无人驾驶汽车系统入门（三）——无损卡尔曼滤波，目标追踪，C++

无人驾驶汽车系统入门（三）——无损卡尔曼滤波，目标追踪，C++

说明：

> 介绍前面两篇文章我们了解了卡尔曼滤波以及扩展卡尔曼滤波在目标追踪的应用，我们在上一篇文章中还具体用Python实现了EKF，但是细心的同学会发现，EKF的效率确实很低，计算雅可比矩阵确实是一个很费时的操作，当问题（非线性的）一旦变得复杂，其计算量就变得十分不可控制。在此再向大家接受一种滤波——无损卡尔曼滤波（Unscented Kalman Filter， UKF）

UKF

通过上一篇文章，我们已经知道KF不适用于非线性系统，为处理非线性系统，我们通过一阶泰勒展式来近似（用线性函数近似）

这个方案直接的结果就是，我们对于具体的问题都要求解对应的一阶偏导（雅可比矩阵），求解雅可比矩阵本身就是费时的计算，而且我们上一篇还使用了Python而非C++，而且我们图省事还用了一个叫做numdifftools的库来实现雅可比矩阵的求解而不是自行推导，这一切都造成了我们上一次博客的代码执行效率奇低！

显然现实应用里面我们的无人车是不能接受这种延迟的，我相信很多同学也和我一样讨厌求解雅可比矩阵，那么，就让我们来学习一种相对简单的状态估计算法——UKF。

UKF使用的是统计线性化技术，我们把这种线性化的方法叫做无损变换（unscented transformation）这一技术主要通过n个在先验分布中采集的点（我们把它们叫sigma points）的线性回归来线性化随机变量的非线性函数

由于我们考虑的是随机变量的扩展，所以这种线性化要比泰勒级数线性化（EKF所使用的策略）更准确。

和EKF一样，UKF也主要分为预测和更新。

运动模型

本篇我们继续使用CTRV运动模型，不了解该模型的同学可以去看上一篇博客，CTRV模型的具体形式如下：

$$\vec{x}(t+\Delta t)=g(x(t))=\begin{pmatrix}\frac{v}{\omega}\sin(\omega\Delta t+\theta)-\frac{v}{\omega}\sin(\theta)+x(t)\\-\frac{v}{\omega}\cos(\omega\Delta t+\theta)+\frac{v}{\omega}\sin(\theta)+y(t)\\v\\\omega\Delta t+\theta\\\omega\end{pmatrix}+\begin{pmatrix}\frac{1}{2}\Delta t^2\mu_a\cos(\theta)\\\frac{1}{2}\Delta t^2\mu_a\sin(\theta)\\\Delta t\mu_a\\\frac{1}{2}\Delta t^2\mu_{\dot{\omega}}\\\Delta t\mu_{\dot{\omega}}\end{pmatrix},\quad \omega\neq 0$$

当偏航角为0时：

$$\vec{x}(t+\Delta t)=g(x(t))=\begin{pmatrix}v\cos(\theta)\Delta t+x(t)\\v\sin(\theta)\Delta t+y(t)\\v\\\omega\Delta t+\theta\\\omega\end{pmatrix}+\begin{pmatrix}\frac{1}{2}\Delta t^2\mu_a\cos(\theta)\\\frac{1}{2}\Delta t^2\mu_a\sin(\theta)\\\Delta t\mu_a\\\frac{1}{2}\Delta t^2\mu_{\dot{\omega}}\\\Delta t\mu_{\dot{\omega}}\end{pmatrix},\quad \omega=0$$

在EKF中，我们将直线加速度和偏航角加速度的影响当作我们的处理噪声，并且假设他们是满足均值为 0，方差为 σ_a 和 $\sigma_{\dot{\omega}}$，在这里，我们将噪声的影响直接考虑到我们的状态转移函数中来。至于函数中的不确定项 μ_a 和 $\mu_{\dot{\omega}}$ 我们后面再分析。

