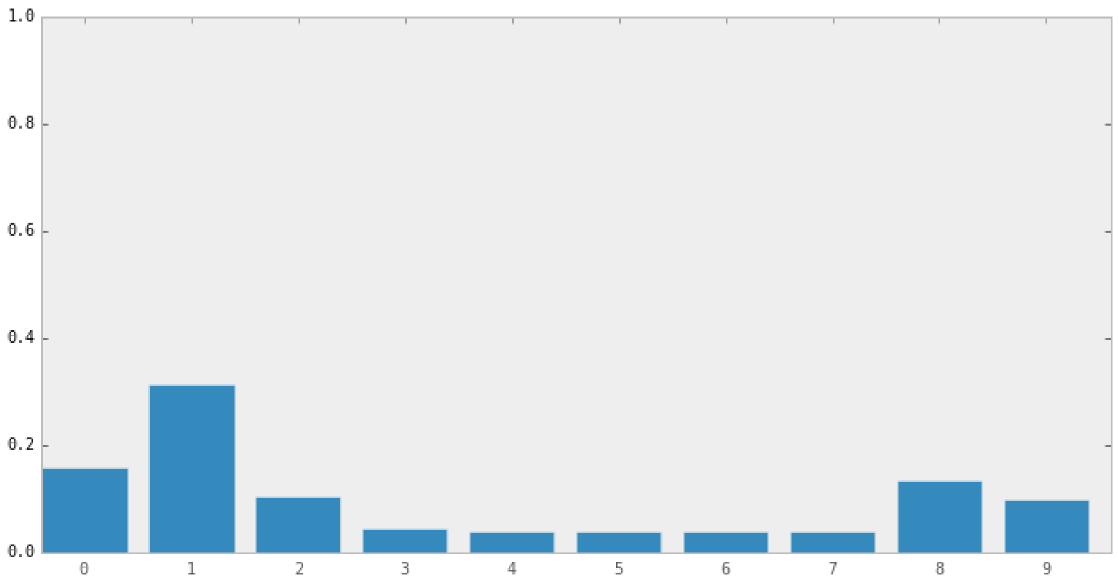
```
[ 0.1875  0.1875  0.0625  0.0625  0.0625  0.0625  0.0625  0.0625  0.1875
  0.0625]
[ 0.0875  0.175   0.175   0.075   0.0625  0.0625  0.0625  0.0625  0.075
  0.1625]
```



So after the first sense we have assigned a high probability to each door position, and a low probability to each wall position. The update step shifted these probabilities to the right, smearing them about a bit. Now lets look at what happens at the next sense.

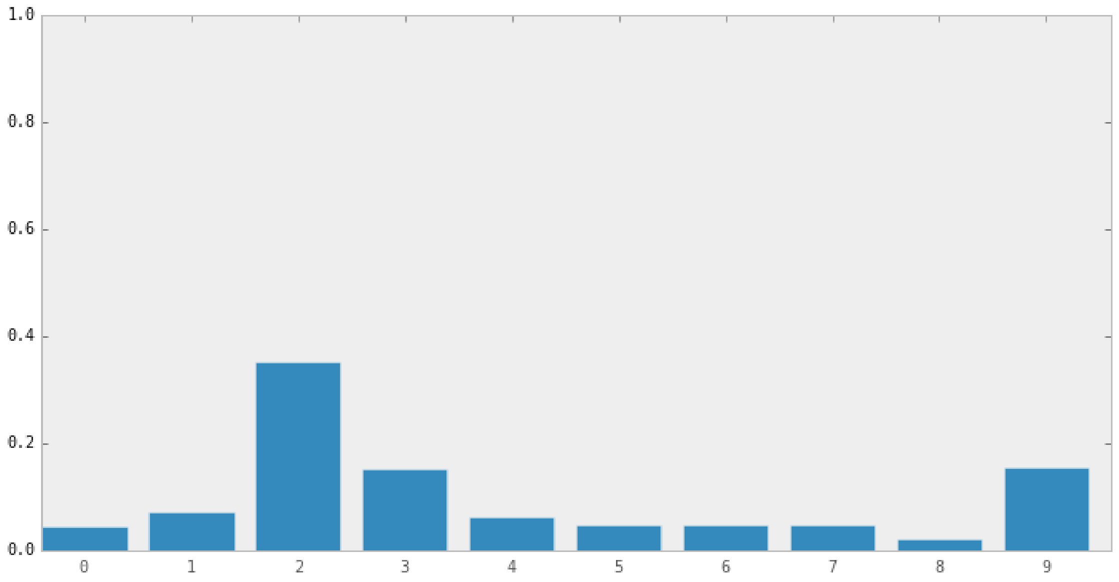
```
In [16]: p = sense(p, 1, .6, .2)
         print(p)
         bar_plot.plot(p)
```

```
[ 0.15671642  0.31343284  0.10447761  0.04477612  0.03731343  0.03731343
  0.03731343  0.03731343  0.13432836  0.09701493]
```



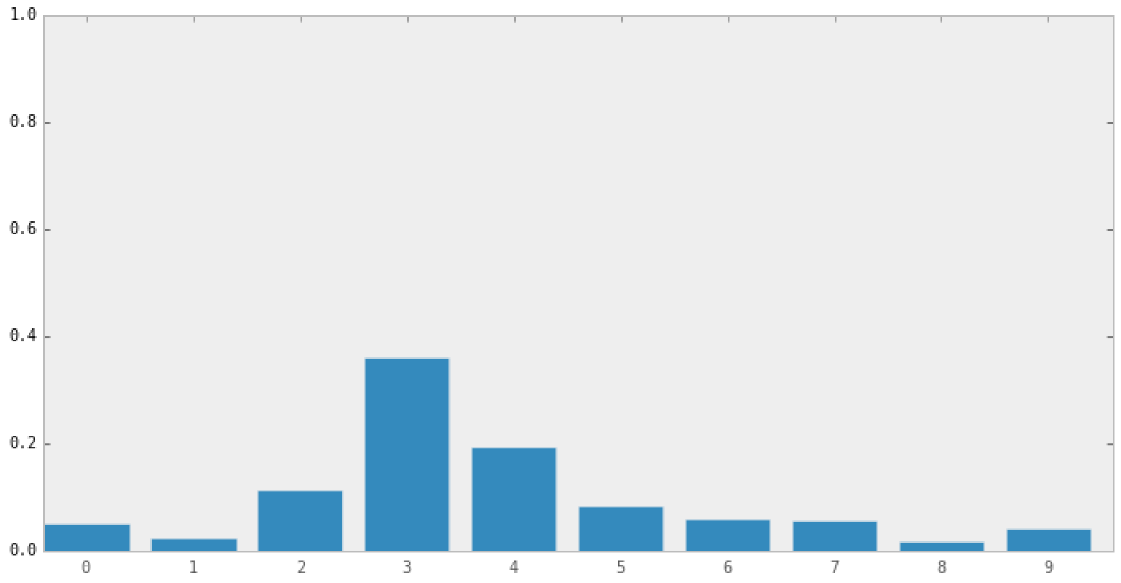
Notice the tall bar at position 1. This corresponds with the (correct) case of starting at position 0, sensing a door, shifting 1 to the right, and sensing another door. No other positions make this set of observations as likely. Now lets add an update and then sense the wall.

```
In [17]: p = update(p, 1, .8, .1, .1)
         p = sense(p, 0, .6, .2)
         bar_plot.plot(p)
```



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 sense->update cycle.

```
In [18]: p = update(p, 1, .8, .1, .1)
         p = sense(p, 0, .6, .2)
         bar_plot.plot(p)
```



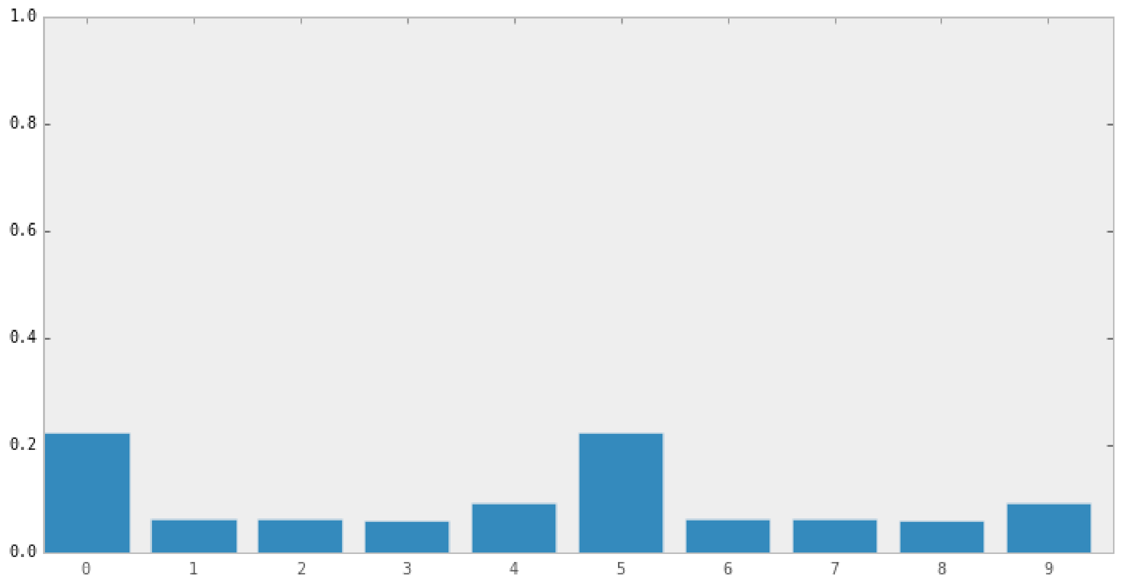
Here things have degraded a bit due to the long string of wall positions in the map. We cannot be as sure where we are when there is an undifferentiated line of wall positions, so naturally our probabilities spread out a bit.

## 3.7 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:

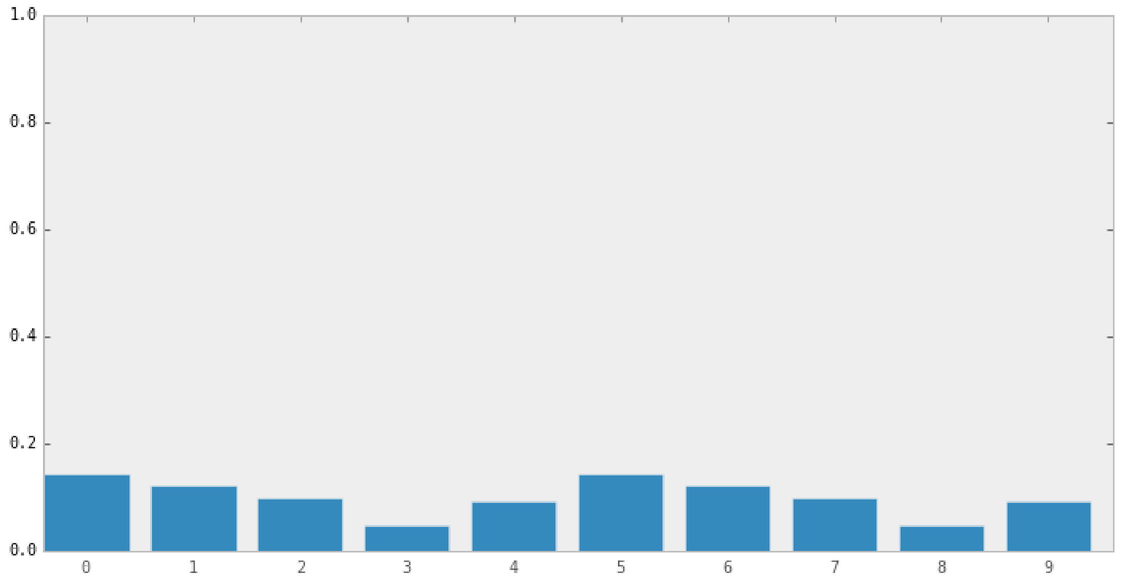
```
In [19]: hallway = [1,0,1,0,0,1,0,1,0,0]
         pos = np.array([.1]*10)
         measurements = [1,0,1,0,0]

         for m in measurements:
             pos = sense(pos, m, .6, .2)
             pos = update(pos, 1, .8, .1, .1)
         bar_plot.plot(pos)
         print(pos)
```



```
[ 0.2245871   0.06288015  0.06109133  0.0581008   0.09334062  0.2245871
  0.06288015  0.06109133  0.0581008   0.09334062]
```

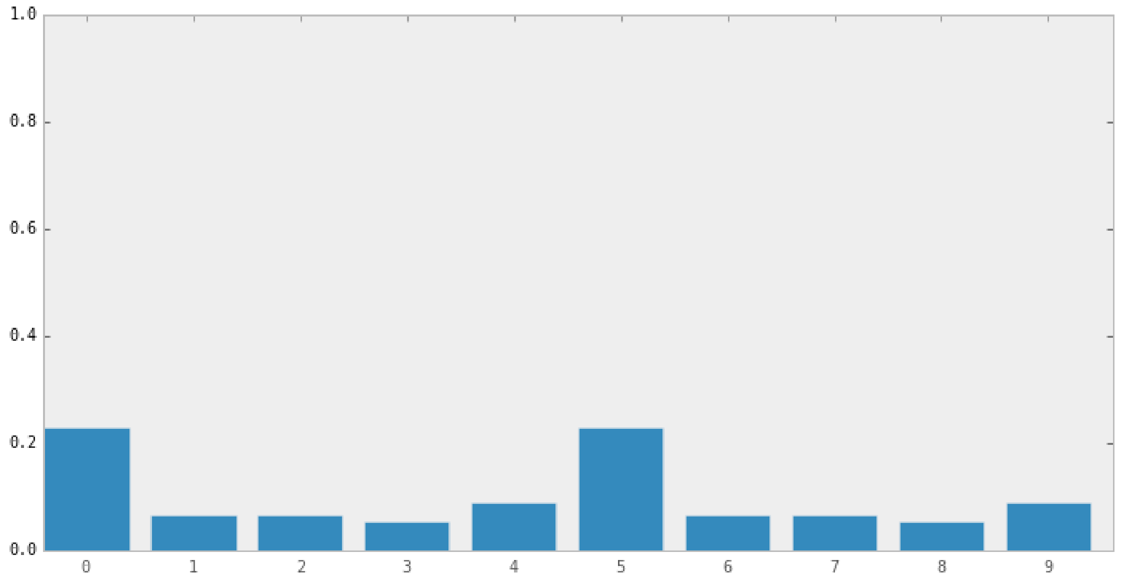
At this point we have correctly identified the likely cases, we either started at position 0 or 5, because we saw the following sequence of doors and walls 1,0,1,0,0. But now lets inject a bad measurement, and see what happens:

```
In [20]: pos = sense(pos, m, .6, .2)
         pos = update(pos, 1, .8, .1, .1)
         bar_plot.plot(pos)
```



That one bad measurment appears to have significantly eroded our knowledge. However, note that our highest probabilities are still at 0 and 5, which is correct. Now let's continue with a series of correct measurements

```
In [21]: measurements = [0,1,0,1,0,0]

         for m in measurements:
             pos = sense(pos, m, .6, .2)
             pos = update(pos, 1, .8, .1, .1)
         bar_plot.plot(pos)
```



As you can see we quickly filtered out the bad sensor reading and converged on the most likely positions for our dog.

## 3.8 Drawbacks and Limitations

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

With that said, while this filter is used in industry, 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 dogs $(x, y)$ coordinate, and probably his velocity $(\dot{x}, \dot{y})$ as well. We have not covered the multidimensional case, but instead of a histogram we use a multidimensional grid to store the probabilities at each discrete location. Each *sense()* and *update()* 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 have 10 or more variables to track, leading to exhorbinant computation requirements.

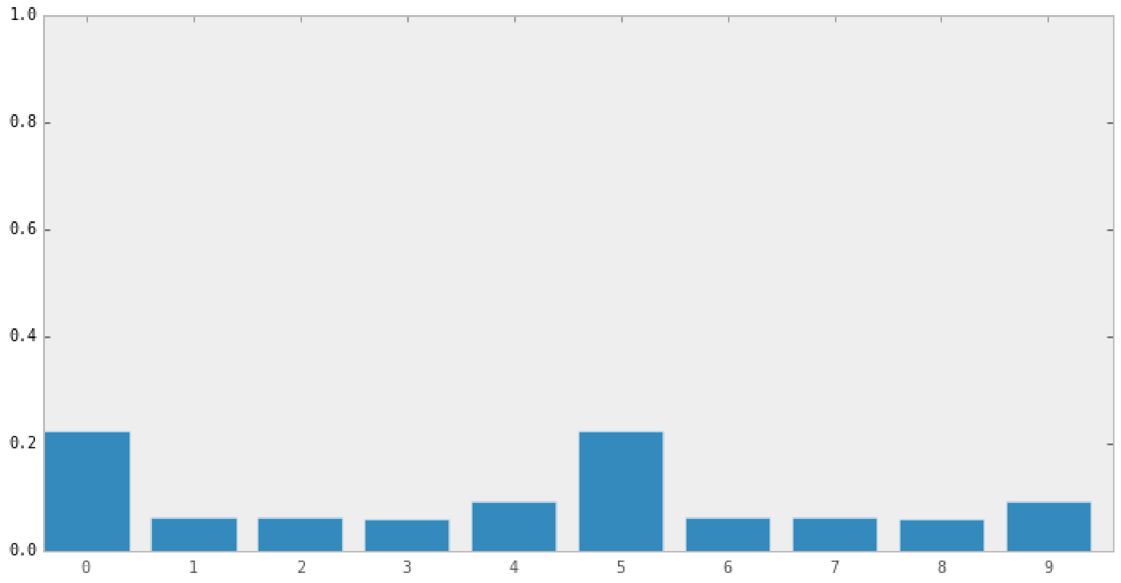
The second problem is that the histogram is discrete, but we live in a continuous world. The histogram requires that you model the output of your filter as a set of discrete points. In our dog in the hallway example, we used 10 positions, which is obviously far too few positions for anything but a toy problem. For example, for a 100 meter hallway you would need 10,000 positions to model the hallway to 1cm accuracy.

So each sense and update operation would entail performing calculations for 10,000 different probabilities. It gets exponentially worse as we add dimensions. If our dog was roaming in a $100x100m^2$ courtyard, we would need 100,000,000 bins ($10,000^2$) to get 1cm accuracy.

A third problem is that the histogram is multimodal. This is not always a problem - an entire class of filters, the particle filters, 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, but 30% sure you are on Willow Avenue. I doubt that you would find that useful. Also, GPSs report their error - they might report that you are at $(109.878W, 38.326N)$ with an error of $9m$. There is no clear mathematical way to extract error information from a histogram. Heuristics suggest themselves to be sure, but there is no exact determination. You may or may not care about that while driving, but you surely do care if you are trying to send a rocket to Mars or track and hit an oncoming missle.

This difficulty is related to the fact that the filter often does not represent what is physically occuring in the world. Consider this distribution for our dog:

```
In [22]: p = [0.2245871, 0.06288015, 0.06109133, 0.0581008, 0.09334062, 0.2245871,
              0.06288015, 0.06109133, 0.0581008,  0.09334062]
         bar_plot.plot(p)
```



The largest probabilities are in position 0 and position 5. This does not fit our physical intuition at all. A dog cannot be in two places at once (my dog Simon certainly tries - his food bowl and my lap often have equal allure to him). We would have to use heuristics to decide how to interpret this distribution, and there is usually no satisfactory answer. This is not always a weakness - a considerable amount of literature has been written on *Multi-Hypothesis Tracking (MHT)*. We cannot always distill our knowledge to one conclusion, and MHT uses various techniques to maintain multiple story lines at once, using backtracking schemes to go *back in time* to correct hypothesis once more information is known. This will be the subject of later chapters. In other cases we truly have a multimodal situation - we may be optically tracking pedistrians on the street and need to represent all of their positions.

In practice it is the exponential increase in computation time that leads to this filter being the least frequently used of all filters in this book. Many problems are best formulated as discrete or multimodal, but we have other filter choices with better performance. 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 in my book.

## 3.9 Generalizing to Multiple Dimensions

## 3.10 Summary

The code is very small, but the result is huge! We will go into the math more later, but we have implemented a form of a Bayesian filter. It is commonly called a Histogram filter. The Kalman filter is also a Bayesian filter, and uses this same logic to produce it's results. The math is a bit more complicated, but not by much. For now, we will just explain that Bayesian statistics compute the liklihood of the present based on the past. If we know there are two doors in a row, and the sensor reported two doors in a row, it is likely that we are positioned near those doors. Bayesian statistics just formalizes that example, and Bayesian filters formalize filtering data based on that math by implementing the sense->update->sense->update process.

We have learned how to start with no information and derive information from noisy sensors. Even though our sensors are very noisey (most sensors are more then 80% accurate, for example) we quickly converge on the most likely position for our dog. We have learned how the update step always degrades our knowledge, but the addition of another measurement, even when it might have noise in it, improves our knowlege, allowing us to converge on the most likely result.

If you followed the math carefully you will realize that all of this math is exact. The bar charts that we are displaying are not an *estimate* or *guess* - they are mathematically exact results that exactly represent our knowledge. The knowledge is probabilistic, to be sure, but it is exact, and correct.

However, we are a long way from tracking an airplane or a car. This code only handles the 1 dimensional case, whereas cars and planes operate in 2 or 3 dimensions. Also, our position vector is *multimodal.* It expresses multiple beliefs at once. Imagine if your GPS told you "it's 20% likely that you are here, but 10% likely that you are on this other road, and 5% likely that you are at one of 14 other locations. That would not be very useful information. Also, the data is discrete. We split an area into 10 (or whatever) different locations, whereas in most real world applications we want to work with continuous data. We want to be able to represent moving 1 km, 1 meter, 1 mm, or any arbitrary amount, such as 2.347 cm.

Finally, the bar charts may strike you as being a bit less certain than we would want. A 25% certaintly may not give you a lot of confidence in the anwser. Of course, what is important here is the ratio of this probability to the other probabilities in your vector. If the next largest bar is 23% then we are not very knowledgable about our position, whereas if the next largest is 3% we are in fact quite certain. But this is not clear or intuitive. However, there is an extremely important insight that Kalman filters implement that will signficantly improve our accuracy from the same data.

**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 intuitively understand why this histogram filter works, and can at least work through the math, 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 you understand everything important you need to grasp the Kalman filter.

**Author notes:** Do I want to go to the multidimensional case? At least describe it, but why not implement it as well

# Chapter 4

# Gaussian Probabilities

```
In [1]: #format the book
        %matplotlib inline
        from __future__ import division, print_function
        import matplotlib.pyplot as plt
        import book_format
        book_format.load_style()

Out[1]: <IPython.core.display.HTML at 0x1080350>
```

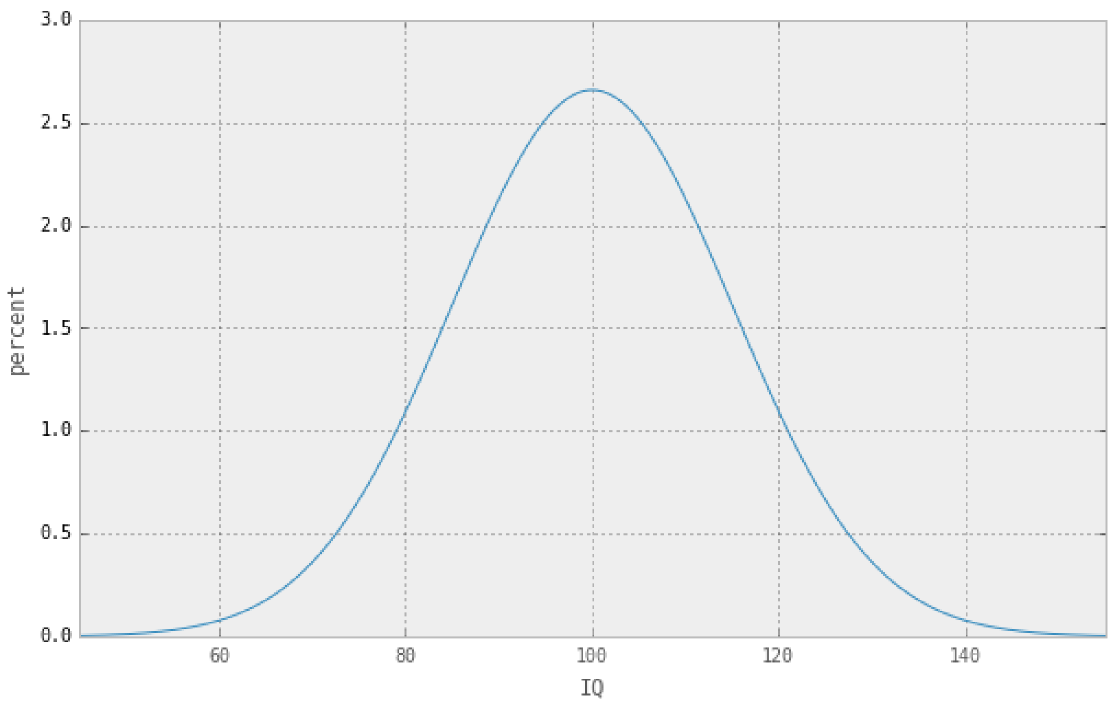
## 4.1 Introduction

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 (12.34381, -95.54321,2389.5) where that is latitude, longitude, and altidue. We do not want our filter to tell us "it might be at (1,65,78) or at (34,656,98)" That doesn't match our physical intuition of how the world works, and as we discussed, it is prohibitively expensive to compute.

> So we desire a unimodal, continuous way to represent probabilities that models how the real world works, and that is very computationally efficient to calculate. As you might guess from the chapter name, Gaussian distributions provide all of these features.

Before we go into the math, lets just look at a graph of the Gaussian distribution to get a sense of what we are talking about.

```
In [3]: #ignore this code.
        %matplotlib inline
        from __future__ import division, print_function
        from gaussian_internal import plot_gaussian
        import matplotlib.pylab as pylab
        pylab.rcParams['figure.figsize'] = 10,6

        plot_gaussian(mu=100, variance=15*15, xlim=(45,155), xlabel='IQ', ylabel='percent')
```



Probably this is immediately recognizable to you as a 'bell curve'. This curve is ubiquitious because under real world conditions most observations are distributed in such a manner. In fact, this is the bell curve for IQ (Intelligence Quotient). You've probably seen this before, and understand it. It tells us that the average IQ is 100, and that the number of people that have IQs higher or lower than that drops off as they get further away from 100. It's hard to see the exact number, but we can see that very few people have an IQ over 150 or under 50, but a lot have an IQ of 90 or 110.

This curve is not unique to IQ distributions - 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 probabilitiy, it is continuous, and it is computationally efficient. We will soon discover that it also other desirable qualities that we do not yet recognize we need.

## 4.2 Nomenclature

A bit of nomenclature before we continue - this chart depicts the probability of of a *random variable* having any value between $(-\infty..\infty)$. For example, for this chart the probability of the variable being 100 is roughly 2.7%, whereas the probability of it being 80 is around 1%. > *Random variable* will be precisely defined later. For now just think of it as a variable that can 'freely' and 'randomly' vary. A dog's position in a hallway, air temperature, and a drone's height above the ground are all random variables. The position of the North Pole is not, nor is a sin wave (a sin wave is anything but 'free').

You may object that human IQs cannot be less than zero, let alone $-\infty$. This is true, but this is a common limitation of mathematical modelling. "The map is not the territory" is a common expression, and it is true for Bayesian filtering and statistics. The Gaussian distribution above very closely models the distribution of IQ test results, but being a model it is necessarily imperfect. The difference between model and reality will come up again and again in these filters.

You will see these distributions called *Gaussian distributions*, *normal distributions*, and *bell curves*. Bell curve is ambiguous because there are other distributions which also look bell shaped but are not Gaussian distributions, so we will not use it further in this book. But *Gaussian* and *normal* both mean the same thing, and are used interchangeably. I will use both throughout this book as different sources will use either term, and so I want you to be used to seeing both. Finally, as in this paragraph, it is typical to shorten the name and just talk about a *Gaussian* or *normal* - these are both typical shortcut names for the *Gaussian distribution*.

## 4.3 Gaussian Distributions

So let us 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:

$$f(x, \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(x-\mu)^2/\sigma^2}$$

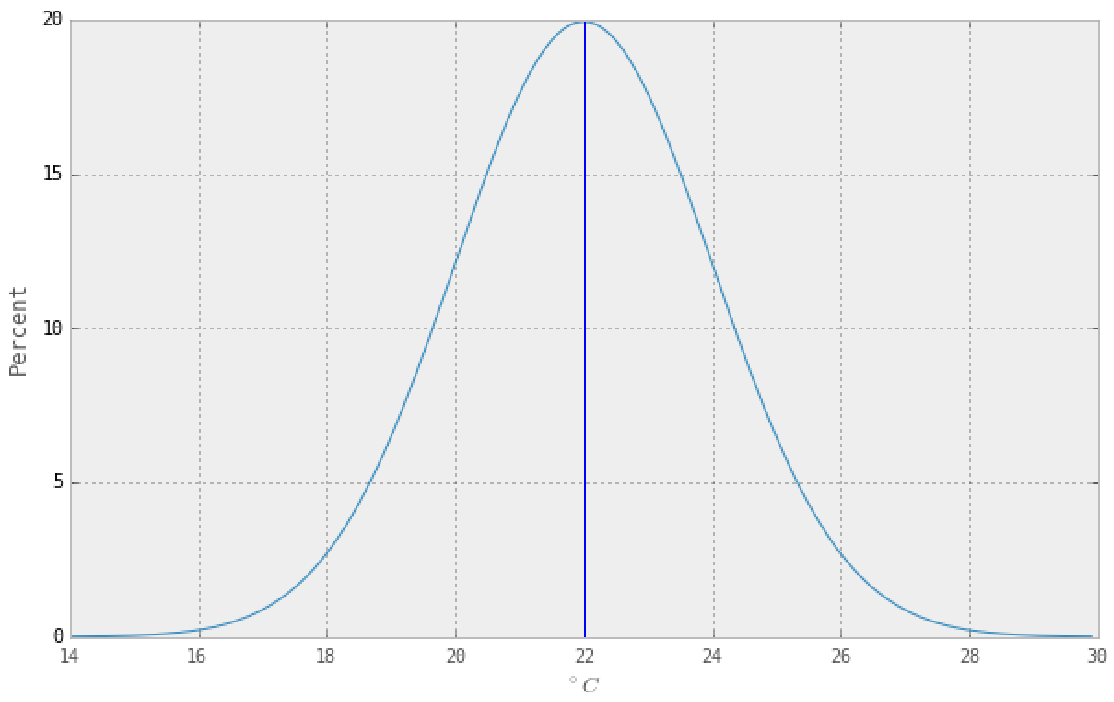
Don't be dissuaded by the equation if you haven't seen it before; you will not need to memorize or manipulate it. The computation of this function is stored in stats.py.

> **Optional:** Let's remind ourselves how to look at a function stored in a file by using the *%load* magic. If you type *%load -s gaussian stats.py* into a code cell and then press CTRL-Enter, the notebook will create a new input cell and load the function into it.

```
%load -s gaussian stats.py

def gaussian(x, mean, var):
    """returns normal distribution for x given a
    gaussian with the specified mean and variance.
    """
    return math.exp((-0.5*(x-mean)**2)/var) / \
                     math.sqrt(_two_pi*var)
```

We will plot a Gaussian with a mean of 22 ($\mu = 22$), with a variance of 4 ($\sigma^2 = 4$), and then discuss what this means.

```
In [4]: from stats import gaussian
        plot_gaussian(22,4,True,xlabel='$^{\circ}C$',ylabel="Percent")

        print('Probability of 22 is %.2f' % (gaussian(22,22,4)*100))
        print('Probability of 24 is %.2f' % (gaussian(24,22,4)*100))
```



```
Probability of 22 is 19.95
Probability of 24 is 12.10
```

So what does this curve *mean*? Assume for a moment that we have a themometer, which reads $22\,°C$. No thermometer is perfectly accurate, and so we normally expect that thermometer will read $\pm$ that temperature by some amount each time we read it. Furthermore, a theorem called **Central Limit Theorem** states that if we make many measurements that the measurements will be normally distributed. If that is true, then this chart can be interpreted as a continuous curve depicting our belief that the temperature is any given temperature. In this curve, we assign a probability of the temperature being exactly $22°C$ is 19.95%. Looking to the right, we assign the probability that the temperature is $24°C$ is 12.10%. Because of the curve's symmetry, the probability of 20°C is also 12.10%.

So 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.

> *Important*: I will repeat what I wrote at the top of this section: "A Gaussian. . . is completely described with two parameters"

The standard notation for a normal distribution for a random variable $X$ is $X \sim \mathcal{N}(\mu, \sigma^2)$. This means I can express the temperature reading of our thermometer as

$$temp = \mathcal{N}(22, 4)$$

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 probability of the temperature being $22\,°C$, $20\,°C$, $87.34\,°C$, or any other arbitrary value.

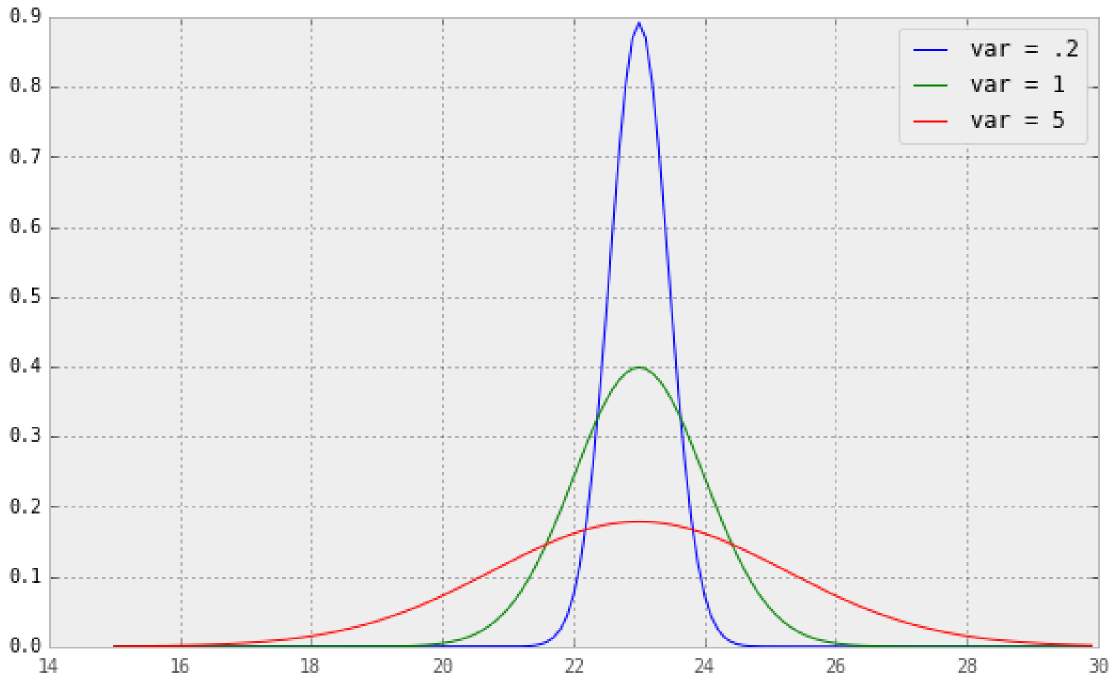
The Variance

Since this is a probability 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 occurances, which must sum to one.

This leads to an important insight. If the variance is small the curve will be narrow. 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:

```
In [5]: import numpy as np
        import matplotlib.pyplot as plt

        xs = np.arange(15,30,0.1)
        p1, = plt.plot (xs,[gaussian(x, 23, .2) for x in xs],'b')
        p2, = plt.plot (xs,[gaussian(x, 23, 1) for x in xs],'g')
        p3, = plt.plot (xs,[gaussian(x, 23, 5) for x in xs],'r')
        plt.legend([p1,p2,p3], ['var = .2', 'var = 1', 'var = 5'])
        plt.show()
```



So what is this telling us? The blue gaussian is very narrow. It is saying that we believe $x = 23$, and that we are very sure about that (90$aremuchlesssureaboutthat$(18and 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. The blue gaussian has almost completely eliminated 22 or 24 as possible value - their probability is almost 0%, whereas the red curve considers them nearly as likely as 23.

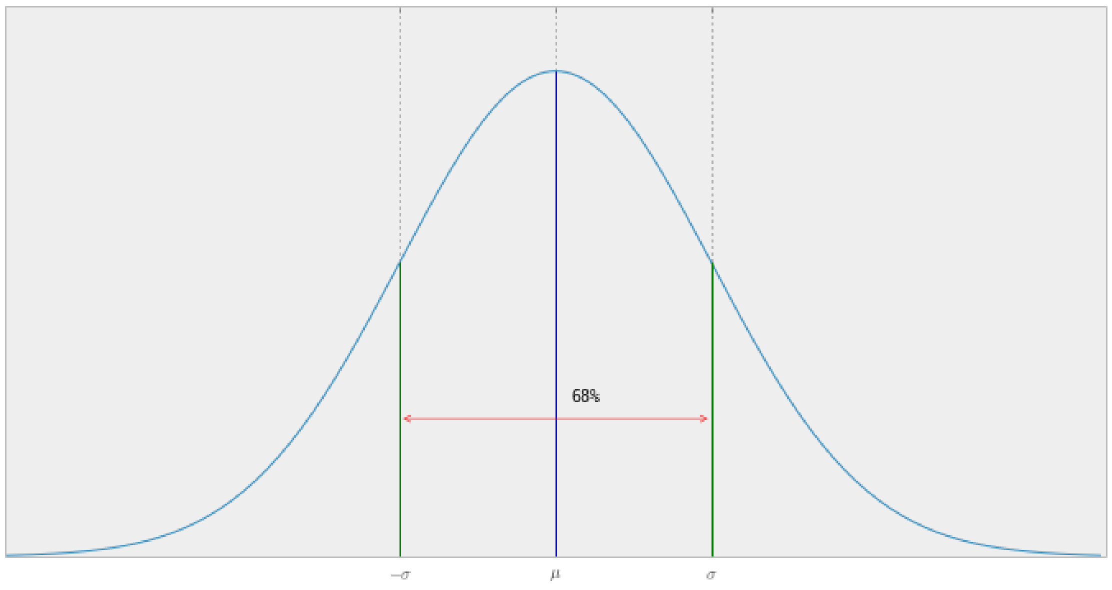
If we think back to the thermometer, we can consider these three curves as representing the readings from three different thermometers. The blue curve represents a very accurate thermometer, and the red one represents a fairly inaccurate one. Green of course represents one in between the two others. 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.

The standard notation for a normal distribution for a random variable $X$ is just $X \sim \mathcal{N}(\mu, \sigma^2)$ where $\mu$ is the mean and $\sigma^2$ is the variance. It may seem odd to use $\sigma$ squared - why not just $\sigma$? We will not go into great detail about the math at this point, but in statistics $\sigma$ is the *standard deviation* of a normal distribution. *Variance* is defined as the square of the standard deviation, hence $\sigma^2$.

It is worth spending a few words on standard deviation now. The standard deviation is a measure of how much variation from the mean exists. For Gaussian distributions, 68% of all the data falls within one standard deviation($1\sigma$) of the mean, 95% falls within two standard deviations ($2\sigma$), and 99.7% within three ($3\sigma$). This is often called the 68-95-99.7 rule. So 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 $71 \pm (2 * 9.4)$).

