

09-Nonlinear-Filtering

November 5, 2022

[Table of Contents](#)

1 Nonlinear Filtering

```
[24]: %matplotlib inline
```

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

```
[25]: <IPython.core.display.HTML object>
```

1.1 Introduction

The Kalman filter that we have developed uses linear equations, and so the filter can only handle linear problems. But the world is nonlinear, and so the classic filter that we have been studying to this point can have very limited utility.

There can be nonlinearity in the process model. Suppose we want to track an object falling through the atmosphere. The acceleration of the object depends on the drag it encounters. Drag depends on air density, and the air density decreases with altitude. In one dimension this can be modelled with the nonlinear differential equation

$$\ddot{x} = \frac{0.0034ge^{-x/22000}\dot{x}^2}{2\beta} - g$$

A second source of nonlinearity comes from the measurements. For example, radars measure the slant range to an object, and we are typically interested in the aircraft's position over the ground. We invoke Pythagoras and get the nonlinear equation:

$$x = \sqrt{\text{slant}^2 - \text{altitude}^2}$$

These facts were not lost on the early adopters of the Kalman filter. Soon after Dr. Kalman published his paper people began working on how to extend the Kalman filter for nonlinear problems.

It is almost true to state that the only equation anyone knows how to solve is $\mathbf{Ax} = \mathbf{b}$. We only really know how to do linear algebra. I can give you any linear set of equations and you can either solve it or prove that it has no solution.

Anyone with formal education in math or physics has spent years learning various analytic ways to solve integrals, differential equations and so on. Yet even trivial physical systems produce equations that cannot be solved analytically. I can take an equation that you are able to integrate, insert a log term, and render it insolvable. This leads to jokes about physicists stating “assume a spherical cow on a frictionless surface in a vacuum...”. Without making extreme simplifications most physical problems do not have analytic solutions.

How do we do things like model airflow over an aircraft in a computer, or predict weather, or track missiles with a Kalman filter? We retreat to what we know: $\mathbf{Ax} = \mathbf{b}$. We find some way to linearize the problem, turning it into a set of linear equations, and then use linear algebra software packages to compute an approximate solution.

Linearizing a nonlinear problem gives us inexact answers, and in a recursive algorithm like a Kalman filter or weather tracking system these small errors can sometimes reinforce each other at each step, quickly causing the algorithm to spit out nonsense.

What we are about to embark upon is a difficult problem. There is not one obvious, correct, mathematically optimal solution anymore. We will be using approximations, we will be introducing errors into our computations, and we will forever be battling filters that *diverge*, that is, filters whose numerical errors overwhelm the solution.

In the remainder of this short chapter I will illustrate the specific problems the nonlinear Kalman filter faces. You can only design a filter after understanding the particular problems the nonlinearity in your problem causes. Subsequent chapters will then teach you how to design and implement different kinds of nonlinear filters.

1.2 The Problem with Nonlinearity

The mathematics of the Kalman filter is beautiful in part due to the Gaussian equation being so special. It is nonlinear, but when we add and multiply them we get another Gaussian as a result. That is very rare. $\sin x * \sin y$ does not yield a sin as an output.

What I mean by linearity may be obvious, but there are some subtleties. The mathematical requirements are twofold:

- additivity: $f(x + y) = f(x) + f(y)$
- homogeneity: $f(ax) = af(x)$

This leads us to say that a linear system is defined as a system whose output is linearly proportional to the sum of all its inputs. A consequence of this is that to be linear if the input is zero then the output must also be zero. Consider an audio amp - if I sing into a microphone, and you start talking, the output should be the sum of our voices (input) scaled by the amplifier gain. But if the amplifier outputs a nonzero signal such as a hum for a zero input the additive relationship no longer holds. This is because linearity requires that $amp(voice) = amp(voice + 0)$. This clearly should give the same output, but if $amp(0)$ is nonzero, then

$$\begin{aligned} amp(voice) &= amp(voice + 0) \\ &= amp(voice) + amp(0) \\ &= amp(voice) + non_zero_value \end{aligned}$$

which is clearly nonsense. Hence, an apparently linear equation such as

$$L(f(t)) = f(t) + 1$$

is not linear because $L(0) = 1$. Be careful!

1.3 An Intuitive Look at the Problem

I particularly like the following way of looking at the problem, which I am borrowing from Dan Simon's *Optimal State Estimation* [1]. Consider a tracking problem where we get the range and bearing to a target, and we want to track its position. The reported distance is 50 km, and the reported angle is 90°. Assume that the errors in both range and angle are distributed in a Gaussian manner. Given an infinite number of measurements what is the expected value of the position?

