机器人导航python实践(入门)

Lecture2-定位

北航 国新院 实验实践课

智能系统与人形机器人国际研究中心

教 师： 欧 阳 老 师

邮 箱： ouyangkid@buaa.edu.cn

学 期： 2025年秋季

目 录

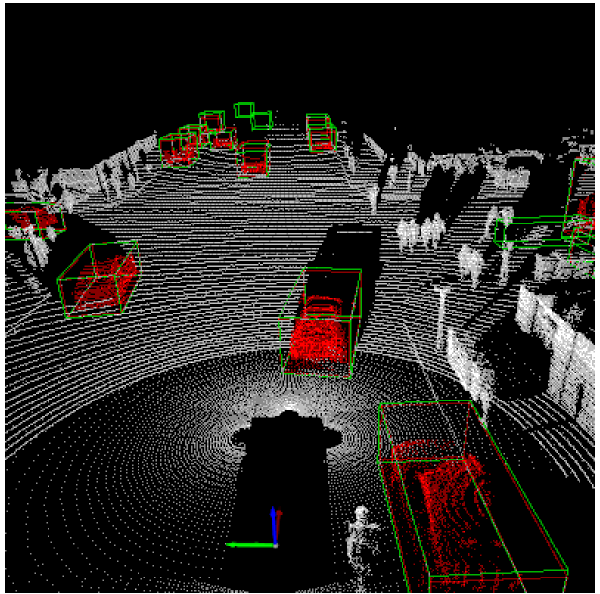
Contents

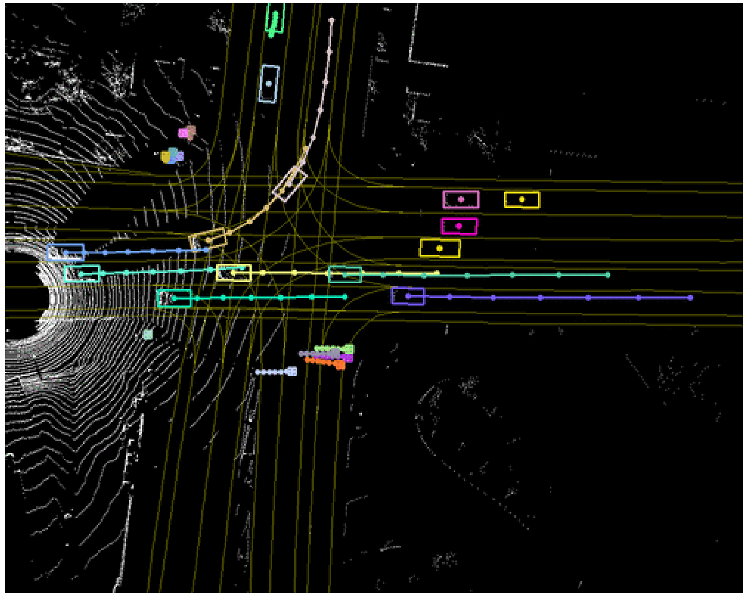
Part 1 课程内容安排

内容安排

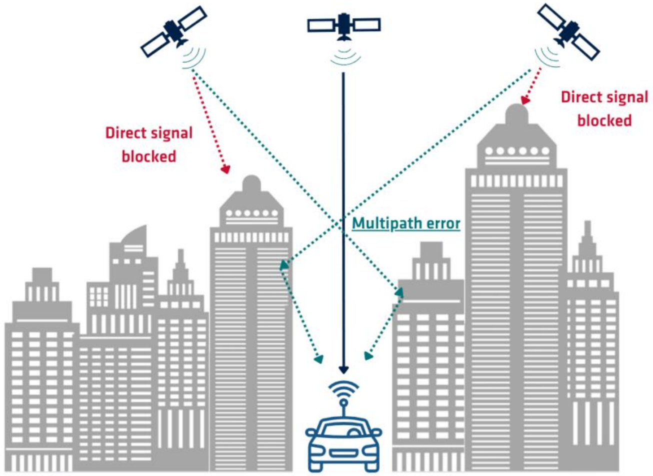
机器人依赖传感器确定自身的（绝对）位置和朝向，从而完成基础的自身感知。常用的传感器包括GNSS（GPS、伽利略、北斗）、差分GNSS（带地面差分站）、RTK，以及室内的UWB等，但所有传感器均存在测量噪声，进一步叠加机器人的运动估计误差，会导致定位失真。

动态目标追踪&轨迹预测优化





峡谷效应
多径误差



Part 1 课程内容安排

内容安排

不同滤波算法基于历史信息和贝叶斯理论**修正观测误差**

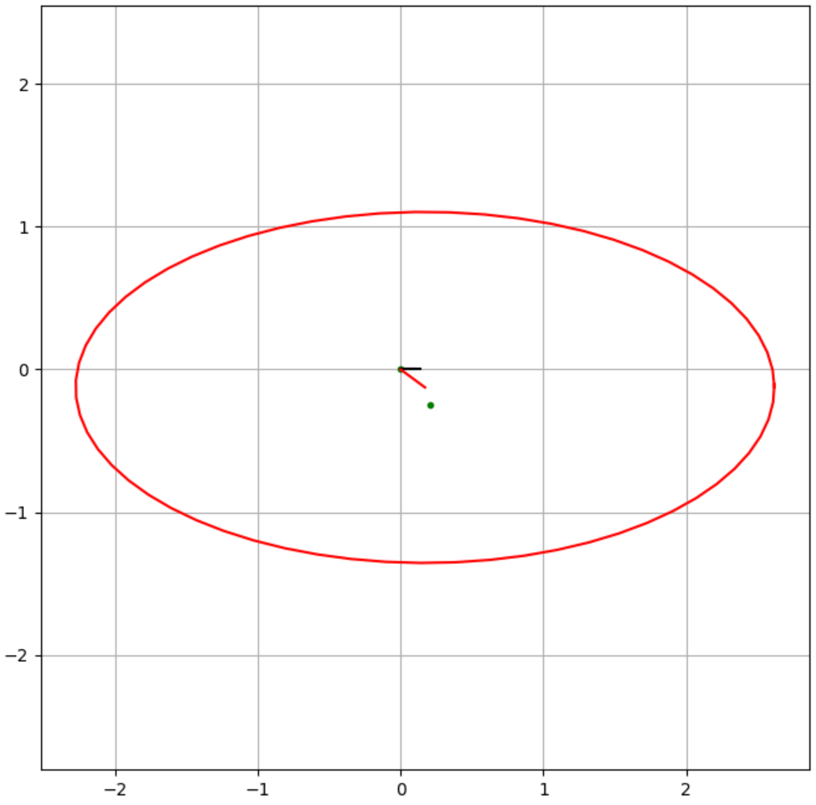
- 定位算法 总体框架
  - Kalman Filter 卡尔曼滤波
    - 扩展卡尔曼滤波定位
    - 集合卡尔曼滤波定位
    - 无迹卡尔曼滤波定位
  - Histogram Filter 直方图滤波
  - Particle Filter 粒子滤波

目录

Contents

Part 2 扩展Kalman滤波定位

Extended（扩展） Kalman filter



- 绿色散点：传感器定位结果，如GPS、RTK等。
- 蓝色曲线：GT轨迹。
- 黑色曲线：航迹推算轨迹。
- 红色折线：扩展Kalman滤波定位结果。
- 红色椭圆：对应EKF的协方差椭圆。

标准卡尔曼滤波（KF）只能处理**线性系统**（比如匀速直线运动），但现实中很多系统是非线性的（比如汽车转向、无人机飞行）。EKF 的 “扩展” 就在于它能处理**非线性系统**，核心方法是：

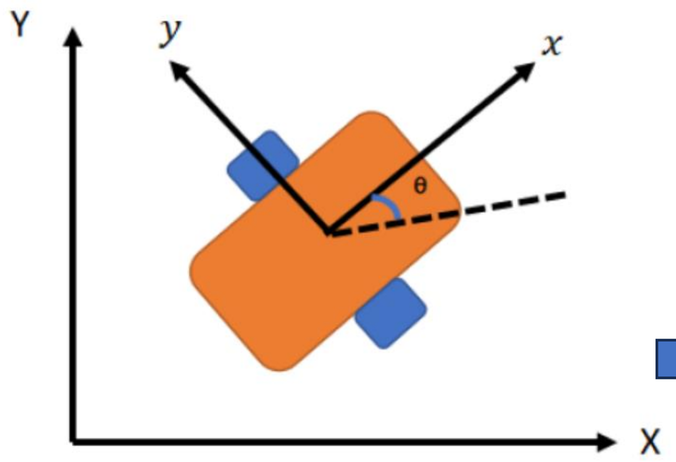
预测阶段：用当前状态估计值计算雅可比矩阵（简单理解为 “局部线性化”），把非线性问题近似成线性问题。
更新阶段：同样用雅可比矩阵处理观测模型的非线性。

可以理解为，EKF 是 KF 的 “升级版”，使Kalman滤波应用到更复杂的场景中。

Part 2 扩展Kalman滤波定位

扩展Kalman—上期知识点回顾

2D运动模型



2D 刚体运动模型描述了平面上物体（如机器人）的位置和姿态随时间的变化规律。用四维状态向量，位置坐标（x,y），航向角θ和速度v构成。

state vector= [x, y, θ, v]^T

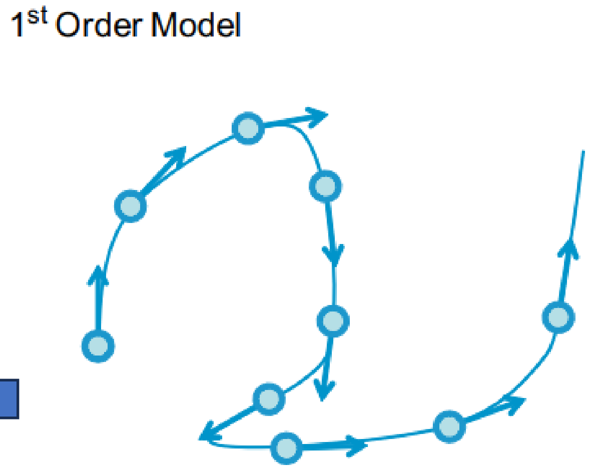
$$\mathbf{u}_t = [v_t, \omega_t]$$

```python
def motion_model(x, u):
    F = np.array([[1.0, 0, 0, 0],
                  [0, 1.0, 0, 0],
                  [0, 0, 1.0, 0],
                  [0, 0, 0, 0]])

    B = np.array([[DT * math.cos(x[2, 0]), 0],
                  [DT * math.sin(x[2, 0]), 0],
                  [0.0, DT],
                  [1.0, 0.0]])

    x = F @ x + B @ u

    return x
```

$$\mathbf{x}_{t+1} = f(\mathbf{x}_t, \mathbf{u}_t) = F\mathbf{x}_t + B\mathbf{u}_t$$

Agent自带速度和角速度传感器（带噪），需要估计自身**位置**信息

航迹推算Dead Reckoning



若初始位置为（x0,y0），某段时间内移动距离为 di、航向角为 θi，则新位置 (xn,yn) 为：

$$Xn = X0 + \sum_{k=0}^{n} d_i cos\theta_i$$

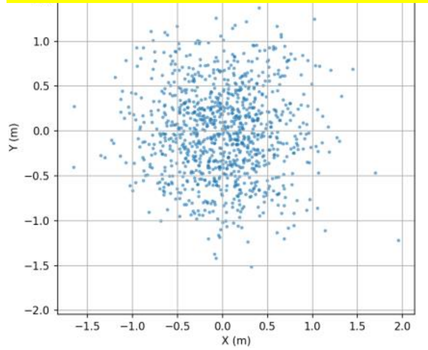
$$Yn = Y0 + \sum_{k=0}^{n} d_i \sin\theta_i$$

Part 2 | 扩展Kalman滤波定位

扩展Kalman—上期知识点回顾

Agent希望通过GPS定位（同样带噪）信息定期修正估计**位置**的误差。

模拟传感器定位误差



定义GT轨迹

通过恒定(线)速度和角速度，模拟圆周运动。

```python
def calc_input():
    v = 1.0  # [m/s]
    yawrate = 0.1  # [rad/s]
    u = np.array([[v], [yawrate]])
    return u
```

$\mathbf{u}_t = [v_t, \omega_t]$

•如果只设置了 v > 0, yawrate = 0，为直线运动；
•如果 v > 0, yawrate ≠ 0，为沿着一个半径为 v / yawrate 的**匀速圆周运动**。

转弯半径 R: $R = \frac{v}{|\text{yawrate}|} = \frac{1.0}{0.1} = 10$米

运动周期 T（绕圆一周的时间）: $T = \frac{2\pi R}{v} = \frac{2\pi \times 10}{1.0} \approx 62.8$秒

引入GPS定位噪声:

X/Y方向引入标准差为0.5m的噪声，两者独立不相关。
GPS_NOISE = np.diag([0.5, 0.5]) ** 2

•**均值**: 0（无偏噪声）。
•**标准差**: 0.5 米。
•**分布**: 高斯分布（正态分布）。

高斯分布

```python
z = observation_model(xTrue) + GPS_NOISE @ np.random.randn(2, 1)
```

观测模型

```python
def observation_model(x):
    H = np.array([
        [1, 0, 0, 0],
        [0, 1, 0, 0]
    ])

    z = H @ x

    return z
```

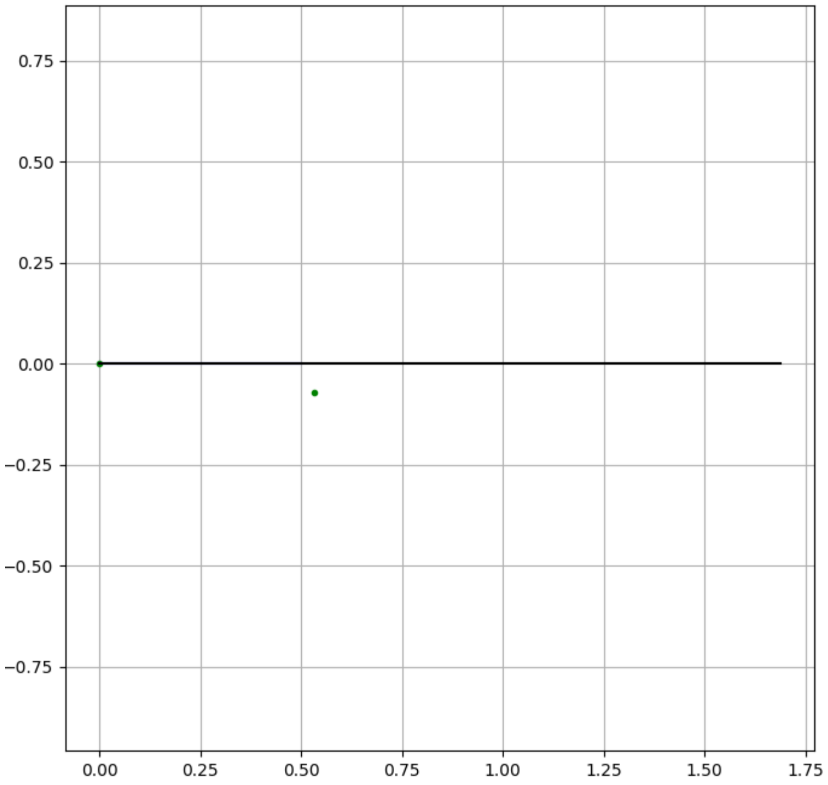
$\mathbf{z}_t = [x_t, y_t]$

只影响xy坐标

Part 2 扩展Kalman滤波定位

扩展Kalman

只考虑航迹推算（黑色）的结果，将模拟周期设置为63s，仿真时间间隔0.5s，覆盖一个圆的轨迹。对应航迹推算存在明显的累积误差， EKF轨迹（蓝色）基本沿GPS观测散点进行圆周运动。