The following graph depicts the relationship between the standard deviation and the normal distribution.

```
In [6]: from gaussian_internal import display_stddev_plot
        display_stddev_plot()
```

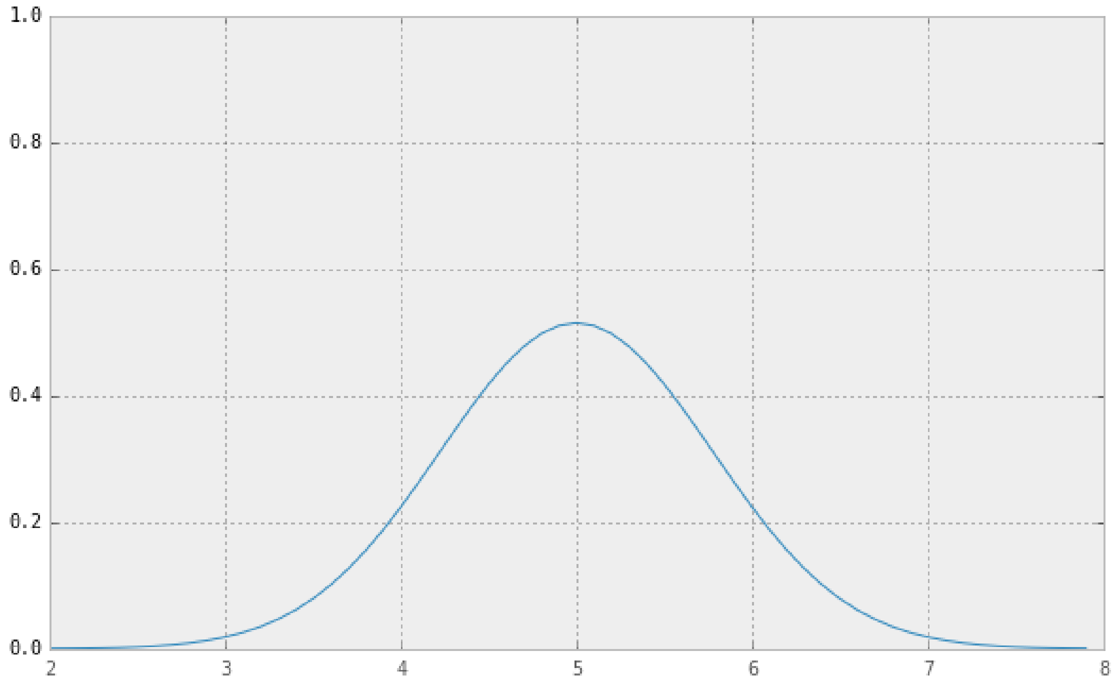


**Sidebar**: An equivalent formation for a Gaussian is $\mathcal{N}(\mu, 1/\tau)$ where $\mu$ is the *mean* and *tau* the *precision*. Here $1/\tau = \sigma^2$; it is the reciprocal of the variance. While we do not use this formulation in this book, it underscores that the variance is a measure of how 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 equivelant forms. 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.

## 4.4 Interactive Gaussians

For those that are reading this in IPython 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 [7]: import math
        from IPython.html.widgets import interact, interactive, fixed
        import IPython.html.widgets as widgets

        def plt_g (mu,variance):
            xs = np.arange(2,8,0.1)
            ys = [gaussian (x, mu,variance) for x in xs]
            plt.plot (xs, ys)
            plt.ylim((0,1))
            plt.show()

        interact (plt_g, mu=(0,10), variance=widgets.FloatSliderWidget(value=0.6,min=0.2,max=4.5))
```



```
Out[7]: <function __main__.plt_g>
```

## 4.5 Computational Properties of the Gaussian

Recall how our discrete Bayesian filter worked. We had a vector implemented as a numpy array representing our belief at a certain moment in time. When we performed another measurement using the *sense()* function we had to multiply probabilities together, and when we performed the motion step using the *update()* function we had to shift and add probabilities. I've promised you that the Kalman filter uses essentially the same process, and that it uses Gaussians instead of histograms, so you might reasonable expect that we will be multipling, adding, and shifting Gaussians in the Kalman filter.

A typical textbook would directly launch into a multipage proof of the behavior of Gaussians under these operations, but I don't see the value in that right now. I think the math will be much more intuitive and clear if we just start developing a Kalman filter using Gaussians. I will provide the equations for multiplying and shifting Gaussians at the appropriate time. You will then be able to develop a physical intuition for what these operations do, rather than be forced to digest a lot of fairly abstract math.

The key point, which I will only assert for now, is that all the operations are very simple, and that they preserve the properties of the Gaussian. This is somewhat remarkable, in that the Gaussian is a nonlinear function, and typically if you multiply a nonlinear equation with itself you end up with a different equation. For example, the shape of $sin(x)sin(x)$ is very different from $sin(x)$. But the result of multiplying two Gaussians is yet another Gaussian. This is a fundamental property, and the key reason why Kalman filters are possible.

## 4.6 Summary and Key Points

The following points **must** be understood by you before we continue:

- Normal distributions occur throughout nature
- They 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
- $\sigma^2$ represents how much our measurements vary from the mean

# Chapter 5

# Kalman Filters

```
In [2]: #format the book
        %matplotlib inline
        from __future__ import division, print_function
        import matplotlib.pyplot as plt
        import book_format
        book_format.load_style()

Out[2]: <IPython.core.display.HTML at 0x197d410>
```

## 5.1 One Dimensional Kalman Filters

Now that we understand the histogram filter and Gaussians we are prepared to implement a 1D Kalman filter. We will do this exactly as we did the histogram filter - rather than going into the theory we will just develop the code step by step. But first, let's set the book style.

## 5.2 Tracking A Dog

As in the histogram chapter we will be tracking a dog in a long hallway at work. However, in our latest hackathon someone created an RFID tracker that provides a reasonable accurate position for our dog. Suppose the hallway is 100m long. The sensor returns the distance of the dog from the left end of the hallway. So, 23.4 would mean the dog is 23.4 meters from the left end of the hallway.

Naturally, the sensor is not perfect. A reading of 23.4 could correspond to a real position of 23.7, or 23.0. However, it is very unlikely to correspond to a real position of say 47.6. Testing during the hackathon confirmed this result - the sensor is reasonably accurate, and while it had errors, the errors are small. Futhermore, the errors seemed to be evenly distributed on both sides of the measurement; a true position of 23m would be equally likely to be measured as 22.9 as 23.1.

Implementing and/or robustly modelling an RFID system is beyond the scope of this book, so we will write a very simple model. We will start with a simulation of the dog moving from left to right at a constant speed with some random noise added.

```
In [3]: from __future__ import print_function, division

        import numpy.random as random
        import math

        class DogSensor(object):

            def __init__(self, x0=0, velocity=1, noise=0.0):
                """ x0 - initial position
            velocity - (+=right, -=left)
            noise - scaling factor for noise, 0== no noise
        """
        self.x = x0
        self.velocity = velocity
        self.noise = math.sqrt(noise)

    def sense(self):
        self.x = self.x + self.velocity
        return self.x + random.randn() * self.noise
```

The constructor `__init()__` initializes the DogSensor class with an initial position (x0), velocity (vel), and an noise scaling factor. The `sense()` function has the dog move by the set velocity and returns its new position, with noise added. If you look at the code for `sense()` you will see a call to `numpy.random.randn()`. This returns a number sampled from a normal distribution with a mean of 0.0. Let's look at some example output for that.

```
In [4]: for i in range(20):
            print("%.4f" % random.randn(),end='\t')
```

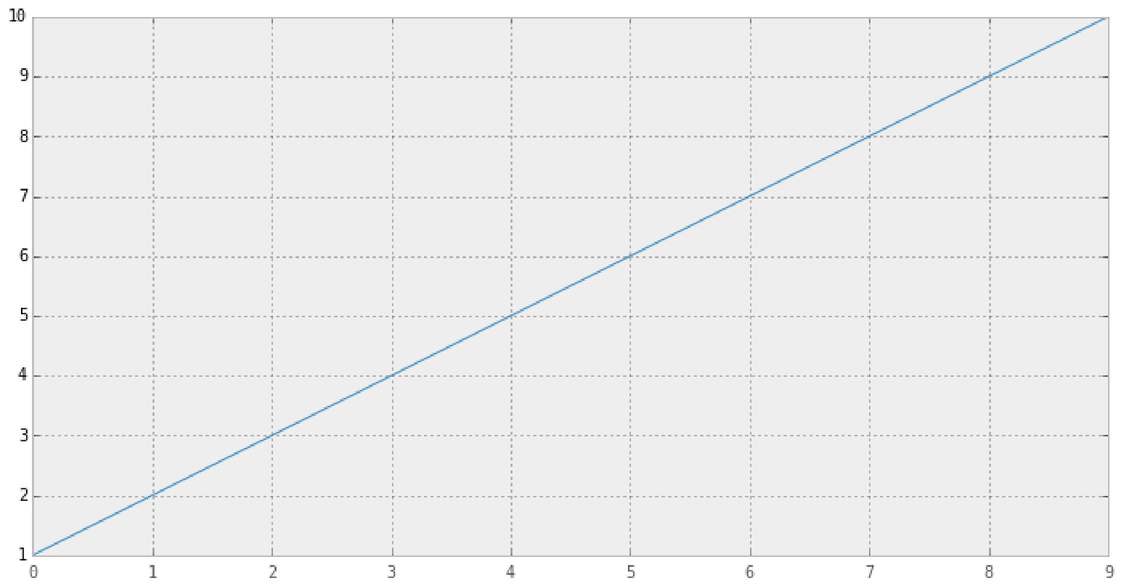
```
-0.8025        0.4440        1.6176        -0.7325        0.0304        0.3592        -0.1916        1.
```

You should see a sequence of numbers near 0, some negative and some positive. Most are probably between -1 and 1, but a few might lie somewhat outside that range. This is what we expect from a normal distribution - values are clustered around the mean, and there are fewer values the further you get from the mean.

Okay, so lets look at the output of the `DogSensor` class. We will start by setting the noise to 0 to check that the class does what we think it does

```
In [5]: import matplotlib.pyplot as plt
        import matplotlib.pylab as pylab

        dog = DogSensor (noise=0.0)
        xs = []
        for i in range(10):
            x = dog.sense()
            xs.append(x)
            print("%.4f" % x, end=' '),
        plt.plot(xs)
        plt.show()
```

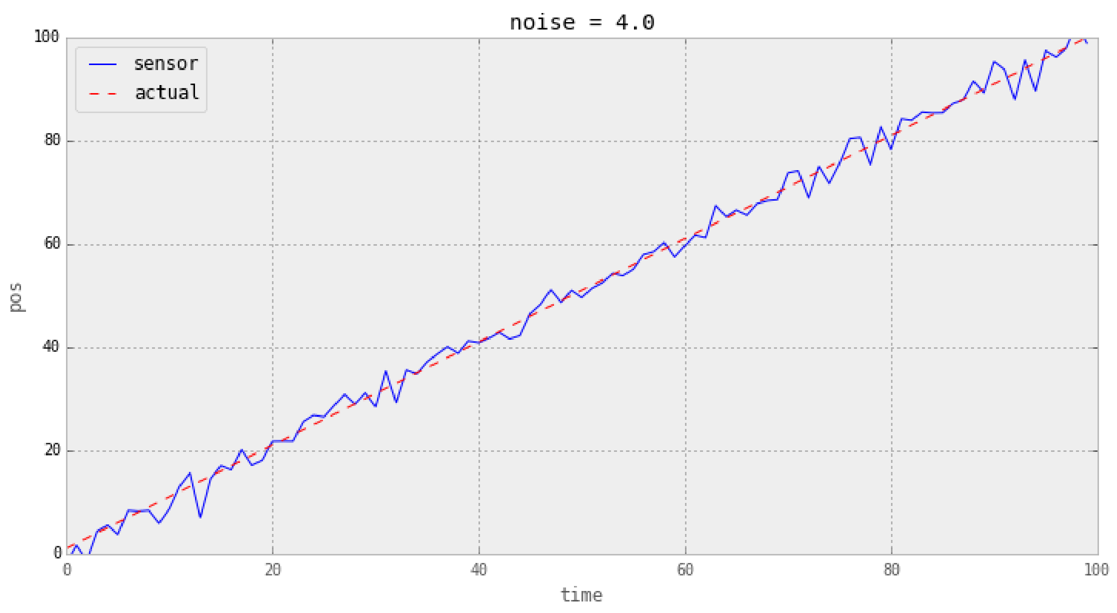
```
1.0000 2.0000 3.0000 4.0000 5.0000 6.0000 7.0000 8.0000 9.0000 10.0000
```



The constructor initialized the dog at position 0 with a velocity of 1 (move 1.0 to the right). So we would expect to see an output of 1..10, and indeed that is what we see. If you thought the correct answer should have been 0..9 recall that *sense()* returns the dog's position *after* updating his position, so the first postion is 0.0 + 1, or 1.0.

Now let's inject some noise in the signal.

```
In [6]: def test_sensor(noise_scale):
            dog = DogSensor(noise=noise_scale)

            xs = []
            for i in range(100):
                x = dog.sense()
                xs.append(x)
            p1, = plt.plot(xs, c='b')
            p2, = plt.plot([0,99],[1,100], 'r--')
            plt.xlabel('time')
            plt.ylabel('pos')
            plt.ylim([0,100])
            plt.title('noise = ' + str(noise_scale))
            plt.legend([p1, p2], ['sensor', 'actual'], loc=2)
            plt.show()

        test_sensor(4.0)
```



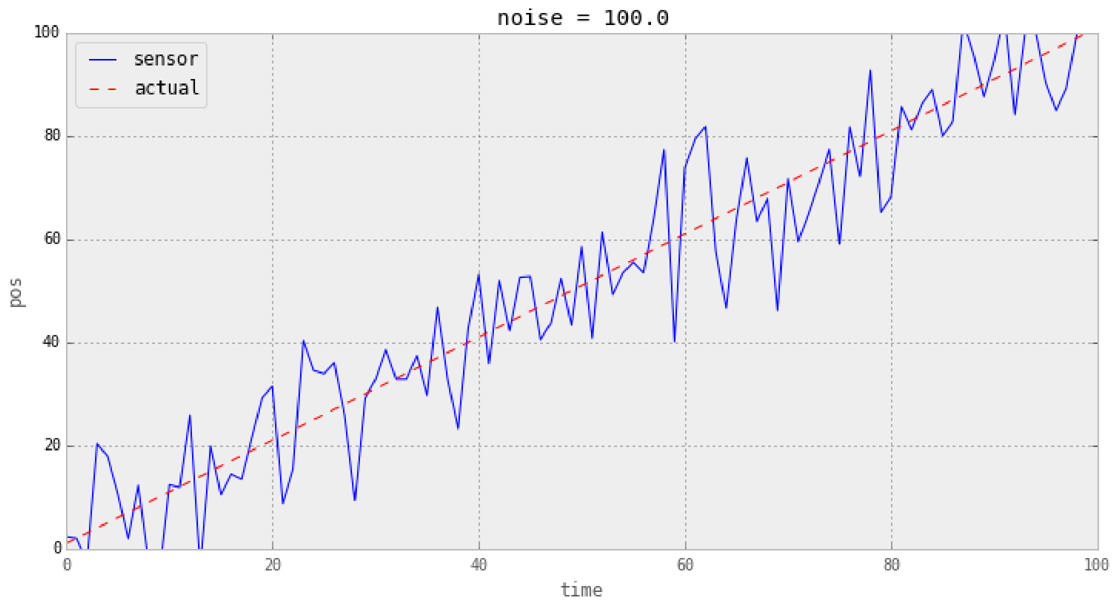
> **Note**: numpy uses a random number generator to generate the normal distribution samples. The numbers I see as I write this are unlikely to be the ones that you see. If you run the cell above multiple times, you should get a slightly different result each time. I could use `numpy.random.seed(some_value)` to force the results to be the same each time. This would simplify my explanations in some cases, but would ruin the interactive nature of this chapter. To get a real feel for how normal distributions and Kalman filters work you will probably want to run cells several times, observing what changes, and what stays roughly the same.

So the output of the sensor should be a wavering blue line drawn over a dotted red line. The dotted red line shows the actual position of the dog, and the blue line is the noise signal produced by the simulated RFID sensor. Please note that the red dotted line was manually plotted - we do not yet have a filter that recovers that information!

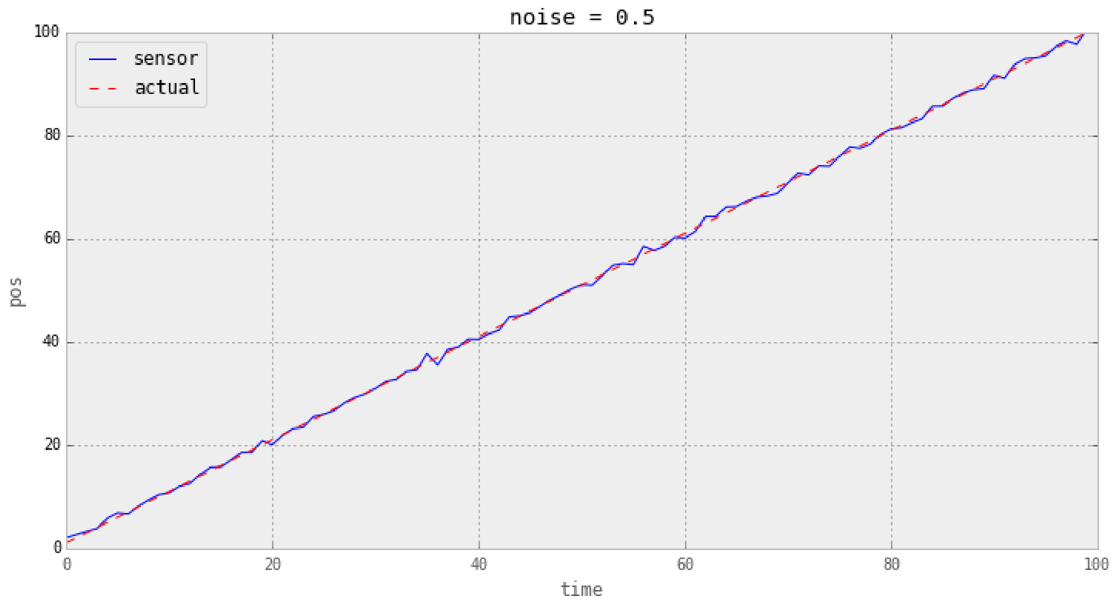
If you are running this in an interactive IPython Notebook, I strongly urge you to run the script several times in a row. You can do this by putting the cursor in the cell containing the Python code and pressing Ctrl+Enter. Each time it runs you should see a different jagged blue line wavering over the top of the dotted red line.

I also urge you to adjust the noise setting to see the result of various values. However, since you may be reading this in a read only notebook, I will show two extreme examples. The first plot shows the noise set to 100.0, and the second shows noise set to 0.5.

```
In [7]: test_sensor(100.0)
```



```
In [8]: test_sensor(0.5)
```

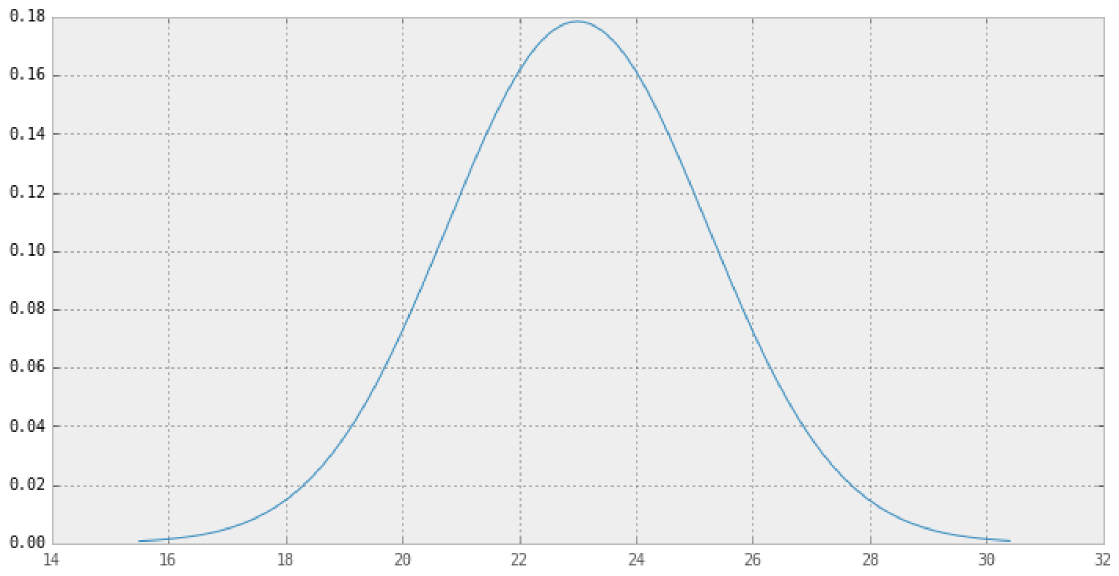


You may not have a full understanding of the exact *meaning* of a noise value of 100.0, but as it turns out if you multiply `randn()` with a number $n$, the result is just a normal distribution with $\sigma = \sqrt{n}$. So the example with noise = 100 is using the normal distribution $\mathcal{N}(0, 100)$. Recall the notation for a normal distribution is $\mathcal{N}(\mu, \sigma^2)$. If the square root is confusing, recall that normal distributions use $\sigma^2$ for the variance, and $\sigma$ is the standard deviation, which we do not use in this book. DogSensor.`__init__()` takes the square root of the noise setting so that the `noise * randn()` call properly computes the normal distribution.

## 5.3 Math with Gaussians

Let's say we believe that our dog is at 23m, and the variance is 5, or $pos_{dog} = \mathcal{N}(23, 5))$. We can represent that in a plot:

```
In [9]: import stats
        stats.norm_plot(23, 5)
```

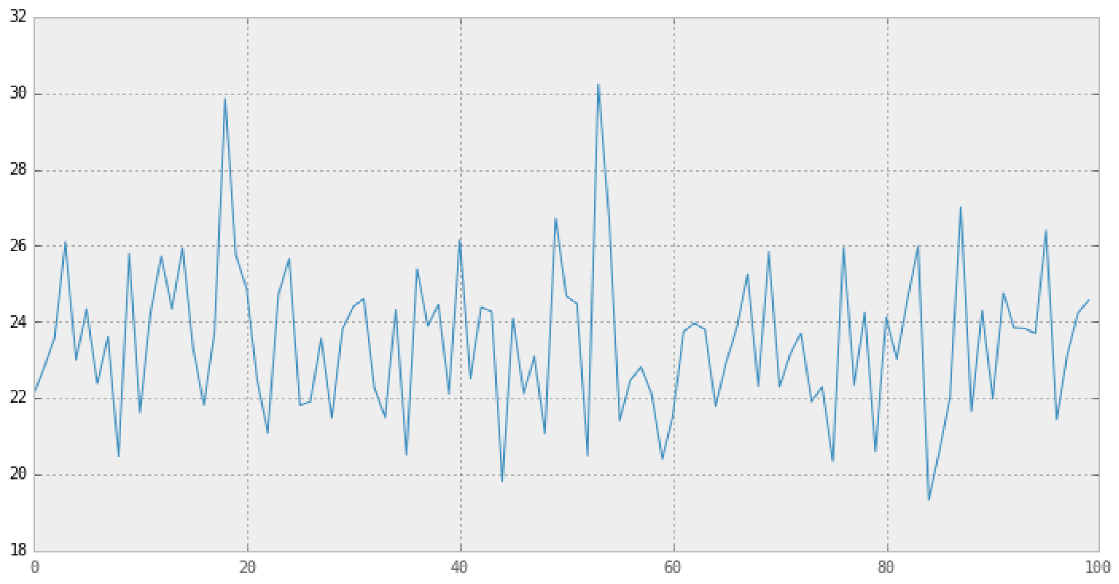


This corresponds to a fairly inexact belief. While we believe that the dog is at 23, note that roughly 21 to 25 are quite likely as well. Let's assume for the moment our dog is standing still, and we query the sensor again. This time it returns 23.2 as the position. Can we use this additional information to improve our estimate of the dog's position?

Intuition suggests 'yes'. Consider: if we read the sensor 100 times and each time it returned a value between 21 and 25, all centered around 23, we should be very confident that the dog is somewhere very near 23. Of course, a different physical interpertation is possible. Perhaps our dog was randomly wandering back and forth in a way that exactly emulated a normal distribution. But that seems extremely unlikely - I certainly have never seen a dog do that. So the only reasonable assumption is that the dog was mostly standing still at 23.0.

Let's look at 100 sensor readings in a plot:

```
In [10]: dog = DogSensor(23, 0, 5)
         xs = range(100)
         ys = []
         for i in xs:
             ys.append(dog.sense())

         plt.plot(xs,ys)
         plt.show()
```



Eyeballing this confirms our intuition - no dog moves like this. However, noisy sensor data certainly looks like this. So let's proceed and try to solve this mathematically. But how?

Recall the histogram code for adding a measurement to a pre-existing belief:

```
def sense(pos, measure, p_hit, p_miss):
    q = array(pos, dtype=float)
    for i in range(len(hallway)):
        if hallway[i] == measure:
            q[i] = pos[i] * p_hit
        else:
            q[i] = pos[i] * p_miss
     normalize(q)
     return q
```

Note that the algorithm is essentially computing:

```
new_belief = old_belief * measurement * sensor_error
```

The measurement term might not be obvious, but recall that measurement in this case was always 1 or 0, and so it was left out for convience.

If we are implementing this with gaussians, we might expect it to be implemented as:

```
new_gaussian = measurement * old_gaussian
```

where measurement is a Gaussian returned from the sensor. But does that make sense? Can we multiply gaussians? If we multiply a Gaussian with a Gaussian is the result another Gaussian, or something else?

It is not particularly difficult to perform the algebra to derive the equation for multiplying two gaussians, but I will just present the result:

$$N(\mu_1, \sigma_1^2) * N(\mu_2, \sigma_2^2) = N(\frac{\sigma_1^2 \mu_2 + \sigma_2^2 \mu_1}{\sigma_1^2 + \sigma_2^2}, \frac{1}{\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}})$$

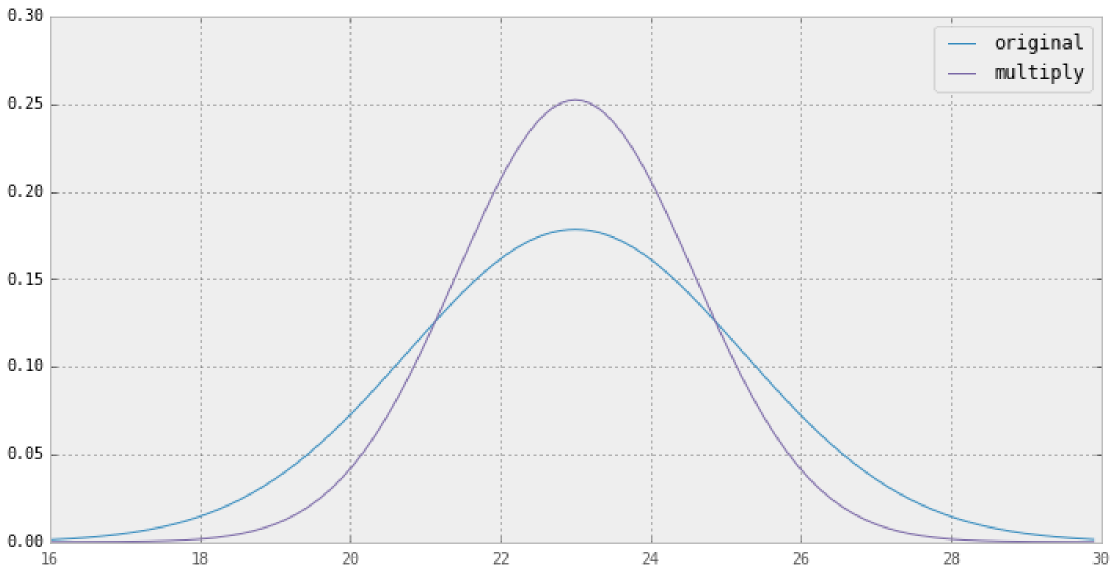
In other words the result is a Gaussian with

$$\mu = \frac{\sigma_1^2\mu_2 + \sigma_2^2\mu_1}{\sigma_1^2 + \sigma_2^2},$$
$$\sigma = \frac{1}{\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}}$$

Without doing a deep analysis we can immediately infer some things. First and most importantly the result of multiplying two Gaussians is another Gaussian. The expression for the mean is not particularly illuminating, except that it is a combination of the means and variances of the input. But the variance of the result is merely some combination of the variances of the variances of the input. We conclude from this that the variances are completely unaffected by the values of the mean!

Let's immediately look at some plots of this. First, let's look at the result of multiplying $N(23, 5)$ to itself. This corresponds to getting 23.0 as the sensor value twice in a row. But before you look at the result, what do you think the result will look like? What should the new mean be? Will the variance by wider, narrower, or the same?

```
In [11]: import numpy as np

         def multiply(mu1, sig1, mu2, sig2):
             m = (sig1*mu2 + sig2*mu1) / (sig1+sig2)
             s = 1. / (1./sig1 + 1./ sig2)
             return (m,s)

         xs = np.arange(16, 30, 0.1)


         m1,s1 = 23, 5
         m, s = multiply(m1,s1,m1,s1)

         ys = [stats.gaussian(x,m1,s1) for x in xs]
         p1, = plt.plot (xs,ys)

         ys = [stats.gaussian(x,m,s) for x in xs]
         p2, = plt.plot (xs,ys)

         plt.legend([p1,p2],['original', 'multiply'])
         plt.show()
```



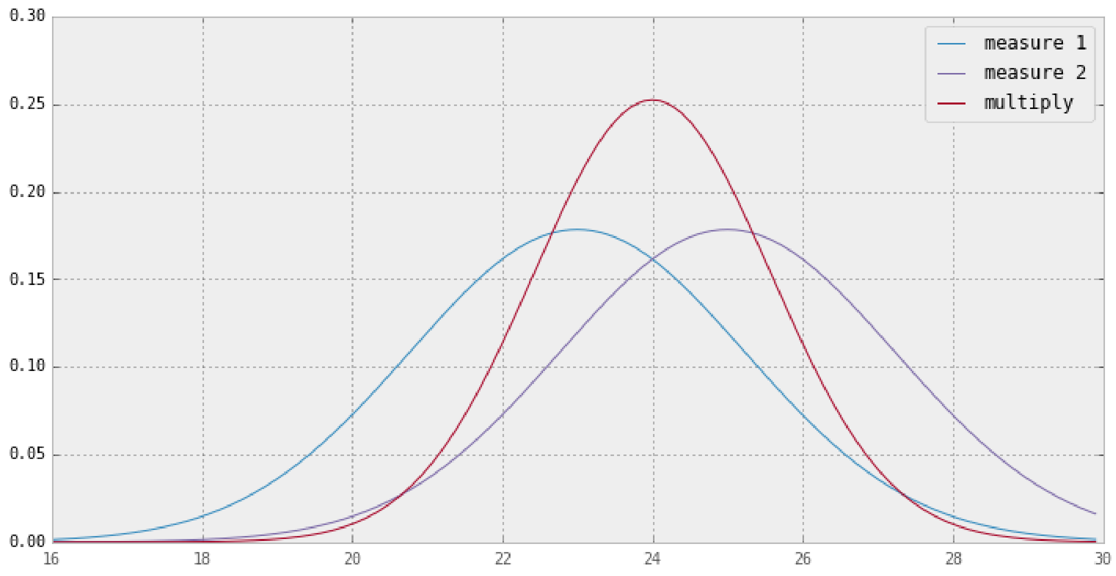
The result is either amazing or what you would expect, depending on your state of mind. I must admit I vacillate freely between the two! Note that the result of the multiplation is taller and narrow than the original Gaussian but the mean is the same. Does this match your intuition of what the result should have been?

If we think of the Gaussians as two measurements, this makes sense. If I measure twice and get the same value, I should be more confident in my answer than if I just measured once. If I measure twice and get $23m$ each time, I should conclude that the length is close to $23m$. So the mean should be 23. 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 useful saying and practice due to this fact! The Gaussian is just a mathematical model of this physical fact, so we should expect the math to follow our physical process.

Now let’s multiply two gaussians (or equivalently, two measurements) that are partially separated. In other words, their means will be different, but their variances will be the same. What do you think the result will be? Think about it, and then look at the graph.

```
In [12]: xs = np.arange(16, 30, 0.1)

         m1,s1 = 23, 5
         m2,s2 = 25, 5
         m, s = multiply(m1,s1,m2,s2)

         ys = [stats.gaussian(x,m1,s1) for x in xs]
         p1, = plt.plot (xs,ys)

         ys = [stats.gaussian(x,m2,s2) for x in xs]
         p2, = plt.plot (xs,ys)

         ys = [stats.gaussian(x,m,s) for x in xs]
         p3, = plt.plot(xs,ys)
         plt.legend([p1,p2,p3],['measure 1', 'measure 2', 'multiply'])
         plt.show()
```



Another beautiful result! If I handed you a measuring tape and asked you to measure the distance from table to a wall, and you got 23m, and then a friend make the same measurement and got 25m, your best guess must be 24m.

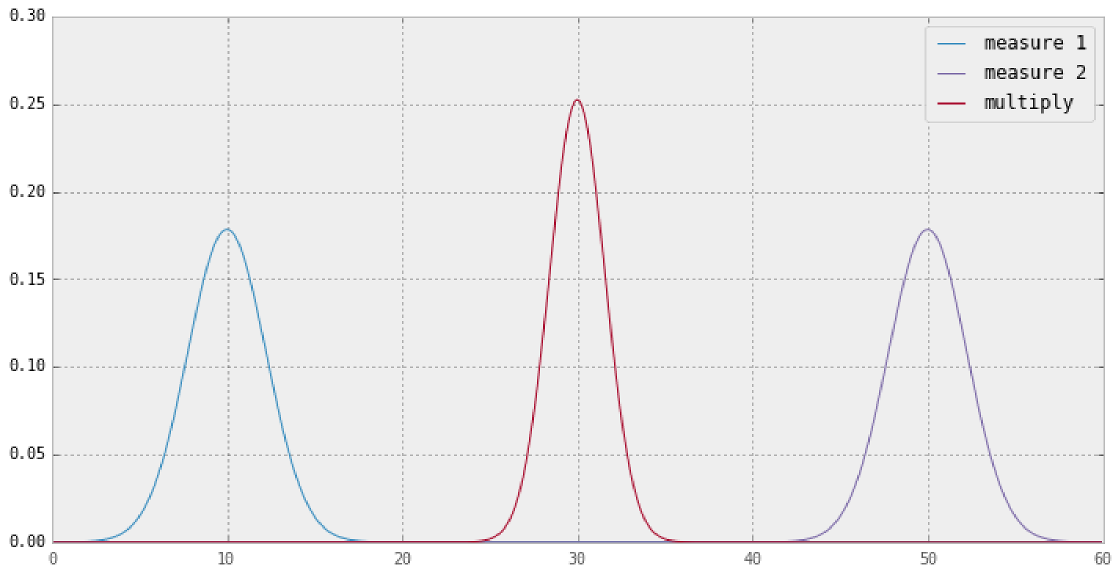
That is fairly counter-intuitive, so let's consider it further. Perhaps a more reasonable assumption would be that either you or your coworker just made a mistake, and the true distance is either 23 or 25, but certainly not 24. Surely that is possible. However, suppose the two measurements you reported as 24.01 and 23.99. In that case you would agree that in this case the best guess for the correct value is 24? Which interpretation we choose depends on the properties of the sensors we are using. Humans make galling mistakes, physical sensors do not.

This topic is fairly deep, and I will explore it once we have completed our Kalman filter. For now I will merely say that the Kalman filter requires the interpretation that measurements are accurate, with Gaussian noise, and that a large error caused by misreading a measuring tape is not Gaussian noise.

For now I ask that you trust me. The math is correct, so we have no choice but to accept it and use it. We will see how the Kalman filter deals with movements vs error very soon. In the meantime, accept that 24 is the correct answer to this problem.

One final test of your intuition. What if the two measurements are widely separated?

```
In [13]: xs = np.arange(0, 60, 0.1)

         m1,s1 = 10, 5
         m2,s2 = 50, 5
         m, s = multiply(m1,s1,m2,s2)

         ys = [stats.gaussian(x,m1,s1) for x in xs]
         p1, = plt.plot (xs,ys)

         ys = [stats.gaussian(x,m2,s2) for x in xs]
         p2, = plt.plot (xs,ys)

         ys = [stats.gaussian(x,m,s) for x in xs]
         p3, = plt.plot(xs,ys)
         plt.legend([p1,p2,p3],['measure 1', 'measure 2', 'multiply'])
         plt.show()
```



This result bothered me quite a bit when I first learned it. If my first measurement was 10, and the next one was 50, why would I choose 30 as a result? And why would I be *more* confident? Doesn't it make sense that either one of the measurements is wrong, or that I am measuring a moving object? Shouldn't the result be nearer 50? And, shouldn't the variance be larger, not smaller?

Well, no. Recall the g-h filter chapter. In that chapter we agreed that if I weighed myself on two scales, and the first read 160lbs while the second read 170lbs, and both were equally accurate, the best estimate was 165lbs. Futhermore I should be a bit more confident about 165lbs vs 160lbs or 170lbs because I know have two readings, both near this estimate, increasing my confidence that neither is wildly wrong.

Let's look at the math again to convince ourselves that the physical interpretation of the Gaussian equations makes sense.

$$\mu = \frac{\sigma_1^2\mu_2 + \sigma_2^2\mu_1}{\sigma_1^2 + \sigma_2^2}$$

If both scales have the same accuracy, then $\sigma_1^2 = \sigma_2^2$, and the resulting equation is

$$\mu = \frac{\mu_1 + \mu_2}{2}$$

which is just the average of the two weighings. If we look at the extreme cases, assume the first scale is very much more accurate than than the second one. At the limit, we can set $\sigma_1^2 = 0$, yielding

$$\mu = \frac{0 * \mu_2 + \sigma_2^2\mu_1}{\sigma_2^2},$$

or just

$$\mu = \mu_1$$

Finally, if we set $\sigma_1^2 = 9\sigma_2^2$, then the resulting equation is

$$\mu = \frac{9\sigma_2^2\mu_2 + \sigma_2^2\mu_1}{9\sigma_2^2 + \sigma_2^2}$$

or just

$$\mu = \frac{1}{10}\mu_1 + \frac{9}{10}\mu_2$$

This again fits our physical intuition of favoring the second, accurate scale over the first, inaccurate scale.

## 5.4 Implementing the Sensing Step

Recall the histogram filter uses a numpy array to encode our belief about the position of our dog at any time. That array stored our belief of our dog's position in the hallway using 10 discrete positions. This was very crude, because with a 100m hallway that corresponded to positions 10m apart. It would have been trivial to expand the number of positions to say 1,000, and that is what we would do if using it for a real problem. But the problem remains that the distribution is discrete and multimodal - it can express strong belief that the dog is in two positions at the same time.

Therefore, we will use a single Gaussian to reflect our current belief of the dog's position. In other words, we will use $dog_{pos} = \mathcal{N}(\mu, \sigma^2)$. Gaussians extend to infinity on both sides of the mean, so the single Gaussian will cover the entire hallway. They are unimodal, and seem to reflect the behavior of real-world sensors - most errors are small and clustered around the mean. Here is the entire implementation of the sense function for a Kalman filter:

```
In [14]: def sense(mu, sigma, measurement, measurement_sigma):
             return multiply(mu, sigma, measurement, measurement_sigma)
```

Kalman filters are supposed to be hard! But this is very short and straightforward. All we are doing is multiplying the Gaussian that reflects our belief of where the dog was with the new measurement. Perhaps this would be clearer if we used more specific names:

```
In [15]: def sense_dog(dog_pos, dog_sigma, measurement, measurement_sigma):
             return multiply(dog_pos, dog_sigma, measurement, measurement_sigma)
```

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 don't use variable names with 'dog' in them. Still, the `sense_dog()` function should make what we are doing very clear.

Let's look at an example. We will suppose that our current belief for the dog's position is $N(2, 5)$. Don't worry about where that number came from. It may appear that we have a chicken and egg problem, in that how do we know the position before we sense it, but we will resolve that shortly. We will create a `DogSensor` object initialized to be at position 0.0, and with no velocity, and modest noise. This corresponds to the dog standing still at the far left side of the hallway. Note that we mistakenly believe the dog is at postion 2.0, not 0.0.