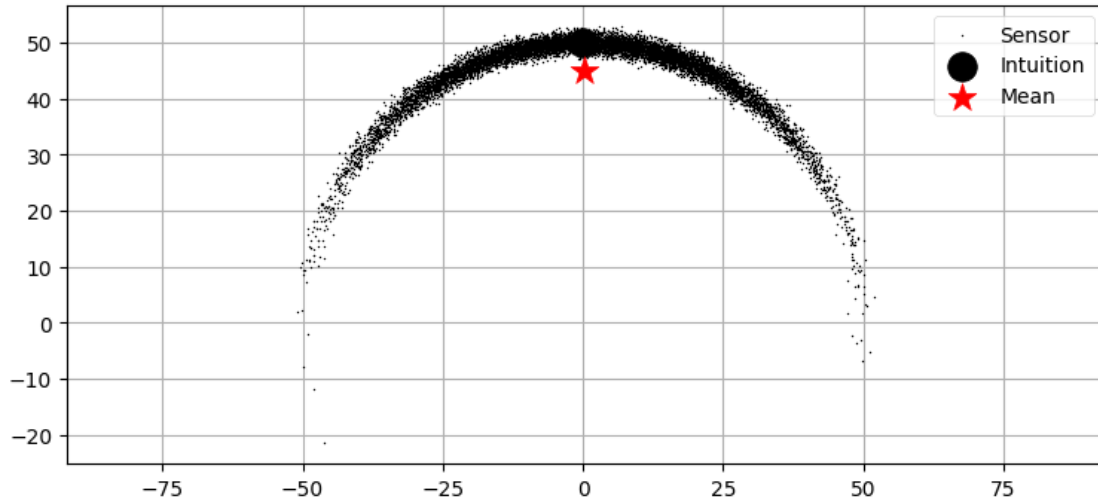
I have been recommending using intuition to gain insight, so let's see how it fares for this problem. We might reason that since the mean of the range will be 50 km, and the mean of the angle will be 90°, that the answer will be x=0 km, y=50 km.

Let's plot that and find out. Here are 3000 points plotted with a normal distribution of the distance of 0.4 km, and the angle having a normal distribution of 0.35 radians. We compute the average of the all of the positions, and display it as a star. Our intuition is displayed with a large circle.

```
[26]: import numpy as np
from numpy.random import randn
import matplotlib.pyplot as plt

N = 10000
a = np.pi/2. + (randn(N) * 0.45)
r = 50.0 + (randn(N) * 1)
xs = r * np.cos(a)
ys = r * np.sin(a)

plt.scatter(xs, ys, label='Sensor', color='k',
            alpha=1, marker='.', s=1)
xmean, ymean = sum(xs) / N, sum(ys) / N
plt.scatter(0, 50, c='k', marker='o', s=200, label='Intuition')
plt.scatter(xmean, ymean, c='r', marker='*', s=200, label='Mean')
plt.axis('equal')
plt.legend();
```



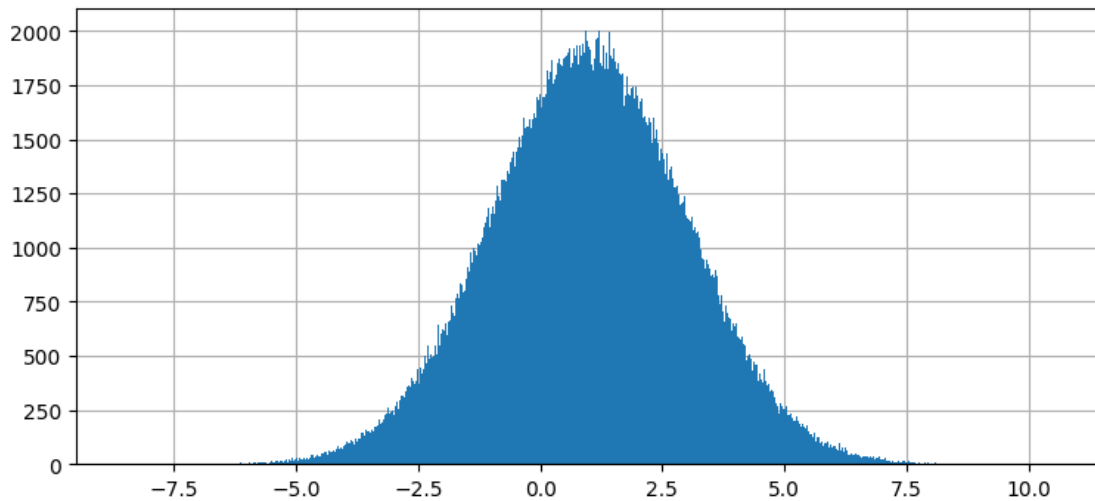
We can see that our intuition failed us because the nonlinearity of the problem forced all of the errors to be biased in one direction. This bias, over many iterations, can cause the Kalman filter to diverge. Even if it doesn't diverge the solution will not be optimal. Linear approximations applied to nonlinear problems yields inaccurate results.

1.4 The Effect of Nonlinear Functions on Gaussians

Gaussians are not closed under an arbitrary nonlinear function. Recall the equations of the Kalman filter - at each evolution we pass the Gaussian representing the state through the process function to get the Gaussian at time k . Our process function was always linear, so the output was always another Gaussian. Let's look at that on a graph. I will take an arbitrary Gaussian and pass it through the function $f(x) = 2x + 1$ and plot the result. We know how to do this analytically, but let's use sampling. I will generate 500,000 points with a normal distribution, pass them through $f(x)$, and plot the results. I do it this way because the next example will be nonlinear, and we will have no way to compute this analytically.

```
[27]: from numpy.random import normal

data = normal(loc=0., scale=1., size=500000)
plt.hist(2*data + 1, 1000);
```