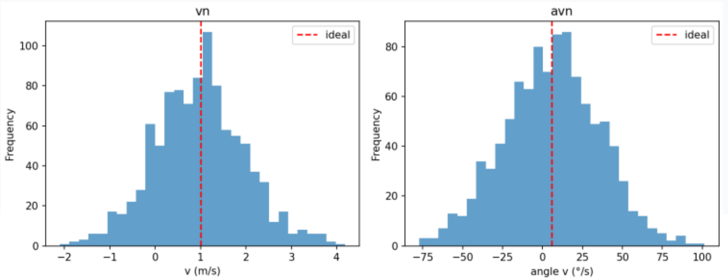


更新真实运动状态 → 添加GPS测量噪声 → 添加运动噪声 → 更新航迹推算

模拟运动体的控制误差

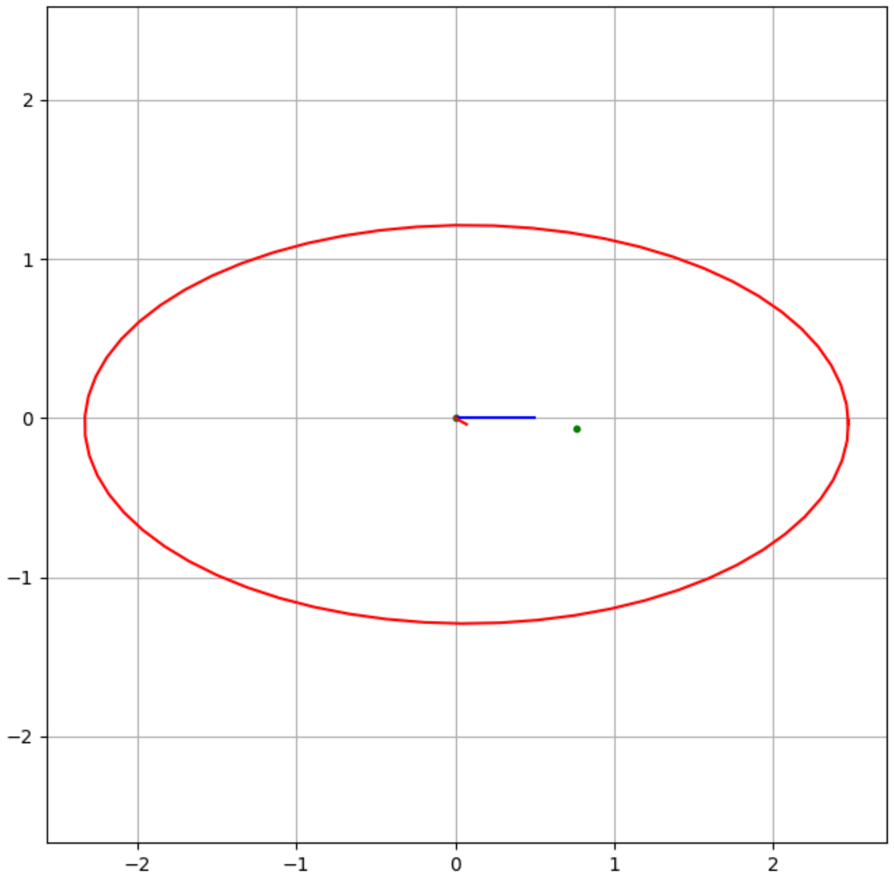
- 线速度误差：均值 0， 标准差 1.0 m/s。
- 角速度误差：均值 0， 标准差 0.295 rad/s（约 17°/s）。
- 两者相互独立。

```python
INPUT_NOISE = np.diag([1.0, np.deg2rad(30.0)]) ** 2

def observation(xTrue, xd, u):
    xTrue = motion_model(xTrue, u)

    # add noise to gps x-y
    z = observation_model(xTrue) + GPS_NOISE @ np.random.randn(2, 1)

    # add noise to input
    ud = u + INPUT_NOISE @ np.random.randn(2, 1)

    xd = motion_model(xd, ud)

    return xTrue, z, xd, ud
```



Part 2 扩展Kalman滤波定位

扩展Kalman



Q为系统运动模型的协方差矩阵，用来估计agent的运动噪声；R为观测的不确定性协方差矩阵（与传感器测量误差相关）。

```
# Covariance for EKF simulation
Q = np.diag([
    0.1,  # variance of location on x-axis
    0.1,  # variance of location on y-axis
    np.deg2rad(1.0),  # variance of yaw angle
    1.0  # variance of velocity
]) ** 2  # predict state covariance
R = np.diag([1.0, 1.0]) ** 2  # Observation x,y position covariance
```

假设模型预测的位置、航向的不确定较小。

速度预测的不确定性较大

观测噪声较大，可基于传感器厂家的标定值配置，也通过静态测试获取

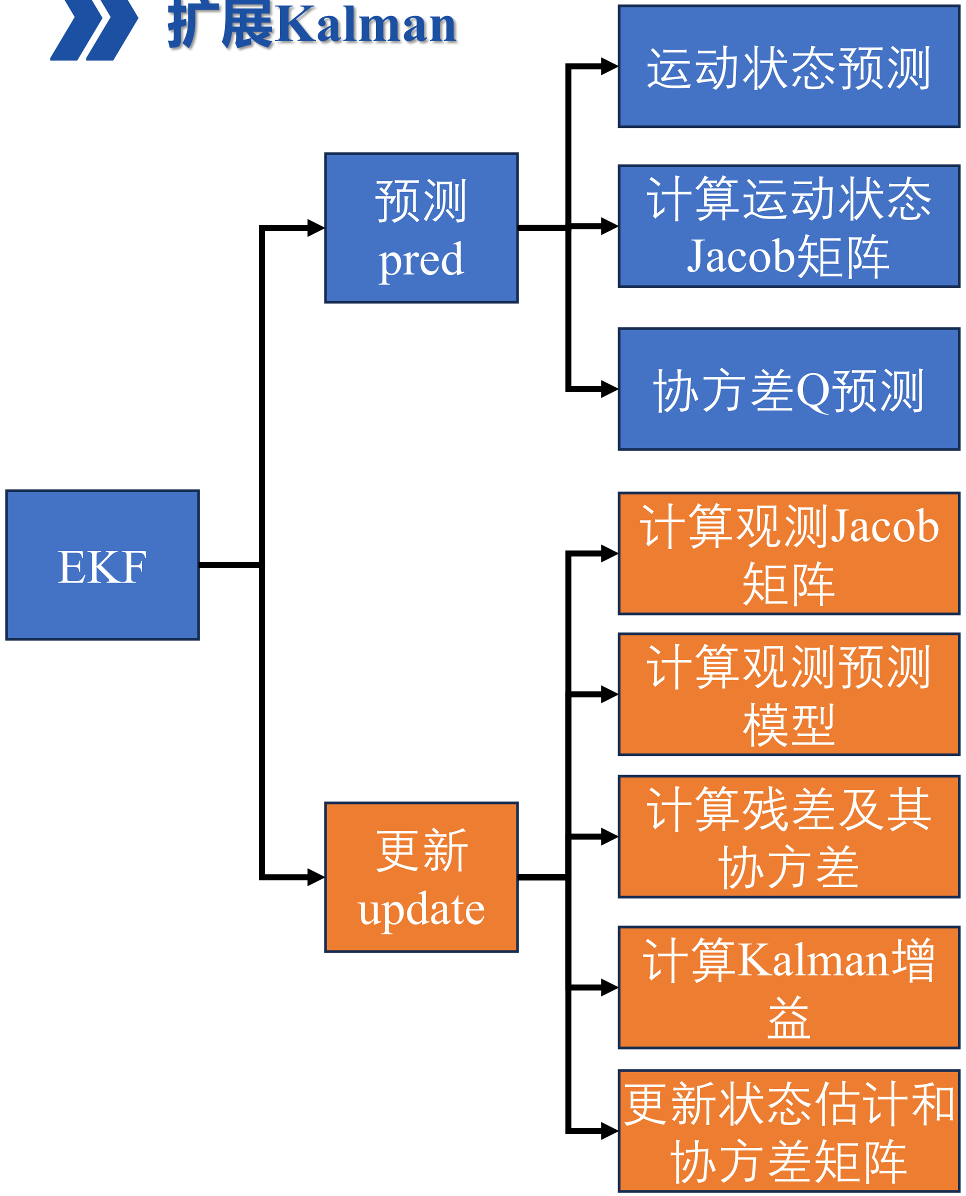
假设：GPS观测噪声大于机器人运动的噪声。

Q：控制**预测模型的信任程度**，值越小越信任模型。反之，设置过大则依赖观测，比如认为传感器精度极高，如：室内的红外动补。

R：控制**观测数据的信任程度**，值越小越信任观测。反之，设置过大则依赖模型预测。

Part 2 | 扩展Kalman滤波定位

扩展Kalman



=== Predict ===

$$x_{Pred} = F x_t + B u_t$$

$$P_{Pred} = J_f P_t J_f^T + Q$$

```
xPred = motion_model(xEst, u)
jF = jacob_f(xEst, u)
PPred = jF @ PEst @ jF.T + Q
```

=== Update ===

$$z_{Pred} = H x_{Pred}$$

$$y = z - z_{Pred}$$

$$S = J_g P_{Pred}. J_g^T + R$$

$$K = P_{Pred}. J_g^T S^{-1}$$

$$x_{t+1} = x_{Pred} + K y$$

$$P_{t+1} = (I - K J_g) P_{Pred}$$

```
jH = jacob_h()
zPred = observation_model(xPred)
y = z - zPred
S = jH @ PPred @ jH.T + R
K = PPred @ jH.T @ np.linalg.inv(S)
xEst = xPred + K @ y
PEst = (np.eye(len(xEst)) - K @ jH) @ PPred
```

Part 2 扩展Kalman滤波定位

扩展Kalman

EKF

预测

运动状态预测

计算运动状态Jacob矩阵

协方差Q预测

=== Predict ===

$$x_{Pred} = Fx_t + Bu_t$$

$$P_{Pred} = J_f P_t J_f^T + Q$$

```
xPred = motion_model(xEst, u)
jF = jacob_f(xEst, u)
PPred = jF @ PEst @ jF.T + Q
```

航迹推算模型

$$\begin{bmatrix} x + v\cos(\phi)\Delta t \\ y + v\sin(\phi)\Delta t \\ \phi + \omega\Delta t \\ v \end{bmatrix}$$

运动模型：当前状态估计

对运动模型求导

$$J_f = \begin{bmatrix} \frac{\partial x'}{\partial x} & \frac{\partial x'}{\partial y} & \frac{\partial x'}{\partial \phi} & \frac{\partial x'}{\partial v} \\ \frac{\partial y'}{\partial x} & \frac{\partial y'}{\partial y} & \frac{\partial y'}{\partial \phi} & \frac{\partial y'}{\partial v} \\ \frac{\partial \phi'}{\partial x} & \frac{\partial \phi'}{\partial y} & \frac{\partial \phi'}{\partial \phi} & \frac{\partial \phi'}{\partial v} \\ \frac{\partial v'}{\partial x} & \frac{\partial v'}{\partial y} & \frac{\partial v'}{\partial \phi} & \frac{\partial v'}{\partial v} \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & -v\sin(\phi)\Delta t & \cos(\phi)\Delta t \\ 0 & 1 & v\cos(\phi)\Delta t & \sin(\phi)\Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
def jacob_f(x, u):
    yaw = x[2, 0]
    v = u[0, 0]
    jF = np.array([
        [1.0, 0.0, -DT * v * math.sin(yaw), DT * math.cos(yaw)],
        [0.0, 1.0,  DT * v * math.cos(yaw), DT * math.sin(yaw)],
        [0.0, 0.0, 1.0, 0.0],
        [0.0, 0.0, 0.0, 1.0]
    ])
    return jF
```

Part 2 扩展Kalman滤波定位

扩展Kalman

```python
def jacob_h():
    # Jacobian of Observation Model
    jH = np.array([
        [1, 0, 0, 0],
        [0, 1, 0, 0]
    ])

    return jH
```

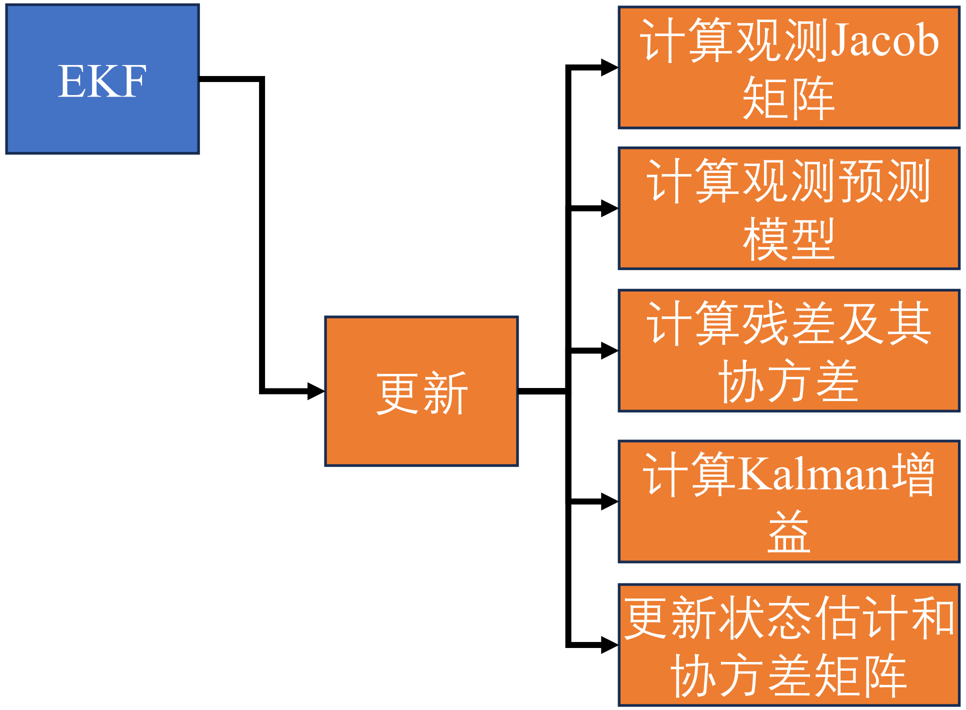
从定位传感器（GPS）获取最新的定位信息

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = g(\mathbf{x}) = \begin{bmatrix} x \\ y \end{bmatrix}$$

对定位状态（观测）求导

$$J_g = \begin{bmatrix} \frac{\partial x'}{\partial x} & \frac{\partial x'}{\partial y} & \frac{\partial x'}{\partial \phi} & \frac{\partial x'}{\partial v} \\ \frac{\partial y'}{\partial x} & \frac{\partial y'}{\partial y} & \frac{\partial y'}{\partial \phi} & \frac{\partial y'}{\partial v} \end{bmatrix}$$

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



=== Update ===

$$z_{Pred} = Hx_{Pred}$$

$$y = z - z_{Pred}$$

$$S = J_g P_{Pred}. J_g^T + R$$

$$K = P_{Pred}. J_g^T S^{-1}$$

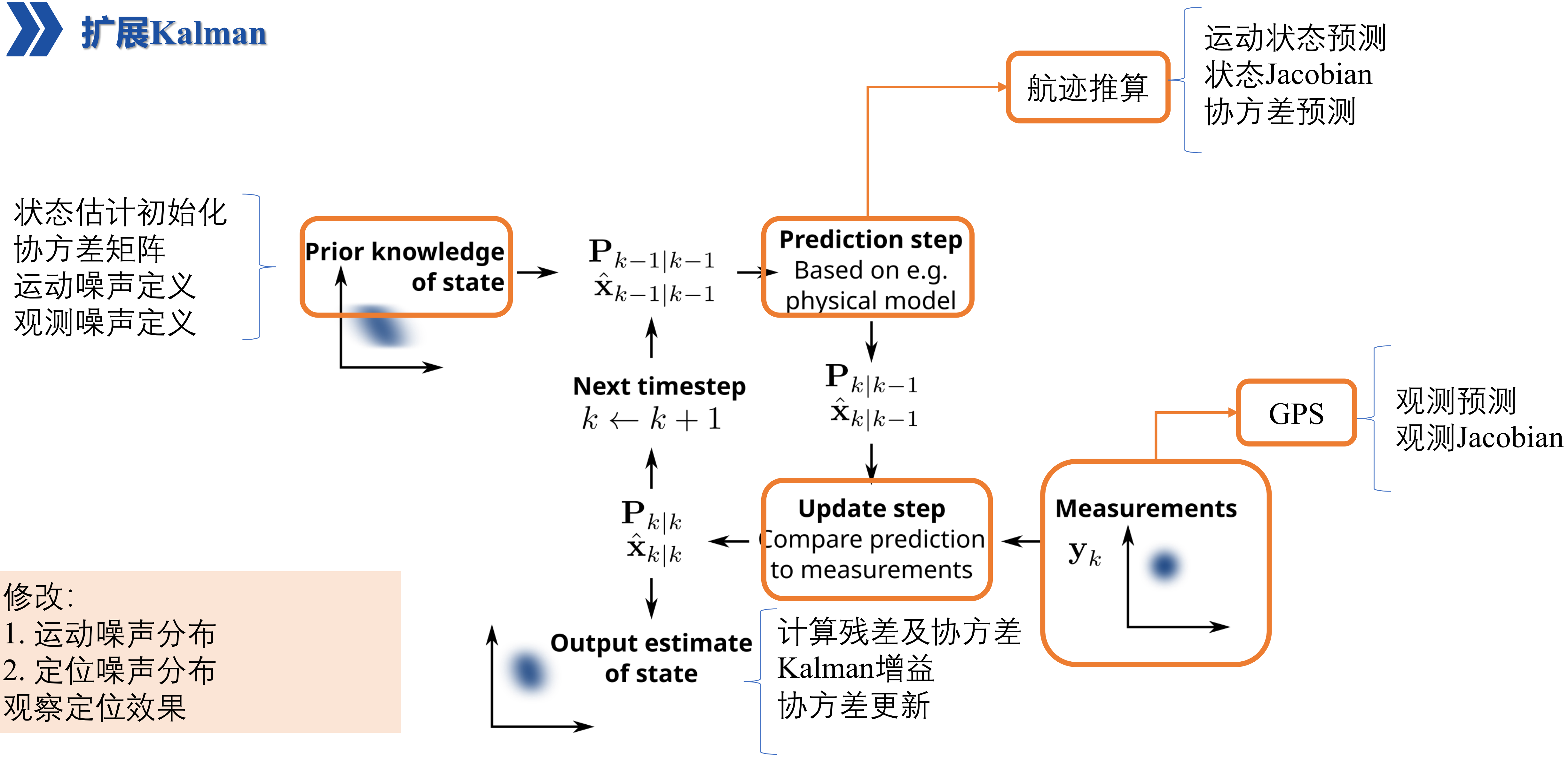
$$x_{t+1} = x_{Pred} + Ky$$

$$P_{t+1} = (I - KJ_g)P_{Pred}$$