```
In [16]: dog = DogSensor(velocity=0, noise=1)

         pos,s = 2, 5
         for i in range(20):
             pos,s = sense(pos, s, dog.sense(), 5)
             print('time:', i, '\tposition =', "%.3f" % pos, '\tvariance =', "%.3f" % s)
```

```
time: 0         position = 0.793        variance = 2.500
time: 1         position = 0.575        variance = 1.667
time: 2         position = 0.277        variance = 1.250
time: 3         position = 0.348        variance = 1.000
time: 4         position = 0.401        variance = 0.833
time: 5         position = 0.231        variance = 0.714
time: 6         position = 0.386        variance = 0.625
time: 7         position = 0.517        variance = 0.556
time: 8         position = 0.452        variance = 0.500
time: 9         position = 0.332        variance = 0.455
time: 10         position = 0.298        variance = 0.417
time: 11         position = 0.280        variance = 0.385
time: 12         position = 0.197        variance = 0.357
time: 13         position = 0.132        variance = 0.333
```

```
time: 14         position = 0.116        variance = 0.312
time: 15         position = 0.076        variance = 0.294
time: 16         position = 0.135        variance = 0.278
time: 17         position = 0.040        variance = 0.263
time: 18         position = 0.003        variance = 0.250
time: 19         position = 0.006        variance = 0.238
```

Because of the random numbers I do not know the exact values that you see, but the position should have converged very quickly to almost 0 despite the initial error of believing that the position was 2.0. Furthermore, the variance should have quickly converged from the intial value of 5.0 to 0.238.

By now the fact that we converged to a position of 0.0 should not be terribly suprising. All we are doing is computing `new_pos = old_pos * measurement` and the measurement is a normal distribution around 0, so we should get very close to 0 after 20 iterations. But the truly amazing part of this code is how the variance became 0.238 despite every measurement having a variance of 5.0.

If we think about the physical interpretation of this is should be clear that this is what should happen. If you sent 20 people into the hall with a tape measure to physically measure the position of the dog you would be very confident in the result after 20 measurements - more confident than after 1 or 2 measurements. So it makes sense that as we make more measurements the variance gets smaller.

Mathematically it makes sense as well. Recall the computation for the variance after the multiplication: $\sigma^2 = 1/(\frac{1}{\sigma_1} + \frac{1}{\sigma_2})$. We take the reciprocals of the sigma from the measurement and prior belief, add them, and take the reciprocal of the result. Think about that for a moment, and you will see that this will always result in smaller numbers as we proceed.

## 5.5 Implementing Updates

That is a beautiful result, but it is not yet a filter. We assumed that the dog was sitting still, an extremely dubious assumption. Certainly it is a useless one - who would need to write a filter to track nonmoving objects? The histogram used a loop of sense and update functions, and we must do the same to accomodate movement.

How how do we perform the update function with gaussians? Recall the histogram method:

```
def update(pos, move, p_correct, p_under, p_over):
    n = len(pos)
    result = array(pos, dtype=float)
    for i in range(n):
            result[i] =  \
            pos[(i-move) % n]   * p_correct + \
            pos[(i-move-1) % n] * p_over + \
            pos[(i-move+1) % n] * p_under
    return result
```

In a nutshell, we shift the probability vector by the amount we believe the animal moved, and adjust the probability. How do we do that with gaussians?

It turns out that we just add gaussians. Think of the case without gaussians. I think my dog is at 7.3m, and he moves 2.6m to right, where is he now? Obviously, $7.3 + 2.6 = 9.9$. He is at 9.9m. Abstractly, the algorithm is `new_pos = old_pos + dist_moved`. It does not matter if we use floating point numbers or gaussians for these values, the algorithm must be the same.

How is addition for gaussians performed? It turns out to be very simple:

$$N(\mu_1, {\sigma_1}^2) + N(\mu_2, {\sigma_2}^2) = N(\mu_1 + \mu_2, {\sigma_1}^2 + {\sigma_2}^2)$$

All we do is add the means and the variance separately! Does that make sense? Think of the physical representation of this abstract equation. $\mu_1$ is the old position, and $\mu_2$ is the distance moved. Surely it makes sense that our new position is $\mu_1 + \mu_2$. What about the variance? It is perhaps harder to form an intuition about this. However, recall that with the `update()` function for the histogram filter we always lost information - our confidence after the update was lower than our confidence before the update. Perhaps this makes sense - we don't really know where the dog is moving, so perhaps the confidence should get smaller (variance gets larger). I assure you that the equation for gaussian addition is correct, and derived by basic algebra. Therefore it is reasonable to expect that if we are using gaussians to model physical events, the results must correctly describe those events.

I recognize the amount of hand waving in that argument. Now is a good time to either work through the algebra to convince yourself of the mathematical correctness of the algorithm, or to work through some examples and see that it behaves reasonably. This book will do the latter.

So, here is our implementation of the update function:

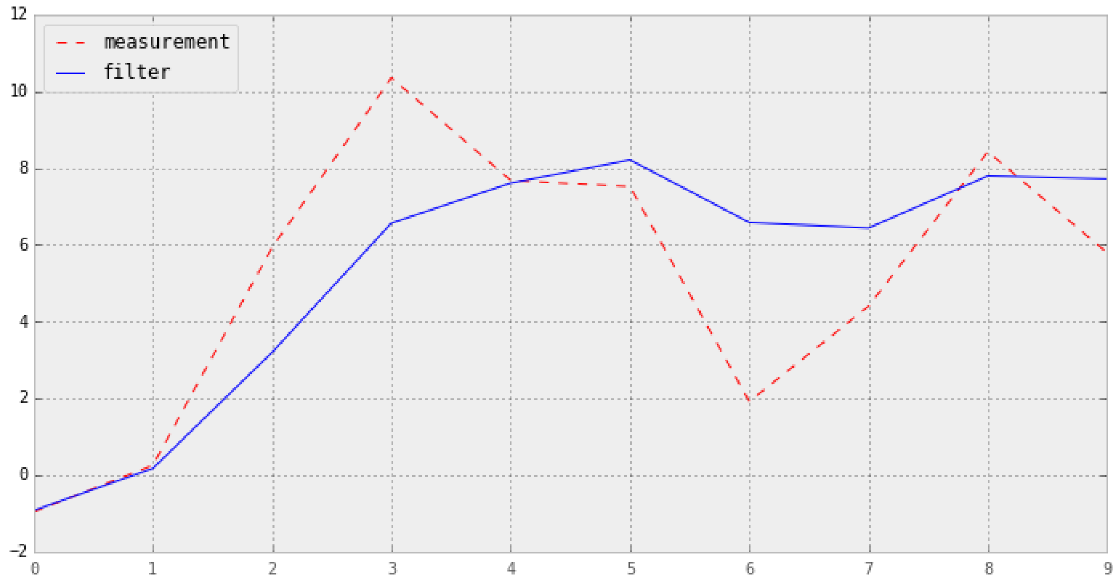
```
In [17]: def update(pos, sigma, movement, movement_sigma):
             return (pos + movement, sigma + movement_sigma)
```

What is left? Just calling these functions. The histogram did nothing more than loop over the `sense()` and `update()` functions, so let's do the same.

```
In [18]: # assume dog is always moving 1m to the right
         movement = 1
         movement_error = 2
         sensor_error = 10
         pos = (0, 500)   # gaussian N(0,50)

         dog = DogSensor(pos[0], velocity=movement, noise=sensor_error)

         zs = []
         ps = []

         for i in range(10):
             pos = update(pos[0], pos[1], movement, movement_error)
             print('UPDATE: %.4f,\t%.4f' % (pos[0], pos[1]))

             Z = dog.sense()
             zs.append(Z)

             pos = sense(pos[0], pos[1], Z, sensor_error)
             ps.append(pos[0])

             print('SENSE:  %.4f,\t%.4f' % (pos[0], pos[1]))
             print()

         p1, = plt.plot(zs,c='r', linestyle='dashed')
         p2, = plt.plot(ps, c='b')
         plt.legend([p1,p2], ['measurement', 'filter'], 2)
         plt.show()
```

```
UPDATE: 1.0000,        502.0000
SENSE:  -0.9237,        9.8047

UPDATE: 0.0763,        11.8047
SENSE:  0.1727,        5.4138

UPDATE: 1.1727,        7.4138
SENSE:  3.2003,        4.2574
```

```
UPDATE: 4.2003,        6.2574
SENSE:  6.5697,        3.8490

UPDATE: 7.5697,        5.8490
SENSE:  7.6101,        3.6904

UPDATE: 8.6101,        5.6904
SENSE:  8.2152,        3.6267

UPDATE: 9.2152,        5.6267
SENSE:  6.5868,        3.6007

UPDATE: 7.5868,        5.6007
SENSE:  6.4438,        3.5900

UPDATE: 7.4438,        5.5900
SENSE:  7.7971,        3.5856

UPDATE: 8.7971,        5.5856
SENSE:  7.7203,        3.5838
```



There is a fair bit of arbitrary constants code above, but don't worry about it. What does require explanation are the first few lines:

```
movement = 1
movement_error = 2
```

For the moment we are assuming that we have some other sensor that detects how the dog is moving. For example, there could be an inertial sensor clipped onto the dog's collar, and it reports how far the dog moved each time it is triggered. The details don't matter. The upshot is that we have a sensor, it has noise, and so we represent it with a Gaussian. Later we will learn what to do if we do not have a sensor for the `update()` step.

For now let's walk through the code and output bit by bit.

```
movement = 1
movement_error = 2
sensor_error = 10
pos = (0, 500)   # gaussian N(0,500)
```

The first lines just set up the initial conditions for our filter. We are assuming that the dog moves steadily to the right 1m at a time. We have a relatively low error of 2 for the movement sensor, and a higher error of 10 for the RFID position sensor. Finally, we set our belief of the dog's initial position as $N(0, 500)$. Why those numbers. Well, 0 is as good as any number if we don't know where the dog is. But we set the variance to 500 to denote that we have no confidence in this value at all. 100m is almost as likely as 0 with this value for the variance.

Next we initialize the RFID simulator with

```
dog = DogSensor(pos[0], velocity=movement, noise=sensor_error)
```

It may seem very 'convienent' to set the simulator to the same position as our guess, and it is. Do not fret. In the next example we will see the effect of a wildly inaccurate guess for the dog's initial position.

The next code allocates an array to store the output of the measurements and filtered positions.

```
zs = []
ps = []
```

This is the first time that I am introducing standard nomenclature used by the Kalman filtering literature. It is traditional to call our measurement $Z$, and so I follow that convention here. As an aside, I find the nomenclature used by the literature very obscure. However, if you wish to read the literature you will have to become used to it, so I will not use a much more readable variable name such as $m$ or $measure$.

Now we just enter our `sense() - >update()` loop.

```
for i in range(10):
    pos = update(pos[0], pos[1], movement, sensor_error)
    print 'UPDATE:', "%.4f" %pos[0], ", %.4f" %pos[1]
```

Wait, why `update()` before sense? It turns out the order does not matter once, but the first call to `DogSensor.sense()` assumes that the dog has already moved, so we start with the update step. In practice you will order these calls based on the details of your sensor, and you will very typically do the `sense()` first.

So we call the update function with the gaussian representing our current belief about our position, the another gaussian representing our belief as to where the dog is moving, and then print the output. Your output will differ, but when writing this I get this as output:

```
UPDATE: 1.000 502.000
```

What is this saying? After the update, we believe that we are at 1.0, and the variance is now 502.0. Recall we started at 500.0. The variance got worse, which is always what happens during the update step.

```
Z = dog.sense()
zs.append(Z)
```

Here we sense the dog's position, and store it in our array so we can plot the results later.

Finally we call the sense function of our filter, save the result in our *ps* array, and print the updated position belief:

```
pos = sense(pos[0], pos[1], Z, movement_error)
ps.append(pos[0])
print 'SENSE:', "%.4f" %pos[0], ", %.4f" %pos[1]
```

Your result will be different, but I get

```
SENSE: 1.6279 , 9.8047
```

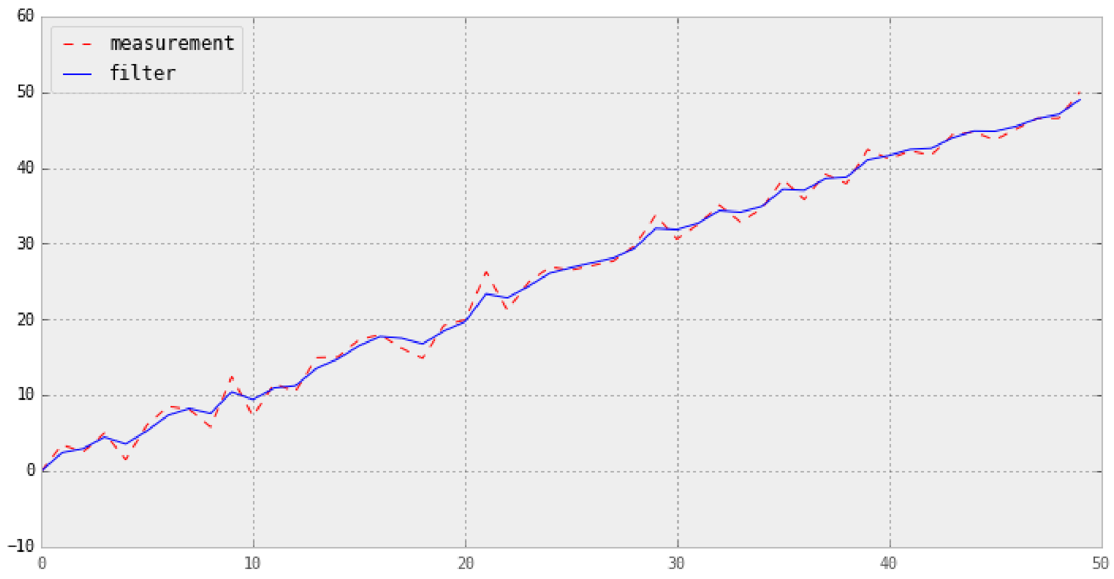
as the result. What is happening? Well, at this point the dog is really at 1.0, however the predicted position is 1.6279. What is happening is the RFID sensor has a fair amount of noise, and so we compute the position as 1.6279. That is pretty far off from 1, but this is just are first time through the loop. Intuition tells us that the results will get better as we make more measurements, so let's hope that this is true for our filter as well. Now look at the variance: 9.8047. It has dropped tremendously from 502.0. Why? Well, the RFID has a reasonably small variance of 2.0, so we trust it far more than our previous belief. At this point there is no way to know for sure that the RFID is outputting reliable data, so the variance is not 2.0, but is has gotten much better.

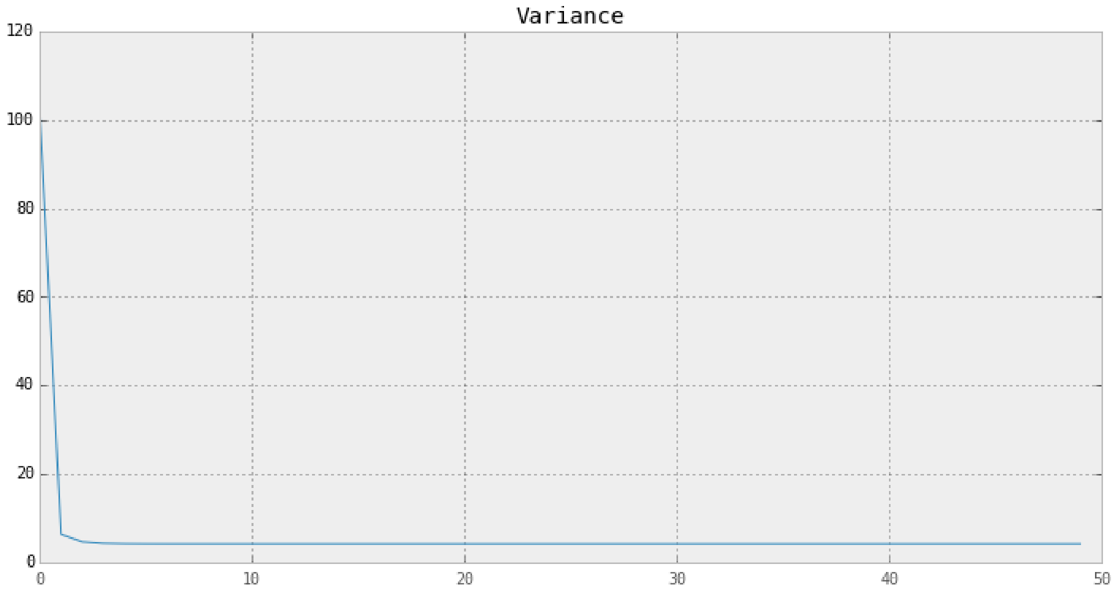
Now the software just loops, calling `update()` and `sense()` in turn. Because of the random sampling I do not know exactly what numbers you are seeing, but the final position is probably between 9 and 11, and the final variance is probably around 3.5. After several runs I did see the final position nearer 7, which would have been the result of several measurements with relatively large errors.

Now look at the plot. The noisy measurements are plotted in with a dotted red line, and the filter results are in the solid blue line. Both are quite noisy, but notice how much noisier the measurements (red line) are. This is your first Kalman filter shown to work!

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. This time we will plot both the output of the filter and the variance.

```
In [19]: %precision 2
         # assume dog is always moving 1m to the right
         movement = 1
         movement_error = 2
         sensor_error = 4.5
         pos = (0, 100)   # gaussian N(0,50)

         dog = DogSensor(pos[0], velocity=movement, noise=sensor_error)

         zs = []
         ps = []
         vs = []

         for i in range(50):
             pos = update(pos[0], pos[1], movement, movement_error)
             Z = dog.sense()
             zs.append(Z)
             vs.append(pos[1])

             pos = sense(pos[0], pos[1], Z, sensor_error)
             ps.append(pos[0])

         #plt.subplot(121)
         p1, = plt.plot(zs,c='r', linestyle='dashed')
         p2, = plt.plot(ps, c='b')
         plt.legend([p1,p2], ['measurement', 'filter'], 2)
         plt.show()

         plt.plot(vs)
         plt.title('Variance')
         plt.show()
         print ([float("%0.4f" % v) for v in vs])
```





```
[102.0, 6.3099, 4.6267, 4.2812, 4.1939, 4.1708, 4.1646, 4.1629, 4.1624, 4.1623, 4.1623, 4.1623, 4.1623,
```

Here we can see that the variance converges very quickly to roughly 4.1623 in 10 steps. We interpret this as meaning that we become very confident in our position estimate very quickly. The first few measurements are unsure due to our uncertainty in our guess at the initial position, but the filter is able to quickly determine an accurate estimate.

Before I go on, I want to emphasize that this code fully implements a 1D Kalman filter. If you have tried to read the literatue, you are perhaps surprised, because this looks nothing like the complex, endless pages of math in those books. To be fair, the math gets a bit more complicated in multiple dimensions, but not by much. So long as we worry about *using* the equations rather than *deriving* them we can create Kalman filters without a lot of effort. Moreover, I hope you'll agree that you have a decent intuitive grasp of what is happening. We represent our beliefs with Gaussians, and our beliefs get better over time because more measurement means more data to work with. "Measure twice, cut once!"

## 5.6 Relationship to the g-h Filter

In the first chapter I stated that the Kalman filter is a form of g-h filter. However, we have been reasoning about the probability of Gaussians, and not used any of the reasoning or equations of the first chapter. A trivial amount of algebra will reveal the relationship, so let's do that now. It's not particularly illuminating algebra, so feel free to skip to the bottom to see the final equation that relates $g$ to the variances.

The equation for our estimate is:

$$\mu_{x'} = \frac{\sigma_1^2\mu_2 + \sigma_2^2\mu_1}{\sigma_1^2 + \sigma_2^2}$$

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

$$\sigma_{x'}^2 = \frac{1}{\frac{1}{\sigma_1^2} + \frac{1}{\sigma_2^2}} = \frac{1}{\frac{1}{a} + \frac{1}{b}}$$

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

$$\begin{aligned}
\mu_{x'} &= x + \frac{a}{a+b}(y-x) \\
&= x + \frac{\sigma^2_{x'}}{b}(y-x) \\
&= x + g_n(y-x)
\end{aligned}$$

$$\blacksquare$$

where

$$g_n = \frac{\sigma^2_{x'}}{\sigma^2_y}$$

The end result is multipying 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 truly a constant, as it varies with each time step as the variance changes, but it is truly the same formula. We can also derive the formula for $h$ in the same way but I don't find this a particularly interesting derivation. The end result is

$$h_n = \frac{COV(x, \dot{x})}{\sigma^2_y}$$

The takeaway point is that $g$ and $h$ are specified fully by the variance and covariances of the measurement and preditions at time $n$. In other words, we are just picking a point between the measurement and prediction by a scale factor determined by the quality of each of those two inputs. That is all the Kalman filter is.

**Excercise:** Modify the values of `movement_error` and `sensor_error` and note the effect on the filter and on the variance. Which has a larger effect on the value that variance converges to. For example, which results in a smaller variance:

```
movement_error = 40
sensor_error = 2
```

or:

```
movement_error = 2
sensor_error = 40
```

## 5.7 Introduction to Designing a Filter

So far we have developed our filter based on the dog sensors introduced in the Discrete Bayesian filter chapter. We are used to this problem by now, and may feel ill-equiped to implement a Kalman filter for a different problem. To be honest, there is still quite a bit of information missing from this presentation. The next chapter will fill in the gaps. Still, lets 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 2.13.

We do not have a real sensor to read, so we will simulate the sensor with the following function. We have hard-coded the voltage to 16.3 - obviously the voltage will differ based on the temperature, but that is not important to our filter design.

```
In [20]: temp_variance = 2.13**2
         def volt():
             return random.randn()*temp_variance + 16.3
```

We generate white noise with a given variance using the equation `random.randn() * variance`. The specification gives us the standard deviation of the noise, not the variance, but recall that variance is just the square of the standard deviation. Hence we raise 2.13 to the second power. > **Sidebar**: spec sheets are just what they sound like - specifications. Any individual sensor will exhibit different performance based on normal manufacturing variations. Numbers given are often maximums - the spec is a guarantee that the performace will be at least that good. So, our sensor might have standard deviation of 1.8. 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 everything 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.

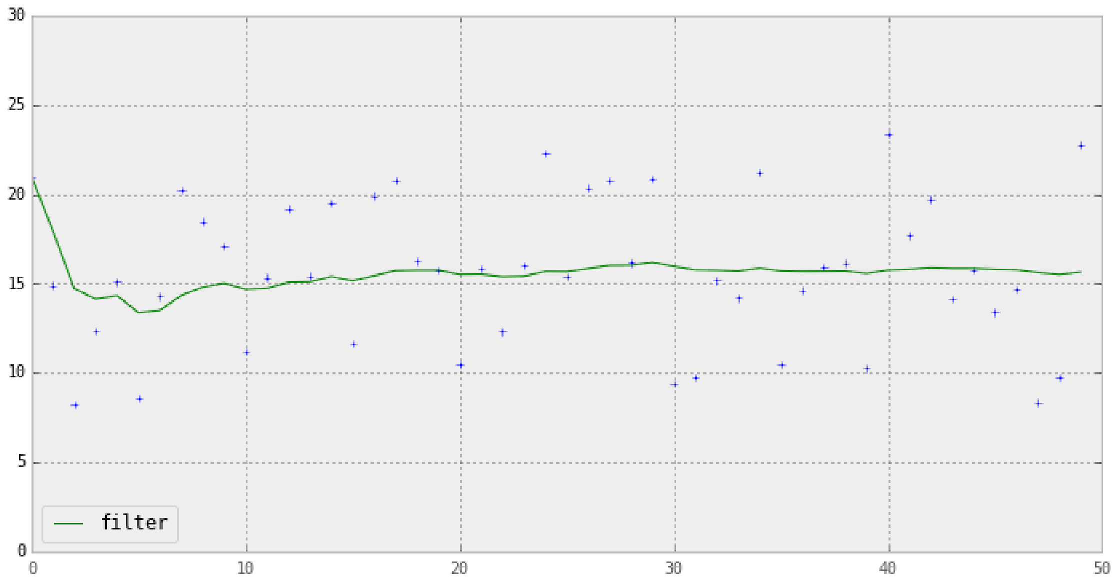
> For example, I just randomly looked up a data sheet for an airflow sensor. There is a field "Repeatability", with the value "±0.50% Reading". Is this a Gaussian? Is there a bias? For example, perhaps the repeatibility is nearly 0.0% at low temperatures, and always nearly +0.50 at high temperatures. 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 leads 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 good Kalman filter.

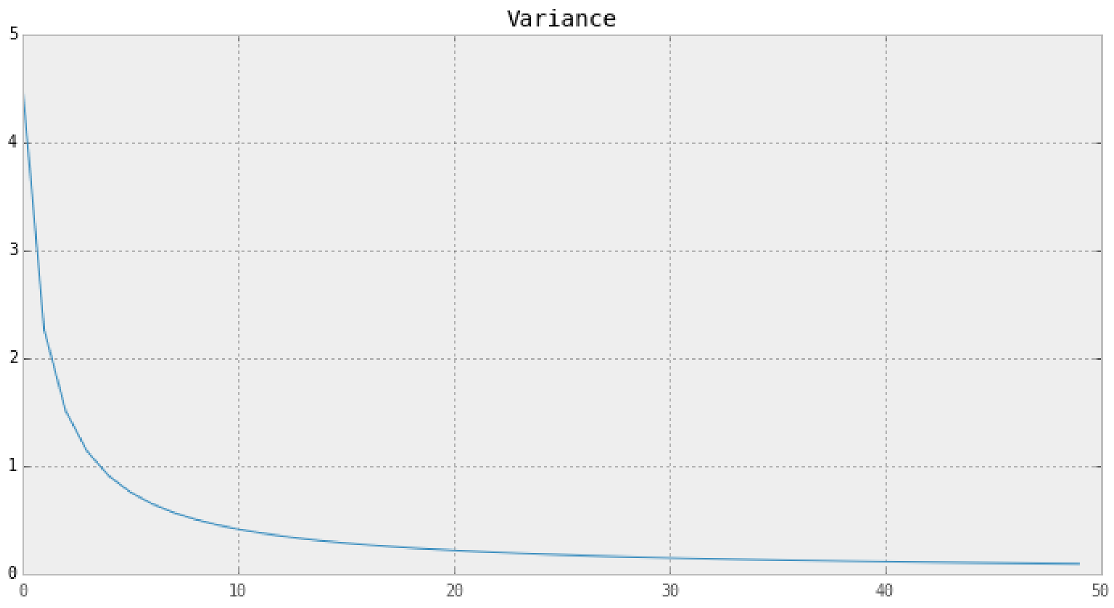
Now we need to write the Kalman filter processing loop. As with our previous problem, we need to perform a cycle of sensing and updating. The sensing step probably seems clear - call `volt()` to get the measurement, pass the result into `sense()` function, but what about the update step? We do not have 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 movement, and no error associated with them, so we will just set both to zero. Let's see what happens.

```
In [21]: sensor_error = temp_variance
         movement_error = 0
         movement = 0
         voltage = (25,1000) #who knows what the first value is?

         zs = []
         ps = []
         vs = []
         N=50

         for i in range(N):
             Z = volt()
             zs.append(Z)

             voltage = sense(voltage[0], voltage[1], Z, sensor_error)
             ps.append(voltage[0])
             vs.append(voltage[1])
```

```
        voltage = update(voltage[0], voltage[1], movement, movement_error)

    plt.scatter(range(N), zs, marker='+')
    p1, = plt.plot(ps, c='g')
    plt.legend([p1], ['filter'], 3)
    plt.xlim((0,N));plt.ylim((0,30))
    plt.show()
    plt.plot(vs)
    plt.title('Variance')
    plt.show()
    print('Variance converges to',vs[-1])
    print('Last voltage is',voltage[0])
```





```
Variance converges to 0.0907297673624
Last voltage is 15.6491261618
```

The first plot shows the individual sensor measurements marked with '+'s 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 up to around 16.9 as an output and also as low as 15.5 or so.

The second plot shows how the variance converges over time. Compare this plot to the variance plot for the dog sensor. While this does converge to a very small value, it is much slower than the dog problem. The next section **Explaining the Results - Multi-Sensor Fusion** explains why this happens.

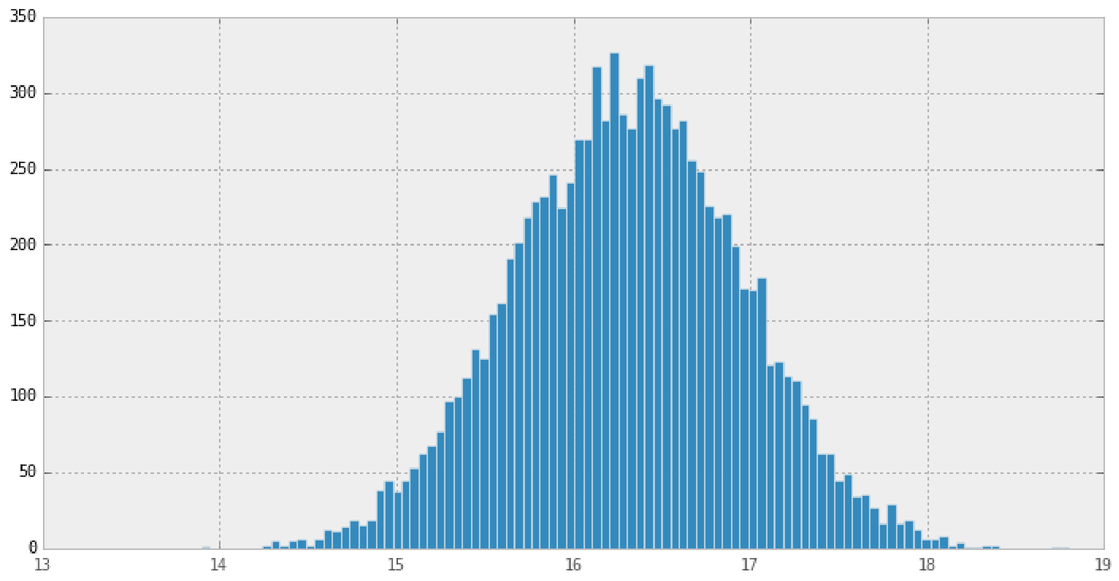
**Exercise(optional):** Write a function that runs the Kalman filter many times and record what value the voltage converges to each time. Plot this as a histogram. After 10,000 runs do the results look normally distributed? Does this match your intuition of what should happen?

use plt.hist(data,bins=100) to plot the histogram.

```
In [22]: #Your code here
```

Solution

```
In [23]: sensor_error = temp_variance

         def VKF():
             voltage=(14,1000)
             for i in range(N):
                 Z = volt()
                 voltage = sense(voltage[0], voltage[1], Z, sensor_error)
             return voltage[0]

         vs = []
         for i in range (10000):
             vs.append (VKF())
         plt.hist(vs, bins=100)
         plt.show()
```



Discussion

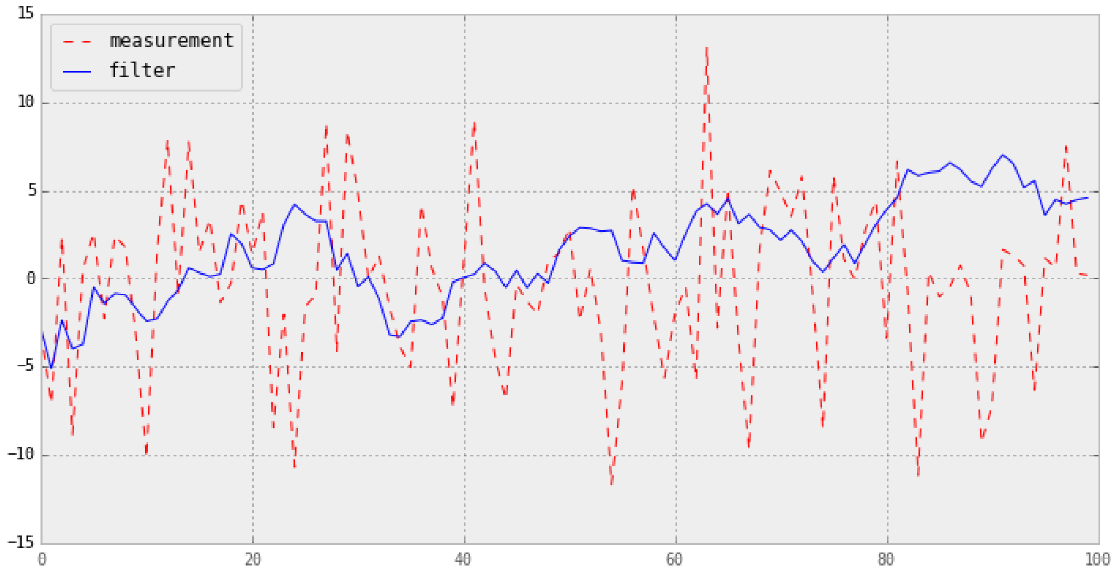
The results do in fact look like a normal distribution. Each voltage is Gaussian, and the **Central Limit Theorem** guarantees that a large number of Gaussians is normally distributed. We will discuss this more in a subsequent math chapter.

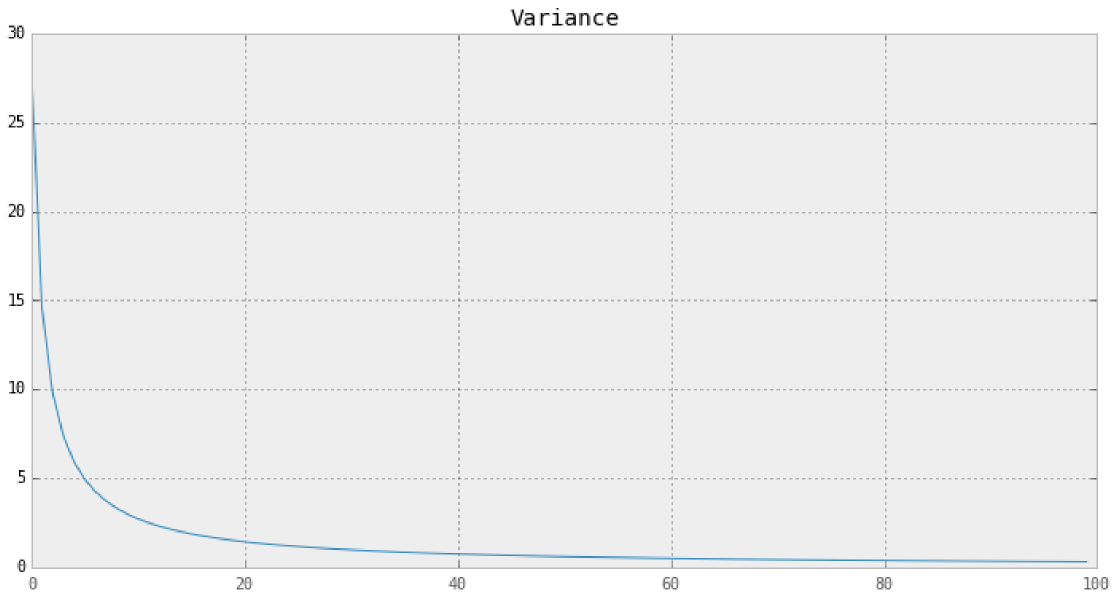
## 5.8 Explaining the Results - Multi-Sensor Fusion

So how does the Kalman filter do so well? I have glossed over one aspect of the filter as it becomes confusing to address too many points at the same time. We will return to the dog tracking problem. We used two sensors to track the dog - the RFID sensor that detects position, and the inertial tracker that tracked movement. However, we have focussed all of our attention on the position sensor. Let's change focus and see how the filter performs if the intertial tracker is also noisy. This will provide us with an vital insight into the performance of Kalman filters.

```
In [24]: sensor_error = 30
         movement_sensor = 30
         pos = (0,500)

         dog = DogSensor(0, velocity=movement, noise=sensor_error)

         zs = []
         ps = []
         vs = []

         for i in range(100):
             Z = dog.sense()
             zs.append(Z)

             pos = sense(pos[0], pos[1], Z, sensor_error)
             ps.append(pos[0])
             vs.append(pos[1])
```

```
        pos = update(pos[0], pos[1], movement+ random.randn(), movement_error)

    p1, = plt.plot(zs,c='r', linestyle='dashed')
    p2, = plt.plot(ps, c='b')
    plt.legend([p1,p2], ['measurement', 'filter'], 2)
    plt.show()
    plt.plot(vs)
    plt.title('Variance')
    plt.show()
```





This result is worse than the example where only the measurement sensor was noisy. Instead of being mostly straight, this time the filter's output is distintly jagged. But, it still mostly tracks the dog. What is happening here?

This illustrates the effects of *multi-sensor fusion*. Suppose we get a position reading of -28.78 followed by 31.43. From that information alone it is impossible to tell if the dog is standing still during very noisy measurements, or perhaps sprinting from -29 to 31 and being accurately measured. But we have a second source of information, his velocity. Even when the velocity is also noisy, it constrains what our beliefs might be. For example, suppose that with the 31.43 position reading we get a velocity reading of 59. That matches the difference between the two positions quite well, so this will lead us to believe the RFID sensor and the velocity sensor. Now suppose we got a velocity reading of 1.7. This doesn't match our RFID reading very well - it suggests that the dog is standing still or moving slowly.

When sensors measure different aspects of the system and they all agree we have strong evidence that the sensors are accurate. And when they do not agree it is a strong indication that one or more of them are inaccurate.

We will formalize this mathematically in the next chapter; for now trust this intuitive explanation. We use this sort of reasoning every day in our lives. If one person tells us something that seems far fetched we are inclined to doubt them. But if several people independently relay the same information we attach higher credence to the data. If one person disagrees with several other people, we tend to distrust the outlier. If we know the people that might alter our belief. If a friend is inclined to practical jokes and tall tales we may put very little trust in what they say. If one lawyer and three lay people opine on some fact of law, and the lawyer disagees with the three you'll probably lend more credence to what the lawyer says because of her expertise. In the next chapter we will learn how to mathematicall model this sort of reasoning.

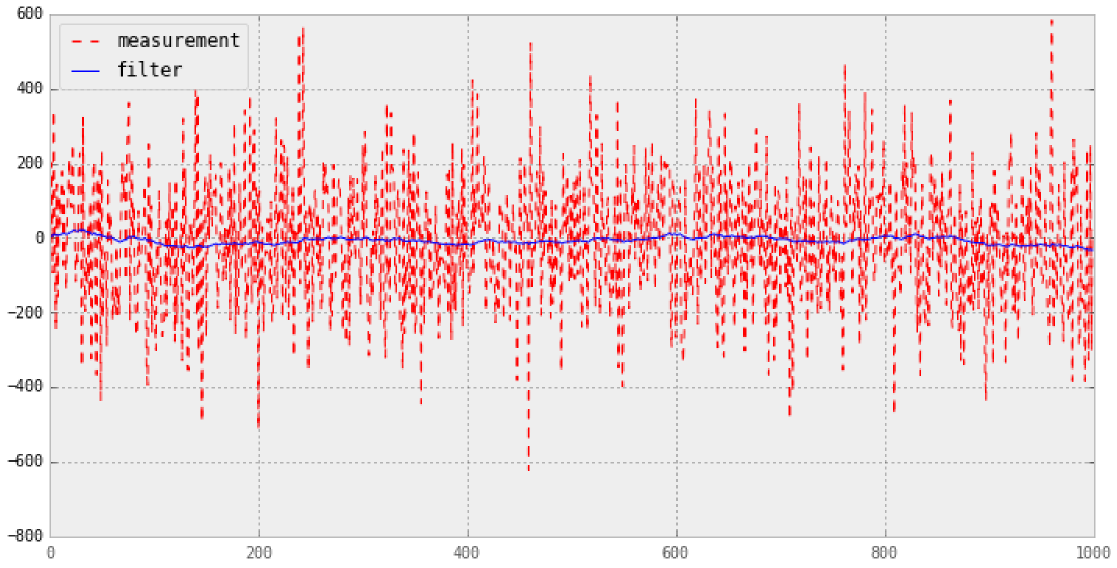
## 5.9 More examples

### 5.9.1 Example: Extreme Amounts of Noise

So I didn't put a lot of noise in the signal, and I also 'correctly guessed' that the dog was at position 0. How does the filter perform in real world conditions? Let's explore and find out. I will start by injecting a lot of noise in the RFID sensor. I will inject an extreme amount of noise - noise that apparently swamps the actual measurement. What does your intution tell about how the filter will perform if the noise is allowed to be anywhere from -300 or 300. In other words, an actual position of 1.0 might be reported as 287.9, or -189.6, or any other number in that range. Think about it before you scroll down.

