无迹卡尔曼滤波

The Unscentd Kalman Filter

```
#format the book
import book_format
book_format.set_style()
```

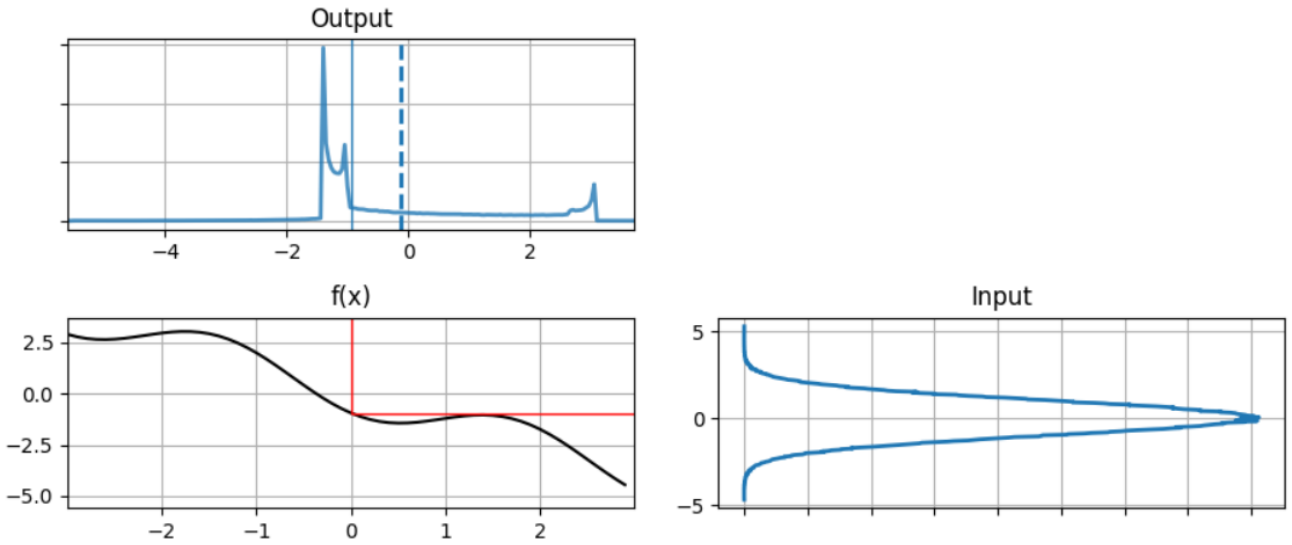
上一章我们讨论了非线性系统带来的困难。这种非线性可能出现在两个地方。它可以存在于我们的测量中，例如测量物体的倾斜距离的雷达。使用Slant range计算x、y坐标需要取平方根:

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

非线性也可以发生在过程模型中——我们可能正在跟踪一个在空气中旅行的球，在那里空气阻力的影响导致非线性行为。对于这类问题，标准卡尔曼滤波器表现不佳或者根本不好

在上一章中，我展示了这样一个图。我稍微改变了一下方程，以强调非线性的影响。

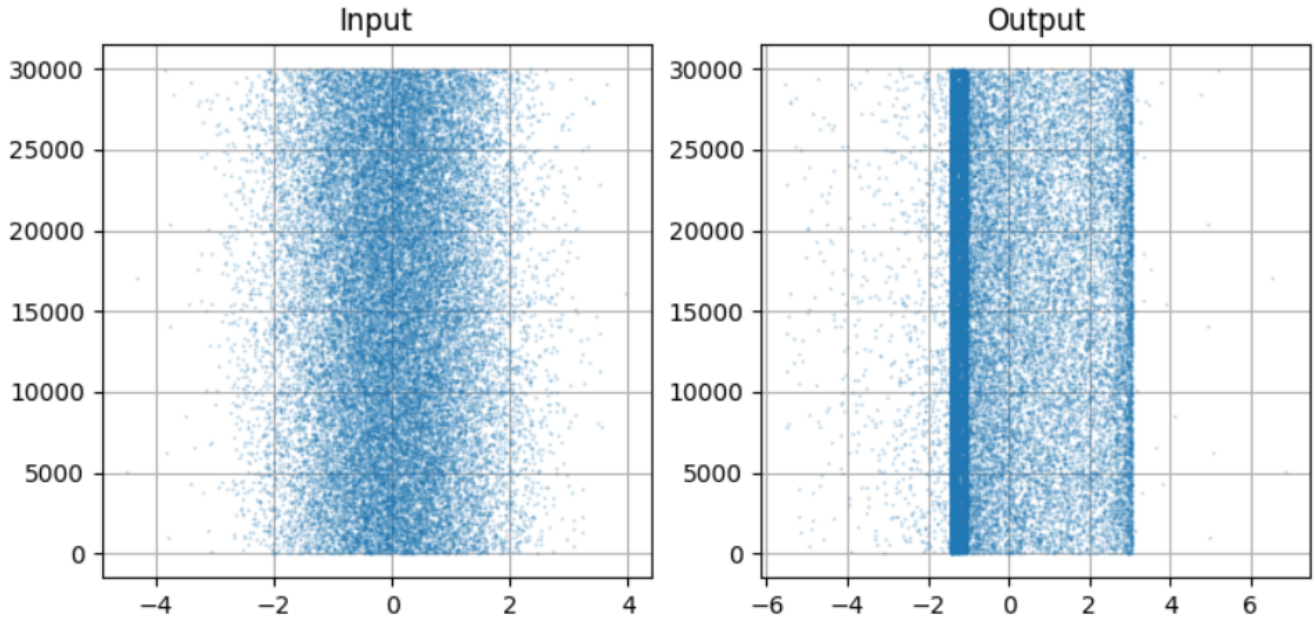
```
from kf_book.book_plots import set_figsize, figsize
import matplotlib.pyplot as plt
from kf_book.nonlinear_plots import plot_nonlinear_func
from numpy.random import normal
import numpy as np

# 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)
```



我通过从输入中获取500,000个样本，通过非线性变换，并构建结果的直方图来生成它。我们称这些点为*点。从输出直方图中，我们可以计算平均值和标准差，这将给我们一个更新的，尽管是近似的高斯分布。

让我给你们看一下数据经过f(x)前后的散点图。

```
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);
```



数据本身看起来是高斯分布，事实也确实如此。我的意思是它看起来像散布在平均零点周围的白噪声。

相反，`g(data)`有一个明确的结构。有两个波段，中间有大量的点。在带的外面有分散的点，但在负半轴的一边有更多的点。

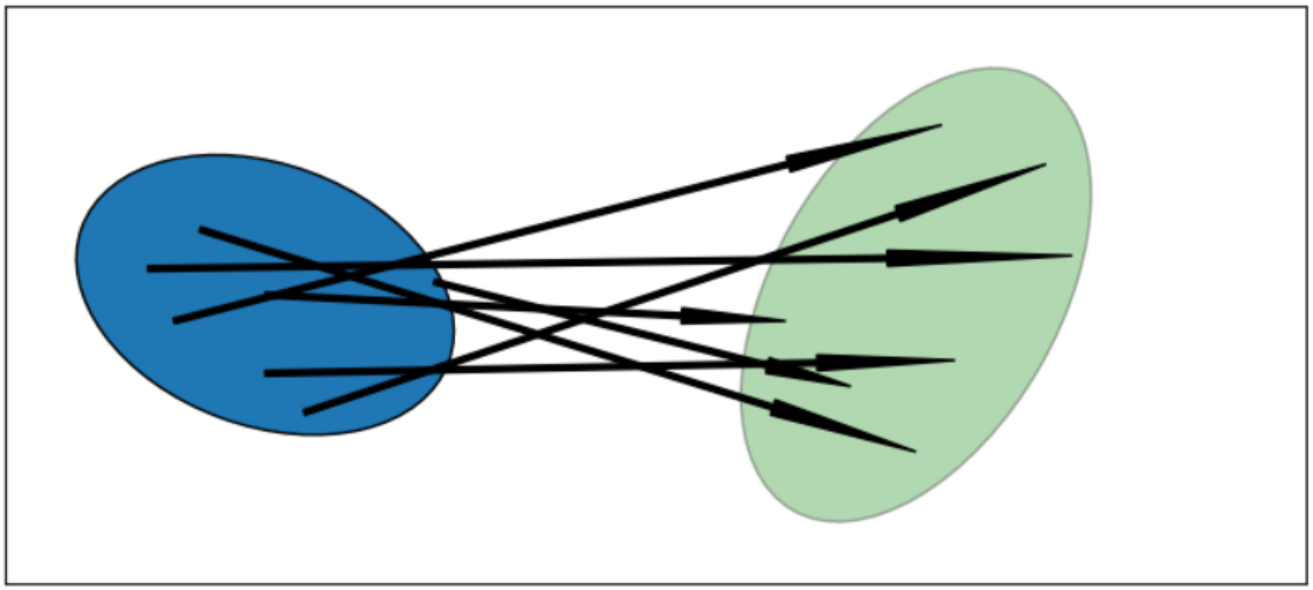
你方也许已经想到，这种抽样过程可以解决我们的问题。假设对于每次更新，我们生成500,000个点，将它们传递给函数，然后计算结果的均值和方差。这被称为蒙特卡罗方法，它被一些卡尔曼滤波器设计所使用，比如集合滤波器和粒子滤波器。抽样不需要专业知识，也不需要封闭形式的解决方案。不管这个函数有多非线性或表现有多差，只要我们用足够的sigma点进行采样，我们就能建立一个准确的输出分布。

“足够的分数”是个难题。上面的图表是用50万个sigma点创建的，输出仍然不平滑。更糟糕的是，这只适用于一维。所需点的数量随着维度数量的增加而增加。如果一个维度只需要500个点，那么两个维度就需要500个点的平方，或者250,000个点，500个点的立方，或者125,000,000个点，以此类推。因此，虽然这种方法确实有效，但它的计算成本非常高。集合滤波器和粒子滤波器使用了巧妙的技术来显著降低这种维数，但计算负担仍然非常大。unscented卡尔曼滤波器使用sigma点，但通过使用确定性方法选择点，大大减少了计算量。

Sigma Points - Sampling from a Distribution

让我们从二维协方差椭圆的角度来看这个问题。我选择2D只是因为它易于绘制;这可以扩展到任何维度。假设某个任意的非线性函数，我们将从第一个协方差椭圆中取随机点，通过非线性函数，并绘制它们的新位置。然后我们可以计算变换点的均值和协方差，并以此作为均值和概率分布的估计。

```
import kf_book.ukf_internal as ukf_internal
ukf_internal.show_2d_transform()
```



在左边，我们展示了一个椭圆，描绘了两个状态变量的1σ分布。箭头显示了几个随机采样点如何被一些任意的非线性函数转换成一个新的分布。右边的椭圆是半透明的，表示它是这组点的均值和方差的估计值。

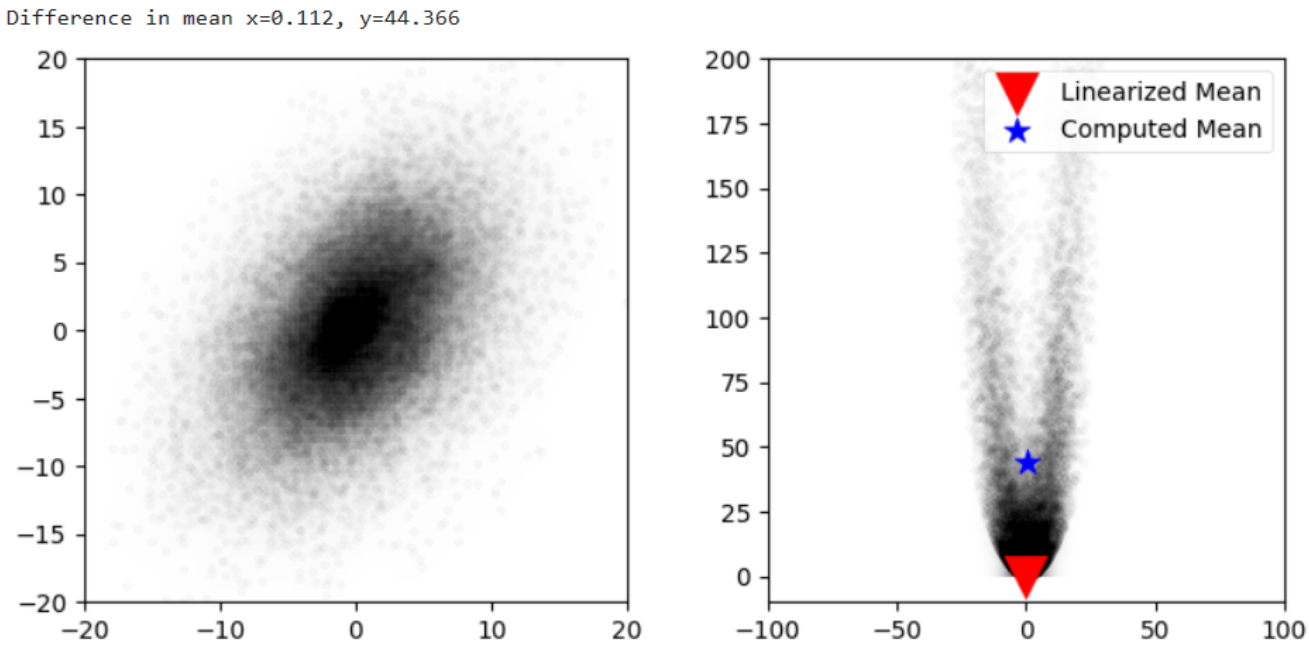
我们写一个函数，它从高斯分布中随机抽取10000个点

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

根据这个非线性系统:

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

```
import numpy as np
from numpy.random import multivariate_normal
from kf_book.nonlinear_plots import plot_monte_carlo_mean

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

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

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



该图显示了该函数发生的强非线性，以及如果我们以扩展卡尔曼滤波器的方式线性化将导致的大误差(我们将在下一章中学习)。

A Quick Example

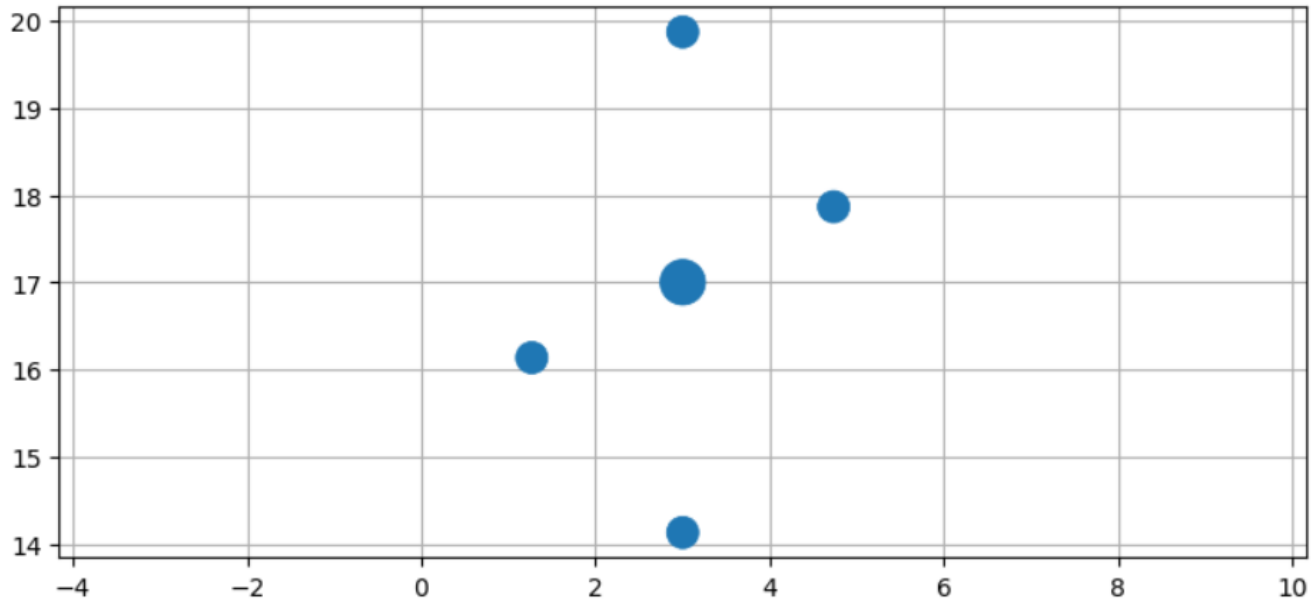
我很快就会进入Unscented卡尔曼滤波器(UKF)用于选择sigma点并执行计算的数学。但是让我们从一个例子开始。

我们将学习UKF可以使用许多不同的算法来生成sigma点。FilterPy提供了几种算法。这里有一种选择:

```
from filterpy.kalman import JulierSigmaPoints

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

稍后会变得更清楚，但这个对象将为任何给定的均值和协方差生成加权sigma点。让我们来看一个例子，其中点的大小表示它的权重:

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



你可以看到，我们有5个点围绕平均值(3,17)。它将比500,000个随机生成的点做得更好这件事可能看起来很荒谬，但它真的会!