```python
jH = jacob_h()
zPred = observation_model(xPred)
y = z - zPred
S = jH @ PPred @ jH.T + R
K = PPred @ jH.T @ np.linalg.inv(S)
xEst = xPred + K @ y
PEst = (np.eye(len(xEst)) - K @ jH) @ PPred
```

Part 2 | 扩展Kalman滤波定位

扩展Kalman



修改:
1. 运动噪声分布
2. 定位噪声分布
观察定位效果

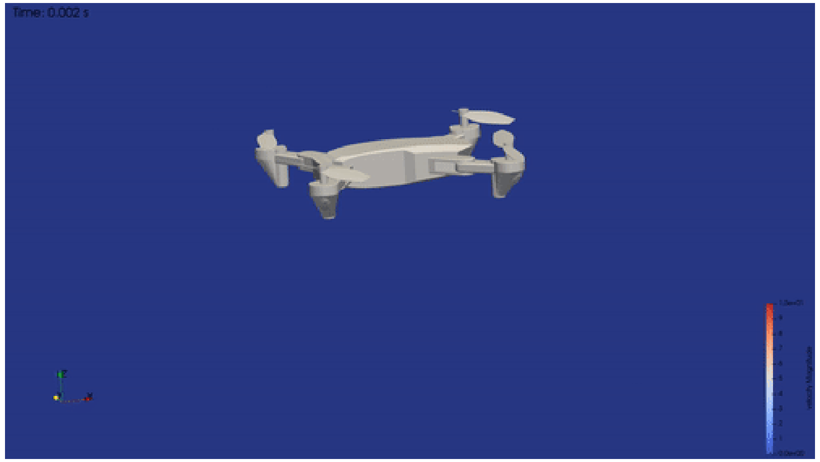
Part 2 扩展Kalman滤波定位

扩展Kalman——考虑速度矫正

车速表的工作原理： 现代汽车的车速表信号通常来自车轮转速传感器（或变速箱输出轴转速传感器）。车速表控制系统知道车辆出厂时原厂轮胎的标准尺寸（直径/周长），并据此计算车速：车速 = 车轮**转速 × 轮胎周长。**

其它影响因素：

- 更换不同直径轮胎
- 胎压
- 不同地面摩擦系数：砂石、积水、
- 里程计标定误差



在EKF的基础上，引入对速度的矫正因子。
进一步提升系统估计精度。

类似**无人机**在不同海拔、风力下的动力速度也受到空气密度影响；

船、潜水艇的速度则受水密度影响，存在一定的波动。

Part 2 扩展Kalman滤波定位

扩展Kalman—考虑速度矫正

在EKF的基础上，引入对速度的矫正因子。

2D 刚体运动模型描述了平面上物体（如机器人）的位置和姿态随时间的变化规律。用四维状态向量，位置坐标（x,y），航向角θ和速度v构成。

state vector= [x, y, θ, v]^T

state vector= [x, y, θ, v, s]^T

增加一个对v变化描述的比例因子s。

整体建模变化

$$\dot{x} = v\cos(\phi) \qquad \dot{x} = vs\cos(\phi)$$

$$\dot{y} = v\sin(\phi) \qquad \dot{y} = vs\sin(\phi)$$

$$\dot{\phi} = \omega \qquad \dot{\phi} = \omega$$

航向角不变

1. 运动模型：计算速度时需要考虑s

$$\mathbf{x}_{t+1} = f(\mathbf{x}_t, \mathbf{u}_t) = F\mathbf{x}_t + B\mathbf{u}_t$$

$$F = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad B = \begin{bmatrix} cos(\phi)\Delta t & 0 \\ sin(\phi)\Delta t & 0 \\ 0 & \Delta t \\ 1 & 0 \end{bmatrix}$$

$$F = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} cos(\phi)\Delta ts & 0 \\ sin(\phi)\Delta ts & 0 \\ 0 & \Delta t \\ 1 & 0 \\ 0 & 0 \end{bmatrix}$$

齐次项

Part 2 扩展Kalman滤波定位

扩展Kalman—考虑速度矫正

在EKF的基础上，引入对速度的矫正因子。

2D 刚体运动模型描述了平面上物体（如机器人）的位置和姿态随时间的变化规律。用四维状态向量，位置坐标（x,y），航向角θ和速度v构成。

state vector= [x, y, θ, v]^T

state vector= [x, y, θ, v, s]^T

增加一个对v变化描述的比例因子s。

2. 运动状态预测的Jacbian矩阵

$$J_f = \begin{bmatrix} \frac{\partial x'}{\partial x} & \frac{\partial x'}{\partial y} & \frac{\partial x'}{\partial \phi} & \frac{\partial x'}{\partial v} \\ \frac{\partial y'}{\partial x} & \frac{\partial y'}{\partial y} & \frac{\partial y'}{\partial \phi} & \frac{\partial y'}{\partial v} \\ \frac{\partial \phi'}{\partial x} & \frac{\partial \phi'}{\partial y} & \frac{\partial \phi'}{\partial \phi} & \frac{\partial \phi'}{\partial v} \\ \frac{\partial v'}{\partial x} & \frac{\partial v'}{\partial y} & \frac{\partial v'}{\partial \phi} & \frac{\partial v'}{\partial v} \end{bmatrix}$$

$$J_f = \begin{bmatrix} \frac{\partial x'}{\partial x} & \frac{\partial x'}{\partial y} & \frac{\partial x'}{\partial \phi} & \frac{\partial x'}{\partial v} & \frac{\partial x'}{\partial s} \\ \frac{\partial y'}{\partial x} & \frac{\partial y'}{\partial y} & \frac{\partial y'}{\partial \phi} & \frac{\partial y'}{\partial v} & \frac{\partial y'}{\partial s} \\ \frac{\partial \phi'}{\partial x} & \frac{\partial \phi'}{\partial y} & \frac{\partial \phi'}{\partial \phi} & \frac{\partial \phi'}{\partial v} & \frac{\partial \phi'}{\partial s} \\ \frac{\partial v'}{\partial x} & \frac{\partial v'}{\partial y} & \frac{\partial v'}{\partial \phi} & \frac{\partial v'}{\partial v} & \frac{\partial v'}{\partial s} \\ \frac{\partial s'}{\partial x} & \frac{\partial s'}{\partial y} & \frac{\partial s'}{\partial \phi} & \frac{\partial s'}{\partial v} & \frac{\partial s'}{\partial s} \end{bmatrix}$$

整体建模变化

$$= \begin{bmatrix} 1 & 0 & -v\sin(\phi)\Delta t & \cos(\phi)\Delta t \\ 0 & 1 & v\cos(\phi)\Delta t & \sin(\phi)\Delta t \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 0 & -vs\sin(\phi)\Delta t & s\cos(\phi)\Delta t & \cos(\phi)v\Delta t \\ 0 & 1 & vs\cos(\phi)\Delta t & s\sin(\phi)\Delta t & v\sin(\phi)\Delta t \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Part 2 扩展Kalman滤波定位

扩展Kalman—考虑速度矫正

在EKF的基础上，引入对速度的矫正因子。

2D 刚体运动模型描述了平面上物体（如机器人）的位置和姿态随时间的变化规律。用四维状态向量，位置坐标（x,y），航向角θ和速度v构成。

state vector= [x, y, θ, v]^T

state vector= [x, y, θ, v, s]^T

增加一个对v变化描述的比例因子s。

整体建模变化

3. 运动状态预测的协方差Q

```
# Covariance for EKF simulation
Q = np.diag([
    0.1,  # variance of location on x-axis
    0.1,  # variance of location on y-axis
    np.deg2rad(1.0),  # variance of yaw angle
    0.4,  # variance of velocity
    0.1  # variance of scale factor
]) ** 2  # predict state covariance
R = np.diag([0.1, 0.1]) ** 2  # Observation x,y position covariance
```

Part 2 | 扩展Kalman滤波定位

扩展Kalman—考虑速度矫正

总结下来，对计算产生的影响主要还是EKF的预测分支

在EKF的基础上，引入对速度的矫正因子。

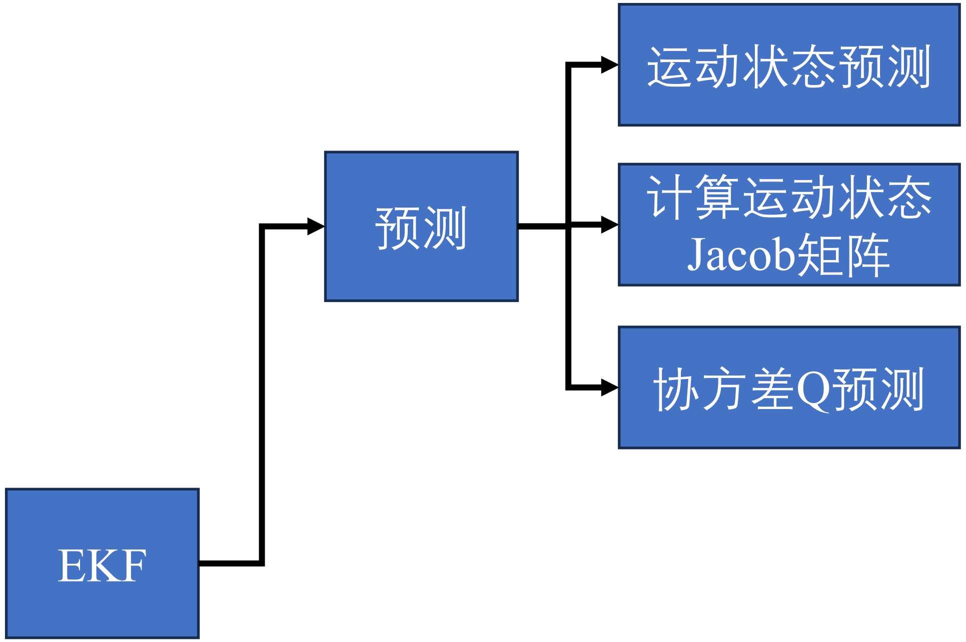
2D 刚体运动模型描述了平面上物体（如机器人）的位置和姿态随时间的变化规律。用四维状态向量，位置坐标（x,y），航向角θ和速度v构成。

state vector= [x, y, θ, v]^T

state vector= [x, y, θ, v, s]^T

增加一个对v变化描述的比例因子s。

整体建模变化