```
In [25]: sensor_error = 30000
         movement_error = 2
         pos = (0,500)

         dog = DogSensor(pos[0], velocity=movement, noise=sensor_error)

         zs = []
         ps = []

         for i in range(1000):
             pos = update(pos[0], pos[1], movement, movement_error)

             Z = dog.sense()
             zs.append(Z)

             pos = sense(pos[0], pos[1], Z, sensor_error)
             ps.append(pos[0])


         p1, = plt.plot(zs,c='r', linestyle='dashed')
```

```
p2, = plt.plot(ps, c='b')
plt.legend([p1,p2], ['measurement', 'filter'], 2)
plt.show()
```



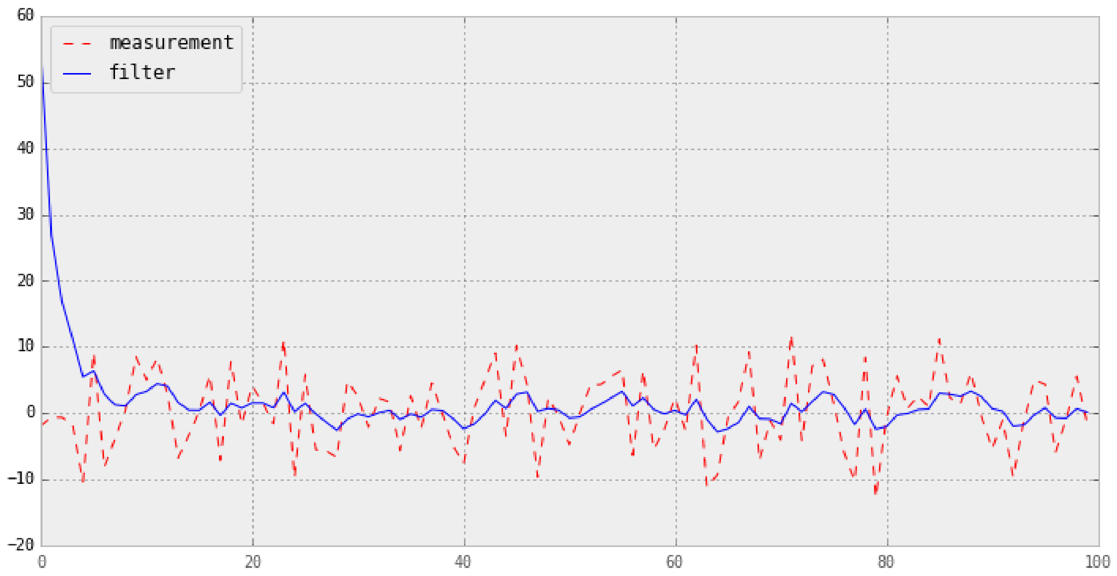
In this example the noise is extreme yet the filter still outputs a nearly straight line! This is an astonishing result! What do you think might be the cause of this performance? If you are not sure, don't worry, we will discuss it latter.

### 5.9.2 Example: Bad Initial Estimate

Now let's lets 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 1000m. Can the filter recover from a 1000m initial error?

```
In [26]: sensor_error = 30
         movement_error = 2
         pos = (1000,500)

         dog = DogSensor(0, velocity=movement, noise=sensor_error)

         zs = []
         ps = []

         for i in range(100):
             pos = update(pos[0], pos[1], movement, movement_error)

             Z = dog.sense()
             zs.append(Z)

             pos = sense(pos[0], pos[1], Z, sensor_error)
             ps.append(pos[0])


         p1, = plt.plot(zs,c='r', linestyle='dashed')
```

```
p2, = plt.plot(ps, c='b')
plt.legend([p1,p2], ['measurement', 'filter'], 2)
plt.show()
```



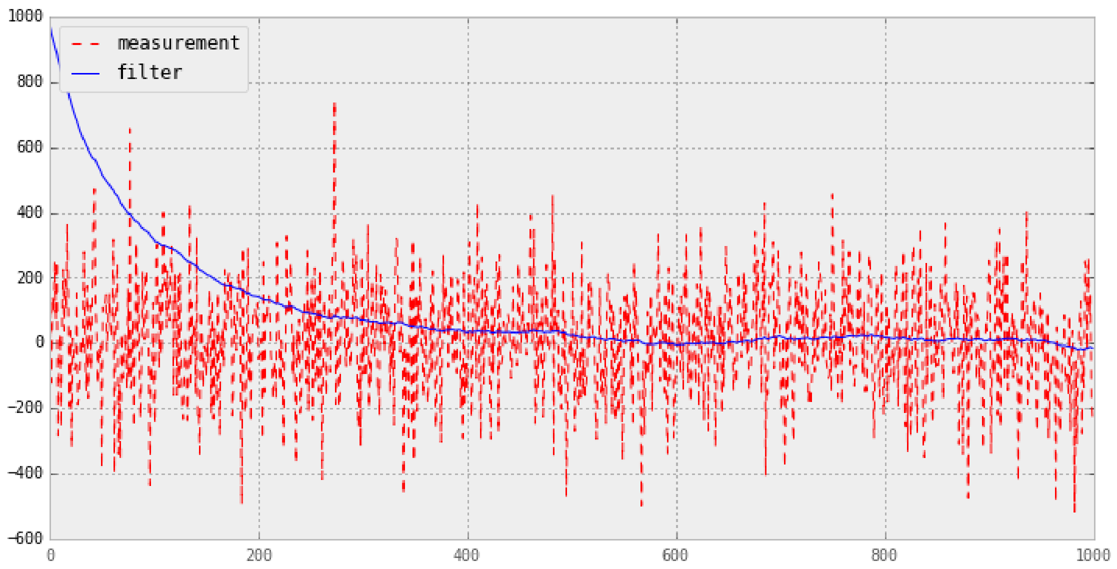
Again the answer is yes! Because we are relatively sure about our belief in the sensor ($\sigma = 30$) even after the first step we have changed our belief in the first position from 1000 to somewhere around 60.0 or so. After another 5-10 measurements we have converged to the correct value! So this is how we get around the chicken and egg problem of initial guesses. In practice we would probably just 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 very quickly.

### 5.9.3 Example: Large Noise and Bad Initial Estimate

What about the worst of both worlds, large noise and a bad initial estimate?

```
In [27]: sensor_error = 30000
         movement_error = 2
         pos = (1000,500)


         dog = DogSensor(0, velocity=movement, noise=sensor_error)
         zs = []
         ps = []

         for i in range(1000):
             pos = update (pos[0], pos[1], movement, movement_error)

             Z = dog.sense()
             zs.append(Z)

             pos = sense (pos[0], pos[1], Z, sensor_error)
             ps.append(pos[0])
```

```
p1, = plt.plot(zs,c='r', linestyle='dashed')
p2, = plt.plot(ps, c='b')
plt.legend([p1,p2], ['measurement', 'filter'], 2)
plt.show()
```

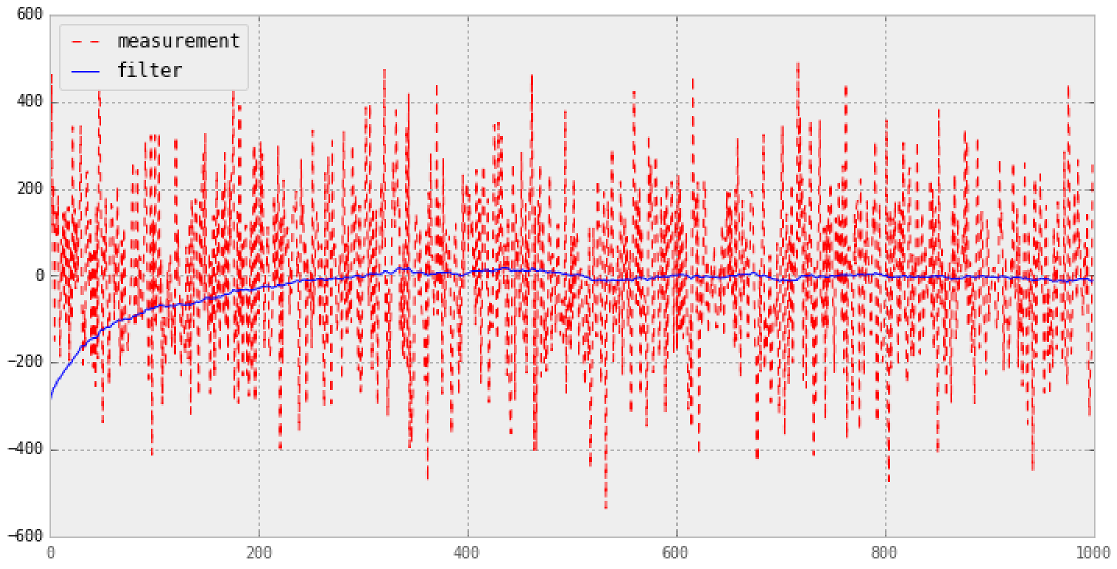


This time the filter does struggle. Notice that the previous example only computed 100 updates, whereas this example uses 1000. By my eye it takes the filter 400 or so iterations to become reasonable accurate, but maybe over 600 before the results are good. Kalman filters are good, but we cannot expect miracles. If we have extremely noisy data and extremely bad initial conditions, this is as good as it gets.

Finally, let's make the suggest change of making our initial position guess just be the first sensor measurement.

```
In [28]: sensor_error = 30000
         movement_error = 2
         pos = None

         dog = DogSensor(0, velocity=movement, noise=sensor_error)

         zs = []
         ps = []

         for i in range(1000):
             Z = dog.sense()
             zs.append(Z)
             if pos == None:
                 pos = (Z, 500)

             pos = sense (pos[0], pos[1], Z, sensor_error)
             ps.append(pos[0])

             pos = update (pos[0], pos[1], movement, movement_error)

         p1, = plt.plot(zs,c='r', linestyle='dashed')
```

```
p2, = plt.plot(ps, c='b')
plt.legend([p1,p2], ['measurement', 'filter'], 2)
plt.show()
```



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 whether the initial measurement $Z$ had a small amount or large amount of noise.

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 rangefinders, GPS satellites, computer vision, and so on. None have the enormous error as shown here. A reasonable value for the variance for a cheap thermometer might be 10, for example, and our code is using 30,000 for the variance.

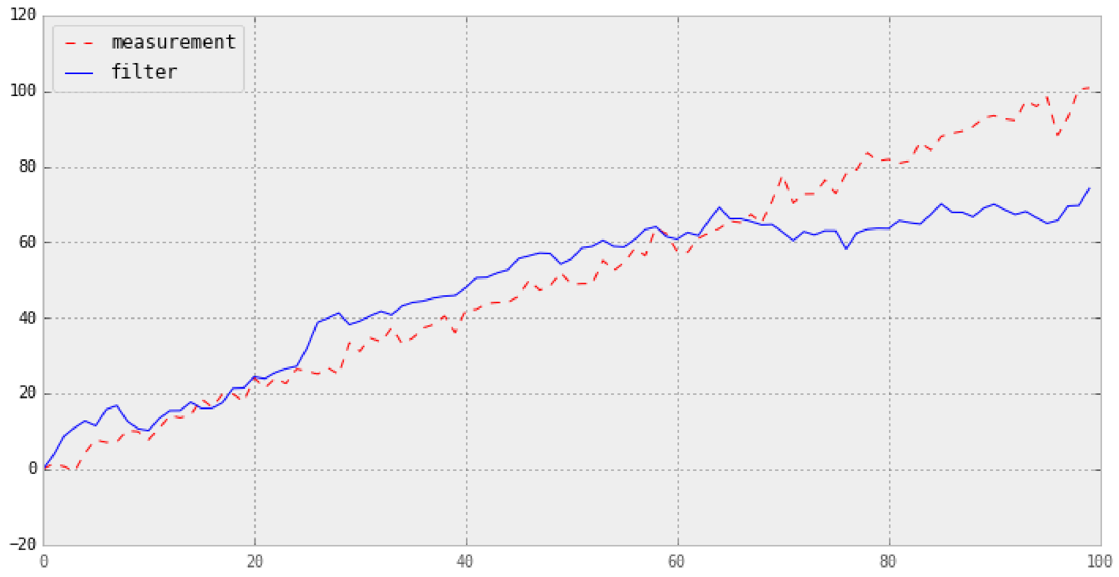
### 5.9.4 Exercise: Interactive Plots

Implement the Kalman filter using IPython 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 Gaussian 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 built the boilerplate for this; just fill in the required code.

```
In [29]: from IPython.html.widgets import interact, interactive, fixed
         import IPython.html.widgets as widgets
         def plot_kalman_filter(start_pos, sensor_noise, movement, movement_noise, noise_scale):
             # your code goes here
             pass

         interact(plot_kalman_filter,
                  start_pos=(-10,10),
                  sensor_noise=widgets.IntSliderWidget(value=5,min=0,max=100),
                  movement=widgets.FloatSliderWidget(value=1,min=-2.,max=2.),
                  movement_noise=widgets.FloatSliderWidget(value=5,min=0,max=100.),
                  noise_scale=widgets.FloatSliderWidget(value=1,min=0,max=2.))

Out[29]: <function __main__.plot_kalman_filter>
```

Solution
One possible solution follows.

```
In [30]: zs = np.zeros(100)
         ps = np.zeros(100)
         def plot_kalman_filter(start_pos, sensor_noise, movement, movement_noise,noise_scale):
             dog = DogSensor(start_pos, velocity=movement, noise=sensor_noise)
             random.seed(303)
             pos = (0,100)

             for i in range(100):
                 Z = dog.sense() + random.randn()*noise_scale
                 zs[i] = Z

                 pos = sense(pos[0], pos[1], Z, sensor_error)
                 ps[i] = pos[0]

                 pos = update(pos[0], pos[1], movement + random.randn()*movement_noise, movement_noise)

             p1, = plt.plot(zs,c='r', linestyle='dashed')
             p2, = plt.plot(ps, c='b')
             plt.legend([p1,p2], ['measurement', 'filter'], 2)
             plt.show()

         interact(plot_kalman_filter,
                  start_pos=(-10,10),
                  sensor_noise=widgets.IntSliderWidget(value=5,min=0,max=100),
                  movement=widgets.FloatSliderWidget(value=1,min=-2.,max=2.),
                  movement_noise=widgets.FloatSliderWidget(value=2,min=0,max=100.),
                  noise_scale=widgets.FloatSliderWidget(value=1,min=0,max=20.))
```



```
Out[30]: <function __main__.plot_kalman_filter>
```

### 5.9.5 Exercise - Nonlinear Systems

Our equations are linear:

$$new\_pos = old\_pos + dist\_moved$$

$$new\_position = old\_position * measurement$$

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):

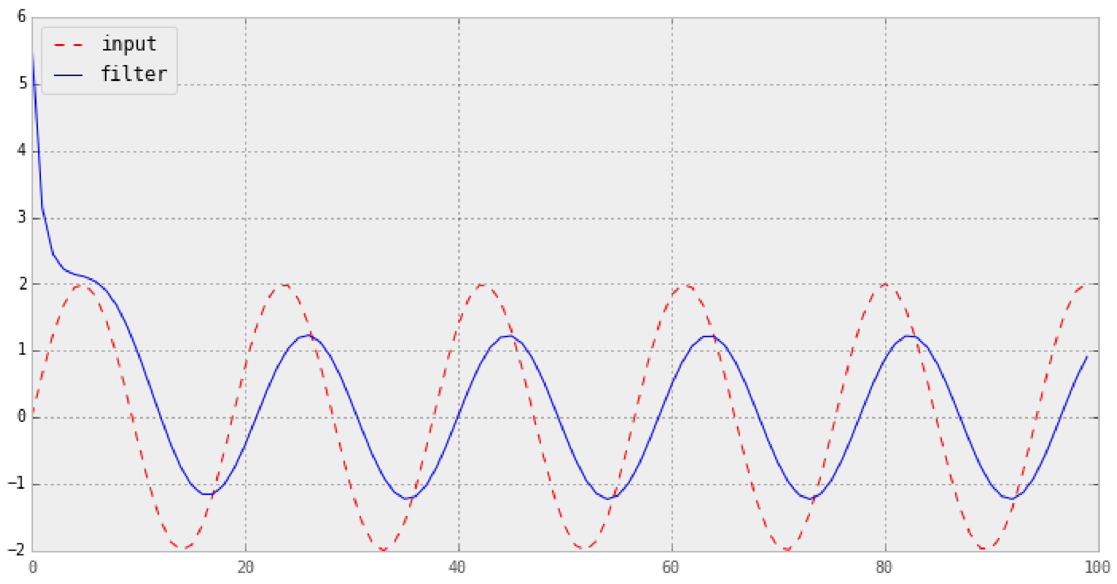
```
Z = math.sin(i/3.) * 2
```

Adust the variance and initial positions to see the effect. What is, for example, the result of a very bad initial guess?

```
In [31]: #enter your code here.
```

Solution:

```
In [32]: sensor_error = 30
         movement_error = 2
         pos = (100,500)

         zs = []
         ps = []


         for i in range(100):
             pos = update(pos[0], pos[1], movement, movement_error)

             Z = math.sin(i/3.)*2
             zs.append(Z)

             pos = sense(pos[0], pos[1], Z, sensor_error)
             ps.append(pos[0])


         p1, = plt.plot(zs,c='r', linestyle='dashed')
         p2, = plt.plot(ps, c='b')
         plt.legend([p1,p2], ['input', 'filter'], 2)
         plt.show()
```



Discussion

Here we set a bad initial guess of 100. We can see that the filter never 'acquires' the signal. Note now the peak of the filter output always lags the peak of the signal by a small amount, and how the filtered signal does not come very close to capturing the high and low peaks of the input signal.

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 predition 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.

Maybe we just didn't adjust things 'quite right'. After all, the output looks like a sin wave, it is just offset some. Let's test this assumption.

### 5.9.6 Exercise - Noisy Nonlinear Systems

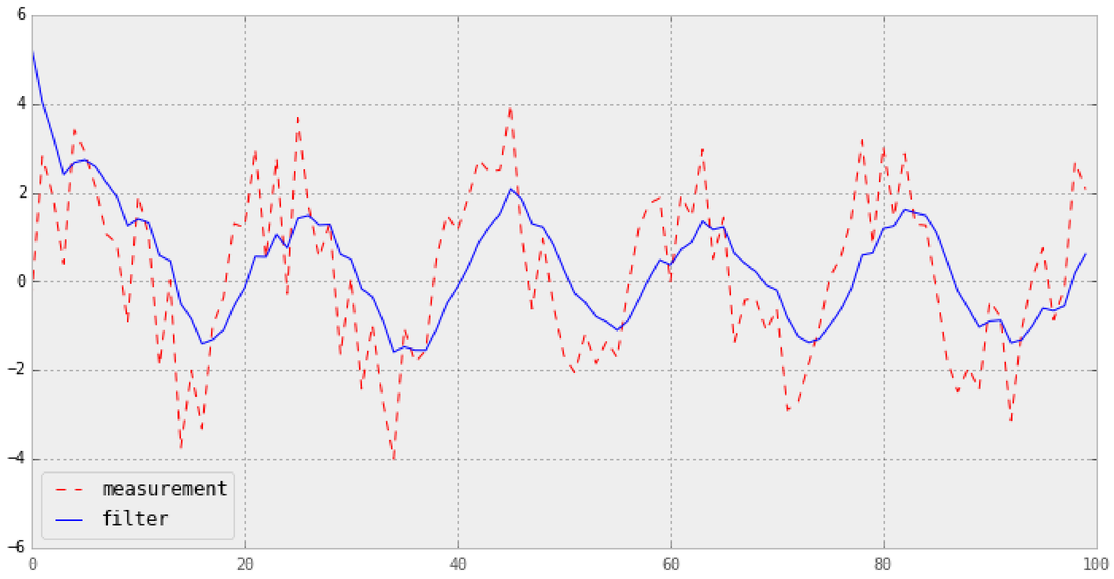
Implement the same system, but add noise to the measurement.

```
In [33]: #enter your code here
```

Solution

```
In [34]: sensor_error = 30
         movement_error = 2
         pos = (100,500)

         zs = []
         ps = []


         for i in range(100):
             pos = update(pos[0], pos[1], movement, movement_error)

             Z = math.sin(i/3.)*2 + random.randn()*1.2
             zs.append(Z)
```

```
        pos = sense(pos[0], pos[1], Z, sensor_error)
        ps.append(pos[0])


    p1, = plt.plot(zs,c='r', linestyle='dashed')
    p2, = plt.plot(ps, c='b')
    plt.legend([p1,p2], ['measurement', 'filter'], 3)
    plt.show()
```



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.

Very shortly after practioners began implementing Kalman filters they recognized the poor performance of them for nonlinear systems and began devising ways of dealing with it. Much of this book is devoted to this problem and its various solutions.

## 5.10 Summary

This information in this chapter takes some time to assimulate. To truly understand this you will probably have to work through this chapter several times. I encourage you to change the various constants and observe the results. Convince yourself that Gaussians are a good representation of a unimodal belief of something like the position of a dog in a hallway. 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 correctly updates your belief. That is all the Kalman filter does. Even now I alternate between complacency and amazement at the results.

If you understand this, you will be able to understand multidimensional Kalman filters and the various extensions that have been make on them. If you do not fully understand this, I strongly suggest rereading this chapter. Try implementing the filter from scratch, just by looking at the equations and reading the text. Change the constants. Maybe try to implement a different tracking problem, like tracking stock prices. Experimentation will build your intuition and understanding of how these marvelous filters work.

```
In [34]:
```

author notes: clean up the code - same stuff duplicated over and over - write a ‘clean implemntation’ at the end.

Chapter 6

# Multidimensional Kalman Filters

```
In [1]: #format the book
        %matplotlib inline
        from __future__ import division, print_function
        import matplotlib.pyplot as plt
        import book_format
        book_format.load_style()

Out[1]: <IPython.core.display.HTML at 0x1353950>
```

## 6.1 Introduction

The techniques in the last chapter are very powerful, but they only work in one dimension. The gaussians represent a mean and variance that are scalars - real numbers. They provide no way to represent multidimensional data, such as the position of a dog in a field. You may retort that you could use two Kalman filters for that case, one tracks the x coordinate and the other tracks the y coordinate. That does work in some cases, but put that thought aside, because soon you will see some enormous benefits to implementing the multidimensional case.

## 6.2 Multivariate Normal Distributions

What might a *multivariate normal distribution* look like? In this context, multivariate just means multiple variables. Our goal is to be able to represent a normal distribution across multiple dimensions. Consider the 2 dimensional case. Let's say we believe that $x = 2$ and $y = 7$. Therefore we can see that for $N$ dimensions, we need $N$ means, like so:

$$\mu = \begin{bmatrix}\mu_1\\ \mu_2\\ \vdots \\ \mu_n\end{bmatrix}$$

Therefore for this example we would have

$$\mu = \begin{bmatrix}2\\ 7\end{bmatrix}$$

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 8, like so.

$$\sigma^2 = \begin{bmatrix}10\\ 8\end{bmatrix}$$

While this is possible, it does not consider the more general case. For example, suppose we were tracking house prices vs total $m^2$ of the floor plan. These numbers are *correlated*. It is not an exact correlation, but in general houses in the same neighborhood are more expensive if they have a larger floor plan. We want a way to express not only what we think the variance is in the price and the $m^2$, but also the degree to which they are correlated. It turns out that we use the following matrix to denote *covariances* with multivariate normal distributions. You might guess, correctly, that *covariance* is short for *correlated variances*.

$$\Sigma = \begin{pmatrix} \sigma_1^2 & p\sigma_1\sigma_2 & \cdots & p\sigma_1\sigma_n \\ p\sigma_2\sigma_1 & \sigma_2^2 & \cdots & p\sigma_2\sigma_n \\ \vdots & \vdots & \ddots & \vdots \\ p\sigma_n\sigma_1 & p\sigma_n\sigma_2 & \cdots & \sigma_n^2 \end{pmatrix}$$

If you haven't seen this before it is probably a bit confusing at the moment. Rather than explain the math right now, we will take our usual tactic of building our intuition first with various physical models. At this point, note that the diagonal contains the variance for each state variable, and that all off-diagonal elements are a product of the $\sigma$ corresponding to the $i$th (row) and $j$th (column) state variable multiplied by a constant $p$.

Now, without explanation, here is the full equation for the multivarate normal distribution in $n$ dimensions.

$$\mathcal{N}(\mu,\,\Sigma) = (2\pi)^{-\frac{n}{2}}|\Sigma|^{-\frac{1}{2}}\,e^{-\frac{1}{2}(\mathbf{x}-\mu)'\Sigma^{-1}(\mathbf{x}-\mu)}$$

I urge you to not try to remember this function. We will program it in a Python function and then call it when we need to compute a specific value. However, if you look at it briefly you will note that it looks quite similar to the *univarate normal distribution* except it uses matrices instead of scalar values, and the root of $\pi$ is scaled by $n$. Here is the *univariate* equation for reference:

$$f(x,\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}(x-\mu)^2/\sigma^2}$$

If you are reasonably well-versed in linear algebra this equation should look quite managable; if not, don't worry! If you want to learn the math we will cover it in detail in the next optional chapter. If you choose to skip that chapter the rest of this book should still be managable for you

I have programmed it and saved it in the file *stats.py* with the function name *multivariate_gaussian*. I am not showing the code here because I have taken advantage of the linear algebra solving apparatus of numpy to efficiently compute a solution - the code does not correspond to the equation in a one to one manner. If you wish to view the code, I urge you to either load it in an editor, or load it into this worksheet by putting `%load -s multivariate_gaussian stats.py` in the next cell and executing it with ctrl-enter.

> As of version 0.14 scipy.stats has implemented the multivariate normal equation with the function `multivariate_normal()`. It is superior to my function in several ways. First, it is implemented in Fortran, and is therefore faster than mine. Second, it implements a 'frozen' form where you set the mean and covariance once, and then calculate the probability for any number of values for x over any arbitrary number of calls. This is much more efficient then recomputing everything in each call. So, if you have version 0.14 or later you may want to substitute my function for the built in version. Use `scipy.version.version` to get the version number. I deliberately named my function `multivariate_gaussian()` to ensure it is never confused with the built in version.
>
> If you intend to use Python for Kalman filters, you will want to read the tutorial for the `scipy.stats` module, which explains 'freezing' distributions and other very useful features. As of this date, it includes an example of using the multivariate_normal function, which does work a bit differently from my function.

```
In [2]: from stats import gaussian, multivariate_gaussian
```

Let's use it to compute a few values just to make sure we know how to call and use the function, and then move on to more interesting things.

First, let's find the probability 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 10 for $y$. This function requires us to pass everything in as numpy arrays (we will soon provide a more robust version that works with numpy matrices, numpy arrays, and/or scalars in any combinations. That code contains a lot of boilerplate which obscures the algorithm).

Start by setting $x$ to (2.5,7.3):

```
In [3]: import numpy as np
        x = np.array([2.5, 7.3])
```

Next, we set the mean of our belief:

```
In [4]: mu = np.array([2,7])
```

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. If there is no correlation between the values you just fill in the diagonal of the covariance matrix with the variances. I will use the seemingly arbitrary name **P** for the covariance matrix. The Kalman filters use the name **P** for this matrix, so I will introduce the terminology now to avoid explaining why I change the name later.

```
In [5]: P = np.array([[8.,0],[0,10.]])
```

Now just call the function

```
In [6]: print(multivariate_gaussian(x,mu,P))
```

```
0.0174395374407
```

Let's check the probability for the dog being at exactly (2,7)

```
In [7]: from __future__ import print_function

        x = np.array([2,7])
        print("Probability dog is at (2,7) is %.2f%%" % (multivariate_gaussian(x,mu,P) * 100.))
```

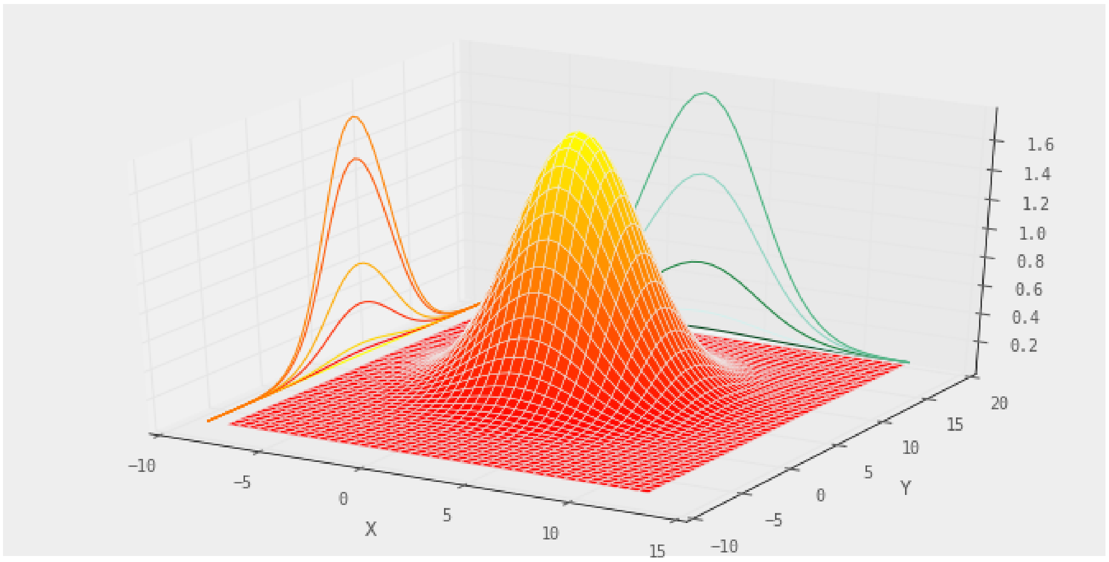
```
Probability dog is at (2,7) is 1.78%
```

These numbers are not easy to interpret. Let's plot this in 3D, with the $z$ (up) coordinate being the probability.

```
In [8]: %matplotlib inline
        import matplotlib.pylab as pylab
        from matplotlib import cm
        from mpl_toolkits.mplot3d import Axes3D
        import numpy as np

        pylab.rcParams['figure.figsize'] = 12,6
        pylab.rcParams['axes.color_cycle'] = '348ABD, 7A68A6, A60628, 467821, CF4457, 188487, E24A33'

        P = np.array([[8.,0],[0,10.]])
        mu = np.array([2,7])

        xs, ys = np.arange(-8, 13, .5), np.arange(-8, 20, .5)
        xv, yv = np.meshgrid (xs, ys)

        zs = np.array([100.* multivariate_gaussian(np.array([x,y]),mu,P) \
                       for x,y in zip(np.ravel(xv), np.ravel(yv))])
```

```
zv = zs.reshape(xv.shape)

ax = plt.figure().add_subplot(111, projection='3d')
ax.plot_surface(xv, yv, zv, rstride=1, cstride=1, cmap=cm.autumn)

ax.set_xlabel('X')
ax.set_ylabel('Y')

ax.contour(xv, yv, zv, zdir='x', offset=-9, cmap=cm.autumn)
ax.contour(xv, yv, zv, zdir='y', offset=20, cmap=cm.BuGn)
plt.xlim((-10,15))
plt.ylim((-10,20))
```

```
Out[8]: (-10, 20)
```

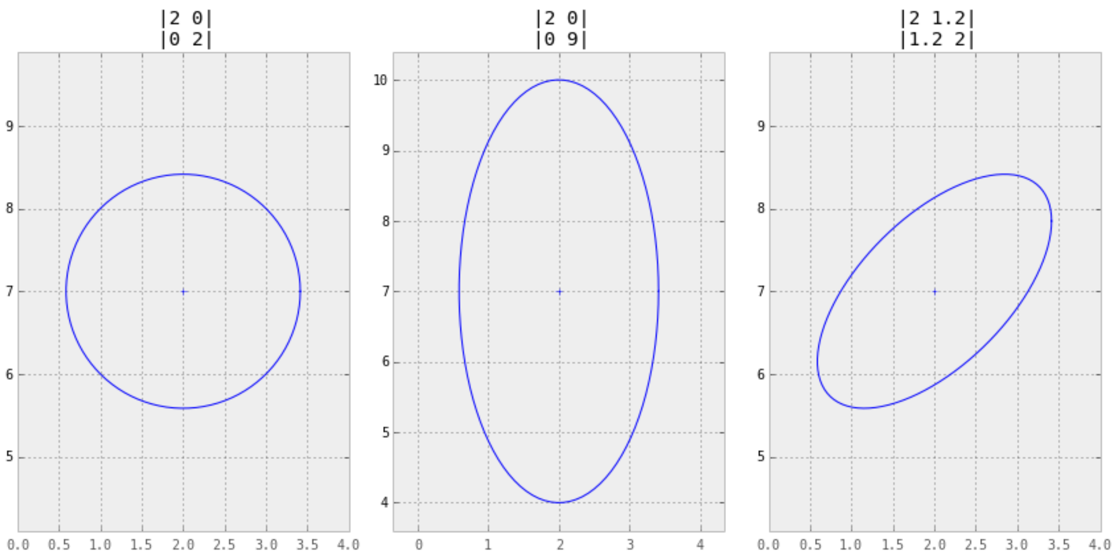


The result is clearly a 3D bell shaped curve. We can see that the gaussian is centered around (2,7), and that the probability quickly drops away in all directions. On the sides of the plot I have drawn the Gaussians for $x$ in greens and for $y$ in orange.

As beautiful as this is, it is perhaps a bit hard to get useful information. For example, it is not easy to tell if $x$ and $y$ both have the same variance or not. So for most of the rest of this book we will display multidimensional Gaussian using contour plots. I will use some helper functions in `gaussian.py` to plot them. If you are interested in linear algebra go ahead and look at the code used to produce these contours, otherwise feel free to ignore it.

```
In [9]: import stats

        P = np.array([[2,0],[0,2]])
        plt.subplot(131)
        stats.plot_covariance_ellipse(P, x=2, y=7, title='|2 0|\n|0 2|')

        plt.subplot(132)
        P = np.array([[2,0],[0,9]])
        stats.plot_covariance_ellipse(P, x=2, y=7, title='|2 0|\n|0 9|')
```

```
plt.subplot(133)
P = np.array([[2,1.2],[1.2,2]])
stats.plot_covariance_ellipse(P, x=2, y=7, title='|2 1.2|\n|1.2 2|')

plt.tight_layout()
plt.show()
```



From a mathematical perspective these display the values that the multivariate gaussian takes for a specific sigma (in this case $\sigma^2 = 1$. Think of it as taking a horizontal slice through the 3D surface plot we did above. However, thinking about the physical interpretation of these plots clarifies their meaning.

The first plot uses the mean and covariance matrices of

$$\mu = \begin{bmatrix}2\\7\end{bmatrix}$$
$$\sigma^2 = \begin{bmatrix}2 & 0\\0 & 2\end{bmatrix}$$

Let this be our current belief about the position of our dog in a field. In other words, we believe that he is positioned at (2,7) with a variance of $\sigma^2 = 2$ for both x and y. The contour plot shows where we believe the dog is located with the '+' in the center of the ellipse. The ellipse shows the boundary for the $1\sigma^2$ probability - points where the dog is quite likely to be based on our current knowledge. Of course, the dog might be very far from this point, as Gaussians allow the mean to be any value. For example, the dog could be at (3234.76,189989.62), but that has vanishing low probability of being true. Generally speaking displaying the $1\sigma^2$ to $2\sigma^2$ contour captures the most likely values for the distribution. An equivelent way of thinking about this is the circle/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. We will use this throughout the rest of the book to display and evaluate the accuracy of our filters at any point in time.

The second plot uses the mean and covariance matrices of

$$\mu = \begin{bmatrix}2\\7\end{bmatrix}$$
$$\sigma^2 = \begin{bmatrix}2 & 0\\0 & 9\end{bmatrix}$$

This time we use a different variance for $x$ (2) vs $y$ (9). The result is an ellipse. When we look at it we can immediately tell that we have a lot more uncertainty in the $y$ value vs the $x$ value. Our belief that the value is (2,7) is the same in both cases, but errors are different. This sort of thing happens naturally as we track objects in the world - one sensor has a better view of the object, or is closer, than another sensor, and so we end up with different error rates in the different axis.

The third plot uses the mean and covariance matrices of:

$$\mu = \begin{bmatrix}2\\7\end{bmatrix}$$

$$\sigma^2 = \begin{bmatrix}2 & 1.2\\1.2 & 2\end{bmatrix}$$

This is the first contour that has values in the off-diagonal elements of $cov$, and this is the first contour plot with a slanted ellipse. This is not a coincidence. The two facts are telling use the same thing. A slanted ellipse tells us that the $x$ and $y$ values are somehow **correlated**. We denote that in the covariance matrix with values off the diagonal. What does this mean in physical terms? Think of trying to park your car in a parking spot. You can not pull up beside the spot and then move sideways into the space because most cars cannot go purely sideways. $x$ and $y$ are not independent. This is a consequence of the steering system in a car. When your tires are turned the car rotates around its rear axle while moving forward. Or think of a horse attached to a pivoting exercise bar in a corral. The horse can only walk in circles, he cannot vary $x$ and $y$ independently, which means he cannot walk straight forward to to the side. If $x$ changes, $y$ must also change in a defined way.

So when we see this ellipse we know that $x$ and $y$ are correlated, and that the correlation is "strong". The size of the ellipse shows how much error we have in each axis, and the slant shows how strongly correlated the values are.

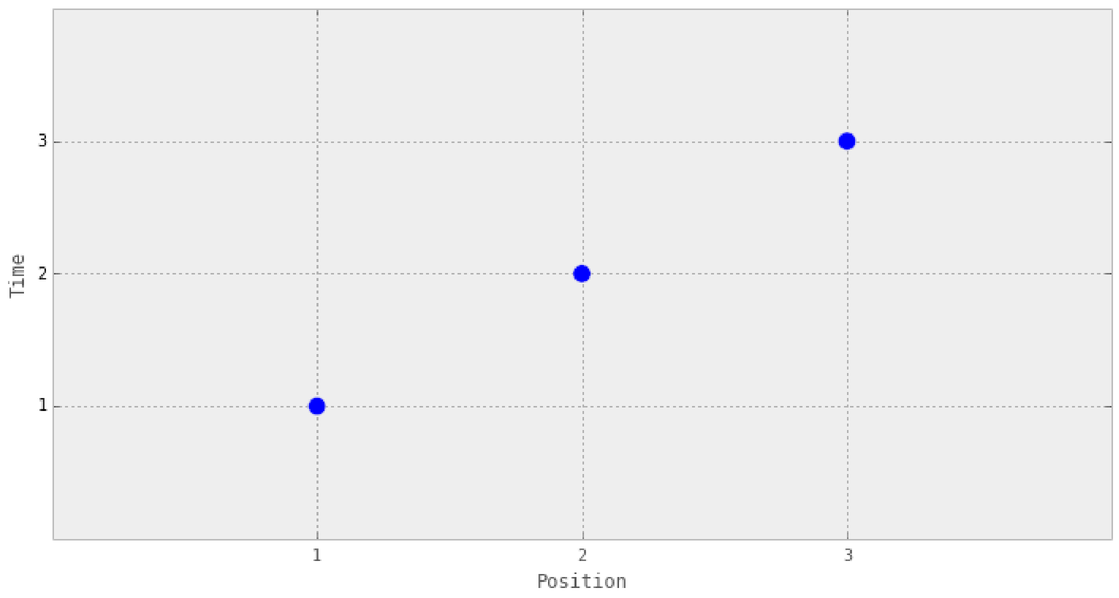
A word about **correlation** and **independence**. If variables are **independent** they can vary separately. If you walk in an open field, you can move in the $x$ direction (east-west), the $y$ direction(north-south), or any combination thereof. Independent variables are always also **uncorrelated**. Except in special cases, the reverse does not hold true. Variables can be uncorrelated, but dependent. For example, consider the pair$(x, y)$ where $y = x^2$. Correlation is a linear measurement, so $x$ and $y$ are uncorrelated. However, they are obviously dependent on each other.

** wikipedia article 'correlation and dependence' claims multivariate normals are a special case, where the correlation coeff $p$ completely defines the dependence. FIGURE THIS OUT!**

## 6.3 Unobserved Variables

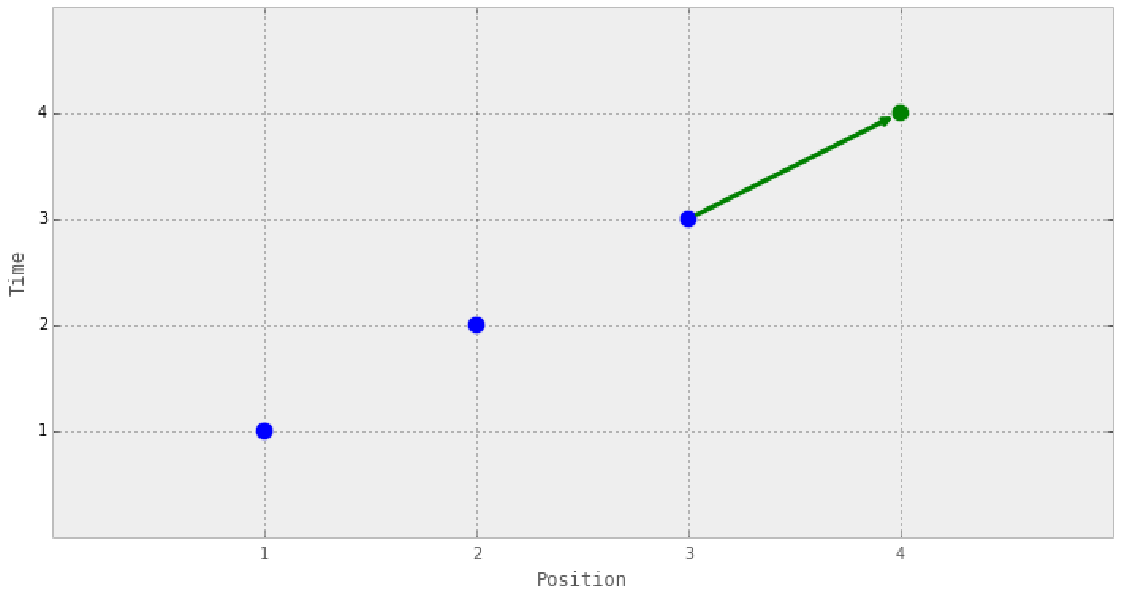
Let's say we are tracking an aircraft and we get the following data for the $x$ coordinate at time $t$=1,2, and 3 seconds. What does your intuition tell you the value of $x$ will be at time $t$=4 seconds?

