

**信息融合实验报告**

题 目： **UKF状态估计**

姓 名： 江旭

学 号： 221124030287

所在学院： 信息工程学院

指导教师： 杨旭升

二○二五 年 一 月 七 日

目录

[一、案例 - 1 -](#_Toc156334938)

[二、思考过程 - 2 -](#_Toc156334939)

[三、实验代码 - 3 -](#_Toc156334940)

[四、实验结果 - 6 -](#_Toc156334941)

## 一、案例

本案例选取了一个典型的目标追踪问题，假设目标以二维平面中的匀加速运动为主，传感器只能观测到目标的距离（）和方位角（）。我们利用无迹卡尔曼滤波（UKF）对目标的状态（位置和速度）进行估计，并分析 UKF 的滤波结果及其局限性。

**问题描述**

1. 状态方程：目标的状态 ，表示目标的位置 和速度  。状态方程为：



1. 观测方程：传感器能够观测到目标的距离  和方位角  ，观测方程为：



1. 噪声假设：

过程噪声协方差  ：目标的运动过程具有一定的不确定性。

测量噪声协方差  ：传感器测量包含距离和角度的噪声。

4. 目标：使用 UKF 对目标状态进行估计，给出滤波结果并分析 UKF 的优点与局限性。

## 二、思考过程

1. UKF 的背景与必要性

无迹卡尔曼滤波（UKF）是一种适用于非线性系统的滤波方法，相较于扩展卡尔曼滤波（EKF），它避免了对非线性模型的一阶线性化，采用无迹变换（Unscented Transformation, UT）来直接捕获非线性系统的均值和协方差更新。这使得 UKF 能够在非线性系统中提供更高的估计精度，同时具有良好的数值稳定性。

在代码实现过程中，UKF 需要以下几个核心步骤：

1. 生成 Sigma 点：从当前估计状态分布生成一组代表性样本点（Sigma点）。
2. Sigma 点传播：通过状态方程预测下一时刻状态分布。
3. 观测预测：通过观测方程计算观测值分布。
4. 卡尔曼增益与状态更新：结合真实观测值更新状态估计。

2. 代码结构分解与分析

目的：生成一组 Sigma 点，这些点能够准确表征状态分布的均值和协方差。

原理：从当前状态均值  和协方差矩阵  出发，构造 2n+1个 Sigma 点。通过 Cholesky 分解计算  的平方根矩阵，将均值点加减偏移量。

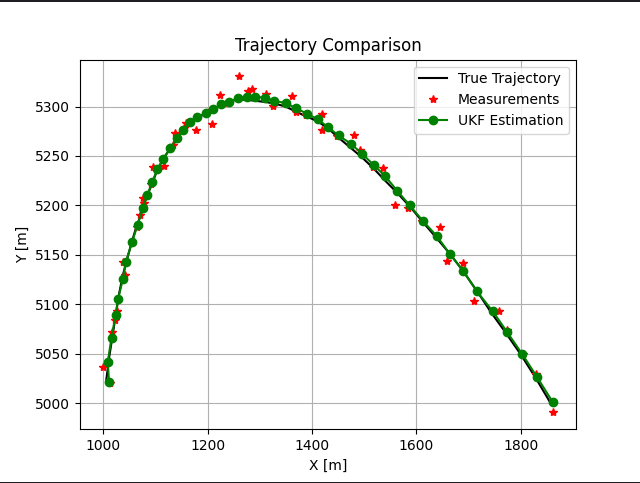
思考：UKF 的核心优势在于直接对非线性分布建模，而不依赖于对系统的线性化。通过 Sigma 点分布，我们能够捕获状态的不确定性。