```
xPred = motion_model(xEst, u)
jF = jacob_f(xEst, u)
PPred = jF @ PEst @ jF.T + Q
```

Part 2 扩展Kalman滤波定位

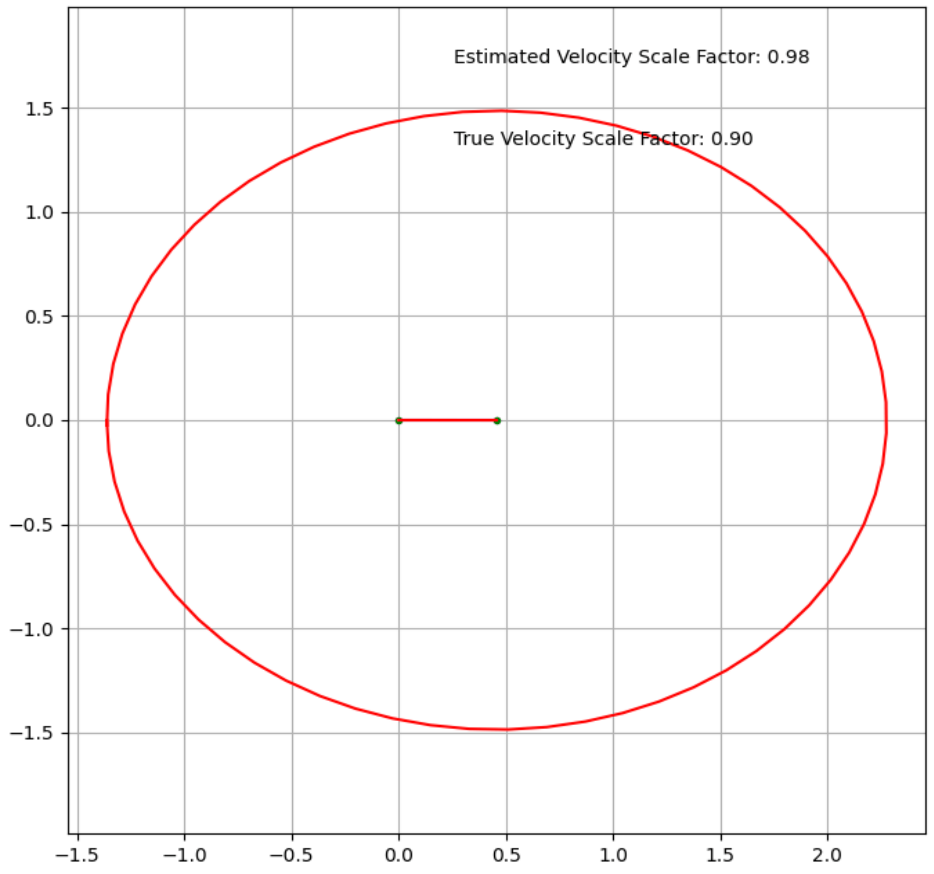
扩展Kalman—考虑速度矫正

实现：
1. 考虑对速度进行矫正，并设置速度矫正因子s

修改：
2. 运动噪声分布
3. 定位噪声分布

噪声过大会淹没速度波动

观察定位效果，以及对s的估计误差。



间接影响：

虽然车辆的运动状态（速度的放缩）不影响定位传感器的观测。但在EKF的更新阶段，观测模型需要从运动模型中提取运动估计坐标，需要进行齐次转化，满足矩阵乘法。

观测模型：

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

观测模型的Jacobian：

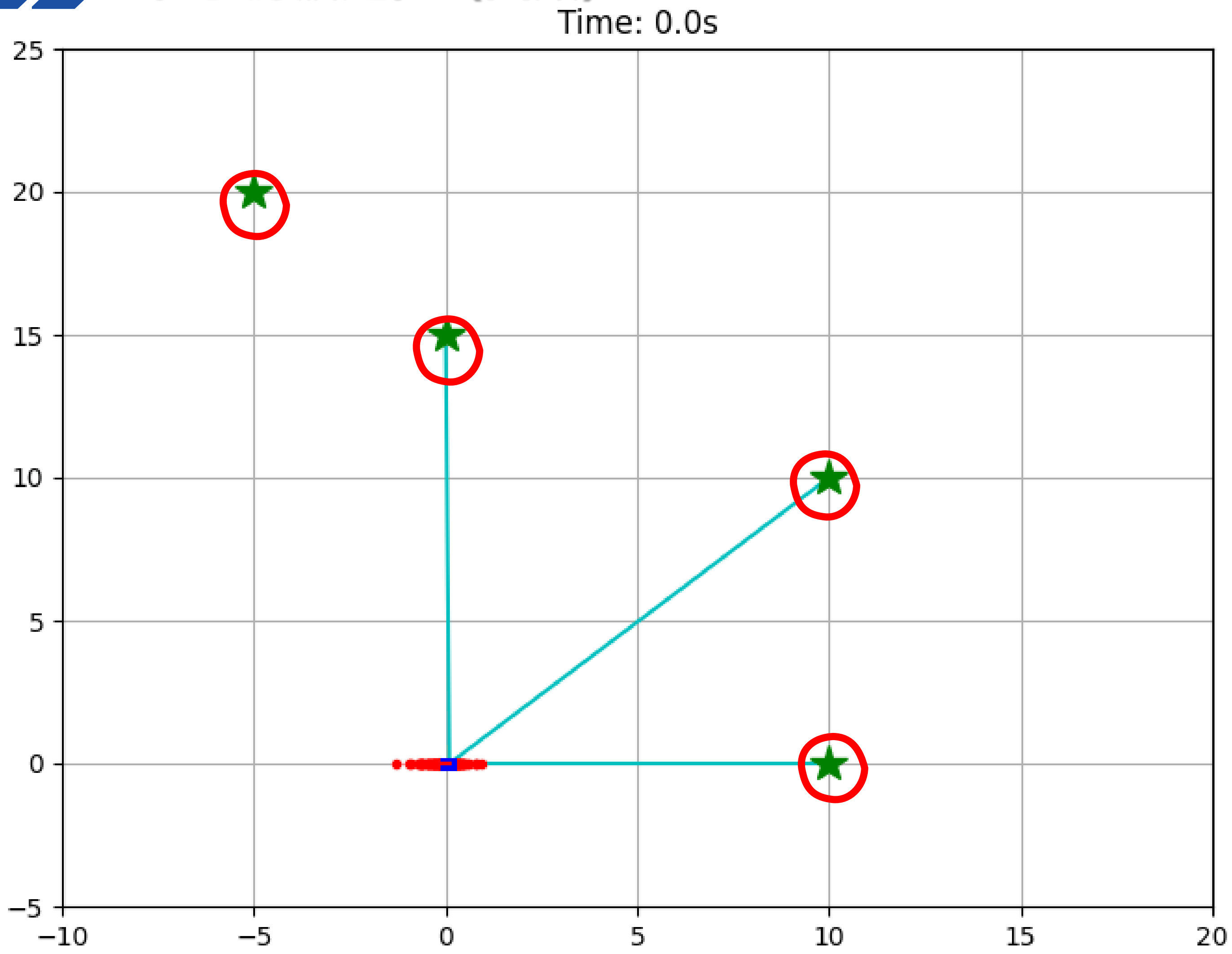
$$J_g = \begin{bmatrix} \frac{\partial x'}{\partial x} & \frac{\partial x'}{\partial y} & \frac{\partial x'}{\partial \phi} & \frac{\partial x'}{\partial v} & \frac{\partial x'}{\partial s} \\ \frac{\partial y'}{\partial x} & \frac{\partial y'}{\partial y} & \frac{\partial y'}{\partial \phi} & \frac{\partial y'}{\partial v} & \frac{\partial y'}{\partial s} \end{bmatrix}$$

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

目录

Contents

粒子滤波定位（回顾）



- matplot绘制并实时更新算法结果。
- 蓝色：GT轨迹，基本与红色轨迹重合。
- 黑色：航位推算轨迹（存在累积误差）
- 红色：粒子滤波算法结果（实时粒子，历史轨迹）
- 绿色五角星：为锚点传感器（如RFID、UWB）位置，并假设机器人能够通过传感器对固定信标进行测距（即为动态刷新时当前节点与五角星的连线）。基础为三角定位。
- 定位过程中，本地节点有策略的选择近邻锚点。

粒子滤波定位算法，一方面基于定位传感器测量自身位置（带噪），另一方面，预测过程通过随机粒子进行潜在情况的预测，再通过统计后验筛选粒子。

粒子滤波定位

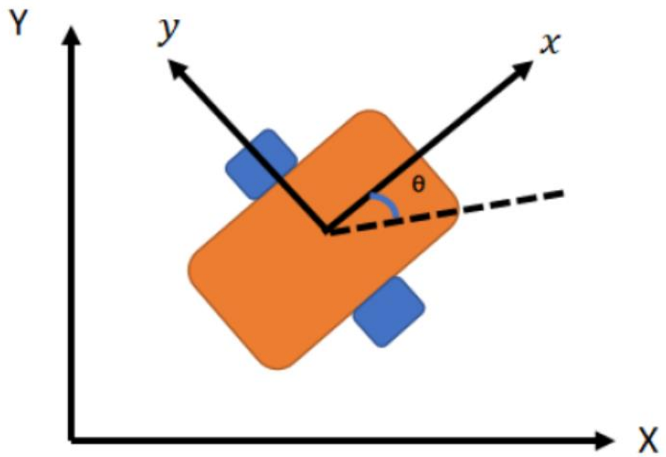
2D运动模型

- 2D 刚体运动模型描述了平面上物体（如机器人）的位置和姿态随时间的变化规律。
- 在平面运动中，刚体的状态由位置（x, y）和航向角Yaw（θ） 确定，构成 3 维状态向量：

state = [x, y, θ]^T

- 在 2D 刚体运动模型中加入速度状态（即扩展为 4 维状态向量）可以更准确地描述系统动态特性，特别是在处理非连续控制输入或需要预测未来轨迹的场景中。

state = [x, y, θ, v]^T

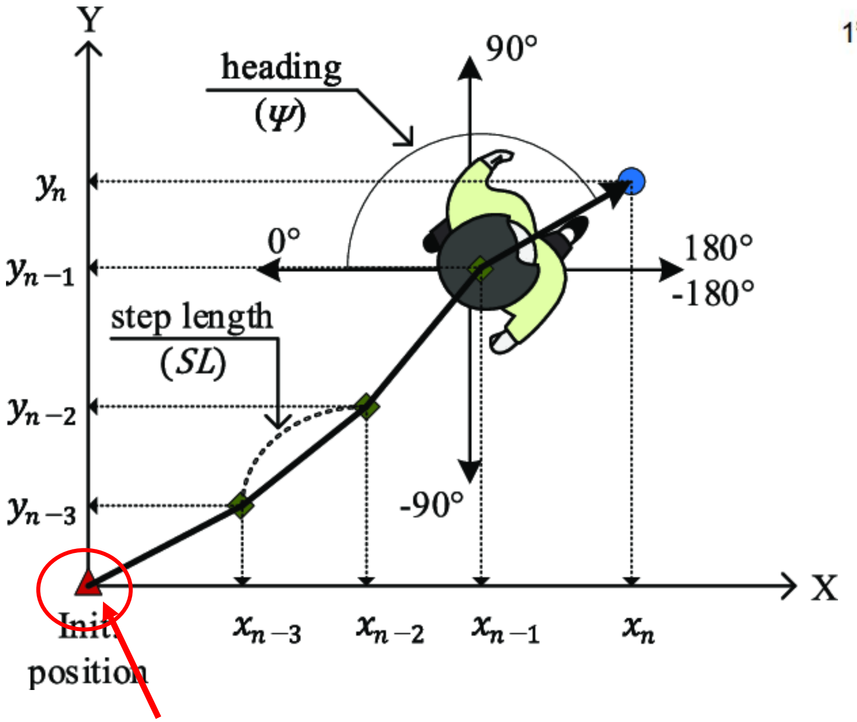


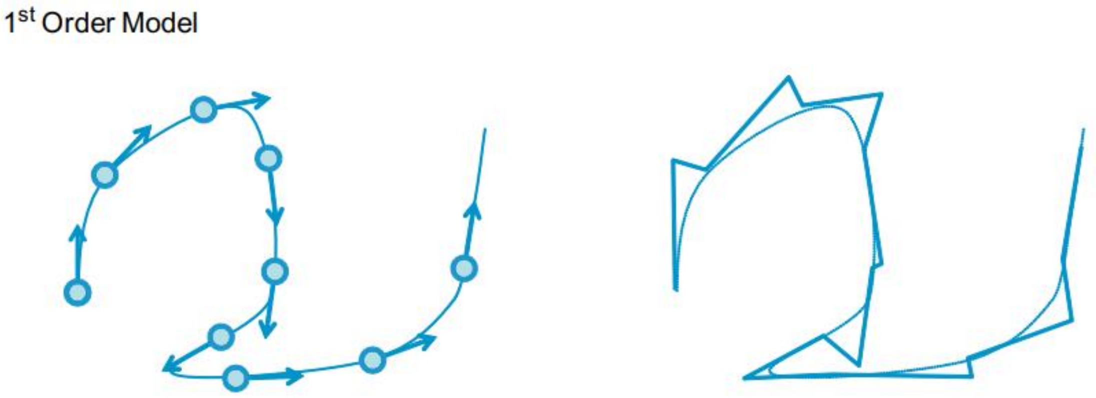
时序变化关系：

$$x(i+1)=x(i)+v\cdot\Delta t\cdot\cos(\theta(i))$$
$$y(i+1)=y(i)+v\cdot\Delta t\cdot\sin(\theta(i))$$
$$\theta(i+1)=\theta(i)+r\cdot\Delta t$$

粒子滤波定位

航迹推算Dead Reckoning





若初始位置为 (x0,y0)，某段时间内移动距离为 di、航向角为 θi，则新位置 (xn,yn) 为：

$$Xn = X0 + \sum_{k=0}^{n} d_i cos\theta_i$$

$$Yn = Y0 + \sum_{k=0}^{n} d_i sin\theta_i$$

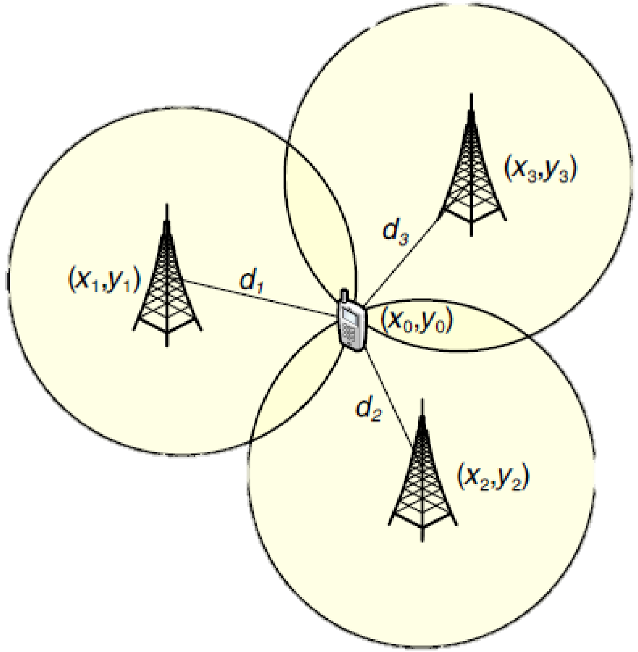
通过**控制矩阵**来描述，DT是采样时间，Xt=[x,y,yaw,v]是机器人状态向量[2,0]是朝向

```
B = np.array([[DT * math.cos(x[2, 0]), 0],
              [DT * math.sin(x[2, 0]), 0],
              [0.0, DT],
              [1.0, 0.0]])
x = F.dot(x) + B.dot(u)
```

- **利用现在物体位置及速度推定未来位置方向的航海技术，容易受噪声误差影响，形成累积误差。**
- **DR一般依赖自身传感器进行测角（磁力计、陀螺仪）和测速（轮速计、加速度计），不依赖外部信息。**

粒子滤波定位

三角定位



- 已知三角形的三个顶点中两个顶点的位置，以及目标与这两个顶点的距离或角度关系，可通过几何计算确定第三个顶点（目标）的位置。
- **平面场景**：至少需要 2 个参考点 + 目标到两点的距离（或角度）。
- **三维场景**：至少需要 3 个参考点（形成立体三角），才能确定目标的三维坐标（如 GPS 定位）。

基于距离的三角定位（Range-based）
通过测量目标到多个参考点的距离，利用 “圆（或球）的交点” 确定位置。

在时间同步的情况下，距离一般通过无线电信号收发的时间（信号内容带有时间戳）t，即光速一定v，距离d=vt可得。

假设：机器人能够通过传感器获取自身到锚点的**距离**，从而利用三角定位估计自身位置。因此，设定若干固定点坐标，假设短时间内它们的绝对位置静止。

```python
rf_id = np.array([[10.0, 0.0],
                  [10.0, 10.0],
                  [0.0, 15.0],
                  [-5.0, 20.0]])
```

```python
for i in range(len(rf_id[:, 0])):
    dx = x_true[0, 0] - rf_id[i, 0]
    dy = x_true[1, 0] - rf_id[i, 1]
    d = math.hypot(dx, dy)
```

Part 6 粒子滤波定位

粒子滤波定位

核心思想是通过大量“粒子”（模拟目标可能位置的样本）来近似目标的概率分布，进而根据观测数据不断更新粒子权重，最终筛选出最可能的目标位置。

依赖： 机器人运动模型、观测结果（即传感器测量信息）

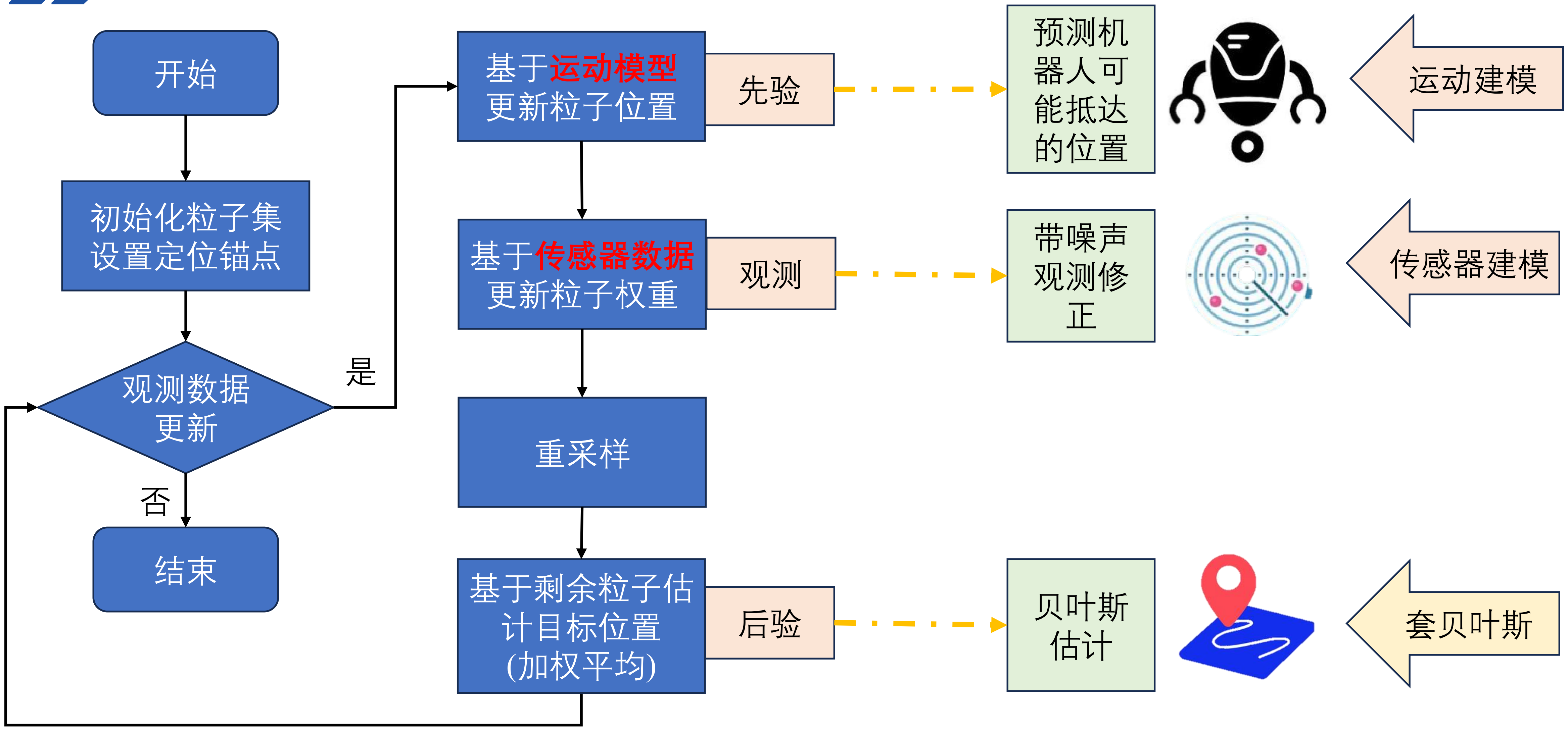
流程：

- **预测阶段**：根据系统运动模型（如目标的速度、加速度），对粒子进行“移动”，模拟目标可能的位置变化（引入随机噪声，体现运动不确定性）。
- **更新阶段**：结合传感器观测数据（如**距离**），计算每个粒子与观测数据的匹配度（似然概率），并以此更新粒子权重（距离越近，匹配度越高，权重越大）。
- **重采样阶段**：为避免权重过低的粒子浪费计算资源，保留高权重粒子并复制，低权重粒子被淘汰，使粒子集合始终聚焦于高概率区域。

Part 6 粒子滤波定位

Particle filter localization

粒子滤波定位



粒子滤波定位

质点运动模型
航迹推算
Dead Reckoning

机器人状态-数据结构

```
# State Vector [x y yaw v]'
x_est = np.zeros((4, 1))
x_true = np.zeros((4, 1))
```

恒定速度和角速度的圆周运动模拟

