无人驾驶汽车系统入门（二）——高级运动模型和扩展卡尔曼滤波

无人驾驶汽车系统入门（二）——高级运动模型和扩展卡尔曼滤波

说明：

介绍高级运动模型和扩展卡尔曼滤波

前言：

上一篇文章的最后我们提到卡尔曼滤波存在着一个非常大的局限性——它仅能对线性的处理模型和测量模型进行精确的估计，在非线性的场景中并不能达到最优的估计效果。

所以之前为了保证我们的处理模型是线性的，我们上一节中使用了恒定速度模型，然后将估计目标的加减速用处理噪声来表示，这一模型用来估算行人的状态其实已经足够了

但是在现实的驾驶环境中，我们不仅要估计行人，我们除了估计行人状态以外，我们还需要估计其他车辆，自行车等等状态，他们的状态估计显然不能使用简单的线性系统来描述

这里，我们介绍非线性系统情况下的一种广泛使用的滤波算法——扩展卡尔曼滤波(Extended Kalman Filter, EKF)

本节讲解非线性系统中广泛使用的扩展卡尔曼滤波算法，我们通常将该算法应用于实际的车辆状态估计（或者说车辆追踪）中。

另外，实际的车辆追踪运动模型显然不能使用简单的恒定速度模型来建模

在本节中会介绍几种应用于车辆追踪的高级运动模型。

并且已经其中的CTRV模型来构造我们的扩展卡尔曼滤波。

最后，在代码实例中，我会介绍如何使用EKF做多传感器融合。

应用于车辆追踪的高级运动模型

首先要明确的一点是，不管是什么运动模型，本质上都是为了帮助我们简化问题，所以我们可以根据运动模型的复杂程度（次数）来给我们常用的运动模型分一下类。

1. 一次运动模型（也别称为线性运动模型）：

   恒定速度模型（Constant Velocity, CV）

   恒定加速度模型（Constant Acceleration, CA）

   这些线性运动模型假定目标是直线运动的，并不考虑物体的转弯。

2. 二次运动模型：

   恒定转率和速度模型（Constant Turn Rate and Velocity，CTRV）

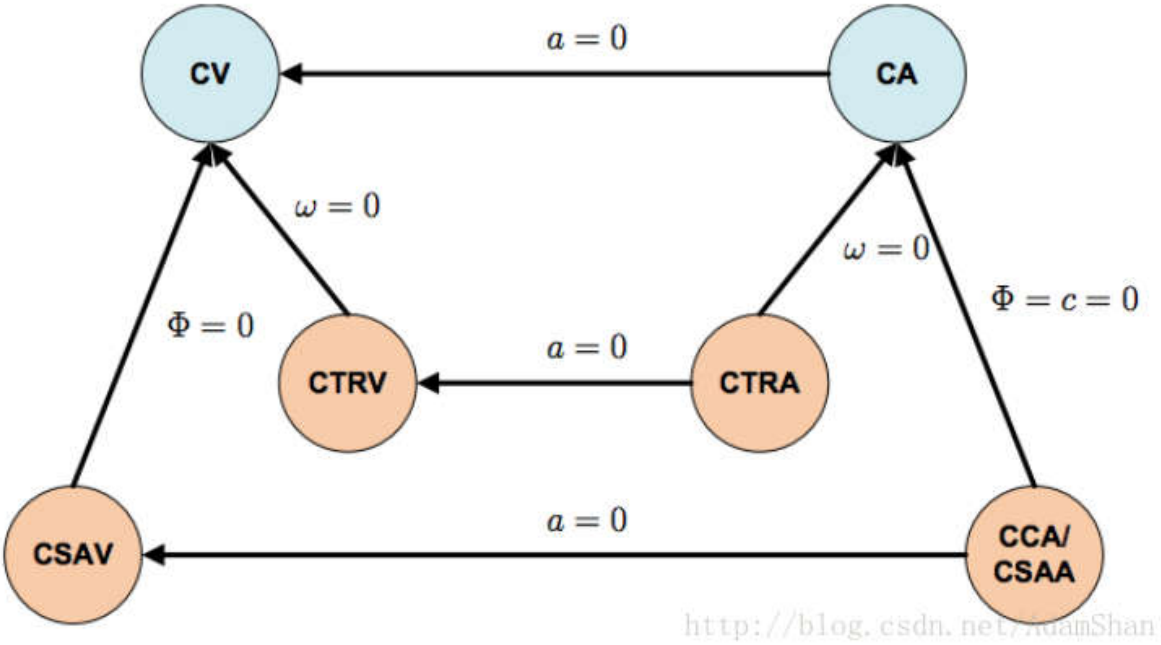
   恒定转率和加速度模型（Constant Turn Rate and Acceleration，CTRA）

3. CTRV

CTRV目前多用于机载追踪系统（飞机），这些二次运动模型大多假定速度 vv 和 偏航角速度（yaw rate） ωω 没有关系，因此，在这类运动模型中，由于偏航角速度测量的扰动（不稳定），即使车辆没有移动，我们的运动模型下的角速度也会发生细微的变化。

