10-Unscented-Kalman-Filter

November 5, 2022

Table of Contents

1 The Unscented Kalman Filter

[12]:
```
%matplotlib inline
```

[13]:
```
#format the book
import book_format
book_format.set_style()
```

[13]:
```
<IPython.core.display.HTML object>
```

In the last chapter we discussed the difficulties that nonlinear systems pose. This nonlinearity can appear in two places. It can be in our measurements, such as a radar that is measuring the slant range to an object. Slant range requires you to take a square root to compute the x,y coordinates:

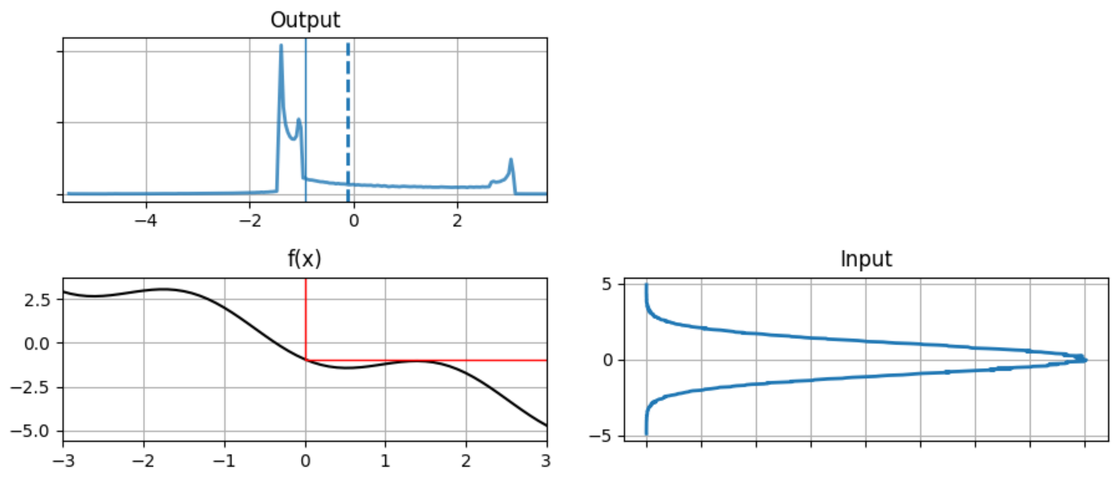
$$x = \sqrt{\text{slant}^2 - \text{altitude}^2}$$

The nonlinearity can also occur in the process model - we may be tracking a ball traveling through the air, where the effects of air drag lead to nonlinear behavior. The standard Kalman filter performs poorly or not at all with these sorts of problems.

In the last chapter I showed you a plot like this. I have altered the equation somewhat to emphasize the effects of nonlinearity.

[14]:
```
from kf_book.book_plots import set_figsize, figsize
import matplotlib.pyplot as plt
from kf_book.nonlinear_plots import plot_nonlinear_func
from numpy.random import normal
import numpy as np

# create 500,000 samples with mean 0, std 1
gaussian = (0., 1.)
data = normal(loc=gaussian[0], scale=gaussian[1], size=500000)

def f(x):
    return (np.cos(4*(x/2 + 0.7))) - 1.3*x
```

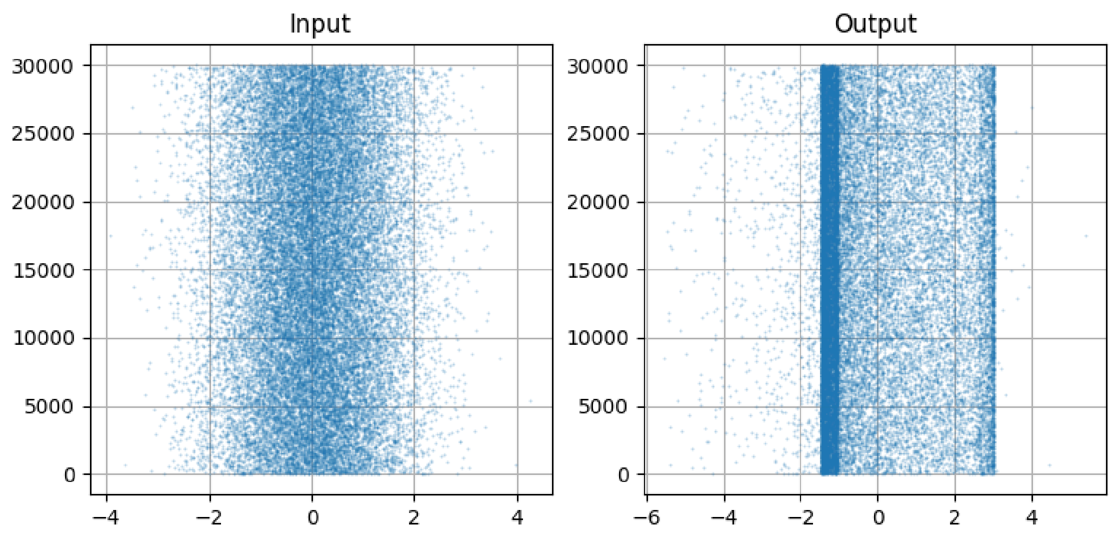
```
plot_nonlinear_func(data, f)
```



I generated this by taking 500,000 samples from the input, passing it through the nonlinear transform, and building a histogram of the result. We call these points *sigma points*. From the output histogram we can compute a mean and standard deviation which would give us an updated, albeit approximated Gaussian.

Let me show you a scatter plot of the data before and after being passed through `f(x)`.

[15]:
```
N = 30000
plt.subplot(121)
plt.scatter(data[:N], range(N), alpha=.2, s=1)
plt.title('Input')
plt.subplot(122)
plt.title('Output')
plt.scatter(f(data[:N]), range(N), alpha=.2, s=1);
```



The data itself appears to be Gaussian, which it is. By that I mean it looks like white noise scattered around the mean zero. In contrast `g(data)` has a defined structure. There are two bands, with a significant number of points in between. On the outside of the bands there are scattered points, but with many more on the negative side.

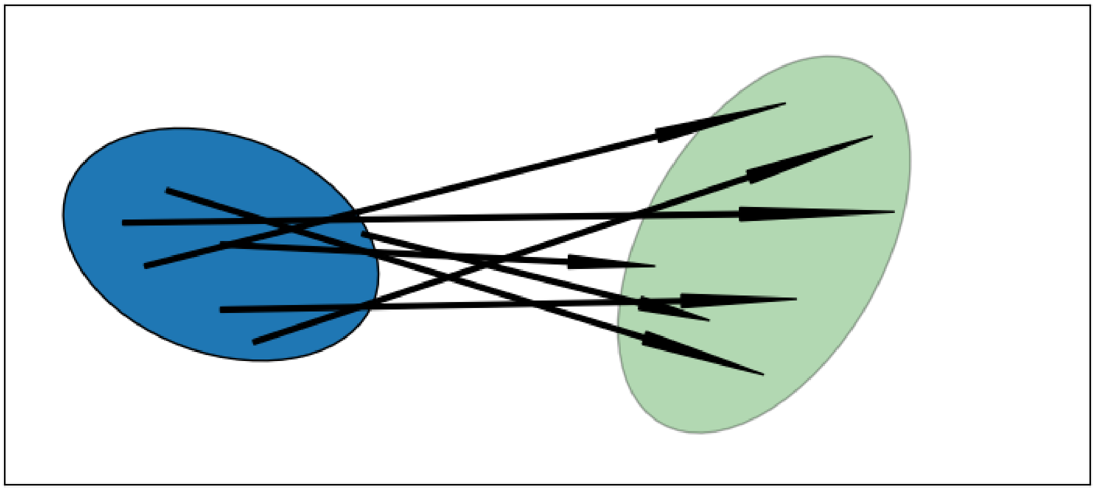
It has perhaps occurred to you that this sampling process constitutes a solution to our problem. Suppose for every update we generated 500,000 points, passed them through the function, and then computed the mean and variance of the result. This is called a *Monte Carlo* approach, and it used by some Kalman filter designs, such as the Ensemble filter and particle filter. Sampling requires no specialized knowledge, and does not require a closed form solution. No matter how nonlinear or poorly behaved the function is, as long as we sample with enough sigma points we will build an accurate output distribution.

"Enough points" is the rub. The graph above was created with 500,000 sigma points, and the output is still not smooth. What's worse, this is only for 1 dimension. The number of points required increases by the power of the number of dimensions. If you only needed 500 points for 1 dimension, you'd need 500 squared, or 250,000 points for two dimensions, 500 cubed, or 125,000,000 points for three dimensions, and so on. So while this approach does work, it is very computationally expensive. Ensemble filters and particle filters use clever techniques to significantly reduce this dimensionality, but the computational burdens are still very large. The unscented Kalman filter uses sigma points but drastically reduces the amount of computation by using a deterministic method to choose the points.

1.1 Sigma Points - Sampling from a Distribution

Let's look at the problem in terms of a 2D covariance ellipse. I choose 2D merely because it is easy to plot; this extends to any number of dimensions. Assuming some arbitrary nonlinear function, we will take random points from the first covariance ellipse, pass them through the nonlinear function, and plot their new position. Then we can compute the mean and covariance of the transformed points, and use that as our estimate of the mean and probability distribution.

[16]:
```
import kf_book.ukf_internal as ukf_internal
ukf_internal.show_2d_transform()
```



On the left we show an ellipse depicting the 1σ distribution of two state variables. The arrows show how several randomly sampled points might be transformed by some arbitrary nonlinear function to a new distribution. The ellipse on the right is drawn semi-transparently to indicate that it is an *estimate* of the mean and variance of this collection of points.

Let's write a function which passes 10,000 points randomly drawn from the Gaussian

$$\mu = \begin{bmatrix}0\\0\end{bmatrix}, \Sigma = \begin{bmatrix}32 & 15\\15 & 40\end{bmatrix}$$

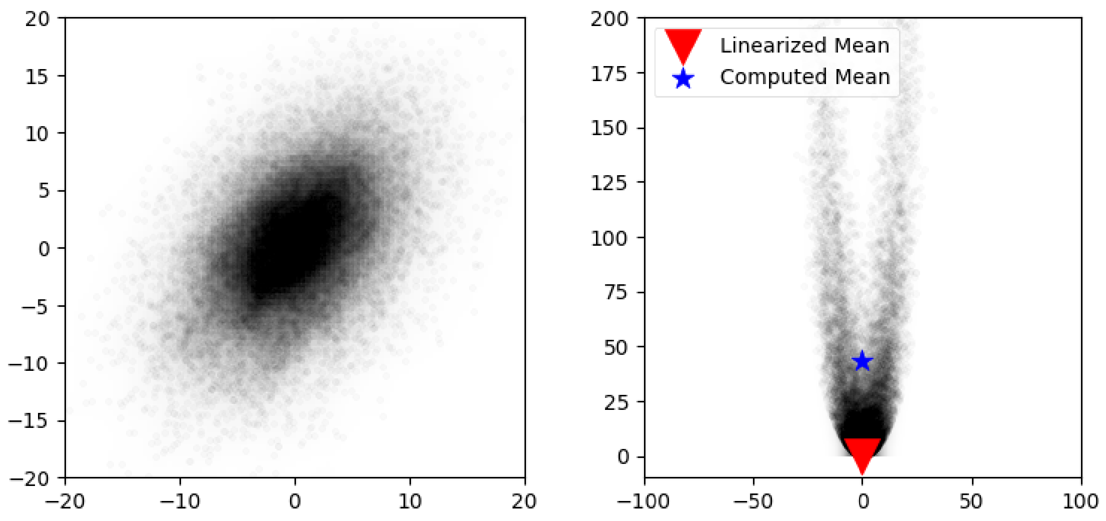
through the nonlinear system:

$$\begin{cases}\bar{x} = x + y\\ \bar{y} = 0.1x^2 + y^2\end{cases}$$

[17]:
```
import numpy as np
from numpy.random import multivariate_normal
from kf_book.nonlinear_plots import plot_monte_carlo_mean

def f_nonlinear_xy(x, y):
    return np.array([x + y, .1*x**2 + y*y])

mean = (0., 0.)
p = np.array([[32., 15.], [15., 40.]])
# Compute linearized mean
mean_fx = f_nonlinear_xy(*mean)
```

```python
#generate random points
xs, ys = multivariate_normal(mean=mean, cov=p, size=10000).T
plot_monte_carlo_mean(xs, ys, f_nonlinear_xy, mean_fx, 'Linearized Mean');
```

```
Difference in mean x=0.016, y=43.567
```



This plot shows the strong nonlinearity that occurs with this function, and the large error that would result if we linearized in the way of the Extended Kalman filter (we will be learning this in the next chapter).

1.2 A Quick Example

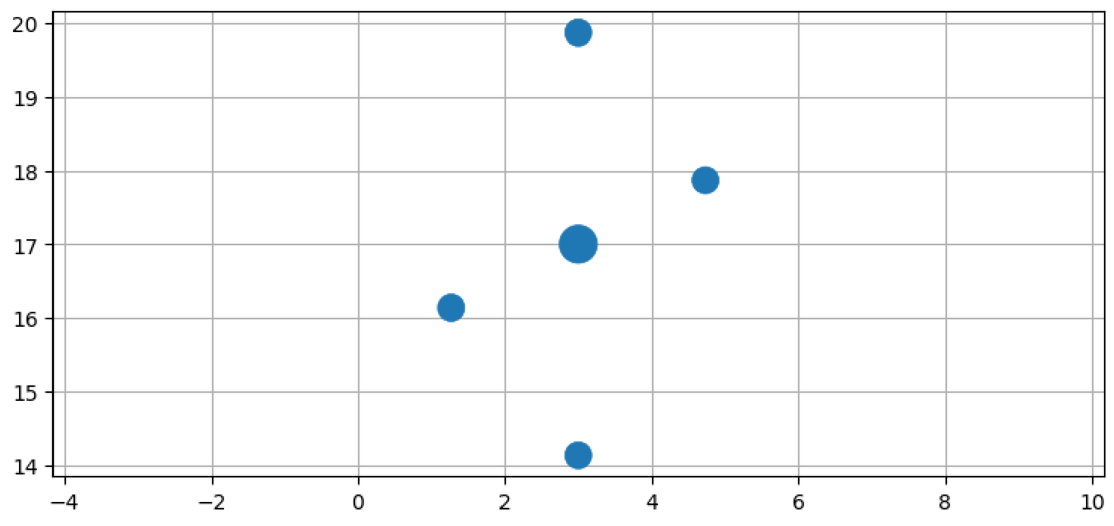
I'm soon going to proceed into the math the Unscented Kalman Filter (UKF) uses to choose sigma points and then perform the computation. But let's just start with an example so you can see the destination, so to speak.

We will learn that the UKF can use many different algorithms to generate the sigma points. Several algorithms are provided by FilterPy. Here's one possibility:

[18]:
```python
from filterpy.kalman import JulierSigmaPoints

sigmas = JulierSigmaPoints(n=2, kappa=1)
```

This will become clearer later, but the object will generate weighted sigma points for any given mean and covariance. Let's just look at an example, where the point's size indicate how much it is weighted:

[19]:
```python
from kf_book.ukf_internal import plot_sigmas
plot_sigmas(sigmas, x=[3, 17], cov=[[1, .5], [.5, 3]])
```



You can see we have 5 points centered around the mean (3, 17) in an odd pattern. It may seem absurd that this will do as well or better than 500,000 randomly generated points, but it will!

Okay, now let's implement the filter. We will implement a standard linear filter in 1D; we aren't quite ready to tackle a nonlinear filter yet. The design of the filter is not much different than what we have learned so far, with one difference. The KalmanFilter class uses the matrix **F** to compute the state transition function. Matrices mean **linear** algrebra, which work for linear problems, but not nonlinear ones. So, instead of a matrix we provide a function, just like we did above. The KalmanFilter class uses another matrix **H** to implement the measurement function, which converts a state into the equivalent measurement. Again, a matrix implies linearity, so instead of a matrix we provide a function. Perhaps it is clear why **H** is called the 'measurement function'; for the linear Kalman filter it is a matrix, but that is just a fast way to compute a function that happens to be linear.