```
In [10]: import mkf_internal
         mkf_internal.show_position_chart()
```



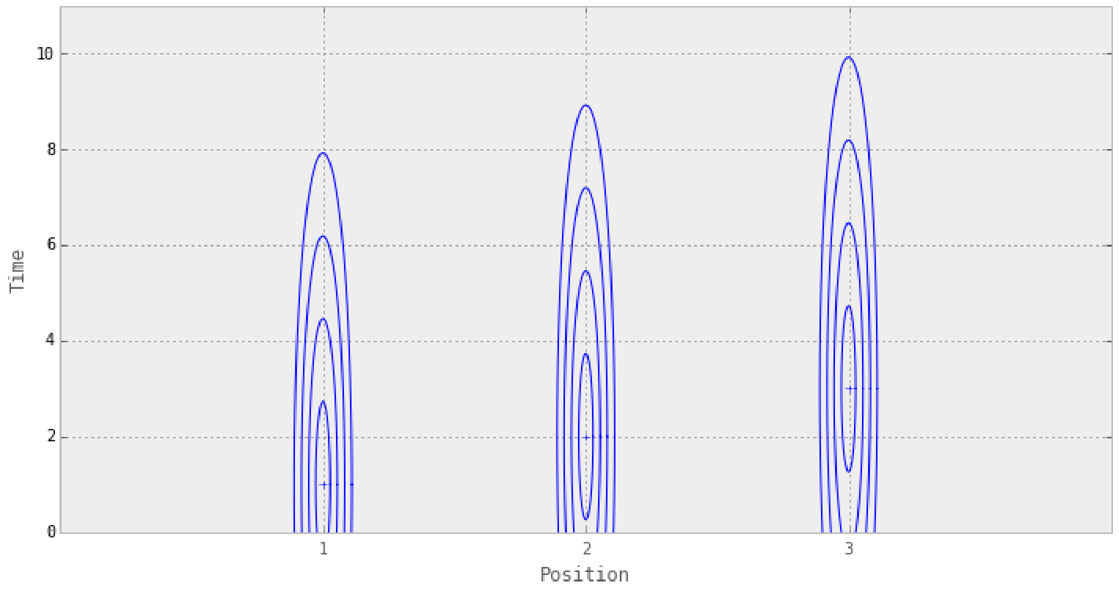
It appears that the aircraft is flying in a straight line because we can draw a line between the three points, and we know that aircraft cannot turn on a dime. The most reasonable guess is that $x$=4 at $t$=4. I will depict that with a green arrow.

```
In [11]: mkf_internal.show_position_prediction_chart()
```



If this is data from a Kalman filter, then each point has both a mean and variance. Let's try to show that by showing the approximate error for each point. Don't worry about why I am using a covariance matrix to depict the variance at this point, it will become clear in a few paragraphs. The intent at this point is to show that while we have $x$=1,2,3 that there is a lot of error associated with each measurement.

```
In [12]: mkf_internal.show_x_error_char()
```

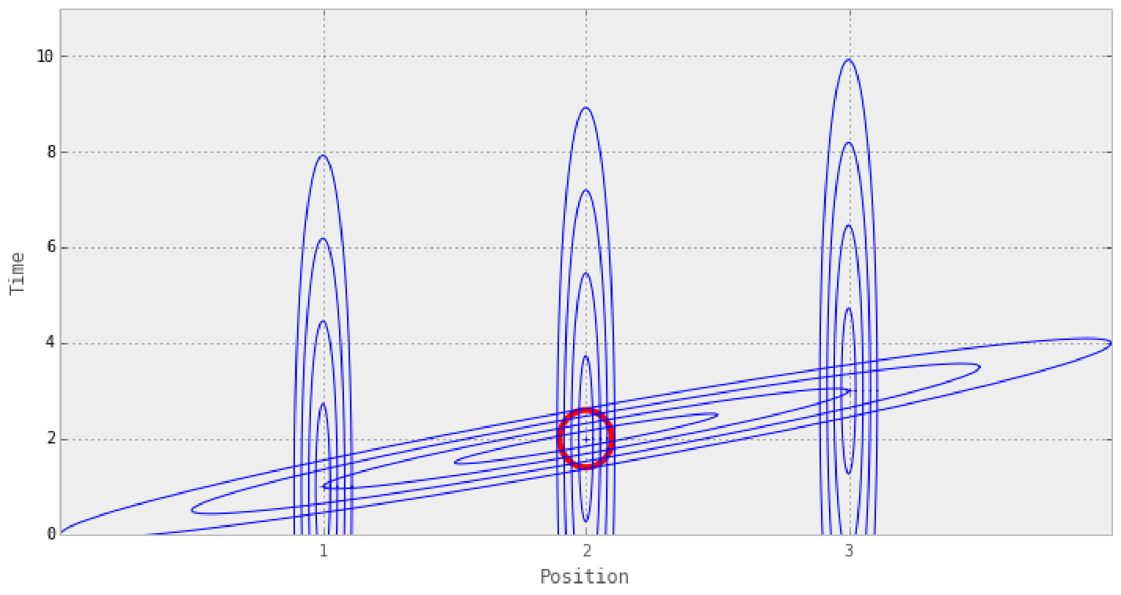


We can see that there is a lot of error associated with each value of $x$. We could write a 1D Kalman filter as we did in the last chapter, but suppose this is the output of that filter, and not just raw sensor measurements. Are we out of luck?

Let us think about how we predicted that $x$=4 at $t$=4. In one sense we just drew a straight line between the points and saw where it lay at $t$=4. My constant refrain: what is the physical interpretation of that? What is the difference in $x$ over time? In other words, what is $\frac{\partial x}{\partial t}$? The derivative, or difference in distance over time is *velocity*.

This is the **key point** in Kalman filters, so read carefully! Our sensor is only detecting the position of the aircraft (how doesn't matter). It does not have any kind of sensor that provides velocity to us. But based on the position estimates we can compute velocity. In Kalman filters we would call the velocity an *unobserved variable*. Unobserved means what it sounds like - there is no sensor that is measuring velocity directly. Since the velocity is based on the position, and the position has error, the velocity will have error as well. What happens if we draw the velocity errors over the positions errors?

```
In [13]: mkf_internal.show_x_with_unobserved()
```



Think about what this plot means. We have a lot of error in our position estimates. We therefore have a lot of error in our velocity estimates. But look at the intersections between the velocity and the positions. Take the intersection at $t$=2. The intersection between the velocity and the position is where our aircraft is most likely to be, which I have roughly depicted with a red ellipse ('roughly' because I set the size via eyeball, not via math). The size of the error is much smaller than the error of the positions, despite the fact that velocity was derived from position.

What makes this possible? Imagine for a moment that we superimposed the velocity from a *different* airplane over the position graph. Cleary 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 the 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 position measurement is not in the direction of the assumed 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.

## 6.4 Kalman Filter Algorithm

So in general terms we can show how a multidimensional Kalman filter works. In the example above, we compute velocity from the previous position measurements using something called the *measurement function*. Then we predict the next position by using the current estimate and something called the *state transition function*. In our example above,

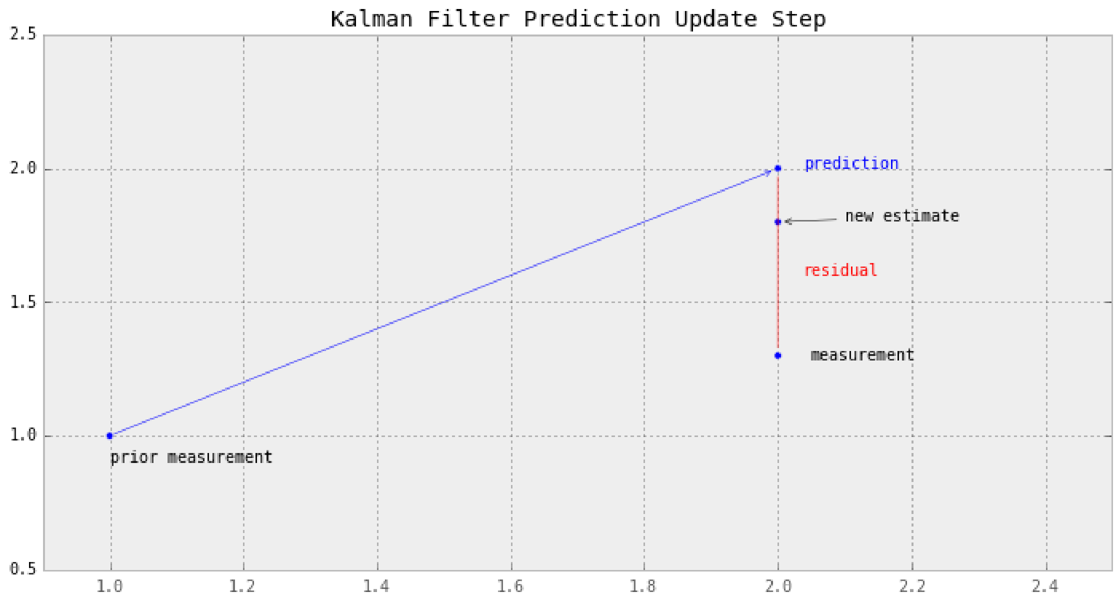
$$new\_position = old\_position + velocity * time$$

Next, we take the measurement from the sensor, and compare it to the prediction we just made. In a world with perfect sensors and perfect airplanes the prediction will always match the measured value. In the real world they will always be at least slightly different. We call the difference between the two the *residual*. Finally, we use something called the *Kalman gain* to update our estimate to be somewhere between the measured position and the predicted position. I will not describe how the gain is set, but suppose we had perfect confidence in our measurement - no error is possible. Then, clearly, we would set the gain so that 100% of the position came from the measurement, and 0% from the prediction. At the other extreme, if he have no confidence at all in the sensor (maybe it reported a hardware fault), we would set the gain so that 100% of the position came from the prediction, and 0% from the measurement. In normal cases, we will take a ratio of the two: maybe 53% of the measurement, and 47% of the prediction. The gain is updated on every cycle based on the variance of the variables (in a way yet to be explained). It should be clear that if the variance of the measurement is low, and the variance of the prediction is high we will favor the measurement, and vice versa.

The chart shows a prior estimate of $x = 1$ and $\dot{x} = 1$ ($\dot{x}$ is the shorthand for the derivative of x, which is velocity). Therefore we predict $\hat{x} = 2$. However, the new measurement $x^{'} = 1.3$, giving a residual $r = 0.7$. Finally, Kalman filter gain $k$ gives us a new estimate of $\hat{x^{'}} = 1.8$.

** CHECK SYMBOLOGY!!!!**

```
In [14]: from mkf_internal import *
         show_residual_chart()
```



## 6.5 The Equations

The brilliance of the Kalman filter is taking the insights of the chapter up to this point and finding an optimal mathematical solution. The Kalman filter finds what is called a *least squared fit* to the set of measurements to produce an optimal output. We will not trouble ourselves with the derivation of these equations. It runs to several pages, and offers a lot less insight than the words above, in my opinion. Furthermore, to create a Kalman filter for your application you will not be manipulating these equations, but only specifing a number of parameters that are used by them. It would be going too far to say that you will never need to understand these equations; but to start we can pass them by and I will present the code that implements them. So, first, let's see the equations. > Kalman Filter Predict Step:

$$\hat{\mathbf{x}}_{t|t-1} = \mathbf{\Phi_t}\hat{\mathbf{x}}_{t-1} + \mathbf{B}\mathbf{u}_t \qquad (1)$$
$$\mathbf{P}_{t|t-1} = \mathbf{\Phi_t}\mathbf{P}_{t-1}\mathbf{\Phi}_t^T + \mathbf{Q}_t \qquad (2)$$

Kalman Filter Update Step:

$$\gamma = \mathbf{z}_t - \mathbf{H}_t\hat{\mathbf{x}}_t \tag{3}$$
$$\mathbf{K}_t = \mathbf{P}_t\mathbf{H}_t^T(\mathbf{H}_t\mathbf{P}_t\mathbf{H}_t^T + \mathbf{R}_t)^{-1} \tag{4}$$

$$\hat{\mathbf{x}}_t = \hat{\mathbf{x}}_{t|t-1} + \mathbf{K}_t\gamma \tag{5}$$
$$\mathbf{P}_{t|t} = (\mathbf{I} - \mathbf{K}_t\mathbf{H}_t)\mathbf{P}_{t|t-1} \tag{6}$$

Dash off, wipe the blood out of your eyes, and we'll disuss what this means.

These are nothing more than linear algebra equations that implement the algorithm we used in the last chapter, but using multidimensional Gaussians instead of univariate Gaussians, and optimized for a least squares fit. Each capital letter denotes a matrix or vector. The subscripts indicate which time step the data comes from; $t$ is now, $t-1$ is the previous step. $A^T$ is the transpose of A, and $A^{-1}$ is the inverse. Finally, the hat denotes an estimate, so $\hat{x}_t$ is the estimate of $x$ at time $t$.

## 6.5.1 Kalman Equations Expressed as an Algorithm

Different texts use different notation and variable names for the Kalman filter. Later we will expose you to these different forms to prepare you for reading the original literature. However, I find much of the notation very dense, and unnecessary for writing code. The subscripts indicate the time step, but we know the left hand side is for this time step, and the right hand side is for the previous step. For most of this book I'm going to use the following simplified equations, which express an algorithm.

Predict Step

$$\mathbf{x}' = \mathbf{F}\mathbf{x} + \mathbf{B}\mathbf{u} \tag{1}$$
$$\mathbf{P} = \mathbf{F}\mathbf{P}\mathbf{F}^T + \mathbf{Q} \tag{2}$$

Update Step

$$\gamma = \mathbf{z} - \mathbf{H}\mathbf{x} \tag{3}$$
$$\mathbf{K} = \mathbf{P}\mathbf{H}^T(\mathbf{H}\mathbf{P}\mathbf{H}^T + \mathbf{R})^{-1} \tag{4}$$
$$\mathbf{x} = \mathbf{x}' + \mathbf{K}\gamma \tag{5}$$
$$\mathbf{P} = (\mathbf{I} - \mathbf{K}\mathbf{H})\mathbf{P} \tag{6}$$

This is an algorithm, so = denotes assignment, not equality. For example, equation (6) has P on both sides of the =. This equation updates the value of P by the computation on the right hand side.

Here, a $'$ means estimate, so $\mathbf{x}'$ is the estimate of the state $\mathbf{x}$. Many texts use $\hat{\mathbf{x}}$ or $\mathbf{x}^*$ to express this. I find these choices unfortunate because we will often want to express these in matrix form. The notation of a hat over a matrix is clumsy at best, and an asterisk followin a matrix normally means the *complex congugate* of the matrix, which is *not* what is intended. So I use $'$.

What do all of the variables mean? What is **P**, for example? Don't worry right now. Instead, I am just going to design a Kalman filter, and introduce the names as we go. Then we will just pass them into Python function that implement the equations above, and we will have our solution. Later sections will then delve into more detail about each step and equation. I think learning by example and practice is far easier than trying to memorize a dozen abstract facts at once.

Look at the code below for the predict step (which we will present a bit later).

```
def predict():
    x = F*x + B*u       # equation (1)
    P = F*P*F.T + Q     # equation (2)
```

Notice how simple it really is. It really isn't much different from the predict step in the previous chapter, and it is a nearly exact transliteration of the equations above. As you become familiar with this notation you will find yourself able to read textbooks and paper and implement the equations without much difficulty.

Later, if you become interested in the details of numerical computation you may change the implementation to be faster or more numerically stable than this written form, but for most of this book our code will follow the Kalman filter equations almost exactly.

## 6.6 Tracking a Dog

Let's go back to our tried and true problem of tracking our dog. This time we will include the fundamental insight of this chapter - that of using *unobserved variables* to improve our estimates. In simple terms, our algorithm is:

```
1. predict the next value for x with "x + vel*time"
2. get measurement for x
3. compute residual as: "x - x_prediction"
4. compute kalman gain based on noise levels
5. compute new position as "residual * kalman gain"
```

That is the entire Kalman filter algorithm. It is both what we described above in words, and it is what the rather obscure Kalman Filter equations do. The Kalman filter equations just express this algorithm by using linear algebra.

As I mentioned above, there is actually very little programming involved in creating a Kalman filter. We will just be defining several matrices and parameters that get passed into the Kalman filter algorithm code. Rather than try to explain each of the steps ahead of time, which can be a bit abstract and hard to follow, let's just do it for our by now well known dog tracking problem. Naturally this one example will not cover every use case of the Kalman filter, but we will learn by starting with a simple problem and then slowly start addressing more complicated situations.

**Step 1: Choose the State Variables** *State variables* are the variables that the Kalman filter estimates. They include the *observed variables* - the data that is directly measured by a sensor, and the *unobserved variables*, which we can infer from the observed variables.

For our dog tracking problem, our observed state variable is position, and the unobserved variable is velocity.

The Kalman filter is implemented using linear algebra. We use an $n \times 1$ matrix 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. The Kalman filter equations use $\mathbf{x}$ for the state, so we define $\mathbf{x}$ as:

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

**Step 2: Design State Transition Function** The next step in designing a Kalman filter is designing what is called the *State Transition Function.* This is a set of equations that mathematically describe the behavior of the system we are filtering. So, for the dog tracking problem we are tracking a moving object, so we just need the Newtonian equations for motion. In other words, these are the equations that we use to predict the next state from the current state.

We know from elementary physics how to compute a new position given a previous position, velocity, and time, like so:

$$x' = velocity * time + x_{previous}$$

In more formal mathematics we would write:

$$x' = \dot{x}(\Delta t) + x$$

where $\dot{x}$ is velocity, and $\Delta t$ is the amount of time between $t-1$ and $t$. In our problems we will be running the Kalman filter at fixed time intervals, so $\Delta t$ is a constant for us. We will just set it to 1 and worry about the units later.

As in step one we must express this in the form of matrices so that our linear algebra software and solve the equations for us. The Kalman filter equations require that we write it in the form:

$$\mathbf{x}' = \mathbf{F}\mathbf{x}$$

where as in step 1 $\mathbf{x}$ is the matrix containing the state variables, and $\mathbf{F}$ is the matrix that when multiplied by $\mathbf{x}$ yields our equations. Note that this is just part of the Kalman filter equation (1) above. We will deal with the second half of equation (1) in the next step.

Since $\mathbf{x}$ is a 2×1 matrix $\mathbf{F}$ must be a 2×2 matrix to yield another 2×1 matrix as a result. The first row of the F is easy to derive:

$$\begin{bmatrix}x\\ \dot{x}\end{bmatrix}' = \begin{bmatrix}1 & \Delta t\\ ? & ?\end{bmatrix} \times \begin{bmatrix}x\\ \dot{x}\end{bmatrix}$$

When we multiply the first row of $\mathbf{F}$ that out we get:

$$x' = 1 \times x + \Delta t * \dot{x}, \text{ or}$$
$$x' = \dot{x}(\Delta t) + x$$

Now we have to account for the second row. I've let it somewhat unstated up to now, but we are assuming constant velocity for this problem. Naturally this assumption is not true; if our dog moves it must accelerate and deaccelerate. If you cast your mind back to the $g - hFilter$ chapter we explored the effect of assuming constant velocity. So long as the acceleration is small compared to $\Delta t$ the filter will still perform well.

Therefore we will assume that

$$\dot{x}' = \dot{x}$$

which gives us the second row of $\mathbf{F}$ as follows, once we set $\Delta t = 1$:

$$\begin{bmatrix}x\\ \dot{x}\end{bmatrix}' = \begin{bmatrix}1 & 1\\ 0 & 1\end{bmatrix} \times \begin{bmatrix}x\\ \dot{x}\end{bmatrix}$$

Which, when multiplied out, yields our desired equations:

$$x' = x + \dot{x}$$
$$\dot{x}' = \dot{x}$$

In the vocabulary of Kalman filters we call this *transforming the state matrix.* We take our state matrix, which for us is $\left(\begin{smallmatrix}x\\ \dot{x}\end{smallmatrix}\right)$,and multipy it by a matrix we will call $F$ to compute the new state. In this case, $F = \left(\begin{smallmatrix}1 & 1\\ 0 & 1\end{smallmatrix}\right)$.

You will do this for every Kalman filter you ever design. Your state matrix will change depending on how many state random variables you have, and then you will create $F$ so that it updates your state based on whatever the physics of your problem dictates. $F$ is always a matrix of constants. If this is not fully clear, don't worry, we will do this many times in this book.

Refer back to the first Kalman filter equation $\hat{\mathbf{x}}_{t|t-1} = \mathbf{F_t}\hat{\mathbf{x}}_{t-1} + \mathbf{B}\mathbf{u}_t$. There is an unexplained $\mathbf{B}\mathbf{u}_t$ term in there, but shorn of all the diacritics it should be clear that we just designed $F$ for this equation!

**Step 3: Design the Motion Function** The Kalman filter does not just filter data, it allows us to incorporate control inputs for systems like robots and airplanes. Consider the state transition function we wrote for the dog:

$$x_t = \dot{x}(\Delta t) + x_{t-1}$$

Suppose that instead of passively tracking our dog we were actively controlling a robot. At each time step we would send control signals to the robot based on our 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.

We will cover this use case later, but for now passive tracking applications we set those terms to 0. In step 2 there was the unexplained term **Bu** in equation (1):

$$\mathbf{x}' = \mathbf{F}\mathbf{x} + \mathbf{B}\mathbf{u}$$

.

Here **u** is the control input, and **B** is its transfer function. For example, **u** might be a voltage controlling how fast the wheel's motor turns, and multiplying by **B** yields $\frac{x}{\dot{x}}$. Since we do not need these terms we will set them both to zero and not concern ourselves with them for now.

**Step 4: Design the Measurement Function** Now we need a way to compute the state variables to our measurements. In our problem we have one sensor for the position, and it outputs position directly. We do not have a sensor for velocity. If we put this in linear algebra terms we get:

$$z = \begin{bmatrix}1 & 0\end{bmatrix} \times \begin{bmatrix}x \\ \dot{x}\end{bmatrix}$$

In other words, the measurement sensor provides one times the sensor's measurement of $x$, and zero times the nonexistent velocity measurement. This is simple, because the problem is simple! A slightly more complicated problem might use a temperature sensor might, for example, output a voltage, and we would need to provide an equation to convert from voltage to temperature.

In the nomenclature of Kalman filters the [10] matrix is called $H$. If you scroll up to the Kalman filter equations you will see an $H$ term in the update step.

$$\gamma = \mathbf{z} - \mathbf{H}\mathbf{x} \tag{3}$$

Believe it or not, we have designed the majority of our Kalman filter!! All that is left is to model the noise in our sensors.

**Step 5: Design the Measurement Noise Matrix** The *measurement noise* is a matrix that models the noise in our sensors as a covariance matrix. This can be admittedly a very difficult thing to do in practice. 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 already read high if the temperature is high, and so the noise is not distributed equally on both sides of the mean. Later we will address this topic in detail. For now I just want you to get used to the idea of the measurement noise matrix so we will keep it deliberately simple.

In the last chapter we used a variance of 5 for our position sensor. Let's use the same value here. The Kalman filter equations uses the symbol $R$ for this matrix.

$$R = 5$$

In general the matrix will have dimension $m \times 1$, where $m$ is the number of sensors. We have only 1 sensor here, so we can get by using a scalar. However, we could equally have written

$$R = \begin{bmatrix}5\end{bmatrix}$$

but I endevour to keep the nomenclature as simple as possible.

**Step 6: Design the Process Noise Matrix** What is *process noise*? Consider the motion of a thrown ball. In a vacuum and with constant gravitational force it moves in a parabola. However, if you throw the ball on the surface of the earth you will also need to model factors like rotation and air drag. However, even when you have done all of that there is usually things you cannot account for. For example, consider wind. On a windy day the ball's trajectory will differ from the computed trajectory, perhaps by a significant amount. Without wind sensors, we may have no way to model the wind. Wind can come from any direction, so it is likely to have a near Gaussian distribution. The Kalman filter models this as *process noise*, and calls it **Q**.

Astute readers will realize that we can inspect the ball's path and extract wind as an unobserved state variable, but the point to grasp here is there will always be some unmodelled noise in our process, and the Kalman filter gives us a way to model it.

Designing the process noise matrix can be quite demanding. For our first example, we will set it to 0, like so: $\mathbf{Q} = 0$. It is unlikely that you would do that for a real filter.

> Some books and papers use **R** for measurement noise and **Q** for the process noise. Others do the opposite, using **Q** for measurement noise and **R** for the process noise! Read carefully, and make sure you don't get confused. I use the following mnemonic. Radars are used to measure positions, and they have measurement error. So, for me, **R** is the **R**adar's measurement noise. I've read a lot of Kalman filter literature in the context of radar tracking, so it makes sense to me. I don't have a good one for **Q**, other than to note that it alphabetically follows the p in **P**rocess.

**Step 7: Design Initial Conditions** Finally, we need to specify the initial conditions for the state variables and their associated covariance matrix. If you have a rough idea of the values you can use that as your initial settings, or, you could always read the sensors for the first time, and calculate a initial value. The Kalman filter will converge and find the solution even if your initial conditions are far off, but the more accurate they are the faster and better the output will be.

The covariance matrix ($\mathbf{P}$) is a $n \times n$ matrix that specifies the variances and covariances of each state variable. This is a complicated topic, and I'd rather demostrate the matrix rather than talk about it abstractly here. For now, recognize that the Kalman filter will be calculating the covariance matrix at each step, just like the 1-D filter computed the variance of the mean at each step. So as with those examples we can make an initial guess and trust that **P** will converge to a smaller value as the filter progresses. I find this description somewhat unsatisfactory for several reasons, but until you've seen some examples it is hard to talk about in an understandable way.

## 6.7 Implementing the Kalman Filter

As promised, the Kalman filter equations are already programmed for you. In many circumstances you will never have to write your own Kalman filter equations. We will look at the code later, but for now we will just import the code and use it. I have placed it in *KalmanFilter.py*, so let's start by importing it and creating a filter.

```
In [15]: import numpy as np
         from KalmanFilter import KalmanFilter
         f = KalmanFilter (dim=2)
```

That's it. We import the filter, and create a filter that uses 2 state variables. We specify the number of state variables with the 'dim=2' expression (dim means dimensions).

The Kalman filter class contains a number of variables that you need to set. x is the state, F is the state transition function, and so on. Rather than talk about it, let's just do it!

```
In [16]: f.x = np.matrix([[0], [0]])    # initial state (location and velocity)
         f.F = np.matrix([[1,1], [0,1]]) # state transition matrix
         f.H = np.matrix([[1,0]])       # Measurement function
         f.R  = 5                       # measurement noise
         f.Q = 0                        # process noise
         f.P *= 500.                    # covariance matrix
```

Let's look at this line by line.

**1**: We just assign the initial value for our state. Here we just initialize both the position and velocity to zero.

**2**: We set $\mathbf{F} = \left(\begin{smallmatrix}1 & 1\\ 0 & 1\end{smallmatrix}\right)$, as in design step 2 above.

**3**: We set $\mathbf{H} = \begin{pmatrix} 1 & 0 \end{pmatrix}$, as in design step 3 above.

**4**: We set $\mathbf{R} = 5$ and $\mathbf{Q} = 0$ as in steps 5 and 6.

**5**: Recall in the last chapter we set our initial belief to $\mathcal{N}(\mu, \sigma^2) = \mathcal{N}(0, 500)$ to signify our lack of knowledge about the initial conditions. We implemented this in Python with a list that contained both $\mu$ and $\sigma^2$ in the variable *pos*:

```
pos = (0,500)
```

Multidimensional Kalman filters stores the state variables in $\mathbf{x}$ and their *covariance* in $\mathbf{P}$. These are `f.x` and `f.P` in the code above. Notionally, this is similar as the one dimension case, but instead of having a mean and variance we have a mean and covariance. For the multidimensional case, we have

$$\mathcal{N}(\mu, \sigma^2) = \mathcal{N}(\mathbf{x}, \mathbf{P})$$

$\mathbf{P}$ is initialized to the identity matrix of size $n{\times}n$, so multiplying by 500 assigns a variance of 500 to $x$ and $\dot{x}$. So `f.P` contains

$$\begin{bmatrix} 500 & 0 \\ 0 & 500 \end{bmatrix}$$

This will become much clearer once we look at the covariance matrix in detail in later sessions. For now recognize that each diagonal element $e_{ii}$ is the variance for the *ith* state variable.

> Summary: For our dog tracking problem, in the 1-D case $\mu$ was the position, and $\sigma^2$ was the variance. In the 2-D case $\mathbf{x}$ is our position and velocity, and $\mathbf{P}$ is the *covariance* of the position and velocity. It is the same thing, just in higher dimensions!

All that is left is to run the code! The `DogSensor` class from the previous chapter has been placed in `DogSensor.py`.

```
In [17]: from DogSensor import DogSensor

         def dog_tracking_filter(R,Q=0,cov=1.):
             f = KalmanFilter (dim=2)
             f.x = np.matrix([[0], [0]])    # initial state (location and velocity)
             f.F = np.matrix([[1,1],[0,1]]) # state transition matrix
             f.H = np.matrix([[1,0]])       # Measurement function
             f.R = R                        # measurement uncertainty
             f.P *= cov                     # covariance matrix
             f.Q = np.eye(2)*Q
             return f


         def filter_dog(noise, count, R, Q=0):
             dog = DogSensor(velocity=1, noise=noise)
             dog_filter = dog_tracking_filter(R=R, Q=Q, cov=500.)

             pos  = [None] * count
             zs  = [None] * count
             cov = [None] * count

             for t in range (count):
                 z = dog.sense()  # get the next measurement

                 pos[t] = dog_filter.x[0,0]
                 cov[t] = dog_filter.P
```

```
            zs[t]  = z

            # perform the kalman filter steps
            dog_filter.update (z)
            dog_filter.predict()

        return (pos, zs, cov)
```

This is the complete code for the filter, and most of it is just boilerplate. The first function `dog_tracking_filter()` is a helper function that creates a `KalmanFilter` object with specified **R**, **Q** and **P** matrices. We've shown this code already, so I will not discuss it more here.

The function `filter_dog()` implements the filter itself. Lets work through it line by line. The first line creates the simulation of the DogSensor, as we have seen in the previous chapter.

```
dog = DogSensor(velocity=1, noise=noise)
```

The next line uses our helper function to create a Kalman filter.

```
dog_filter = dog_tracking_filter(R=R, Q=Q, cov=500.)
```

We will want to plot the filtered position, the measurements, and the covariance, so we will need to store them in lists. The next three lines initialize empty lists of length *count* in a pythonic way.

```
pos = [None] * count
zs  = [None] * count
cov = [None] * count
```

Finally we get to the filter. 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 functions `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 bookkeeping work of storing the filter's data, the for loop reads:

```
for t in range (count):
    z = dog.sense()
    dog_filter.update (z)
    dog_filter.predict()
```

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` variables; executing the filter is trivial.

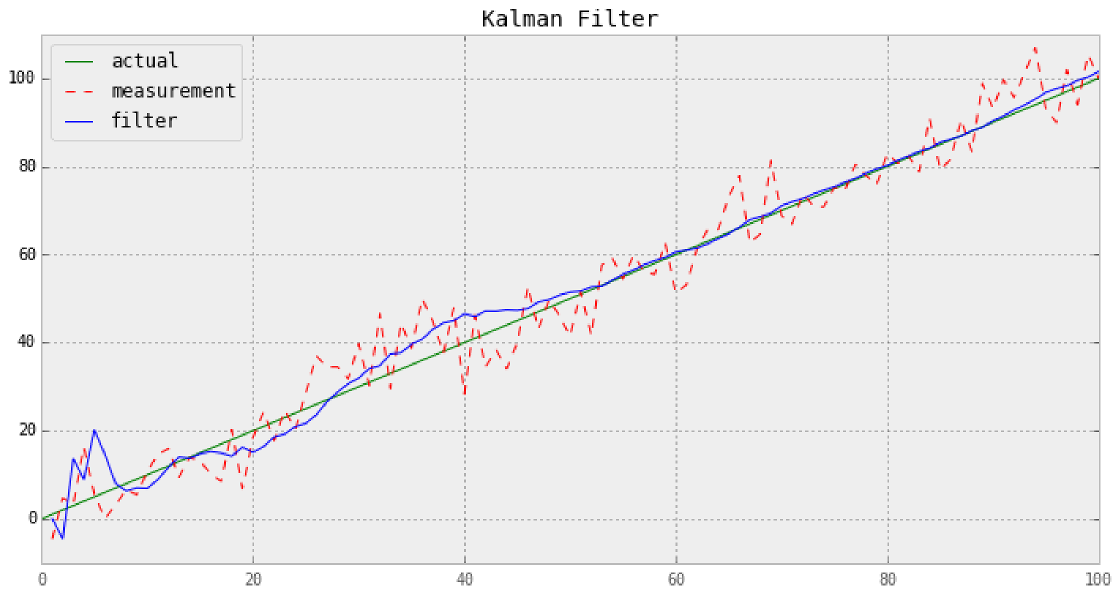
Now let's look at the result. Here is some code that calls `filter_track()` and then plots the result. It is fairly uninteresting code, so I will not walk through it.

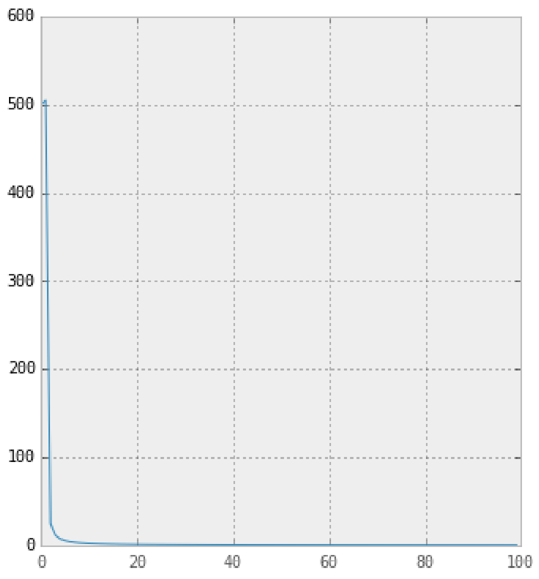
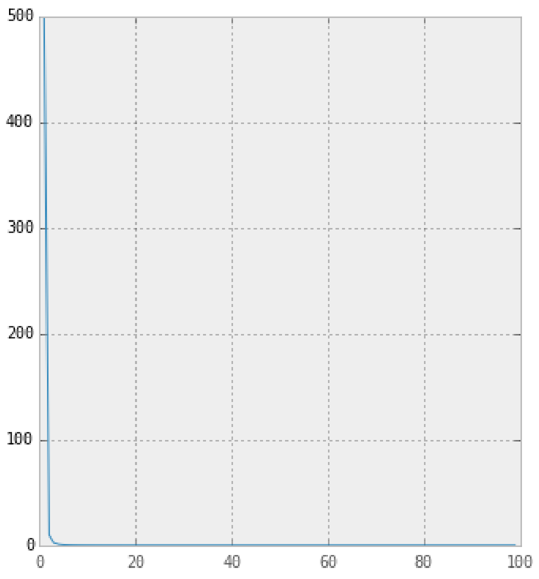
```
In [18]: def plot_track(noise, count, R, Q=0, plot_P=True, title='Kalman Filter'):

             ps, zs, cov = filter_dog(noise, count, R, Q)

             p0, = plt.plot([0,count],[0,count],'g')
             p1, = plt.plot(range(1,count+1),zs,c='r', linestyle='dashed')
             p2, = plt.plot(range(1,count+1),ps, c='b')
             plt.legend([p0,p1,p2], ['actual','measurement', 'filter'], 2)
             plt.ylim((0-10,count+10))
             plt.title(title)

             plt.show()
```

```
        if plot_P:
            plt.subplot(121)
            plot_covariance(cov, (0,0))
            plt.subplot(122)
            plot_covariance(cov, (1,1))
            plt.show()

    def plot_covariance(P, index=(0,0)):
        ps = []
        for p in P:
            ps.append(p[index[0],index[1]])
        plt.plot(ps)
```

Finally, call it. We will start by filtering 100 measurements with a noise factor of 30, $\mathbf{R} = 5$ and $\mathbf{Q} = 0$.