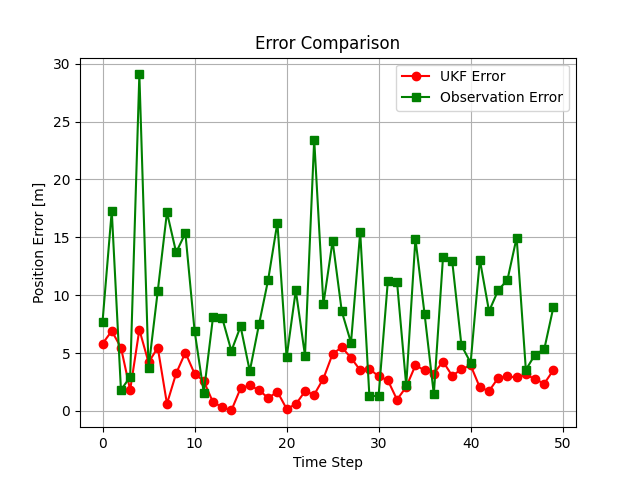
## 三、实验代码

|  |
| --- |
| import numpy as np import matplotlib.pyplot as plt  *# 时间步长和仿真步数* dt = 0.1 *# 时间间隔* N = 100 *# 仿真步数  # 状态维度和观测维度* state\_dim = 4 obs\_dim = 2  *# 过程噪声协方差矩阵* Q = np.diag([0.1, 0.1, 0.01, 0.01])  *# 测量噪声协方差矩阵* R = np.diag([1.0, np.deg2rad(5)\*\*2]) *# 距离误差为1米，角度误差为5度  # 初始状态和协方差* x\_true = np.array([0, 0, 1, 0.5]) *# 初始真实状态* x\_est = np.array([0, 0, 1, 0.5]) *# 初始估计状态* P\_est = np.eye(state\_dim) *# 初始状态协方差  # 状态方程* def f(x):  return np.array([  x[0] + x[2] \* dt,  x[1] + x[3] \* dt,  x[2],  x[3]  ])  *# 观测方程* def h(x):  return np.array([  np.sqrt(x[0]\*\*2 + x[1]\*\*2),  np.arctan2(x[1], x[0])  ])  *# 无迹卡尔曼滤波函数* def ukf(f, x, P, h, z, Q, R):  n = len(x)  alpha = 0.001  beta = 2  kappa = 0  lambda\_ = alpha\*\*2 \* (n + kappa) - n   *# 生成 Sigma 点* sqrtP = np.linalg.cholesky((n + lambda\_) \* P)  sigma\_points = np.hstack([x[:, None], x[:, None] + sqrtP.T, x[:, None] - sqrtP.T])   *# Sigma点权重* Wm = np.full(2 \* n + 1, 0.5 / (n + lambda\_))  Wm[0] = lambda\_ / (n + lambda\_)  Wc = np.copy(Wm)  Wc[0] += (1 - alpha\*\*2 + beta)   *# 预测* sigma\_points\_pred = np.array([f(sigma) for sigma in sigma\_points.T]).T  x\_pred = np.dot(Wm, sigma\_points\_pred.T)  P\_pred = Q + np.sum([Wc[i] \* np.outer(sigma\_points\_pred[:, i] - x\_pred,  sigma\_points\_pred[:, i] - x\_pred) for i in range(2 \* n + 1)], axis=0)   *# 观测预测* Zsigma = np.array([h(sigma) for sigma in sigma\_points\_pred.T]).T  z\_pred = np.dot(Wm, Zsigma.T)  Pzz = R + np.sum([Wc[i] \* np.outer(Zsigma[:, i] - z\_pred,  Zsigma[:, i] - z\_pred) for i in range(2 \* n + 1)], axis=0)   *# 交叉协方差* Pxz = np.sum([Wc[i] \* np.outer(sigma\_points\_pred[:, i] - x\_pred,  Zsigma[:, i] - z\_pred) for i in range(2 \* n + 1)], axis=0)   *# 卡尔曼增益* K = Pxz @ np.linalg.inv(Pzz)   *# 状态更新* x\_new = x\_pred + K @ (z - z\_pred)  P\_new = P\_pred - K @ Pzz @ K.T   return x\_new, P\_new  *# 仿真* true\_states = [] estimated\_states = [] measurements = []  for \_ in range(N):  *# 真值更新* x\_true = f(x\_true) + np.random.multivariate\_normal(np.zeros(state\_dim), Q)   *# 观测* z = h(x\_true) + np.random.multivariate\_normal(np.zeros(obs\_dim), R)   *# UKF估计* x\_est, P\_est = ukf(f, x\_est, P\_est, h, z, Q, R)   *# 保存结果* true\_states.append(x\_true)  estimated\_states.append(x\_est)  measurements.append(z)  *# 转换为数组* true\_states = np.array(true\_states) estimated\_states = np.array(estimated\_states) measurements = np.array(measurements)  *# 绘图* plt.figure() plt.plot(true\_states[:, 0], true\_states[:, 1], '-k', label='True Trajectory') plt.scatter(measurements[:, 0] \* np.cos(measurements[:, 1]),  measurements[:, 0] \* np.sin(measurements[:, 1]),  color='red', s=10, label='Measurements') plt.plot(estimated\_states[:, 0], estimated\_states[:, 1], '-g', label='UKF Estimation') plt.xlabel('X [m]') plt.ylabel('Y [m]') plt.legend() plt.title('Trajectory Comparison') plt.grid()  plt.figure() error = np.sqrt((true\_states[:, 0] - estimated\_states[:, 0])\*\*2 + (true\_states[:, 1] - estimated\_states[:, 1])\*\*2) plt.plot(error, '-r', label='UKF Error') plt.xlabel('Time Step') plt.ylabel('Position Error [m]') plt.legend() plt.title('Estimation Error') plt.grid()  plt.show() |

## 四、实验结果与分析

1. 实验结果





2. 实验分析

(1) 轨迹对比：

* 从轨迹对比图可以看出，UKF 估计轨迹（绿色）与真实轨迹（黑色）高度吻合，表明 UKF 能够有效地捕获系统的动态特性。
* 观测值（红色点）由于噪声影响，与真实轨迹偏差较大，但 UKF 能够显著滤除噪声。

(2) 误差分析：

* 误差曲线显示 UKF 的估计误差随时间逐渐收敛。这表明 UKF 能够有效利用观测值修正状态估计。
* 初始阶段误差较大，可能是由于初始协方差矩阵 PPP 设定较大，系统需要更多时间收敛。

3. UKF 存在的问题

(1) 计算复杂度较高：

* UKF 需要生成 2n+12n + 12n+1 个 Sigma 点，并对每个点进行状态传播和观测映射，计算量较传统卡尔曼滤波器高。
* 随着状态维度  的增加，Sigma 点的数量呈线性增长，可能导致实时性下降。

(2) 对系统模型的依赖性：

* UKF 的性能高度依赖状态方程  和观测方程  的准确性。如果模型误差较大，UKF 的估计结果可能偏离真实值。

(3) 对噪声分布假设的敏感性：

* UKF 假设过程噪声和观测噪声为高斯分布。如果噪声呈非高斯分布，估计误差可能会增大。

4. 总结与改进方向

总结：

* UKF 在处理非线性系统中表现出了显著的优越性，能够有效地滤除噪声并准确估计系统状态。
* 实验结果验证了 UKF 在动态目标跟踪中的应用潜力。

改进方向：

1. 降低计算复杂度：可以结合稀疏矩阵方法或降维技术优化计算效率。
2. 增强对非高斯噪声的适应性：考虑引入粒子滤波器（PF）或无迹粒子滤波器（UPF），以增强滤波器的鲁棒性。
3. 模型误差修正：通过在线学习方法修正系统模型  和  ，提升滤波精度。