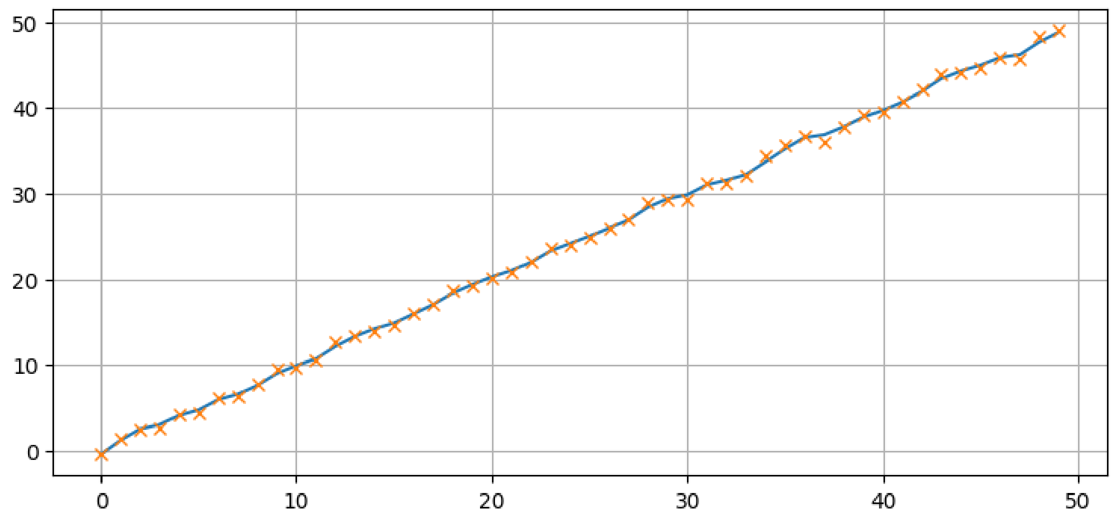
Without further ado, here are the state transistion function and measurement function for a 1D tracking problem, where the state is $\mathbf{x} = [x\ \dot{x}]^\mathsf{T}$:

[20]:
```python
def fx(x, dt):
    xout = np.empty_like(x)
    xout[0] = x[1] * dt + x[0]
    xout[1] = x[1]
    return xout

def hx(x):
    return x[:1] # return position [x]
```

Let's be clear, this is a linear example. There is no need to use a UKF for a linear problem, but I'm starting with the simplest possible example. But notice how I wrote `fx()` to compute $\bar{\mathbf{x}}$ as a set of equations instead of a matrix multiplication. This is to illustrate that I could implement any arbitrary nonlinear function here; we are not constrained to linear equations.

The rest of the design is the same. Design **P**, **R**, and **Q**. You know how to do that, so let's just finish up the filter and run it.

[21]:
```python
from numpy.random import randn
from filterpy.kalman import UnscentedKalmanFilter
from filterpy.common import Q_discrete_white_noise

ukf = UnscentedKalmanFilter(dim_x=2, dim_z=1, dt=1., hx=hx, fx=fx,␣
  ↪points=sigmas)
ukf.P *= 10
ukf.R *= .5
ukf.Q = Q_discrete_white_noise(2, dt=1., var=0.03)

zs, xs = [], []
for i in range(50):
    z = i + randn()*.5
    ukf.predict()
    ukf.update(z)
    xs.append(ukf.x[0])
    zs.append(z)

plt.plot(xs);
plt.plot(zs, marker='x', ls='');
```



There really isn't a lot new here. You have to create an object that creates sigma points for you, and provide functions instead of matrices for **F** and **H**, but the rest is the same as before. This should give you enough confidence to plow through a bit of math and algorithms so you can understand what the UKF is doing.

1.3 Choosing Sigma Points

At the start of the chapter I used 500,000 randomly generated sigma points to compute the the probability distribution of a Gaussian passed through a nonlinear function. While the computed mean is quite accurate, computing 500,000 points for every update would cause our filter to be extremely slow. So, what would be fewest number of sampled points that we can use, and what kinds of constraints does this problem formulation put on the points? We will assume that we have no special knowledge about the nonlinear function as we want to find a generalized algorithm that works for any function.

Let's consider the simplest possible case and see if it offers any insight. The simplest possible system is the **identity function**: $f(x) = x$. If our algorithm does not work for the identity function then the filter cannot converge. In other words, if the input is 1 (for a one dimensional system), the output must also be 1. If the output was different, such as 1.1, then when we fed 1.1 into the transform at the next time step, we'd get out yet another number, maybe 1.23. This filter diverges.

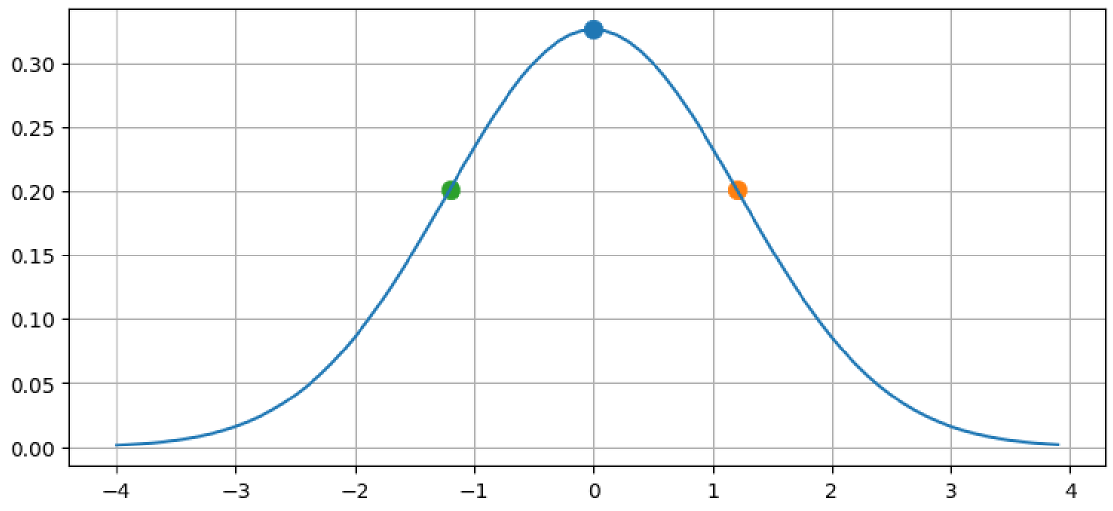
The fewest number of points that we can use is one per dimension. This is the number that the linear Kalman filter uses. The input to a Kalman filter for the distribution $\mathcal{N}(\mu, \sigma^2)$ is μ itself. So while this works for the linear case, it is not a good answer for the nonlinear case.

Perhaps we can use one point per dimension, but altered somehow. However, if we were to pass some value $\mu + \Delta$ into the identity function $f(x) = x$ it would not converge, so this will not work. If we didn't alter μ then this would be the standard Kalman filter. We must conclude that one sample will not work.

What is the next lowest number we can choose? Two. Consider the fact that Gaussians are symmetric, and that we probably want to always have one of our sample points be the mean of the input for the identity function to work. Two points would require us to select the mean, and then one other point. That one other point would introduce an asymmetry in our input that we probably don't want. It would be very difficult to make this work for the identity function $f(x) = x$.

The next lowest number is 3 points. 3 points allows us to select the mean, and then one point on each side of the mean, as depicted on the chart below.

[22]:
```
ukf_internal.show_3_sigma_points()
```



We can pass these points through a nonlinear function f(x) and compute the resulting mean and variance. The mean can be computed as the average of the 3 points, but that is not very general. For example, for a very nonlinear problem we might want to weight the center point much higher than the outside points, or we might want to weight the outside points higher.

A more general approach is to compute the weighted mean $\mu = \sum_i w_i\, f(\mathcal{X}_i)$, where the calligraphic $\mathcal{X}$ are the sigma points. We need the sums of the weights to equal one. Given that requirement, our task is to select $\mathcal{X}$ and their corresponding weights so that they compute to the mean and variance of the transformed sigma points.

If we weight the means it also makes sense to weight the covariances. It is possible to use different weights for the mean (w^m) and for the covariance (w^c). I use superscripts to allow space for indexes in the following equations. We can write

$$\begin{aligned}
\mathbf{Constraints:}\\
1 &= \sum_i w^m_i \\
1 &= \sum_i w^c_i \\
\mu &= \sum_i w_i^m f(\mathcal{X}_i) \\
\Sigma &= \sum_i w_i^c (f(\mathcal{X})_i-\mu)(f(\mathcal{X})_i-\mu)^\intercal
\end{aligned}$$

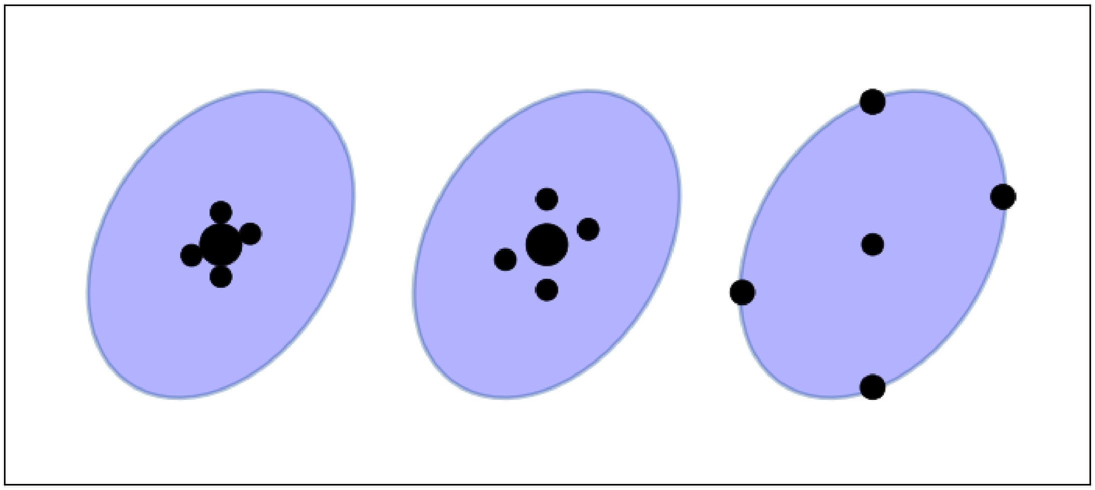
The first two equations are the constraint that the weights must sum to one. The third equation is how you compute a weight mean. The forth equation may be less familiar, but recall that the equation for the covariance of two random variables is:

$$COV(x,y) = \frac{\sum(x-\bar x)(y-\bar{y})}{n}$$

These constraints do not form a unique solution. For example, if you make w^m_0 smaller you can compensate by making w^m_1 and w^m_2 larger. You can use different weights for the mean and covariances, or the same weights. Indeed, these equations do not require that any of the points be the mean of the input at all, though it seems 'nice' to do so, so to speak.

We want an algorithm that satisfies the constraints, preferably with only 3 points per dimension. Before we go on I want to make sure the idea is clear. Below are three different examples for the same covariance ellipse with different sigma points. The size of the sigma points is proportional to the weight given to each.

[23]:
```
ukf_internal.show_sigma_selections()
```



The points do not lie along the major and minor axis of the ellipse; nothing in the constraints require me to do that. I show the points evenly spaced, but the constraints do not require that.

The arrangement and weighting of the sigma points affect how we sample the distribution. Points that are close together will sample local effects, and thus probably work better for very nonlinear problems. Points that are far apart, or far off the axis of the ellipse will sample non-local effects and non Gaussian behavior. However, by varying the weights used for each point we can mitigate this. If the points are far from the mean but weighted very slightly we will incorporate some of the knowledge about the distribution without allowing the nonlinearity of the problem to create a bad estimate.

Please understand there are infinite ways to select sigma points. The constraints I choose are just one way to do it. For example, not all algorithms for creating the sigma points require the weights to sum to one. In fact, the algorithm I favor in this book does not have that property.

1.4 The Unscented Transform

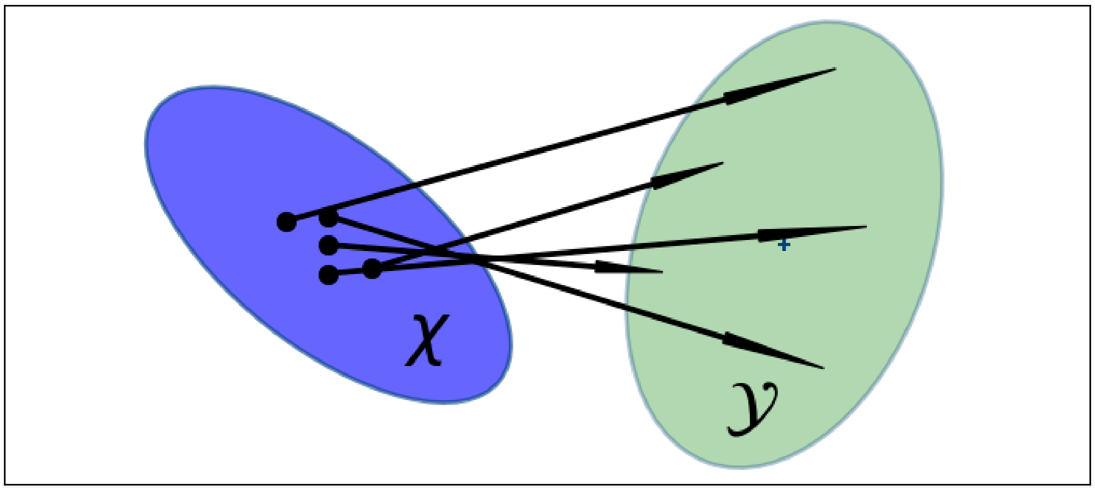
For the moment, assume an algorithm for selecting the sigma points and weights exists. How are the sigma points used to implement a filter?

The *unscented transform* is the core of the algorithm yet it is remarkably simple. It passes the sigma points χ through a nonlinear function yielding a transformed set of points.

$$\mathcal{Y} = f(\chi)$$

It then computes the mean and covariance of the transformed points. That mean and covariance becomes the new estimate. The figure below depicts the operation of the unscented transform. The green ellipse on the right represents the computed mean and covariance to the transformed sigma points.

```
[24]: ukf_internal.show_sigma_transform(with_text=True)
```



The mean and covariance of the sigma points are computed as:

$$\mu = \sum_{i=0}^{2n} w_i^m \mathcal{Y}_i$$
$$\Sigma = \sum_{i=0}^{2n} w_i^c (\mathcal{Y}_i - \mu)(\mathcal{Y}_i - \mu)^\mathsf{T}$$

These equations should be familiar - they are the constraint equations we developed above.

In short, the unscented transform takes points sampled from some arbitary probability distribution, passes them through an arbitrary, nonlinear function and produces a Gaussian for each transformed points. I hope you can envision how we can use this to implement a nonlinear Kalman filter. Once we have Gaussians all of the mathematical apparatus we have already developed comes into play!

The name "unscented" might be confusing. It doesn't really mean much. It was a joke fostered by the inventor that his algorithm didn't "stink", and soon the name stuck. There is no mathematical meaning to the term.