非线性处理/测量模型

我们知道我们在应用KF是面临的主要问题就是非线性处理模型（比如说CTRV）和非线性测量模型（RADAR测量）的处理。

我们从概率分布的角度来描述这个问题：

对于我们想要估计的状态，在k时刻满足均值为 $x_{k|k}$,方差为 $P_{k|k}$ 这样的一个高斯分布，这个是我们在k时刻的 **后验(Posterior)**（如果我们把整个卡尔曼滤波的过程迭代的来考虑的话），现在我们以这个后验为出发点，结合一定的先验知识（比如说CTRV运动模型）去估计我们在 $k+1$ 时刻的状态的均值和方差，这个过程就是卡尔曼滤波的预测，如果变换是线性的，那么预测的结果仍然是高斯分布，但是现实是我们的处理和测量模型都是非线性的，结果就是一个不规则分布，KF能够使用的前提就是所处理的状态是满足高斯分布的，为了解决这个问题，EKF是寻找一个线性函数来近似这个非线性函数，**而UKF就是去找一个与真实分布近似的高斯分布。**

UKF的基本思路就是： 近似非线性函数的概率分布要比近似非线性函数本身要容易！

那么如何去找一个与真实分布近似的高斯分布呢？——找一个与真实分布有着相同的均值和协方差的高斯分布。

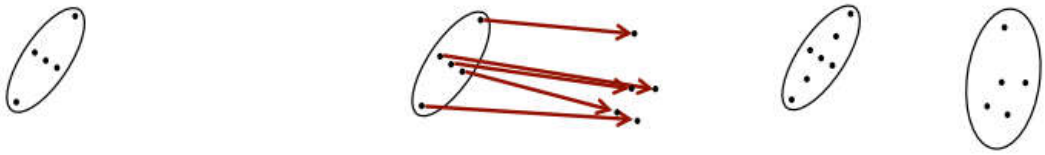
那么如何去找这样的均值和方差呢？——无损变换。

无损变换

计算一堆点（术语叫做sigma point），通过一定的手段产生的这些sigma点能够代表当前的分布，

然后将这些点通过非线性函数（处理模型）变换成一些新的点，

然后基于这些新的sigma点计算出一个高斯分布（带有权重地计算出来）如图：



预测

由一个高斯分布产生sigma point

通常，假定状态的个数为 n ， 我们会产生 2n+1 个sigma点，

其中第一个就是我们当前状态的均值 μ， sigma点集的均值的计算公式为：

$$\chi^{[1]} = \mu$$

$$\chi^{[i]} = \mu + \left(\sqrt{(n+\lambda)P}\right)_i \quad for\ i = 2, \ldots, n+1$$

$$\chi^{[i]} = \mu - \left(\sqrt{(n+\lambda)P}\right)_{i-n} \quad for\ i = n+2, \ldots, 2n+1$$

其中的 λ 是一个超参数，根据公式，λ 越大， sigma点就越远离状态的均值，λ 越小， sigma点就越靠近状态的均值。需要注意的是，在我们的CTRV模型中，状态数量 n 除了要包含5个状态以外，还要包含处理噪声 μ_a 和 $\mu_{\dot{\omega}}$,因为这些处理噪声对模型也有着非线性的影响。在增加了处理噪声的影响以后，我们的不确定性矩阵 P 就变成了：

$$P = \begin{pmatrix} P' & 0 \\ 0 & Q \end{pmatrix}$$

其中，P' 就是我们原来的不确定性矩阵（在CTRV模型中就是一个 5×5 的矩阵），Q是处理噪声的协方差矩阵，在CTRV模型中考虑到直线加速度核Q的形式为：

$$Q = \begin{bmatrix} \sigma_a^2 & 0 \\ 0 & \sigma_{\dot{\omega}}^2 \end{bmatrix}$$

其中的 σ_a^2，$\sigma_{\dot{\omega}}^2$ 和我们上一篇博客讲的一样。以上公式中还存在一个问题，那就是矩阵开平方根怎么计算的问题，同产情况下，我们求得是：

$$A = \sqrt{P}$$

其中，

$$AA^T = P$$

求解上式中的 A 是一个相对复杂的过程，但是如果 P 是对角矩阵的话，那么这个求解就会简化，在我们的实例中，P表示对估计状态的不确定性（协方差矩阵），我们会发现 P 基本上是对角矩阵（状态量之间的相关性几乎为0），所以我们可以首先对 P 做 **Cholesky分解(Cholesky decomposition)** ,然后分解得到的矩阵的下三角矩阵就是我们要求的 A ，在这里我们就不详细讨论了，在我们后面的实际例子中，我们直接调用库中相应的方法即可(注意：本次的实例我们换C++来实现，相比于Python，C++更加贴近我们实际的开发)：

```c++
MatrixXd A = P_aug.llt().matrixL();
```