为了解决这个问题，速度 vv 和 偏航角速度 ωω 的关联可以通过设定转向角 ΦΦ 恒定来建立，这样就引出了 恒定转向角和速度模型（Constant Steering Angle and Velocity，CSAV），另外，速度可以别假定为线性变化的，进而引出了常曲率和加速度模型（Constant Curvature and Acceleration，CCA）。

这些运动模型的关系如图：



运动模型的状态转移公式

由于除CCA以外，以上的运动模型都非常著名，故本文不提供详细的推导过程。本文提供CV和CTRV模型的状态转移公式。

状态转移公式：就是我们的处理模型由上一状态的估计计算下一个状态的先验分布的计算公式，可以理解为我们基于一定的先验

知识总结出来的运动公式。

1. **CV模型：**
   CV模型的状态空间可以表示为：

$$\vec{x}(t) = (x \quad y \quad v_x \quad v_y)^T$$

那么转移函数为：

$$\vec{x}(t+\Delta t) = \begin{pmatrix} x(t) + \Delta t v_x \\ y(t) + \Delta t v_y \\ v_x \\ v_y \end{pmatrix}$$

2. **CTRV模型：**

在CTRV中，目标的状态量为：

$$\overrightarrow{x}(t) = (x \quad y \quad v \quad \theta \quad \omega)^T$$

其中，θ 为偏航角，是追踪的目标车辆在当前车辆坐标系下与x轴的夹角，逆时针方向为正，取值范围是$[0, 2\pi)$，ω是偏航角速度。CTRV的状态转移函数为：

$$\overrightarrow{x}(t+\Delta t) = \begin{pmatrix} \frac{v}{\omega}\sin(\omega\Delta t+\theta) - \frac{v}{\omega}\sin(\theta) + x(t) \\ -\frac{v}{\omega}\cos(\omega\Delta t+\theta) + \frac{v}{\omega}\cos(\theta) + y(t) \\ v \\ \omega\Delta t+\theta \\ \omega \end{pmatrix}$$

本文下面的内容将以CTRV模型作为我们的运动模型。使用CTRV还存在一个问题，那就是 $\omega = 0$ 的情况，此时我们的状态转移函数公式中的 (x, y) 将变成无穷大。为了解决这个问题，我们考察一下$\omega = 0$ 的情况，此时我们追踪的车辆实际上是直线行驶的，所以我们的 (x, y) 的计算公式就变成了：

$$x(t+\Delta t) = v\cos(\theta)\Delta t + x(t)$$

$$y(t+\Delta t) = v\sin(\theta)\Delta t + y(t)$$

那么现在问题来了，我们知道，卡尔曼滤波仅仅用于处理线性问题，那么很显然我们现在的处理模型是非线性的，这个时候我们就不能简单使用卡尔曼滤波进行预测和更新了，此时预测的第一步变成了如下非线性函数：

$$x_{k+1} = g(x_k, u)$$

其中，$g()$ 表示CTRV运动模型的状态转移函数，u 表示处理噪声。为了解决非线性系统下的问题，我们引入**扩展卡尔曼滤波（Extended Kalman Filter，EKF）**

扩展卡尔曼滤波

1. 雅可比矩阵

扩展卡尔曼滤波使用线性变换来近似非线性线性变换，具体来说，EKF使用一阶泰勒展式来进行线性化:

$$h(x) \approx h(u) + \frac{\partial h(u)}{\partial x}(x-u)$$

> 数学中，泰勒公式是一个用函数在某点的信息描述其附近取值的公式。如果函数足够平滑的话，在已知函数在某一点的各阶导数值的情况之下，泰勒公式可以用这些导数值做系数构建一个多项式来近似函数在这一点的邻域中的值。泰勒公式还给出了这个多项式和实际的函数值之间的偏差。

回到我们的处理模型中，我们的状态转移函数为：

$$\overrightarrow{x}(t+\Delta t)=g(x(t))=\begin{pmatrix}\frac{v}{\omega}\sin(\omega\Delta t+\theta)-\frac{v}{\omega}\sin(\theta)+x(t)\\-\frac{v}{\omega}\cos(\omega\Delta t+\theta)+\frac{v}{\omega}\cos(\theta)+y(t)\\v\\\omega\Delta t+\theta\\\omega\end{pmatrix},\quad \omega\neq 0$$

$$\overrightarrow{x}(t+\Delta t)=g(x(t))=\begin{pmatrix}v\cos(\theta)\Delta t+x(t)\\v\sin(\theta)\Delta t+y(t)\\v\\\omega\Delta t+\theta\\\omega\end{pmatrix},\quad \omega=0$$