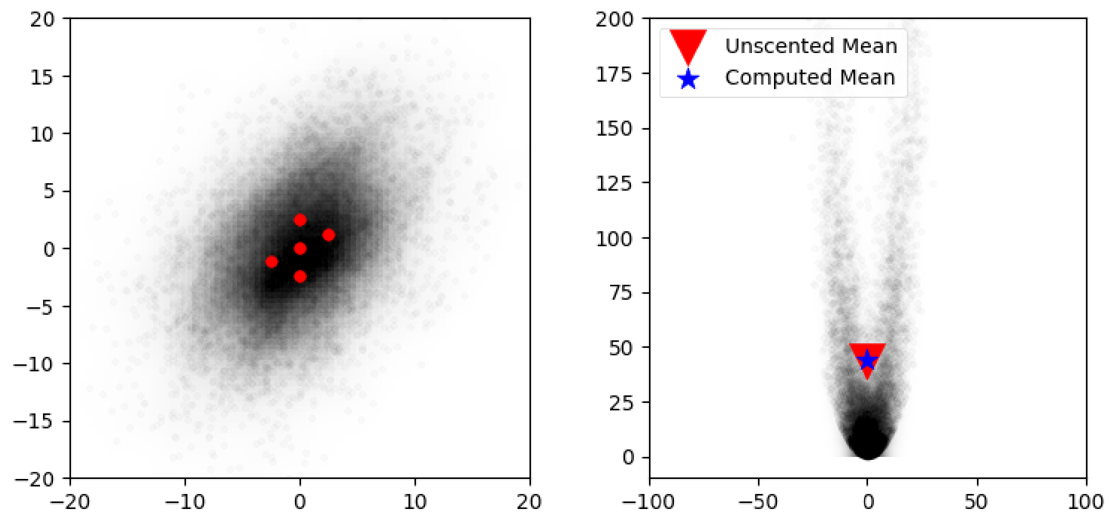
1.4.1 Accuracy of the Unscented Transform

Earlier we wrote a function that found the mean of a distribution by passing 50,000 points through a nonlinear function. Let's now pass 5 sigma points through the same function, and compute their mean with the unscented transform. We will use the FilterPy function *MerweScaledSigmaPoints()* to create the sigma points and `unscented_transform` to perform the transform; we will learn about these functions later. In the first example in this chapter I used `JulierSigmaPoints`; they both choose sigma points, but in different ways which I will explain later.

[25]:
```
from filterpy.kalman import unscented_transform, MerweScaledSigmaPoints
import scipy.stats as stats
```

```
#initial mean and covariance
mean = (0., 0.)
p = np.array([[32., 15], [15., 40.]])

# create sigma points and weights
points = MerweScaledSigmaPoints(n=2, alpha=.3, beta=2., kappa=.1)
sigmas = points.sigma_points(mean, p)

### pass through nonlinear function
sigmas_f = np.empty((5, 2))
for i in range(5):
    sigmas_f[i] = f_nonlinear_xy(sigmas[i, 0], sigmas[i ,1])

### use unscented transform to get new mean and covariance
ukf_mean, ukf_cov = unscented_transform(sigmas_f, points.Wm, points.Wc)

#generate random points
np.random.seed(100)
xs, ys = multivariate_normal(mean=mean, cov=p, size=5000).T

plot_monte_carlo_mean(xs, ys, f_nonlinear_xy, ukf_mean, 'Unscented Mean')
ax = plt.gcf().axes[0]
ax.scatter(sigmas[:,0], sigmas[:,1], c='r', s=30);
```

```
Difference in mean x=-0.097, y=0.549
```



I find this result remarkable. Using only 5 points we were able to compute the mean with amazing accuracy. The error in x is only -0.097, and the error in y is 0.549. In contrast, a linearized approach (used by the EKF, which we will learn in the next chapter) gave an error of over 43 in y.

If you look at the code that generates the sigma points you'll see that it has no knowledge of the nonlinear function, only of the mean and covariance of our initial distribution. The same 5 sigma points would be generated if we had a completely different nonlinear function.

I will admit to choosing a nonlinear function that makes the performance of the unscented tranform striking compared to the EKF. But the physical world is filled with very nonlinear behavior, and the UKF takes it in stride. I did not 'work' to find a function where the unscented transform happened to work well. You will see in the next chapter how more traditional techniques struggle with strong nonlinearities. This graph is the foundation of why I advise you to use the UKF or similar modern technique whenever possible.

1.5 The Unscented Kalman Filter

We can now present the UKF algorithm.

1.5.1 Predict Step

The UKF's predict step computes the prior using the process model $f()$. $f()$ is assumed to be nonlinear, so we generate sigma points $\mathcal{X}$ and their corresponding weights W^m, W^c according to some function:

$$\begin{aligned}\chi &= \text{sigma-function}(\mathbf{x}, \mathbf{P}) \\ W^m, W^c &= \text{weight-function}(\mathtt{n}, \mathtt{parameters})\end{aligned}$$

We pass each sigma point through $f(\mathbf{x}, \Delta t)$. This projects the sigma points forward in time according to the process model, forming the new prior, which is a set of sigma points we name $\mathcal{Y}$:

$$\mathcal{Y} = f(\chi, \Delta t)$$

We compute the mean and covariance of the prior using the *unscented transform* on the transformed sigma points.

$$\bar{\mathbf{x}}, \bar{\mathbf{P}} = UT(\mathcal{Y}, w_m, w_c, \mathbf{Q})$$

These are the equations for the unscented transform:

$$\begin{aligned}\bar{\mathbf{x}} &= \sum_{i=0}^{2n} w_i^m \mathcal{Y}_i \\ \bar{\mathbf{P}} &= \sum_{i=0}^{2n} w_i^c (\mathcal{Y}_i - \bar{\mathbf{x}})(\mathcal{Y}_i - \bar{\mathbf{x}})^\mathsf{T} + \mathbf{Q}\end{aligned}$$

This table compares the linear Kalman filter with the Unscented Kalman Filter equations. I've dropped the subscript i for readability.

Kalman	Unscented
	$\mathcal{Y} = f(\chi)$
$\bar{\mathbf{x}} = \mathbf{F}\mathbf{x}$	$\bar{\mathbf{x}} = \sum w^m \mathcal{Y}$
$\bar{\mathbf{P}} = \mathbf{F}\mathbf{P}\mathbf{F}^\mathsf{T} + \mathbf{Q}$	$\bar{\mathbf{P}} = \sum w^c (\mathcal{Y} - \bar{\mathbf{x}})(\mathcal{Y} - \bar{\mathbf{x}})^\mathsf{T} + \mathbf{Q}$

1.5.2 Update Step

Kalman filters perform the update in measurement space. Thus we must convert the sigma points of the prior into measurements using a measurement function $h(x)$ that you define.

$$\mathcal{Z} = h(\mathcal{Y})$$

We compute the mean and covariance of these points using the unscented transform. The z subscript denotes that these are the mean and covariance of the measurement sigma points.

$$\begin{aligned}
\mu_z, \mathbf{P}_z &= UT(\mathcal{Z}, w_m, w_c, \mathbf{R}) \\
\mu_z &= \sum_{i=0}^{2n} w_i^m \mathcal{Z}_i \\
\mathbf{P}_z &= \sum_{i=0}^{2n} w_i^c (\mathcal{Z}_i - \mu_z)(\mathcal{Z}_i - \mu_z)^\mathsf{T} + \mathbf{R}
\end{aligned}$$

Next we compute the residual and Kalman gain. The residual of the measurement $\mathbf{z}$ is trivial to compute:

$$\mathbf{y} = \mathbf{z} - \mu_z$$

To compute the Kalman gain we first compute the cross covariance of the state and the measurements, which is defined as:

$$\mathbf{P}_{xz} = \sum_{i=0}^{2n} w_i^c (\mathcal{Y}_i - \bar{\mathbf{x}})(\mathcal{Z}_i - \mu_z)^\mathsf{T}$$

And then the Kalman gain is defined as

$$\mathbf{K} = \mathbf{P}_{xz} \mathbf{P}_z^{-1}$$

If you think of the inverse as a *kind of* matrix reciprocal, you can see that the Kalman gain is a simple ratio which computes:

$$\mathbf{K} \approx \frac{\mathbf{P}_{xz}}{\mathbf{P}_z} \approx \frac{\text{belief in state}}{\text{belief in measurement}}$$

Finally, we compute the new state estimate using the residual and Kalman gain:

$$\mathbf{x} = \bar{\mathbf{x}} + \mathbf{Ky}$$

and the new covariance is computed as:

$$\mathbf{P} = \bar{\mathbf{P}} - \mathbf{KP_z K}^\mathsf{T}$$

This step contains a few equations you have to take on faith, but you should be able to see how they relate to the linear Kalman filter equations. The linear algebra is slightly different from the linear Kalman filter, but the algorithm is the same Bayesian algorithm we have been implementing throughout the book.

This table compares the equations of the linear KF and UKF equations.

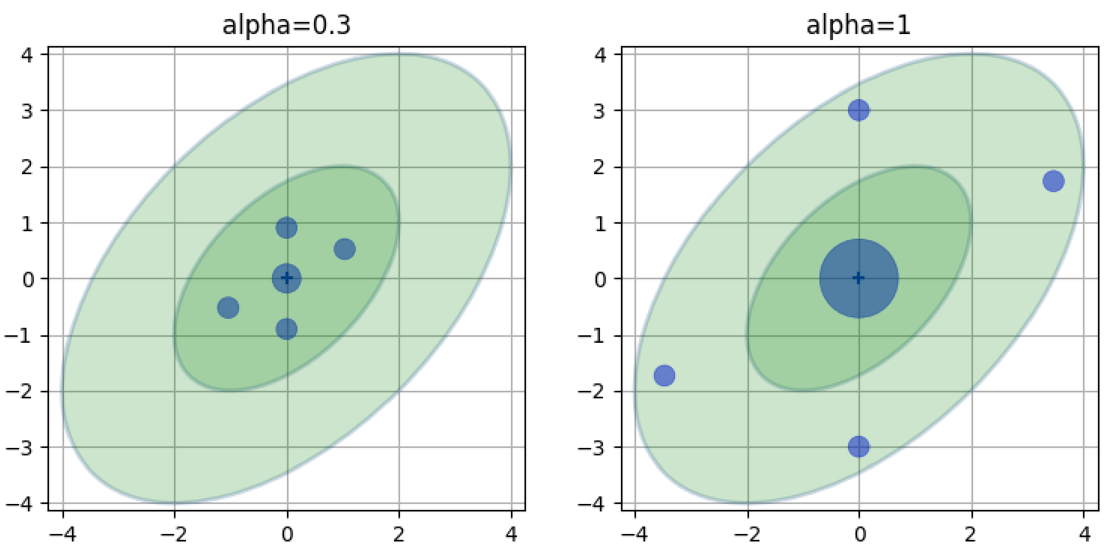
Kalman Filter	Unscented Kalman Filter
	$\mathcal{Y} = f(\chi)$
$\bar{\mathbf{x}} = \mathbf{Fx}$	$\bar{\mathbf{x}} = \sum w^m \mathcal{Y}$
$\bar{\mathbf{P}} = \mathbf{FPF}^\mathsf{T} + \mathbf{Q}$	$\bar{\mathbf{P}} = \sum w^c (\mathcal{Y} - \bar{\mathbf{x}})(\mathcal{Y} - \bar{\mathbf{x}})^\mathsf{T} + \mathbf{Q}$
	$\mathcal{Z} = h(\mathcal{Y})$
	$\mu_z = \sum w^m \mathcal{Z}$
$\mathbf{y} = \mathbf{z} - \mathbf{Hx}$	$\mathbf{y} = \mathbf{z} - \mu_z$
$\mathbf{S} = \mathbf{H}\bar{\mathbf{P}}\mathbf{H}^\mathsf{T} + \mathbf{R}$	$\mathbf{P}_z = \sum w^c (\mathcal{Z} - \mu_z)(\mathcal{Z} - \mu_z)^\mathsf{T} + \mathbf{R}$
$\mathbf{K} = \bar{\mathbf{P}}\mathbf{H}^\mathsf{T}\mathbf{S}^{-1}$	$\mathbf{K} = \left[\sum w^c (\mathcal{Y} - \bar{\mathbf{x}})(\mathcal{Z} - \mu_z)^\mathsf{T}\right] \mathbf{P}_z^{-1}$
$\mathbf{x} = \bar{\mathbf{x}} + \mathbf{Ky}$	$\mathbf{x} = \bar{\mathbf{x}} + \mathbf{Ky}$
$\mathbf{P} = (\mathbf{I} - \mathbf{KH})\bar{\mathbf{P}}$	$\mathbf{P} = \bar{\mathbf{P}} - \mathbf{KP_z K}^\mathsf{T}$

1.6 Van der Merwe's Scaled Sigma Point Algorithm

There are many algorithms for selecting sigma points. Since 2005 or so research and industry have mostly settled on the version published by Rudolph Van der Merwe in his 2004 PhD dissertation [1]. It performs well with a variety of problems and it has a good tradeoff between performance and accuracy. It is a slight reformulation of the *Scaled Unscented Transform* published by Simon J. Julier [2].

This formulation uses 3 parameters to control how the sigma points are distributed and weighted: α, β, and κ. Before we work through the equations, let's look at an example. I will plot the sigma points on top of a covariance ellipse showing the first and second standard deviations, and scale the points based on the mean weights.

[26]:
```
ukf_internal.plot_sigma_points()
```



We can see that the sigma points lie between the first and second standard deviation, and that the larger α spreads the points out. Furthermore, the larger α weights the mean (center point) higher than the smaller α, and weights the rest less. This should fit our intuition - the further a point is from the mean the less we should weight it. We don't know how these weights and sigma points are selected yet, but the choices look reasonable.

1.6.1 Sigma Point Computation

The first sigma point is the mean of the input. This is the sigma point displayed in the center of the ellipses in the diagram above. We will call this χ_0.

$$\mathcal{X}_0 = \mu$$

For notational convenience we define $\lambda = \alpha^2(n+\kappa) - n$, where n is the dimension of $\mathbf{x}$. The remaining sigma points are computed as

$$\chi_i = \begin{cases} \mu + \left[\sqrt{(n+\lambda)\Sigma}\right]_i & i = 1..n \\ \mu - \left[\sqrt{(n+\lambda)\Sigma}\right]_{i-n} & i = (n+1)..2n \end{cases}$$

The i subscript chooses the ith row vector of the matrix.

In other words, we scale the covariance matrix by a constant, take the square root of it, and ensure symmetry by both adding and subtracting it from the mean. We will discuss how you take the square root of a matrix later.

1.6.2 Weight Computation

This formulation uses one set of weights for the means, and another set for the covariance. The weights for the mean of $\mathcal{X}_0$ is computed as

$$W_0^m = \frac{\lambda}{n+\lambda}$$

The weight for the covariance of $\mathcal{X}_0$ is

$$W_0^c = \frac{\lambda}{n+\lambda} + 1 - \alpha^2 + \beta$$

The weights for the rest of the sigma points $\chi_1 ... \chi_{2n}$ are the same for the mean and covariance. They are

$$W_i^m = W_i^c = \frac{1}{2(n+\lambda)} \quad i = 1..2n$$

It may not be obvious why this is 'correct', and indeed, it cannot be proven that this is ideal for all nonlinear problems. But you can see that we are choosing the sigma points proportional to the square root of the covariance matrix, and the square root of variance is standard deviation. So, the sigma points are spread roughly according to $\pm 1\sigma$ times some scaling factor. There is an n term in the denominator, so with more dimensions the points will be spread out and weighed less.

Important note: Ordinarily these weights do not sum to one. I get many questions about this. Getting weights that sum to greater than one, or even negative values is expected. I cover this in more detail below.

1.6.3 Reasonable Choices for the Parameters

$\beta = 2$ is a good choice for Gaussian problems, $\kappa = 3 - n$ where n is the dimension of $\mathbf{x}$ is a good choice for κ, and $0 \le \alpha \le 1$ is an appropriate choice for α, where a larger value for α spreads the sigma points further from the mean.

1.7 Using the UKF

Let's solve some problems so you can gain confidence in how easy the UKF is to use. We will start with a linear problem you already know how to solve with the linear Kalman filter. Although the UKF was designed for nonlinear problems, it finds the same optimal result as the linear Kalman filter for linear problems. We will write a filter to track an object in 2D using a constant velocity model. This will allow us to focus on what is the same (and most is the same!) and what is different with the UKF.