```
In [19]: plot_track (noise=30, R=5, Q=0, count=100)
```



There is still a lot to learn, but we have implemented our first, full Kalman filter using the same theory and equations as published by Nobert Kalman! Code very much like this runs inside of your GPS and phone, inside every airliner, inside of robots, and so on.

The first plot plots the output of the Kalman filter against the measurements and the actual position of our dog (drawn in green). After the initial settling in period the filter should track the dog's position very closely.

The next two plots show the variance of $x$ and of $\dot{x}$. If you look at the code, you will see that 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 despite initializing $\mathbf{P} = \left(\begin{smallmatrix}500 & 0\\ 0 & 500\end{smallmatrix}\right)$ we quickly converge to small variances for both the position and velocity. We will spend a lot of time on the covariance matrix later, so for now I will leave it at that.

In the previous 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 filter with this code merely by altering are 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 accelerations in 3D (which requires 9 state variables). We can do that with the code in this chapter.

## 6.8 Walking Through the KalmanFilter Code (Optional)

The code in the `KalmanFilter` is a nearly verbatim transcription of the linear algebra equations. I take advantage of numpy matrices to implement the linear algebra. It is worth looking at this code if for no other reason than to realize how easy it is to implement linear algebra with Python and numpy. For most of this book you will only really need to know how to call this class, not how to implement it from scratch.

> **sidebar**: numpy provides two data structures which can be used to perform linear algebra: `numpy.array` and `numpy.matrix`. The usual advice is to use `numpy.array`, not `numpy.matrix`. Ever the contrarian, I have chosen to use `numpy.matrix`, but for what I think are good pedalogical reasons. `numpy.array` is usually recommended because it can be sized to any arbitrary number of dimensions, and `numpy.matrix` is constrained to two dimensions. However, for Kalman filters we only need 2 dimensions. More importantly, `numpy.matrix` allows you to use very natural sytax.

Multipying a by b is written `a*b` if using `numpy.matrix`, but `a.dot(b)` if they are `numpy.array`. It is also more natural to mix scalars and matrices using `numpy.matrix`. Finally, the resulting code is extremely close to the equivalent Matlab code; if you are more familiar with Matlab than Python this code should feel very familiar to you.

The constructor of the class creates variables for each of the Kalman filter variables, and assigns them a reasonable default value. This is the code in its entirety:

```
def __init__(self, dim):
    """ Create a Kalman filter of dimension 'dim'"""

    self.x = 0
    self.P = np.matrix(np.eye(dim))
    self.Q = np.matrix(np.eye(dim))
    self.u = np.matrix(np.zeros((dim,1)))
    self.B = 0
    self.F = 0
    self.H = 0
    self.R = np.matrix(np.eye(1))
    self.I = np.matrix(np.eye(dim))
```

The function `predict()` implements the Kalman filter prediction equations.

```
def predict(self):
    self.x = (self.F*self.x) + (self.B * self.u)
    self.P = (self.F * self.P * self.F.T) + self.Q
```

This is nothing more than a transliteration of these equations.

$$\mathbf{x}' = \mathbf{F}\mathbf{x} + \mathbf{B}\mathbf{u}$$
$$\mathbf{P} = \mathbf{F}\mathbf{P}\mathbf{F}^T + \mathbf{Q}$$

Finally, the `update()` function implements the Kalman filter update equations in an equally straightforward way:

```
def update(self, Z):
    """
    Add a new measurement to the kalman filter.
    """
    y = Z - (self.H * self.x)
    S = (self.H * self.P * self.H.T) + self.R


    K = self.P * self.H.T * linalg.inv(S)
    self.x = self.x + (K*y)
    self.P = (self.I - (K*self.H))*self.P
```

Finally, for those reading this online or in a printed form, here is the code in `KalmanFilter.py` absent the unit testing code that is included in that file.

```
In [20]: import numpy as np
         import scipy.linalg as linalg

         class KalmanFilter:
```

```
    def __init__(self, dim):
        """ Create a Kalman filter of dimension 'dim'"""

        self.x = 0 # state
        self.P = np.matrix(np.eye(dim)) # uncertainty covariance
        self.Q = np.matrix(np.eye(dim)) # process uncertainty
        self.u = np.matrix(np.zeros((dim,1))) # motion vector
        self.B = 0
        self.F = 0 # state transition matrix
        self.H = 0 # Measurement function (maps state to measurements)
        self.R = np.matrix(np.eye(1)) # state uncertainty
        self.I = np.matrix(np.eye(dim))


    def update(self, Z):
        """
        Add a new measurement to the kalman filter.
        """

        # measurement update
        y = Z - (self.H * self.x)                     # error (residual) between measurement
                                                      # and prediction
        S = (self.H * self.P * self.H.T) + self.R     # project system uncertainty into
                                                      # measurment space + measurement noise(R)


        K = self.P * self.H.T * linalg.inv(S) # map system uncertainty into kalman gain

        self.x = self.x + (K*y)                  # predict new x with residual scaled
                                                 #by the kalman gain
        self.P = (self.I - (K*self.H))*self.P    # and compute the new covariance

    def predict(self):
        # prediction
        self.x = (self.F*self.x) + self.u
        self.P = self.F * self.P * self.F.T + self.Q
```

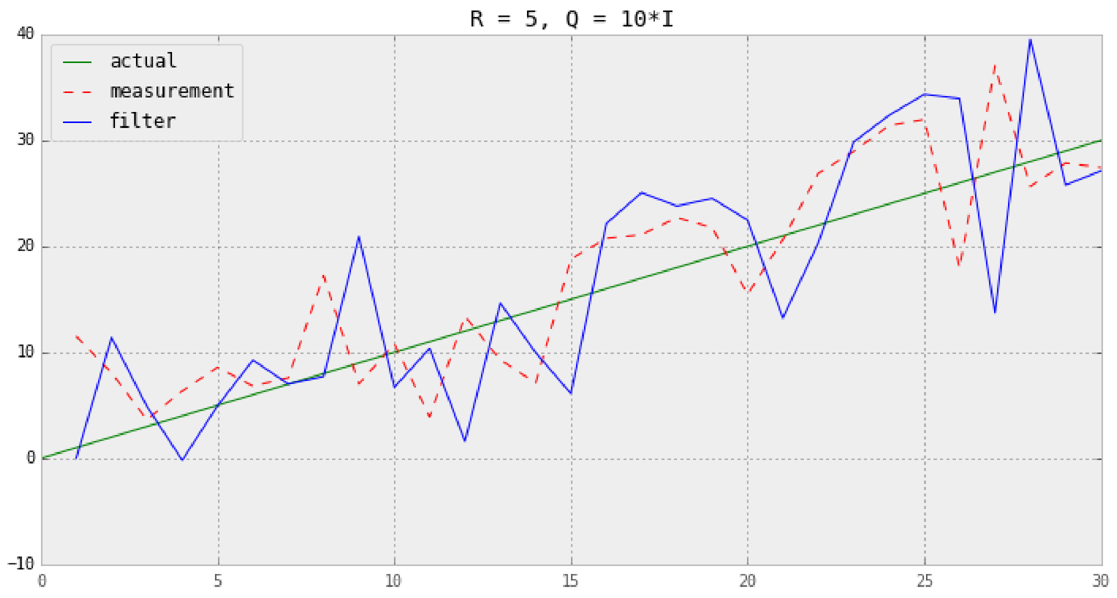
## 6.9 Adjusting the Filter

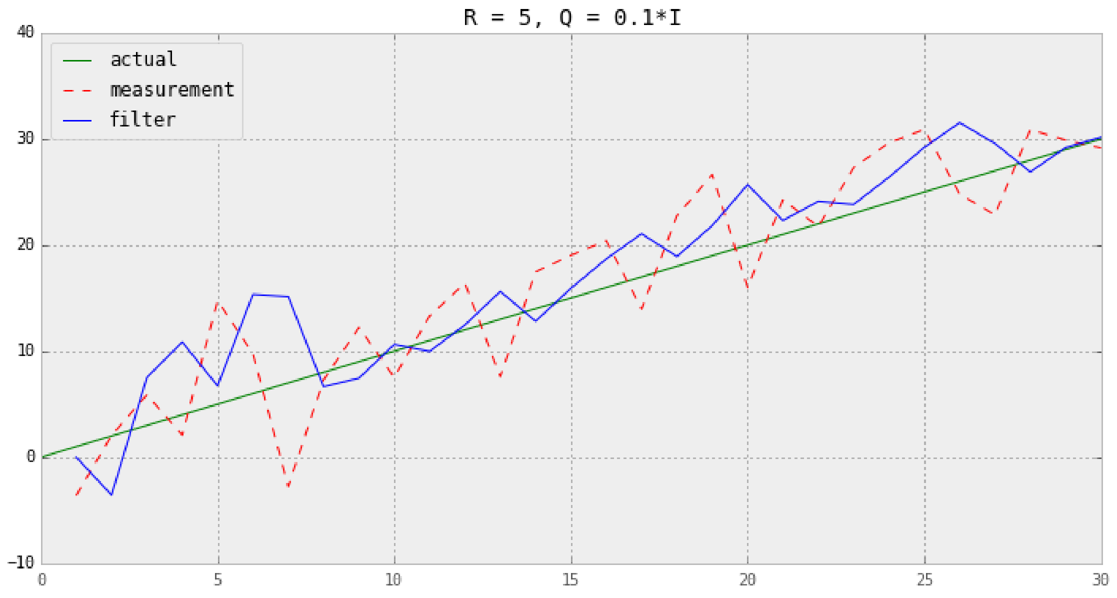
Your results will vary slightly depending on what numbers your random generator creates for the noise componenet of the noise, but the filter in the last section should track the actual position quite well. Typically as the filter starts up the first several predictions are quite bad, and varies a lot. But as the filter builds its state the estimates become much better.

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. As I wrote earlier, designing a Kalman filter is as much art as science. The science is, roughly, designing the **H** and **F** matrices - they develop in an obvious manner based on the physics of the system we are modelling. The art comes in modelling the sensors and selecting appropriate values for the rest of our variables.

Let's look at the effects of the noise parameters **R** and **Q**. I will only run the filter for twenty steps to ensure we can see see the difference between the measurements and filter output. I will start by holding **R** to 5 and vary **Q**.

```
In [21]: plot_track (noise=30, R=5, Q=10,count=30, plot_P=False, title='R = 5, Q = 10*I')
         plot_track (noise=30, R=5, Q=0.1,count=30, plot_P=False, title='R = 5, Q = 0.1*I')
```





The filter in the first plot should follow the noisy measurement almost exactly. 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.

In the Kalman filter **R** is the *measurement noise* and **Q** is the *process uncertainty*. **R** is the same in both plots, so ignore it for the moment. Why does **Q** affect the plots this way?

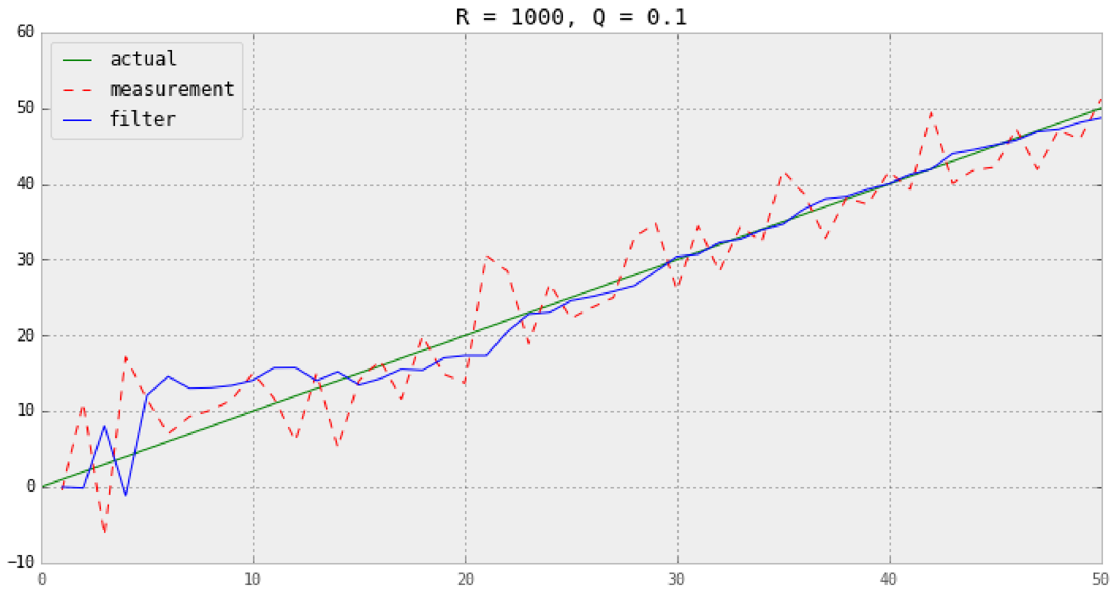
Let's remind ourselves of what the term *process uncertainty* means. Consider the problem of tracking a ball. We can accurately model its behavior in statid air with math, but if there is any wind our model will diverge from reality.

In the first case we set $\mathbf{Q} = 100$, which is quite large. In physical terms this is telling the filter "I don't trust my motion prediction step". Strictly speaking, we are telling the filter there is a lot of external noise that we are not modeling with $\mathbf{F}$, but the upshot of that is to not trust the motion prediction step. So 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 use but the measurements, and thus it follows the measurements closely.

In the second case we set $\mathbf{Q} = 0.1$, which is quite small. In physical terms we are telling the filter "trust the motion computation, it is really good!". Again, more strictly this actually says there is very small amounts of process noise, so the motion computation will be 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 leave $\mathbf{Q} = 0.1$, but bump $\mathbf{R}$ up to 1000. This is telling the filter that the measurement noise is very large.

```
In [22]: plot_track (noise=30, R=1000, Q=0.1,count=50, plot_P=False, title='R = 1000, Q = 0.1')
```



The filter output should be much closer to the green line, especially after 10-20 cycles. If you are running this in Ipython Notebook, I strongly urge you to run this many times in a row (click inside the code box, and press CTRL-Enter). Most times the filter tracks almost exactly with the actual position, randomly going slightly above and below the green line, but sometimes it stays well over or under the green line for a long time. What is happening in the latter case?

The filter is strongly preferring the motion update to the measurement, so if the prediction is off it takes a lot of measurements to correct it. It will eventually correct because the velocity is a hidden variable - it is computed from the measurements, but it will take awhile.

To some extent you can get similar looking output by varying either $\mathbf{R}$ or $\mathbf{Q}$, 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 $\mathbf{R}$ and/or $\mathbf{Q}$ based on your knowledge of the system you are filtering.

## 6.10 A Detailed Examination of the Covariance Matrix

So far I have not given a lot of coverage of the covariance matrix. **P**, the covariance matrix is nothing more than the variance of our state - such as the position of our dog. It has many elements in it, but don't be daunted; we will learn how to interpret a very large 9×9 covariance matrix, or even larger.

Recall the beginning of the chapter, where we provided the equation for the covariance matrix. It read:

$$\mathbf{P} = \begin{pmatrix} {\sigma_1}^2 & p\sigma_1\sigma_2 & \cdots & p\sigma_1\sigma_n \\ p\sigma_2\sigma_1 & {\sigma_2}^2 & \cdots & p\sigma_2\sigma_n \\ \vdots & \vdots & \ddots & \vdots \\ p\sigma_n\sigma_1 & p\sigma_n\sigma_2 & \cdots & {\sigma_n}^2 \end{pmatrix}$$

(I have subtituted **P** for $\Sigma$ because of the nomenclature used by the Kalman filter literature).

The diagonal contains the variance of each of our state variables. So, if our state variables are

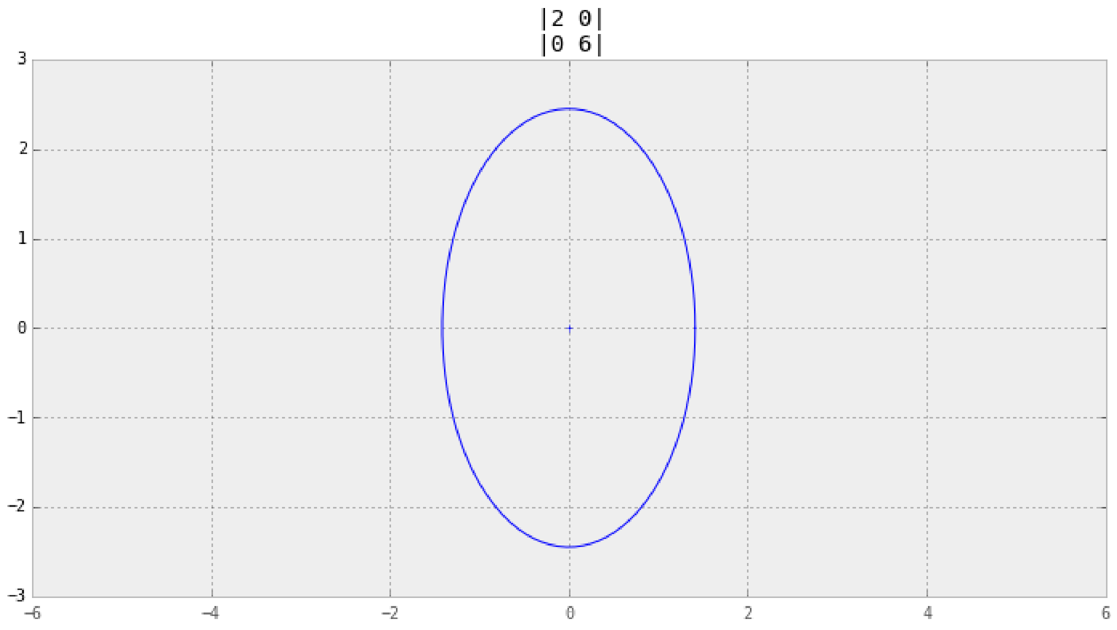
$$\begin{pmatrix} x \\ \dot{x} \end{pmatrix}$$

and the covariance matrix happens to be

$$\begin{pmatrix} 2 & 0 \\ 0 & 6 \end{pmatrix}$$

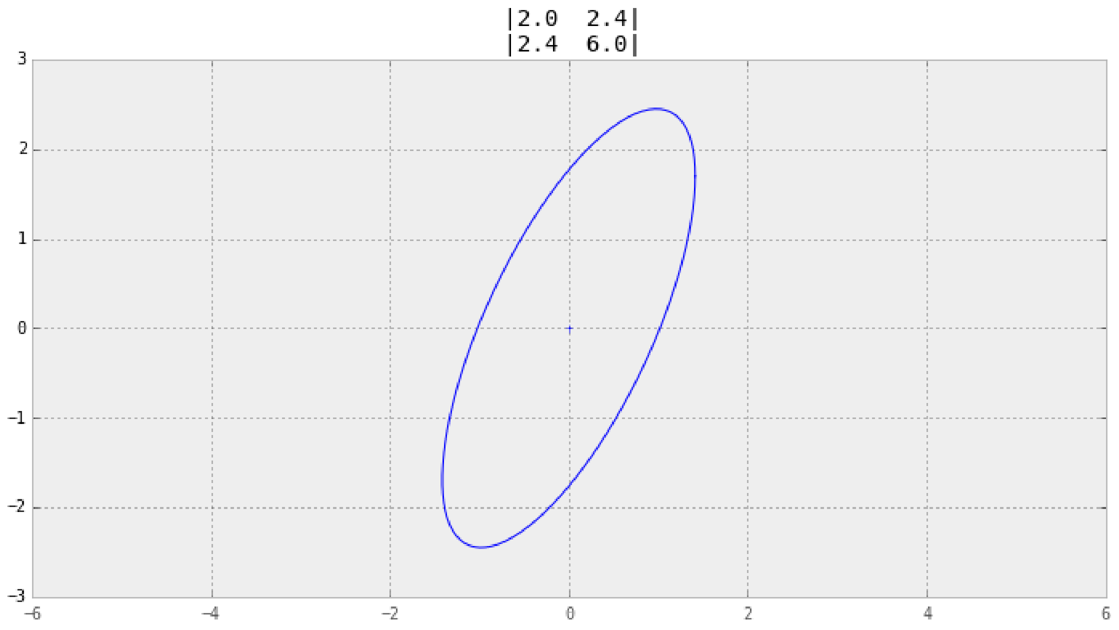
we know that the variance of $x$ is 2, and the variance of $\dot{x}$ is 6. The off diagonal elements are all 0, so we also know that $x$ and $\dot{x}$ are not correlated. Recall the ellipses that we drew of the covariance matrices. Let's look at the ellipse for the matrix.

```
In [23]: P = np.array([[2,0],[0,6]])
         e = stats.sigma_ellipse (P, 0, 0)
         stats.plot_sigma_ellipse(e, '|2 0|\n|0 6|')
```



Of course it is unlikely that the position and velocity of an object remain uncorrelated for long. Let's look at a more typical covariance matrix

```
In [24]: P = np.array([[2,2.4],[2.4,6]])
         e = stats.sigma_ellipse (P, 0, 0)
         stats.plot_sigma_ellipse(e, '|2.0  2.4|\n|2.4  6.0|')
```



Here the ellipse is slanted, signifying that $x$ and $\dot{x}$ are correlated (and, of course, dependent - all correlated variables are dependent). You may or may not have noticed that the off diagonal elements were set to the same value, 2.4. This was not an accident. Let's look at the equation for the covariance for the case where the number of dimensions is two.

$$\mathbf{P} = \begin{pmatrix} \sigma_1^2 & p\sigma_1\sigma_2 \\ p\sigma_2\sigma_1 & \sigma_2^2 \end{pmatrix}$$

Look at the computation for the off diagonal elements.

$$\mathbf{P}_{0,1} = p\sigma_1\sigma_2$$
$$\mathbf{P}_{1,0} = p\sigma_2\sigma_1.$$

If we re-arrange terms we get

$$\mathbf{P}_{0,1} = p\sigma_1\sigma_2$$
$$\mathbf{P}_{1,0} = p\sigma_1\sigma_1, \text{ yielding}$$
$$\mathbf{P}_{0,1} = P_{1,0}$$

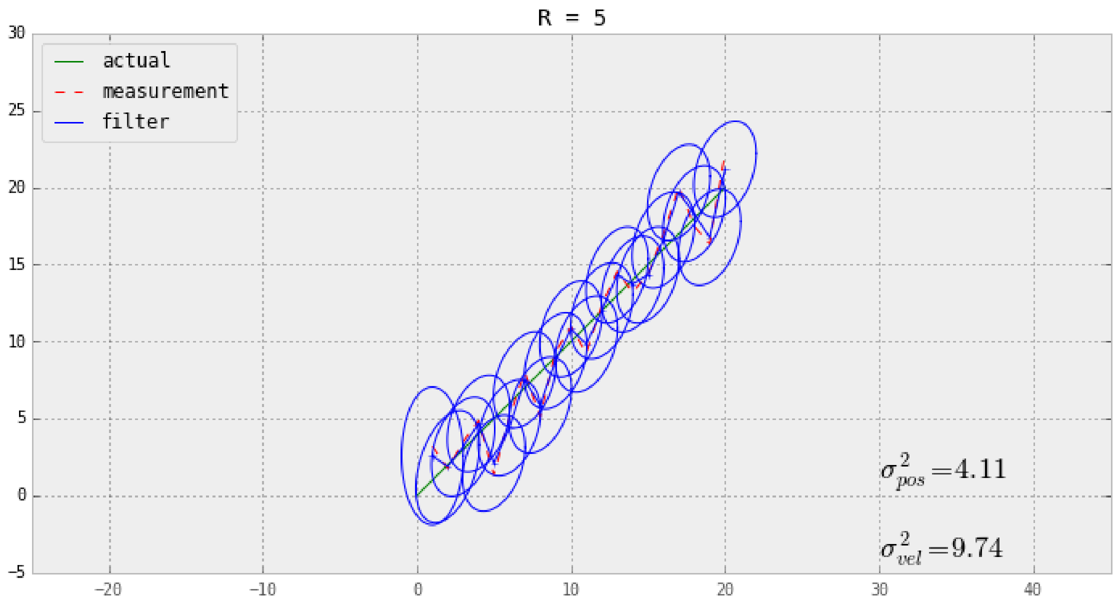
In general, we can state that $\mathbf{P}_{i,j} = \mathbf{P}_{j,i}$.

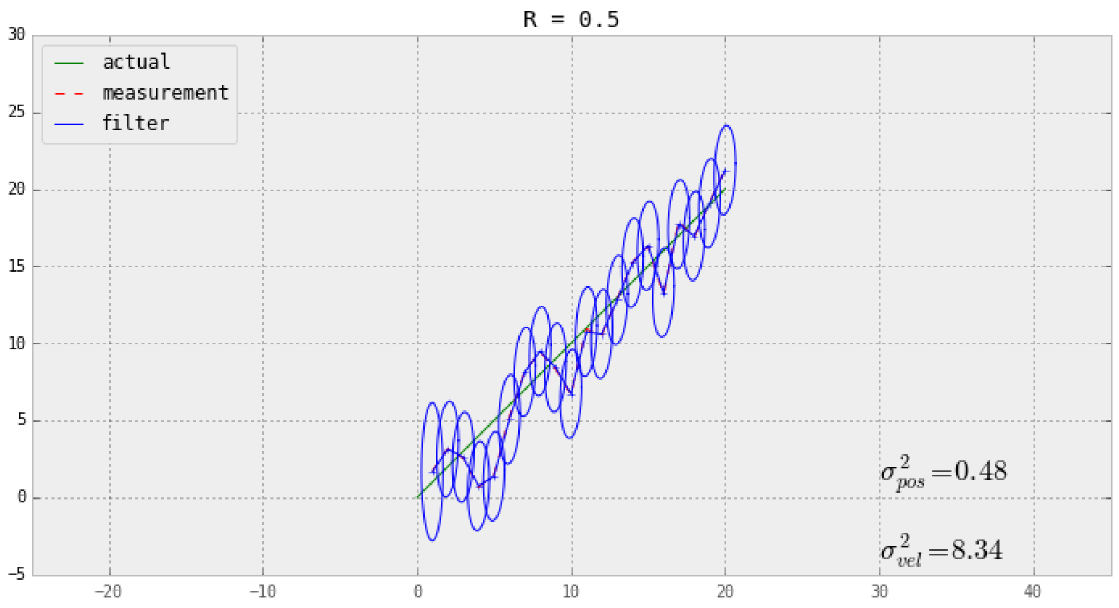
So for my example I multiplied the diagonals, 2 and 6, to get 12, and then scaled that with the arbitrarily chosen $p = .2$ to get 2.4.

Let's get back to concrete terms. Lets do another Kalman filter for our dog, and this time plot the covariance ellipses on the same plot as the position.

```
In [25]: def plot_track(noise, count, R, Q=0, plot_P=True, title='Kalman Filter'):
             dog = DogSensor(velocity=1, noise=noise)
             f = dog_tracking_filter(R=R, Q=Q, cov=20.)

             ps = []
```

```
    zs = []
    cov = []
    for t in range (count):
        z = dog.sense()
        f.update (z)
        ps.append (f.x[0,0])
        cov.append(f.P)
        zs.append(z)
        f.predict()

    p0, = plt.plot([0,count],[0,count],'g')
    p1, = plt.plot(range(1,count+1),zs,c='r', linestyle='dashed')
    p2, = plt.plot(range(1,count+1),ps, c='b')
    plt.legend([p0,p1,p2], ['actual','measurement', 'filter'], 2)
    plt.title(title)

    for i,p in enumerate(cov):
        e = stats.sigma_ellipse (cov=p, x=i+1, y=ps[i])
        stats.plot_sigma_ellipse(ellipse=e)
        if i == len(cov)-1:
            s = ('$\sigma^2_{pos} = %.2f$' % p[0,0])
            plt.text (30,1,s,fontsize=18)
            s = ('$\sigma^2_{vel} = %.2f$' % p[1,1])
            plt.text (30,-4,s,fontsize=18)
    plt.xlim((0,40))
    plt.ylim((0,40))
    plt.axis('equal')

    plt.show()


plot_track (noise=5, R=5, Q=5, count=20, title='R = 5')
plot_track (noise=5, R=.5, Q=5, count=20, title='R = 0.5')
```





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 $\mathbf{P} = \left(\begin{smallmatrix} 50 & 0 \\ 0 & 50 \end{smallmatrix}\right)$, 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 slighly 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 $\mathbf{R} = 5$, 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. The variance for position is 3.85, and the variance for velocity is 3.0.

In contrast, the second plot, with $\mathbf{R} = 0.5$, 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.

## 6.11 Question: Explain Ellipse Differences

Why are the ellipses for $\mathbf{R} = 5$ shorter, and more tilted than the ellipses for $\mathbf{R} = 0.5$. 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 and 0.1, observe the changes, and think about what this means.

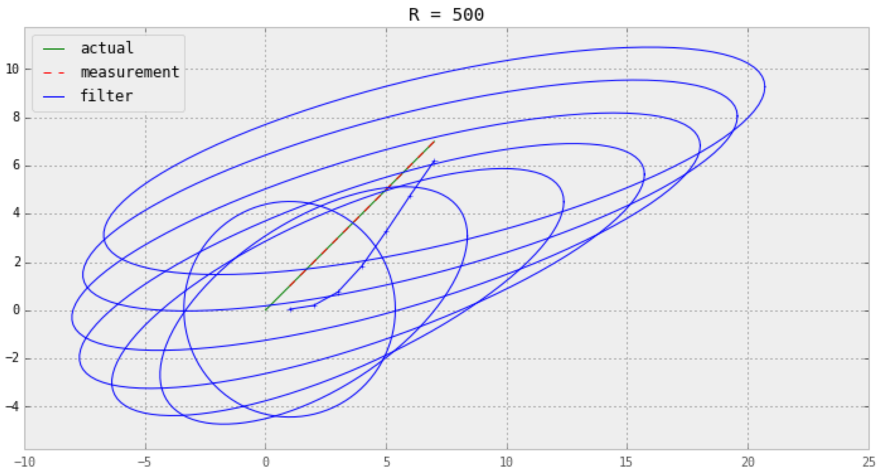
### 6.11.1 Solution

The $x$ axis is for position, and $y$ is velocity. An ellipse that is vertical, or nearly so, says there is no correlation between position and velocity, and an ellipse that is diagnal says that there is a lot of correlation. Phrased that way, it sounds unlikely - either they are correlated or not. 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. So, 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 just 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 correlation in the mathematical model. It's just 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 $\mathbf{R} = 500$. Think about what the plot will look like before scrolling down. To emphasize the issue, I will set the amount of noise injected into the measurements to 0, so the measurement will exactly equal the actual position.

```
In [26]: plot_track (noise=0, R=500, Q=5, count=7, title='R = 500')
```



$\sigma^2_{pos}$ =188.11

$\sigma^2_{vel}$ =22.25

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 start dealing with a, say, 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 will not be able to plot a 9×9 matrix on the screen because it would require living in 10-D space, so you have to develop your intution and understanding in this simple, 2-D case.

> **sidebar**: when plotting covariance ellipses, make sure to always use *plt.axis('equal')* in your code. 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.

```
In [26]:
```

# Chapter 7

# Kalman Filter Math

```
In [1]: #format the book
        %matplotlib inline
        from __future__ import division, print_function
        import matplotlib.pyplot as plt
        import book_format
        book_format.load_style()

Out[1]: <IPython.core.display.HTML at 0x284b910>
```

This chapter is optional, especially the first time you go through this material. If you are a hobbiest you will be able to get quite far by just skipping this chapter and going on to learn the examples. However, to have any hope of reading the original Kalman filter or optimal estimation literature you will have to acquire some of the more formal mathematics. I will not endevour to teach linear algebra, statistics, or calculus here as each topic deserves a book on its own.

blah blah blah

## 7.1 Walking Through the Kalman Filter Equations

I promised that you would not have to understand how to derive Kalman filter equations, and that is true. However, I do think it is worth walking through the equations one by one and becoming familiar with the variables. If this is your first time through the material feel free to skip ahead to the next section. However, you will eventually want to work through this material, so why not now? You will need to have passing familarity with these equations to read material written about the Kalman filter, as they all presuppose that you are familiar with the equations. I will reiterate them here for easy reference.

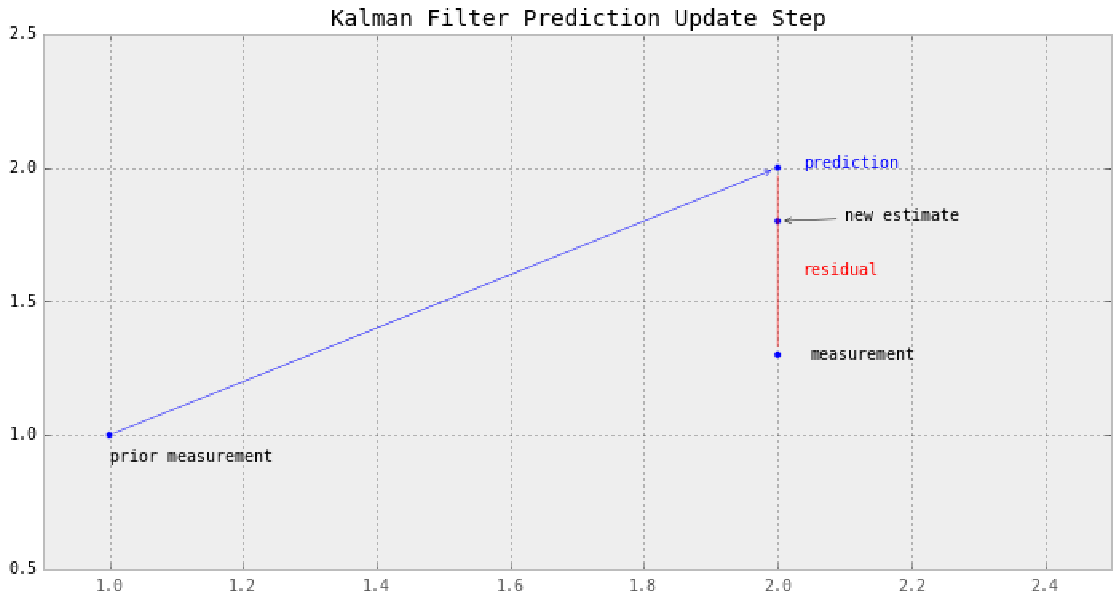
$$\begin{aligned}
&\text{Predict Step}\\
\mathbf{x}' &= \mathbf{F}\mathbf{x} + \mathbf{B}\mathbf{u} \quad &(1)\\
\mathbf{P} &= \mathbf{F}\mathbf{P}\mathbf{F}^T + \mathbf{Q} \quad &(2)\\
\\
&\text{Update Step}\\
\gamma &= \mathbf{z} - \mathbf{H}\mathbf{x} \quad &(3)\\
\mathbf{K} &= \mathbf{P}\mathbf{H}^T(\mathbf{H}\mathbf{P}\mathbf{H}^T + \mathbf{R})^{-1} \quad &(4)\\
\mathbf{x} &= \mathbf{x}' + \mathbf{K}\gamma \quad &(5)\\
\mathbf{P} &= (\mathbf{I} - \mathbf{K}\mathbf{H})\mathbf{P} \quad &(6)
\end{aligned}$$

I will start with the measurement step, as that is what we started with in the one dimensional Kalman filter case. Our first equation is

$$\gamma = \mathbf{z} - \mathbf{Hx} \tag{3}$$

On the right we have $\mathbf{Hx}$. That should be recognizable as the measurement function. Multiplying $\mathbf{H}$ with $\mathbf{x}$ puts $\mathbf{x}$ into *measurement space*; in other words, the same basis and units as the sensor's measurements. The variable $\mathbf{z}$ is just the measurement; it is typical, but not universal to use $\mathbf{z}$ to denote measurements in the literature ($\mathbf{y}$ is also sometimes used). Do you remember this chart?

```
In [3]: import mkf_internal
        mkf_internal.show_residual_chart()
```



The blue prediction line is the output of $\mathbf{Hx}$, and the dot labelled "measurement" is $\mathbf{z}$. Therefore, $\gamma = \mathbf{z} - \mathbf{Hx}$ is how we compute the residual, drawn in red. So $\gamma$ is the residual.

The next line is the formidable:

$$\mathbf{K} = \mathbf{PH}^T(\mathbf{HPH}^T + \mathbf{R})^{-1} \tag{4}$$

Unfortunately it is a fair amount of linear algebra to derive this. The derivation can be quite elegant, and I urge you to look it up if you have the mathematical education to follow it. But $\mathbf{K}$ is just the *Kalman gain* - the ratio of how much measurement vs prediction we should use to create the new estimate. $\mathbf{R}$ is the *measurement noise*, and $\mathbf{P}$ is our *uncertainty covariance matrix.*

So let's work through this expression by expression. Start with $\mathbf{HPH}^T$. The linear equation $\mathbf{ABA}^T$ can be thought of as changing the basis of $\mathbf{B}$ to $\mathbf{A}$. So $\mathbf{HPH}^T$ is taking the covariance $\mathbf{P}$ and putting it in measurement ($\mathbf{H}$) space. Then, once in measurement space, we can add the measurement noise $\mathbf{R}$ to it. Hence, the expression for the uncertainty in the measurement is:

$$(\mathbf{HPH}^T + \mathbf{R})^{-1}$$

Taking the inverse is linear algebra's way of doing $\frac{1}{x}$. So if you accept my admittedly hand wavely explanation it can be seen to be computing:

$$gain_{measurement\,space} = \frac{uncertainty_{prediction}}{uncertainty_{measurement}}$$

In other words, the *Kalman gain* equation is doing nothing more than computing a ratio based on how much we trust the prediction vs the measurement. If we are confident in our measurements and unconfident in our predictions $\mathbf{K}$ will favor the measurement, and vice versa. The equation is complicated because we are doing this in multiple dimensions via matrices, but the concept is simple - scale by a ratio.

Without going into the derivation of $\mathbf{K}$, I'll say that this equation is the result of finding a value of $\mathbf{K}$ that optimizes the *mean-square estimation error.* It does this by finding the minimal values for $\mathbf{P}$ along it's diagonal. Recall that the diagonal of $P$ is just the variance for each state variable. So, this equation for $\mathbf{K}$ ensures that the Kalman filter output is optimal. To put this in concrete terms, for our dog tracking problem this means that the estimates for both position and velocity will be optimal - a value of $\mathbf{K}$ that made the position extremely accurate but the velocity very inaccurate would be rejected in favor of a $\mathbf{K}$ that made both position and velocity just somewhat accurate.

Our next line is:

$$\mathbf{x} = \mathbf{x}' + \mathbf{K}\gamma \tag{5}$$

This just multiplies the residual by the Kalman gain, and adds it to the state variable. In other words, this is the computation of our new estimate.

Finally, we have:

$$\mathbf{P} = (\mathbf{I} - \mathbf{KH})\mathbf{P} \tag{6}$$

$I$ is the identity matrix, and is the way we represent 1 in multiple dimensions. $H$ is our measurement function, and is a constant. So, simplified, this is simply $P = (1 - cK)P$. $K$ is our ratio of how much prediction vs measurement we use. So, if $K$ is large then $(1 - cK)$ is small, and P will be made smaller than it was. If $K$ is small, then $(1 - cK)$ is large, and P will be made larger than it was. So we adjust the size of our uncertainty by some factor of the *Kalman gain.* I would like to draw your attention back to the g-h filter, which included this Python code:

```
# update filter
w = w * (1-scale_factor) + z * scale_factor
```

This multidimensional Kalman filter equation is partially implementing this calculation for the variance instead of the state variable.

Now we have the measurement steps. The first equation is

$$\mathbf{x}' = \mathbf{Fx} + \mathbf{Bu} \tag{1}$$

This is just our state transition equation which we have already discussed. $\mathbf{Fx}$ multiplies $\mathbf{x}$ with the state transition matrix to compute the next state. $B$ and $u$ add in the contribution of the control input $\mathbf{u}$, if any.

The final equation is:

$$\mathbf{P} = \mathbf{FPF}^T + \mathbf{Q} \tag{2}$$

$\mathbf{FPF}^T$ is the way we put $\mathbf{P}$ into the process space using linear algebra so that we can add in the process noise $\mathbf{Q}$ to it.

# Chapter 8

# Designing Kalman Filters

```
In [1]: #format the book
        %matplotlib inline
        from __future__ import division, print_function
        import matplotlib.pyplot as plt
        import book_format
        book_format.load_style()

Out[1]: <IPython.core.display.HTML at 0x1ae78d0>
```

## 8.1 Introduction

In this chapter we will work through the design of several Kalman filters to gain experience and confidence with the various equations and techniques.

For our first multidimensional problem we will track a robot in a 2D space, such as a field. 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. blah blah

## 8.2 Tracking a Robot

This first attempt at tracking a robot will closely resemble the 1-D dog tracking problem of previous chapters. This will allow us to 'get our feet wet' with Kalman filtering. So, instead of a sensor that outputs position in a hallway, we now have a sensor that supplies a noisy measurement of position in a 2-D space, such as an open field. That is, at each time $T$ it will provide an $(x, y)$ coordinate pair specifying the measurement of the sensor's position in the field.

Implemention of code to interact with real sensors is beyond the scope of this book, so as before we will program simple simuations in Python to represent 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. `pos_sensor1 ()` is the first sensor we write, and so on.

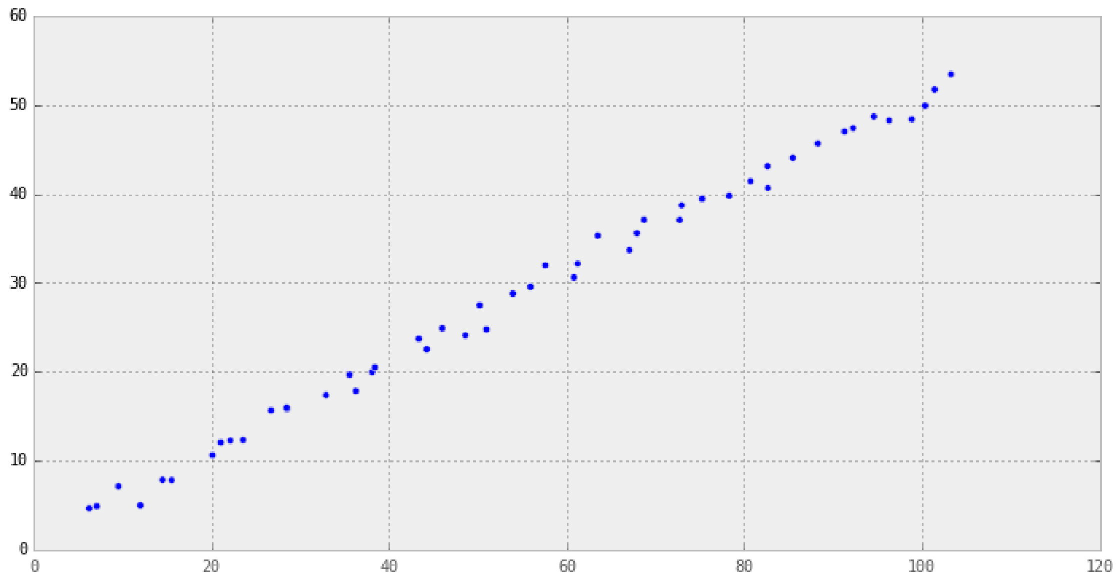
So let's start with a very simple sensor, one that travels in a straight line. It takes as input the last position, velocity, and how much noise we want, and returns the new position.