那么，对于这个多元函数，我们需要使用多元泰勒级数：

$$T(x)=f(u)+(x-u)Df(u)+\frac{1}{2!}(x-u)^2D^2f(u)+\dots$$

其中，$Df(a)$叫雅可比矩阵，它是多元函数中各个因变量关于各个自变量的一阶偏导数构成的矩阵。

$$\mathbf{J}=\begin{bmatrix}\dfrac{\partial \mathbf{f}}{\partial x_1} & \cdots & \dfrac{\partial \mathbf{f}}{\partial x_n}\end{bmatrix}=\begin{bmatrix}\dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_n}\\\vdots & \ddots & \vdots\\\dfrac{\partial f_m}{\partial x_1} & \cdots & \dfrac{\partial f_m}{\partial x_n}\end{bmatrix}$$

> 在向量微积分中，雅可比矩阵是一阶偏导数以一定方式排列成的矩阵，其行列式称为雅可比行列式。雅可比矩阵的重要性在于它体现了一个可微方程与给出点的最优线性逼近。因此，雅可比矩阵类似于多元函数的导数。

在扩展卡尔曼滤波中，由于$(x-u)$本身数值很小，那么$(x-u)$就更小了，所以更高阶的级数在此问题中忽略不计，我们只考虑到利用雅可比矩阵进行线性化。

那么接下来就是求解雅可比矩阵，在CTRV模型中，对各个元素求偏导数可以得到雅可比矩阵（$\omega \neq 0$）：

$$
\begin{aligned}
&J_A \\
&= \\
&\begin{bmatrix}
1 & 0 & \frac{1}{\omega}(-\sin(\theta)+\sin(\Delta t\omega+\theta)) & \frac{v}{\omega}(-\cos(\theta)+\cos(\Delta t\omega+\theta)) & \frac{\Delta tv}{\omega}\cos(\Delta t\omega+\theta)-\frac{v}{\omega^2}(-\sin(\theta)+\sin(\Delta t\omega+\theta)) \\
0 & 1 & \frac{v}{\omega}(-\sin(\theta)+\sin(\Delta t\omega+\theta)) & \frac{1}{\omega}(\cos(\theta)-\cos(\Delta t\omega+\theta)) & \frac{\Delta tv}{\omega}\sin(\Delta t\omega+\theta)-\frac{v}{\omega^2}(\cos(\theta)-\cos(\Delta t\omega+\theta)) \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & \Delta t \\
0 & 0 & 0 & 0 & 1
\end{bmatrix}
\end{aligned}
$$

当 $\omega = 0$ 时，雅可比矩阵为：

$$
J_A = \begin{bmatrix}
1 & 0 & \Delta t\cos(\theta) & -\Delta tv\sin(\theta) & 0 \\
0 & 1 & \Delta t\sin(\theta) & \Delta tv\cos(\theta) & 0 \\
0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & \Delta t \\
0 & 0 & 0 & 0 & 1
\end{bmatrix}
$$

在我们后面的Python实现中，我们将使用numdifftools包直接计算雅可比矩阵，而不需要我们使用代码写这个雅可比矩阵。在得到我们CTRV模型的雅可比矩阵以后，我们的处理模型就可以写成：

$$
\begin{aligned}
x_{k+1} &= g(x_k, u) \\
P_{k+1} &= J_A P_k J_A^T + Q
\end{aligned}
$$

2. 处理噪声

处理噪声模拟了运动模型中的扰动，我们引入运动模型的出发点就是要简化我们要处理的问题，这个简化是建立在多个假设的基础上（在CTRV中，这些假设就是恒定偏航角速度和速度），但是在现实问题中这些假设就会带来一定的误差，处理噪声实际上描述了当我们的系统在运行一个指定时间段以后可能面临多大的这样的误差。在CTRV模型中噪声的引入主要来源于两处：直线加速度和偏航角加速度噪声，我们假定直线加速度和偏航角加速度满足均值为 0，方差分别为 σ_a^2，$\sigma_{\dot{\omega}}^2$ 的高斯分布，由于均值为 0, 我们在状态转移公式中的 u就可以不予考虑，我们来看噪声带来的不确定性 Q，直线加速度和偏航角加速度将影响我们的状态量$(x, y, v, \theta, \omega)$，这两个加速度量对我们的状态的影响的表达式如下：

$$
noise\_term = \begin{bmatrix}
\frac{1}{2}\Delta t^2\mu_a\cos(\theta) \\
\frac{1}{2}\Delta t^2\mu_a\sin(\theta) \\
\Delta t\mu_a \\
\frac{1}{2}\Delta t^2\mu_{\dot{\omega}} \\
\Delta t\mu_{\dot{\omega}}
\end{bmatrix}
$$

其中 $\mu_a, \mu_{\dot{\omega}}$ 为直线上和转角上的加速度（在这个模型中，我们把把它们看作处理噪声），我们分解这个矩阵：

$$noise\_term = \begin{bmatrix} \frac{1}{2}\Delta t^2\cos(\theta) & 0 \\ \frac{1}{2}\Delta t^2\sin(\theta) & 0 \\ \Delta t & 0 \\ 0 & \frac{1}{2}\Delta t^2 \\ 0 & \Delta t \end{bmatrix} \cdot \begin{bmatrix} \mu_a \\ \mu_{\dot{\omega}} \end{bmatrix} = G \cdot \mu$$