Designing a Kalman filter requires you to specify the $\mathbf{x}$, $\mathbf{F}$, $\mathbf{H}$, $\mathbf{R}$, and $\mathbf{Q}$ matrices. We have done this many times so I will give you the matrices without a lot of discussion. We want a constant velocity model, so we define $\mathbf{x}$ to be

$$\mathbf{x} = \begin{bmatrix} x & \dot{x} & y & \dot{y} \end{bmatrix}^\mathsf{T}$$

With this ordering of state variables the state transition matrix is

$$\mathbf{F} = \begin{bmatrix}1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1\end{bmatrix}$$

which implement the Newtonian equations

$$\begin{aligned}x_k &= x_{k-1} + \dot{x}_{k-1}\Delta t \\ y_k &= y_{k-1} + \dot{y}_{k-1}\Delta t\end{aligned}$$

Our sensors provide position but not velocity, so the measurement function is

$$\mathbf{H} = \begin{bmatrix}1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0\end{bmatrix}$$

The sensor readings are in meters with an error of $\sigma = 0.3$ meters in both x and y. This gives us a measurement noise matrix of

$$\mathbf{R} = \begin{bmatrix}0.3^2 & 0 \\ 0 & 0.3^2\end{bmatrix}$$

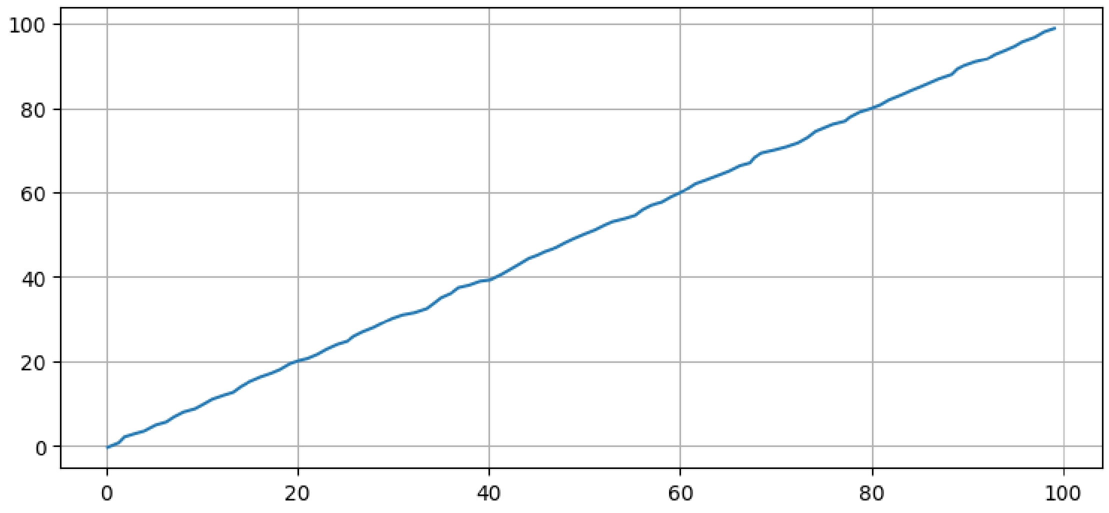
Finally, let's assume that the process noise can be represented by the discrete white noise model - that is, that over each time period the acceleration is constant. We can use `FilterPy`'s `Q_discrete_white_noise()` to create this matrix for us, but for review the matrix is

$$\mathbf{Q} = \begin{bmatrix}\frac{1}{4}\Delta t^4 & \frac{1}{2}\Delta t^3 \\ \frac{1}{2}\Delta t^3 & \Delta t^2\end{bmatrix}\sigma^2$$

My implementation of this filter is:

[27]:
```python
from filterpy.kalman import KalmanFilter
from filterpy.common import Q_discrete_white_noise
from numpy.random import randn

std_x, std_y = .3, .3
dt = 1.0

np.random.seed(1234)
kf = KalmanFilter(4, 2)
kf.x = np.array([0., 0., 0., 0.])
kf.R = np.diag([std_x**2, std_y**2])
kf.F = np.array([[1, dt, 0, 0],
                 [0, 1, 0, 0],
                 [0, 0, 1, dt],
                 [0, 0, 0, 1]])
kf.H = np.array([[1, 0, 0, 0],
                 [0, 0, 1, 0]])
```

```
kf.Q[0:2, 0:2] = Q_discrete_white_noise(2, dt=1, var=0.02)
kf.Q[2:4, 2:4] = Q_discrete_white_noise(2, dt=1, var=0.02)

zs = [np.array([i + randn()*std_x,
                i + randn()*std_y]) for i in range(100)]
xs, _, _, _ = kf.batch_filter(zs)
plt.plot(xs[:, 0], xs[:, 2]);
```



This should hold no surprises for you. Now let's implement a UKF. Again, this is purely for educational purposes; using a UKF for a linear problem confers no benefit. `FilterPy` implements the UKF with the class `UnscentedKalmanFilter`.

The first thing to do is implement the functions `f(x, dt)` and `h(x)`. `f(x, dt)` implements the state transition function, and `h(x)` implements the measurement function. These correspond to the matrices **F** and **H** in the linear filter.

Below is a reasonable implementation of these two functions. Each is expected to return a 1D NumPy array or list containing the result. You may give them more readable names than `f` and `h`.

[28]:
```
def f_cv(x, dt):
    """ state transition function for a
    constant velocity aircraft"""

    F = np.array([[1, dt, 0,  0],
                  [0,  1, 0,  0],
                  [0,  0, 1, dt],
                  [0,  0, 0,  1]])
    return F @ x

def h_cv(x):
    return x[[0, 2]]
```

Next you specify how to compute the sigma points and weights. We gave Van der Merwe's version above, but there are many different choices. FilterPy uses a class named `SigmaPoints` which must implement a method:

```
def sigma_points(self, x, P)
```

and contain attributes `Wm` and `Wc`, which hold the weights for computing the mean and covariance, respectively.

FilterPy derives the class `MerweScaledSigmaPoints` from `SigmaPoints` and implements the aforementioned methods.

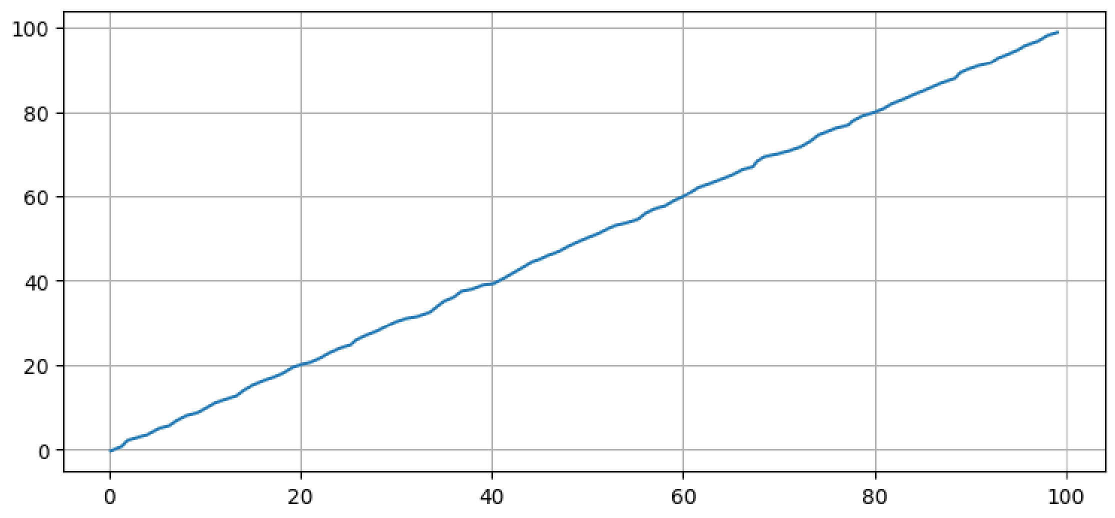
When you create the UKF you will pass in the $f()$ and $h()$ functions and the sigma point object, as in this example:

```
from filterpy.kalman import MerweScaledSigmaPoints
from filterpy.kalman import UnscentedKalmanFilter as UKF

points = MerweScaledSigmaPoints(n=4, alpha=.1, beta=2., kappa=-1)
ukf = UKF(dim_x=4, dim_z=2, fx=f_cv, hx=h_cv, dt=dt, points=points)
```

The rest of the code is the same as for the linear kalman filter. I'll use the same measurements and compute the standard deviation of the difference between the two solutions.

[29]:
```
from filterpy.kalman import UnscentedKalmanFilter as UKF

import numpy as np

sigmas = MerweScaledSigmaPoints(4, alpha=.1, beta=2., kappa=1.)
ukf = UKF(dim_x=4, dim_z=2, fx=f_cv,
          hx=h_cv, dt=dt, points=sigmas)
ukf.x = np.array([0., 0., 0., 0.])
ukf.R = np.diag([0.09, 0.09])
ukf.Q[0:2, 0:2] = Q_discrete_white_noise(2, dt=1, var=0.02)
ukf.Q[2:4, 2:4] = Q_discrete_white_noise(2, dt=1, var=0.02)

uxs = []
for z in zs:
    ukf.predict()
    ukf.update(z)
    uxs.append(ukf.x.copy())
uxs = np.array(uxs)

plt.plot(uxs[:, 0], uxs[:, 2])
print(f'UKF standard deviation {np.std(uxs - xs):.3f} meters')
```

```
UKF standard deviation 0.013 meters
```



This gave me a standard deviation of 0.013 meters, which is quite small.

The implementation of the UKF is not that different from the linear Kalman filter. Instead of implementing the state transition and measurement functions as the matrices **F** and **H** you supply nonlinear functions `f()` and `h()`. The rest of the theory and implementation remains the same. The code implementing `predict()` and `update()` differs, but from a designer's point of view the problem formulation and filter design is very similar.

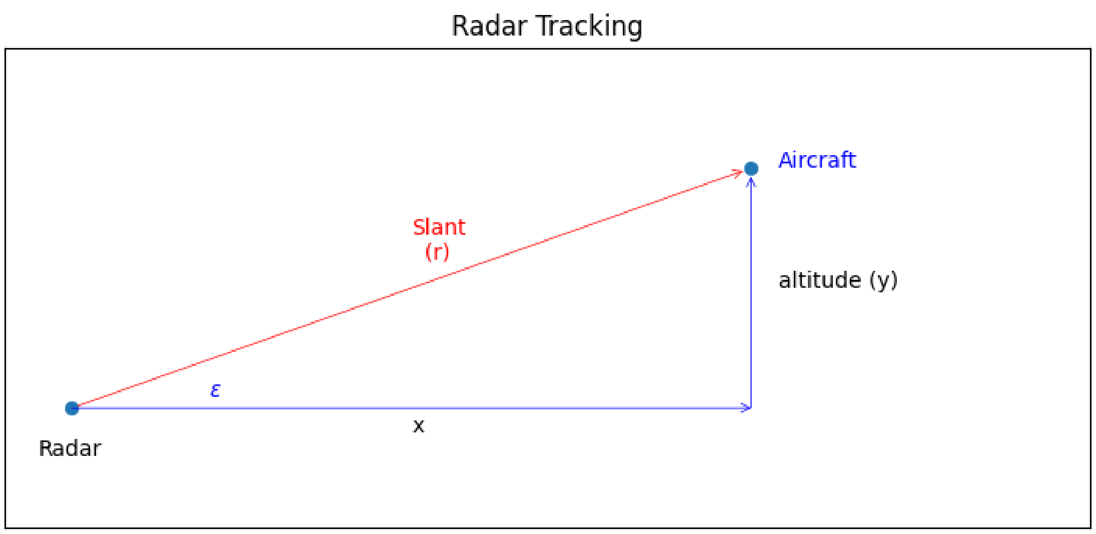
1.8 Tracking an Airplane

Let's tackle our first nonlinear problem. We will write a filter to track an airplane using radar as the sensor. To keep the problem as similar to the previous one as possible we will track in two dimensions. We will track one dimension on the ground and the altitude of the aircraft. Each dimension is independent so we can do this with no loss of generality.

Radars work by emitting radio waves or microwaves. Anything in the beam's path will reflect some of the signal back to the radar. By timing how long it takes for the reflected signal to return it can compute the *slant distance* to the target. Slant distance is the straight line distance from the radar to the object. Bearing is computed using the *directive gain* of the antenna.

We compute the (x,y) position of the aircraft from the slant distance and elevation angle as illustrated by this diagram:

[30]:
```
import kf_book.ekf_internal as ekf_internal
ekf_internal.show_radar_chart()
```



The *elevation angle* ϵ is the angle above the line of sight formed by the ground.

We will assume that the aircraft is flying at a constant altitude. Thus we have a three variable state vector:

$$\mathbf{x} = \begin{bmatrix}\mathtt{distance} \\ \mathtt{velocity} \\ \mathtt{altitude}\end{bmatrix} = \begin{bmatrix}x \\ \dot{x} \\ y\end{bmatrix}$$

The state transition function is linear

$$\bar{\mathbf{x}} = \begin{bmatrix}1 & \Delta t & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1\end{bmatrix} \begin{bmatrix}x \\ \dot{x} \\ y\end{bmatrix}$$

and can be computed with:

[31]:
```python
def f_radar(x, dt):
    """ state transition function for a constant velocity
    aircraft with state vector [x, velocity, altitude]'"""

    F = np.array([[1, dt, 0],
                  [0,  1, 0],
                  [0,  0, 1]], dtype=float)
    return F @ x
```

Next we design the measurement function. As in the linear Kalman filter the measurement function converts the filter's prior into a measurement. We need to convert the position and velocity of the aircraft into the elevation angle and range from the radar station.

Range is computed with the Pythagorean theorem:

$$\text{range} = \sqrt{(x_{\text{ac}} - x_{\text{radar}})^2 + (y_{\text{ac}} - y_{\texttt{radar}})^2}$$

The elevation angle ϵ is the arctangent of y/x:

$$\epsilon = \tan^{-1} \frac{y_{\texttt{ac}} - y_{\text{radar}}}{x_{\text{ac}} - x_{\text{radar}}}$$

We need to define a Python function to compute this. I'll take advantage of the fact that a function can own a variable to store the radar's position. While this isn't necessary for this problem (we could hard code the value, or use a global), this gives the function more flexibility.

[32]:
```python
def h_radar(x):
    dx = x[0] - h_radar.radar_pos[0]
    dy = x[2] - h_radar.radar_pos[1]
    slant_range = math.sqrt(dx**2 + dy**2)
    elevation_angle = math.atan2(dy, dx)
    return [slant_range, elevation_angle]

h_radar.radar_pos = (0, 0)
```

There is a nonlinearity that we are not considering, the fact that angles are modular. The residual is the difference between the measurement and the prior projected into measurement space. The angular difference between 359° and 1° is 2°, but 359° - 1° = 358°. This is exacerbated by the UKF which computes sums of weighted values in the unscented transform. For now we will place our sensors and targets in positions that avoid these nonlinear regions. Later I will show you how to handle this problem.

We need to simulate the radar and the aircraft. By now this should be second nature for you, so I offer the code without discussion.

[33]:
```python
from numpy.linalg import norm
from math import atan2

class RadarStation:

    def __init__(self, pos, range_std, elev_angle_std):
        self.pos = np.asarray(pos)
        self.range_std = range_std
        self.elev_angle_std = elev_angle_std


    def reading_of(self, ac_pos):
        """ Returns (range, elevation angle) to aircraft.
        Elevation angle is in radians.
        """
        diff = np.subtract(ac_pos, self.pos)
        rng = norm(diff)
        brg = atan2(diff[1], diff[0])
        return rng, brg


    def noisy_reading(self, ac_pos):
        """ Compute range and elevation angle to aircraft with
        simulated noise"""

        rng, brg = self.reading_of(ac_pos)
        rng += randn() * self.range_std
        brg += randn() * self.elev_angle_std
        return rng, brg

class ACSim:
    def __init__(self, pos, vel, vel_std):
        self.pos = np.asarray(pos, dtype=float)
        self.vel = np.asarray(vel, dtype=float)
        self.vel_std = vel_std

    def update(self, dt):
        """ Compute and returns next position. Incorporates
        random variation in velocity. """

        dx = self.vel*dt + (randn() * self.vel_std) * dt
        self.pos += dx
        return self.pos
```

A military grade radar achieves 1 meter RMS range accuracy, and 1 mrad RMS for elevation angle [3]. We will assume a more modest 5 meter range accuracy, and 0.5° angular accuracy as this provides a more challenging data set for the filter.