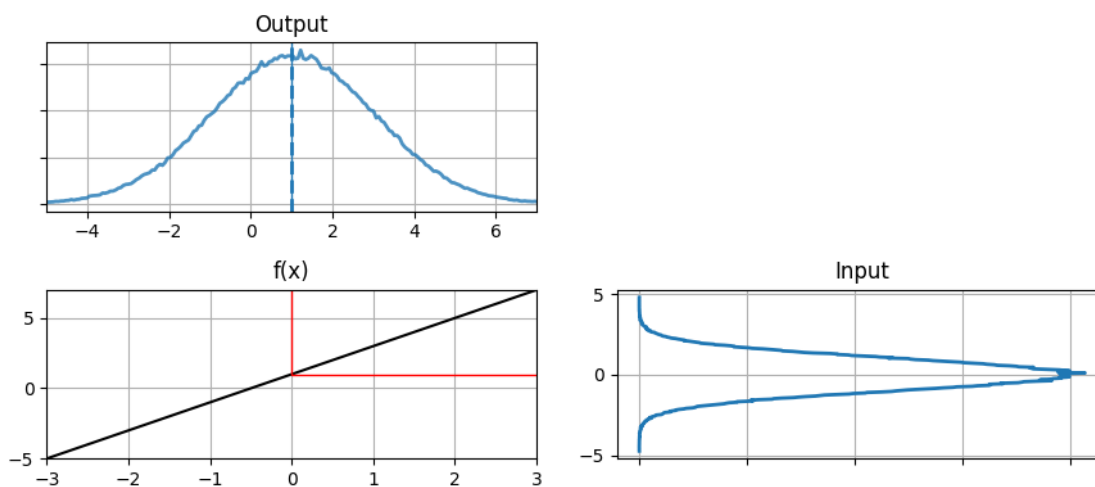


This is an unsurprising result. The result of passing the Gaussian through $f(x) = 2x + 1$ is another Gaussian centered around 1. Let's look at the input, nonlinear function, and output at once.

```
[28]: from kf_book.book_plots import set_figsize, figsize
      from kf_book.nonlinear_plots import plot_nonlinear_func

      def g1(x):
          return 2*x+1

      plot_nonlinear_func(data, g1)
```



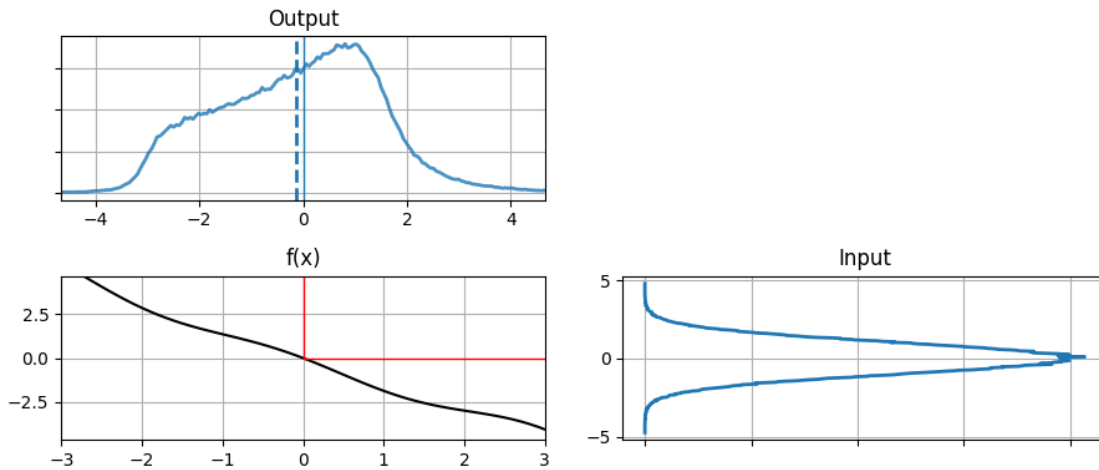
I explain how to plot Gaussians, and much more, in the Notebook *Computing_and_Plotting_PDFs* in the *Supporting_Notebooks* folder. You can also read it

online [here](#)[1]

The plot labeled ‘Input’ is the histogram of the original data. This is passed through the function $f(x) = 2x + 1$ which is displayed in the chart on the bottom left. The red lines shows how one value, $x = 0$ is passed through the function. Each value from input is passed through in the same way to the output function on the right. For the output I computed the mean by taking the average of all the points, and drew the results with the dotted blue line. A solid blue line shows the actual mean for the point $x = 0$. The output looks like a Gaussian, and is in fact a Gaussian. We can see that the variance in the output is larger than the variance in the input, and the mean has been shifted from 0 to 1, which is what we would expect given the transfer function $f(x) = 2x + 1$. The $2x$ affects the variance, and the $+1$ shifts the mean. The computed mean, represented by the dotted blue line, is nearly equal to the actual mean. If we used more points in our computation we could get arbitrarily close to the actual value.