```
In [2]: import numpy.random as random
        import copy
        class PosSensor1(object):
            def __init__(self, pos = [0,0], vel = (0,0), noise_scale = 1.):
                self.vel = vel
                self.noise_scale = noise_scale
                self.pos = copy.deepcopy(pos)
```

```
    def read(self):
        self.pos[0] += self.vel[0]
        self.pos[1] += self.vel[1]

        return [self.pos[0] + random.randn() * self.noise_scale,
                self.pos[1] + random.randn() * self.noise_scale]
```

A quick test to verify that it works as we expect.

```
In [3]: pos = [4,3]
        s = PosSensor1 (pos, (2,1), 1)

        for i in range (50):
            pos = s.read()
            plt.scatter(pos[0], pos[1])
        plt.show()
```



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).

**Step 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

$$\mathbf{x} = \begin{bmatrix}x \\ v_x \\ y \\ v_y\end{bmatrix}$$

There is nothing special about this organization. I could have listed the (xy) coordinates first followed by the velocities, and/or I could done this as a row matrix instead of a column matrix. For example, I could have chosen:

$$\mathbf{x} = \begin{bmatrix} x & y & v_x & v_y \end{bmatrix}$$

All that matters is that the rest of my derivation uses this same scheme. However, it is typical to use column matrices for state variables, and I prefer it, so that is what we will use.

It might be a good time to pause and address how you identify the unobserved variables. This particular example is somewhat obvious because we already worked through the 1D case in the previous chapters. Would it be so obvious if we were filtering market data, population data from a biology experiement, and so on? Probably not. 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.

**Step 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 **F** that we multipy 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. Our state equations for position and velocity would be:

$$\begin{aligned}
x' &= (1 * x) + (\Delta t * v_x) + (0 * y) + (0 * v_y) \\
v_x &= (0 * x) + (1 * v_x) + (0 * y) + (0 * v_y) \\
y' &= (0 * x) + (0 * v_x) + (1 * y) + (\Delta t * v_y) \\
v_y &= (0 * x) + (0 * v_x) + (0 * y) + (1 * v_y)
\end{aligned}$$

Laying it out that way shows us both the values and row-column organization required for **F**. In linear algebra, we would write this as:

$$\begin{bmatrix} x \\ v_x \\ y \\ v_y \end{bmatrix}' = \begin{bmatrix} 1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ v_x \\ y \\ v_y \end{bmatrix}$$

So, let's do this in Python. It is very simple:

```
In [4]: from KalmanFilter import KalmanFilter
        import numpy as np

        f1 = KalmanFilter(dim=4)
        dt = 1.   # time step

        f1.F = np.mat ([[1, dt, 0,  0],
                        [0,  1, 0,  0],
                        [0,  0, 1, dt],
                        [0,  0, 0,  1]])
```

**Step 3: Design the Motion Function** We have no control inputs to our robot (yet!), so this step is trivial - set the motion transition function **B** to zero. This is done for us by the class when it is created so we can skip this step, but for completeness we will be explicit.

```
In [5]: f1.B = 0.
```

**Step 4: Design the Measurement Function** The measurement function defines how we go from the state variables to the measurements. For this problem this is very easy - the measurements are the position themselves(x,y), and we have no data for velocity. Let's make this just slightly more interesting by deciding we want to change units. So we will assume that the measurements are returned in feet, and that we desire to work in meters. Converting from feet to meters is a simple as multiplying by 0.3048. So, to get:

$$\mathbf{z} = \mathbf{H}\mathbf{x}$$

we write the following:

$$\begin{aligned} z'_x &= (\frac{x}{0.3048}) + (0 * v_x) && +(0 * y) + (0 * v_y) \\ z'_y &= (0 * x) + (0 * v_x) + (\frac{y}{0.3048}) + (0 * v_y) \end{aligned}$$

Perhaps the $\frac{1}{0.3048}$ caught you off guard. At first I always found **H** a bit counter-intuitive to design because it takes you from the state variables to the measurements, but the Kalman filter is trying to take measurements and produce state variables to them. If you read the math chapter you will understand why **H** is designed to go in this direction. If not, well, you'll have to remember how this works and trust that it is correct.

So, here is the Python that implements this:

```
In [6]: f1.H = np.mat ([[1/0.3048, 0, 0, 0],
                        [0, 0, 1/0.3048, 0]])
        print(f1.H)

[[ 3.2808399  0.         0.         0.       ]
 [ 0.         0.         3.2808399  0.       ]]
```

**Step 5: Design the Measurement Noise Matrix** In this step we need to mathematically model the noise in our sensor. For now we will make the simple assumption that the $x$ and $y$ variables are independent 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. For now let's set the variance for $x$ and $y$ to be 5 for each. They are independent, so there is no covariance, and our off diagonals will be 0. This gives us:

$$\mathbf{R} = \begin{bmatrix}5 & 0 \\ 0 & 5\end{bmatrix}$$

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 [7]: f1.R = np.mat([[5,0],
                       [0, 5]])
        print (f1.R)

[[5 0]
 [0 5]]
```

**Step 6: Design the Process Noise Matrix** Finally, we design the process noise. We don't yet have a good way to model process noise, so for now we will assume there is a small amount of process noise, say 0.1 for each state variable. Later we will tackle this admittedly difficult topic in more detail. We have 4 state variables, so we need a 4×4 covariance matrix:

$$\mathbf{Q} = \begin{bmatrix}0.1 & 0 & 0 & 0 \\ 0 & 0.1 & 0 & 0 \\ 0 & 0 & 0.1 & 0 \\ 0 & 0 & 0 & 0.1\end{bmatrix}$$

In Pyton I will use the numpy eye helper function to create an identity matrix for us, and multipy it by 0.1 to get the desired result.

```
In [8]: f1.Q = np.eye(4) * 0.1
        print(f1.Q)
```

```
[[ 0.1  0.   0.   0. ]
 [ 0.   0.1  0.   0. ]
 [ 0.   0.   0.1  0. ]
 [ 0.   0.   0.   0.1]]
```

**Step 7: Design 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 **P** to a large value.

$$\mathbf{x} = \begin{bmatrix}0\\0\\0\\0\end{bmatrix} \mathbf{P} = \begin{bmatrix}500 & 0 & 0 & 0 \\ 0 & 500 & 0 & 0 \\ 0 & 0 & 500 & 0 \\ 0 & 0 & 0 & 500\end{bmatrix}$$

In Python we implement that with

```
In [9]: f1.x = np.mat([0,0,0,0]).T
        f1.P = np.eye(4) * 500.
        print(f1.x)
        print()
        print (f1.P)
```

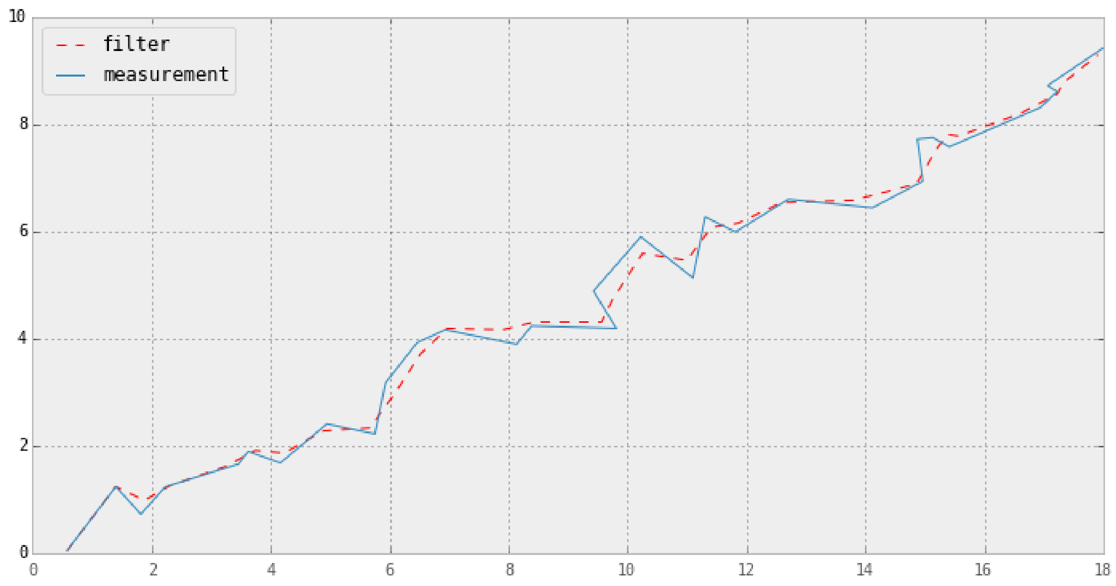
```
[[0]
 [0]
 [0]
 [0]]

[[ 500.    0.    0.    0.]
 [   0.  500.    0.    0.]
 [   0.    0.  500.    0.]
 [   0.    0.    0.  500.]]
```

## 8.3 Implement the Filter Code

Design is complete, now we just have to write the Python code to run our filter, and output the data in the format of our choice. To keep the code clear, let's just print a plot of the track. We will run the code for 30 iterations.

```
In [10]: f1 = KalmanFilter(dim=4)
         dt = 1.0   # time step
```

```
f1.F = np.mat ([[1, dt, 0,  0],
                [0,  1, 0,  0],
                [0,  0, 1, dt],
                [0,  0, 0,  1]])
f1.B = 0.
f1.H = np.mat ([[1/0.3048, 0, 0, 0],
               [0, 0, 1/0.3048, 0]])

f1.R = np.eye(2) * 5
f1.Q = np.eye(4) * .1

f1.x = np.mat([0,0,0,0]).T
f1.P = np.eye(4) * 500.

# initialize storage and other variables for the run
count = 30
xs, ys = [],[]
pxs, pys = [],[]

s = PosSensor1 ([0,0], (2,1), 1.)

for i in range(count):
    pos = s.read()
    z = np.mat([[pos[0]],[pos[1]]])

    f1.predict ()
    f1.update (z)

    xs.append (f1.x[0,0])
    ys.append (f1.x[2,0])
    pxs.append (pos[0]*.3048)
    pys.append(pos[1]*.3048)

p1, = plt.plot (xs, ys, 'r--')
p2, = plt.plot (pxs, pys)
plt.legend([p1,p2], ['filter', 'measurement'], 2)
plt.show()
```

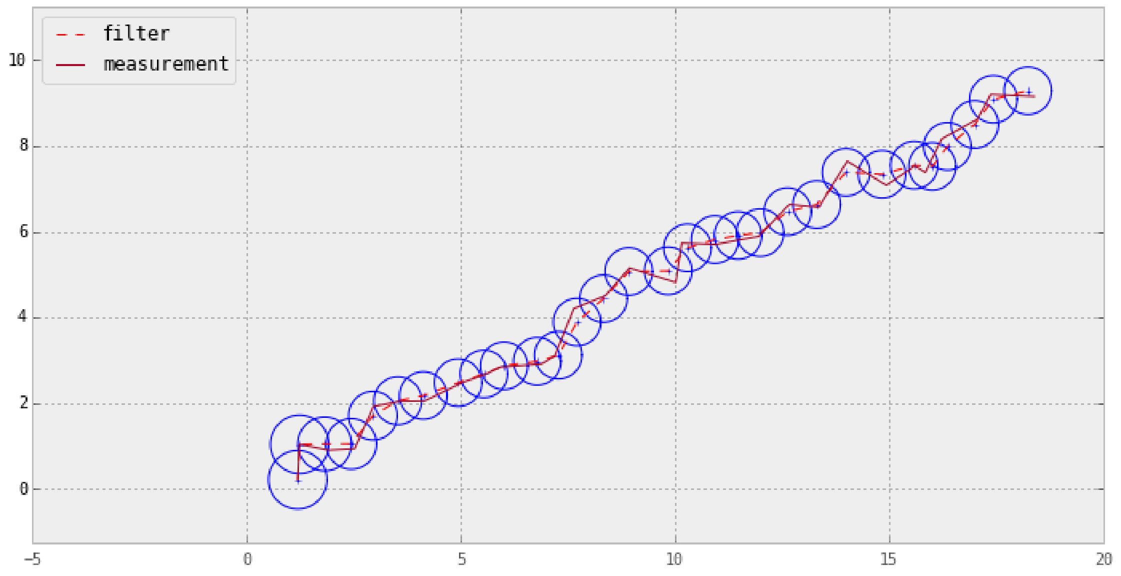


I encourage you to play with this, setting **Q** and **R** to various values. However, we did a fair amount of that sort of thing in the last chapters, and we have a lot of material to cover, so I will move on to more complicated cases where we will also have a chance to experience changing these values.

Now I will run the same Kalman filter with the same settings, but also plot the covariance ellipse for $x$ and $y$. First, the code without explanation, so we can see the output. I print the last covariance to see what it looks like. But before you scroll down to look at the results, what do you think it will look like? You have enough information to figure this out but this is still new to you, so don't be discouraged if you get it wrong.

```
In [11]: import stats

         f1 = KalmanFilter(dim=4)
         dt = 1.0    # time step

         f1.F = np.mat ([[1, dt, 0,  0],
                         [0,  1, 0,  0],
                         [0,  0, 1, dt],
                         [0,  0, 0,  1]])
         f1.B = 0.
         f1.H = np.mat ([[1/0.3048, 0, 0, 0],
                        [0, 0, 1/0.3048, 0]])

         f1.R = np.eye(2) * 5
         f1.Q = np.eye(4) * .1

         f1.x = np.mat([0,0,0,0]).T
         f1.P = np.eye(4) * 500.

         # initialize storage and other variables for the run
         count = 30
         xs, ys = [],[]
         pxs, pys = [],[]

         s = PosSensor1 ([0,0], (2,1), 1.)
```

```
for i in range(count):
    pos = s.read()
    z = np.mat([[pos[0]],[pos[1]]])

    f1.predict ()
    f1.update (z)

    xs.append (f1.x[0,0])
    ys.append (f1.x[2,0])
    pxs.append (pos[0]*.3048)
    pys.append(pos[1]*.3048)

    # plot covariance of x and y
    cov = np.mat([[f1.P[0,0], f1.P[2,0]],
                  [f1.P[0,2], f1.P[2,2]]])

    e = stats.sigma_ellipse (cov=cov, x=f1.x[0,0], y=f1.x[2,0])
    stats.plot_sigma_ellipse(ellipse=e)

p1, = plt.plot (xs, ys, 'r--')
p2, = plt.plot (pxs, pys)
plt.legend([p1,p2], ['filter', 'measurement'], 2)
plt.show()
print("final P is:")
print(f1.P)
```



```
final P is:
[[ 0.30660483  0.12566239  0.          0.        ]
 [ 0.12566239  0.24399092  0.          0.        ]
 [ 0.          0.          0.30660483  0.12566239]
 [ 0.          0.          0.12566239  0.24399092]]
```

Did you correctly predict what the covariance matrix and plots would look like? 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 modelled to have the same value, 5, in **R**. If we were to set R to, for example,

$$\mathbf{R} = \begin{bmatrix}10 & 0\\ 0 & 5\end{bmatrix}$$

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 P tells us everything we need to know about the correlation beween 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()*.

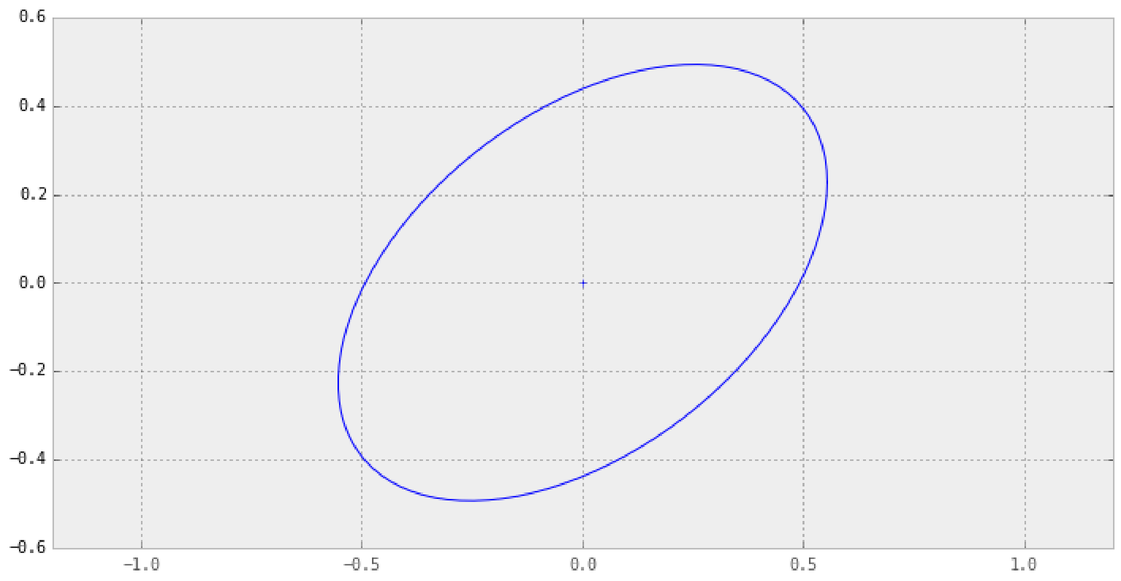
```
In [12]: print(np.diag(f1.P))
```

```
[ 0.30660483  0.24399092  0.30660483  0.24399092]
```

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 $\dot{x}$. Let's extract and print, and plot it.

```
In [13]: c = f1.P[0:2,0:2]
         print(c)
         stats.plot_covariance_ellipse(c)
```

```
[[ 0.30660483  0.12566239]
 [ 0.12566239  0.24399092]]
```



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 $p\sigma_1\sigma_2$.

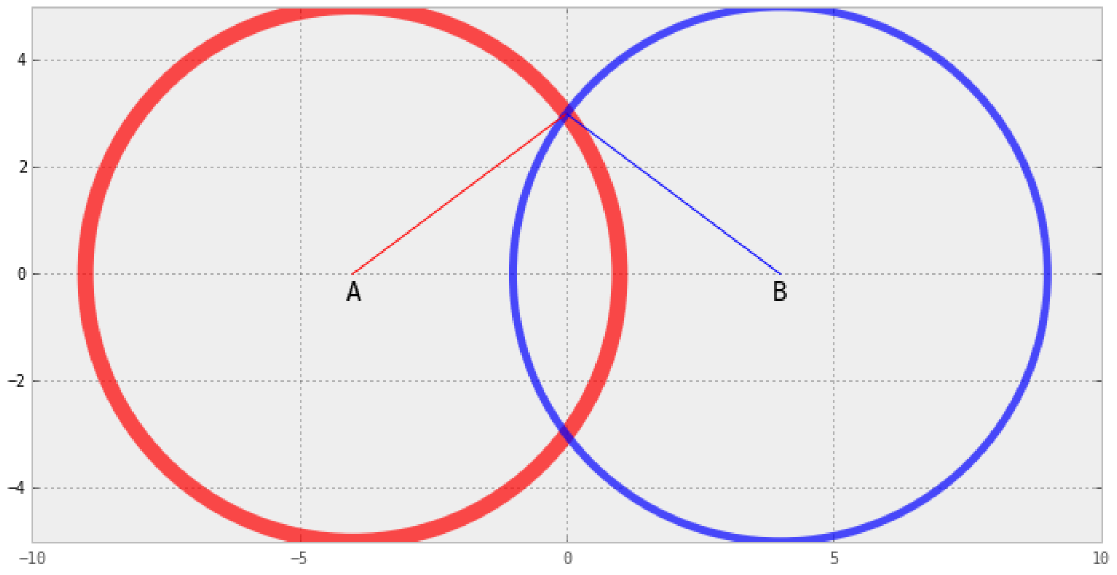
Finally, let's look at the lower left side of **P**, which is all 0s. Why 0s? Consider $\mathbf{P}_{3,0}$. That stores the term $p\sigma_3\sigma_0$, which is the covariance between $\dot{y}$ and $x$. These are independent, so the term will be 0. The rest of the terms are for similarly independent variables.

## 8.4 Realistic 2D Position Sensors

The position sensor in the last example are not very realistic. In general there is no 'raw' sensor that provides (x,y) coordinates. We have GPS, but GPS already uses a Kalman filter to create a filtered output; we should not be able to improve the signal by passing it through another Kalman filter unless we incorporate additional sensors to provide additional information. We will tackle that problem later.

Consider the following set up. In an open field we put two transmitters at a known location, each transmitting a signal that we can detect. We process the signal and determine how far we are from that signal, with some noise. First, let's look at a visual depiction of that.

```
In [14]: circle1=plt.Circle((-4,0),5,color='r',fill=False,linewidth=10, alpha=.7)
         circle2=plt.Circle((4,0),5,color='b', fill=False, linewidth=5, alpha=.7)

         fig = plt.gcf()
         ax = fig.gca()

         plt.axis('equal')
         plt.xlim((-10,10))
         plt.ylim((-10,10))

         plt.plot ([-4,0], [0,3], 'r')
         plt.plot ([4,0], [0,3], 'b')
         plt.text(-4, -.5, "A", fontsize=16, horizontalalignment='center')
         plt.text(4, -.5, "B", fontsize=16, horizontalalignment='center')
         #plt.scatter ([-4],[0], 'r')

         ax.add_artist(circle1)
         ax.add_artist(circle2)
         plt.show()
```



Here I have attempted to show transmitter A, drawn in red, at (-4,0) and a second one B, drawn in blue, at (4,0). The red and blue circles show the range from the transmitters to the robot, with the width illustrating the effect of the $1\sigma$ angular error for each transmitter. Here I have given the red transmitter more error than the blue one. The most probable position for the robot is where the two circles intersect, which I have depicted with the red and blue lines. You will object that we have two intersections, not one, but we will see how we deal with that when we design the measurement function.

This is a very common sensor set up. Aircraft still use this system to navigate, where it is called DME (Distance Measuring Equipment). Today GPS is a much more common navigation sytem, but I have worked on an aircraft where we integrated sensors like this into our filter along with the GPS, INS, altimeters, etc. We will tackle what is called *multi-sensor fusion* later; for now we will just address this simple configuration.

The first step is to design our state variables. We will assume that the robot is travelling in a straight direction with constant velocity. This is unlikely to be true for a long period of time, but is acceptable for short periods of time. This does not differ from the previous problem - we will want to track the values for the robot's position and velocity. Hence,

$$\mathbf{x} = \begin{bmatrix} x \\ v_x \\ y \\ v_y \end{bmatrix}$$

The next step is to design the state transistion function. This also will be the same as the previous problem, so without further ado,

$$\mathbf{x}' = \begin{bmatrix} 1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1 \end{bmatrix} \mathbf{x}$$

The next step is to design the control inputs. We have none, so we set $\mathbf{B} = 0$.

The next step is to design the measurement function $\mathbf{z} = \mathbf{Hx}$. We can model the measurement using the Pythagorean theorem.

$$z_a = \sqrt{(x - x_A)^2 + (y - y_A)^2} + v_a z_b = \sqrt{(x - x_B])^2 + (y - y_B)^2} + v_b$$

where $v_a$ and $v_b$ are white noise.

We see an immediate problem. The Kalman filter is designed for linear equations, and this is obviously nonlinear. In the next chapters we will look at several ways to handle nonlinear problems in a robust way, but for now we will do something simpler. If we know the approximate position of the robot than we can linearize these equations around that point. I could develop the generalized mathematics for this technique now, but instead let me just present the worked example to give context to that development.

Instead of computing $\mathbf{H}$ we will compute the partial derivative of $\mathbf{H}$ with respect to the robot's position $\mathbf{x}$. You are probably familiar with the concept of partial derivative, but if not, it just means how $\mathbf{H}$ changes with respect to the robot's position.

$$\frac{\partial \mathbf{h}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{(x_r - x_A)}{\sqrt{(x_r - x_A)^2 + (y_r - y_A)^2}} & 0 & \frac{(y_r - y_A)}{\sqrt{(x_r - x_A)^2 + (y_r - y_A)^2}} & 0 \\ \frac{(x_r - x_B)}{\sqrt{(x_r - x_B)^2 + (y_r - y_B)^2}} & 0 & \frac{(y_r - y_B)}{\sqrt{(x_r - x_B)^2 + (y_r - y_B)^2}} & 0 \end{bmatrix}$$

In a nutshell, the entry (0,0) contains the difference between the x coordinate of the robot and transmitter A's x coordinate divided by the distance between the robot and A. (2,0) contains the same, except for the y coordintates of the robot and transmitters. The bottom row contains the same computations, except for transmitter B. The 0 entries account for the velocity components of the state variables; naturally the range does notvide us with velocity.

The values in this matrix change as the robot's position changes, so this is no longer a constant; we will have to recompute it for every time step of the filter.

If you look at this you may realize that this is just a computation of x/dist and y/dist, so we can switch this to a trigometic form with no loss of generality:

$$\frac{\partial \mathbf{h}}{\partial \mathbf{x}} = \begin{bmatrix} -\cos\theta_A & 0 & -\sin\theta_A & 0 \\ -\cos\theta_B & 0 & -\sin\theta_B & 0 \end{bmatrix}$$

However, this raises a huge problem. We are no longer computing **H**, but **ΔH**, the change of **H**. If we passed this into our Kalman filter without altering the rest of the design the output would be nonsense. Recall, for example, that we multiply **Hx** to generate the measurements that would result from the given estimate of **x** But now that **H** is linearized around our position it contains the *change* in the measurement function.

We are forced, therefore, to use the *change* in **x** for our state variables. So we have to go back and redesign our state variables.

> Please note this is a completely normal occurance in designing Kalman filters. The textbooks present examples like this as *fait accompli*, as if it is trivially obvious that the state variables needed to be velocities, not positions. Perhaps once you do enough of these problems it would be trivially obvious, but at that point why are you reading a textbook? I find myself reading through a presentation multiple times, trying to figure out why they made a choice, finally to realize that it is because of the consequences of something on the next page. My presentation is longer, but it reflects what actually happens when you design a filter. You make what seem reasonable design choices, and as you move forward you discover properties that require you to recast your earlier steps. As a result, I am going to somewhat abandon my **step 1**, **step 2**, etc., approach, since so many real problems are not quite that straightforward.

If our state variables contain the velocities of the robot and not the position then how do we track where the robot is? We can't. Kalman filters that are linearized in this fashion use what is called a *nominal trajectory* - i.e. you assume a position and track direction, and then apply the changes in velocity and acceleration to compute the changes in that trajectory. How could it be otherwise? Recall the graphic showing the intersection of the two range circles - there are two areas of intersection. Think of what this would look like if the two transmitters were very close to each other - the intersections would be two very long cresent shapes. This Kalman filter, as designed, has no way of knowing your true position from only distance measurements to the transmitters. Perhaps your mind is already leaping to ways of working around this problem. If so, stay engaged, as later sections and chapters will provide you with these techniques. Presenting the full solution all at once leads to more confusion than insight, in my opinion.

So let's redesign our *state transition function.* We are assuming constant velocity and no acceleration, giving state equations of

$$\dot{x}' = \dot{x}\ddot{x}' = 0\dot{y}' = \dot{y}\dot{y}' = 0$$

This gives us the the *state transition function* of

$$\mathbf{F} = \begin{bmatrix}0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 0 & 0\end{bmatrix}$$

A final complication comes from the measurements that we pass in. **Hx** is now computing the *change* in the measurement from our nominal position, so the measurement that we pass in needs to be not the range to A and B, but the *change* in range from our measured range to our nomimal position.

There is a lot here to take in, so let's work through the code bit by bit. First we will define a function to compute $\frac{\partial \mathbf{h}}{\partial \mathbf{x}}$ for each time step.

```
In [15]: def H_of (pos, pos_A, pos_B):
             """ Given the position of our object at 'pos' in 2D, and two transmitters
             A and B at positions 'pos_A' and 'pos_B', return the partial derivative
             of H
             """

             from math import sin, cos, atan2

             theta_a = atan2(pos_a[1]-pos[1], pos_a[0] - pos[0])
             theta_b = atan2(pos_b[1]-pos[1], pos_b[0] - pos[0])
```

```
            return np.mat([[-cos(theta_a), 0, -sin(theta_a), 0],
                           [-cos(theta_b), 0, -sin(theta_b), 0]])
```

Now we need to create our simulated sensor.

```
In [16]: class DMESensor(object):
             def __init__(self, pos_a, pos_b, noise_factor=1.0):
                 self.A = pos_a
                 self.B = pos_b
                 self.noise_factor = noise_factor

             def range_of (self, pos):
                 """ returns tuple containing noisy range data to A and B
                 given a position 'pos'
                 """

                 ra = math.sqrt((self.A[0] - pos[0])**2 + (self.A[1] - pos[1])**2)
                 rb = math.sqrt((self.B[0] - pos[0])**2 + (self.B[1] - pos[1])**2)

                 return (ra + random.randn()*self.noise_factor,
                         rb + random.randn()*self.noise_factor)
```

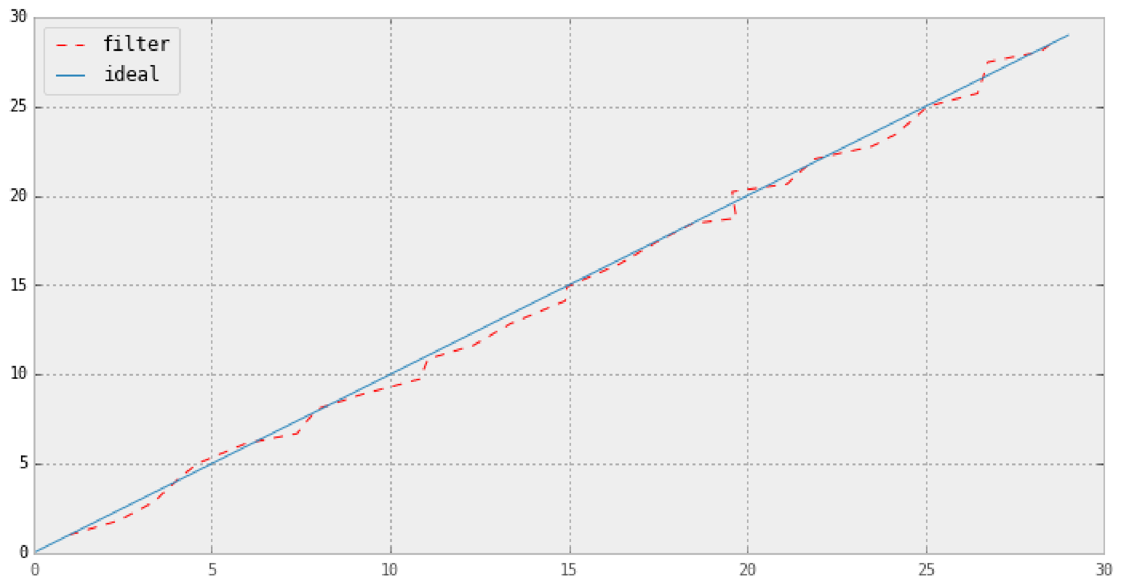
Now let's write the code to compute $\frac{\partial \mathbf{h}}{\partial \mathbf{x}}$.

```
In [17]: def dist(a,b):
             return math.sqrt ((a[0]-b[0])**2 + (a[1]-b[1])**2)

         def H_of (pos, pos_A, pos_B):
             dist_a = dist(pos, pos_A)
             dist_b = dist(pos, pos_B)

             return np.mat([[0, (pos[0]-pos_A[0])/dist_a, 0, (pos[1]-pos_A[1])/dist_a],
                            [0, (pos[0]-pos_B[0])/dist_b, 0, (pos[1]-pos_B[1])/dist_b]])
             '''
             return np.mat([[(pos[0]-pos_A[0])/dist_a, 0, (pos[1]-pos_A[1])/dist_a,0],
                            [(pos[0]-pos_B[0])/dist_b, 0, (pos[1]-pos_B[1])/dist_b,0]])
             '''
```

Finally, we are ready for the Kalman filter code. I will position the transmitters at x=-100 and 100, both with y=-20. This gives me enough space to get good triangulation from both as the robot moves. I will start the robot at (0,0) and move by (1,1) each time step.

```
In [18]: import math

         pos_a = (100,-20)
         pos_b = (-100, -20)

         f1 = KalmanFilter(dim=4)

         f1.F = np.mat ([[0, 1, 0, 0],
                         [0, 0, 0, 0],
                         [0, 0, 0, 1],
                         [0, 0, 0, 0]])
```

```
f1.B = 0.

f1.R = np.eye(2) * 1.
f1.Q = np.eye(4) * .1

f1.x = np.mat([1,0,1,0]).T
f1.P = np.eye(4) * 5.

# initialize storage and other variables for the run
count = 30
xs, ys = [],[]
pxs, pys = [],[]

# create the simulated sensor
d = DMESensor (pos_a, pos_b, noise_factor=3.)

# pos will contain our nominal position since the filter does not
# maintain position.
pos = [0,0]

for i in range(count):
    # move (1,1) each step, so just use i
    pos = [i,i]

    # compute the difference in range between the nominal track and measured
    # ranges
    ra,rb = d.range_of(pos)
    rx,ry = d.range_of((i+f1.x[0,0], i+f1.x[2,0]))
    z = np.mat([[ra-rx],[rb-ry]])

    # compute linearized H for this time step
    f1.H = H_of (pos, pos_a, pos_b)

    # store stuff so we can plot it later
    xs.append (f1.x[0,0]+i)
    ys.append (f1.x[2,0]+i)
    pxs.append (pos[0])
    pys.append(pos[1])

    # perform the Kalman filter steps
    f1.predict ()
    f1.update(z)


p1, = plt.plot (xs, ys, 'r--')
p2, = plt.plot (pxs, pys)
plt.legend([p1,p2], ['filter', 'ideal'], 2)
plt.show()
```



## 8.5 Linearizing the Kalman Filter

Now that we have seen an example of linearizing the Kalman filter we are in a position to better understand the math.

We start by assuming some function **f**

```
In [18]:
```

```
In [18]:
```

```
In [18]:
```

```
In [18]:
```

```
In [18]:
```

# Chapter 9

# The Extended Kalman Filter

```
In [1]: #format the book
        %matplotlib inline
        from __future__ import division, print_function
        import matplotlib.pyplot as plt
        import book_format
        book_format.load_style()
```

```
Out[1]: <IPython.core.display.HTML at 0x2904910>
```

The Kalman filter that we have developed to this point is extremely good, but it is also limited. Its derivation is in the linear space, and hence it only works for linear problems. Let's be a bit more rigorous here. You can, and we have in this book, apply the Kalman filter to nonlinear problems. For example, in the g-h filter chapter we explored using a g-h filter in a problem with constant acceleration. It 'worked', in that it remained numerically stable and the filtered output did track the input, but there was always a lag. It is easy to prove that there will always be a lag when $\ddot{\mathbf{x}} > 0$. The filter no longer produces an optimal result. If we make our time step arbitrarily small we can still handle many problems, but typically we are using Kalman filters with physical sensors and solving real-time problems. Either fast enough sensors do not exist, are prohibitively expensive, or the computation time required is excessive. It is not a workable solution.

The early adopters of Kalman filters were the radar people, and this fact was not lost on them. Radar is inherently nonlinear. 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:

$$x = \sqrt{slant^2 - altitude^2}$$

So shortly after the Kalman filter was enthusiastically taken up by the radar industry people began working on how to extend the Kalman filter into nonlinear problems. It is still an area of ongoing research, and in the Unscented Kalman filter chapter we will implement a powerful, recent result of that research. But in this chapter we will cover the most common form, the Extended Kalman filter, or EKF. Today, most real world "Kalman filters" are actually EKFs. The Kalman filter in your car's and phone's GPS is an EKF, for example.

## 9.1 The Problem with Nonlinearity

You may not realize it, but the only math you really know how to do is linear math. Equations of the form

$$A\mathbf{x} = \mathbf{b}$$

.