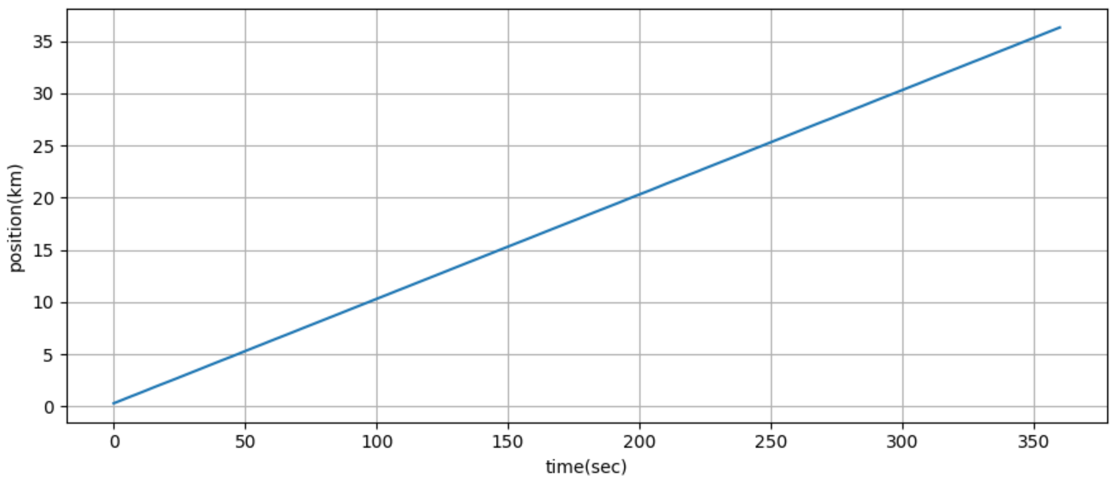
The design of $\mathbf{Q}$ requires some discussion. The state is $\begin{bmatrix}x & \dot{x} & y\end{bmatrix}^\mathsf{T}$. The first two elements are down range distance and velocity, so we can use `Q_discrete_white_noise` noise to compute the values for the upper left hand side of Q. The third element is altitude, which we assume is independent of x. That results in a block design for $\mathbf{Q}$:

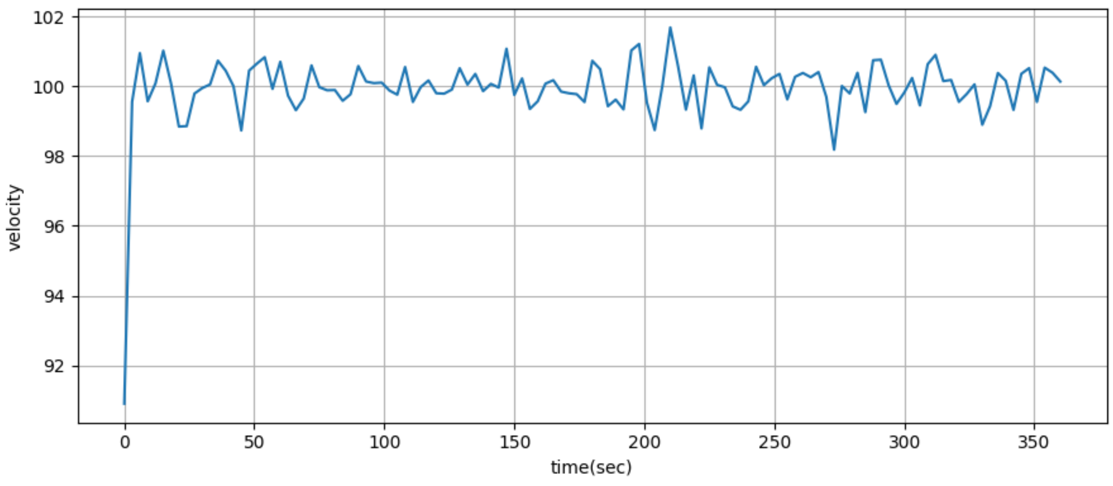
$$\mathbf{Q} = \begin{bmatrix}\mathbf{Q}_\mathtt{x} & 0 \\ 0 & Q_\mathtt{y}\end{bmatrix}$$

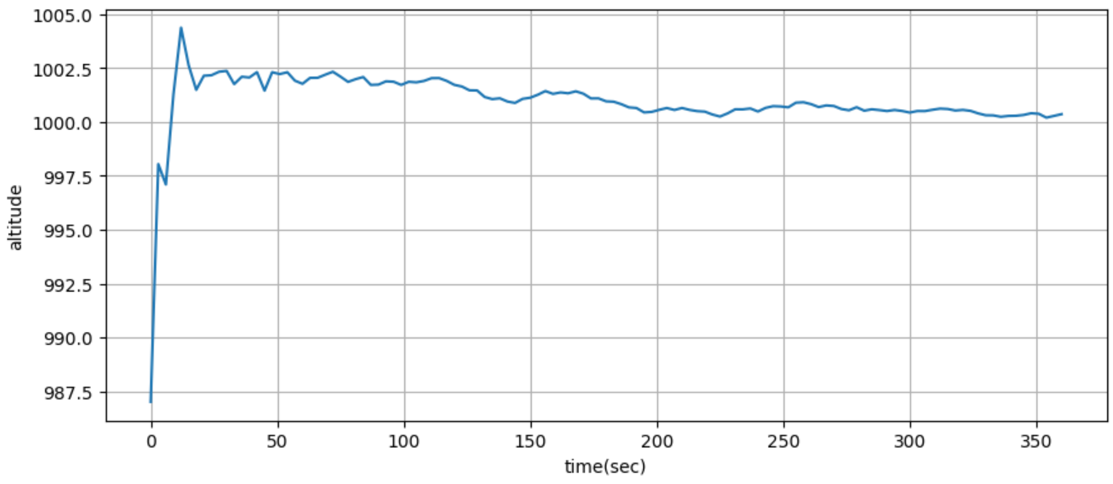
I'll start with the aircraft positioned directly over the radar station, flying at 100 m/s. A typical height finder radar might update only once every 3 seconds so we will use that for our epoch period.

[34]:
```
import math
from kf_book.ukf_internal import plot_radar

dt = 3. # 12 seconds between readings
```

```
range_std = 5 # meters
elevation_angle_std = math.radians(0.5)
ac_pos = (0., 1000.)
ac_vel = (100., 0.)
radar_pos = (0., 0.)
h_radar.radar_pos = radar_pos

points = MerweScaledSigmaPoints(n=3, alpha=.1, beta=2., kappa=0.)
kf = UKF(3, 2, dt, fx=f_radar, hx=h_radar, points=points)

kf.Q[0:2, 0:2] = Q_discrete_white_noise(2, dt=dt, var=0.1)
kf.Q[2,2] = 0.1

kf.R = np.diag([range_std**2, elevation_angle_std**2])
kf.x = np.array([0., 90., 1100.])
kf.P = np.diag([300**2, 30**2, 150**2])

np.random.seed(200)
pos = (0, 0)
radar = RadarStation(pos, range_std, elevation_angle_std)
ac = ACSim(ac_pos, (100, 0), 0.02)

time = np.arange(0, 360 + dt, dt)
xs = []
for _ in time:
    ac.update(dt)
    r = radar.noisy_reading(ac.pos)
    kf.predict()
    kf.update([r[0], r[1]])
    xs.append(kf.x)
plot_radar(xs, time)
```







This may or may not impress you, but it impresses me! In the Extended Kalman filter chapter we will solve the same problem, but it will take a significant amount of mathematics.

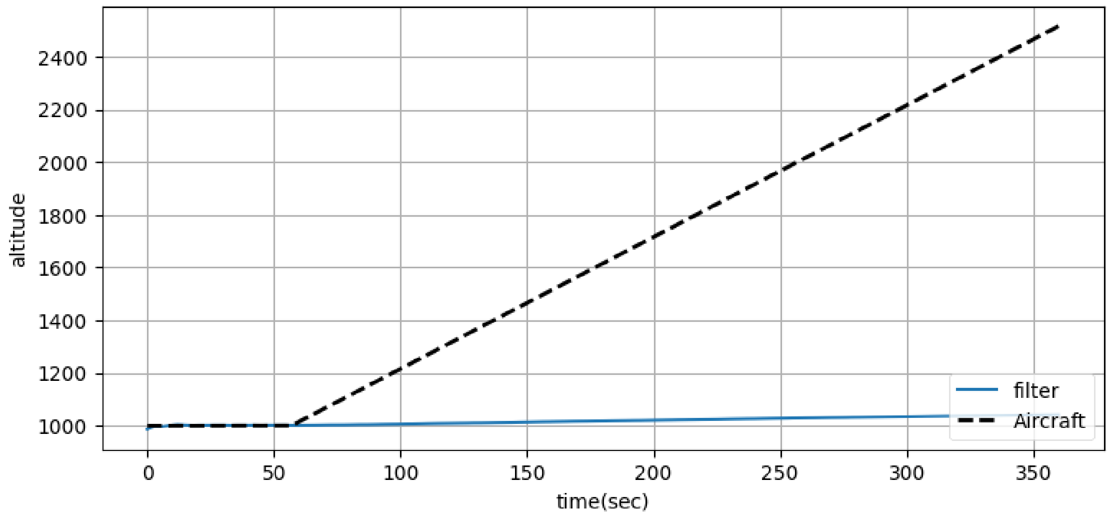
1.8.1 Tracking Maneuvering Aircraft

The previous example produced good results, but it assumed the aircraft did not change altitude. Here are the filter results if the aircraft starts climbing after one minute.

[35]:
```
from kf_book.ukf_internal import plot_altitude

# reset aircraft position
kf.x = np.array([0., 90., 1100.])
kf.P = np.diag([300**2, 30**2, 150**2])
```

```python
ac = ACSim(ac_pos, (100, 0), 0.02)

np.random.seed(200)
time = np.arange(0, 360 + dt, dt)
xs, ys = [], []
for t in time:
    if t >= 60:
        ac.vel[1] = 300/60 # 300 meters/minute climb
    ac.update(dt)
    r = radar.noisy_reading(ac.pos)
    ys.append(ac.pos[1])
    kf.predict()
    kf.update([r[0], r[1]])
    xs.append(kf.x)

plot_altitude(xs, time, ys)
print(f'Actual altitude: {ac.pos[1]:.1f}')
print(f'UKF altitude    : {xs[-1][2]:.1f}')
```

```
Actual altitude: 2515.6
UKF altitude    : 1042.1
```



The filter is unable to track the changing altitude. What do we have to change in our design?

I hope you answered "add climb rate to the state", like so:

$$\mathbf{x} = \begin{bmatrix}\texttt{distance} \\ \texttt{velocity} \\ \texttt{altitude} \\ \texttt{climb rate}\end{bmatrix} = \begin{bmatrix}x \\ \dot{x} \\ y \\ \dot{y}\end{bmatrix}$$

This requires the following change to the state transition function, which is still linear.

$$\mathbf{F} = \begin{bmatrix}1 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 1\end{bmatrix} \begin{bmatrix}x \\ \dot{x} \\ y \\ \dot{y}\end{bmatrix}$$

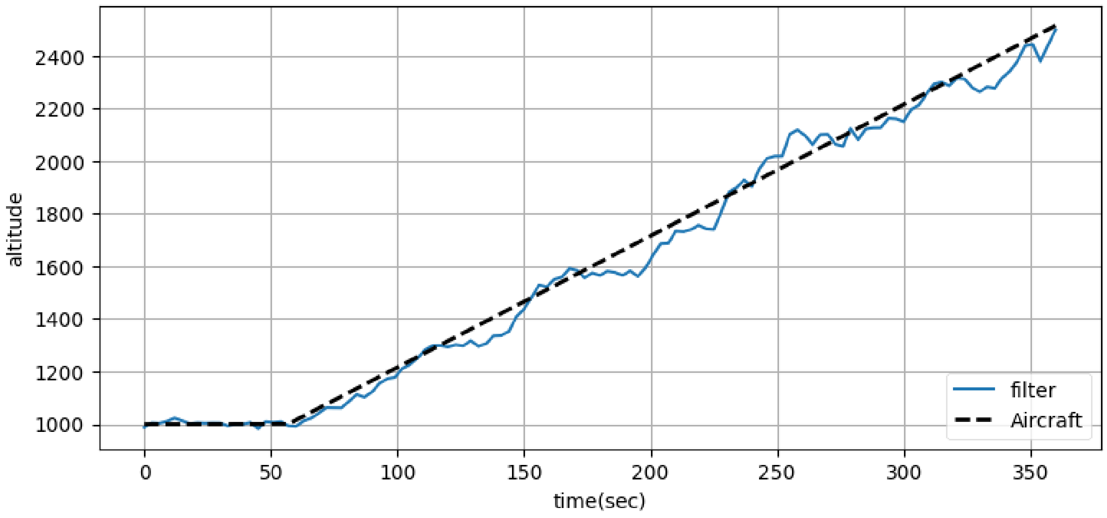
The measurement function stays the same, but we must alter **Q** to account for the dimensionality change of **x**.

[36]:
```
def f_cv_radar(x, dt):
    """ state transition function for a constant velocity
    aircraft"""
    F = np.array([[1, dt, 0, 0],
                  [0,  1, 0, 0],
                  [0,  0, 1, dt],
                  [0,  0, 0, 1]], dtype=float)
    return F @ x

def cv_UKF(fx, hx, R_std):
    points = MerweScaledSigmaPoints(n=4, alpha=.1, beta=2., kappa=-1.)
    kf = UKF(4, len(R_std), dt, fx=fx, hx=hx, points=points)

    kf.Q[0:2, 0:2] = Q_discrete_white_noise(2, dt=dt, var=0.1)
    kf.Q[2:4, 2:4] = Q_discrete_white_noise(2, dt=dt, var=0.1)
    kf.R = np.diag(R_std)
    kf.R = kf.R @ kf.R  # square to get variance
    kf.x = np.array([0., 90., 1100., 0.])
    kf.P = np.diag([300**2, 3**2, 150**2, 3**2])
    return kf
```

[37]:
```
np.random.seed(200)
ac = ACSim(ac_pos, (100, 0), 0.02)

kf_cv = cv_UKF(f_cv_radar, h_radar, R_std=[range_std, elevation_angle_std])
time = np.arange(0, 360 + dt, dt)
xs, ys = [], []
for t in time:
    if t >= 60:
        ac.vel[1] = 300/60 # 300 meters/minute climb
    ac.update(dt)
    r = radar.noisy_reading(ac.pos)
    ys.append(ac.pos[1])
    kf_cv.predict()
    kf_cv.update([r[0], r[1]])
    xs.append(kf_cv.x)
```

```
plot_altitude(xs, time, ys)
print(f'Actual altitude: {ac.pos[1]:.1f}')
print(f'UKF altitude    : {xs[-1][2]:.1f}')
```

```
Actual altitude: 2515.6
UKF altitude    : 2500.1
```



A significant amount of noise has been introduced into the altitude estimate, but we are now accurately tracking altitude.

1.8.2 Sensor Fusion

Now let's consider an example of sensor fusion. We have some type of Doppler system that produces a velocity estimate with 2 m/s RMS accuracy. I say "some type" because as with the radar I am not trying to teach you how to create an accurate filter for a Doppler system. A full implementation must account for the signal to noise ratio, atmospheric effects, the geometry of the system, and so on.

The radar's accuracy in the last examples allowed us to estimate velocities to within one m/s or so, I will degrade that accuracy to illustrate the effect of sensor fusion. Let's change the range error to $\sigma = 500$ meters and then compute the standard deviation of the estimated velocity. I'll skip the first several measurements because the filter is converging during that time, causing artificially large deviations.