Now let’s look at a nonlinear function and see how it affects the probability distribution.

```
[29]: def g2(x):  
      return (np.cos(3*(x/2 + 0.7))) * np.sin(0.3*x) - 1.6*x  
  
plot_nonlinear_func(data, g2)
```



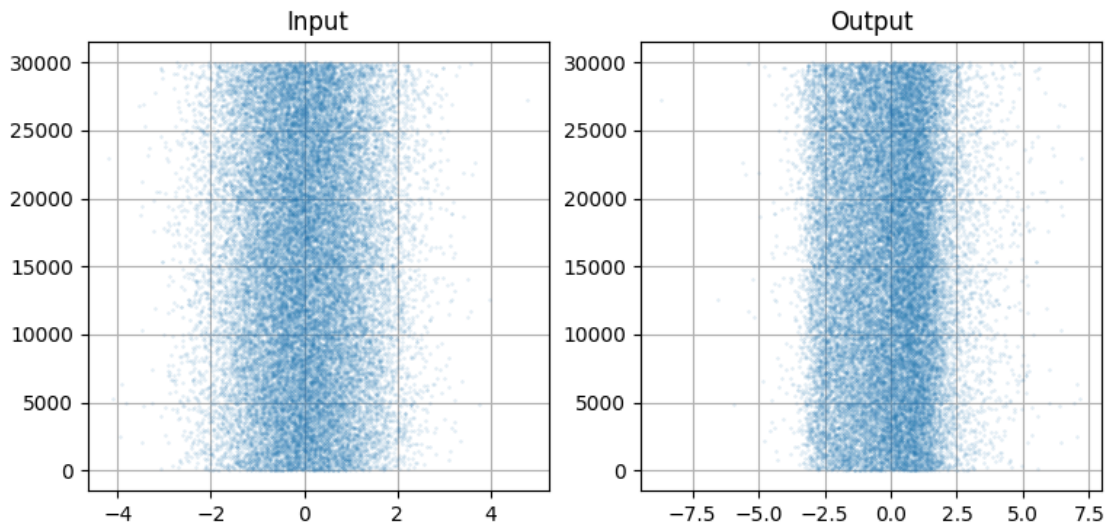
This result may be somewhat surprising to you. The function looks “fairly” linear, but the probability distribution of the output is completely different from a Gaussian. Recall the equations for multiplying two univariate Gaussians:

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

These equations do not hold for non-Gaussians, and certainly do not hold for the probability distribution shown in the ‘Output’ chart above.

Here's another way to look at the same data as scatter plots.

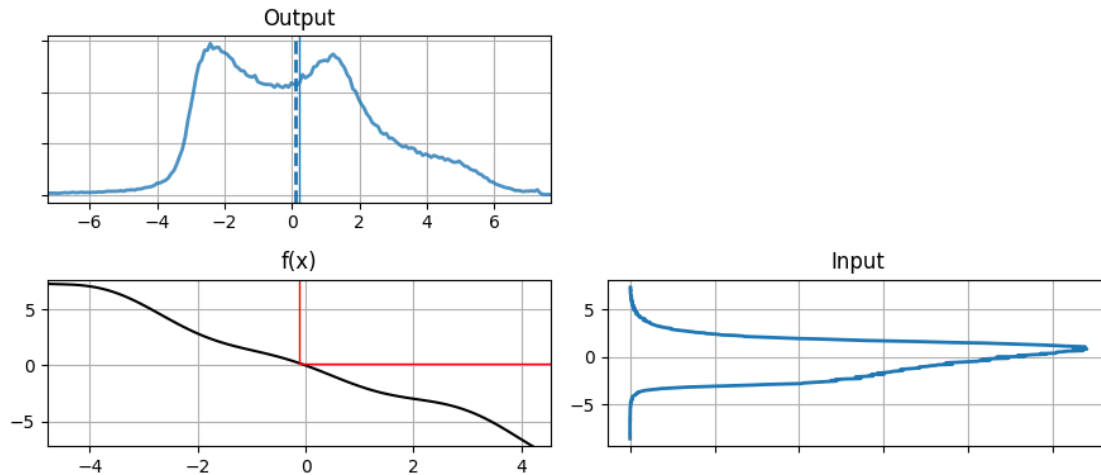
```
[30]: N = 30000
plt.subplot(121)
plt.scatter(data[:N], range(N), alpha=.1, s=1.5)
plt.title('Input')
plt.subplot(122)
plt.title('Output')
plt.scatter(g2(data[:N]), range(N), alpha=.1, s=1.5);
```



The original data is clearly Gaussian, but the data passed through $g2(x)$ is no longer normally distributed. There is a thick band near -3, and the points are unequally distributed on either side of the band. If you compare this to the pdf labelled 'output' in the previous chart you should be able to see how the pdf shape matches the distribution of $g(data)$.

Think of what this implies for the Kalman filter algorithm of the previous chapter. All of the equations assume that a Gaussian passed through the process function results in another Gaussian. If this is not true then all of the assumptions and guarantees of the Kalman filter do not hold. Let's look at what happens when we pass the output back through the function again, simulating the next step time step of the Kalman filter.

```
[31]: y = g2(data)
plot_nonlinear_func(y, g2)
```



As you can see the probability function is further distorted from the original Gaussian. However, the graph is still somewhat symmetric around $x=0$, let's see what the mean is.

```
[32]: print('input mean, variance: %.4f, %.4f' %
          (np.mean(data), np.var(data)))
       print('output mean, variance: %.4f, %.4f' %
          (np.mean(y), np.var(y)))
```