```
v = 1.0  # [m/s]
yaw_rate = 0.1  # [rad/s]
u = np.array([[v, yaw_rate]]).T
```

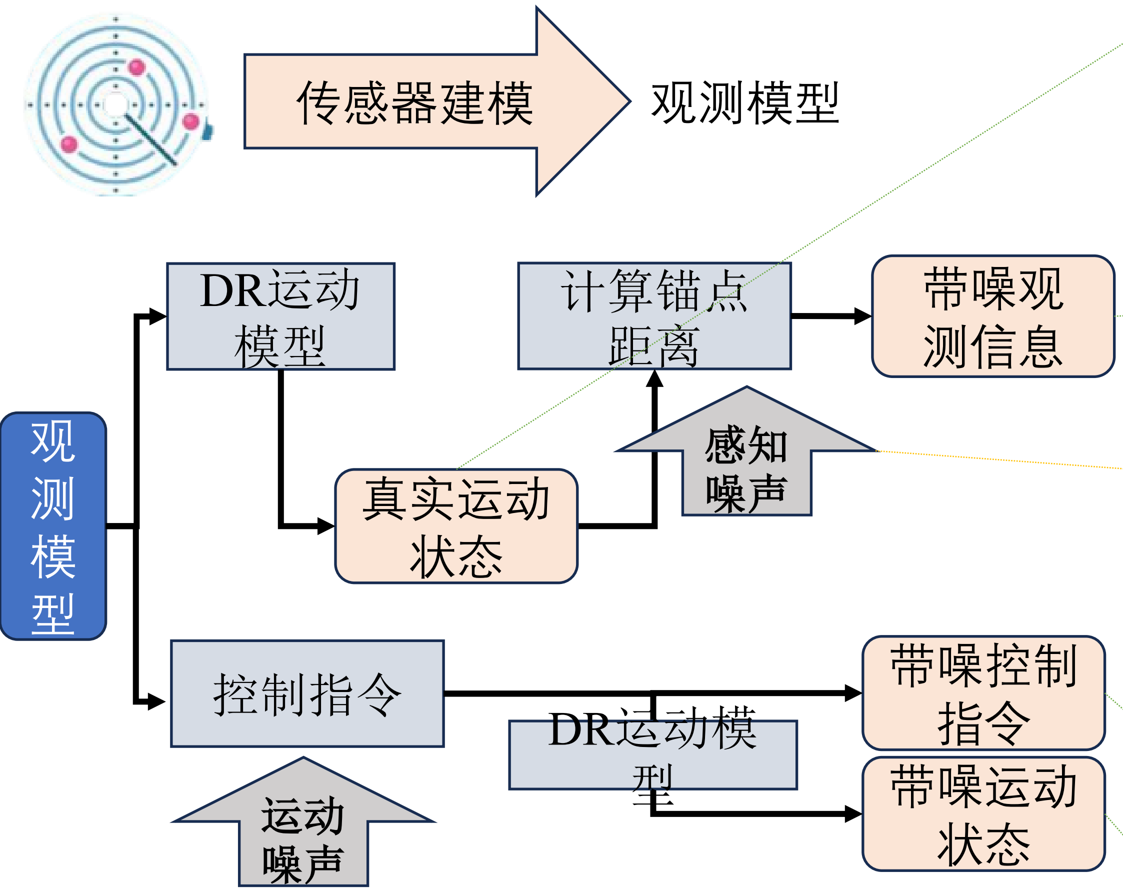
```
def motion_model(x, u):
    # 状态转移矩阵F（4x4）: [x, y, yaw, v]的状态更新
    F = np.array([[1.0, 0, 0, 0],
                  [0, 1.0, 0, 0],
                  [0, 0, 1.0, 0],
                  [0, 0, 0, 0]])
    # 控制矩阵B（4x2）: 将控制输入转换为状态变化
    B = np.array([[DT * math.cos(x[2, 0]), 0],  # x方向位移 = 速度*时间*cos(航向角)
                  [DT * math.sin(x[2, 0]), 0],  # y方向位移 = 速度*时间*sin(航向角)
                  [0.0, DT],  # 航向角变化 = 角速度*时间
                  [1.0, 0.0]])  # 速度保持不变
    x = F.dot(x) + B.dot(u)  # 状态更新: x_new = F·x + B·u
    return x
```

原速度 + 当前时间片增量

Part 6 粒子滤波定位

Particle filter localization

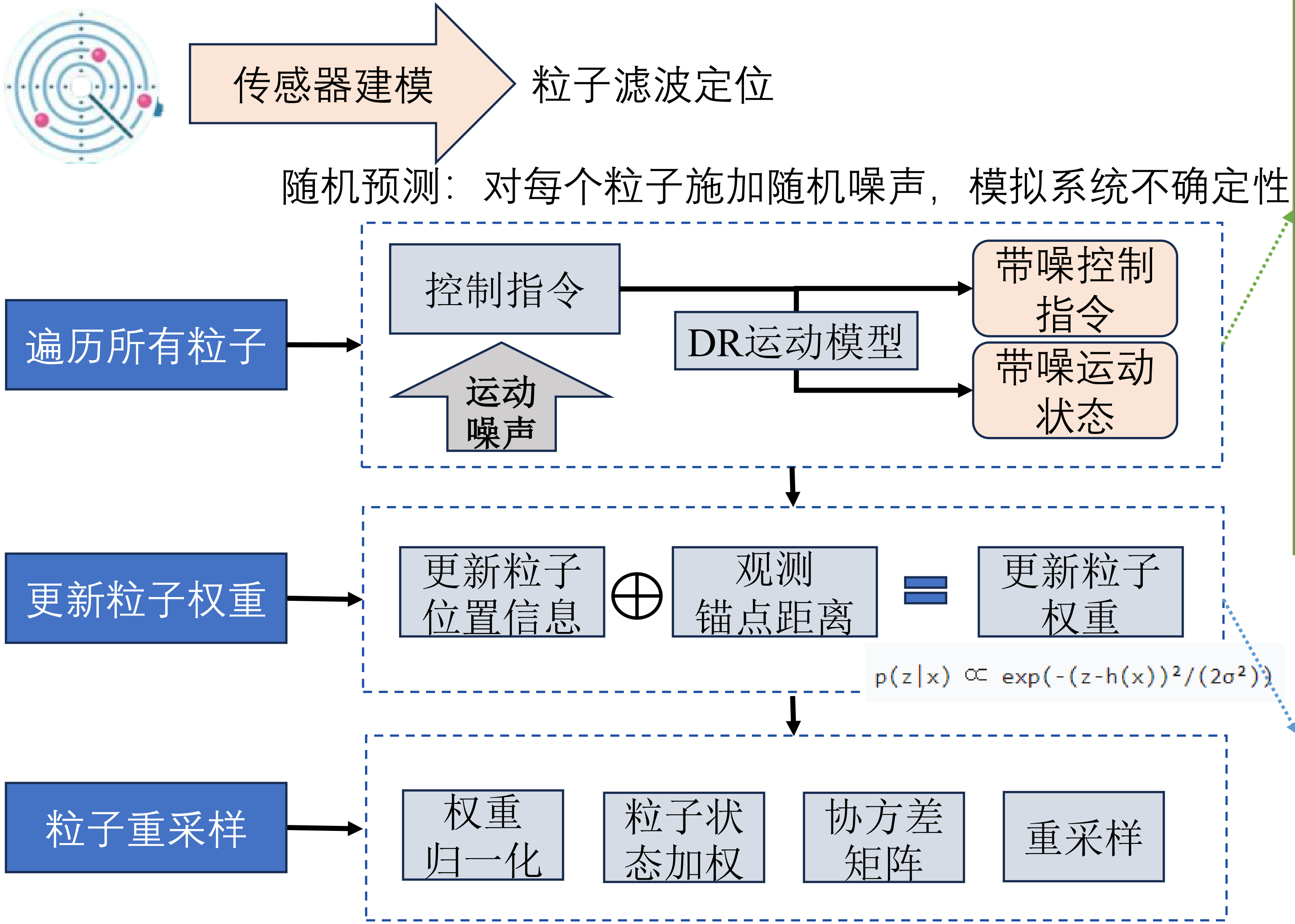
粒子滤波定位




```python
def observation(x_true, xd, u, rf_id):
    x_true = motion_model(x_true, u)  # 真实状态更新
    z = np.zeros((0, 3))  # 观测数据：[距离, 地标x, 地标y]

    for i in range(len(rf_id[:, 0])):
        # 计算真实距离
        dx = x_true[0, 0] - rf_id[i, 0]
        dy = x_true[1, 0] - rf_id[i, 1]
        d = math.hypot(dx, dy)  # 欧氏距离

        if d <= MAX_RANGE:  # 仅保留有效范围内的观测
            dn = d + np.random.randn() * Q_sim[0, 0] **0.5  # 加入观测噪声
            zi = np.array([[dn, rf_id[i, 0], rf_id[i, 1]]])
            z = np.vstack((z, zi))  # 堆叠观测数据

    # 控制输入加入噪声（模拟实际控制误差）
    ud1 = u[0, 0] + np.random.randn() * R_sim[0, 0]** 0.5
    ud2 = u[1, 0] + np.random.randn() * R_sim[1, 1] **0.5
    ud = np.array([[ud1, ud2]]).T

    xd = motion_model(xd, ud)  # 航迹推演（Dead Reckoning）状态更新
    return x_true, z, xd, ud
```

Part 6 | 粒子滤波定位

粒子滤波定位



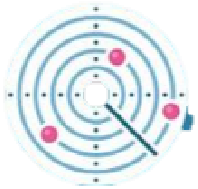

```python
for ip in range(NP):
    x = np.array([px[:, ip]]).T  # 获取第ip个粒子的状态
    w = pw[0, ip]  # 获取对应权重

    # 为控制输入添加噪声（模拟控制不确定性）
    ud1 = u[0, 0] + np.random.randn() * R[0, 0] ** 0.5
    ud2 = u[1, 0] + np.random.randn() * R[1, 1] ** 0.5
    ud = np.array([[ud1, ud2]]).T

    # 使用带噪声的控制输入更新粒子状态
    x = motion_model(x, ud)

    # 更新粒子存储和权重
    px[:, ip] = x[:, 0]
    pw[0, ip] = w
```

```python
for i in range(len(z[:, 0])):  # 对每个观测数据
    dx = x[0, 0] - z[i, 1]  # 粒子位置与地标x的差值
    dy = x[1, 0] - z[i, 2]  # 粒子位置与地标y的差值
    pre_z = math.hypot(dx, dy)  # 粒子预测的与地标的距离
    dz = pre_z - z[i, 0]  # 预测距离与实际观测距离的残差

    # 基于残差计算高斯似然，更新粒子权重
    w = w * gauss_likelihood(dz, math.sqrt(Q[0, 0]))
```

Part 6 粒子滤波定位

Particle filter localization

粒子滤波定位



传感器建模 → 粒子滤波定位

随机预测：对每个粒子施加随机噪声，模拟系统不确定性

遍历所有粒子 → 控制指令、运动噪声、DR运动模型 → 带噪控制指令、带噪运动状态

更新粒子权重 → 更新粒子位置信息 ⊕ 观测锚点距离 = 更新粒子权重

```
p(z|x) ∝ exp(-(z-h(x))²/(2σ²))
```

粒子重采样 → 权重归一化、粒子状态加权、协方差矩阵、重采样

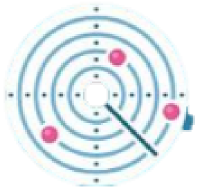
```
pw = pw / pw.sum()  # 确保所有粒子权重之和为1
```

```
x_est = px.dot(pw.T)  # 加权平均：粒子状态的加权和
p_est = calc_covariance(x_est, px, pw)  # 计算估计的协方差矩阵
```

权重pw<1, 分母为pw的平方和 $N_{\text{eff}} = \frac{1}{\sum_{i=1}^{NP} w_i^2}$

```
N_eff = 1.0 / (pw.dot(pw.T))[0, 0]  # 有效粒子数
if N_eff < NTh:  # 若有效粒子数低于阈值（通常为NP/2）
    px, pw = re_sampling(px, pw)  # 执行重采样
```

粒子滤波定位



传感器建模 → 噪声

```
Q = np.diag([0.2]) ** 2  # 对粒子重采样时的高斯似然， 对角阵 0.2^2
R = np.diag([2.0, np.deg2rad(40.0)]) ** 2  # 噪声加在 粒子采样时的 控制输入 上

#  Simulation parameter
Q_sim = np.diag([0.2]) ** 2   #感知噪声 噪声加在到锚点的距离估计上
R_sim = np.diag([1.0, np.deg2rad(30.0)]) ** 2  # 噪声加在对机器人的  控制输入 上
```

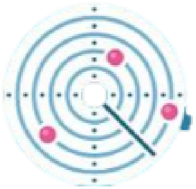
仿真模拟运动过程 → 锚点定位 ← 定位噪声 Q_sim

仿真模拟运动过程 → 运动控制 ← 控制噪声 R_sim

粒子滤波定位过程 → 粒子运动 ← 控制噪声 R

粒子运动 → 权重更新 ← Q观测不确定性

粒子滤波定位



传感器建模 → 噪声

实验要求：调整定位锚点位置（改变分布），模拟不同移动轨迹，如双曲线、椭圆、梨形等机器人运动轨迹；并分析粒子采样策略、噪声分布对算法的影响。

矩阵	类型	作用对象	物理意义	在代码中的使用
R	控制噪声	运动模型（控制输入 u）	滤波算法假设的**控制执行误差**（如电机精度不足导致的速度 / 角速度偏差）	在 pf_localization() 中为每个粒子的控制输入添加噪声
R_sim	控制噪声	运动模型（控制输入 u）	仿真环境中**真实的控制执行误差**	在 observation() 中生成带噪声的控制输入 ud
Q	观测噪声	观测模型（距离测量）	滤波算法假设的**距离观测误差**（如 TOF 传感器的测量噪声）	在 gauss_likelihood() 中计算粒子权重时作为观测噪声方差
Q_sim	观测噪声	观测模型（距离测量）	仿真环境中**真实的距离观测误差**	在 observation() 中生成带噪声的锚点观测

粒子滤波定位算法总结：

目标运动（如位置、速度）和传感器观测（如距离、角度测量）都存在噪声（随机性），导致无法通过确定性模型（如航迹推演）精确预测位置。

粒子滤波通过用**大量粒子模拟这些随机性**，每个粒子代表一种"可能的运动状态"（包含位置、速度等信息）。

粒子集的分布（数量、位置）理论上需要覆盖所有"合理的可能状态"。当粒子数趋于无穷时，粒子集的分布会逼近真实的**后验概率分布**（贝叶斯滤波的理想解）。

每次观测通过**权重计算**（粒子与观测值的匹配度）筛选出"更可能符合真实状态"的粒子（权重高）；权重高的粒子被复制保留，权重低的被淘汰，粒子集逐渐聚焦到真实状态附近（后验概率收敛）。

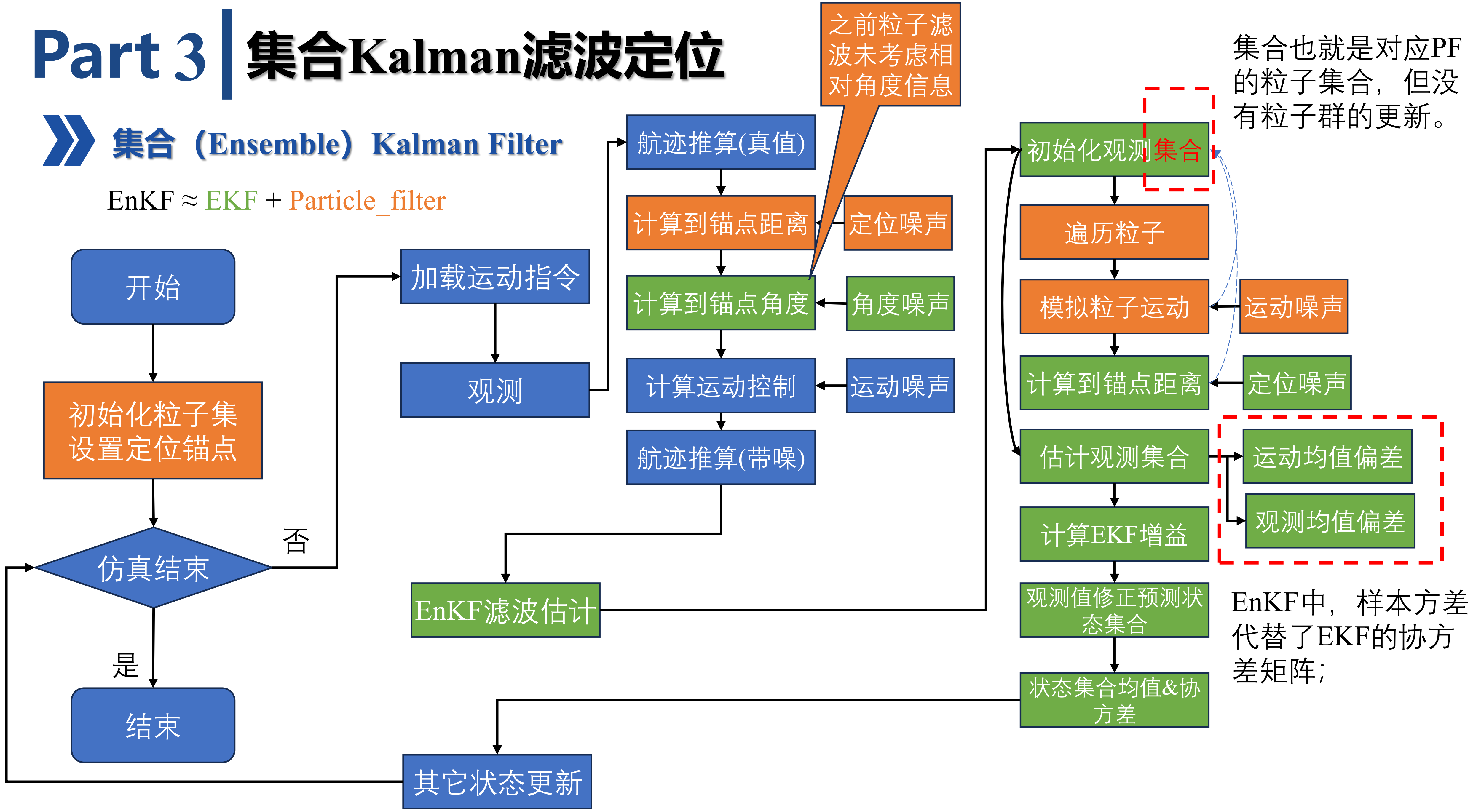
目 录

Contents

Part 3 | 集合Kalman滤波定位

集合（Ensemble）Kalman Filter

EnKF ≈ EKF + Particle_filter

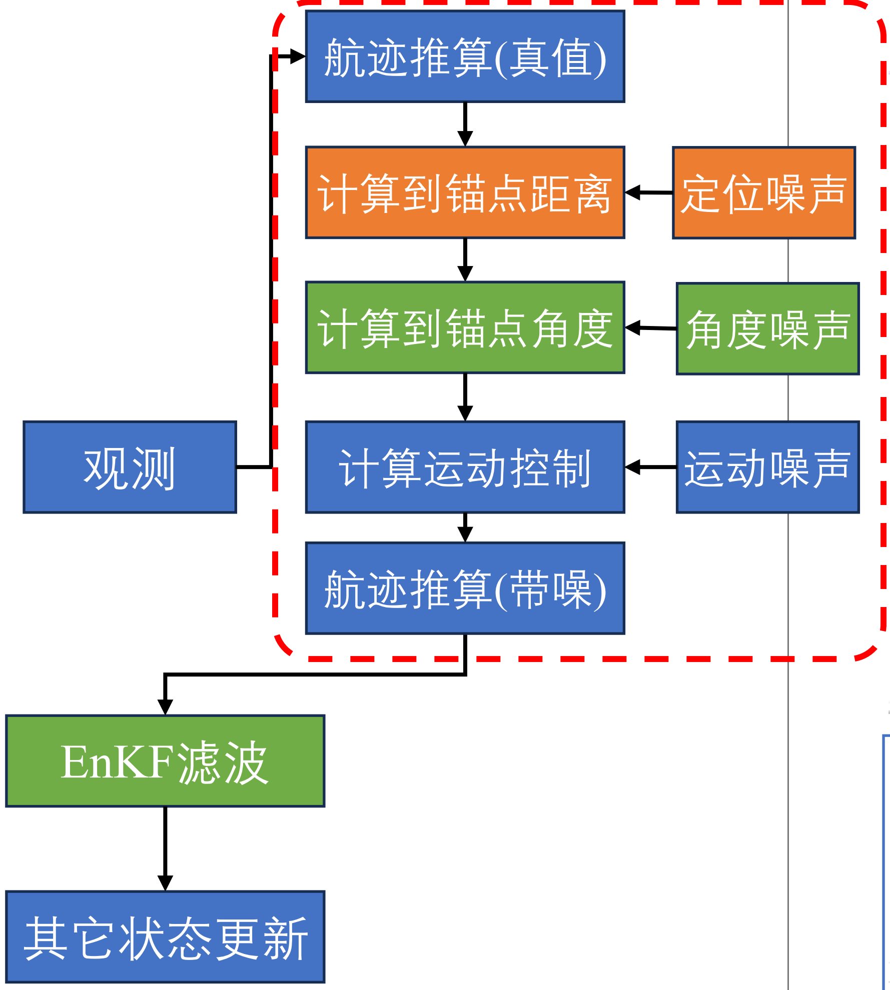


通过流程图不同配色来说明集成的各部分来自原先的哪个算法，同时维持相同操作（蓝色）

Part 3 集合Kalman滤波定位

EnKF

EnKF ≈ EKF + Particle_filter

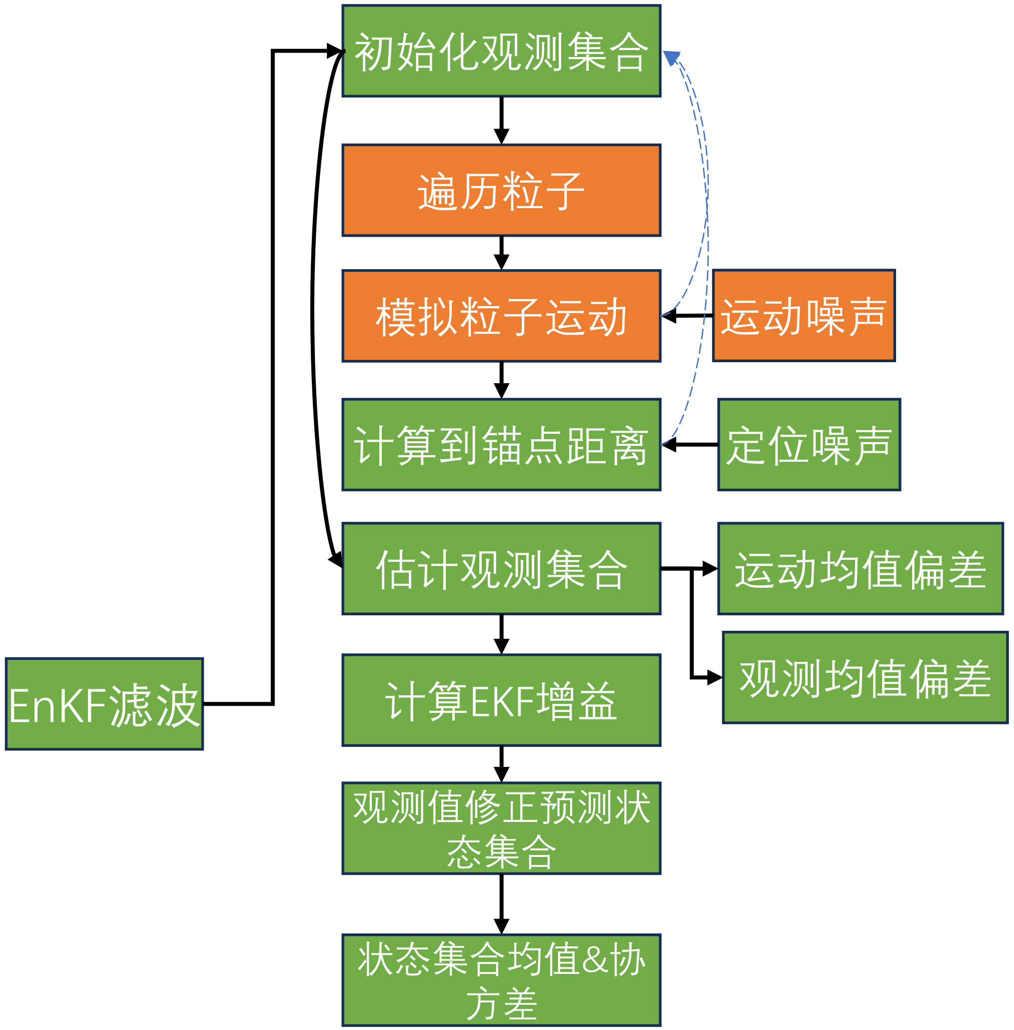