The standard deviation without using Doppler is:

[ ]:
```
range_std = 500.
elevation_angle_std = math.degrees(0.5)
np.random.seed(200)
pos = (0, 0)
radar = RadarStation(pos, range_std, elevation_angle_std)
```

```
ac = ACSim(ac_pos, (100, 0), 0.02)

kf_sf = cv_UKF(f_cv_radar, h_radar, R_std=[range_std, elevation_angle_std])
time = np.arange(0, 360 + dt, dt)
xs = []
for _ in time:
    ac.update(dt)
    r = radar.noisy_reading(ac.pos)
    kf_sf.predict()
    kf_sf.update([r[0], r[1]])
    xs.append(kf_sf.x)

xs = np.asarray(xs)
plot_radar(xs, time, plot_x=False, plot_vel=True, plot_alt=False)
print(f'Velocity std {np.std(xs[10:, 1]):.1f} m/s')
```

For Doppler we need to include the velocity in x and y into the measurement. The `ACSim` class stores velocity in the data member `vel`. To perform the Kalman filter update we just need to call `update` with a list containing the slant distance, elevation angle, and velocity in x and y:

$$z = [\texttt{slant\_range}, \text{elevation angle}, \dot{x}, \dot{y}]$$

The measurement contains four values so the measurement function also needs to return four values. The slant range and elevation angle will be computed as before, and we do not need to compute the velocity in x and y as they are provided by the state estimate.

```
[ ]: def h_vel(x):
         dx = x[0] - h_vel.radar_pos[0]
         dz = x[2] - h_vel.radar_pos[1]
         slant_range = math.sqrt(dx**2 + dz**2)
         elevation_angle = math.atan2(dz, dx)
         return slant_range, elevation_angle, x[1], x[3]
```

Now we can implement our filter.

```
[ ]: h_radar.radar_pos = (0, 0)
     h_vel.radar_pos = (0, 0)

     range_std = 500.
     elevation_angle_std = math.degrees(0.5)
     vel_std = 2.

     np.random.seed(200)
     ac = ACSim(ac_pos, (100, 0), 0.02)
     radar = RadarStation((0, 0), range_std, elevation_angle_std)

     kf_sf2 = cv_UKF(f_cv_radar, h_vel,
                 R_std=[range_std, elevation_angle_std, vel_std, vel_std])

     time = np.arange(0, 360 + dt, dt)
     xs = []
     for t in time:
         ac.update(dt)
         r = radar.noisy_reading(ac.pos)
         # simulate the doppler velocity reading
         vx = ac.vel[0] + randn()*vel_std
         vz = ac.vel[1] + randn()*vel_std
         kf_sf2.predict()
         kf_sf2.update([r[0], r[1], vx, vz])
         xs.append(kf_sf2.x)
     xs = np.asarray(xs)
     plot_radar(xs, time, plot_x=False, plot_vel=True, plot_alt=False)
     print(f'Velocity std {np.std(xs[10:,1]):.1f} m/s')
```

By incorporating the velocity sensor we were able to reduce the standard deviation from 3.5 m/s to 1.3 m/s.

Sensor fusion is a large topic, and this is a rather simplistic implementation. In a typical navigation problem we have sensors that provide *complementary* information. For example, a GPS might provide somewhat accurate position updates once a second with poor velocity estimation while an inertial system might provide very accurate velocity updates at 50Hz but terrible position estimates. The strengths and weaknesses of each sensor are orthogonal to each other. This leads to the *Complementary filter*, which blends the high update rate inertial velocity measurements with the accurate but slowly updated position estimates of the GPS to produce high rate and accurate position and velocity estimates. The high rate velocity estimates are integrated between the GPS updates to produce accurate and high rate position estimates.

1.8.3 Multiple Position Sensors

The last sensor fusion problem was a toy example. Let's tackle a problem that is not so toy-like. Before GPS ships and aircraft navigated via various range and bearing systems such as VOR, LORAN, TACAN, DME, and so on. These systems emit beacons in the form of radio waves. The sensor extracts the range and/or bearing to the beacon from the signal. For example, an aircraft might have two VOR receivers. The pilot tunes each receiver to a different VOR station. Each VOR receiver displays the *radial* - the direction from the VOR station on the ground to the aircraft. The pilot uses a chart to find the intersection point of the radials, which identifies the location of the aircraft.

That is a manual approach with low accuracy. A Kalman filter will produce far more accurate position estimates. Assume we have two sensors, each of which provides a bearing only measurement to the target, as in the chart below. The width of the perimeters are proportional to the 3σ of the sensor noise. The aircraft must be positioned somewhere within the intersection of the two perimeters with a high degree of probability.

```
[ ]: ukf_internal.show_two_sensor_bearing()
```

We compute the bearing between a sensor and the target as:

```
[ ]: def bearing(sensor, target_pos):
         return math.atan2(target_pos[1] - sensor[1],
                           target_pos[0] - sensor[0])
```

The filter receives the measurement from the two sensors in a vector. The code will accept any iterable container, so I use a Python list for efficiency. We can implement that as:

```
[ ]: def measurement(A_pos, B_pos, pos):
         angle_a = bearing(A_pos, pos)
         angle_b = bearing(B_pos, pos)
         return [angle_a, angle_b]
```

Assume a constant velocity model for the aircraft. For a change of pace I compute the new positions explicitly rather than using matrix-vector multiplication:

```
[ ]: def fx_VOR(x, dt):
         x[0] += x[1] * dt
         x[2] += x[3] * dt
         return x
```

Next we implement the measurement function. It converts the prior to an array containing the measurement to both stations. I'm not a fan of global variables, but I put the position of the stations in the global variables `sa_pos` and `sb_pos` to demonstrate this method of sharing data with $h()$:

```
[ ]: sa_pos = [-400, 0]
     sb_pos = [400, 0]

     def hx_VOR(x):
         # measurement to A
         pos = (x[0], x[2])
         return measurement(sa_pos, sb_pos, pos)
```

Now we write boilerplate which constructs the filter, runs it, and plots the results:

```
[ ]: def moving_target_filter(pos, std_noise, Q, dt=0.1, kappa=0.0):
         points = MerweScaledSigmaPoints(n=4, alpha=.1, beta=2., kappa=kappa)
         f = UKF(dim_x=4, dim_z=2, dt=dt,
                 hx=hx_VOR, fx=fx_VOR, points=points)
         f.x = np.array([pos[0], 1., pos[1], 1.])

         q = Q_discrete_white_noise(2, dt, Q)
         f.Q[0:2, 0:2] = q
         f.Q[2:4, 2:4] = q
         f.R *= std_noise**2
         f.P *= 1000
         return f
```

```
def plot_straight_line_target(f, std_noise):
    xs, txs = [], []
    for i in range(300):
        target_pos[0] += 1 + randn()*0.0001
        target_pos[1] += 1 + randn()*0.0001
        txs.append((target_pos[0], target_pos[1]))

        z = measurement(sa_pos, sb_pos, target_pos)
        z[0] += randn() * std_noise
        z[1] += randn() * std_noise

        f.predict()
        f.update(z)
        xs.append(f.x)

    xs = np.asarray(xs)
    txs = np.asarray(txs)

    plt.plot(xs[:, 0], xs[:, 2])
    plt.plot(txs[:, 0], txs[:, 1], ls='--', lw=2, c='k')
    plt.show()

np.random.seed(123)
target_pos = [100, 200]

std_noise = math.radians(0.5)
f = moving_target_filter(target_pos, std_noise, Q=1.0)
plot_straight_line_target(f, std_noise)
```

This looks quite good to me. The beginning of the track exhibits large errors, but the filter settles down and produces good estimates.

Let's revisit the nonlinearity of the angles. I will position the target between the two sensors at (0,0). This will cause a nonlinearity in the computation of the residuals because the mean angle will be near zero. As the angle goes below 0 the measurement function will compute a large positive angle of nearly 2π. The residual between the prediction and measurement will thus be very large, nearly 2π instead of nearly 0. This makes it impossible for the filter to perform accurately, as seen in the example below.

[ ]:
```
target_pos = [0, 0]
f = moving_target_filter(target_pos, std_noise, Q=1.0)
plot_straight_line_target(f, std_noise)
```

This performance is unacceptable. `FilterPy`'s UKF code allows you to specify a function which computes the residuals in cases of nonlinear behavior like this,. The final example in this chapter demonstrates its use.

1.9 Effects of Sensor Error and Geometry

The geometry of the sensors relative to the tracked object imposes a physical limitation that can be extremely difficult to deal with when designing filters. If the radials of the VOR stations are nearly parallel to each other then a very small angular error translates into a very large distance error. What is worse, this behavior is nonlinear - the error in the *x-axis* vs the *y-axis* will vary depending on the actual bearing. These scatter plots show the error distribution for a $1°\sigma$ error for two different bearings.

```
[ ]: ukf_internal.plot_scatter_of_bearing_error()
```

1.10 Exercise: Explain Filter Performance

We can see that for small angular errors the positional errors are very large. Explain how we got such relatively good performance out of the UKF in the target tracking problems above. Answer for both the one sensor and multiple sensor problem.

1.10.1 Solution

This is very important to understand. Try very hard to answer this before reading the answer below. If you cannot answer this you may need to revisit some of the earlier material in the **Multidimensional Kalman Filter** chapter.

There are several factors contributing to our success. First, let's consider the case of having only one sensor. Any single measurement has an extreme range of possible positions. But, our target is moving, and the UKF is taking that into account. Let's plot the results of several measurements taken in a row for a moving target.

```
[ ]: ukf_internal.plot_scatter_moving_target()
```

Each individual measurement has a very large position error. However, a plot of successive measurements shows a clear trend - the target is obviously moving towards the upper right. When a Kalman filter computes the Kalman gain it takes the distribution of errors into account by using the measurement function. In this example the error lies on an approximately 45° line, so the filter will discount errors in that direction. On the other hand, there is almost no error in measurement orthogonal to that, and again the Kalman gain will take that into account.

This graph makes it look easy because we have plotted 100 measurements for each position update. The movement of the aircraft is obvious. In contrast, the Kalman filter only gets one measurement per update. Therefore the filter will not be able to generate as good a fit as the dotted green line implies.

Now consider that the bearing gives us no distance information. Suppose we set the initial estimate to 1,000 kilometers away from the sensor (vs the actual distance of 7.07 km) and make $\mathbf{P}$ very small. At that distance a 1° error translates into a positional error of 17.5 km. The KF would never be able to converge onto the actual target position because the filter is incorrectly very certain about its position estimates and because there is no distance information provided in the measurements.

Now let's consider the effect of adding a second sensor. Here are two plots showing the effects of different sensor placements. I used a square and a triangle as a symbol for the two sensors, and drew a distribution of the errors due to each sensor using the same symbol shape and color. I then computed the (x, y) coordinate corresponding to the two noisy bearing measurements and plotted them with red dots to show the distribution of the noisy measurements in x and y.

```
[ ]: with figsize(10,5):
         ukf_internal.plot_iscts_two_sensors()
```

In the first plot I placed the sensors nearly orthogonal to the target's initial position so we get these lovely 'x' shape intersections. We can see how the errors in x and y change as the target moves by the shape the scattered red dots make - as the target gets further away from the sensors, but nearer the y coordinate of sensor B the shape becomes strongly elliptical.

In the second plot the airplane starts very near one sensor, and then flies past the second sensor. The intersections of the errors are very non-orthogonal, and the resulting position errors become very spread out.

1.11 Implementation of the UKF

FilterPy implements the UKF, but it is instructive to learn how to translate equations into code. Implementing the UKF is quite straightforward. First, let's write code to compute the mean and covariance of the sigma points.

We will store the sigma points and weights in matrices, like so:

$$\begin{aligned}
\text{weights} &= \begin{bmatrix} w_0 & w_1 & \dots & w_{2n} \end{bmatrix} \\
\text{sigmas} &= \begin{bmatrix} \mathcal{X}_{0,0} & \mathcal{X}_{0,1} & \dots & \mathcal{X}_{0,n-1} \\ \mathcal{X}_{1,0} & \mathcal{X}_{1,1} & \dots & \mathcal{X}_{1,n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{X}_{2n,0} & \mathcal{X}_{2n,1} & \dots & \mathcal{X}_{2n,n-1} \end{bmatrix}
\end{aligned}$$

That's a lot of subscripts to describe something very simple, so here's an example for a two dimensional problem (n=2):

```
[ ]: points = MerweScaledSigmaPoints(n=2, alpha=.1, beta=2., kappa=1.)
     points.sigma_points(x=[0.,0], P=[[1.,.1],[.1, 1]])
```

The sigma point for the mean is on the first row. Its position is (0, 0), which is equal to the mean (0,0). The second sigma point is at position (0.173, 0.017), and so on. There are are $2n + 1 = 5$ rows, one row per sigma point. If $n = 3$, then there would be 3 columns and 7 rows.

The choice to store the sigmas in row-column vs column row format is somewhat arbitrary; my choice makes the rest of the code clearer as I can refer to the ith sigma point as `sigmas[i]` instead of `sigmas[:, i]`.

1.11.1 Weights

Computing the weights with NumPy is easy. Recall that the Van der Merwe scaled sigma point implementation states:

$$\begin{aligned}
\lambda &= \alpha^2(n+\kappa)-n \\
W^m_0 &= \frac{\lambda}{n+\lambda} \\
W^c_0 &= \frac{\lambda}{n+\lambda} + 1 - \alpha^2 + \beta \\
W^m_i = W^c_i &= \frac{1}{2(n+\lambda)}\quad i = 1..2n
\end{aligned}$$

Code for these is:

```
lambda_ = alpha**2 * (n + kappa) - n
Wc = np.full(2*n + 1,  1. / (2*(n + lambda_))
Wm = np.full(2*n + 1,  1. / (2*(n + lambda_))
Wc[0] = lambda_ / (n + lambda_) + (1. - alpha**2 + beta)
Wm[0] = lambda_ / (n + lambda_)
```

I use the underscore in `lambda_` because `lambda` is a reserved word in Python. A trailing underscore is the Pythonic workaround.

1.11.2 Sigma Points

The equations for the sigma points are:

$$\begin{cases}
\mathcal{X}_0 = \mu \\
\mathcal{X}_i = \mu + \left[\sqrt{(n+\lambda)\Sigma}\right]_i, & i = 1..n \\
\mathcal{X}_i = \mu - \left[\sqrt{(n+\lambda)\Sigma}\right]_{i-n} & i = (n+1)..2n
\end{cases}$$

The Python is not difficult once we understand the $\left[\sqrt{(n+\lambda)\Sigma}\right]_i$ term.

The term $\sqrt{(n+\lambda)\Sigma}$ is a matrix because Σ is a matrix. The subscript i in $[\sqrt{(n+\lambda)\Sigma}]_i$ is choosing the i-th row vector of the matrix. What is the square root of a matrix? There is no unique definition. One definition is that the square root of a matrix Σ is the matrix S that, when multiplied by itself, yields Σ: if $\Sigma = SS$ then $S = \sqrt{\Sigma}$.

We will choose an alternative definition that has numerical properties which make it easier to compute. We can define the square root as the matrix S, which when multiplied by its transpose, returns Σ:

$$\Sigma = \mathbf{SS}^\mathsf{T}$$

This definition is favored because $\mathbf{S}$ is computed using the *Cholesky decomposition* [5]. It decomposes a Hermitian, positive-definite matrix into a triangular matrix and its conjugate transpose. The matrix can be either upper or lower triangular, like so:

$$A = LL^* A = U^* U$$

The asterick denotes the conjugate transpose; we have only real numbers so for us we can write:

$$A = LL^\intercal A = U^\intercal U$$

P has these properties, so we can treat $\mathbf{S} = \text{cholesky}(\mathbf{P})$ as the square root of **P**.

SciPy provides `cholesky()` method in `scipy.linalg`. If your language of choice is Fortran, C, or C++, libraries such as LAPACK provide this routine. Matlab provides `chol()`.