预测sigma point

现在我们有sigma点集，我们就用非线性函数 $g()$ 来进行预测：

$$\chi_{k+1} = g(\chi_k, \mu_k)$$

需要注意的是，这里的输入 χ_k 是一个 $(7, 15)$ 的矩阵（因为考虑了两个噪声量），但是输出 $\chi_{k+1|k}$ 是一个$(5, 15)$ 的矩阵（因为这是预测的结果，本质上是基于运动模型的先验，先验中的均值不应当包含 a，$\dot{\omega}$ 这类不确定的量）

预测均值和方差

首先要计算出各个sigma点的权重，权重的计算公式为：

$$w^{[i]} = \frac{\lambda}{\lambda + n}, \quad i = 1$$

$$w^{[i]} = \frac{1}{2(\lambda + n)}, \quad i = 2, \ldots, 2n + 1$$

然后基于每个sigma点的权重去求新的分布的均值和方差：

$$\mu' = \sum_{i=1}^{2n+1} w^{[i]} \chi_{k+1}^{[i]}$$

$$P' = \sum_{i=1}^{2n+1} w^{[i]} (\chi_{k+1}^{[i]} - \mu')(\chi_{k+1}^{[i]} - \mu')^T$$

其中 μ' 即为我们基于CTRV模型预测的目标状态的先验分布的均值 $x_{k+1|k}$ ，它是sigma点集中每个点各个状态量的加权和， P' 即为先验分布的协方差（不确定性） $P_{k+1|k}$ ，由每个sigma点的方差的加权和求得。至此，预测的部分也就走完了，下面进入了UKF的测量更新部分。

测量更新

1.预测测量（将先验映射到测量空间然后算出均值和方差）

这篇博客继续使用上一篇（EKF）中的测量实验数据，那么我们知道，测量更新分为两个部分，LIDAR测量和RADAR测量，

其中LIDAR测量模型本身就是线性的，所以我们重点还是放在RADAR测量模型的处理上面，RADAR的测量映射函数为：

$$Z_{k+1|k} = \begin{pmatrix} \rho \\ \psi \\ \dot{\rho} \end{pmatrix} = \begin{pmatrix} \sqrt{x^2 + y^2} \\ \operatorname{atan}_2(y, x) \\ \frac{v\cos(\theta)x + v\sin(\theta)y}{\sqrt{x^2+y^2}} \end{pmatrix}$$

这也是个非线性函数，我们用 $h()$ 来表示它，再一次，我们使用无损转换来解决，但是这里，我们可以不用再产生sigma points了，我们可以直接使用预测出来的sigma点集，并且可以忽略掉处理噪声部分。那么对先验的非线性映射就可以表示为如下的sigma point预测（即预测非线性变换以后的均值和协方差）：

$$z_{k+1|k} = \sum_{i=1}^{2n+1} w^{[i]} Z_{k+1|k}^{[i]}$$

$$S_{k+1|k} = \sum_{i=1}^{2n+1} w^{[i]} (Z_{k+1|k}^{[i]} - z_{k+1|k})(Z_{k+1|k}^{[i]} - z_{k+1|k})^T + R$$

和前面的文章一样，这里的 R 也是测量噪声，在这里我们直接将测量噪声的协方差加到测量协方差上是因为该噪声对系统没有非线性影响。在本例中，以RADAR的测量为例，那么测量噪声R为：

$$R = E[ww^T] = \begin{pmatrix} \sigma_\rho^2 & 0 & 0 \\ 0 & \sigma_\psi^2 & 0 \\ 0 & 0 & \sigma_{\dot{\rho}}^2 \end{pmatrix}$$

2.更新状态

首先计算出sigma点集在状态空间和测量空间的互相关函数，计算公式如下：

$$T_{k+1|k} = \sum_{i=1}^{2n+1} w^{[i]}(X_{k+1|k}^{[i]} - x_{k+1|k})(Z_{k+1|k}^{[i]} - z_{k+1|k})^T$$