我们知道 Q 就是处理噪声的协方差矩阵，其表达式为：

$$Q = E[noise\_term \cdot noise\_term^T] = E[G\mu\mu^T G^T] = G \cdot E[\mu\mu^T] \cdot G^T$$

其中：

$$E[\mu\mu^T] = \begin{pmatrix} \sigma_a^2 & 0 \\ 0 & \sigma_{\dot{\omega}}^2 \end{pmatrix}$$

所以，我们在CTRV模型中的处理噪声的协方差矩阵 Q 的计算公式就是：

$$Q = \begin{bmatrix} (\frac{1}{2}\Delta t^2\sigma_a\cos(\theta))^2 & \frac{1}{4}\Delta t^4\sigma_a^2\sin(\theta)\cos(\theta) & \frac{1}{2}\Delta t^3\sigma_a^2\cos(\theta) & 0 & 0 \\ \frac{1}{4}\Delta t^4\sigma_a^2\sin(\theta)\cos(\theta) & (\frac{1}{2}\Delta t^2\sigma_a\sin(\theta))^2 & \frac{1}{2}\Delta t^3\sigma_a^2\sin(\theta) & 0 & 0 \\ \frac{1}{2}\Delta t^3\sigma_a^2\cos(\theta) & \frac{1}{2}\Delta t^3\sigma_a^2\sin(\theta) & \Delta t^2\sigma_a^2 & 0 & 0 \\ 0 & 0 & 0 & (\frac{1}{2}\Delta t^2\sigma_{\dot{\omega}})^2 & \frac{1}{2}\Delta t^3\sigma_{\dot{\omega}}^2 \\ 0 & 0 & 0 & \frac{1}{2}\Delta t^3\sigma_{\dot{\omega}} & \Delta t^2\sigma_{\dot{\omega}} \end{bmatrix}$$

3. 测量

假设我们有激光雷达和雷达两个传感器，它们分别以一定的频率来测量如下数据：

- 激光雷达：测量目标车辆的坐标 (x, y) 。这里的x,y是相对于我们的车辆的坐标系的，即我们的车辆为坐标系的原点，我们的车头为x轴，车的左侧为y轴。
- 雷达：测量目标车辆在我们车辆坐标系下与本车的距离 ρ，目标车辆与x轴的夹角 ψ，以及目标车辆与我们自己的相对距离变化率 $\dot{\rho}$（本质上就是目标车辆的实际速度在我们和目标车辆的连线上的分量）

前面的卡尔曼滤波器中，我们使用一个测量矩阵 H 将预测的结果映射到测量空间，那是因为这个映射本身就是线性的，现在，我们使用雷达和激光雷达来测量目标车辆(我们把这个过程称为传感器融合)，这个时候会有两种情况，即：

1. 激光雷达的测量模型仍然是线性的，其测量矩阵为：

$$H_L = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

将预测映射到激光雷达测量空间：

$$H_L \overrightarrow{x} = (x, y)^T$$

2. 雷达的预测映射到测量空间是非线性的，其表达式为：

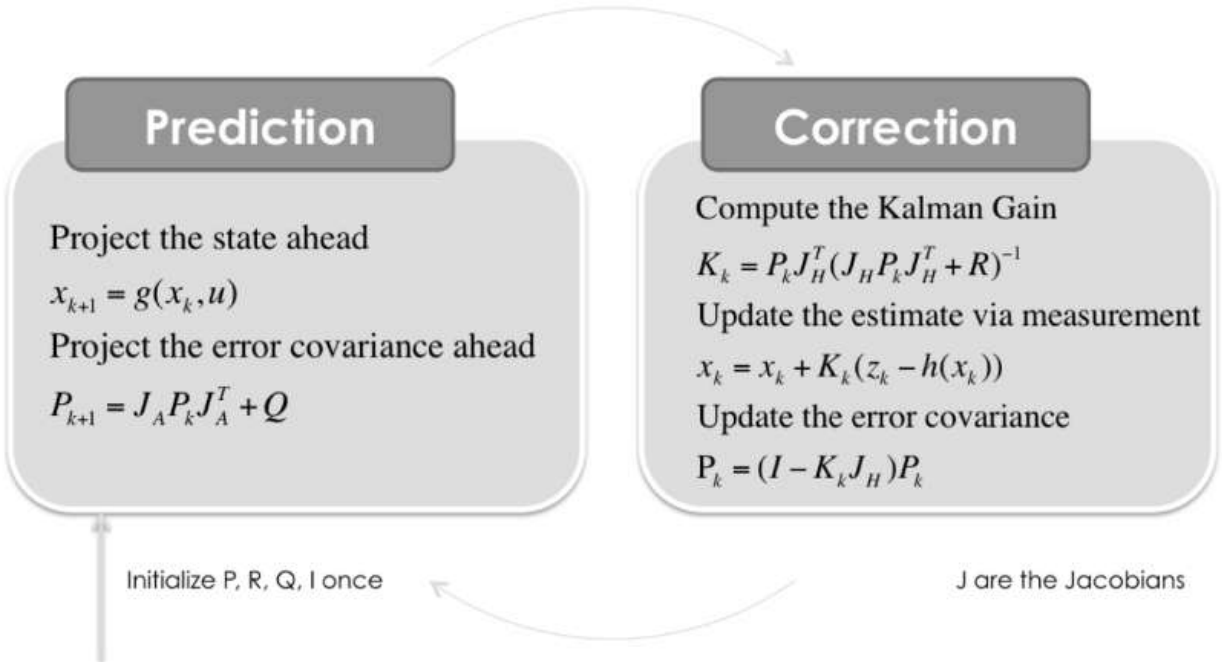
$$\begin{pmatrix} \rho \\ \psi \\ \dot{\rho} \end{pmatrix} = \begin{pmatrix} \sqrt{x^2+y^2} \\ \text{atan}_2(y, x) \\ \frac{vx+vy}{\sqrt{x^2+y^2}} \end{pmatrix}$$