By default `scipy.linalg.cholesky()` returns a upper triangular matrix, so I elected to write the code to expect an upper triangular matrix. For this reason I access the result by row, so that the first sigma point, which is the center point, is affected by a full row of non-zero values. If you provide your own square root implementation you will need to take this into account. You will find UKF algorithms in the literature that take the values column first. This is fine if the cholesky is lower-triangular, or if you are using a different algorithm that computes a symmetric matrix so row vs column order doesn't matter.

```
[ ]: import scipy
     a = np.array([[2., .1], [.1, 3]])
     s = scipy.linalg.cholesky(a)
     print("cholesky:")
     print(s)
     print("\nsquare of cholesky:")
     print(s @ s.T)
```

So we can implement the sigma points with this code:

```
sigmas = np.zeros((2*n+1, n))
U = scipy.linalg.cholesky((n+lambda_)*P) # sqrt

sigmas[0] = X
for k in range (n):
    sigmas[k+1]   = X + U[k]
    sigmas[n+k+1] = X - U[k]
```

Now let's implement the unscented transform. Recall the equations

$$\begin{aligned} \mu &= \sum_i w_i^m \mathcal{X}_i \\ \Sigma &= \sum_i w_i^c (\mathcal{X}_i - \mu)(\mathcal{X}_i - \mu)^\intercal \end{aligned}$$

We implement the sum of the means with

```
x = np.dot(Wm, sigmas)
```

If you are not a heavy user of NumPy this may look foreign to you. NumPy is not just a library that make linear algebra possible; under the hood it is written in C and Fortran to achieve much faster speeds than Python can reach. A typical speedup is 20x to 100x. To get that speedup we must avoid using for loops, and instead use NumPy's built in functions to perform calculations. So, instead of writing a for loop to compute the sum of products, we call the built in `numpy.dot(x,`

y) method. The dot product of two vectors is the sum of the element-wise multiplications of each element. If passed a 1D array and a 2D array it will compute the sum of inner products:

[ ]:
```
a = np.array([10, 100])
b = np.array([[1, 2, 3],
              [4, 5, 6]])
np.dot(a, b)
```

All that is left is to compute $\mathbf{P} = \sum_i w_i(\mathcal{X}_i - \mu)(\mathcal{X}_i - \mu)^\mathsf{T} + \mathbf{Q}$:

```
kmax, n = sigmas.shape
P = zeros((n, n))
for k in range(kmax):
    y = sigmas[k] - x
    P += Wc[k] * np.outer(y, y)
P += Q
```

This introduces another feature of NumPy. The state variable `x` is one dimensional, as is `sigmas[k]`, so the difference `sigmas[k]-X` is also one dimensional. NumPy will not compute the transpose of a 1-D array; it considers the transpose of `[1,2,3]` to be `[1,2,3]`. So we call the function `np.outer(y,y)` which computes the value of $\mathbf{yy}^\mathsf{T}$ for the 1D array $\mathbf{y}$. An alternative implementation could be:

```
y = (sigmas[k] - x).reshape(kmax, 1) # convert into 2D array
P += Wc[K] * np.dot(y, y.T)
```

This code is slower and not idiomatic, so we will not use it.

1.11.3 Predict Step

For the predict step, we will generate the weights and sigma points as specified above. We pass each sigma point through the function f.

$$\mathcal{Y} = f(\chi)$$

Then we compute the predicted mean and covariance using the unscented transform. In the code below you can see that I am assuming that this is a method in a class that stores the various matrices and vectors needed by the filter.

```
def predict(self, sigma_points_fn):
    """ Performs the predict step of the UKF. On return,
    self.xp and self.Pp contain the predicted state (xp)
    and covariance (Pp). 'p' stands for prediction.
    """

    # calculate sigma points for given mean and covariance
    sigmas = sigma_points_fn(self.x, self.Pp)

    for i in range(self._num_sigmas):
        self.sigmas_f[i] = self.fx(sigmas[i], self._dt)
    self.xp, self.Pp = unscented_transform(
                           self.sigmas_f, self.Wm, self.Wc, self.Q)
```

1.11.4 Update Step

The update step converts the sigmas into measurement space via the function `h(x)`.

$$\mathcal{Z} = h(\mathcal{Y})$$

The mean and covariance of those points is computed with the unscented transform. The residual and Kalman gain is then computed. The cross variance is computed as:

$$\mathbf{P}_{xz} = \sum_{i=0}^{2n} w_i^c (\mathcal{Y}_i - \mu)(\mathcal{Z}_i - \mu_z)^\mathsf{T}$$

Finally, we compute the new state estimate using the residual and Kalman gain:

$$K = \mathbf{P}_{xz}\mathbf{P}_z^{-1}$$
$$\mathbf{x} = \bar{\mathbf{x}} + \mathbf{K}\mathbf{y}$$

and the new covariance is computed as:

$$\mathbf{P} = \bar{\mathbf{P}} - \mathbf{K}\mathbf{P}_z\mathbf{K}^\mathsf{T}$$

This function can be implemented as follows, assuming it is a method of a class that stores the necessary matrices and data.

```
def update(self, z):
    # rename for readability
    sigmas_f = self.sigmas_f
    sigmas_h = self.sigmas_h

    # transform sigma points into measurement space
    for i in range(self._num_sigmas):
        sigmas_h[i] = self.hx(sigmas_f[i])

    # mean and covariance of prediction passed through UT
    zp, Pz = unscented_transform(sigmas_h, self.Wm, self.Wc, self.R)

    # compute cross variance of the state and the measurements
    Pxz = np.zeros((self._dim_x, self._dim_z))
    for i in range(self._num_sigmas):
        Pxz += self.Wc[i] * np.outer(sigmas_f[i] - self.xp,
                                     sigmas_h[i] - zp)

    K = np.dot(Pxz, inv(Pz)) # Kalman gain
    self.x = self.xp + np.dot(K, z - zp)
    self.P = self.Pp - np.dot(K, Pz).dot(K.T)
```

1.11.5 FilterPy's Implementation

FilterPy has generalized the code somewhat. You can specify different sigma point algorithms, specify how to compute the residual of the state variables (you can not subtract angles because they are modular), provide a matrix square root function, and more. See the help for details.

https://filterpy.readthedocs.org/#unscented-kalman-filter

1.12 Batch Processing

The Kalman filter is recursive - estimates are based on the current measurement and prior estimate. But it is very common to have a set of data that have been already collected which we want to filter. In this case the filter can be run in a *batch* mode, where all of the measurements are filtered at once.

Collect your measurements into an array or list.

```
zs = read_altitude_from_csv()
```

Then call the `batch_filter()` method.

```
Xs, Ps = ukf.batch_filter(zs)
```

The function takes the list/array of measurements, filters it, and returns an array of state estimates (`Xs`) and covariance matrices (`Ps`) for the entire data set.

Here is a complete example drawing from the radar tracking problem above.

[ ]:
```
dt = 12. # 12 seconds between readings
range_std = 5 # meters
bearing_std = math.radians(0.5)

ac_pos = (0., 1000.)
ac_vel = (100., 0.)
radar_pos = (0., 0.)
h_radar.radar_pos = radar_pos

points = MerweScaledSigmaPoints(n=3, alpha=.1, beta=2., kappa=0.)
kf = UKF(3, 2, dt, fx=f_radar, hx=h_radar, points=points)

kf.Q[0:2 ,0:2] = Q_discrete_white_noise(2, dt=dt, var=0.1)
kf.Q[2, 2] = 0.1

kf.R = np.diag([range_std**2, bearing_std**2])
kf.x = np.array([0., 90., 1100.])
kf.P = np.diag([300**2, 30**2, 150**2])

radar = RadarStation((0, 0), range_std, bearing_std)
```

```
ac = ACSim(ac_pos, (100, 0), 0.02)

np.random.seed(200)

t = np.arange(0, 360 + dt, dt)
n = len(t)

zs = []
for i in range(len(t)):
    ac.update(dt)
    r = radar.noisy_reading(ac.pos)
    zs.append([r[0], r[1]])

xs, covs = kf.batch_filter(zs)
ukf_internal.plot_radar(xs, t)
```

1.13 Smoothing the Results

Assume that we are tracking a car. Suppose we get a noisy measurement that implies that the car is starting to turn to the left, but the state function has predicted that the car is moving straight. The Kalman filter has no choice but to move the state estimate somewhat towards the noisy measurement, as it cannot judge whether this is just a particularly noisy measurement or the true start of a turn.

If we are collecting data and post-processing it we have measurements after the questionable one that informs us if a turn was made or not. Suppose the subsequent measurements all continue turning left. We can then be sure that the measurement was not very noisy, but instead a turn was initiated.

We will not develop the math or algorithm here, I will just show you how to call the algorithm in `FilterPy`. The algorithm that we have implemented is called an *RTS smoother*, after the three inventors of the algorithm: Rauch, Tung, and Striebel.

The routine is `UnscentedKalmanFilter.rts_smoother()`. Using it is trivial; we pass in the means and covariances computed from the `batch_filter` step, and receive back the smoothed means, covariances, and Kalman gain.

[ ]:
```
Ms, P, K = kf.rts_smoother(xs, covs)
ukf_internal.plot_rts_output(xs, Ms, t)
```

From these charts we can see that the improvement in the position is small, but the improvement in the velocity is good, and spectacular for the altitude. The difference in the position are very small, so I printed the difference between the UKF and the smoothed results for the last 5 points. I recommend always using the RTS smoother if you can post-process your data.

1.14 Choosing the Sigma Parameters

I have found the literature on choosing values for α, β, and κ to be rather lacking. Van der Merwe's dissertation contains the most information, but it is not exhaustive. So let's explore what they do.

Van der Merwe suggests using $\beta = 2$ for Gaussian problems, and $\kappa = 3 - n$. So let's start there and vary α. I will let $n = 1$ to minimize the size of the arrays we need to look at and to avoid having to compute the square root of matrices.

```
[ ]: from kf_book.ukf_internal import print_sigmas
     print_sigmas(mean=0, cov=3, alpha=1)
```

So what is going on here? We can see that for a mean of 0 the algorithm chooses sigma points of 0, 3, and -3, but why? Recall the equation for computing the sigma points:

$$\begin{aligned}\mathcal{X}_0 &= \mu \\ \mathcal{X}_i &= \mu \pm \sqrt{(n+\lambda)\Sigma}\end{aligned}$$

My choice of $n = 1$ reduces everything to scalars, allowing us to avoid computing the square root of matrices. So, for our values the equation is

$$\begin{aligned}\mathcal{X}_0 &= 0 \\ \mathcal{X}_i &= 0 \pm \sqrt{(1+2)\times 3} \\ &= \pm 3\end{aligned}$$

So as α gets larger the sigma points get more spread out. Let's set it to an absurd value.

```
[ ]: print_sigmas(mean=0, cov=3, alpha=200)
```

We can see that the sigma points spread over 100 standard deviations. If our data was Gaussian we'd be incorporating data many standard deviations away from the mean; for nonlinear problems this is unlikely to produce good results. But suppose our distribution was not Gaussian, but instead had very fat tails? We might need to sample from those tails to get a good estimate, and hence it would make sense to make κ larger (not 200, which was absurdly large to make the change in the sigma points stark).

With a similar line of reasoning, suppose that our distribution has nearly no tails - the probability distribution looks more like an inverted parabola. In such a case we'd probably want to pull the sigma points in closer to the mean to avoid sampling in regions where there will never be real data.

Now let's look at the change in the weights. When we have $k + n = 3$ the weights were 0.6667 for the mean, and 0.1667 for the two outlying sigma points. On the other hand, when $\alpha = 200$ the mean weight shot up to 0.99999 and the outlier weights were set to 0.000004. Recall the equations for the weights:

$$\begin{aligned}W_0 &= \frac{\lambda}{n+\lambda} \\ W_i &= \frac{1}{2(n+\lambda)}\end{aligned}$$

We can see that as λ gets larger the fraction for the weight of the mean ($\lambda/(n+\lambda)$) approaches 1, and the fraction for the weights of the rest of the sigma points approaches 0. This is invariant on the size of your covariance. So as we sample further and further away from the mean we end up giving less weight to those samples, and if we sampled very close to the mean we'd give very similar weights to all.

However, the advice that Van der Merwe gives is to constrain α in the range $0 > \alpha \geq 1$. He suggests 10^{-3} as a good value. Let's try that.

```
[ ]: print_sigmas(mean=0, cov=13, alpha=.001, kappa=0)
```

1.15 Robot Localization - A Fully Worked Example

It is time to undertake a significant problem. Most books choose simple, textbook problems with simple answers, and you are left wondering how to implement a real world problem. This example will not teach you how to tackle any problem, but illustrates the type of things you will have to consider as you design and implement a filter.

We will consider the problem of robot localization. In this scenario we have a robot that is moving through a landscape using a sensor to detect landmarks. This could be a self driving car using computer vision to identify trees, buildings, and other landmarks. It might be one of those small robots that vacuum your house, or a robot in a warehouse.

The robot has 4 wheels in the same configuration used by automobiles. It maneuvers by pivoting the front wheels. This causes the robot to pivot around the rear axle while moving forward. This is nonlinear behavior which we will have to model.

The robot has a sensor that gives it approximate range and bearing to known targets in the landscape. This is nonlinear because computing a position from a range and bearing requires square roots and trigonometry.

Both the process model and measurement models are nonlinear. The UKF accommodates both, so we provisionally conclude that the UKF is a viable choice for this problem.

1.15.1 Robot Motion Model

At a first approximation an automobile steers by pivoting the front tires while moving forward. The front of the car moves in the direction that the wheels are pointing while pivoting around the rear tires. This simple description is complicated by issues such as slippage due to friction, the differing behavior of the rubber tires at different speeds, and the need for the outside tire to travel a different radius than the inner tire. Accurately modeling steering requires a complicated set of differential equations.

For Kalman filtering, especially for lower speed robotic applications a simpler *bicycle model* has been found to perform well. This is a depiction of the model:

```
[ ]: ekf_internal.plot_bicycle()
```

Here we see the front tire is pointing in direction α relative to the wheelbase. Over a short time period the car moves forward and the rear wheel ends up further ahead and slightly turned inward, as depicted with the blue shaded tire. Over such a short time frame we can approximate this as a turn around a radius R. We can compute the turn angle β with

$$\beta = \frac{d}{w}\tan\left(\alpha\right)$$

and the turning radius R is given by

$$R = \frac{d}{\beta}$$

where the distance the rear wheel travels given a forward velocity v is $d = v\Delta t$.

With θ being the robot's orientation we compute the position C before the turn starts as

$$\begin{aligned}
C_x &= x - R\sin(\theta) \\
C_y &= y + R\cos(\theta)
\end{aligned}$$

After the move forward for time Δt the new position and orientation of the robot is

$$\begin{aligned}
\bar{x} &= C_x + R\sin(\theta + \beta) \\
\bar{y} &= C_y - R\cos(\theta + \beta) \\
\bar{\theta} &= \theta + \beta
\end{aligned}$$

Once we substitute in for C we get

$$\begin{aligned}
\bar{x} &= x - R\sin(\theta) + R\sin(\theta + \beta) \\
\bar{y} &= y + R\cos(\theta) - R\cos(\theta + \beta) \\
\bar{\theta} &= \theta + \beta
\end{aligned}$$

You do not need to understand this math in detail if you are not interested in steering models. The important thing to recognize is that our motion model is nonlinear, and we will need to deal with that with our Kalman filter.

1.15.2 Design the State Variables

For our robot we will maintain the position and orientation:

$$\mathbf{x} = \begin{bmatrix} x & y & \theta \end{bmatrix}^\mathsf{T}$$

I could include velocities into this model, but as you will see the math will already be quite challenging.

The control input $\mathbf{u}$ is the commanded velocity and steering angle

$$\mathbf{u} = \begin{bmatrix} v & \alpha \end{bmatrix}^\mathsf{T}$$

1.15.3 Design the System Model

We model our system as a nonlinear motion model plus white noise.

$$\bar{x} = x + f(x, u) + \mathcal{N}(0, Q)$$

Using the motion model for a robot that we created above, we can write:

```
[ ]: from math import tan, sin, cos, sqrt

     def move(x, dt, u, wheelbase):
         hdg = x[2]
         vel = u[0]
         steering_angle = u[1]
         dist = vel * dt

         if abs(steering_angle) > 0.001: # is robot turning?
             beta = (dist / wheelbase) * tan(steering_angle)
             r = wheelbase / tan(steering_angle) # radius

             sinh, sinhb = sin(hdg), sin(hdg + beta)
             cosh, coshb = cos(hdg), cos(hdg + beta)
             return x + np.array([-r*sinh + r*sinhb,
                                   r*cosh - r*coshb, beta])
         else: # moving in straight line
             return x + np.array([dist*cos(hdg), dist*sin(hdg), 0])
```

We will use this function to implement the state transition function `f(x)`.