好，现在让我们实现这个过滤器。我们将在一维中实现一个标准线性滤波器;我们还没有准备好处理非线性滤波器。过滤器的设计与我们目前所学的没有太大的不同，只有一点不同。KalmanFilter类使用矩阵F来计算状态转换函数。矩阵意味着线性代数，它适用于线性问题，但不适用于非线性问题。所以，我们用函数代替矩阵，就像我们上面做的那样。KalmanFilter类使用另一个矩阵H来实现测量函数，该函数将状态转换为等效的测量。再说一遍，矩阵意味着线性，所以我们用函数代替矩阵。也许很清楚为什么H被称为“度量函数”;对于线性卡尔曼滤波器，它是一个矩阵，但这只是一种快速计算线性函数的方法。

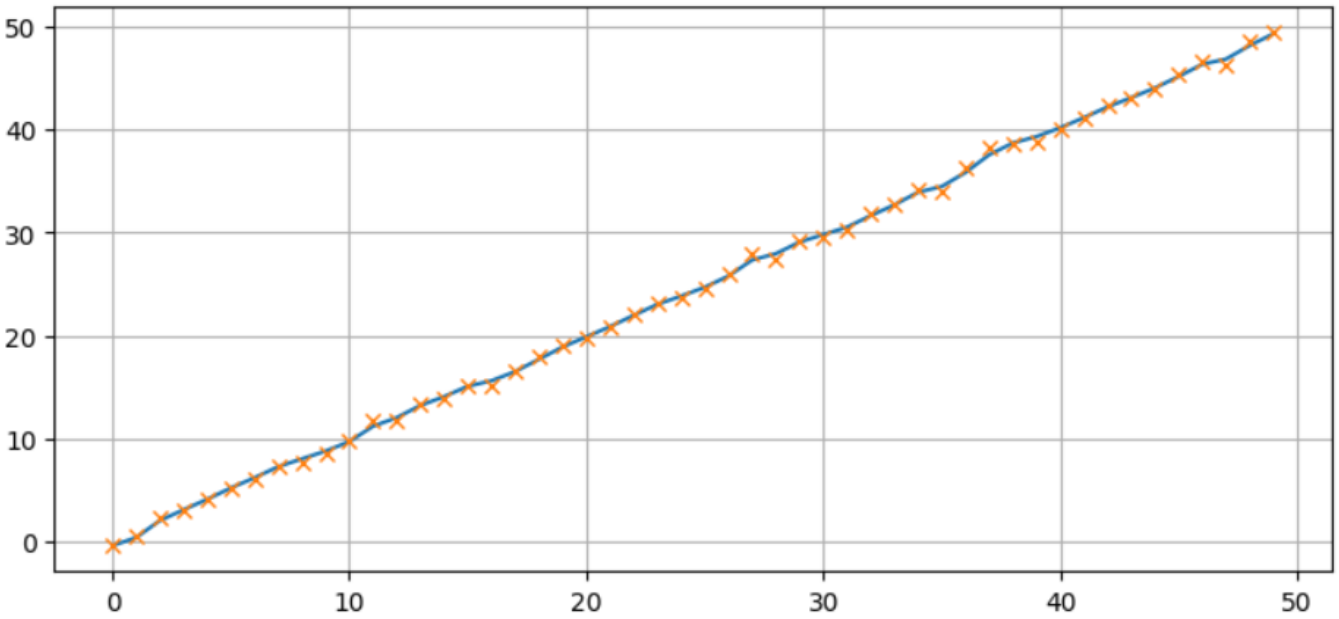
话不多说，下面是一维跟踪问题的状态转换函数和测量函数，其中状态为$\mathbf{x} = [x\ \dot{x}]^\mathsf{T}$:

```
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]
```

让我们弄清楚，这是一个线性例子。对于线性问题，不需要使用UKF，但是我从最简单的例子开始。但是注意我是如何写fx()来计算$\bar{x}$作为一组方程而不是矩阵乘法。这是为了说明我可以实现任意的非线性函数;我们不受线性方程的约束。
其余的设计都是一样的。设计P、R和Q。你知道怎么做，让我们完成滤波器并运行它。

```
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='');
```



这里并没有什么新东西。您必须创建一个为您创建sigma点的对象，并为F和H提供函数而不是矩阵，但

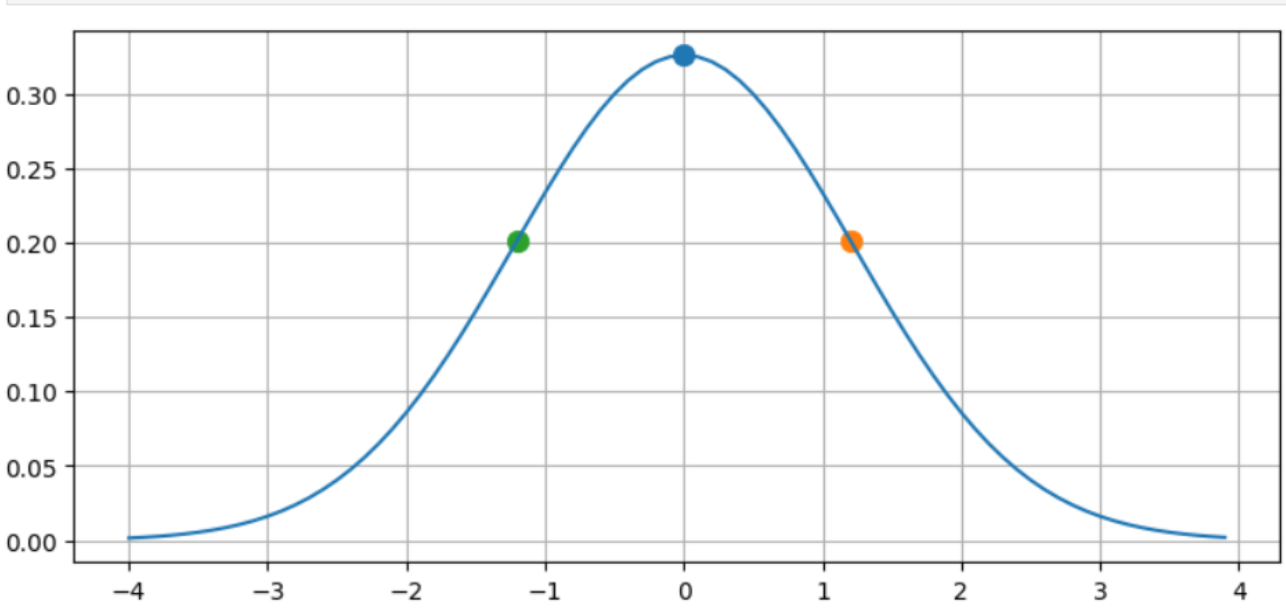
其余部分与之前相同。这应该给您足够的信心来学习一些数学和算法，这样您就可以理解UKF在做什么。

Choosing Sigma Points

在本章的开头，我使用500,000个随机生成的sigma点来计算高斯函数通过非线性函数的概率分布。虽然计算的平均值相当准确，但每次更新计算500,000个点会导致我们的过滤器非常慢。那么，我们可以使用的最少采样点的数量是多少，这个问题的表述对这些点有什么样的约束?我们将假设我们没有关于非线性函数的特殊知识，因为我们想要找到一个适用于任何函数的广义算法。
让我们考虑最简单的可能情况，看看它是否提供了任何见解。最简单的系统是恒等函数:f(x)= x。如果我们的算法对恒等函数不起作用，则滤波器不能收敛。换句话说，如果输入为1(对于一维系统)，输出也必须为1。如果输出是不同的，例如1.1，那么当我们在下一个时间步骤中将1.1输入到变换中时，我们将得到另一个数字，可能是1.23。这个过滤器是发散的。
我们可以使用的最小点数是每维1。这是线性卡尔曼滤波器使用的数字。分布的卡尔曼滤波器$N(\mu, \sigma^2)$的输入是μ本身。虽然这对线性情况有效，但对非线性情况却不是一个好答案。
也许我们可以使用每个维度一个点，但要做一些改变。但是，如果我们将一些值$\mu + \Delta$

传递给恒等函数f(x)=x, 它将不会收敛，因此这将不起作用。如果我们不改变μ，那么这将是标准的卡尔曼滤波器。我们必须断定一个样本是行不通的。
下一个最小的数是多少?两个。考虑到高斯分布是对称的，并且我们可能希望始终有一个样本点是恒等函数工作的输入的均值。有两个点需要我们选择均值，然后选择另一个点。另一个点会在我们的输入中引入不对称，这可能是我们不想要的。对于恒等函数f(x)= x这将是非常困难的。
下一个最低的数字是3个点。3个点允许我们选择平均值，然后在平均值的两侧各选择一个点，如下图所示。

```
ukf_internal.show_3_sigma_points()
```



我们可以将这些点传递给非线性函数f(x)，并计算结果的均值和方差。均值可以计算为这3个点的平均值，但这不是很一般。例如，对于一个非常非线性的问题，我们可能希望中心点的权重比外部点大得

多，或者我们可能希望外部点的权重更高。
更一般的方法是计算加权平均值$\mu = \sum_i w_i \, f(\mathcal{X}_i)$, 其中$\mathcal{X}_i$是sigma点。我们需要这些权重之和等于1。给定这个要求，我们的任务是选择$\mathcal{X}_i$及其相应的权重，以便它们计算转换后sigma点的均值和方差。

如果我们对均值进行加权，那么对协方差进行加权也是有意义的。可以对均值(ω^m)和协方差(ω^c)使用不同的权重。在下面的等式中，我使用上标来为索引留出空间。我们可以这样写

$$\begin{aligned}
\textbf{Constraints}: \\
1 &= \sum_i w_i^m \\
1 &= \sum_i w_i^c \\
\mu &= \sum_i w_i^m f(\mathcal{X}_i) \\
\Sigma &= \sum_i w_i^c (f(\mathcal{X})_i - \mu)(f(\mathcal{X})_i - \mu)^\mathsf{T}
\end{aligned}$$

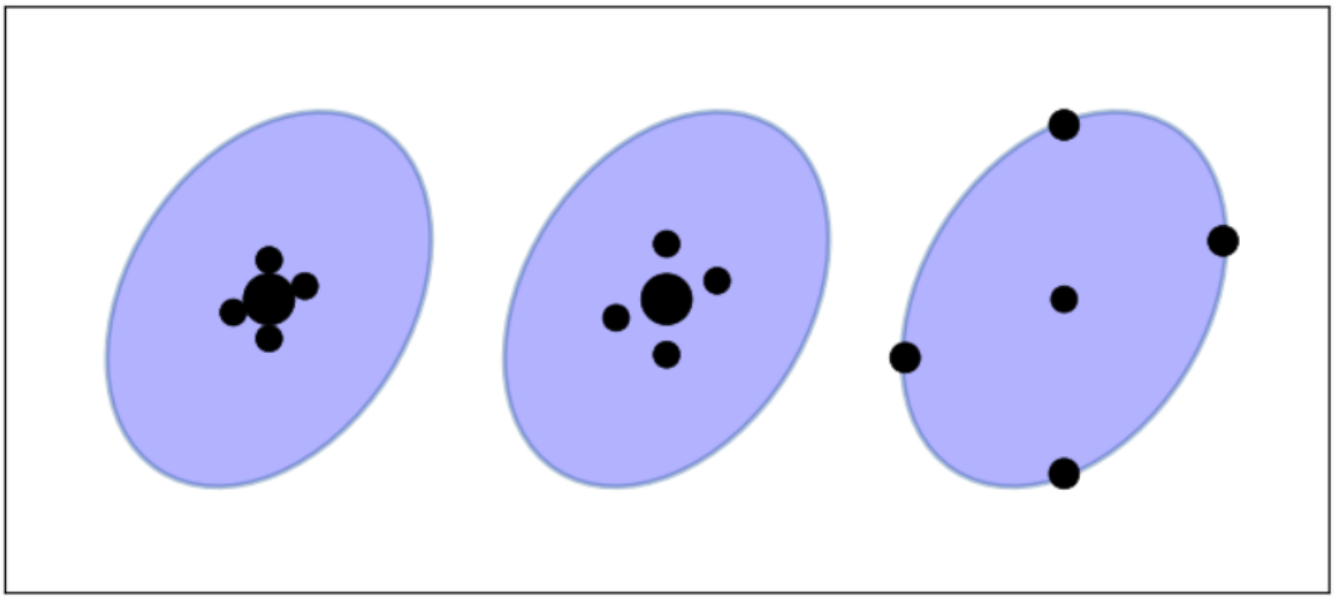
前两个方程是权重之和必须为1的约束条件。第三个等式是如何计算权重均值。第四个方程可能不太熟悉，但回想一下，两个随机变量的协方差方程为:

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

这些约束不能形成唯一的解。例如，如果你使ω_0^m小，你可以通过使ω_1^m和ω_2^m变大来补偿。你可以对均值和协方差使用不同的权重，也可以使用相同的权重。实际上，这些方程根本不要求任何点是输入的平均值，尽管可以说，这样做似乎“很好”。

我们需要一个满足约束条件的算法，最好每个维度只有3个点。在我们继续之前，我想确认一下思路。下面是具有不同sigma点的相同协方差椭圆的三个不同例子。σ点的大小与每个点的权重成正比。

```
ukf_internal.show_sigma_selections()
```



这些点不在椭圆的长轴和短轴上;约束条件没有要求我这么做。这些点是均匀分布的，但约束条件并不

要求这样做。

σ点的排列和权重会影响我们对分布的采样方式。距离很近的点会产生局部效应，因此可能对非常非线性的问题效果更好。距离较远或远离椭圆轴的点将采样非局部效果和非高斯行为。但是，通过改变每个点的权重，我们可以缓解这个问题。如果点远离均值，但权重很小，我们将纳入一些关于分布的知识，而不允许问题的非线性创建一个糟糕的估计。

请理解有无限种方法来选择sigma点。我选择的约束条件只是一种方法。例如，并不是所有用于创建sigma点的算法都要求权重的和为1。事实上，我在本书中推荐的算法并不具备这种特性。

The Unscented Transform

目前，假设存在一个选择sigma点和权重的算法。如何使用σ点来实现滤波器?

*无迹变换*是算法的核心，但它非常简单。它将sigma点χ传递给一个非线性函数，得到一个变换后的点集。

$$\boldsymbol{\mathcal{Y}} = f(\boldsymbol{\mathcal{X}})$$

然后计算变换后点的均值和协方差。均值和协方差成为新的估计。下图描述了unscented变换的操作。右边的绿色椭圆表示变换后sigma点的均值和协方差。

```
ukf_internal.show_sigma_transform(with_text=True)
```

sigma点的均值和协方差计算如下:

$$\mu = \sum_{i=0}^{2n} w_i^m \boldsymbol{\mathcal{Y}}_i$$

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

这些方程应该很熟悉——它们是我们在上面开发的约束方程。

简而言之，unscented变换从任意概率分布中采样点，将它们传递给任意的非线性函数，并为每个变换后的点生成高斯分布。我希望你能想象我们如何使用它来实现非线性卡尔曼滤波器。一旦我们有了高斯分布，我们已经开发的所有数学工具都将发挥作用!

“unscented”这个名字可能会让人困惑。这并不意味着什么。这是发明者开的一个玩笑，他说他的算法不“臭”，很快这个名字就被记住了。这个术语没有数学意义。

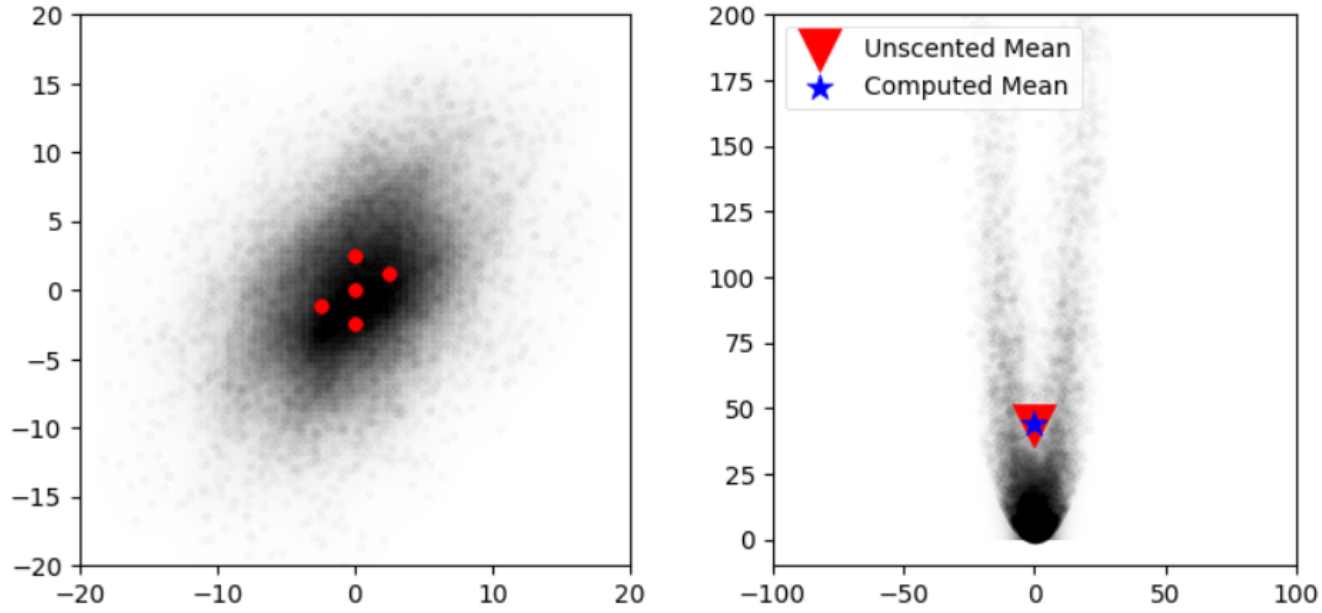
Accuracy of the Unscented Transform

前面我们写过一个函数，它通过向一个非线性函数传递5万个点来计算分布的均值。现在让我们通过同一个函数传递5 sigma点，并使用unscented变换计算它们的均值。我们将使用 `FilterPy` 函数 `merwesescaledsigmapopoints()` 来创建sigma点，并使用 `unscented_transform` 来执行转换;稍后我们将