That may strike you as hyperbole. After all, in this book we have integrated a polynomial to get distance from velocity and time: We know how to integrate a polynomial, for example, and so we are able to find the closed form equation for distance given velocity and time:

$$\int (vt + v_0)\, dt = \frac{a}{2}t^2 + v_0 t + d_0$$

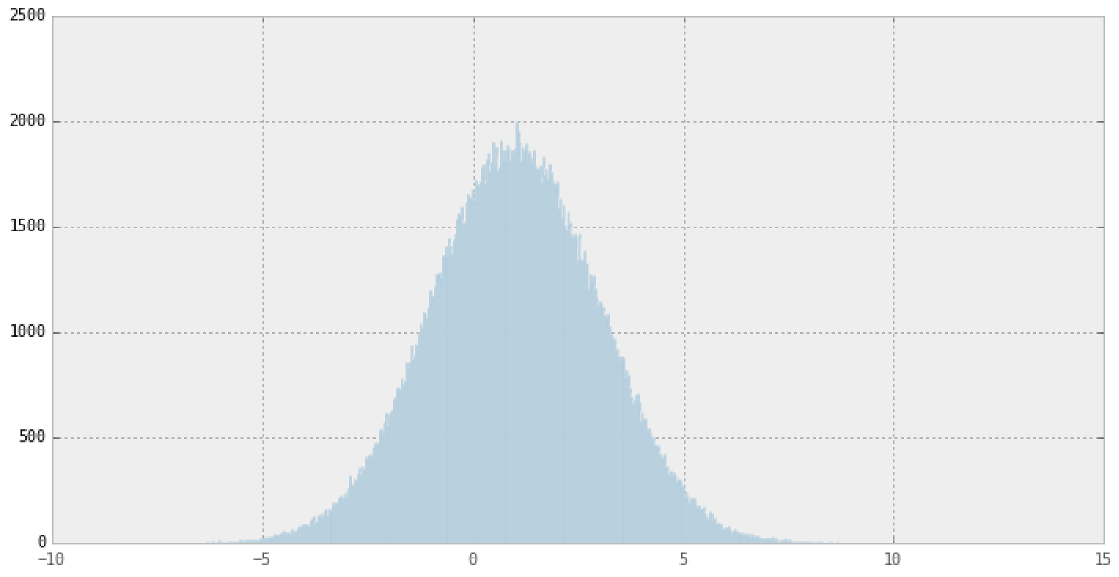
That's nonlinear. But it is also a very special form. You spent a lot of time, probably at least a year, learning how to integrate various terms, and you still can not integrate some arbitrary equation - no one can. We don't know how. If you took freshman Physics you perhaps remember homework involving sliding frictionless blocks on a plane and other toy problems. At the end of the course you were almost entirely unequipped to solve real world problems because the real world is nonlinear, and you were taught linear, closed forms of equations. It made the math tractable, but mostly useless.

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 multipy it using linear algebra we get another Gaussian equation as a result. That is very rare. $\sin x * \sin y$ does not yield a $\sin(\cdot)$ as an output.

## 9.2 The Effect of Nonlinear Transfer Functions on Gaussians

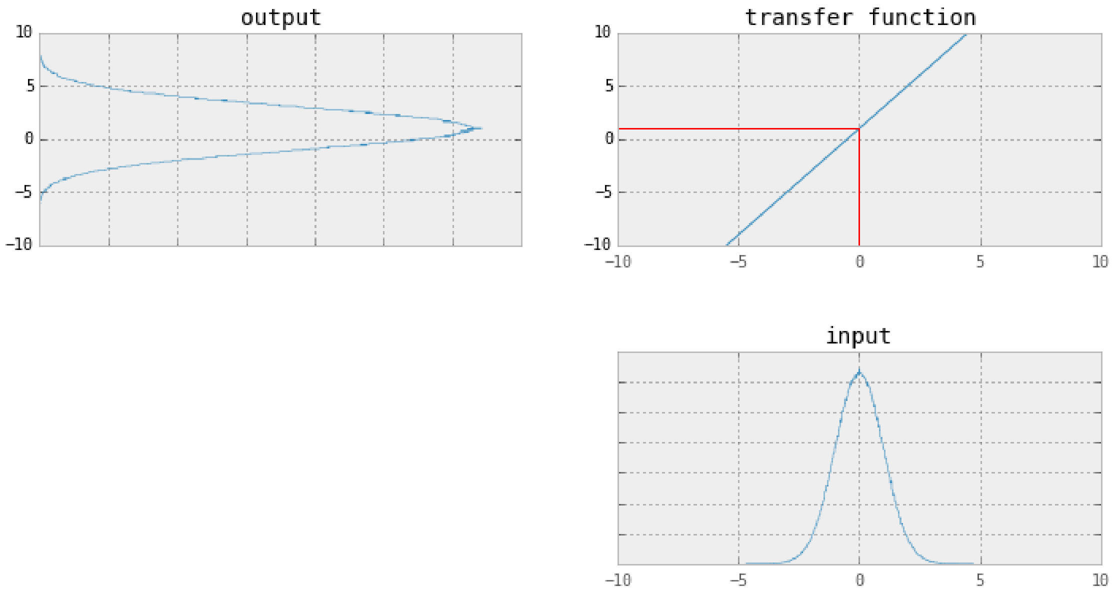
Unfortunately Gaussians are not closed under an arbitrary nonlinear function. Recall the equations of the Kalman filter - at each step of its evolution we do things like pass the covariances through our process function to get the new covariance at time $k$. Our process function was always linear, so 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 lets do this with sampling. I will generate 500,000 points on the Gaussian curve, pass it through the function, and then plot the results. I will do it this way because the next example will be nonlinear, and we will have no way to compute this analytically.

```
In [2]: import numpy as np
        from numpy.random import normal

        data = normal(loc=0.0, scale=1, size=500000)
        ys = 2*data + 1

        plt.hist(ys,1000)
        plt.show()
```



This is an unsuprising result. The result of passing the Gaussian through $f(x) = 2x + 1$ is another Gaussian centered around 1. Let's look at the input, transfer function, and output at once.

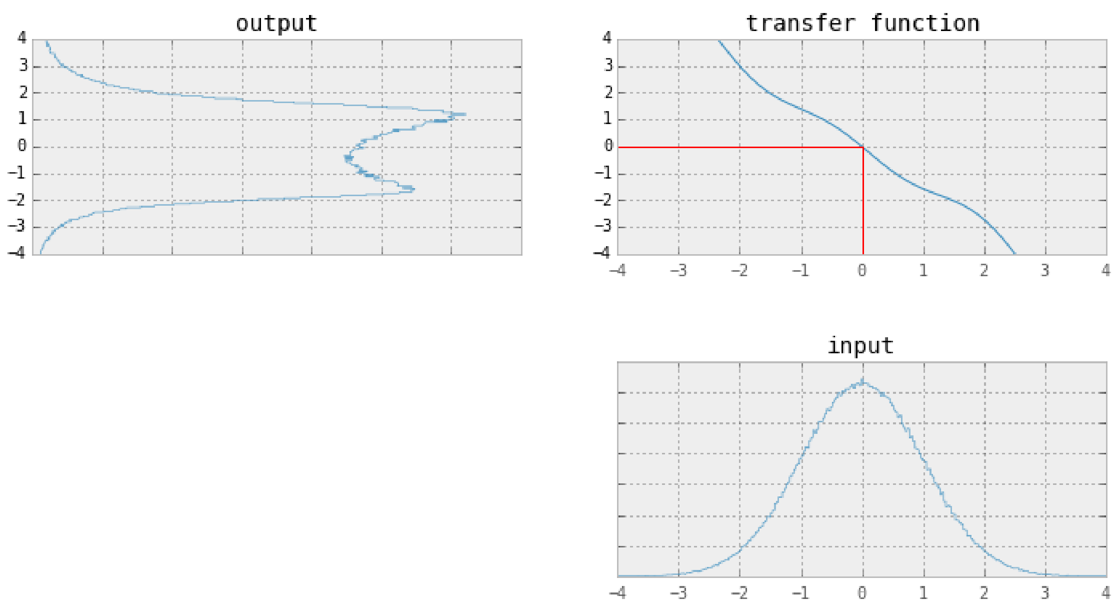
```
In [3]: from nonlinear_plots import plot_transfer_func

        def g(x):
            return 2*x+1

        plot_transfer_func (data, g, lims=(-10,10), num_bins=300)
```



The plot labelled 'input' is the histogram of the original data. This is passed through the transfer function $f(x) = 2x + 1$ which is displayed in the chart to the upper right. 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 left. The output looks like a Gaussian, and is in fact a Gaussian. We can see that it is altered -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.

Now let's look at a nonlinear function and see how it affects the probability distribution.

```
In [4]: from nonlinear_plots import plot_transfer_func

        def g(x):
            return (np.cos(4*(x/2+0.7)))*np.sin(0.3*x)-1.6*x

        plot_transfer_func (data, g, lims=(-4,4), num_bins=300)
```



This result may be somewhat suprising to you. The transfer function looks “fairly” linear - it is pretty close to a straight line, but the probability distribution of the output is completely different from a Gaussian. Recall the equations for multiplying two univariate Gaussians:

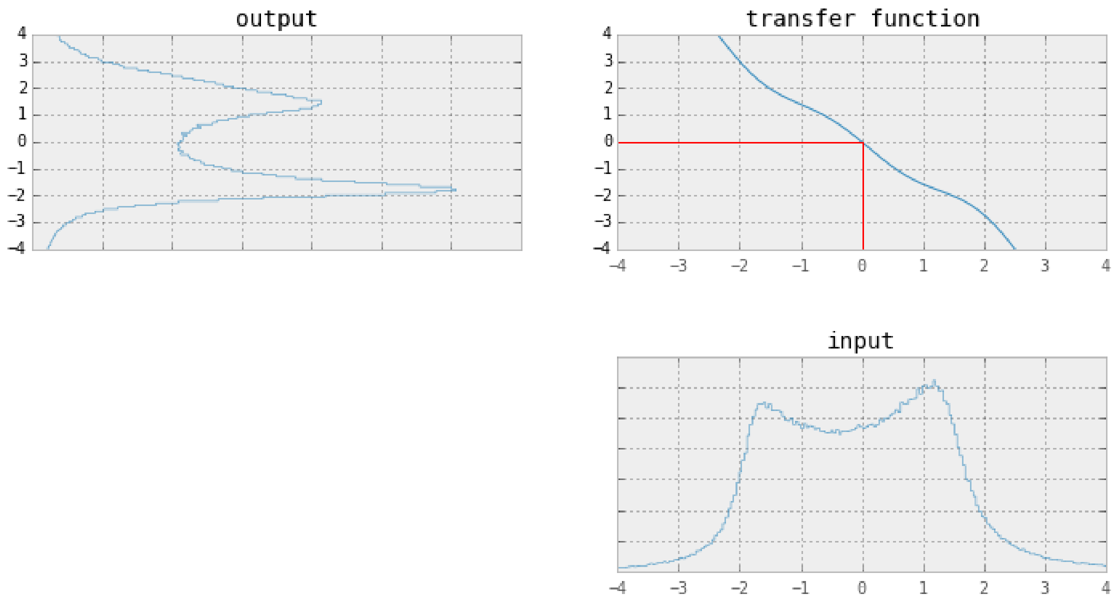
$$\mu = \frac{\sigma_1^2\mu_2 + \sigma_2^2\mu_1}{\sigma_1^2+\sigma_2^2}, \, \sigma = \frac{1}{\frac{1}{\sigma_1^2}+\frac{1}{\sigma_2^2}}$$

These equations do not hold for non-Gaussians, and certainly do not hold for the probability distribution shown in the ‘output’ chart above.

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 transfer function again, simulating the next step time step of the Kalman filter.

**process function? what is the correct name**

```
In [5]: y=g(data)
        plot_transfer_func (y, g, lims=(-4,4), num_bins=300)
```



As you can see the probability function is futher distorted from the original Gaussian. However, the graph is still somewhat symmetric around 0, let's see what the mean is.

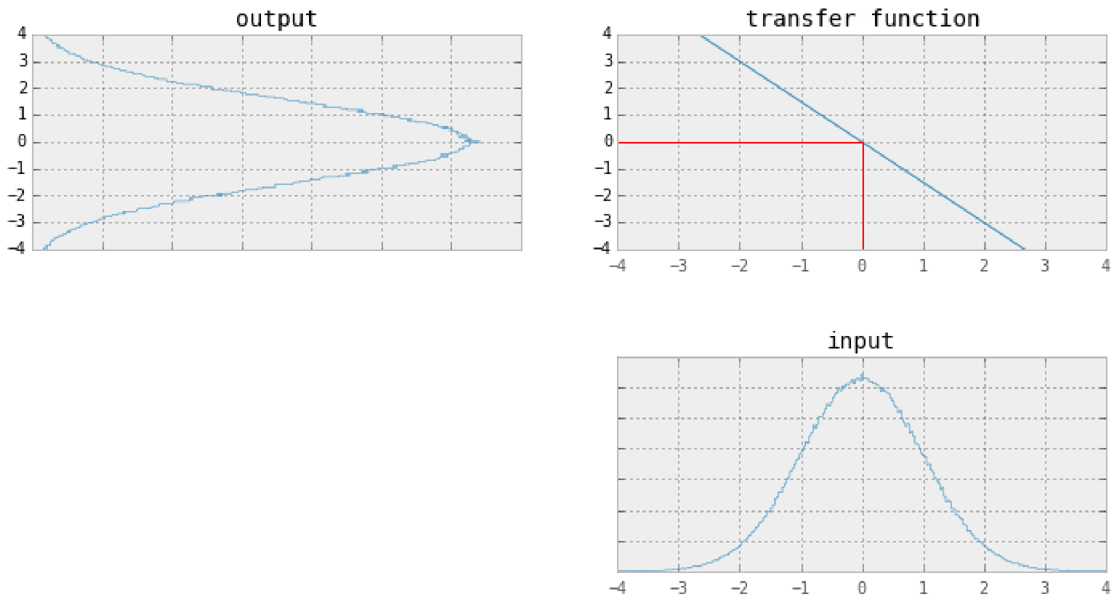
```
In [6]: print ('input  mean, variance: %.4f, %.4f'% (np.average(data), np.std(data)**2))
        print ('output mean, variance: %.4f, %.4f'% (np.average(y), np.std(y)**2))
```

```
input  mean, variance: 0.0013, 0.9985
output mean, variance: -0.0293, 2.2394
```

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 [7]: def h(x): return -1.5*x
        plot_transfer_func (data, h, lims=(-4,4), num_bins=300)
        out = h(data)
        print ('output mean, variance: %.4f, %.4f'% (np.average(out), np.std(out)**2))
```



```
output mean, variance: -0.0019, 2.2466
```

Although the shapes 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 [8]: out = h(data)
        out2 = g(data)

        for i in range(10):
            out = h(out)
            out2 = g(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: -0.1088, 7470.5861
nonlinear output mean, variance: -2.0348, 26163.5214
```

Unfortunately we can see that the nonlinear version is not stable. We have drifted significantly from the mean of 0, and the variance is half an order of magnitude larger.

# Chapter 10

# Unscented Kalman Filters

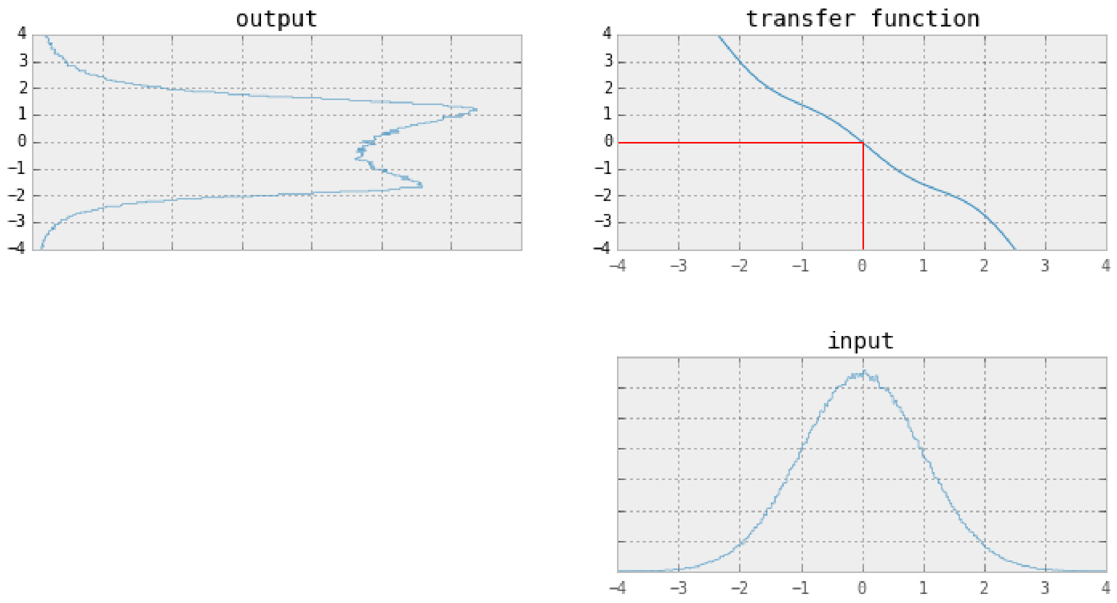
```
In [1]: #format the book
        %matplotlib inline
        from __future__ import division, print_function
        import matplotlib.pyplot as plt
        import book_format
        book_format.load_style()
```

```
Out[1]: <IPython.core.display.HTML at 0xece990>
```

In the previous chapter we developed the Extended Kalman Filter to allow us to use the Kalman filter with nonlinear problems. It is by far the most commonly used Kalman filter. However, it requires that you be able to analytically derive the Jacobian blah blah limp prose.

However, for many problems finding the Jacobian is either very difficult or impossible. Futhermore, being an approximation, the EKF can diverge. For all these reasons there is a need for a different way to approximate the Gaussian being passed through a nonlinear transfer function. In the last chapter I showed you this plot:

```
In [2]: from nonlinear_plots import plot_transfer_func
        from numpy.random import normal
        import numpy as np

        data = normal(loc=0.0, scale=1, size=500000)

        def g(x):
            return (np.cos(4*(x/2+0.7)))*np.sin(0.3*x)-1.6*x

        plot_transfer_func (data, g, lims=(-4,4), num_bins=300)
```



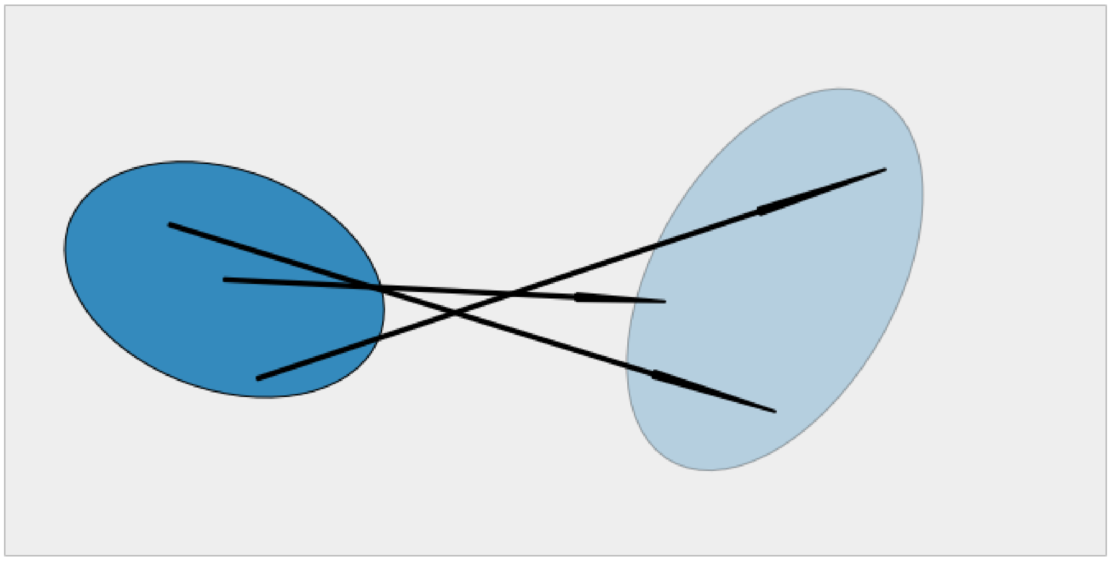
I generated this by taking 500,000 samples from the input, passing it through the nonlinear tranform, and building a histogram of the result. From that histogram we can then compute a mean and a variance that we compared to the output of the EKF.

It has perhaps occured to you that this sampling process constitutes a solution to our problem. This is called a 'monte carlo' approach, and it used by some Kalman filter designs, such as the *Ensemble filter*. Sampling requires no specialized knowledge programming, and does not require a closed form solution. No matter how nonlinear or poorly behaved the tranfer function is, as long as we sample with enough points we will build an accurate output distribution.

"Enought points" is the rub. The graph above was created with 500,000 points, and the output is still not smooth. You wouldn't need to use that many points to get a reasonable estimate of the mean and variance, but it will require many points. What's worse, this is only for 1 dimension. In general, the number of points required increases by the power of the number of dimensions. If you need 50 points for 1 dimension, you need $50^2$ for two dimensions, $50^3$ for three dimensions, and so on. So while this approach does work, it is very computationally expensive. The Unscented Kalman filter, the topic of this chapter, uses a somewhat similar technique but reduces the amount of computation needed by a drastic amount.

It is somewhat hard to understand some aspects of this problem by looking at the histogram, so consider this alternative representation, this time for 2 variables/dimensions.

```
In [3]: import ukf_internal
        ukf_internal.show_2d_transform()
```



Here on the left we show an ellipse depicting the $1\sigma$ distribution of two variables. The arrows show how three 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 - if we were to sample, say, a million points the shape of the points might be very far from an ellipse.

## 10.1 Choosing Sigma Points

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 tranform as we want to find a generalized algorithm. For reasons that come clear in the next section, we will call these points *sigma points*.

Let's consider the simplest possible case, and see if it offers any insight. The simplest possible system is *identity* - the transformation does not alter the input. It should be clear that if this does not work then the filter will never converge.

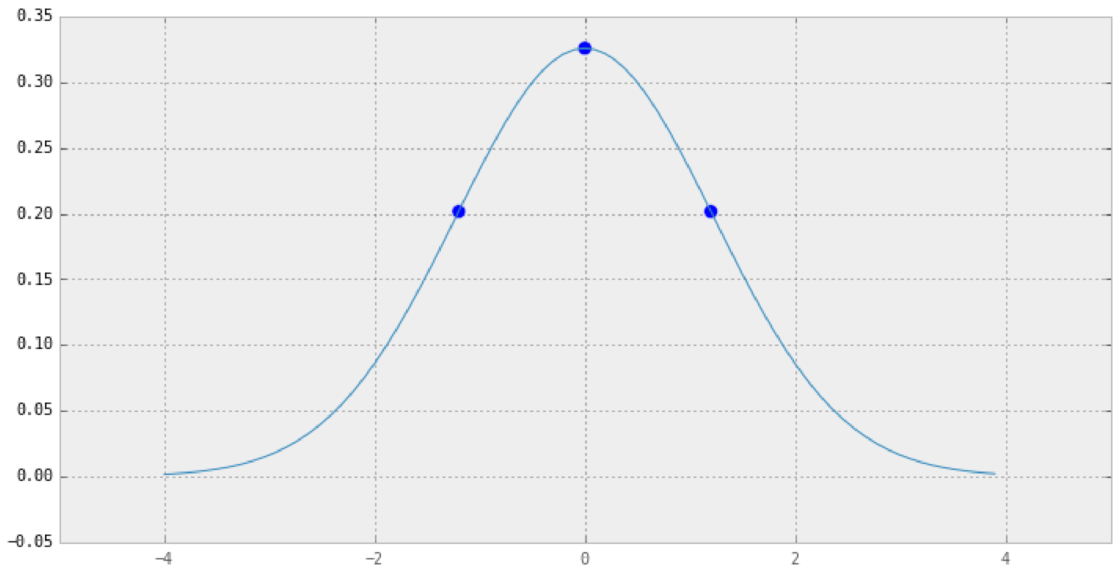
The fewest number of points that we can use is one per dimension. This is the number that the linear Kalman filter uses. The input to a Kalman filter for the distribution $\mathcal{N}(\mu, \sigma^2)$ is just $\mu$ itself. So while this works for the linear case, it is not a good answer for the nonlinear case.

If we were to pass some value $\mu + \Delta$ instead, the identity system would not converge, so this is not a possible algorithm. Since we cannot set our one point sample to $\mu$, or any value that is not $\mu$, we must conclude that a one point sample will not work.

So, what is the next lowest number we can choose? 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. 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. I recognize that this is rather vague, but I don't want to spend a lot of time on a scheme that doesn't work.

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 [4]: ukf_internal.show_3_sigma_points()
```



For this to work for identity we will want the sums of the weights to equal one. We can always come up with counterexamples, but in general if the sum is greater or less than one the sampling will not yield the correct output. Given that, we then have to select *sigma points* $\mathcal{X}$ and their corresponding weights so that they compute to the mean and variance of the input Gaussian. So we can write

$$1 = \sum_i w_i \tag{1}$$

$$\mu = \sum_i w_i \mathcal{X}_i \tag{2}$$

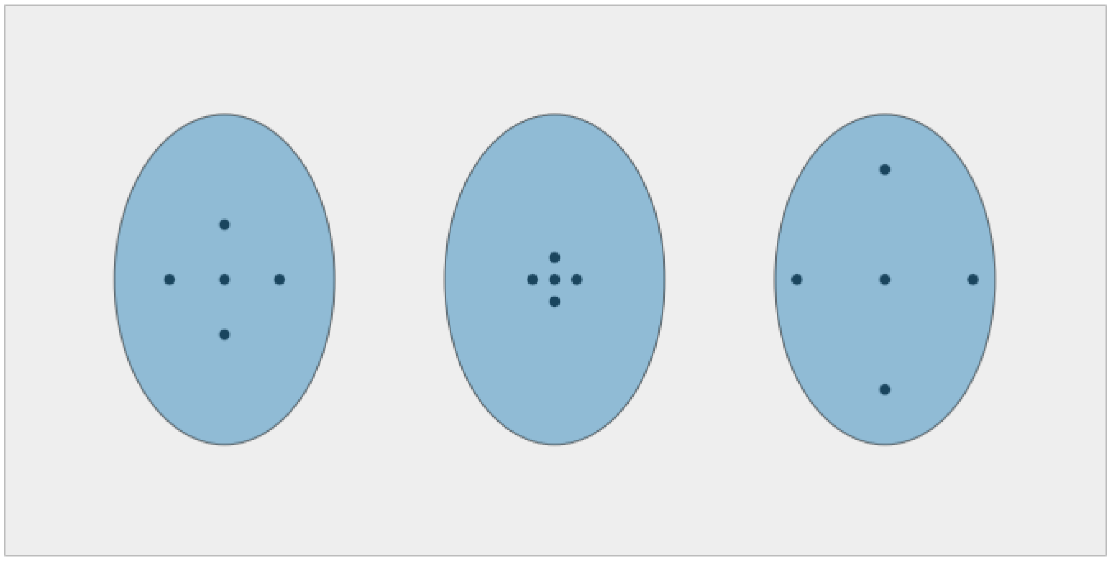
$$\Sigma = \sum_i w_i (\mathcal{X}_i - \mu)(\mathcal{X}_i - \mu)^T \tag{3}$$

If we look at this is should be clear that there is no one unique answer - the problem is unconstrained. For example, if you choose a smaller weight for the point at the mean for the input, you could compensate by choosing larger weights for the rest of the $\mathcal{X}$, and vice versa if you chose a larger weight for it. 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.

Methods for selecting these sigma points is it own topic. In the next section I will develop the most typically used method in practice. It has the virtue of requiring only 3 sigma points per dimension, which is far lower than we might expect to provide good results. Despite the low number of points, the computations for the weight selections are very easy and efficient, and the numerical performance of the filter is as good as, and usually better than the EKF.

But before we go on I want to make sure the idea is clear. We are choosing 3 points for each dimension in our covariances. That choice is *entirely deterministic*. Below are three different examples for the same covariance ellipse.

```
In [5]: ukf_internal.show_sigma_selections()
```



Note that while I chose the points to lie along the major and minor axis of the ellipse, nothing in the constraints above require me to do that; however, it is fairly typical to do this. Furthermore, in each case I show the points evenly spaced; again, the constraints above do not require that. However, the technique that we develop in the next section *does* do this. It is a reasonable choice, after all; if we want to accurately sample our input it makes sense to sample in a symmetric manner.

## 10.2 The Unscented Transform

So our desire is to have an algorithm for selecting sigma points based on some criteria. Maybe we know something about our nonlinear problem, and we know we want our sigma points to be very close together, or very far apart. Or through experimentation we decide that a certain choice of basis vectors from our hyperellipse are the best axis to choose our sigma points from. But we want this to be an algorithm - we don't want to have to hard code in a specific selection algorithm for each different problem. So we are going to want to be able to set some parameters to tell the algorithm how to automatically select the points and weights for us. That may seem a bit abstract, so let's just launch into it, and try to develop an intuitive understanding as we go.

Our first choice is always going to be the mean of our input. We will number this $\mathcal{X}_0$. So,

$$\mathcal{X}_0 = \mu$$

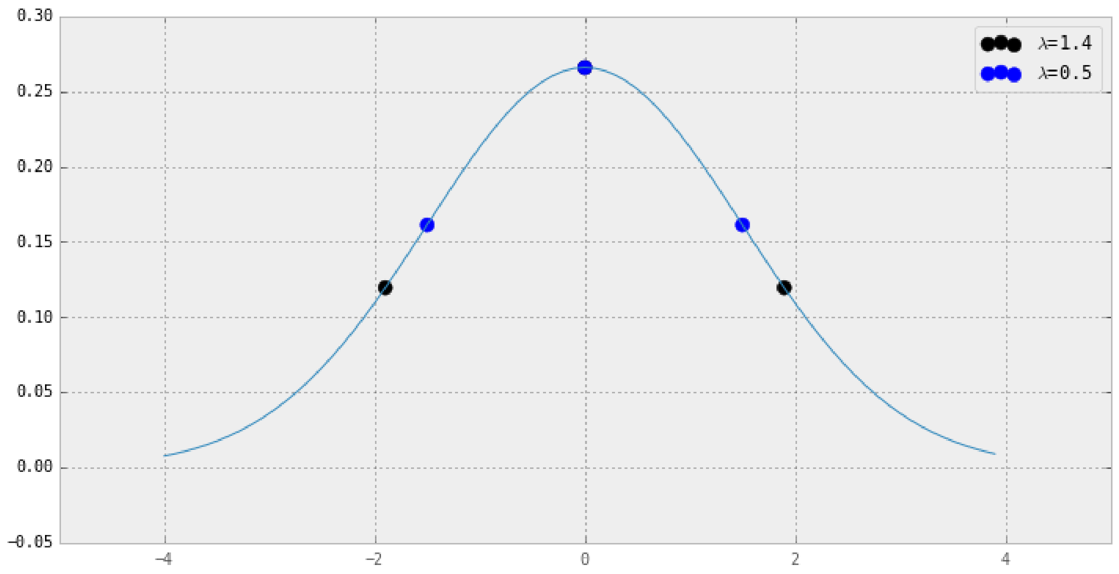
So for each dimension we need to select 2 more points. We want them to be symmetric around the mean so that for the linear case they cancel out and we are just left with the mean as the result. Here is how we are going to do that:

$$\begin{aligned} \mathcal{X}_i &= \mu + (\sqrt{(n+\lambda)\Sigma})_i && \text{for i=1..n} \\ \mathcal{X}_i &= \mu - (\sqrt{(n+\lambda)\Sigma})_{i-n} && \text{for i=n+1..2n} \end{aligned}$$

Let's talk through this. For the moment, just think of this in one dimension, so we can ignore the subscript $i$. $n$ is just our dimensionality, and $\lambda$ is a scaling factor that controls how far away from the mean we want the points to be. A larger lambda will choose points further away from the mean, and a smaller lambda will choose points nearer the mean. So in one dimension we get something like:

```
In [6]: import stats
        import math
```

```python
# generate the Gaussian data

xs = np.arange(-4, 4, 0.1)
mean = 0
sigma = 1.5
ys = [stats.gaussian(x, mean, sigma*sigma) for x in xs]

def sigma_points (mean, sigma, lambda_):
    sigma1 = mean + math.sqrt ((1+lambda_)*sigma)
    sigma2 = mean - math.sqrt ((1+lambda_)*sigma)
    return mean, sigma1, sigma2

#generate our samples
lambda_ = 1.4
x0,x1,x2 = sigma_points (mean,sigma,lambda_)

samples = [x0,x1,x2]
for x in samples:
    p1 = plt.scatter ([x], [stats.gaussian(x, mean, sigma*sigma)], s=80, color='k')

lambda_ = 0.5
x0,x1,x2 = sigma_points (mean,sigma,lambda_)

samples = [x0,x1,x2]
for x in samples:
    p2 = plt.scatter ([x], [stats.gaussian(x, mean, sigma*sigma)], s=80, color='b')

plt.legend([p1,p2],['$\lambda$=1.4', '$\lambda$=0.5'])
plt.plot(xs, ys)
plt.show()
```



Here I have plotted two different choices for lambda - 05 and 1.4 - to show how lambd affects the distribution of the points.

Perhaps the reason for calling these sigma points is clear. While we are not selecting exactly $1\sigma$ for the position of our points, we are scaling by some factor of $\sigma$.

On to larger dimensions ($n > 1$). The term $(\sqrt{(n+\lambda)\Sigma})_i$ has to be a matrix because $\Sigma$ is a matrix. The subscript $i$ is choosing the column vector of the matrix. And thus the square root is not the square root of a scalar, but of a matrix.

What is the 'square root of a matrix'? The usual definition is that the square root of a matrix $\Sigma$ is just the matrix $S$ that, when multiplied by itself, yields $\Sigma$.

$$\text{if } \Sigma = SS$$

$$\text{then } S = \sqrt{\Sigma}$$

However, there is an alternative definition, and we will chose that because it has numerical properties that makes it much easier for us to compute its value. We can alternatively define the square root as a matrix S, when multiplied by its transpose, returns $\Sigma$:

$$\Sigma = SS^T$$

If this makes you uncomfortable, I advise not worrying about it. We are just defining an algorithm for choosing our sigma points; if the 'square' root in method of computation ($SS$) is slightly different than the other case ($SSˆT), it shouldn't really matter. Plus, this still works for the one dimensional case, as the transpose of a scalar is just the scalar.

I will derive the math for computing this in a later section in this chapter. For now I will say that this is a well worn area of linear algebra, and performing this 'square root' uses something called the Cholesky decomposition. This is a library procedure that is available in any linear algebra library. For example, numpy provides it in `numpy.linalg.cholesky()`, and we will be using that function in our code. If your language of choice is Fortran, C, C++, or the like the standard libraries like LAPACK also provide this routine. And, of course, matlab provides `chol()`, which does the same thing.

## 10.3 Implementation

So let's just implement this algorithm. First, let's write the code to compute the mean and covariance given the sigma points. Unfortunately, this requires a short diversion into the intricacies of numpy's data types. Through historical accidents, numpy provides several data structures to represent matrices, and they do not 'play well' with each other. Most sources recommend using the type `numpy.array`, however, it is quite cumbersome to use for intensive linear algebra. Since everything I do in this book is linear algebra, I choose to use `numpy.matrix` instead. Fortunately, numpy provides `numpy.asmatrix()`. It will convert a variable to `numpy.matrix` if it is not already one; better yet, it will turn a scalar into a 1×1 matrix. The result is that we can pass scalars, `numpy.array`'s, or `numpy.matrix`s into my code and it will work.

So we will store the sigma points and weights in matrices, like so:

$$weights = \begin{bmatrix} w_1 & w_2 & \dots & w_n \end{bmatrix}$$
$$sigmas = \begin{bmatrix} \mathcal{X}_{0,0} & \mathcal{X}_{0,1} & \dots & \mathcal{X}_{0,n} \\ \mathcal{X}_{1,0} & \mathcal{X}_{1,1} & \dots & \mathcal{X}_{1,n} \\ \mathcal{X}_{2,0} & \mathcal{X}_{2,1} & \dots & \mathcal{X}_{2,n} \end{bmatrix}$$

In other words, each column contains the 3 sigma points for one dimension in our problem. The $0th$ sigma point is always the mean, so first row of sigma's contains the mean of each of our dimensions. The second row contains the $\mu + (\sqrt{(n+\lambda)\Sigma})$ term, and the third row contains the $\mu - (\sqrt{(n+\lambda)\Sigma})$ term.

```
In [7]: import numpy as np

        def unscented_transform (Xi, W, NoiseCov=None):
            """ computes the unscented transform of a set of signma points and weights.
            returns the mean and covariance in a tuple
```

```
    """
    W = np.asmatrix(W)
    Xi = np.asmatrix(Xi)

    n, kmax = Xi.shape

    # initialize results to 0
    mu  = np.mat (np.zeros((n,0)))
    cov = np.mat (np.zeros((n,n)))

    for k in range (kmax):
        mu += W[0,k] * Xi[:,k]

    for k in range (kmax):
        cov += W[0,k]*(Xi[:,k]-xm) * (Xi[:,k]-xm).T

    return (mu, cov)
```

Let's review a bit of numpy nomenclature. The odd looking

```
Xi[:,i]
```

term is merely saying 'take the *ith* column of Xi'. So This produces a $1 \times n$ column vector, like so:

$$\begin{bmatrix} \mathcal{X}_{0,i} \\ \mathcal{X}_{1,i} \\ \mathcal{X}_{2,i} \\ \vdots \\ \mathcal{X}_{n,i} \end{bmatrix}$$

This is confusing if you haven't seen it before so I will belabor this point. Read this as `Xi[A, B]`, where `A` indexes the row, and `B` the column. In this case the row value is ':', which is Python's way to taking a *slice*. For example, for a one dimensional list `x`, `x[3 : 5]` returns a list containing `x[3]` and `x[4]`. If you leave one of the slice elements out it just selects everything on that side of the slice, so `x[3 :]` would return a list of elements number 2 to the last element in `x`. Therefore, `x[:]` just returns `x` itself - it asks for all elements in `x`. So, in a form like `Xi[:, i]` we take all rows from the ith column; effectively, a 1xn column vector.

We write this way in numpy for two reasons. First, it is very close to the mathematical notation our equations use. Second, and more importantly, when you write this way numpy uses C routines to implement the functionality, making the code much faster. Avoiding for loops in numerical code is 'Pythonic'. If you are used to packages like Matlab you are already familar with notation; if you are coming from a language like C recognize I could have written this loop:

```
for k in range (kmax):
    mu += W[0,k] * Xi[:,k]
```

as:

```
for k in range (kmax):
   for i in range (n):
    mu += W[0,k] * Xi[i,k]
```

So if you look at this code you should see that it is just performing the computations

$$\mu = \sum_i w_i \mathcal{X}_i$$

$$\Sigma = \sum_i w_i (\mathcal{X}_i - \mu)(\mathcal{X}_i - \mu)^T$$

Even if you are not a strong numpy programmer it should be reasonably clear that this is not a difficult bit of code. Furthermore, as with our previous Kalman filter codes, we just need to write it once - we will call this function for *any* problem we are solving, and it will perform the Unscented transform for us.

We are half done! Now we need to write the function that selects the sigma points based on this term $\sqrt{(n+\lambda)\Sigma}$. It is a bit longer, but still straightforward.

```
In [8]: def sigma_points (mu, P, kappa):
            """ Computes the sigma points and weights for an unscented Kalman filter.
            xm are the means, and P is the covariance. kappa is an arbitrary constant
            constant. Returns tuple of the sigma points and weights.

            Works with scalar, array, and matrix inputs:
            sigma_points (5, 9, 2) # mean 5, covariance 9
            sigma_points ([5, 2], 9*eye(2), 2) # means 5 and 2, covariance 9I
            """

            mu  = asmatrix(mu)
            cov = asmatrix(cov)

            n = size(mu)

            # initialize to zero
            Xi = asmatrix (zeros((n,2*n+1)))
            W  = asmatrix (zeros(2*n+1))


            # all weights are 1/ 2(n+kappa)) except the first one.
            W[0,1:] = 1. / (2*(n+kappa))
            W[0,0] = float(kappa) / (n + kappa)


            # use cholesky to find matrix square root of (n+kappa)*cov
            #  U'*U = (n+kappa)*P
            U = asmatrix (cholesky((n+kappa)*cov))

            # mean is in location 0.
            Xi[:,0] = mu

            for k in range (n):
                Xi[:,k+1] = mu + U[:,k]

            for k in range (n):
                Xi[:, n+k+1] = mu - U[:,k]

            return (Xi, W)
```