I will design the UKF so that Δt is small. If the robot is moving slowly then this function should give a reasonably accurate prediction. If Δt is large or your system's dynamics are very nonlinear this method will fail. In those cases you will need to implement it using a more sophisticated numerical integration technique such as Runge Kutta. Numerical integration is covered briefly in the **Kalman Filter Math** chapter.

1.15.4 Design the Measurement Model

The sensor provides a noisy bearing and range to multiple known locations in the landscape. The measurement model must convert the state $\begin{bmatrix}x & y & \theta\end{bmatrix}^\mathsf{T}$ into a range and bearing to the landmark. If p is the position of a landmark, the range r is

$$r = \sqrt{(p_x - x)^2 + (p_y - y)^2}$$

We assume that the sensor provides bearing relative to the orientation of the robot, so we must subtract the robot's orientation from the bearing to get the sensor reading, like so:

$$\phi = \tan^{-1}(\frac{p_y - y}{p_x - x}) - \theta$$

Thus our measurement function is

$$\begin{aligned}
\mathbf{z} = h(\mathbf{x}, \mathbf{P}) &\qquad + \mathcal{N}(0, R) \\
= \begin{bmatrix} \sqrt{(p_x - x)^2 + (p_y - y)^2} \\ \tan^{-1}(\frac{p_y - y}{p_x - x}) - \theta \end{bmatrix} &\qquad + \mathcal{N}(0, R)
\end{aligned}$$

I will not implement this yet as there is a difficulty that will be discussed in the *Implementation* section below.

1.15.5 Design Measurement Noise

It is reasonable to assume that the range and bearing measurement noise is independent, hence

$$\mathbf{R} = \begin{bmatrix} \sigma_{range}^2 & 0 \\ 0 & \sigma_{bearing}^2 \end{bmatrix}$$

1.15.6 Implementation

Before we begin coding we have another issue to handle. The residual is $y = z - h(x)$. Suppose z has a bearing of 1° and $h(x)$ is 359°. Subtracting them gives -358°. This will throw off the computation of the Kalman gain because the correct angular difference is 2°. So we will have to write code to correctly compute the bearing residual.

```
[ ]: def normalize_angle(x):
         x = x % (2 * np.pi)    # force in range [0, 2 pi)
         if x > np.pi:          # move to [-pi, pi)
             x -= 2 * np.pi
         return x
```

```
[ ]: print(np.degrees(normalize_angle(np.radians(1-359))))
```

The state vector has the bearing at index 2, but the measurement vector has it at index 1, so we need to write functions to handle each. Another issue we face is that as the robot maneuvers different landmarks will be visible, so we need to handle a variable number of measurements. The function for the residual in the measurement will be passed an array of several measurements, one per landmark.

```
[ ]: def residual_h(a, b):
         y = a - b
         # data in format [dist_1, bearing_1, dist_2, bearing_2,...]
         for i in range(0, len(y), 2):
             y[i + 1] = normalize_angle(y[i + 1])
         return y

     def residual_x(a, b):
         y = a - b
         y[2] = normalize_angle(y[2])
```

```
    return y
```

We can now implement the measurement model. The equation is

$$h(\mathbf{x}, \mathbf{P}) = \begin{bmatrix} \sqrt{(p_x - x)^2 + (p_y - y)^2} \\ \tan^{-1}(\frac{p_y - y}{p_x - x}) - \theta \end{bmatrix}$$

The expression $\tan^{-1}(\frac{p_y - y}{p_x - x}) - \theta$ can produce a result outside the range $[-\pi, \pi)$, so we should normalize the angle to that range.

The function will be passed an array of landmarks and needs to produce an array of measurements in the form `[dist_to_1, bearing_to_1, dist_to_2, bearing_to_2, ...]`.

```
[ ]: def Hx(x, landmarks):
         """ takes a state variable and returns the measurement
         that would correspond to that state. """
         hx = []
         for lmark in landmarks:
             px, py = lmark
             dist = sqrt((px - x[0])**2 + (py - x[1])**2)
             angle = atan2(py - x[1], px - x[0])
             hx.extend([dist, normalize_angle(angle - x[2])])
         return np.array(hx)
```

Our difficulties are not over. The unscented transform computes the average of the state and measurement vectors, but each contains a bearing. There is no unique way to compute the average of a set of angles. For example, what is the average of 359° and 3°? Intuition suggests the answer should be 1°, but a naive $\frac{1}{n}\sum x$ approach yields 181°.

One common approach is to take the arctan of the sum of the sins and cosines.

$$\bar{\theta} = atan2\left(\frac{\sum_{i=1}^n \sin\theta_i}{n}, \frac{\sum_{i=1}^n \cos\theta_i}{n}\right)$$

`UnscentedKalmanFilter.__init__()` has an argument `x_mean_fn` for a function which computes the mean of the state, and `z_mean_fn` for a function which computes the mean of the measurement. We will code these function as:

```
[ ]: def state_mean(sigmas, Wm):
         x = np.zeros(3)

         sum_sin = np.sum(np.dot(np.sin(sigmas[:, 2]), Wm))
         sum_cos = np.sum(np.dot(np.cos(sigmas[:, 2]), Wm))
         x[0] = np.sum(np.dot(sigmas[:, 0], Wm))
         x[1] = np.sum(np.dot(sigmas[:, 1], Wm))
         x[2] = atan2(sum_sin, sum_cos)
         return x
```

```python
def z_mean(sigmas, Wm):
    z_count = sigmas.shape[1]
    x = np.zeros(z_count)

    for z in range(0, z_count, 2):
        sum_sin = np.sum(np.dot(np.sin(sigmas[:, z+1]), Wm))
        sum_cos = np.sum(np.dot(np.cos(sigmas[:, z+1]), Wm))

        x[z] = np.sum(np.dot(sigmas[:,z], Wm))
        x[z+1] = atan2(sum_sin, sum_cos)
    return x
```

These functions take advantage of the fact that NumPy's trigometric functions operate on arrays, and `dot` performs element-wise multiplication. NumPy is implemented in C and Fortran, so `sum(dot(sin(x), w))` is much faster than writing the equivalent loop in Python.

With that done we are now ready to implement the UKF. I want to point out that when I designed this filter I did not just design all of functions above in one sitting, from scratch. I put together a basic UKF with predefined landmarks, verified it worked, then started filling in the pieces. "What if I see different landmarks?" That lead me to change the measurement function to accept an array of landmarks. "How do I deal with computing the residual of angles?" This led me to write the angle normalization code. "What is the *mean* of a set of angles?" I searched on the internet, found an article on Wikipedia, and implemented that algorithm. Do not be daunted. Design what you can, then ask questions and solve them, one by one.

You've seen the UKF implemention already, so I will not describe it in detail. There are two new things here. When we construct the sigma points and filter we have to provide it the functions that we have written to compute the residuals and means.

```python
points = SigmaPoints(n=3, alpha=.00001, beta=2, kappa=0,
                     subtract=residual_x)

ukf = UKF(dim_x=3, dim_z=2, fx=fx, hx=Hx, dt=dt, points=points,
         x_mean_fn=state_mean, z_mean_fn=z_mean,
         residual_x=residual_x, residual_z=residual_h)
```

Next, we need to pass extra data into our `f(x, dt)` and `h(x)` functions. We want to use `move(x, dt, u, wheelbase)` for `f(x, dt)`, and `Hx(x, landmarks)` for `h(x)`. We can do this, we just have to pass the extra parameters into `predict()` and `update()` as keyword arguments like so:

```python
ukf.predict(u=u, wheelbase=wheelbase)
ukf.update(z, landmarks=landmarks)
```

The rest of the code runs the simulation and plots the results. I create a variable `landmarks` that contains the coordinates of the landmarks. I update the simulated robot position 10 times a second, but run the UKF only once per second. We are not using Runge Kutta to integrate the differential equations of motion, so a small time step makes the simulation more accurate.

[ ]:
```python
from filterpy.stats import plot_covariance_ellipse
```

```python
dt = 1.0
wheelbase = 0.5

def run_localization(
    cmds, landmarks, sigma_vel, sigma_steer, sigma_range,
    sigma_bearing, ellipse_step=1, step=10):

    plt.figure()
    points = MerweScaledSigmaPoints(n=3, alpha=.00001, beta=2, kappa=0,
                                    subtract=residual_x)
    ukf = UKF(dim_x=3, dim_z=2*len(landmarks), fx=move, hx=Hx,
              dt=dt, points=points, x_mean_fn=state_mean,
              z_mean_fn=z_mean, residual_x=residual_x,
              residual_z=residual_h)

    ukf.x = np.array([2, 6, .3])
    ukf.P = np.diag([.1, .1, .05])
    ukf.R = np.diag([sigma_range**2,
                     sigma_bearing**2]*len(landmarks))
    ukf.Q = np.eye(3)*0.0001

    sim_pos = ukf.x.copy()

    # plot landmarks
    if len(landmarks) > 0:
        plt.scatter(landmarks[:, 0], landmarks[:, 1],
                    marker='s', s=60)

    track = []
    for i, u in enumerate(cmds):
        sim_pos = move(sim_pos, dt/step, u, wheelbase)
        track.append(sim_pos)

        if i % step == 0:
            ukf.predict(u=u, wheelbase=wheelbase)

            if i % ellipse_step == 0:
                plot_covariance_ellipse(
                    (ukf.x[0], ukf.x[1]), ukf.P[0:2, 0:2], std=6,
                     facecolor='k', alpha=0.3)

            x, y = sim_pos[0], sim_pos[1]
            z = []
            for lmark in landmarks:
                dx, dy = lmark[0] - x, lmark[1] - y
                d = sqrt(dx**2 + dy**2) + randn()*sigma_range
                bearing = atan2(lmark[1] - y, lmark[0] - x)
                a = (normalize_angle(bearing - sim_pos[2] +
                     randn()*sigma_bearing))
                z.extend([d, a])
            ukf.update(z, landmarks=landmarks)

            if i % ellipse_step == 0:
                plot_covariance_ellipse(
                    (ukf.x[0], ukf.x[1]), ukf.P[0:2, 0:2], std=6,
                     facecolor='g', alpha=0.8)
    track = np.array(track)
    plt.plot(track[:, 0], track[:,1], color='k', lw=2)
    plt.axis('equal')
    plt.title("UKF Robot localization")
    plt.show()
    return ukf
```

[ ]:
```
landmarks = np.array([[5, 10], [10, 5], [15, 15]])
cmds = [np.array([1.1, .01])] * 200
ukf = run_localization(
    cmds, landmarks, sigma_vel=0.1, sigma_steer=np.radians(1),
    sigma_range=0.3, sigma_bearing=0.1)
print('Final P:', ukf.P.diagonal())
```

The rest of the code runs the simulation and plots the results. I create a variable `landmarks` that contains the coordinates of the landmarks. I update the simulated robot position 10 times a second, but run the UKF only once. This is for two reasons. First, we are not using Runge Kutta to integrate the differental equations of motion, so a narrow time step allows our simulation to be more accurate. Second, it is fairly normal in embedded systems to have limited processing speed. This forces you to run your Kalman filter only as frequently as absolutely needed.

1.15.7 Steering the Robot

The steering simulation in the run above is not realistic. The velocity and steering angles never changed, which doesn't pose much of a problem for the Kalman filter. We could implement a complicated PID controlled robot simulation, but I will just generate varying steering commands using NumPy's `linspace` method. I'll also add more landmarks as the robot will be traveling much farther than in the first example.

[ ]:
```
landmarks = np.array([[5, 10], [10, 5], [15, 15], [20, 5],
                      [0, 30], [50, 30], [40, 10]])
dt = 0.1
wheelbase = 0.5
sigma_range=0.3
sigma_bearing=0.1

def turn(v, t0, t1, steps):
  return [[v, a] for a in np.linspace(
               np.radians(t0), np.radians(t1), steps)]
```

```
# accelerate from a stop
cmds = [[v, .0] for v in np.linspace(0.001, 1.1, 30)]
cmds.extend([cmds[-1]]*50)

# turn left
v = cmds[-1][0]
cmds.extend(turn(v, 0, 2, 15))
cmds.extend([cmds[-1]]*100)

#turn right
cmds.extend(turn(v, 2, -2, 15))
cmds.extend([cmds[-1]]*200)

cmds.extend(turn(v, -2, 0, 15))
cmds.extend([cmds[-1]]*150)

cmds.extend(turn(v, 0, 1, 25))
cmds.extend([cmds[-1]]*100)
```

```
[ ]: ukf = run_localization(
         cmds, landmarks, sigma_vel=0.1, sigma_steer=np.radians(1),
         sigma_range=0.3, sigma_bearing=0.1, step=1,
         ellipse_step=20)
     print('final covariance', ukf.P.diagonal())
```

The uncertainty becomes very small very quickly. The covariance ellipses are displaying the 6σ covariance, yet the ellipses are so small they are hard to see. We can incorporate more error into the answer by only supplying two landmarks near the start point. When we run this filter the errors increase as the robot gets further away from these landmarks.

```
[ ]: ukf = run_localization(
         cmds, landmarks[0:2], sigma_vel=0.1, sigma_steer=np.radians(1),
         sigma_range=0.3, sigma_bearing=0.1, step=1,
         ellipse_step=20)
     print('final covariance', ukf.P.diagonal())
```

1.16 Discussion

Your impression of this chapter probably depends on how many nonlinear Kalman filters you have implemented in the past. If this is your first exposure perhaps the computation of $2n + 1$ sigma points and the subsequent writing of the $f(x)$ and $h(x)$ function struck you as a bit finicky. Indeed, I spent more time than I'd care to admit getting everything working because of the need to handle the modular math of angles. On the other hand, if you have implemented an extended Kalman filter (EKF) perhaps you are bouncing gleefully in your seat. There is a small amount of tedium in writing the functions for the UKF, but the concepts are very basic. The EKF for the same problem requires some fairly difficult mathematics. For many problems we cannot find a closed form solution for the equations of the EKF, and we must retreat to some sort of iterated solution.

The advantage of the UKF over the EKF is not only the relative ease of implementation. It is somewhat premature to discuss this because you haven't learned the EKF yet, but the EKF linearizes the problem at one point and passes that point through a linear Kalman filter. In contrast, the UKF takes $2n+1$ samples. Therefore the UKF is often more accurate than the EKF, especially when the problem is highly nonlinear. While it is not true that the UKF is guaranteed to always outperform the EKF, in practice it has been shown to perform at least as well, and usually much better than the EKF.

Hence my recommendation is to always start by implementing the UKF. If your filter has real world consequences if it diverges (people die, lots of money lost, power plant blows up) of course you will have to engage in sophisticated analysis and experimentation to choose the best filter. That is beyond the scope of this book, and you should be going to graduate school to learn this theory.

Finally, I have spoken of the UKF as *the* way to perform sigma point filters. This is not true. The specific version I chose is Julier's scaled unscented filter as parameterized by Van der Merwe in his 2004 dissertation. If you search for Julier, Van der Merwe, Uhlmann, and Wan you will find a family of similar sigma point filters that they developed. Each technique uses a different way of choosing and weighting the sigma points. But the choices don't stop there. For example, the SVD Kalman filter uses singular value decomposition (SVD) to find the approximate mean and covariance of the probability distribution. Think of this chapter as an introduction to the sigma point filters, rather than a definitive treatment of how they work.

1.17 References

- [1] Rudolph Van der Merwe. "Sigma-Point Kalman Filters for Probabilistic Inference in Dynamic State-Space Models" dissertation (2004).
- [2] Simon J. Julier. "The Scaled Unscented Transformation". Proceedings of the American Control Conference 6. IEEE. (2002)
- [3] http://www.esdradar.com/brochures/Compact%20Tracking%2037250X.pdf
- [4] Julier, Simon J.; Uhlmann, Jeffrey "A New Extension of the Kalman Filter to Nonlinear Systems". Proc. SPIE 3068, Signal Processing, Sensor Fusion, and Target Recognition VI, 182 (July 28, 1997)
- [5] Cholesky decomposition. Wikipedia. http://en.wikipedia.org/wiki/Cholesky_decomposition