学习这些函数。在本章的第一个例子中，我使用了`juliersigmpoints`;它们都选择sigma点，但方式不同，我稍后会解释。

```
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
```



我发现这个结果很显著。仅使用5个点，我们就能够以惊人的精度计算平均值。x的误差仅为-0.097,y的误差为0.549。相比之下，线性化方法(用于EKF，我们将在下一章中学习)得到的y的误差超过43。如果你看生成sigma点的代码，你会发现它不知道非线性函数，只知道初始分布的均值和协方差。如果我们有一个完全不同的非线性函数，同样的5 sigma点也会产生。

我承认选择了一个非线性函数，使unscented变换的性能与EKF相比显著。但是物理世界充满了非常非线性的行为，而UKF可以从容应对。我没有找到一个unscented变换运行良好的函数。在下一章中，你将看到更多传统技术如何与强非线性作斗争。这个图表是为什么我建议您尽可能使用UKF或类似的现代技术的基础。

The Unscented Kalman Filter

我们现在可以展示UKF算法了

Predict Step

UKF的预测步骤使用过程模型f()计算先验。假设f()是非线性的，因此我们生成sigma点X以及它们对应的权重Wm,Wc
根据这些函数:

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

我们将每个sigma点传递给$f(\mathbf{x}, \Delta t)$。根据过程模型，sigma点按时间向前投影，形成新的先验模型。新的先验模型由一系列sigma点组成，我们命名为$\boldsymbol{\mathcal{Y}}$:

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

我们对变换后的sigma点进行**无迹变换**，计算先验信息的均值和协方差。

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

这些是无迹变换的方程：

$$\bar{\mathbf{x}} = \sum_{i=0}^{2n} w_i^m \boldsymbol{\mathcal{Y}}_i$$

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

下表比较了线性卡尔曼滤波和无迹卡尔曼滤波方程。为了可读性，我省略了下标I

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

Update Step

卡尔曼滤波器在测量空间进行更新。因此，我们必须使用您定义的测量函数h(x)将先验的sigma点转换为测量值。

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

我们使用unscented变换计算这些点的均值和协方差。z下标表示这些是测量σ点的均值和协方差。

$$\boldsymbol{\mu}_z, \mathbf{P}_z = UT(\boldsymbol{\mathcal{Z}}, w_m, w_c, \mathbf{R})$$

$$\boldsymbol{\mu}_z = \sum_{i=0}^{2n} w_i^m \boldsymbol{\mathcal{Z}}_i$$

$$\mathbf{P}_z = \sum_{i=0}^{2n} w_i^c (\boldsymbol{\mathcal{Z}}_i - \boldsymbol{\mu}_z)(\boldsymbol{\mathcal{Z}}_i - \boldsymbol{\mu}_z)^\mathsf{T} + \mathbf{R}$$

接下来我们计算残差和卡尔曼增益。z的测量残差很容易计算:

$$\mathbf{y} = \mathbf{z} - \boldsymbol{\mu}_z$$

为了计算卡尔曼增益，我们首先计算状态和测量的交叉协方差，定义为:

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

然后卡尔曼增益如下被定义为:

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

如果你把卡尔曼增益看作是**某种**矩阵的倒数，你可以看到卡尔曼增益是一个简单的比率，计算如下:

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

最后，利用残差和卡尔曼增益计算新的状态估计:

$$\mathbf{x} = \bar{\mathbf{x}} + \mathbf{K}\mathbf{y}$$

新的协方差计算如下:

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

这一步包含了一些你必须相信的方程，但你应该能够看到它们与线性卡尔曼滤波方程的关系。线性代数与线性卡尔曼滤波略有不同，但算法是本书中一直在实现的贝叶斯算法。

下表比较了线性KF和UKF方程。

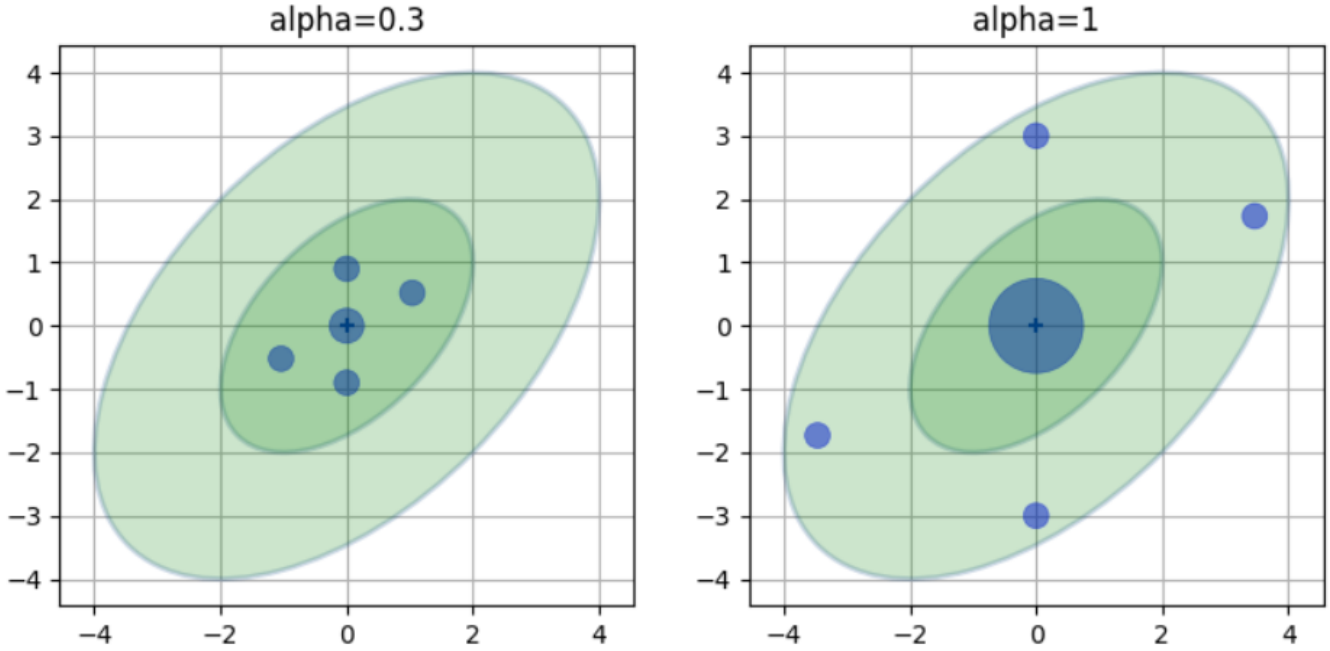
Kalman Filter	Unscented Kalman Filter	
	$\boldsymbol{\mathcal{Y}} = f(\boldsymbol{\chi})$	
$\bar{\mathbf{x}} = \mathbf{F}\mathbf{x}$	$\bar{\mathbf{x}} = \sum w^m \boldsymbol{\mathcal{Y}}$	$nbsp;$
$\bar{\mathbf{P}} = \mathbf{F}\mathbf{P}\mathbf{F}^\mathsf{T} + \mathbf{Q}$	$nbsp;$	$\bar{\mathbf{P}} = \sum w^c (\boldsymbol{\mathcal{Y}} - \bar{\mathbf{x}})(\boldsymbol{\mathcal{Y}} - \bar{\mathbf{x}})^\mathsf{T} + \mathbf{Q}$
	$\boldsymbol{\mathcal{Z}} =$	$nbsp; h(\boldsymbol{\mathcal{Y}})$
	$\boldsymbol{\mu}_z = \sum w^m \boldsymbol{\mathcal{Z}}$	
$\mathbf{y} = \mathbf{z} - \mathbf{H}\mathbf{x}$	$\mathbf{y} = \mathbf{z} - \boldsymbol{\mu}_z$	
$\mathbf{S} = \mathbf{H}\bar{\mathbf{P}}\mathbf{H}^\mathsf{T} + \mathbf{R}$	$\mathbf{P}_z = \sum w^c (\boldsymbol{\mathcal{Z}} - \boldsymbol{\mu}_z)(\boldsymbol{\mathcal{Z}} - \boldsymbol{\mu}_z)^\mathsf{T} + \mathbf{R}$	
$\mathbf{K} = \bar{\mathbf{P}}\mathbf{H}^\mathsf{T}\mathbf{S}^{-1}$	$\mathbf{K} = \left[\sum w^c (\boldsymbol{\mathcal{Y}} - \bar{\mathbf{x}})(\boldsymbol{\mathcal{Z}} - \boldsymbol{\mu}_z)^\mathsf{T}\right]\mathbf{P}_z^{-1}$	
$\mathbf{x} = \bar{\mathbf{x}} + \mathbf{K}\mathbf{y}$	$\mathbf{x} = \bar{\mathbf{x}} + \mathbf{K}\mathbf{y}$	
$\mathbf{P} = (\mathbf{I} - \mathbf{K}\mathbf{H})\bar{\mathbf{P}}$	$\mathbf{P} = \bar{\mathbf{P}} - \mathbf{K}\mathbf{P}_\mathbf{z}\mathbf{K}^\mathsf{T}$	

Van der Merwe's Scaled Sigma Point Algorithm

选择sigma点的算法有很多。自2005年左右以来，研究和工业界基本上都把重点放在Rudolph Van der Merwe在他2004年的博士论文[1]中发表的版本上。它在各种问题上都表现良好，并且在性能和准确性之间有很好的权衡。它是Simon J. Julier[2]发布的*Scaled Unscented Transform*的轻微重构。

这个公式使用3个参数来控制sigma点的分布和加权方式:α、β和κ。在计算这些方程之前，让我们来看一个例子。我将在协方差椭圆的顶部绘制sigma点，以显示第一和第二标准差，并基于平均权重缩放这些点。
这个公式使用3个参数来控制sigma点的分布和加权方式:α、β和κ。在计算这些方程之前，让我们来看一个例子。我将在协方差椭圆的顶部绘制sigma点，以显示第一和第二标准差，并基于平均权重缩放这些点。

```
ukf_internal.plot_sigma_points()
```



我们可以看到，sigma点位于第一个和第二个标准差之间，而较大的α将这些点分散开来。此外，较大的α对均值(中心点)的权重大于较小的α，其余的权重较小。这应该符合我们的直觉——距离均值越远的点，我们应该对其加权越小。我们还不知道这些权重和sigma点是如何选择的，但这些选择看起来是合理的。

Sigma Point Computation

第一个sigma点是输入的均值。这就是上图中显示在椭圆中心的sigma点。我们将其命名为$\boldsymbol{\chi}_0$.

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

为了符号方便， 我们定义$\lambda = \alpha^2(n + \kappa) - n$其中n是x的维度。剩余的sigma点计算为：

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

i下标表示矩阵的第i^{th}行向量。

换句话说，我们将协方差矩阵缩放一个常数，取其平方根，并通过从均值中加减它来确保对称性。稍后我们将讨论如何计算矩阵的平方根。

Weight Computation

这个公式使用一组权重表示均值，另一组权重表示协方差。$\boldsymbol{\chi}_0$的均值的权重计算为

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

$\boldsymbol{\chi}_0$的协方差的权重为：

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

其余sigma点的权重$\boldsymbol{\chi}_1 \ldots \boldsymbol{\chi}_{2n}$对于均值和协方差来说，是相同的。他们是

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

为什么这是“正确的”可能并不明显，事实上，它不能证明这是理想的所有非线性问题。但你可以看到，我们选择的sigma点与协方差矩阵的平方根成正比，而方差的平方根是标准差。因此，σ点大致按照±1σ乘以某个比例因子分布。分母中有一个n项，因此维度越多，点将被分散，权重越小。
重要提示:通常这些权重的和不等于1。我收到了很多关于这个的问题。期望得到的权重之和大于1，甚至是负值。我将在下面更详细地介绍这一点。

Reasonable Choices for the Parameters

β=2 对于高斯问题来说是一个很好的选择，κ=3−n其中n是x的维度，对于κ来说是一个很好的选择，而0≤α≤1对于α来说是一个合适的选择，α的较大值会使sigma点远离均值。

Using the UKF

让我们来解决一些问题，这样你就可以对UKF的使用有信心。我们将从一个你已经知道如何用线性卡尔曼滤波器解决的线性问题开始。虽然UKF是为非线性问题设计的，但它与线性卡尔曼滤波器对线性问题的最优结果相同。我们将编写一个过滤器来使用匀速模型在2D中跟踪物体。这将允许我们专注于什么是相同的(和大多数是相同的!)和什么是不同的UKF。
设计卡尔曼滤波器需要指定x、F、H、R和Q矩阵。我们已经做过很多次了所以我将不做过多讨论地给出矩阵。我们想要一个恒定速度的模型，所以我们将x定义为

$$\mathbf{x} = [x \quad nbsp; \dot{x} \quad y \quad \dot{y}]^\mathsf{T}$$

通过状这个态变量的排列顺序得到的状态转移矩阵是:

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

根据牛顿运动原理：

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

我们的传感器提供位置但不提供速度，因此测量函数为

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

传感器读数以米为单位，*x*和*y*的误差均为σ=0.3m。这给了我们一个测量噪声矩阵

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

最后，我们假设过程噪声可以用离散白噪声模型表示——也就是说，在每个时间周期内加速度是恒定

的。我们可以使用`FilterPy`的`Q_discrete_white_noise()`来为我们创建这个矩阵，但为了方便复习，矩阵是

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

我们如下计算:

```
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
```

接下来我们设计一个测量函数. 正如在线性卡尔曼滤波器中的测量方程将滤波器的先验转为测量值.我们需要将飞行器的位置和速度转为仰角和与雷达站的距离
距离可以如下计算:

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

仰角ϵ是y/x的反正切:

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

我们需要定义一个python函数来计算他。我将利用函数可以拥有一个变量来存储雷达的位置这一特性。虽然对于这个问题来说这是不必要的(我们可以硬编码值，或使用全局变量)，但这给了函数更多的灵活性。

```
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)
```

我们没有考虑非线性，即角度是模的。残差是测量值与先验投影到测量空间的差值。359°和1°的角度差是2°，而359°- 1°= 358°。计算unscented变换中加权值之和的UKF加剧了这一问题。现在，我们将传感
```

器和目标放置在避免这些非线性区域的位置。稍后我将向你展示如何处理这个问题。

我们需要模拟雷达和飞机。到目前为止，这应该是你的习惯，所以我不加讨论地提供代码。

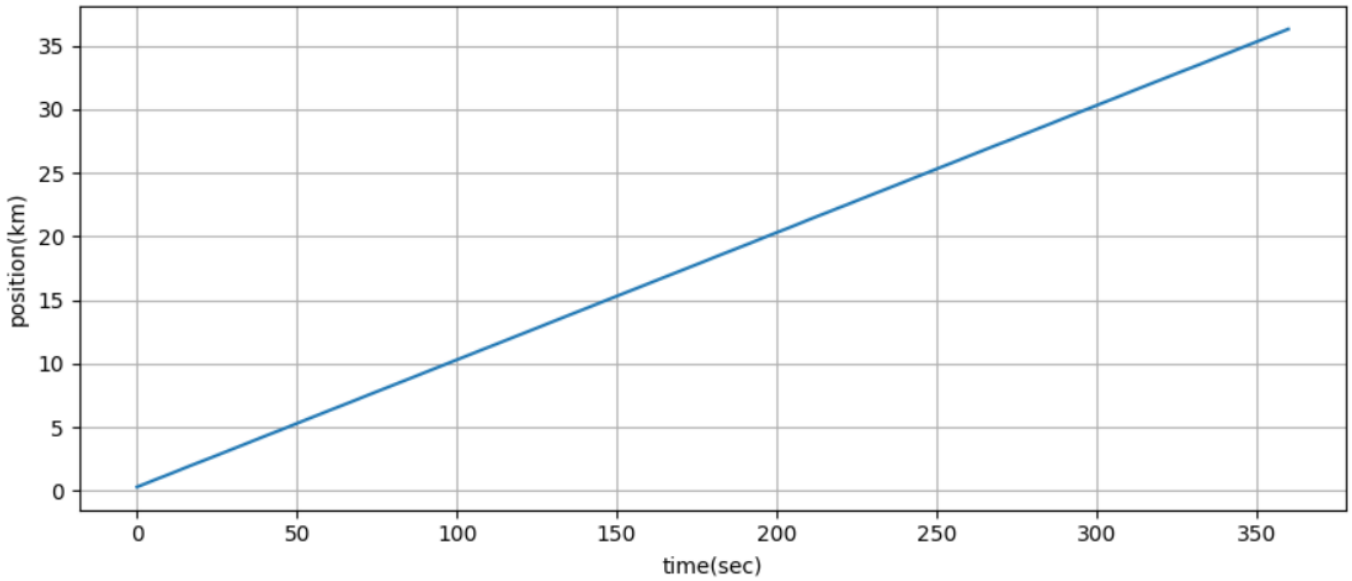
```
from numpy.linalg import norm
from math import atan2