此时我们使用 $h(x)$ 来表示这样一个非线性映射，那么在求解卡尔曼增益时我们也要将该非线性过程使用泰勒公式将其线性化，参照预测过程，我们也只要求解 $h(x)$的雅可比矩阵：

$$J_H = \begin{bmatrix} \frac{x}{\sqrt{x^2+y^2}} & \frac{y}{\sqrt{x^2+y^2}} & 0 & 0 & 0 \\ -\frac{y}{x^2+y^2} & \frac{x}{x^2+y^2} & 0 & 0 & 0 \\ \frac{v}{\sqrt{x^2+y^2}} - \frac{x(vx+vy)}{(x^2+y^2)^{\frac{3}{2}}} & \frac{v}{\sqrt{x^2+y^2}} - \frac{y(vx+vy)}{(x^2+y^2)^{\frac{3}{2}}} & \frac{x+y}{\sqrt{x^2+y^2}} & 0 & 0 \end{bmatrix}$$

虽然这个雅可比矩阵看似非常复杂，但是我们待会编程的时候并不需要完整的推导出这个雅可比矩阵的表达式，在本篇中，我们采用numdifftools这个公式来直接求解雅可比矩阵。

综上，EKF的整个过程为：



4.Python实现

和之前一样，为了实现交互式的代码，实例的代码为了便于理解，我们仍然使用大家熟悉的Python来实现

当然，实际无人车项目肯定是需要用C++来实现的，要将下面的示例代码使用C++来改写是非常简单快速的。

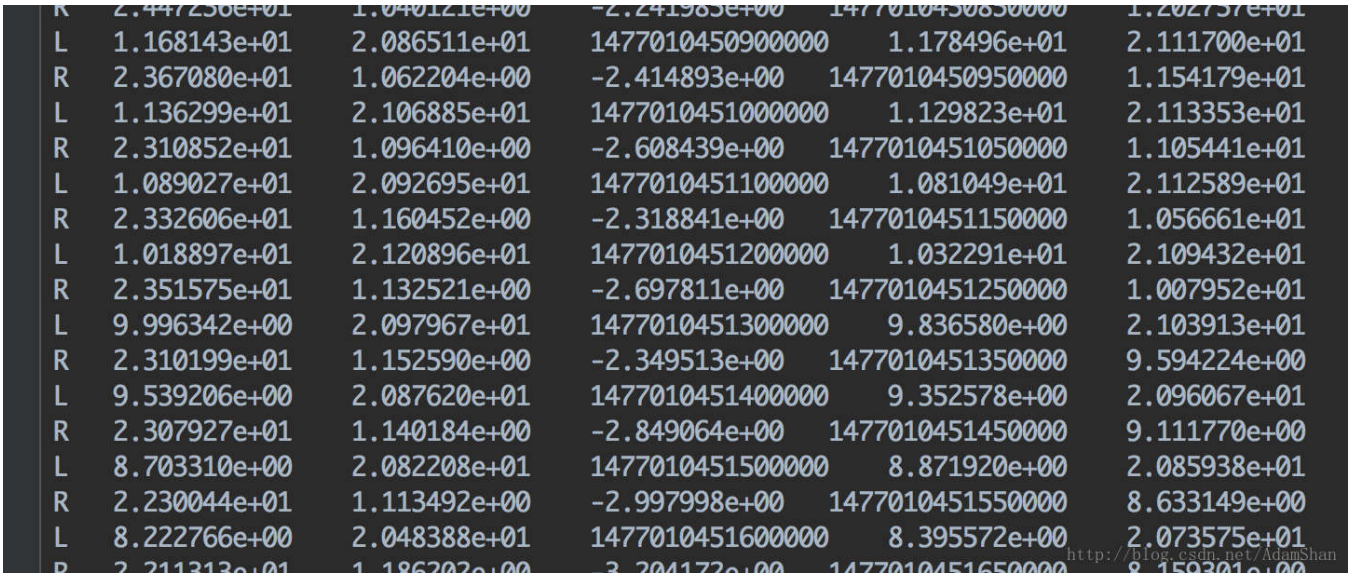
首先引入相关的库：

```
from __future__ import print_function
import numpy as np
import matplotlib.dates as mdates
import matplotlib.pyplot as plt
```

```
from scipy.stats import norm
from sympy import Symbol, symbols, Matrix, sin, cos, sqrt, atan2
from sympy import init_printing
init_printing(use_latex=True)
import numdifftools as nd
import math
```