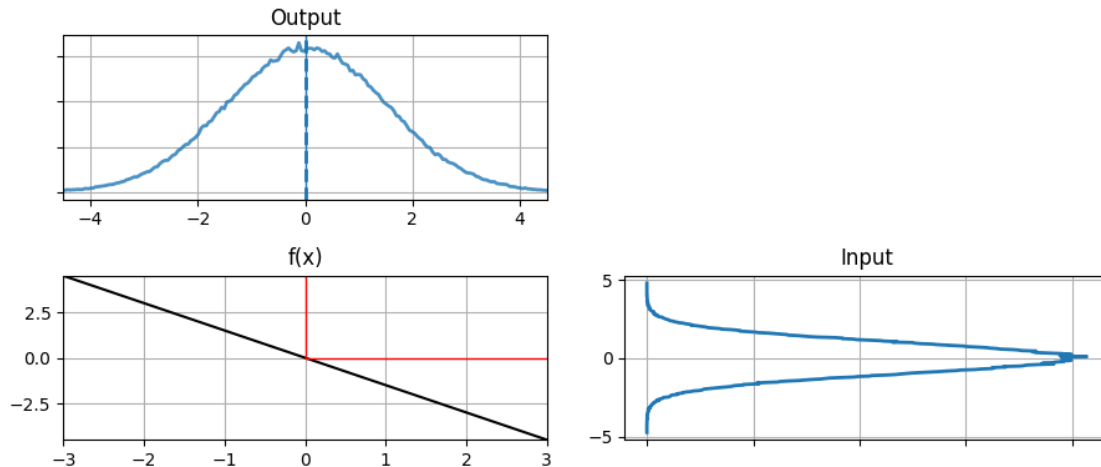
```
input mean, variance: -0.0006, 1.0021
output mean, variance: -0.1236, 2.4099
```

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

$$m = \frac{-3 - 3}{2 - (-2)} = -1.5$$

```
[33]: def g3(x):
       return -1.5 * x

       plot_nonlinear_func(data, g3)
       out = g3(data)
       print('output mean, variance: %.4f, %.4f' %
          (np.mean(out), np.var(out)))
```

output mean, variance: 0.0009, 2.2548

Although the shapes of the output are very different, the mean and variance of each are almost the same. This may lead us to reasoning that perhaps we can ignore this problem if the nonlinear equation is 'close to' linear. To test that, we can iterate several times and then compare the results.

```
[34]: out = g3(data)
      out2 = g2(data)

      for i in range(10):
          out = g3(out)
          out2 = g2(out2)
      print('linear    output mean, variance: %.4f, %.4f' %
            (np.average(out), np.std(out)**2))
      print('nonlinear output mean, variance: %.4f, %.4f' %
            (np.average(out2), np.std(out2)**2))
```

linear output mean, variance: 0.0536, 7497.8108

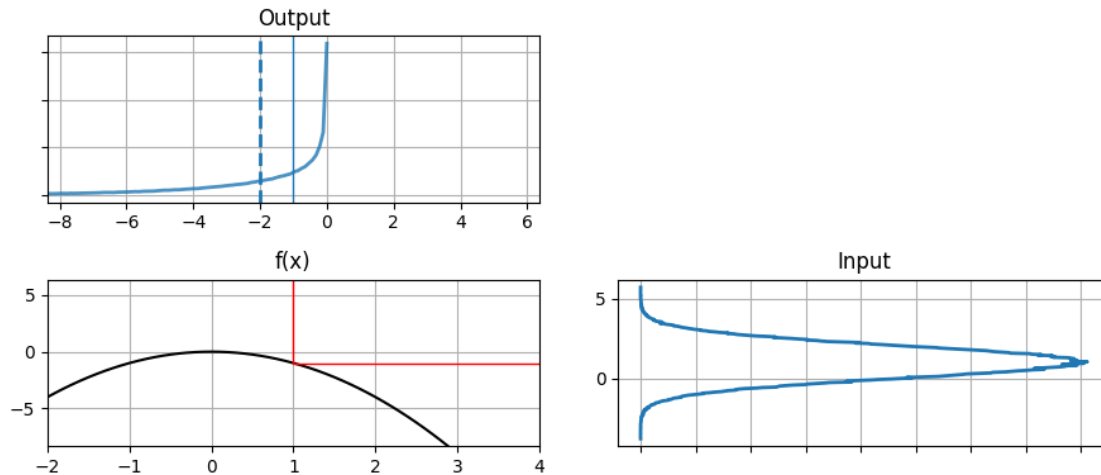
nonlinear output mean, variance: -9.0612, 30538.8055

Unfortunately the nonlinear version is not stable. It drifted significantly from the mean of 0, and the variance is half an order of magnitude larger.