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
```

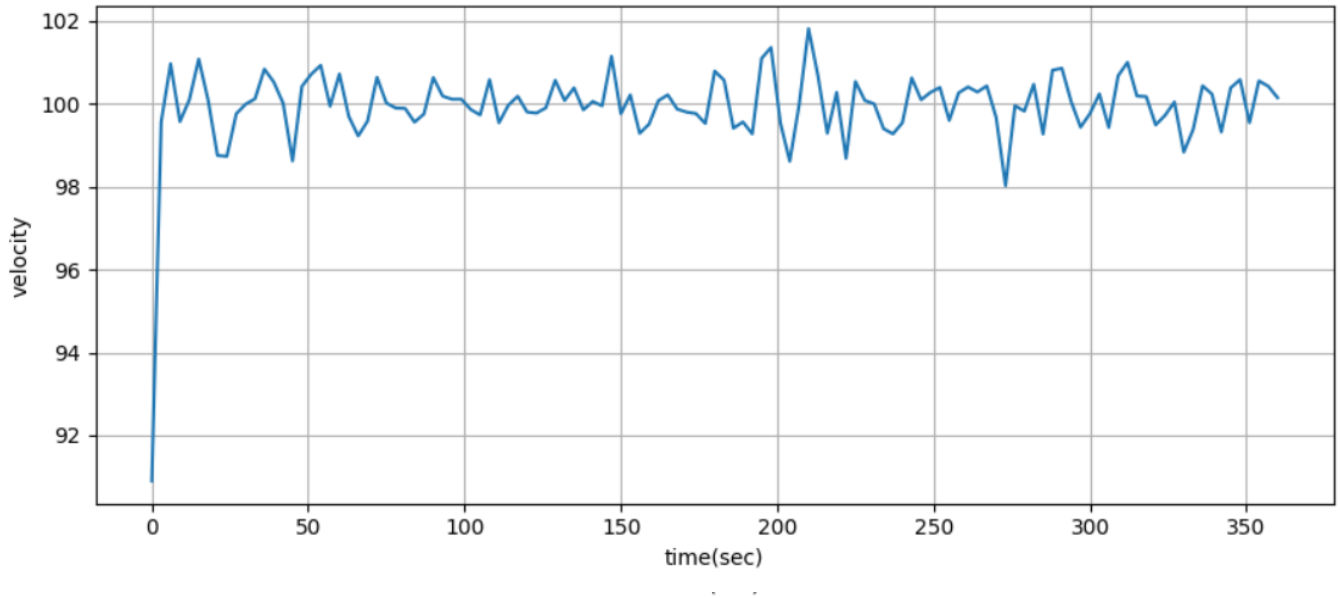
军用级雷达达到1米的RMS距离精度，1 mrad RMS俯仰角[3]。我们将假设更适中的5米距离精度和0.5°角精度，因为这为滤波器提供了更具有挑战性的数据集。
Q的设计需要讨论。状态是$[x \quad \dot{x} \quad y]^{\mathrm{T}}$。前两个元素是向下的距离和速度，因此我们可以使用`Q_discrete_white_noise`噪声来计算q的左上角的值。第三个元素是高度，我们假设它与x无关。这样就得到了Q的设计:

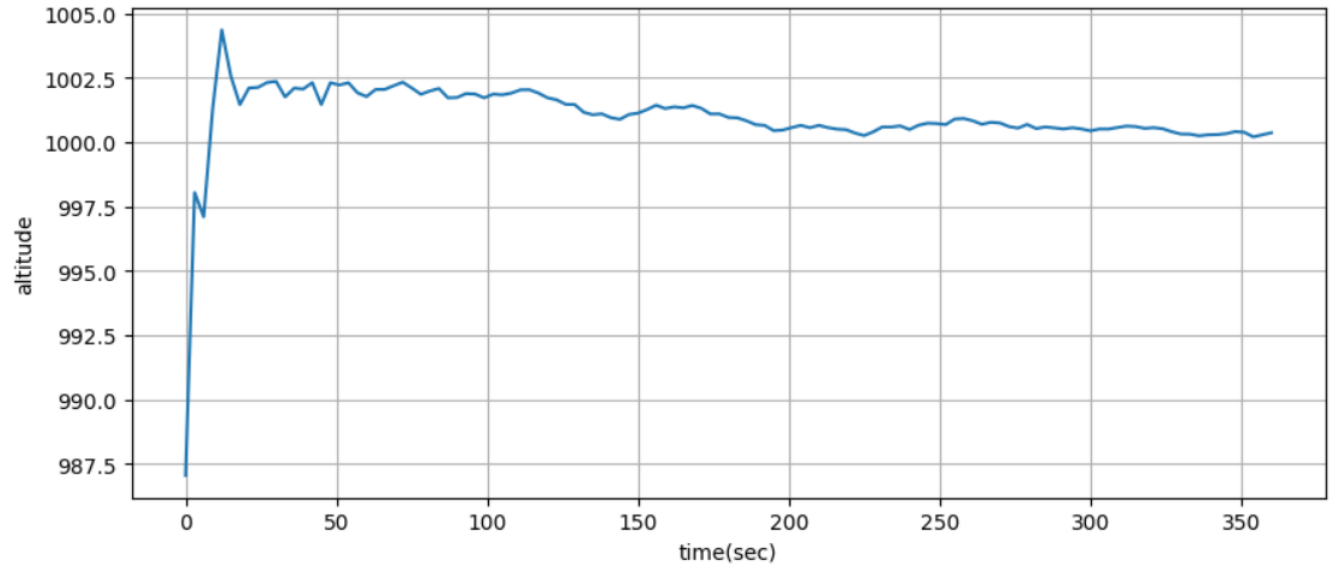
$$\mathbf{Q} = \begin{bmatrix} \mathbf{Q}_{\mathrm{x}} & \mathbf{0} \\ \mathbf{0} & Q_{\mathrm{y}} \end{bmatrix}$$

我将从位于雷达站正上方的飞机开始，以100米/秒的速度飞行。典型的高度计雷达可能每3秒更新一次，所以我们将使用它作为epoch周期。

```
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)
```







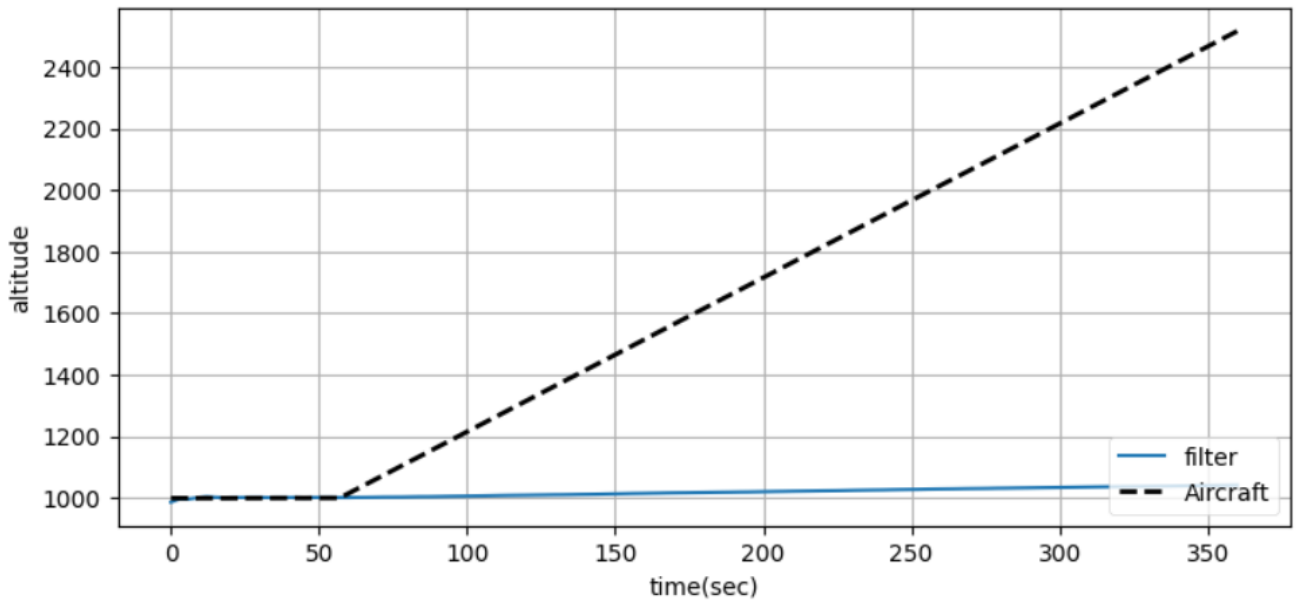
这也许不会给惊艳你, 但这的确惊艳了我! 在扩展卡尔曼滤波的章节我们会解决同样地问题, 但会有相当多的数学推导。

Tracking Maneuvering Aircraft

前一个例子产生了很好的结果, 但它假设了飞行器不会改变海拔高度。这里飞行器在1分钟后开始攀升高度的结果

```
from kf_book.ukf_internal import plot_altitude

# reset aircraft position
kf.x = np.array([0., 90., 1100.])
kf.P = np.diag([300**2, 30**2, 150**2])
ac = ACSim(ac_pos, (100, 0), 0.02)

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

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

```
Actual altitude: 2515.6
UKF altitude   : 1042.0
```



滤波器没能跟踪爬升的海拔高度。我们应该怎么改变我们的设计呢?

我希望你能回答:"增加爬升率到状态向量中", 像这样:

```
\mathbf x = \begin{bmatrix}\mathtt{distance} \\\mathtt{velocity}\\ \mathtt{altitude} \\ \
```

这需要将线性的状态转移方程也更改成下面的样子:

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

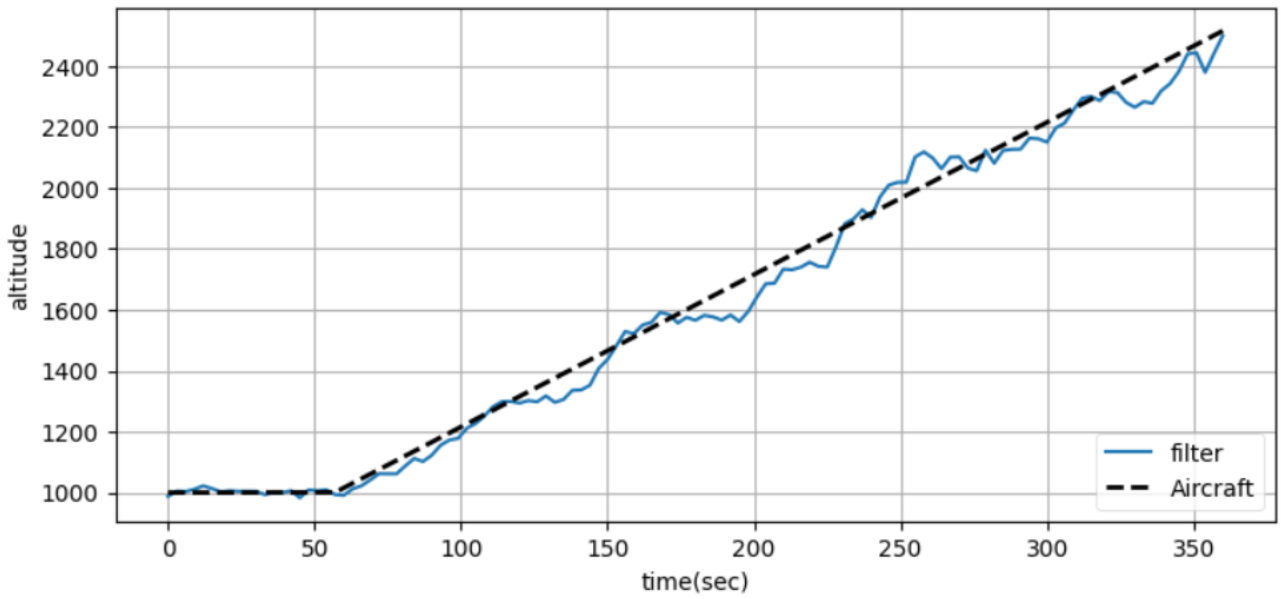
测量方程保持不变, 但我们必须改变Q来应对x维度上的变化

```
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
```

```
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   : 2499.7
```



我们对海拔高度的测量添加了相当大的噪声, 但我们现在能正确地跟踪海拔高度的变化

Sensor Fusion

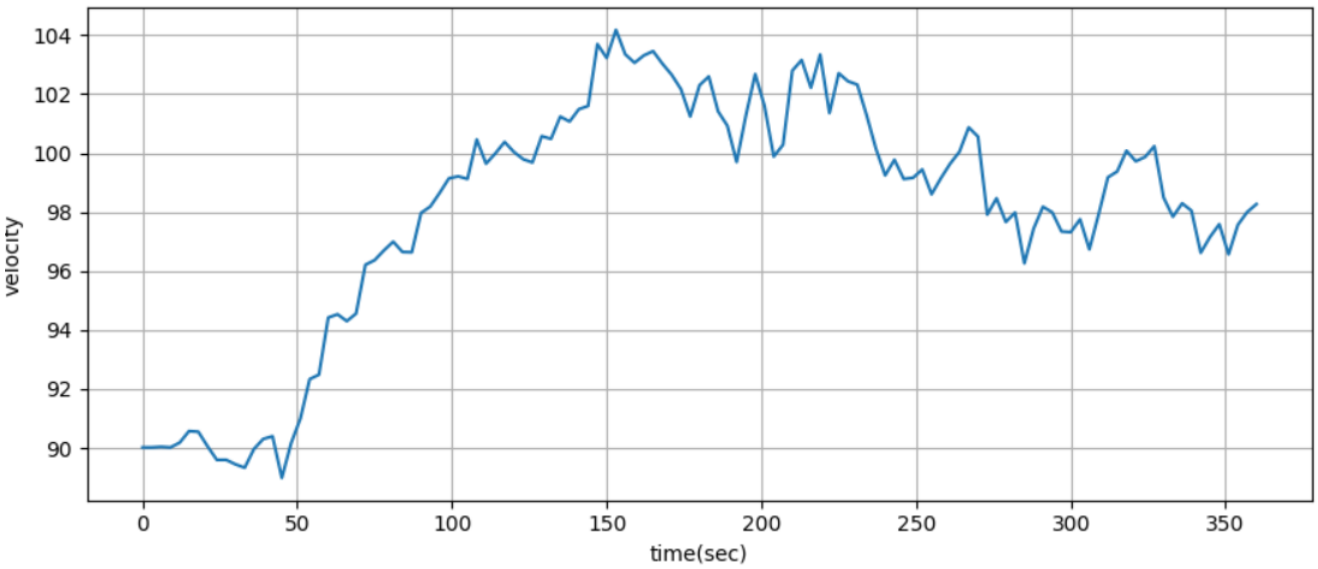
现在让我们考虑一个传感器融合的例子。我们有某种类型的多普勒系统，可以产生RMS精度为2米/秒的速度估计。我说“某种类型”是因为和雷达一样，我不想教你如何为多普勒系统创建一个精确的滤波器。一个完整的实现必须考虑到信噪比、大气效应、系统的几何形状，等等.

在上一个例子中，雷达的精度允许我们估计速度在1米/秒左右，我将降低精度来说明传感器融合的效果。我们将距离误差改为σ=500，然后计算估计速度的标准差。我将跳过前几个测量，因为滤波器在这段时间内是收敛的，会人为地造成较大的偏差。

不使用多普勒的标准偏差为:

```
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')
```



对于多普勒，我们需要在测量中包括x和y的速度。`ACSim`类将速度存储在数据成员`vel`中。要执行卡尔曼滤波器更新，我们只需要调用`update`，其中包含x和y的倾斜距离、仰角和速度向量:

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

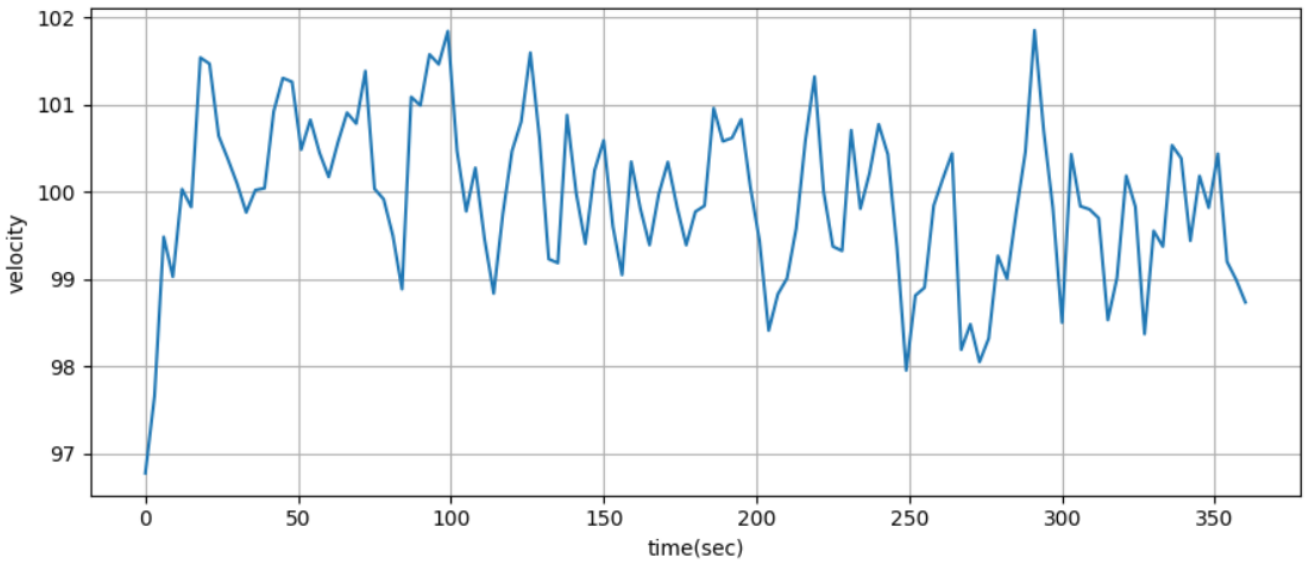
测量值包含4个值，因此测量函数也需要返回4个值。倾斜范围和仰角将像以前一样计算，并且我们不需要以x和y计算速度，因为它们由状态估计提供。

```
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]
```