后面我们就完整地按照卡尔曼滤波的更新步骤计算即可，先计算卡尔曼增益：

$$K_{k+1|k} = T_{k+1|k} \cdot S_{k+1|k}^{-1}$$

更新状态（也就是作出最后的状态估计）：

$$x_{k+1|k+1} = x_{k+1|k} + K_{k+1|k}(z_{k+1} - z_{k+1|k})$$

其中 z_{k+1} 是新得到的测量，而 $z_{k+1|k}$ 则是我们根据先验计算出来的在测量空间的测量。

更新状态协方差矩阵：

$$P_{k+1|k+1} = P_{k+1|k} - K_{k+1|k}S_{k+1|k}K_{k+1|k}^T$$

至此，UKF的完整理论部分就介绍完了，具体的无损变换的意义和推导由于太过于理论化了，大家如果感兴趣可以去看这几片文献：

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.136.6539&rep=rep1&type=pdf

https://www.cse.sc.edu/~terejanu/files/tutorialUKF.pdf

http://ais.informatik.uni-freiburg.de/teaching/ws13/mapping/pdf/slam06-ukf-4.pdf

UKF实例

我们继续基于上一篇文章的数据来做状态估计的实例，不过，为了更加接近实际实际的应用，我们本节采用C++实现。

由于本节使用的C++代码量相对较大，而且为多文件的项目，代码就不再一一贴出，所有代码我都已经上传至如下地址：

http://download.csdn.net/download/adamshan/10041096

1. 运行-效果

首先解压，编译:

```
mkdir build
cd build
cmake ..
make
```

运行：

```
./ukf ../data/data_synthetic.txt ../data/output.txt
```

得到计算的结果：

```
Accuracy - RMSE:
0.0723408
0.0821208
```

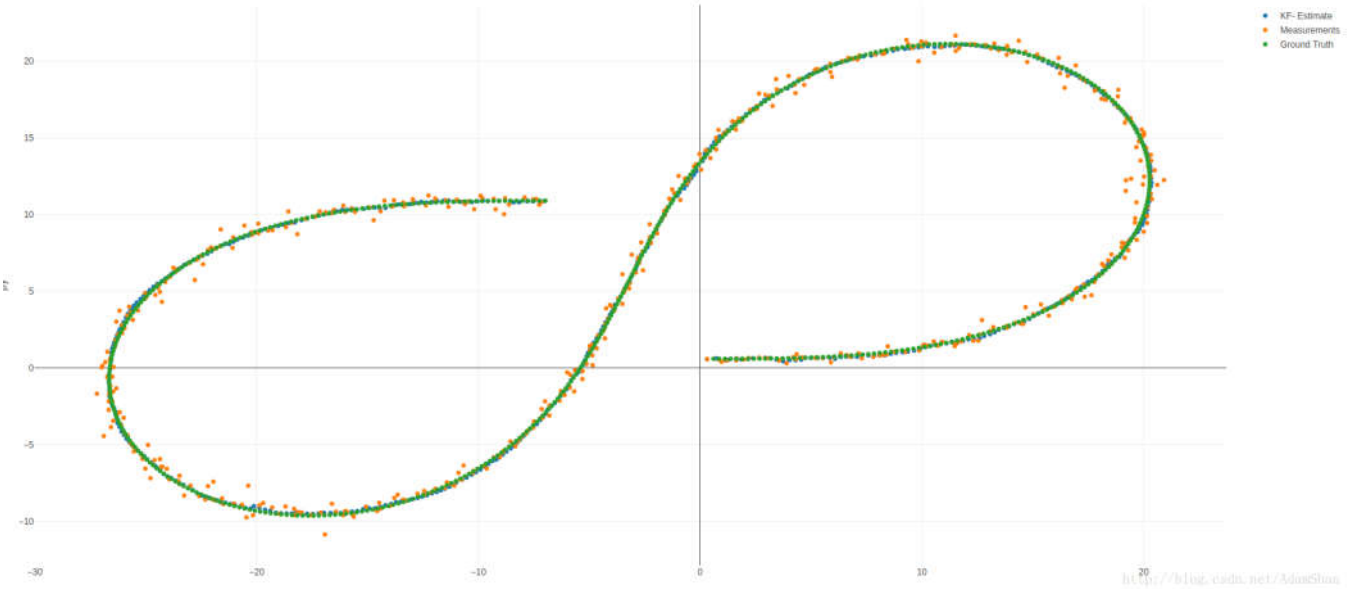
```
0.342265
 0.23017
```

我们发现，这个UKF要比我们上一篇博客写的EKF的效率要高得多得多。

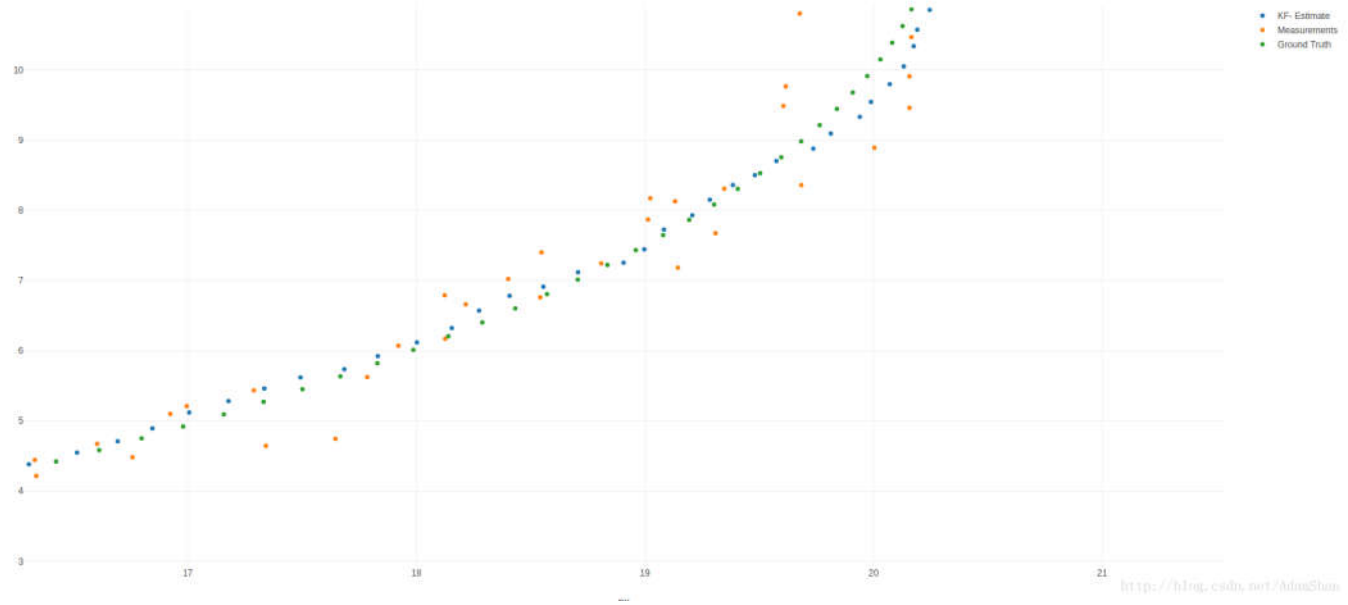
我们执行可视化的python脚本看一下效果：

```
cd ../data
python plot.py
```

得到如下结果：



放大一点：



UKF在我们这个问题中的执行效率和估计精度都高于我们上一篇文章中实现的EKF,下面就让我们看一下核心代码。

核心代码

首先是预测，预测主要包含三部分，分别是：

产生sigma点集

基于CTRV模型预测sigma点集

计算新的均值核方差

1. 产生sigma点集

```