首先我们读取我们的数据集，该数据集包含了追踪目标的LIDAR和RADAR的测量值，以及测量的时间点

同时为了验证我们追踪目标的精度，该数据还包含了追踪目标的真实坐标。（数据下载链接见文章末尾）



```
L   1.168143e+01    2.086511e+01    1477010450900000    1.178496e+01    2.111700e+01
R   2.367080e+01    1.062204e+00    -2.414893e+00   1477010450950000    1.154179e+01
L   1.136299e+01    2.106885e+01    1477010451000000    1.129823e+01    2.113353e+01
R   2.310852e+01    1.096410e+00    -2.608439e+00   1477010451050000    1.105441e+01
L   1.089027e+01    2.092695e+01    1477010451100000    1.081049e+01    2.112589e+01
R   2.332606e+01    1.160452e+00    -2.318841e+00   1477010451150000    1.056661e+01
L   1.018897e+01    2.120896e+01    1477010451200000    1.032291e+01    2.109432e+01
R   2.351575e+01    1.132521e+00    -2.697811e+00   1477010451250000    1.007952e+01
L   9.996342e+00    2.097967e+01    1477010451300000    9.836580e+00    2.103913e+01
R   2.310199e+01    1.152590e+00    -2.349513e+00   1477010451350000    9.594224e+00
L   9.539206e+00    2.087620e+01    1477010451400000    9.352578e+00    2.096067e+01
R   2.307927e+01    1.140184e+00    -2.849064e+00   1477010451450000    9.111770e+00
L   8.703310e+00    2.082208e+01    1477010451500000    8.871920e+00    2.085938e+01
R   2.230044e+01    1.113492e+00    -2.997998e+00   1477010451550000    8.633149e+00
L   8.222766e+00    2.048388e+01    1477010451600000    8.395572e+00    2.073575e+01
```


其中第一列表示测量数据来自LIDAR还是RADAR，LIDAR的2，3列表示测量的目标 (x,y)(x,y),第4列表示测量的时间点，第5，6，7，8表示真实的(x,y,vx,vy)(x,y,vx,vy), RADAR测量的(前三列)是$(\rho,\psi,\dot{\rho})(\rho,\psi,\dot{\rho})$，其余列的数据的意义和LIDAR一样。

首先我们读取整个数据：

```
dataset = []

# read the measurement data, use 0.0 to stand LIDAR data
# and 1.0 stand RADAR data
with open('data_synthetic.txt', 'rb') as f:
    lines = f.readlines()
    for line in lines:
        line = line.strip('\n')
        line = line.strip()
        numbers = line.split()
        result = []
        for i, item in enumerate(numbers):
            item.strip()
            if i == 0:
                if item == 'L':
                    result.append(0.0)
                else:
                    result.append(1.0)
```

```
            else:
```

初始化P，激光雷达的测量矩阵（线性）H_L，测量噪声R, 处理噪声的中直线加速度项的标准差 σ_a，转角加速度项的标准差$\sigma_{\ddot{\omega}}$：

```
P = np.diag([1.0, 1.0, 1.0, 1.0, 1.0])
print(P, P.shape)
H_lidar = np.array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.]])
print(H_lidar, H_lidar.shape)

R_lidar = np.array([[0.0225, 0.],[0., 0.0225]])
R_radar = np.array([[0.09, 0., 0.],[0., 0.0009, 0.], [0., 0., 0.09]])
print(R_lidar, R_lidar.shape)
print(R_radar, R_radar.shape)

# process noise standard deviation for a
std_noise_a = 2.0
# process noise standard deviation for yaw acceleration
std_noise_yaw_dd = 0.3
```

在整个预测和测量更新过程中，所有角度量的数值都应该控制在 [–π,π]，我们知道角度加减 2π 不变，所以用如下函数表示函数来调整角度：

```
def control_psi(psi):
    while (psi > np.pi or psi < -np.pi):
        if psi > np.pi:
            psi = psi - 2 * np.pi
        if psi < -np.pi:
            psi = psi + 2 * np.pi
    return psi
```