这下我们可以实现这个滤波器了

```
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')
```



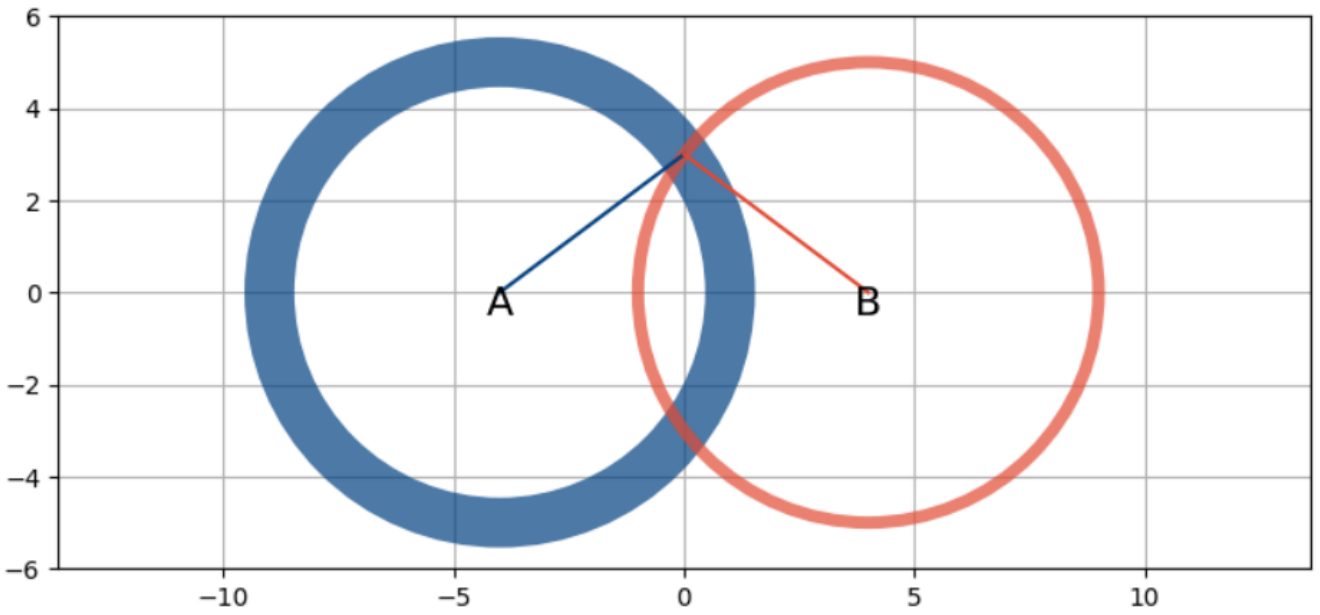
通过使用速度传感器，我们能够将标准偏差从3.5米/秒降低到1.3米/秒。

传感器融合是一个很大的话题，这是一个相当简单的实现。在典型的导航问题中，我们有传感器提供*补充*信息。例如，GPS可能每秒提供一次比较准确的位置更新，但速度估计较差，而惯性系统可能在50Hz时提供非常准确的速度更新，但位置估计很糟糕。每个传感器的优缺点相互正交。这就产生了*互补滤波器*，它将高更新率惯性速度测量与精确但缓慢更新的GPS位置估计相融合，以产生高速率和准确的位置和速度估计。高速速度估计集成在GPS更新之间，以产生准确和高速率的位置估计。

通过使用速度传感器，我们能够将标准偏差从3.5米/秒降低到1.3米/秒。

传感器融合是一个很大的话题，这是一个相当简单的实现。在典型的导航问题中，我们有传感器提供*补充*信息。例如，GPS可能每秒提供一次比较准确的位置更新，但速度估计较差，而惯性系统可能在50Hz时提供非常准确的速度更新，但位置估计很糟糕。每个传感器的优缺点相互正交。这就产生了*互补滤波器*，它将高更新率惯性速度测量与精确但缓慢更新的GPS位置估计相融合，以产生高速率和准确的位置和速度估计。高速速度估计集成在GPS更新之间，以产生准确和高速率的位置估计。

```
ukf_internal.show_two_sensor_bearing()
```



我们计算传感器和目标之间的方位

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

滤波器以矢量的形式接收来自两个传感器的测量值。代码可以接受任何可迭代的容器，因此我使用Python列表来提高效率。我们可以这样实现:

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

假设飞机是匀速模型。为了改变计算速度，我显式地计算新的位置，而不是使用矩阵向量乘法:\

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

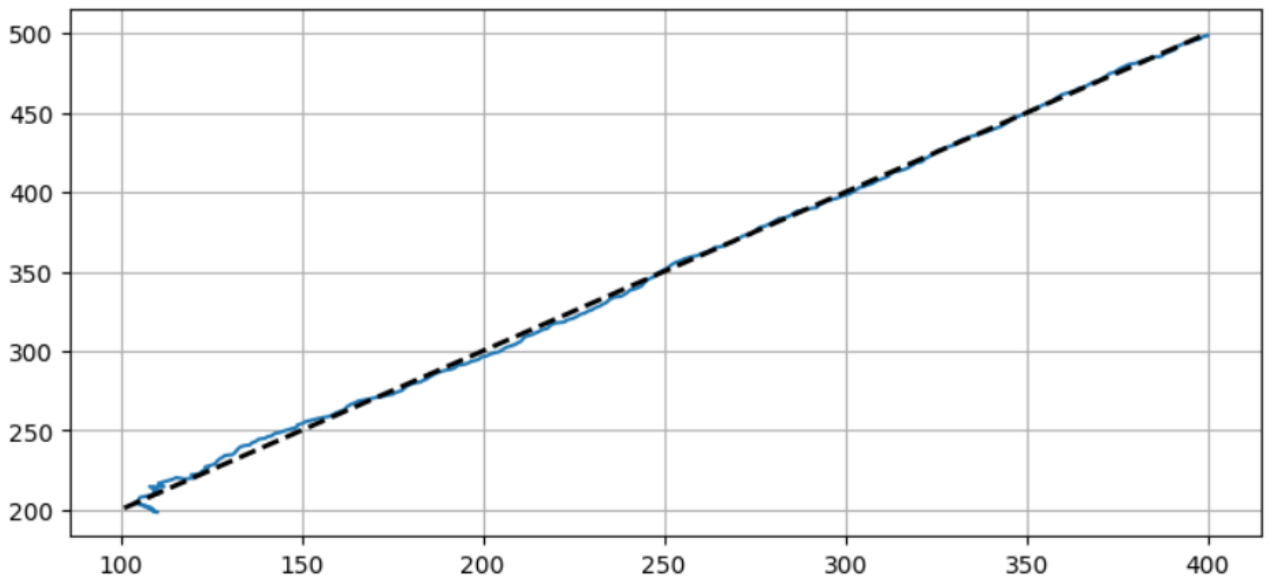
接下来我们实现度量函数。它将包含测量的阵列转换为两个站点的先验。我不是全局变量的粉丝，但我将站点的位置放在全局变量 `sa_pos` 和 `sb_pos` 中，以演示使用$h()$共享数据的方法:

```
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)
```

现在我们编写boilerplate来构建过滤器，运行它，并绘制结果:

```
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()
```

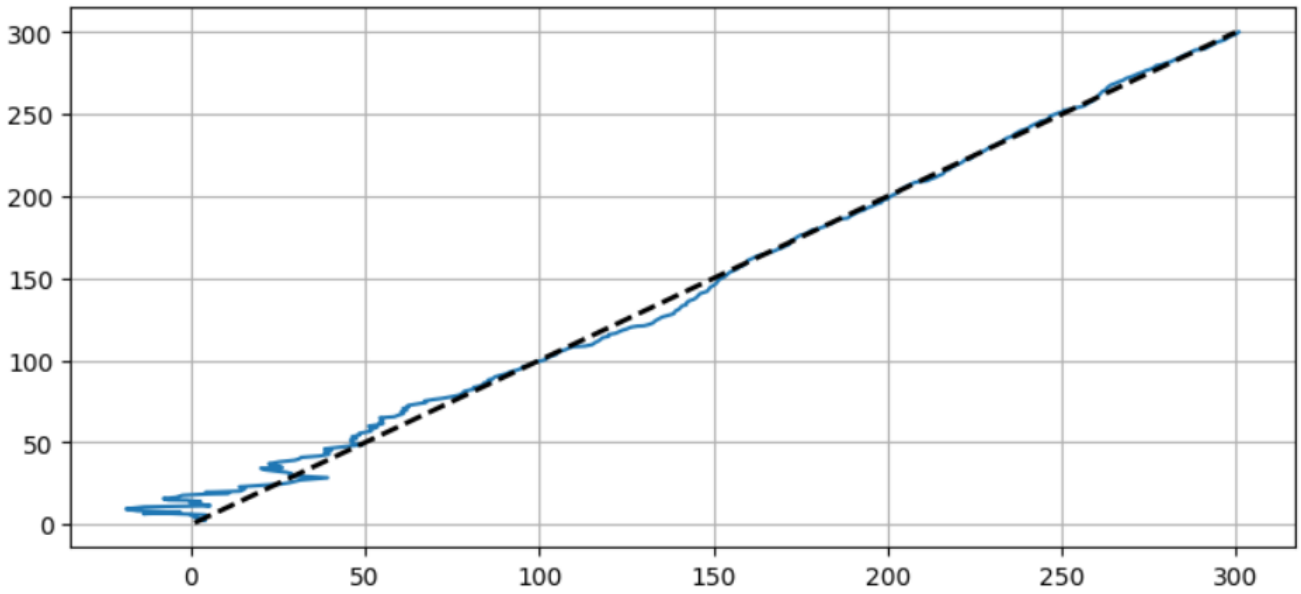
```
np.random.seed(123)
target_pos = [100, 200]

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



我觉得这个不错。航迹的开始出现很大的误差，但滤波器稳定下来并产生良好的估计。

让我们回顾一下角度的非线性。我将目标定位在两个传感器之间(0,0)。这将导致残差计算的非线性，因为平均角度将接近零。当角度小于0时，测量函数将计算出一个接近2π的大正角。因此，预测和测量之间的残差将非常大，接近2π，而不是接近0。这使得过滤器无法准确地执行，如下面的例子所示。

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

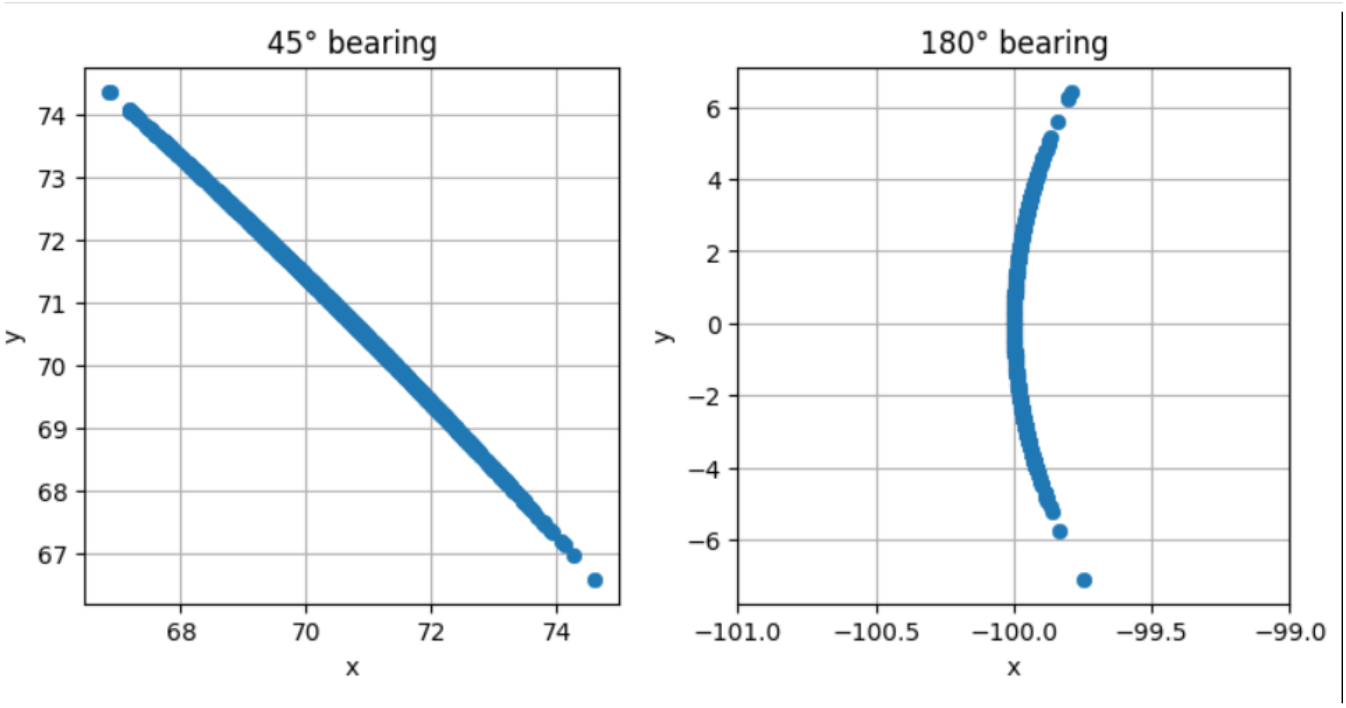


这种表现令人无法接受。`FilterPy`的UKF代码允许你指定一个函数，在类似这样的非线性情况下计算残差。本章的最后一个例子演示了它的用法。

Effects of Sensor Error and Geometry

传感器相对于被跟踪对象的几何形状施加了一个物理限制，在设计滤波器时可能非常难以处理。如果VOR台的径向几乎平行于彼此，那么非常小的角度误差会转化为非常大的距离误差。更糟糕的是，这种行为是非线性的——*x轴*与*y轴*的误差将根据实际轴承而变化。这些散点图显示了两个不同轴承1°σ误差的误差分布。

```
ukf_internal.plot_scatter_of_bearing_error()
```



Exercise: Explain Filter Performance

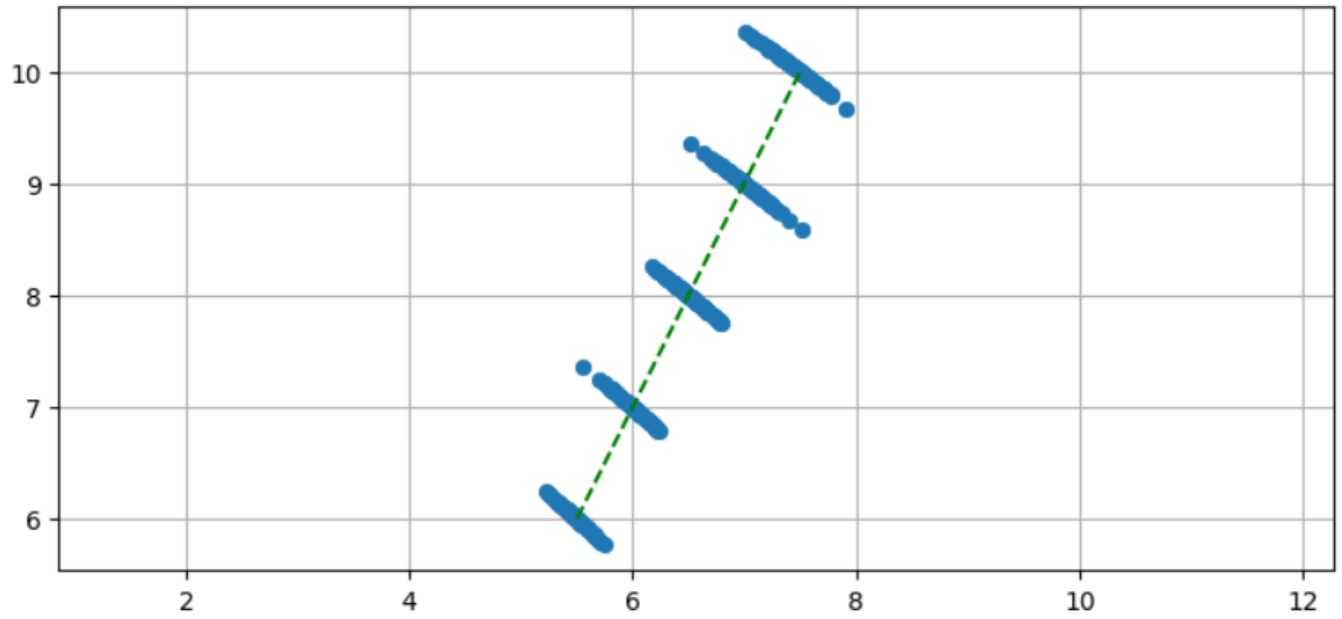
我们可以看到，对于较小的角度误差，位置误差非常大。解释我们如何在上述目标跟踪问题中从UKF中获得相对良好的性能。解决了单传感器和多传感器问题。

Solution

理解这一点非常重要。在阅读下面的答案之前，努力回答这个问题。如果你不能回答这个问题，你可能需要重新阅读*多维卡尔曼滤波*一章中的一些内容。

有几个因素促成我们的成功。首先，让我们考虑只有一个传感器的情况。任何单一的测量都有一个可能位置的极端范围。但是，我们的目标在移动，UKF正在考虑这一点。让我们绘制一个移动目标的连续几次测量结果。