void UKF::AugmentedSigmaPoints(MatrixXd *Xsig_out) {

    //create augmented mean vector
    VectorXd x_aug = VectorXd(7);

    //create augmented state covariance
    MatrixXd P_aug = MatrixXd(7, 7);

    //create sigma point matrix
    MatrixXd Xsig_aug = MatrixXd(n_aug_, 2 * n_aug_ + 1);

    //create augmented mean state
    //create augmented covariance matrix
    //create square root matrix
    //create augmented sigma points
    x_aug.head(5) = x_;
    x_aug(5) = 0;
    x_aug(6) = 0;

    P_aug.fill(0.0);
```

2.基于CTRV模型预测sigma点集

```
void UKF::SigmaPointPrediction(MatrixXd &Xsig_aug, double delta_t) {

    for(int i =0; i < (2 * n_aug_ + 1); i++){
        VectorXd input_x = Xsig_aug.col(i);
        float px = input_x[0];
        float py = input_x[1];
        float v = input_x[2];
        float psi = input_x[3];
        float psi_dot = input_x[4];
        float mu_a = input_x[5];
        float mu_psi_dot_dot = input_x[6];

        VectorXd term2 = VectorXd(5);
        VectorXd term3 = VectorXd(5);

        VectorXd result = VectorXd(5);
        if(psi_dot < 0.001){
            term2 << v * cos(psi) * delta_t, v * sin(psi) * delta_t, 0, psi_dot * delta_t, 0;
            term3 << 0.5 * delta_t*delta_t * cos(psi) * mu_a,
                    0.5 * delta_t*delta_t * sin(psi) * mu_a
```