使用第一个雷达（或者激光雷达）的测量数据初始化我们的状态，对于激光雷达数据，可以直接将测量到的目标的 (x, y) 坐标作为初始 (x, y)，其余状态项初始化为0，对于雷达数据，可以使用如下公式由测量的 $\rho, \psi, \dot{\rho}$ 得到目标的坐标 (x, y)：

$$x = \rho \times \cos(\psi)$$

$$y = \rho \times \sin(\psi)$$

具体状态初始化代码为：

```
state = np.zeros(5)
init_measurement = dataset[0]
current_time = 0.0
if init_measurement[0] == 0.0:
    print('Initialize with LIDAR measurement!')
    current_time = init_measurement[3]
    state[0] = init_measurement[1]
    state[1] = init_measurement[2]
```

```
else:
    print('Initialize with RADAR measurement!')
    current_time = init_measurement[4]
    init_rho = init_measurement[1]
    init_psi = init_measurement[2]
    init_psi = control_psi(init_psi)
    state[0] = init_rho * np.cos(init_psi)
    state[1] = init_rho * np.sin(init_psi)
print(state, state.shape)
```

写一个辅助函数用于保存数值：

```
# Preallocation for Saving
px = []
py = []
vx = []
vy = []

gpx = []
gpy = []
gvx = []
gvy = []

mx = []
my = []

def savestates(ss, gx, gy, gv1, gv2, m1, m2):
    px.append(ss[0])
    py.append(ss[1])
    vx.append(np.cos(ss[3]) * ss[2])
    vy.append(np.sin(ss[3]) * ss[2])
```

定义状态转移函数和测量函数，使用numdifftools库来计算其对应的雅可比矩阵，

这里我们先设 Δt=0.05，只是为了占一个位置，当实际运行EKF时会计算出前后两次测量的时间差，一次来替换这里的 Δt。

```
measurement_step = len(dataset)
state = state.reshape([5, 1])
dt = 0.05

I = np.eye(5)

transition_function = lambda y: np.vstack((
    y[0] + (y[2] / y[4]) * (np.sin(y[3] + y[4] * dt) - np.sin(y[3])),
    y[1] + (y[2] / y[4]) * (-np.cos(y[3] + y[4] * dt) + np.cos(y[3])),
    y[2],
    y[3] + y[4] * dt,
    y[4]))

# when omega is 0
```

```
transition_function_1 = lambda m: np.vstack((m[0] + m[2] * np.cos(m[3]) * dt,
                                             m[1] + m[2] * np.sin(m[3]) * dt,
                                             m[2],
                                             m[3] + m[4] * dt,
                                             m[4]))
```

EKF的过程代码：

```
for step in range(1, measurement_step):

    # Prediction
    # ========================
    t_measurement = dataset[step]
    if t_measurement[0] == 0.0:
        m_x = t_measurement[1]
        m_y = t_measurement[2]
        z = np.array([[m_x], [m_y]])

        dt = (t_measurement[3] - current_time) / 1000000.0
        current_time = t_measurement[3]

        # true position
        g_x = t_measurement[4]
        g_y = t_measurement[5]
        g_v_x = t_measurement[6]
        g_v_y = t_measurement[7]

    else:
```

这里有几点需要注意，首先，要考虑清楚有几个地方被除数有可能为 0，比如说ω=0，以及 ρ=0 的情况。

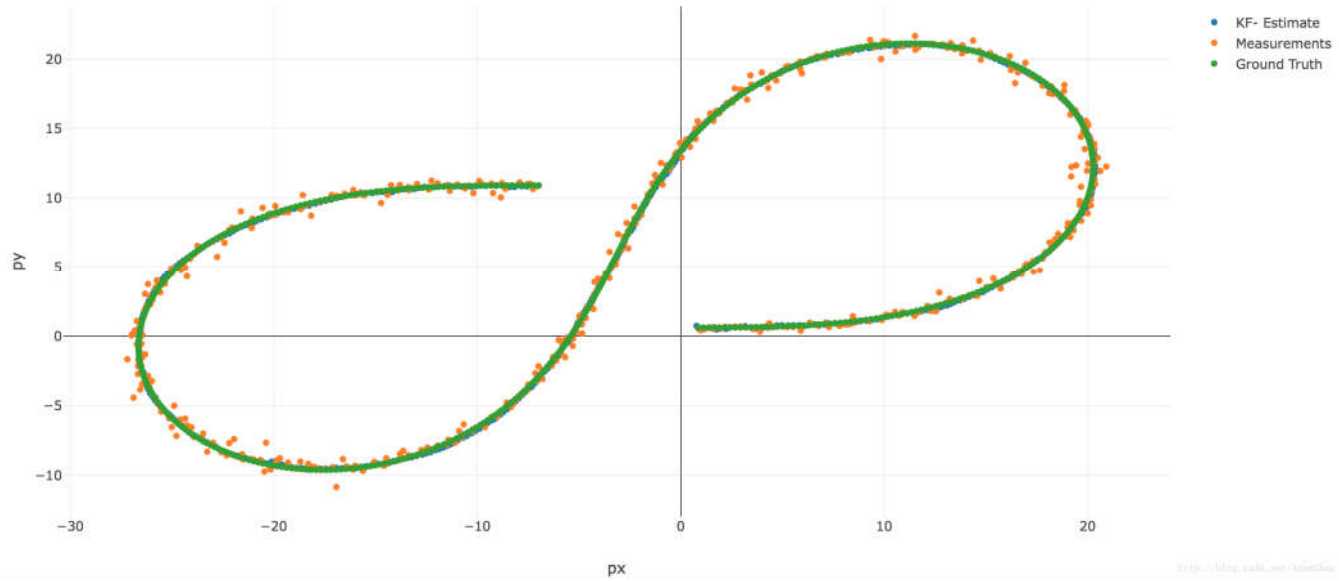
处理完以后我们输出估计的均方误差（RMSE），并且把各类数据保存以便我们可视化我们的EKF的效果：

```
def rmse(estimates, actual):
    result = np.sqrt(np.mean((estimates-actual)**2))
    return result

print(rmse(np.array(px), np.array(gpx)),
      rmse(np.array(py), np.array(gpy)),
      rmse(np.array(vx), np.array(gvx)),
      rmse(np.array(vy), np.array(gvy)))

# write to the output file
stack = [px, py, vx, vy, mx, my, gpx, gpy, gvx, gvy]
stack = np.array(stack)
stack = stack.T
np.savetxt('output.csv', stack, '%.6f')
```