```
ukf_internal.plot_scatter_moving_target()
```



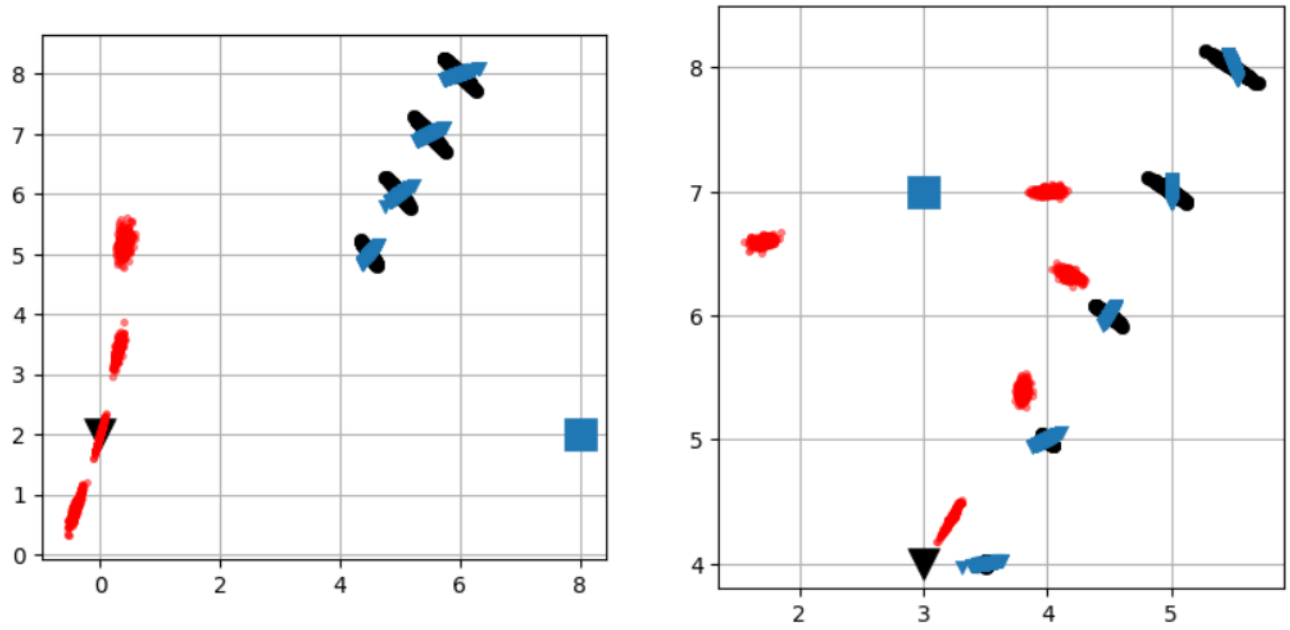
每次单独测量都有非常大的位置误差。然而，连续测量的图显示了一个明显的趋势——目标明显地向右上角移动。卡尔曼滤波器在计算卡尔曼增益时，利用测量函数考虑了误差的分布。在这个例子中，误差位于大约45°的直线上，因此滤波器将扣除该方向上的误差。另一方面，与此正交的测量几乎没有误差，卡尔曼增益也会考虑到这一点。

这个图看起来很简单，因为我们为每个位置更新绘制了100个测量值。飞机的运动很明显。相比之下，卡尔曼滤波器每次更新只能得到一个测量值。因此，滤波器将无法生成像绿色虚线所示的那样好的拟合。

现在考虑一下，轴承没有给我们距离信息。假设我们将初始估计值设置为距离传感器1000公里(与实际距离7.07公里相比)，并使P非常小。在这个距离上，1°的误差转化为17.5千米的位置误差。KF永远无法收敛到实际目标位置，因为滤波器对其位置估计不正确地非常确定，而且在测量中没有提供距离信息。

现在让我们考虑添加第二个传感器的影响。这里有两张图显示了不同传感器放置的影响。我使用正方形和三角形作为两个传感器的符号，并使用相同的符号形状和颜色绘制了每个传感器的误差分布。然后，我计算了与两个噪声轴承测量相对应的(x,y)坐标，并用红点绘制它们，以显示x和y中的噪声测量的分布。

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



在第一张图中，我将传感器放置在几乎与目标初始位置正交的位置，因此我们得到这些可爱的“x”形状的交叉点。我们可以看到x和y中的误差如何随着目标按分散红点的形状移动而变化——当目标离传感器越远，但越接近传感器B的y坐标时，形状变得强烈椭圆。

在第二幅图中，飞机从一个传感器附近开始，然后飞越第二个传感器。误差的交点是非正交的，由此产生的位置误差变得非常分散。

Implementation of the UKF

FilterPy实现了UKF，但学习如何将方程转换为代码是有指导意义的。实现UKF非常简单。首先，让我们编写代码来计算sigma点的均值和协方差。

我们将sigma点和权重存储在矩阵中，如下所示:

$$\text{weights} = \begin{bmatrix} w_0 & w_1 & \dots & w_{2n} \end{bmatrix}$$

$$\text{sigmas} = \begin{bmatrix} \mathcal{X}_{0,0} & \mathcal{X}_{0,1} & \dots & \mathcal{X}_{0,n-1} \\ \mathcal{X}_{1,0} & \mathcal{X}_{1,1} & & nbsp;\dots & \mathcal{X}_{1,n-1} \\ \vdots & \vdots & & nbsp;\ddots & \vdots \\ \mathcal{X}_{2n,0} & \mathcal{X}_{2n,1} & \dots & \mathcal{X}_{2n,n-1} \end{bmatrix}$$

为了描述一个非常简单的东西，我们使用了很多下标，下面是一个二维问题的例子(n=2):

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

均值的sigma点在第一行。它的位置是(0,0)，等于均值(0,0)。第二个sigma点位于位置(0.173,0.017)，以此类推。有2n+1=5行，每sigma点一行。如果n=3，那么将有3列7行。 将sigma存储为行-列格式还是列行格式在某种程度上是任意的;我的选择使剩下的代码更清晰，因为我可以将I^{th} sigma点称为`sigmas[I]`而不是`sigmas[:, I]`。

Weights

用NumPy计算权重很简单。回想一下Van der Merwe scaled sigma point的实现过程:

$$\begin{aligned}
\lambda &= \alpha^2(n+\kappa) - n \\
W_0^m &= \frac{\lambda}{n+\lambda} \\
W_0^c &= \frac{\lambda}{n+\lambda} + 1 - \alpha^2 + \beta \\
W_i^m = W_i^c &= \frac{1}{2(n+\lambda)} \quad i = 1..2n
\end{aligned}$$

计算这些的代码片段为:

```
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_)
```

Sigma Points

sigma点的方程如下:

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

一旦我们理解了\left[\sqrt{(n+\lambda)\Sigma} \right]_i使用python实现并不难
$\sqrt{(n+\lambda)\Sigma}$是一个矩阵, 因为Σ是一个矩阵, \left[\sqrt{(n+\lambda)\Sigma} \right]_i中的下标i表示选择矩阵的第i行向量。矩阵的平方根是多少?没有唯一的定义。一种定义是，矩阵Σ的平方根是矩阵S，当与自身相乘时，产生Σ, 如果$\Sigma = SS^T$, 则$S = \sqrt{\Sigma}$
这种定义之所以受到青睐，是因为S是使用Cholesky分解[5]计算的。它将厄米特正定矩阵分解为三角矩阵及其共轭转置。矩阵可以是上矩阵

或者下三角形，像这样:

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

星号表示共轭转置;我们只有实数，因此可以这样写:

$$A = LL^\mathsf{T} A = U^\mathsf{T} U$$

P具有这些属性，因此我们可以将S=cholesky(P)视为P的平方根。

SciPy在 `SciPy .linalg` 中提供了 `cholesky()` 方法。如果你选择的语言是Fortran、C或C++， LAPACK之类的库可以提供这个例程。Matlab提供了 `chol()` 。

默认情况下， `scipy.linalg.cholesky()` 返回一个上三角矩阵，所以我选择编写期望得到上三角矩阵的代码。因此，我按行访问结果，因此第一个sigma点，也就是中心点，会受到一整行非零值的影响。如果您提供自己的平方根实现，则需要考虑这一点。你可以在文献中找到先取values列的UKF算法。如果cholesky是下三角矩阵，或者你正在使用另一种计算对称矩阵的算法，那么行与列的顺序无关。

```
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)
```

所以我们可以用下面的代码来实现sigma点:

```
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]
```

现在让我们实现unscented转换。回想一下这些方程

$$\mu = \sum_i w_i^m \mathcal{X}_i$$
$$\Sigma = \sum_i w_i^c (\mathcal{X}_i - \mu)(\mathcal{X}_i - \mu)^\mathsf{T}$$

我们实现了均值的和

x = np.dot(Wm, sigmas)

如果你不是NumPy的忠实用户，这可能对你来说很陌生。NumPy不仅仅是一个线性代数库;在底层，它是用C和Fortran编写的，实现了比Python快得多的速度。典型的加速比是20到100倍。为了获得这种加速，我们必须避免使用for循环，而是使用NumPy内置的函数来执行计算。因此，我们不再编写for循环

来计算乘积的和，而是调用内置的`numpy. py`。Dot (x, y)`方法。两个向量的点积是每个元素相乘的和。如果传入一个一维数组和一个二维数组，它将计算内积之和:

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

剩下的就是计算了$\mathbf{P} = \sum_i w_i(\mathcal{X}_i - \mu)(\mathcal{X}_i - \mu)^\mathsf{T} + \mathbf{Q}$:

```
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
```

这介绍了NumPy的另一个特性。状态变量`x`是一维的，`sigmas[k]`也是一维的，因此差值`sigmas[k]-X`也是一维的。NumPy不会计算一维数组的转置。它认为`[1,2,3]`的转置是`[1,2,3]`。所以我们调用函数`np.outer(y,y)`来计算一维数组yy^T的值y。另一种实现方式是:

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

这段代码比较慢，也不符合习惯用法，所以我们不使用它。

Predict Step

对于预测步骤，我们将按照上面的规定生成权重和sigma点。我们让每个点通过函数f。

$$\boldsymbol{\mathcal{Y}} = f(\boldsymbol{\mathcal{X}})$$

然后我们使用无迹变换计算预测的均值和协方差。在下面的代码中，你可以看到我假设这是一个类中的方法，它存储了过滤器所需的各种矩阵和向量。

```
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)
```

Update Step

更新步骤通过函数`h(x)`将sigma转换到测量空间。

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

使用unscented变换计算这些点的均值和协方差。然后计算残差和卡尔曼增益。交叉方差的计算如下:

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

最后，我们使用残差和卡尔曼增益来计算新的状态估计:

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

新的协方差计算如下:

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

这个函数的实现方式如下，假设它是某个类的一个方法，用于存储必要的矩阵和数据.\

```
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)
```

FilterPy's Implementation

FilterPy在某种程度上概括了代码。你可以指定不同的sigma点算法，指定如何计算状态变量的残差(不能减去角度，因为它们是模函数)，提供矩阵平方根函数，等等。有关详细信息，请参阅帮助。
https://filterpy.readthedocs.org/#unscented-kalman-filter

Batch Processing

卡尔曼滤波是递归的——估计是基于当前测量值和先验估计。但是，我们想要过滤一组已经收集的数据是很常见的。在这种情况下，过滤器可以以*批处理*模式运行，一次性过滤所有测量值。

将测量值收集到一个数组或列表中。

zs = read_altitude_from_csv()

然后调用`batch filter()`方法。

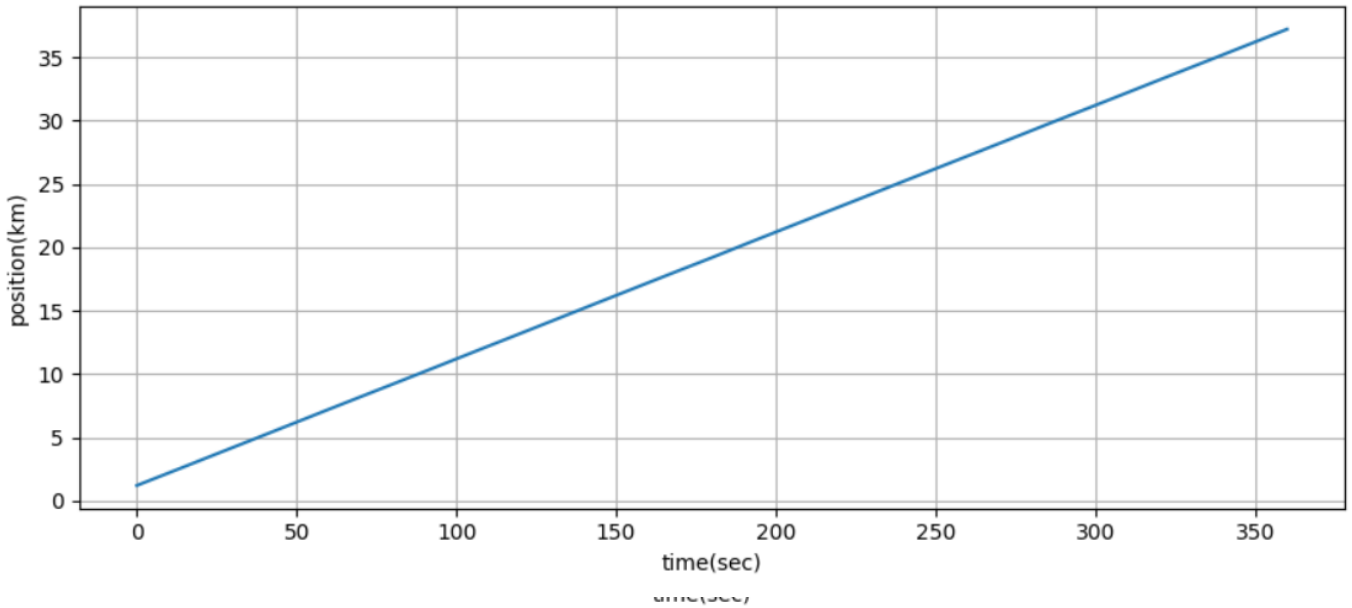
Xs, Ps = ukf.batch_filter(zs)

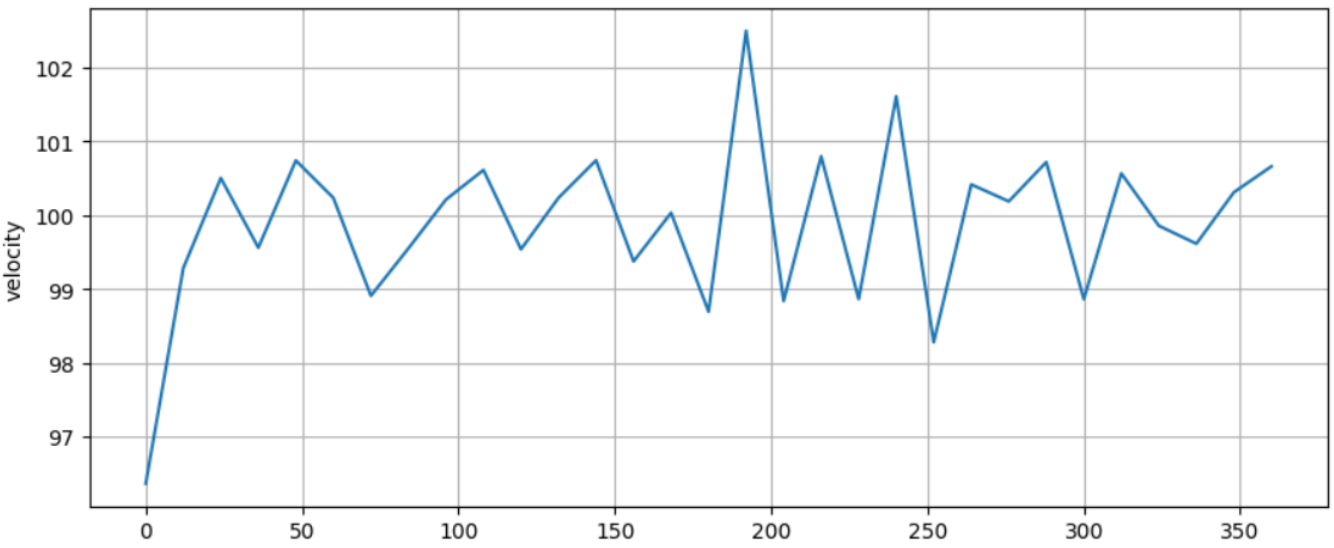
该函数接受测量值列表/数组，对其进行过滤，并返回整个数据集的状态估计数组(`Xs`)和协方差矩阵(`Ps`)。

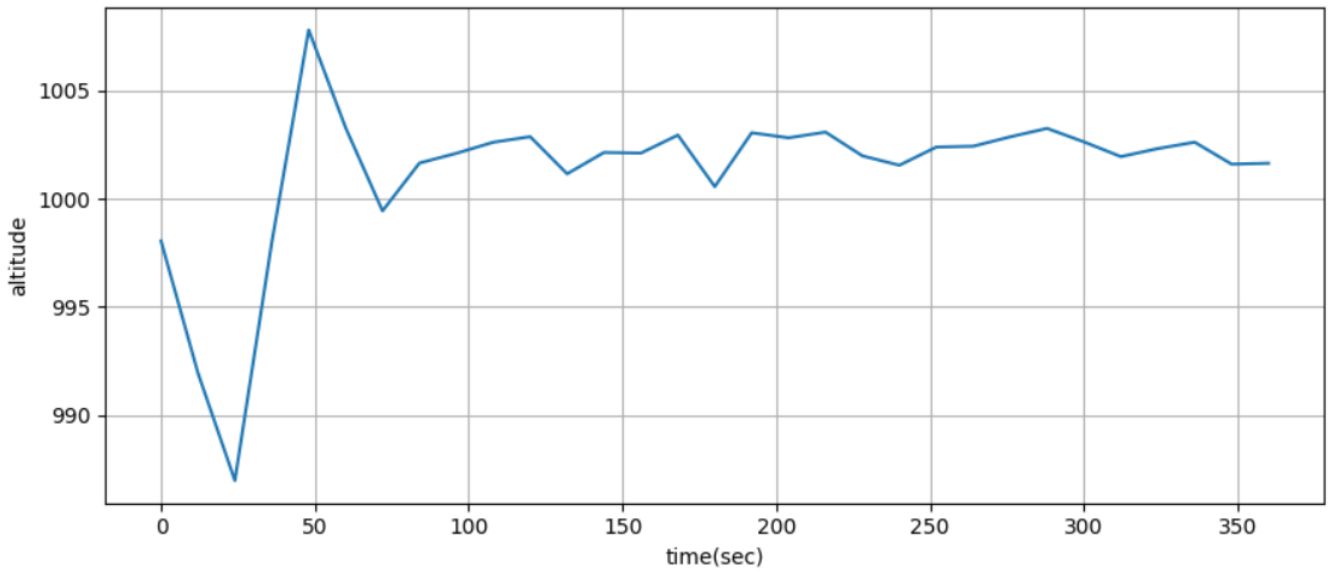
这是上面雷达跟踪问题的一个完整示例。