3.计算新的均值核方差

```
void UKF::PredictMeanAndCovariance() {
    x_.fill(0.0);
    for(int i=0; i<2*n_aug_+1; i++){
        x_ = x_+ weights_[i] * Xsig_pred_.col(i);
    }

    P_.fill(0.0);
    for(int i=0;  i<2*n_aug_+1; i++){
        VectorXd x_diff = Xsig_pred_.col(i) - x_;
        while (x_diff[3]> M_PI)
            x_diff[3] -= 2.*M_PI;
        while (x_diff[3] <-M_PI)
            x_diff[3]+=2.*M_PI;
        P_ = P_ + weights_[i] * x_diff * x_diff.transpose();
    }
}
```

测量更新主要分为

预测LIDAR测量

预测RADAR测量

更新状态

1. 预测LIDAR测量

```
void UKF::PredictLaserMeasurement(VectorXd &z_pred, MatrixXd &S, MatrixXd &Zsig, long n_z) {
    for(int i=0; i < 2*n_aug_+1; i++){
        float px = Xsig_pred_.col(i)[0];
        float py = Xsig_pred_.col(i)[1];

        VectorXd temp = VectorXd(n_z);
        temp << px, py;
        Zsig.col(i) = temp;
    }

    z_pred.fill(0.0);
    for(int i=0; i < 2*n_aug_+1; i++){
        z_pred = z_pred + weights_[i] * Zsig.col(i);
    }

    S.fill(0.0);
    for(int i=0; i < 2*n_aug_+1; i++){
        //residual
        VectorXd z_diff = Zsig.col(i) - z_pred;
```

2. 预测RADAR测量

```
void UKF::PredictRadarMeasurement(VectorXd &z_pred, MatrixXd &S, MatrixXd &Zsig, long n_z) {
    for(int i=0; i < 2*n_aug_+1; i++){
        float px = Xsig_pred_.col(i)[0];
        float py = Xsig_pred_.col(i)[1];
        float v = Xsig_pred_.col(i)[2];
        float psi = Xsig_pred_.col(i)[3];
        float psi_dot = Xsig_pred_.col(i)[4];

        float temp = px * px + py * py;
        float rho = sqrt(temp);
        float phi = atan2(py, px);
        float rho_dot;
        if(fabs(rho) < 0.0001){
            rho_dot = 0;
        } else{
            rho_dot =(px * cos(psi) * v + py * sin(psi) * v)/rho;
        }

        VectorXd temp1 = VectorXd(3);
        temp1 << rho, phi, rho_dot;
```

3.更新状态

```
void UKF::UpdateState(VectorXd &z, VectorXd &z_pred, MatrixXd &S, MatrixXd &Zsig, long n_z) {

    //create matrix for cross correlation Tc
    MatrixXd Tc = MatrixXd(n_x_, n_z);

    //calculate cross correlation matrix
    //calculate Kalman gain K;
    //update state mean and covariance matrix

    Tc.fill(0.0);
    for(int i=0; i < 2*n_aug_+1; i++){
        VectorXd x_diff = Xsig_pred_.col(i) - x_;

        while (x_diff(3)> M_PI) x_diff(3)-=2.*M_PI;
        while (x_diff(3)<-M_PI) x_diff(3)+=2.*M_PI;

        //residual
        VectorXd z_diff = Zsig.col(i) - z_pred;
        if(n_z == 3){
            //angle normalization
```

以上就是我们的UKF的核心算法实现了，可能光看这些核心代码还是没办法理解，所以感兴趣的童鞋就去下载来再详细研究把～

上一篇文章我买了个关子，没有贴出把结果可视化的代码，本次代码已经一本包含在项目里（具体见/data/plot.py）

前面三篇文章都介绍了卡尔曼滤波相关的算法，大家应该已经看出卡尔曼滤波这类贝叶斯滤波器在无人驾驶汽车系统哦的重要性了，卡尔曼滤波作为重要的状态估计算法，是无人驾驶的必备技能点，所以相关的扩展阅读大家可以多多开展，

本系列暂时就不再讨论了，下面我想向大家介绍应用于无人驾驶中的控制算法，控制作为自动化中重要的技术，有着很多的分支，后面的文章我会继续由理论到实践逐一向大家介绍相关技术～

参考：