I minimized the issue by using a function that is quite close to a straight line. What happens if the function is $y(x) = -x^2$?

```
[35]: def g3(x):
      return -x*x

      data = normal(loc=1, scale=1, size=500000)
      plot_nonlinear_func(data, g3)
```



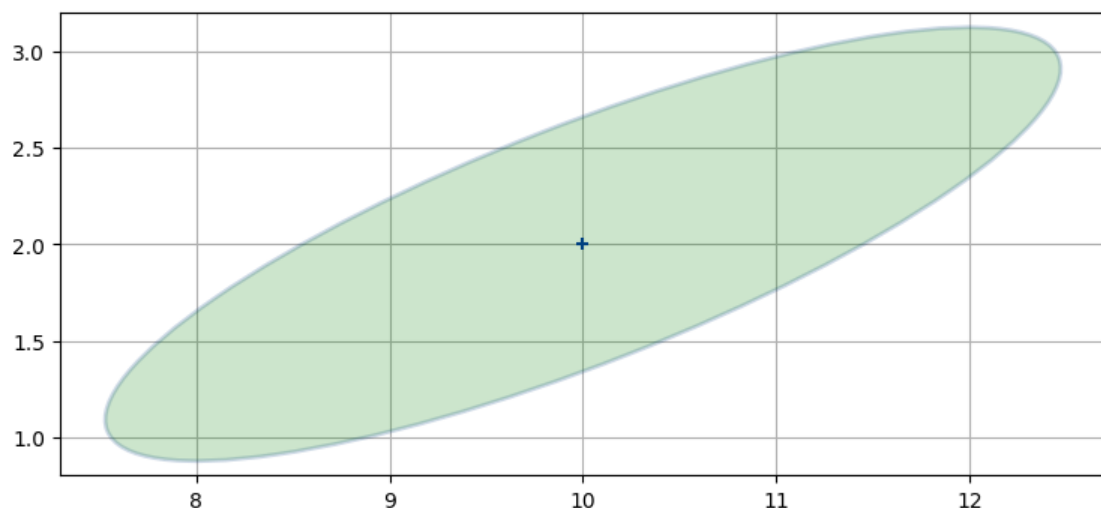
Despite the curve being smooth and reasonably straight at $x = 1$ the probability distribution of the output doesn't look anything like a Gaussian and the computed mean of the output is quite different than the value computed directly. This is not an unusual function - a ballistic object moves in a parabola, and this is the sort of nonlinearity your filter will need to handle. If you recall we've tried to track a ball and failed miserably. This graph should give you insight into why the filter performed so poorly.

1.5 A 2D Example

It is hard to look at probability distributions and reason about what will happen in a filter. So let's think about tracking an aircraft with radar. The estimate may have a covariance that looks like this:

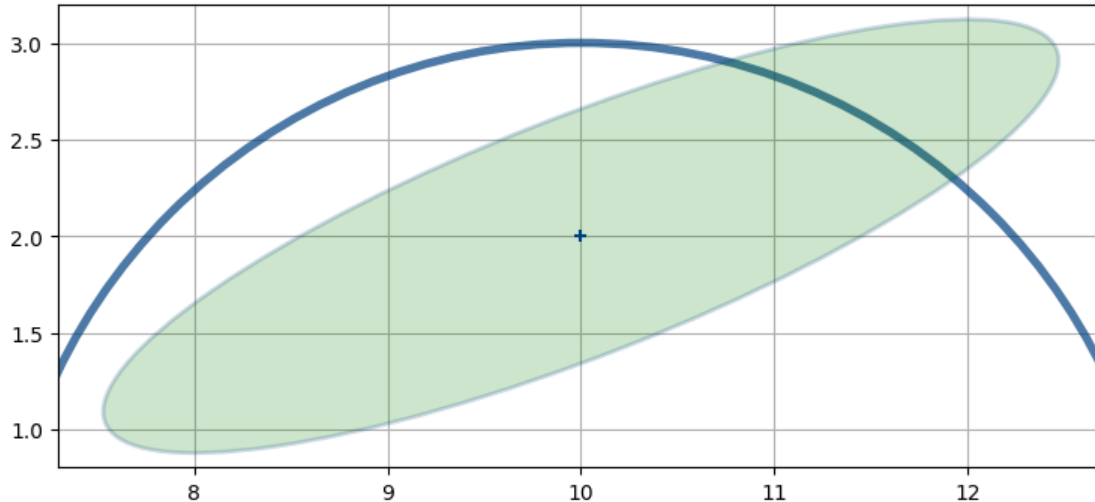
```
[36]: import kf_book.nonlinear_internal as nonlinear_internal

nonlinear_internal.plot1()
```



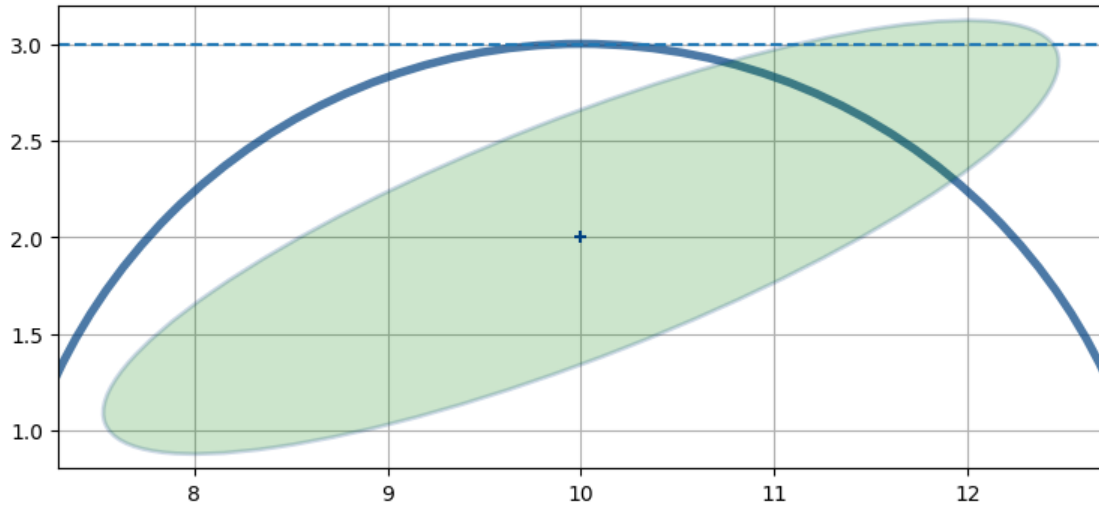
What happens when we try to linearize this problem? The radar gives us a range to the aircraft. Suppose the radar is directly under the aircraft ($x=10$) and the next measurement states that the aircraft is 3 miles away ($y=3$). The positions that could match that measurement form a circle with radius 3 miles, like so.

```
[37]: nonlinear_internal.plot2()
```