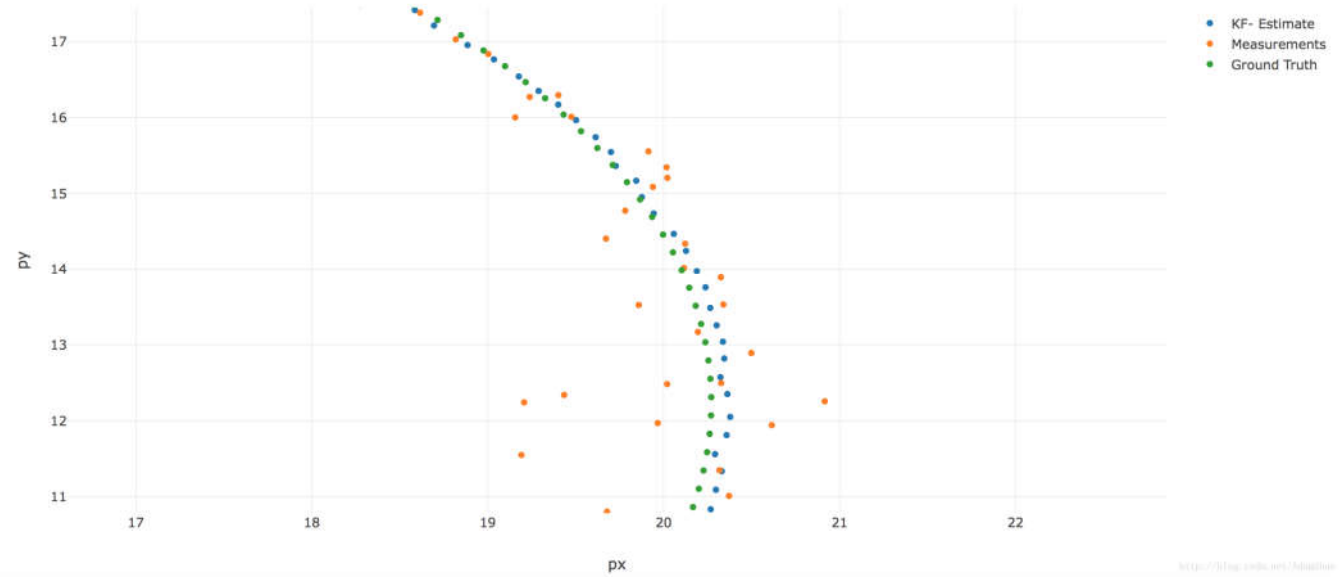
最后我们来看一下我们的EKF在追踪目标的时候的均方误差：

```
0.0736336090893 0.0804598933194 0.229165985264 0.309993887661
```

我们把我们的EKF的估计结果可视化：



我们放大一个局部看一下：



其中，蓝色的点为我们的EKF对目标位置的估计，橙色的点为来自LIDAR和RADAR的测量值，绿色的点是目标的真实位置

由此可知，我们的测量是不准确的，因此我们使用EKF在结合运动模型的先验知识以及融合两个传感器的测量的基础上做出了非常接近目标正是状态的估计。

EKF的魅力其实还不止在此！我们使用EKF，不仅能够对目标的状态（我们关心的）进行准确的估计，从这个例子我们会发现，EKF还能估计我们的传感器无法测量的量（比如本例中的$v,\psi,\dot{\psi}$）

那么，扩展卡尔曼滤波就到此结束了，大家可能会问，怎么没看到可视化结果那一部分的代码？哈哈，为了吸引一波人气，请大家关注我的博客，我会在下一期更新中（无损卡尔曼滤波）提供这一部分代码。

最后，细心的同学可能会发现，我们这个EKF执行的效率太低了，实际上，EKF的一个最大的问题就是求解雅可比矩阵计算量比较大，而且相关的偏导数推导也是非常无聊的工作，因此引出了我们下一篇要讲的内容，无损卡尔曼滤波（Unscented

Kalman Filter，UKF)