```python
def observation(xTrue, xd, u, RFID):
    xTrue = motion_model(xTrue, u)

    z = np.zeros((0, 4))

    for i in range(len(RFID[:, 0])):

        dx = RFID[i, 0] - xTrue[0, 0]
        dy = RFID[i, 1] - xTrue[1, 0]
        d = math.hypot(dx, dy)
        angle = pi_2_pi(math.atan2(dy, dx) - xTrue[2, 0])
        if d <= MAX_RANGE:
            dn = d + np.random.randn() * Q_sim[0, 0] ** 0.5  # add noise
            angle_with_noise = angle + np.random.randn() * Q_sim[1, 1] ** 0.5
            zi = np.array([dn, angle_with_noise, RFID[i, 0], RFID[i, 1]])
            z = np.vstack((z, zi))

    # add noise to input
    ud = np.array([[
        u[0, 0] + np.random.randn() * R_sim[0, 0] ** 0.5,
        u[1, 0] + np.random.randn() * R_sim[1, 1] ** 0.5]]).T

    xd = motion_model(xd, ud)
    return xTrue, z, xd, ud
```

Part 3 | 集合Kalman滤波定位

EnKF

EnKF ≈ EKF + Particle_filter

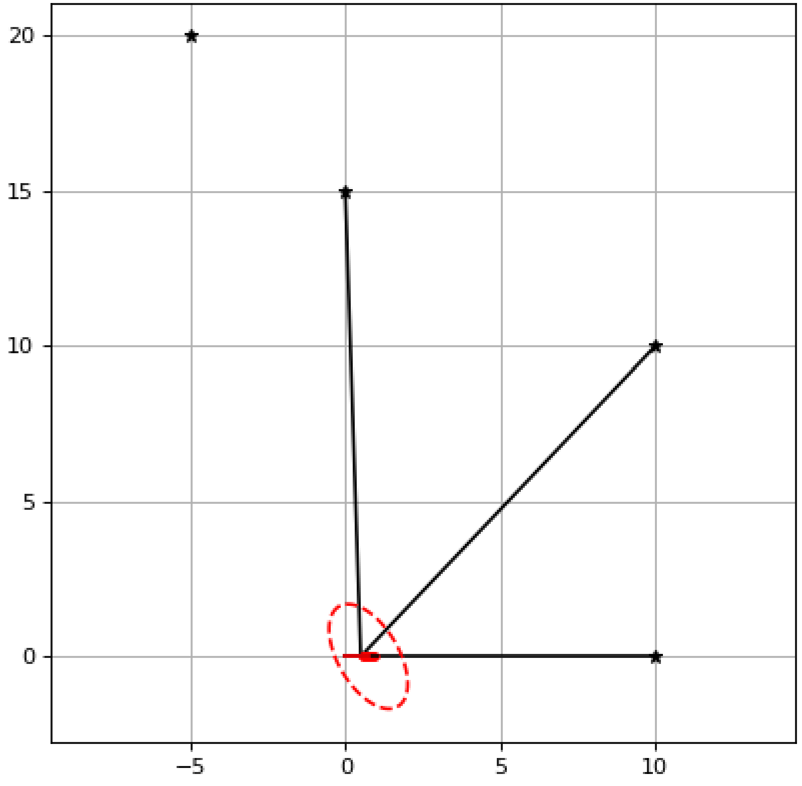



```python
def enkf_localization(px, z, u):
    """
    Localization with Ensemble Kalman filter
    """
    pz = np.zeros((z.shape[0] * 2, NP))  # Particle store of z
    for ip in range(NP):
        x = np.array([px[:, ip]]).T

        #  Predict with random input sampling
        ud1 = u[0, 0] + np.random.randn() * R_sim[0, 0] ** 0.5
        ud2 = u[1, 0] + np.random.randn() * R_sim[1, 1] ** 0.5
        ud = np.array([[ud1, ud2]]).T
        x = motion_model(x, ud)
        px[:, ip] = x[:, 0]
        z_pos = observe_landmark_position(x, z)
        pz[:, ip] = z_pos[:, 0]

    x_ave = np.mean(px, axis=1)
    x_dif = px - np.tile(x_ave, (NP, 1)).T

    z_ave = np.mean(pz, axis=1)
    z_dif = pz - np.tile(z_ave, (NP, 1)).T

    U = 1 / (NP - 1) * x_dif @ z_dif.T
    V = 1 / (NP - 1) * z_dif @ z_dif.T

    K = U @ np.linalg.inv(V)  # Kalman Gain

    z_lm_pos = z[:, [2, 3]].reshape(-1, )

    px_hat = px + K @ (np.tile(z_lm_pos, (NP, 1)).T - pz)

    xEst = np.average(px_hat, axis=1).reshape(4, 1)
    PEst = calc_covariance(xEst, px_hat)

    return xEst, PEst, px_hat
```

Part 3 集合Kalman滤波定位

EnKF

EnKF ≈ EKF + Particle_filter



	EKF	PF	EnKF
非线性处理	局部线性化（雅可比矩阵）	直接处理（粒子非线性映射）	直接处理（集合非线性映射）
状态表示	单一向量 + 协方差矩阵（高斯假设）	带权重粒子集（任意分布）	无权重集合样本（近似高斯）
噪声适应性	仅高斯噪声	任意噪声	高斯 / 弱非高斯噪声
缺点	线性化误差（强非线性失效）	粒子退化、计算量大	强非高斯噪声下精度下降
计算复杂度	低，O (n²) n=状态维度	高（随粒子数增加），O (N・n) N=粒子规模	中等（随集合大小增加），O (M・n²) M=集合大小,<N
精度	一般，线性误差累积	拟合任意分布噪声，理论上更高	介于两者之间

目录

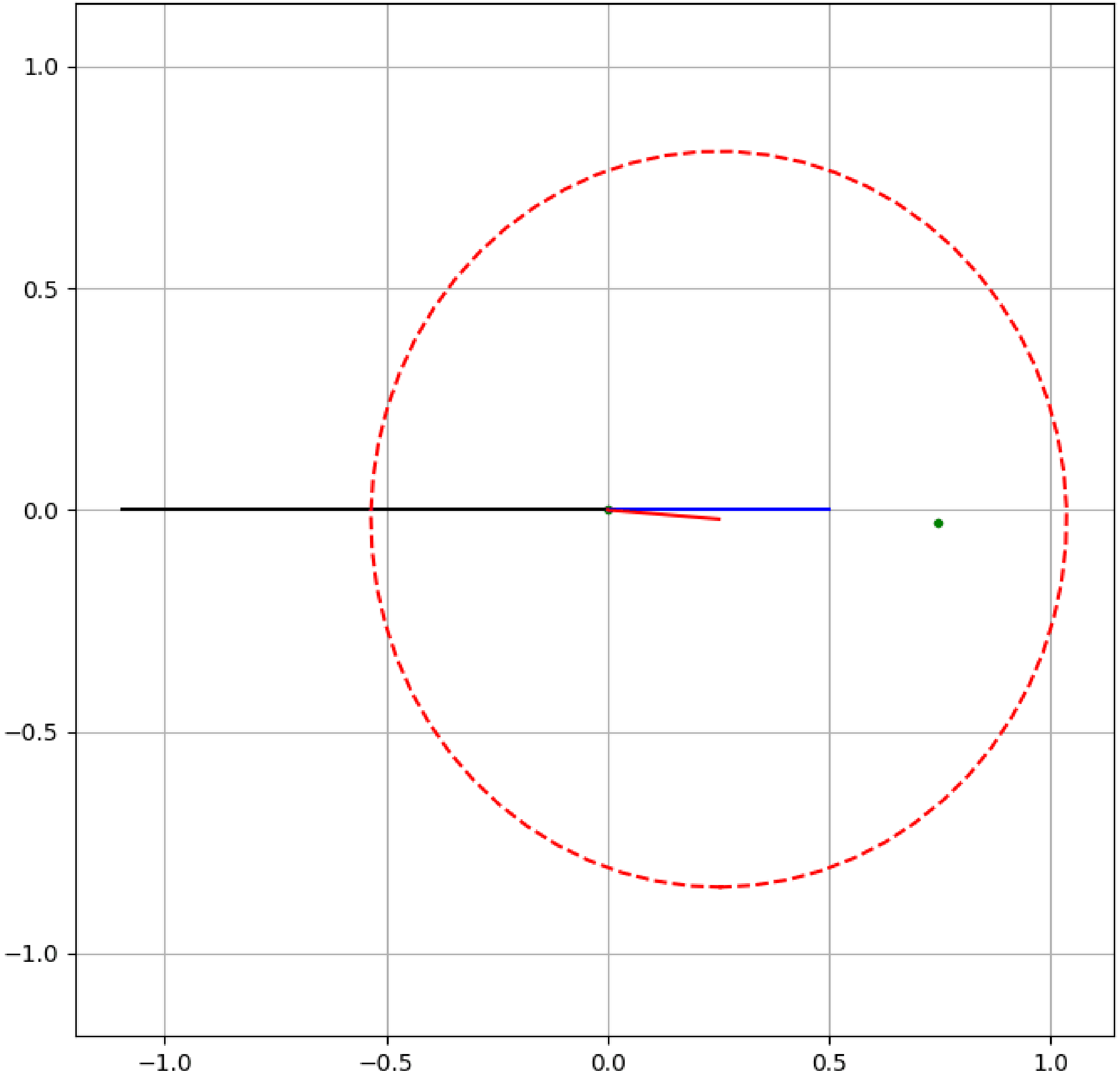
Contents

Part 4 | 无迹Kalman滤波定位

无迹Kalman（UKF）

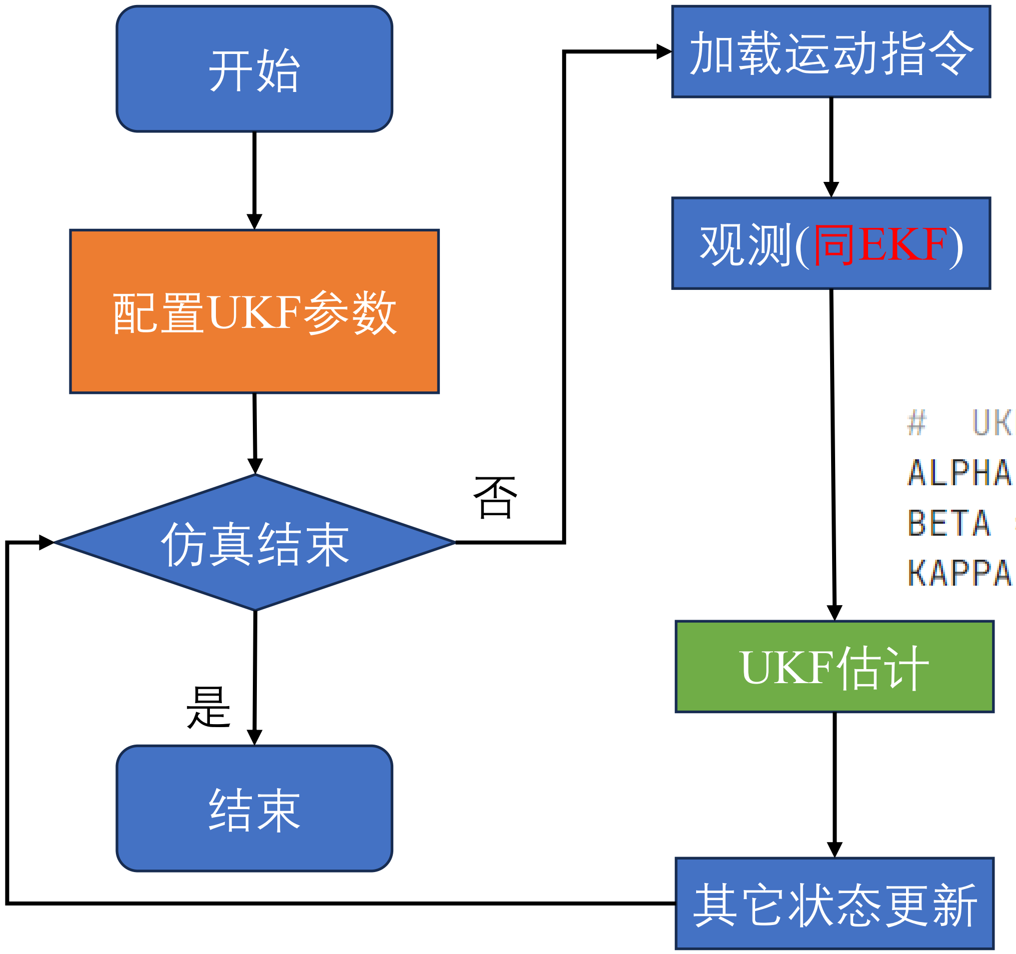
与EKF通过对非线性函数进行一阶泰勒展开（求Jacobian矩阵），实现局部线性化相比。UKF通过无迹变换（Unscented Transformation，UT）进行确定性采样，也就是围绕**均值最小样本点集合**(Sigma点)估计新的均值和协方差，从而提高拟合精度。还避免了EKF求Jacobian矩阵的需求，特别是复杂函数的求导，以及存在不可微函数的情形。

无迹变换：对非线性变换后的概率分布进行统计近似。具体来说，它通过选取一组能“代表”状态分布的**sigma点**，让这些点通过非线性函数传播，再从传播后的点中重构出变换后的均值和协方差。由于 sigma 点能捕捉状态的高阶统计特性（如二阶矩），因此对非线性变换的近似精度远高于 EKF 的线性化方法。



Part 4 无迹Kalman滤波定位

无迹Kalman（UKF）