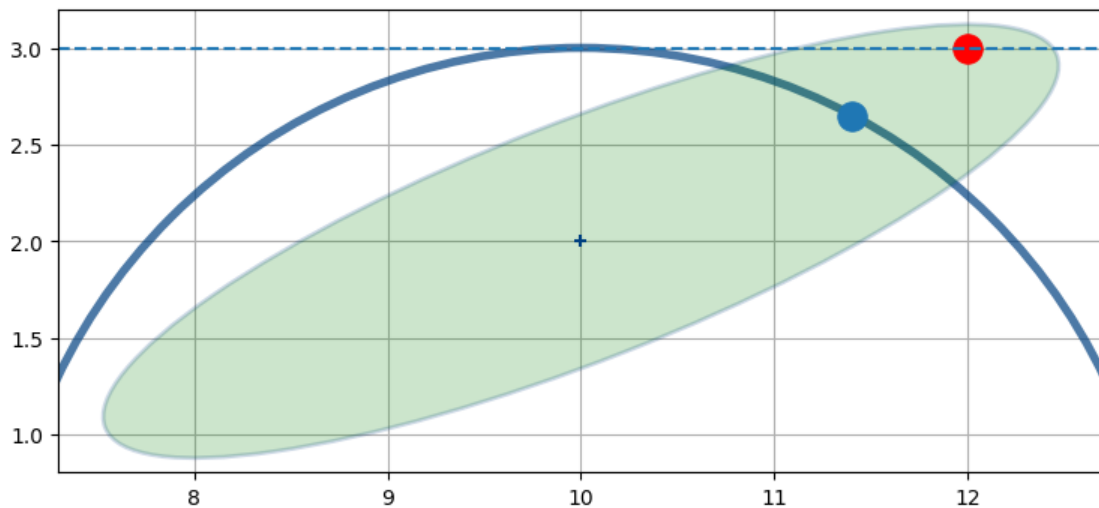
We can see by inspection that the probable position of the aircraft is somewhere near $x=11.4$, $y=2.7$ because that is where the covariance ellipse and range measurement overlap. But the range measurement is nonlinear so we have to linearize it. We haven't covered this material yet, but the Extended Kalman filter will linearize at the last position of the aircraft - (10,2). At $x=10$ the range measurement has $y=3$, and so we linearize at that point.

```
[38]: nonlinear_internal.plot3()
```



Now we have a linear representation of the problem (literally a straight line) which we can solve. Unfortunately you can see that the intersection of the line and the covariance ellipse is a long way from the actual aircraft position.

```
[39]: nonlinear_internal.plot4()
```



That sort of error often leads to disastrous results. The error in this estimate is large. But in the next innovation of the filter that very bad estimate will be used to linearize the next radar measurement, so the next estimate is likely to be markedly worse than this one. After only a few iterations the Kalman filter will diverge, and start producing results that have no correspondence to reality.

This covariance ellipse spans miles. I exaggerated the size to illustrate the difficulties of highly

nonlinear systems. In real radar tracking problems the nonlinearity is usually not that bad, but the errors will still accumulate. Other systems you might work with could have this amount of nonlinearity - this was not an exaggeration only to make a point. You will always be battling divergence when working with nonlinear systems.

1.6 The Algorithms

You may be impatient to solve a specific problem, and wondering which filter to use. I will quickly survey the options. The subsequent chapters are somewhat independent of each other, and you can fruitfully skip around, though I recommend reading linearly if you truly want to master all of the material.

The workhorses of nonlinear filters are the *linearized Kalman filter* and *extended Kalman filter* (EKF). These two techniques were invented shortly after Kalman published his paper and they have been the main techniques used since then. The flight software in airplanes, the GPS in your car or phone almost certainly use one of these techniques.

However, these techniques are extremely demanding. The EKF linearizes the differential equations at one point, which requires you to find a solution to a matrix of partial derivatives (a Jacobian). This can be difficult or impossible to do analytically. If impossible, you have to use numerical techniques to find the Jacobian, but this is expensive computationally and introduces more error into the system. Finally, if the problem is quite nonlinear the linearization leads to a lot of error being introduced in each step, and the filters frequently diverge. You cannot throw some equations into some arbitrary solver and expect to get good results. It's a difficult field for professionals. I note that most Kalman filtering textbooks merely gloss over the EKF despite it being the most frequently used technique in real world applications.