```
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)
```







Smoothing the Results

假设我们在追踪一辆车。假设我们得到一个噪声测量值，表明汽车开始向左转弯，但状态函数预测汽车是直线运动。卡尔曼滤波器别无选择，只能将状态估计向噪声测量方向移动，因为它无法判断这只是一个特别噪声的测量还是一个真正的开始。

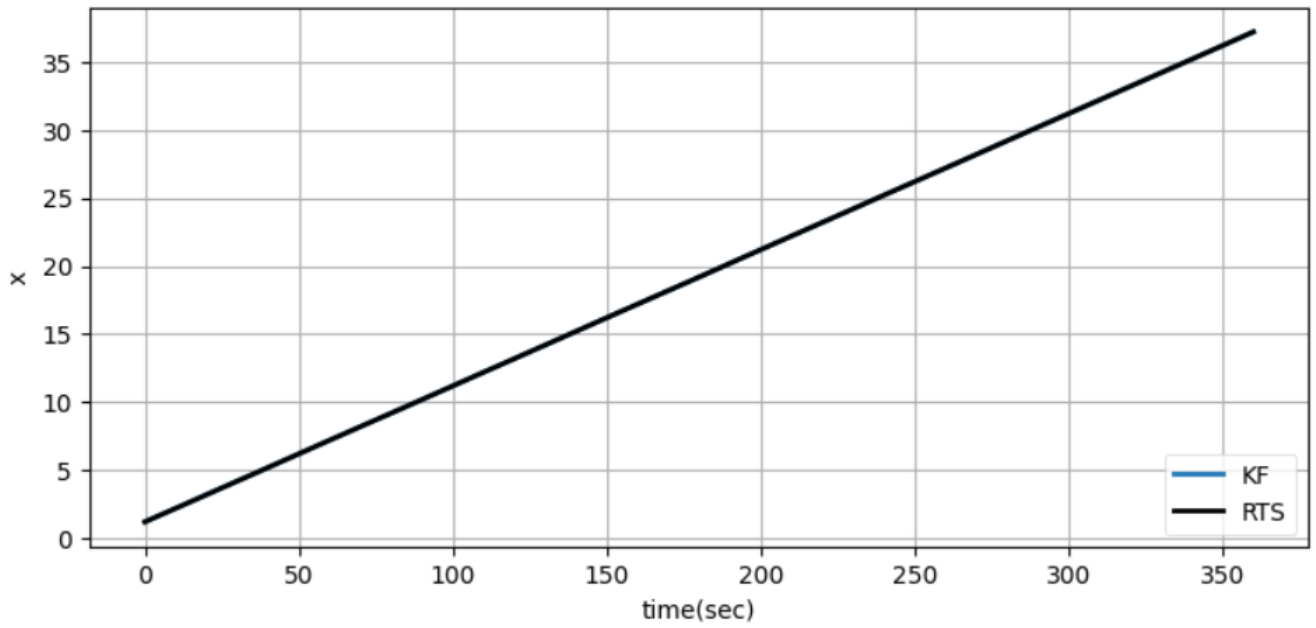
如果我们正在收集数据并对其进行后处理，我们会在有问题的数据之后进行测量，以告知我们是否进行了转弯。假设后续的测量值都继续向左转。然后我们可以确定测量没有很大的噪声，而是开始了一个回合。

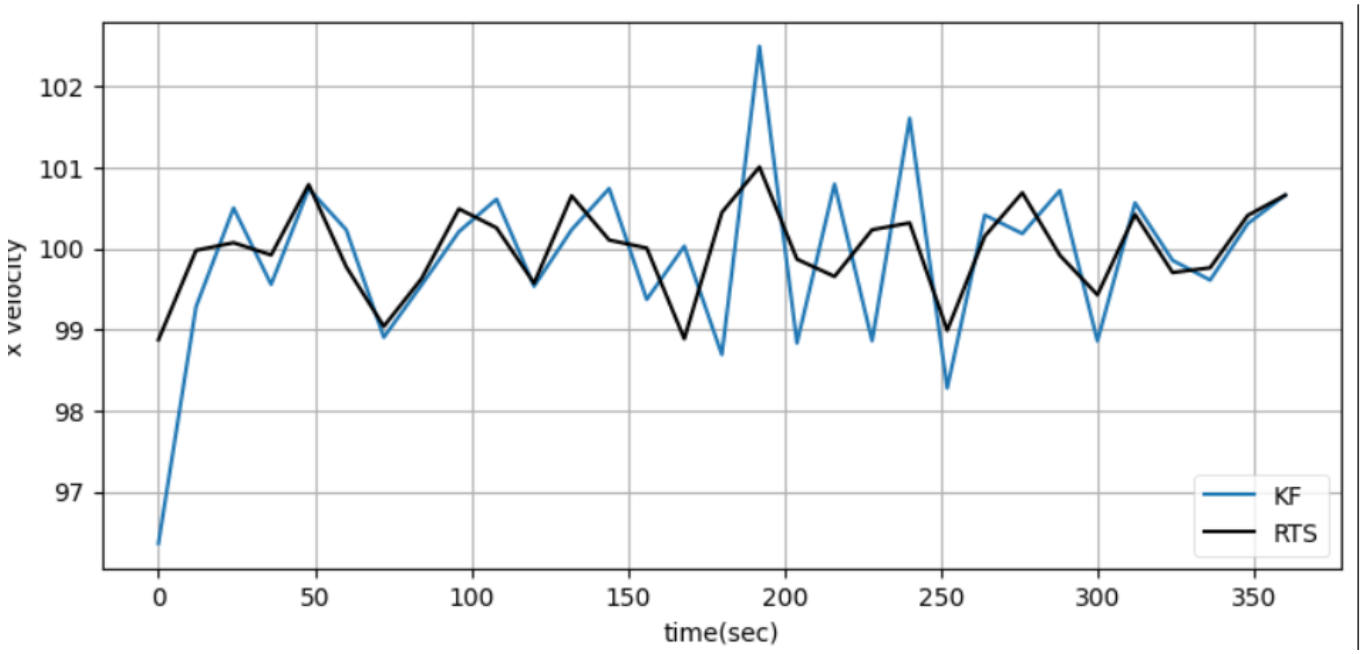
我们不会在这里开发数学或算法，我只是向你展示如何在`FilterPy`中调用算法。我们实现的算法被称为*RTS平滑器*，以该算法的三位发明者Rauch、Tung和Striebel的名字命名。

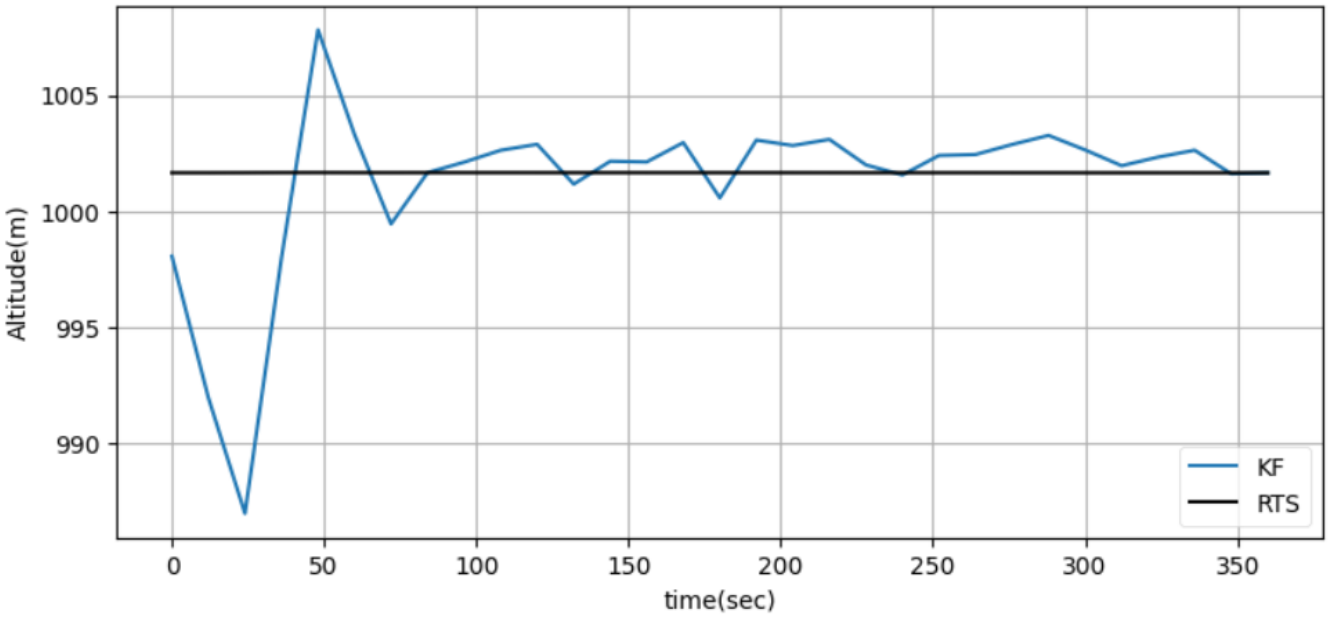
例程是`unscentedkalmanfilter . rts_smoothing()`。使用它是微不足道的;我们传入从`batch_filter`步骤计算出的均值和协方差，并接收回平滑的均值、协方差和卡尔曼增益。

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

```
Difference in position in meters:
        [-1.0928  0.1593  0.3259 -0.2323 -0.1727]
```







从这些图表中，我们可以看到位置上的改善很小，但速度上的改善很好，高度上的改善很壮观。位置上的差异非常小，所以我打印了最后5个点的UKF和平滑结果之间的差异。如果你可以对数据进行后处理，我建议你总是使用RTS smoothing。

Choosing the Sigma Parameters

我发现关于选择α, β和κ的价值的文献相当缺乏。范德默维的论文包含了最多的信息，但并不详尽。让我们来探索一下它们的作用。

对于高斯问题，Van der Merwe建议使用β=2，而κ=3−n。所以让我们从那里开始，改变α。我将使用n=1来最小化我们需要查看的数组的大小，并避免计算矩阵的平方根。

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

sigmas: [0. 3. -3.]
mean weights: [0.6667 0.1667 0.1667]
cov weights: [2.6667 0.1667 0.1667]
lambda: 2
sum cov 2.9999999999999996
那么这是怎么回事呢?我们可以看到，当均值为0时，算法选择sigma点为0、3和-3，但为什么呢?回想一下计算sigma点的公式:

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

我选择n=1将所有的都简化为标量，使我们可以避免计算矩阵的平方根。对于我们的值，方程是\

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

随着α的增大，点的分布也越来越分散。让我们把它设置为一个荒谬的值。

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

sigmas: [0. 600. -600.]
mean weights: [1. 0. 0.]
cov weights: [-39996. 0. 0.]
lambda: 119999
sum cov -39996.00000000001
可以看到这些点分布在100个标准差范围内。如果我们的数据是高斯分布的，我们将合并离均值有很多标准差的数据;对于非线性问题，这不大可能产生好的结果。但假设我们的分布不是高斯分布，而是尾巴很粗?我们可能需要从这些尾部采样来获得一个好的估计，因此将κ变大是有意义的(而不是200，它大得荒谬，使sigma点的变化明显)。

按照类似的推理方式，假设我们的分布几乎没有尾部——概率分布看起来更像一个倒抛物线。在这种情况下，我们可能希望将sigma点拉得更接近均值，以避免在永远不会有真实数据的区域进行采样。

现在我们来看看权重的变化。当我们有k+n=3时，均值的权重为0.6667，两个离群的sigma点的权重为0.1667。另一方面，当α=200时，平均权重飙升到0.99999，异常值权重被设置为0.000004。回想一下权重公式:

$$W_0 = \frac{\lambda}{n+\lambda}$$
$$W_i = \frac{1}{2(n+\lambda)}$$

我们可以看到，随着λ变大，均值权重的分数λ/(n+λ)接近1，其余sigma点的权重的分数接近0。这对于协方差的大小是不变的。抽样时，离均值越远赋予的权重越小，离均值越近赋予的权重越小。

然而，Van der Merwe给出的建议是将α限制在0>α≥10的范围内。他建议10^{-3}是个不错的选择。让我们试一下。

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

sigmas: [0. 0.0036 -0.0036]
mean weights: [-999999. 500000. 500000.]
cov weights: [-999996. 500000. 500000.]
lambda: -0.999999
sum cov 3.9999989999923855

Robot Localization - A Fully Worked Example

现在是解决一个重大问题的时候了。大多数书籍选择简单的教科书问题，并给出简单的答案，而你却想知道如何实现现实世界的问题。这个例子不会教你如何解决任何问题，但说明了在设计和实现过滤器时必须考虑的类型。

我们将考虑机器人定位问题。在这种情况下，我们有一个机器人正在通过一个景观，使用传感器来检测地标。这可能是一辆使用计算机视觉来识别树木、建筑物和其他地标的自动驾驶汽车。它可能是那些用吸尘器打扫你的房子的小机器人，或者是仓库里的机器人。

机器人有4个轮子，配置与汽车相同。它通过转动前轮来操纵。这使得机器人在前进时围绕后轴旋转。这是非线性行为，我们必须对其建模。

机器人有一个传感器，可以使它在地形中与已知目标的大致距离和方位。这是非线性的，因为根据距离和方位计算位置需要平方根和三角函数。

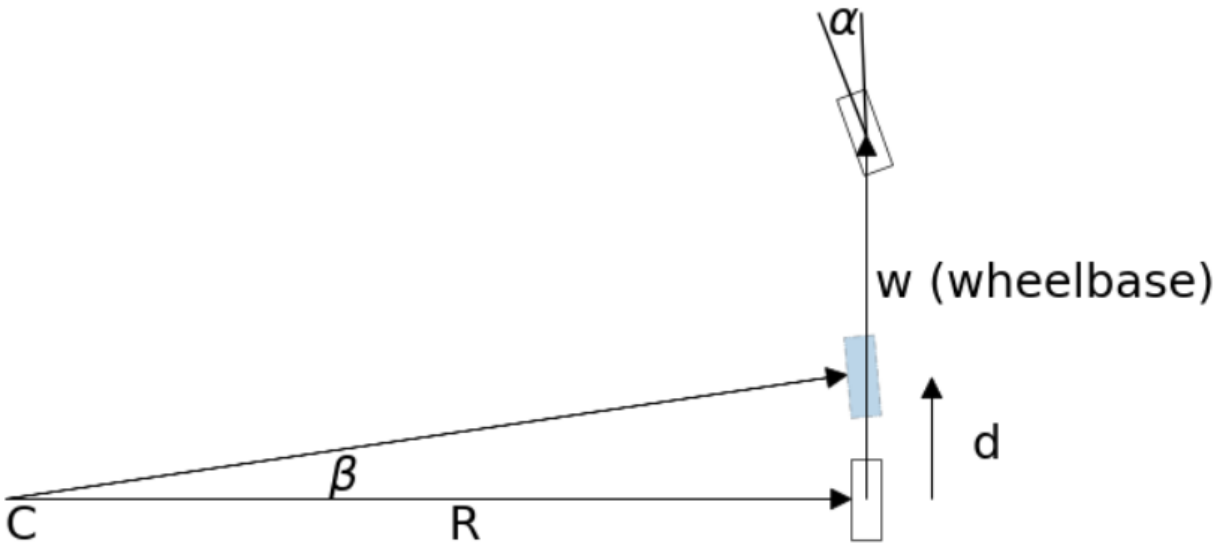
过程模型和度量模型都是非线性的。UKF容纳了这两者，因此我们暂时得出结论，对于这个问题，UKF是一个可行的选择。

Robot Motion Model

汽车在向前行驶时通过旋转前轮来控制方向。在围绕后轮旋转时，汽车的前部朝轮子所指的方向移动。这个简单的描述由于一些问题而变得复杂，例如摩擦引起的打滑，不同速度下橡胶轮胎的不同性能，以及外胎需要比内胎行驶不同的半径。精确建模转向需要一套复杂的微分方程。

对于卡尔曼滤波，特别是低速机器人应用，一个更简单的*自行车模型*被发现表现良好。下面是模型的描述:

```
ekf_internal.plot_bicycle()
```



在这里，我们看到前轮是指向方向α相对于轴距。在很短的一段时间内，汽车向前移动，后轮结束在更远的前方，并稍稍向内转向，如图所示的蓝色阴影轮胎。在如此短的时间内，我们可以将其近似为一个半径的转身R。我们可以用β计算转弯的角度

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

转弯半径R由如下式给出

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

在这里，后轮行驶的距离给出一个前进速度v是d=vΔt。
用θ作为机器人的方向，我们在转弯开始前计算位置C

$$C_x = x - R\sin(\theta)$$
$$C_y = y + R\cos(\theta)$$

向前移动时间Δt后，机器人的新位置和方向为

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

一旦我们代入C，我们得到

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

如果您对转向模型不感兴趣，则不需要详细了解此数学。重要的是要认识到我们的运动模型是非线性的，我们需要用卡尔曼滤波器来处理它。

Design the State Variables

对于我们的机器人，我们将保持位置和方向:

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

我可以在这个模型中包含速度，但正如你将看到的那样，数学已经相当具有挑战性。 控制输入u是所要求的速度和转向角

$$u = \begin{bmatrix} v & a \end{bmatrix}^T$$

Design the System Model

将系统建模为非线性运动模型加白噪声。

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

使用我们上面创建的机器人的运动模型，我们可以这样写:

```
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])
```

我们将使用这个函数来实现状态转换函数`f(x)`。

我将设计UKF，使Δt很小。如果机器人移动速度很慢，那么这个函数应该给出一个合理准确的预测。如果Δt很大或者系统的动态是非常非线性的，这种方法将会失败。在这些情况下，您将需要使用更复杂的数值积分技术(如龙格库塔)来实现它。数值积分在**卡尔曼滤波数学**一章中有简要介绍。

Design the Measurement Model

该传感器提供了一个噪声方位和范围的多个已知地点的景观。测量模型必须将状态$[x \quad y \quad \theta]^\mathsf{T}$转换为与地标相关的范围。如果p是地标的位置，则r为

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

我们假设传感器提供了相对于机器人方向的方位，因此必须从方位中减去机器人的方向才能得到传感器的读数，如下所示:

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

因此我们的测量函数是

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

我还不会实现它，因为有一个困难将在下面的*实现*部分讨论。

Design Measurement Noise

合理的假设是距离和方位测量噪声是独立的，因此

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

Implementation

在我们开始编码之前，我们还有另一个问题要处理。残差为y=z−h(x)。假设z为1◦， h(x)为359◦。减去它们得到−358◦。这将影响卡尔曼增益的计算，因为正确的角度差是2◦。因此，我们必须编写代码来正确计算轴承残差。

```
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))))
```

1.9999999999999774

状态向量的方位在索引2处，但测量向量的方位在索引1处，因此我们需要编写函数来处理它们。我们面临的另一个问题是，随着机器人机动，将看到不同的地标，因此我们需要处理可变数量的测量。测量中的残差函数将传递多个测量值的数组，每个地标一个。

```
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
```

我们现在可以实现度量模型了。方程是

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

表达式$\tan^{-1}(\frac{p_y - y}{p_x - x}) - \theta$ 可以产生超出范围$[-\pi, \pi)$的结果，因此我们应该将角度归一化到该范围

```
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)
```

我们的困难还没有结束。unscented变换计算状态和测量向量的平均值，但每个向量都包含一个方位。没有唯一的方法可以计算一组角度的平均值。例如，359◦和3◦的平均值是多少?直觉告诉我们答案应该是1◦，但是如果用$\frac{1}{n}\Sigmax$来计算，结果是181◦
一种常见的方法是对sin和cos的和求arctan。

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

`UnscentedKalmanFilter.__init__()`有一个参数`x_mean_fn`，用于计算状态的均值，`z_mean_fn`用于计算测量均值。我们将这些函数编码为:

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

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

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

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

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

这些函数利用了NumPy的三角函数对数组进行操作的事实，`dot`执行元素乘法。NumPy是用C和Fortran实现的，所以`sum(dot(sin(x), w))`比用Python编写等效的循环要快得多。

完成这些之后，我们现在准备实现UKF。我想指出的是，当我设计这个滤波器时，我并不是从头开始一次性设计上述所有功能。我把一个基本的UKF与预定义的地标，验证它的工作，然后开始填充碎片。“如果我看到不同的地标呢?”这让我改变了测量函数，让它接受一个地标数组。“如何计算角度的残差?”这促使我编写了角度归一化代码。“一组角度的 *均值* 是什么?”我在互联网上搜索，在维基百科上找到一篇文章，并实现了该算法。不要被吓倒。设计你能设计的，然后提出问题并一个接一个地解决它们。

读者已经看到了UKF的实现，所以我不会详细描述它。这里有两个新东西。当我们构建sigma点和滤波器时，我们必须为它提供我们编写的计算残差和均值的函数。

```
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)
```

接下来我们需要传递额外的数据给我们的`f(x, dt)`和`h(x)`。我们想要在`f(x, dt)`中使用move(x, dt, u, wheelbase)， 在`h(x)`中使用`Hx(x, landmarks`。我们可以这样做，只需要以关键字传递额外的参数给`predict()`和`update()`像下面这样：

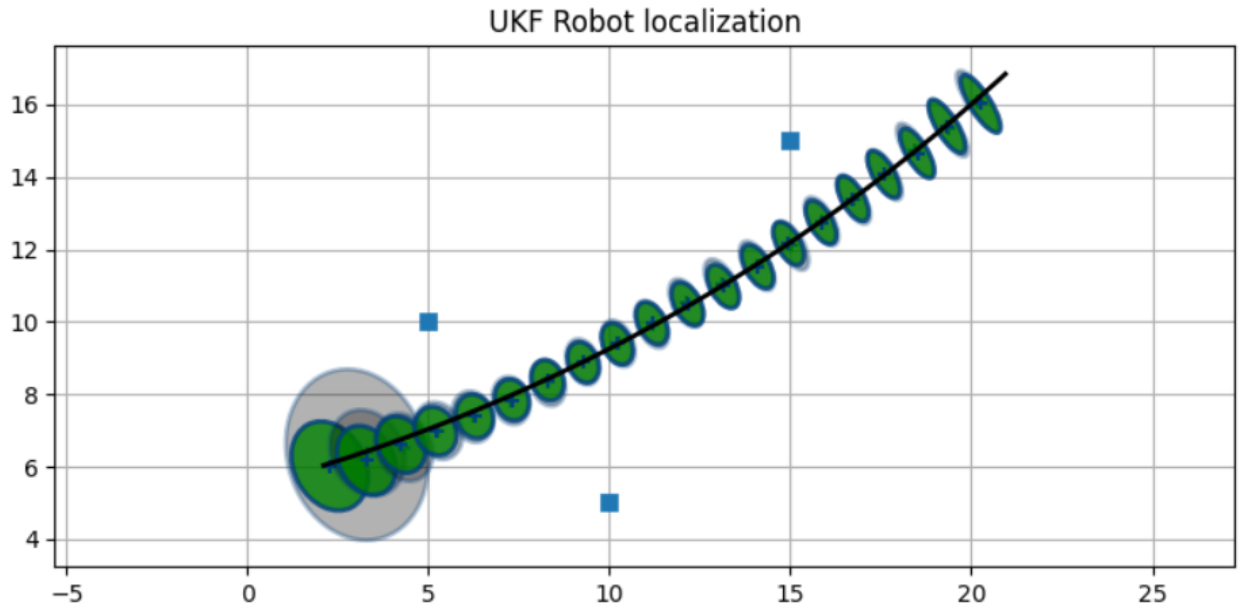
```
ukf.predict(u=u, wheelbase=wheelbase)
            ukf.update(z, landmarks=landmarks)
```

其余的代码运行模拟并绘制结果。我创建了一个包含地标坐标的变量`landmarks`。我每秒更新模拟机器人位置10次，但每秒只运行一次UKF。我们没有使用Runge Kutta来整合运动微分方程，因此小的时间步长使模拟更准确。

```
from filterpy.stats import plot_covariance_ellipse

dt = 1.0
wheelbase = 0.5

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

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

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

    sim_pos = ukf.x.copy()
```

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

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

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

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

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

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

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



Final P: [0.0085 0.0177 0.0006]

其余的代码运行模拟并绘制结果。我创建了一个包含地标坐标的变量`landmarks`。我每秒更新模拟机器人位置10次，但只运行一次UKF。这有两个原因。首先，我们没有使用龙格库塔来整合运动微分方程，因此较窄的时间步长可以使我们的模拟更加准确。其次，在嵌入式系统中，处理速度有限是很常见的。这迫使您只在绝对需要时频繁地运行卡尔曼滤波器。

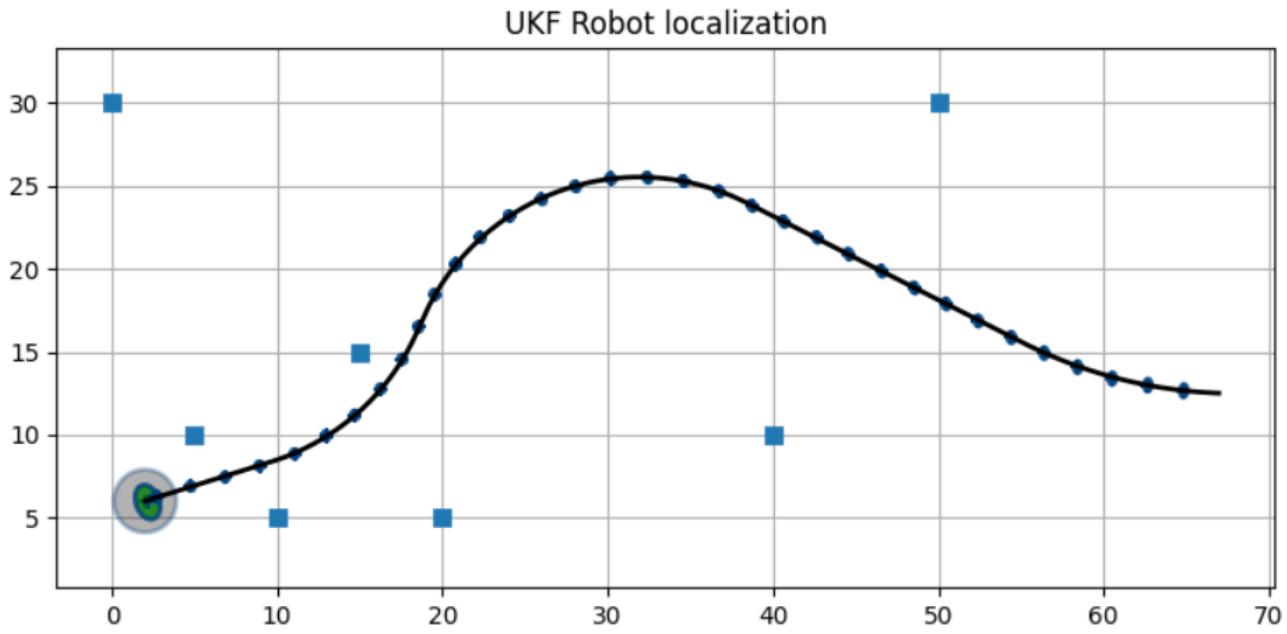
Steering the Robot

上面运行中的转向模拟是不现实的。速度和转向角从未改变，这对卡尔曼滤波没有太大的问题。我们可以实现一个复杂的PID控制机器人仿真，但我将使用NumPy的`linspace`方法生成不同的转向命令。我还将添加更多的地标，因为机器人将比第一个示例中移动得更远。

```
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)
```

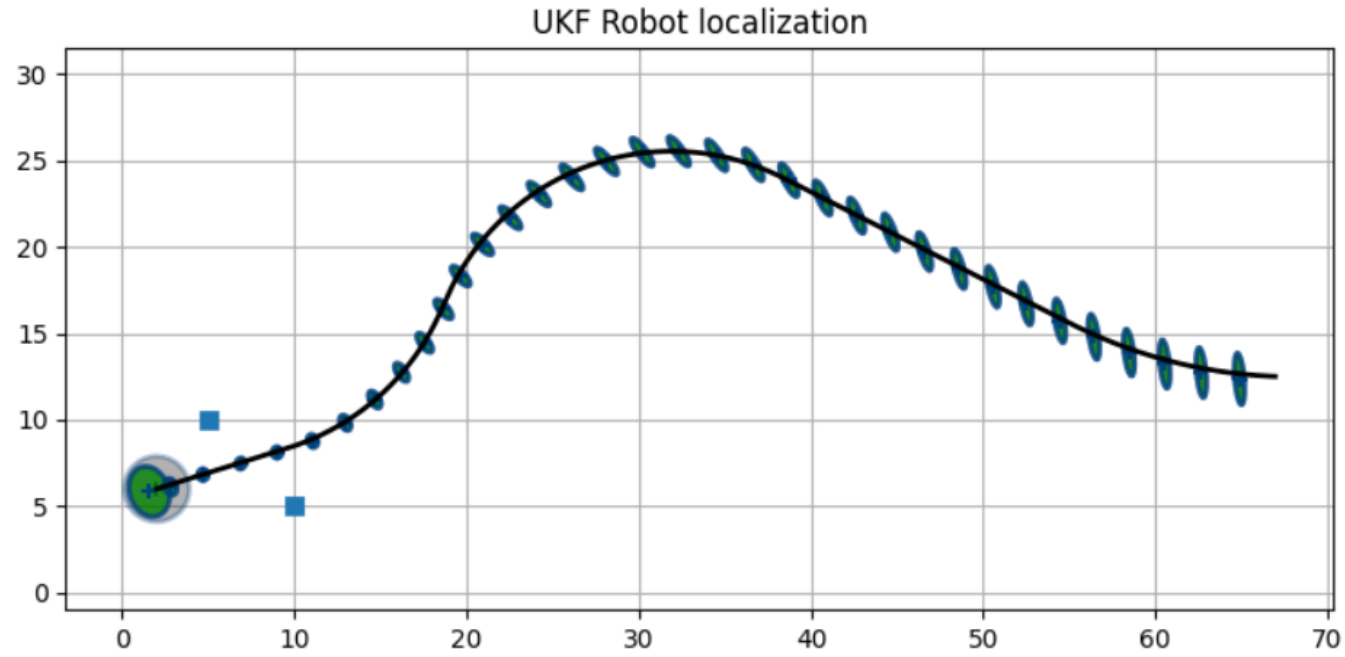
```
ukf = run_localization(
    cmds, landmarks, sigma_vel=0.1, sigma_steer=np.radians(1),
    sigma_range=0.3, sigma_bearing=0.1, step=1,
    ellipse_step=20)
print('final covariance', ukf.P.diagonal())
```



final covariance [0.0012 0.0042 0.0003]

不确定性很快就会变得非常小。协方差椭圆显示6σ协方差，但椭圆太小了，很难看到。我们可以通过仅在起始点附近提供两个地标来将更多误差合并到答案中。当我们运行这个滤波器时，随着机器人远离这些地标，误差会增加。

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



final covariance [0.0025 0.0654 0.0007]

Discussion

你对本章的印象可能取决于你过去实现过多少个非线性卡尔曼滤波器。如果这是你第一次接触，那么2n+1 sigma点的计算以及随后f(x)和h(x)函数的编写可能会让你觉得有点挑剔。事实上，由于需要处理角度的模数学，我花了很多时间来让一切正常运行。另一方面，如果你已经实现了扩展卡尔曼滤波器(EKF)，也许你会高兴地在座位上跳来跳去。编写UKF的函数有一点乏味，但概念非常基本。同样的问题的EKF需要一些相当困难的数学。对于许多问题，我们无法找到EKF方程的闭式解，我们必须退回到某种迭代解。

与EKF相比，UKF的优点不仅是相对容易实现。现在讨论这个有点为时过早，因为你还没有学习EKF，但EKF在某一点上将问题线性化，并通过线性卡尔曼滤波器使该点线性化。相比之下，UKF需要2n+1样本。因此，UKF通常比EKF更准确，特别是当问题是高度非线性的时候。虽然不能保证UKF总是优于EKF，但在实践中，它的表现至少和EKF一样好，通常比EKF好得多。

因此，我的建议是始终从实现UKF开始。如果你的过滤器有现实世界的后果，如果它发散(人们死亡，大量的金钱损失，发电厂爆炸)，当然你将不得不进行复杂的分析和实验，以选择最好的过滤器。这超出了本书的范围，你应该去研究生院学习这个理论。

最后，我说UKF是执行sigma点滤波器的*方法*。这不是真的。我选择的具体版本是由Van der Merwe在他2004年的论文中参数化的Julier的scaled unscented滤波器。如果你搜索Julier, Van der Merwe, Uhlmann和Wan，你会发现他们开发的一系列类似的sigma点滤波器。每种技术使用不同的方式选择和加权sigma点。但选择还不止于此。例如，SVD卡尔曼滤波器使用奇异值分解(SVD)来找到概率分布的近似均值和协方差。本章将介绍sigma点滤波器，而不是详细介绍它们的工作原理。

References

- Rudolph Van der Merwe. "Sigma-Point Kalman Filters for Probabilistic Inference in Dynamic State-Space Models" dissertation (2004).
- 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
- 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)
- Cholesky decomposition. Wikipedia. http://en.wikipedia.org/wiki/Cholesky_decomposition