```
#  UKF Parameter
ALPHA = 0.001
BETA = 2
KAPPA = 0
```

还是以相同的运动控制，匀速圆周运动，下分析UKF的整体流程。采用相似的噪声。

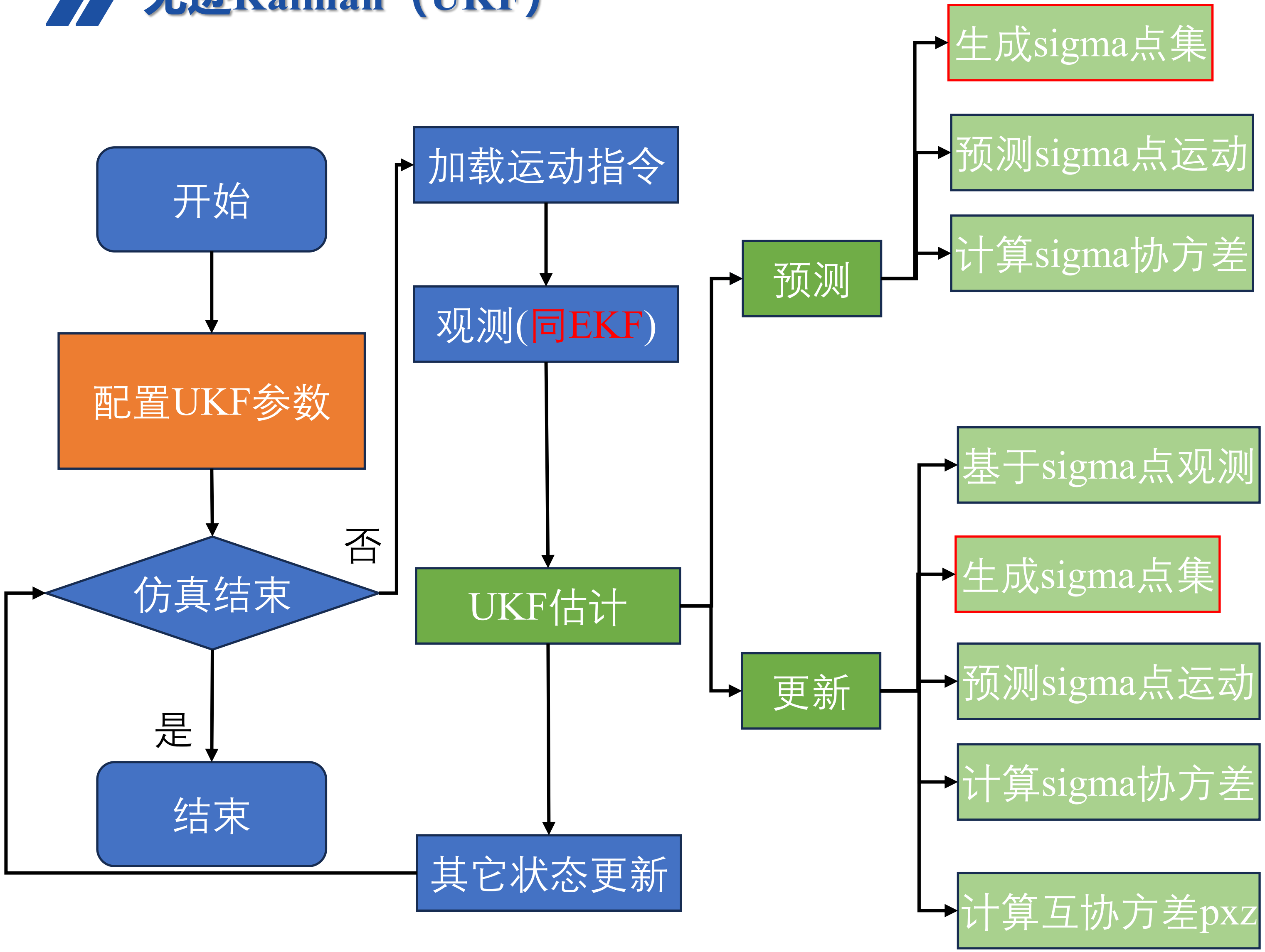
1. UKF配置（α，β，κ）：用于筛选sigma点和更新权重
2. nx为运动模型维度4

```
def setup_ukf(nx):
    lamb = ALPHA ** 2 * (nx + KAPPA) - nx
    # calculate weights
    wm = [lamb / (lamb + nx)]
    wc = [(lamb / (lamb + nx)) + (1 - ALPHA ** 2 + BETA)]
    for i in range(2 * nx):
        wm.append(1.0 / (2 * (nx + lamb)))
        wc.append(1.0 / (2 * (nx + lamb)))
    gamma = math.sqrt(nx + lamb)

    wm = np.array([wm])
    wc = np.array([wc])

    return wm, wc, gamma
```

3. 返回值：均值权重（wm）、协方差权重（wc）和缩放因子（gamma），**是生成 Sigma 点**（用于近似状态分布的采样点）和后续滤波计算的基础。

Part 4 | 无迹Kalman滤波定位

无迹Kalman（UKF）



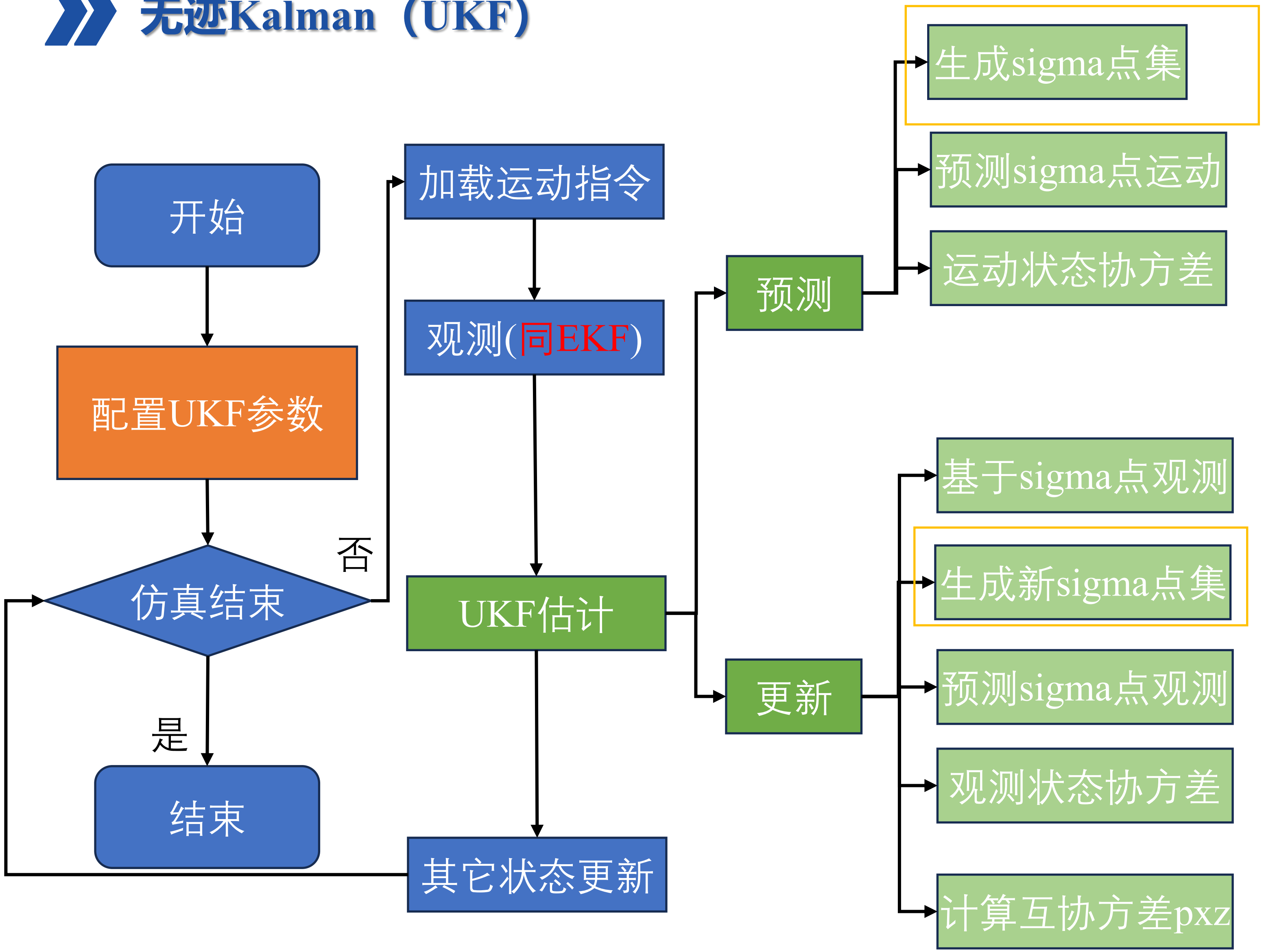
UKF实际在**预测**阶段和**更新**阶段，分别估计了一次sigma点。

- **预测**阶段sigma点计算：
  - 基于当前运动控制状态和协方差生成sigma点：每次延运动状态协方差矩阵的平方根矩阵，其列向量代表了状态空间中不确定性的主要方向，的正/负方向生成4个点，加上自身，构建sigma点集（9）。
  - 再借助运动模型更新每个sigma点的运动
  - 基于sigma点运动，更新均值权重，估计新运动状态xEst
  - 计算新协方差，预测不确定性PPred

利用历史信息预测当前信息

Part 4 | 无迹Kalman滤波定位

无迹Kalman（UKF）



UKF实际在**预测**阶段和**更新**阶段，分别估计了一次sigma点。

- **更新**阶段sigma点计算：
  - 预测sigma点可能的定位结果，并计算观测残差。
  - 重新生成sigma点
  - 提取每个sigma点的观测结果，即定位，更新均值权重，估计新观测状态zb
  - 计算观测协方差，预测观测的不确定性
  - 计算 **运动状态&观测** 的互协方差
  - 计算卡尔曼增益

利用观测信息修正预测信息

Part 4 无迹Kalman滤波定位

无迹Kalman（UKF）

特性	EKF	EnKF	UKF
非线性适应性	弱，线性化误差可能累积（尤其强非线性系统）	强（样本数量足够时可近似任意分布，适用于强非线性）	强，通过 Sigma 点更准确捕捉非线性分布
计算复杂度	低（雅可比矩阵计算量随状态维度线性增长）,O(n^2)，n为状态维度	高（O($N \cdot n^2$)，N为集合大小，通常N>>n）	高（需处理2n+1个 Sigma 点，n为状态维度）,O(n^3)
实现难度	中等（需推导雅可比矩阵，对模型形式敏感）	较低（无需推导复杂矩阵，样本处理逻辑简单）	较高（需设计 Sigma 点生成和权重计算逻辑）
理解	仅基于GPS和递归历史信息预测	借助随机粒子预测	粒子的目的性更强，在状态不确定性方向撒点

目录

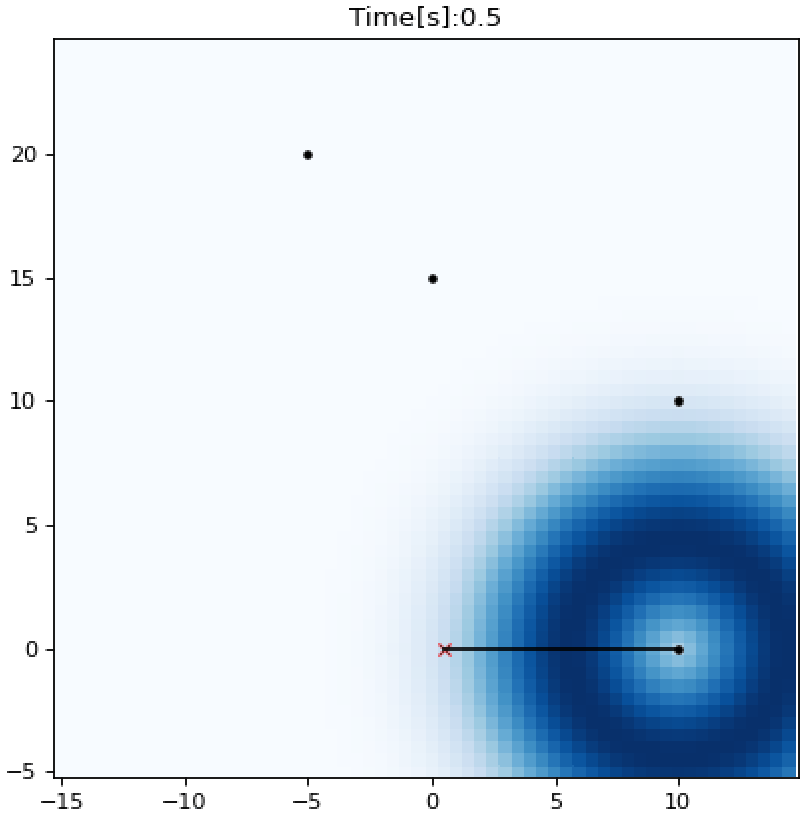
Contents

Part 5 直方图滤波定位

直方图滤波定位（histogram filter location）

- 直方图滤波定位，假设Agent**的航向角已知（带有电子罗盘/磁力计的机器人），RF_ID坐标已知**，基于Agent的速度和到锚点距离进行定位，估计坐标（x,y）。
- Agent的**初始坐标可以未知**，但至少需要能够感知到**至少任意一个已知锚点**（标签）。
- 定位过程的计算依赖 **栅格地图**（Grid Map），但不考虑障碍物，分辨率0.5m。栅格地图越精细，定位估计的最小理论精度越高，但计算量增加。

- 采用相同的运动指令，Agent进行匀速圆周运动[1, 0.1º]。
- 基础运动仍采用，航迹推算。
- 定位依赖场景中的RF_ID标签（位置同前）进行定位，感知半径10m。
- 运动过程存在运动噪声；定位信息也存在观测噪声。



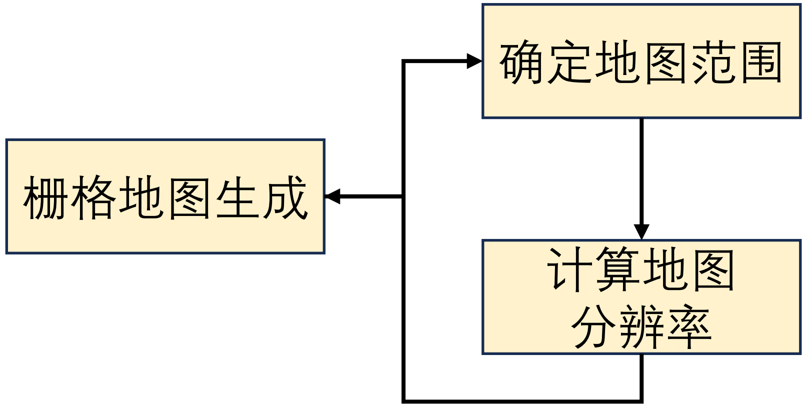
- **黑色点**：RF_ID位置。
- **黑色线**：感知范围内。
- 红色X：真值点位置。
- 蓝色热力图：HF估计定位分布。

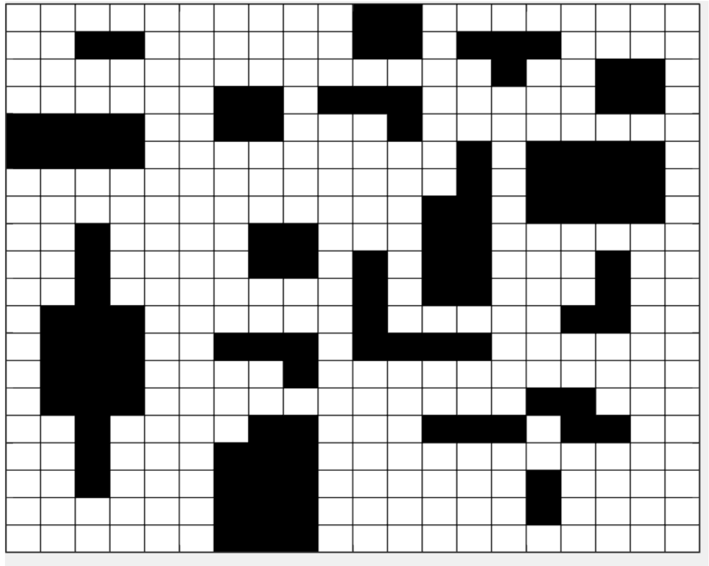
Part 5 直方图滤波定位

栅格地图