Recently the field has been changing in exciting ways. First, computing power has grown to the point that we can use techniques that were once beyond the ability of a supercomputer. These use *Monte Carlo* techniques - the computer generates thousands to tens of thousands of random points and tests all of them against the measurements. It then probabilistically kills or duplicates points based on how well they match the measurements. A point far away from the measurement is unlikely to be retained, whereas a point very close is quite likely to be retained. After a few iterations there is a clump of particles closely tracking your object, and a sparse cloud of points where there is no object.

This has two benefits. First, the algorithm is robust even for extremely nonlinear problems. Second, the algorithm can track arbitrarily many objects at once - some particles will match the behavior of one object, and other particles will match other objects. So this technique is often used to track automobile traffic, people in crowds, and so on.

The costs should be clear. It is computationally expensive to test tens of thousands of points for every step in the filter. But modern CPUs are very fast, and this is a good problem for GPUs because the part of the algorithm is parallelizable. Another cost is that the answer is not mathematical. With a Kalman filter my covariance matrix gives me important information about the amount of error in the estimate. The particle filter does not give me a rigorous way to compute this. Finally, the output of the filter is a cloud of points; I then have to figure out how to interpret it. Usually you will be doing something like taking the mean and standard deviations of the points, but this is a difficult problem. There are still many points that do not 'belong' to a tracked object, so you first have to run some sort of clustering algorithm to first find the points that seem to be

tracking an object, and then you need another algorithm to produce a state estimate from those points. None of this is intractable, but it is all quite computationally expensive.

Finally, we have a new algorithm called the *unscented Kalman filter* (UKF). It does not require you to find analytic solutions to nonlinear equations, and yet almost always performs better than the EKF. It does well with nonlinear problems - problems where the EKF has significant difficulties. Designing the filter is extremely easy. Some will say the jury is still out on the UKF, but to my mind the UKF is superior in almost every way to the EKF. I suggest that the UKF should be the starting point for any implementation, especially if you are not a Kalman filter professional with a graduate degree in control theory. The main downside is that the UKF can be a few times slower than the EKF, but this really depends on whether the EKF solves the Jacobian analytically or numerically. If numerically the UKF is almost certainly faster. It has not been proven (and probably it cannot be proven) that the UKF always yields more accurate results than the EKF. In practice it almost always does, often significantly so. It is very easy to understand and implement, and I strongly suggest this filter as your starting point.

1.7 Summary

The world is nonlinear, but we only really know how to solve linear problems. This introduces significant difficulties for Kalman filters. We've looked at how nonlinearity affects filtering in 3 different but equivalent ways, and I've given you a brief summary of the major approaches: the linearized Kalman filter, the extended Kalman filter, the Unscented Kalman filter, and the particle filter.

Until recently the linearized Kalman filter and EKF have been the standard way to solve these problems. They are very difficult to understand and use, and they are also potentially very unstable.

Recent developments have offered what are to my mind superior approaches. The UKF dispenses with the need to find solutions to partial differential equations, yet it is also usually more accurate than the EKF. It is easy to use and understand. I can get a basic UKF going in a few minutes by using FilterPy. The particle filter dispenses with mathematical modeling completely in favor of a Monte Carlo technique of generating a random cloud of thousands of points. It runs slowly, but it can solve otherwise intractable problems with relative ease.

I get more email about the EKF than anything else; I suspect that this is because most treatments in books, papers, and on the internet use the EKF. If your interest is in mastering the field of course you will want to learn about the EKF. But if you are just trying to get good results I point you to the UKF and particle filter first. They are much easier to implement, understand, and use, and they are typically far more stable than the EKF.

Some will quibble with that advice. A lot of recent publications are devoted to a comparison of the EKF, UKF, and perhaps a few other choices for a given problem. Do you not need to perform a similar comparison for your problem? If you are sending a rocket to Mars then of course you do. You will be balancing issues such as accuracy, round off errors, divergence, mathematical proof of correctness, and the computational effort required. I can't imagine not knowing the EKF intimately.

On the other hand the UKF works spectacularly! I use it at work for real world applications. I mostly haven't even tried to implement an EKF for these applications because I can verify that the UKF is working fine. Is it possible that I might eke out another 0.2% of performance from the EKF in certain situations? Sure! Do I care? No! I completely understand the UKF implementation, it is

easy to test and verify, I can pass the code to others and be confident that they can understand and modify it, and I am not a masochist that wants to battle difficult equations when I already have a working solution. If the UKF or particle filters start to perform poorly for some problem then I will turn to other techniques, but not before then. And realistically, the UKF usually provides substantially better performance than the EKF over a wide range of problems and conditions. If “really good” is good enough I’m going to spend my time working on other problems.

I’m belaboring this point because in most textbooks the EKF is given center stage, and the UKF is either not mentioned at all or just given a 2 page gloss that leaves you completely unprepared to use the filter. The UKF is still relatively new, and it takes time to write new editions of books. At the time many books were written the UKF was either not discovered yet, or it was just an unproven but promising curiosity. But as I am writing this now, the UKF has had enormous success, and it needs to be in your toolkit. That is what I will spend most of my effort trying to teach you.

1.8 References

[1] https://github.com/rlabbe/Kalman-and-Bayesian-Filters-in-Python/blob/master/Supporting_Notebooks/Con