```
# grid map param
XY_RESOLUTION = 0.5  # xy grid resolution
MIN_X = -15.0
MIN_Y = -5.0
MAX_X = 15.0
MAX_Y = 25.0
```

覆盖潜在运动范围即可
（匀速圆周运动为直径20m的圆）
范围-30 ~ 30m





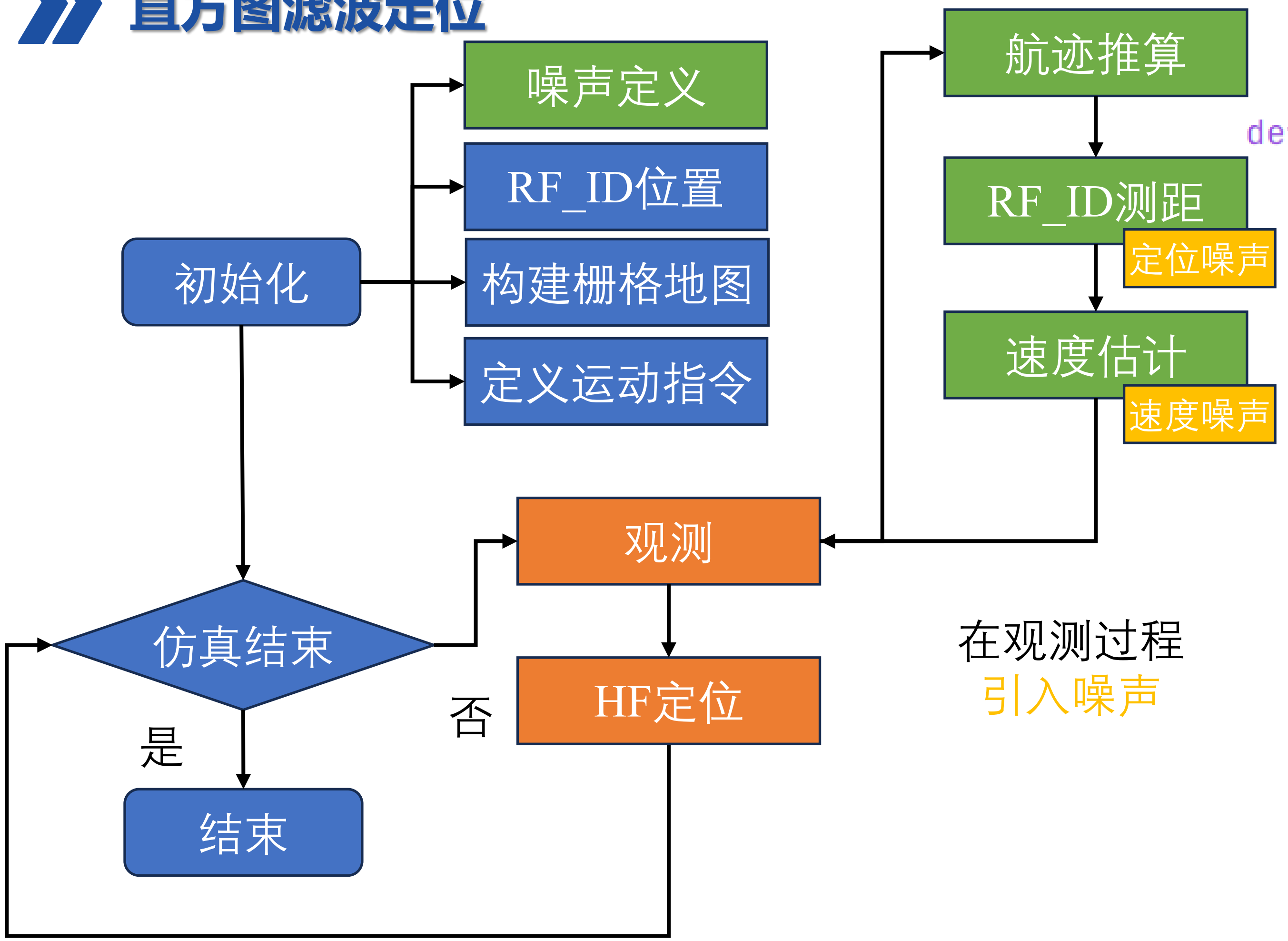
```
def init_grid_map(xy_resolution, min_x, min_y, max_x, max_y):
    grid_map = GridMap()

    grid_map.xy_resolution = xy_resolution
    grid_map.min_x = min_x
    grid_map.min_y = min_y
    grid_map.max_x = max_x
    grid_map.max_y = max_y
    grid_map.x_w = int(round((grid_map.max_x - grid_map.min_x)
                             / grid_map.xy_resolution))
    grid_map.y_w = int(round((grid_map.max_y - grid_map.min_y)
                             / grid_map.xy_resolution))

    grid_map.data = [[1.0 for _ in range(grid_map.y_w)]
                     for _ in range(grid_map.x_w)]
    grid_map = normalize_probability(grid_map)

    return grid_map
```

计算分辨率

初始化概率
全局归一化

Part 5 直方图滤波定位

直方图滤波定位



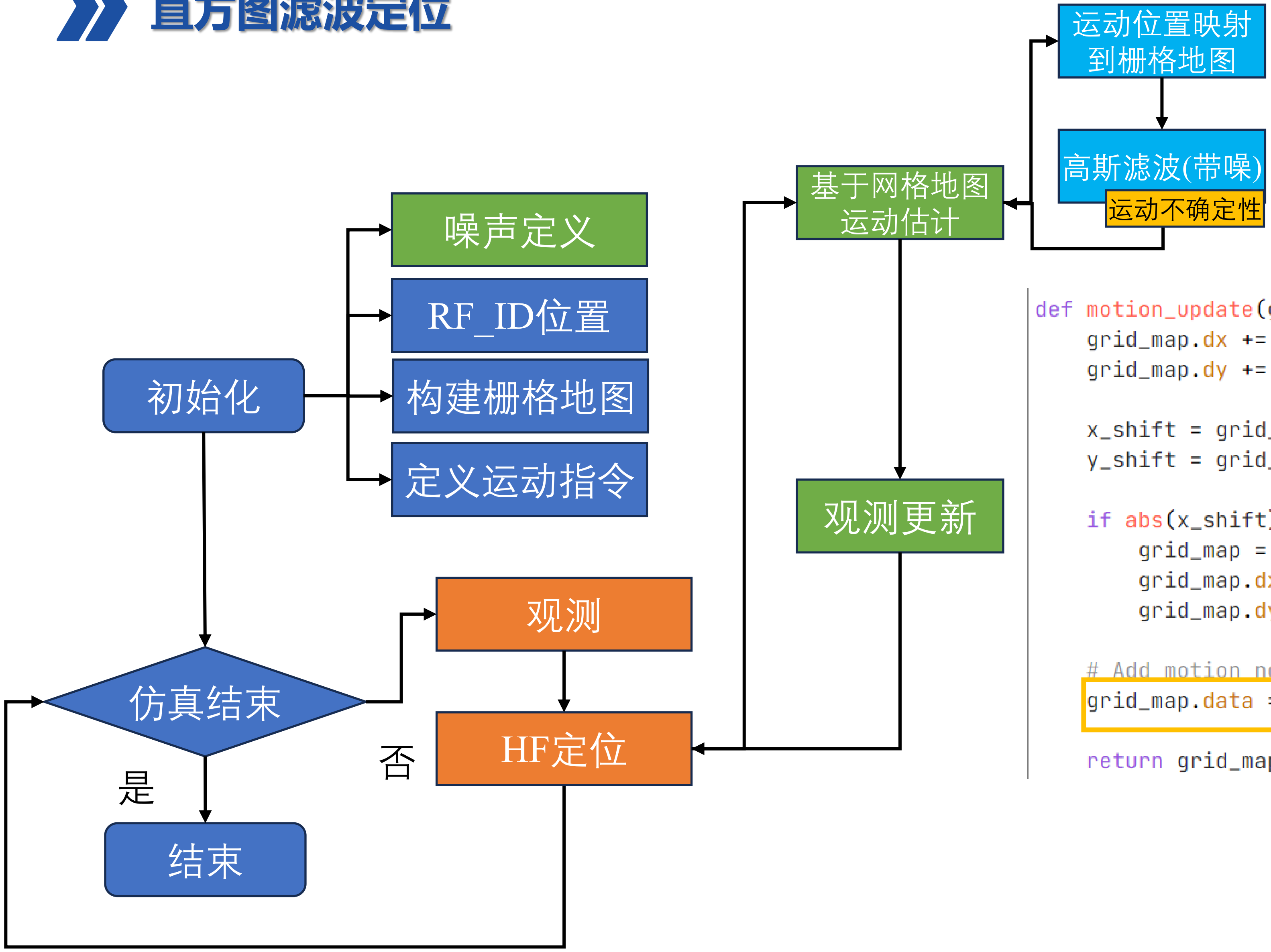
在观测过程
引入噪声

- 噪声：测距噪声，速度噪声。运动方向假设已知。
- 观测过程，仅对速度添加噪声。
- 测距时，仅感知半径内的RF_ID参与计算，并添加定位噪声，构建观测向量

```
def observation(xTrue, u, RFID):
    xTrue = motion_model(xTrue, u)

    z = np.zeros((0, 3))

    for i in range(len(RFID[:, 0])):

        dx = xTrue[0, 0] - RFID[i, 0]
        dy = xTrue[1, 0] - RFID[i, 1]
        d = math.hypot(dx, dy)
        if d <= MAX_RANGE:
            # add noise to range observation
            dn = d + np.random.randn() * NOISE_RANGE
            zi = np.array([dn, RFID[i, 0], RFID[i, 1]])
            z = np.vstack((z, zi))

    # add noise to speed
    ud = u[:, :]
    ud[0] += np.random.randn() * NOISE_SPEED

    return xTrue, z, ud
```

Part 5 | 直方图滤波定位

直方图滤波定位



需要将笛卡尔坐标系下的运动映射到栅格地图中；

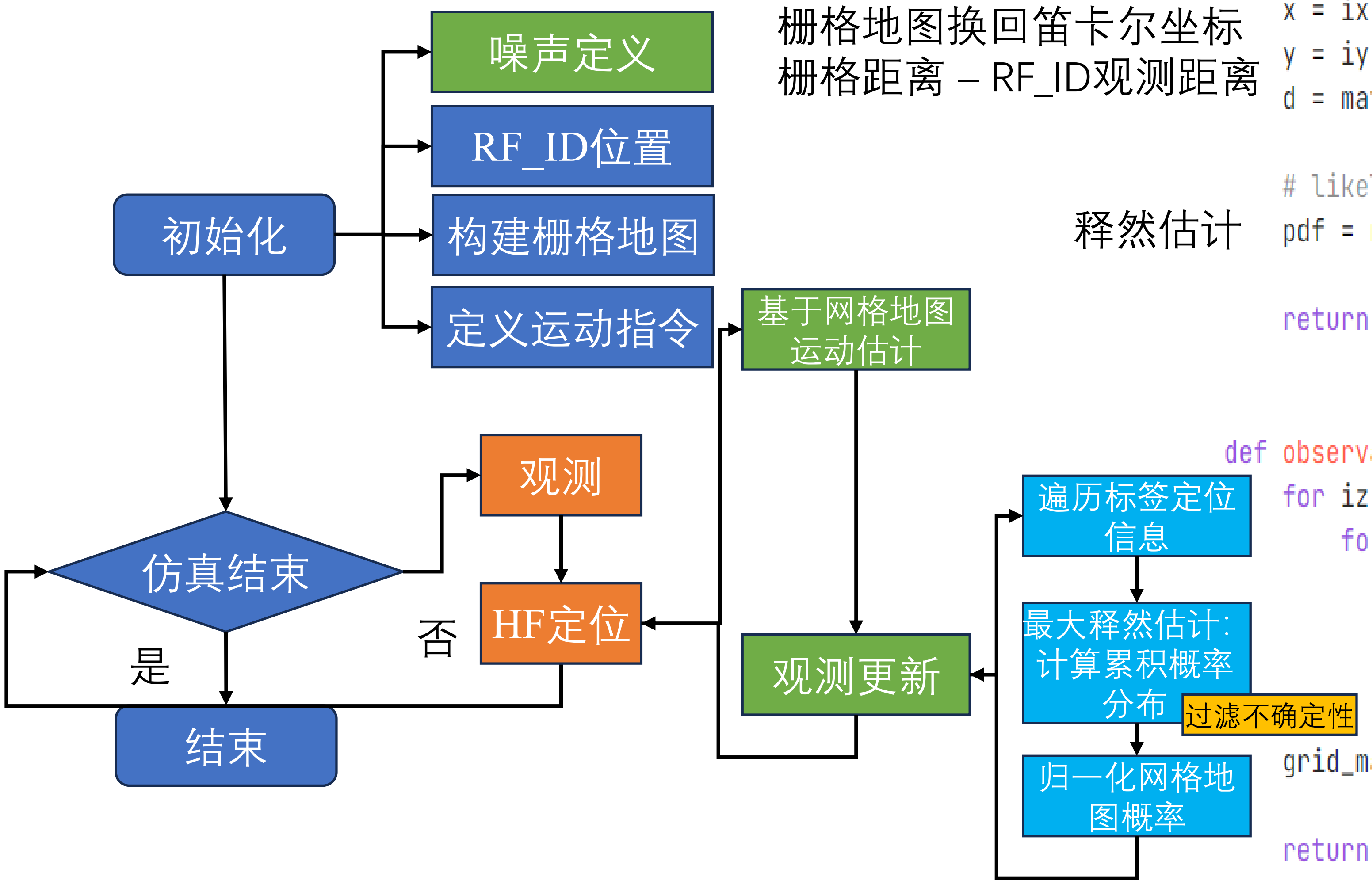
建模运动过程的不确定性，即沿着映射坐标的栅格递减扩散到周围（精确描述噪声）。

```python
def motion_update(grid_map, u, yaw):
    grid_map.dx += DT * math.cos(yaw) * u[0]
    grid_map.dy += DT * math.sin(yaw) * u[0]

    x_shift = grid_map.dx // grid_map.xy_resolution
    y_shift = grid_map.dy // grid_map.xy_resolution

    if abs(x_shift) >= 1.0 or abs(y_shift) >= 1.0:  # map should be shifted
        grid_map = map_shift(grid_map, int(x_shift[0]), int(y_shift[0]))
        grid_map.dx -= x_shift * grid_map.xy_resolution
        grid_map.dy -= y_shift * grid_map.xy_resolution

    # Add motion noise
    grid_map.data = gaussian_filter(grid_map.data, sigma=MOTION_STD)

    return grid_map
```

#Scipy.ndimage库

Part 5 直方图滤波定位

直方图滤波定位



栅格地图换回笛卡尔坐标
栅格距离 – RF_ID观测距离

释然估计

```python
def calc_gaussian_observation_pdf(grid_map, z, iz, ix, iy, std):
    # predicted range
    x = ix * grid_map.xy_resolution + grid_map.min_x
    y = iy * grid_map.xy_resolution + grid_map.min_y
    d = math.hypot(x - z[iz, 1], y - z[iz, 2])

    # likelihood
    pdf = norm.pdf(d - z[iz, 0], 0.0, std)

    return pdf


def observation_update(grid_map, z, std):
    for iz in range(z.shape[0]):
        for ix in range(grid_map.x_w):
            for iy in range(grid_map.y_w):
                grid_map.data[ix][iy] *= calc_gaussian_observation_pdf(
                    grid_map, z, iz, ix, iy, std)

    grid_map = normalize_probability(grid_map)

    return grid_map
```

Part 5 直方图滤波定位

直方图滤波定位

	直方图滤波定位	粒子滤波定位
状态空间	连续空间（x, y, yaw, v均为连续变量）	离散网格（x, y被划分为固定分辨率的网格）
计算复杂度	与粒子数量NP相关（代码中NP=100），粒子越多精度越高，但计算量线性增加	与网格数量相关（由XY_RESOLUTION决定），分辨率越高（网格越多）计算量越大，高维度时易爆炸
运动模型处理	对每个粒子独立应用带噪声的运动模型（motion_model）	通过网格平移（map_shift）模拟运动，再用高斯滤波（gaussian_filter）添加噪声
观测更新	计算每个粒子与观测值的似然度（gauss_likelihood），更新权重	计算每个网格与观测值的似然度，直接修正网格概率
重采样机制	当有效粒子数N_eff < NTh时重采样，保留高权重粒子	无需重采样，通过归一化（normalize_probability）维持概率和为 1
状态维度	处理高维度状态（代码中 4 维：x, y, yaw, v）	仅处理低维度状态（代码中 2 维：x, y，因高维度网格数量爆炸）

开始实验