工程实践与科技创新 IV-E

Lecture 3: 机器人定位与质点动力学模型

授课老师: 张晗、王景川 *2023* 年春季

1 移动机器人的建图、定位与导航简介

导航是移动机器人执行任务所需要必备的能力，移动机器人成功的导航需要四部分的支持：感知，机器人收集并理解传感器的信息，提取有意义的数据；定位，机器人确定它在环境中的位置；认知，机器人决定如何行动以达到目标；运动控制，机器人调节它的运动输出，以实现期望的轨迹、到达目标。

1.1 定位问题

在这四个部分中，定位一直是最受关注的部分，也是研究的热点问题，定位的实质是解决机器人“我在哪里?”的问题。定位不仅意味着要确定机器人在地球参考系中的绝对位置与姿态。考虑到机器人与环境或是人的交互，也许需要确定它在环境中或者与环境中的人相对的位置与姿态。同时，在机器人环境认知的部分，为了达到目标，机器人会选择一种可以到达特定目标位置的策略，这时仅仅知道机器人在环境中的位置与姿态还不够，机器人还需要获取或建立一个环境模型，即一张环境地图，它帮助机器人规划一条达到目标的路径。所以，定位仅仅是简单地确定机器人在空间中一个绝对位姿，它意味构建一张地图，而后确定机器人相对于地图的位置与姿态。定位研究中有三种类型问题，它们分别是：位置跟踪、全局定位和绑架机器人问题。

- 位置跟踪：在位置跟踪中，机器人的当前位置是根据对它的以前位置（跟踪）的知识而更新的。这说明机器人的初始位置假定为已知的。另外，机器人姿态的不确定性必须小。如果不确定性太大，位置跟踪会使定位机器人失败。
- 全局定位：相反，全局定位假定机器人的初始位置是不知道的。这意味着机器人可以被放在环境中的任何地方，不必有关于环境的知识——能够在环境内全局地定位。
- 绑架机器人问题：绑架机器人问题实际是机器人并被转移到环境中任意位置的情况。绑架机器人问题与全局定位问题相似，只要机器人知道已被绑架就需要确定它在环境中的位置与姿态。绑架问题的困难来源于，机器人不知道是否被转移至另一个地方，而从绑架中能够获得机器人的位姿信息，是机器人自主运行的一个必要条件。

1.2 地图

地图的描述实际上就是表示机器人移动所处的环境的问题，是表示机器人可能位置或位置的对偶问题。有关环境表示方法所做的决策，会影响到机器人位置表示可用的选择。通常，位置表示的准确性受到地图表示准确性的限制。在选择一个特殊的地图表示方法时，必须了解三个基本关系：

- 地图的精度必须恰当地匹配机器人需要达到目标的精度；
- 地图的精度和所表示的特征类型必须匹配机器人传感器所返回的数据类型和精确性；
- 地图表示的复杂性直接影响有关建图、定位和导航推理的计算复杂性。

1.3 概率栅格地图表示

连续值的地图是环境精确分解的一种方法。环境特征的位置可以在连续空间中精密地予以标记。但是仅仅在二维空间中进行连续地图的表示，都会因为更高的维数引起计算上的爆炸。而将地图进行某种形式的分解是对环境描述的一种抽象化表示，栅格地图就是一种固定分解的地图表达形式，在这里环境是棋盘格化的。为了获得地图表达，将连续的现实环境变换成离散近似，在一个占有栅格中，环境被表示成离散栅格。这里，各个单元或者被填满（障碍部分）；或者是空的（自由空间部分）。当机器人装备基于测距传感器时，这特别有价值，因为各传感器的测距值与机器人的绝对位置结合在一起，可以直接被用于更新各单元的填充值或空值。在占有栅格中，各单元可能有一个计数器。借此，0 值说明单元还未被任何测距所 “击中”，所以单元可能是自由空间。当测距点测的数目增加时，单元的值递增超过一定的阈值，单元必定成为障碍物（1 值）。当测距点击越过单元，点击一个更远单元时，单元的值通常会被打折扣（0-1 值），这个折扣值就代表该栅格被占据的概率。概率栅格方法有两个主要缺点：

- 在机器人存储器中地图的尺寸随着环境规模的增长而增大。如果用小的单元尺寸，这个尺寸迅速变得难以办到。占有栅格方法与现实环境的假设不相容，后者能使连续表示在大的、稀疏环境中具有潜在的非常小的存储需求。相反，概率栅格必须为矩阵中的每一个单元保留存储器；
- 任何像这样的固定分解，不管环境细节如何，在环境上都要预先加上一个几何栅格，在几何特征并不明显的环境里，这显然是不合适的。

1.4 基于概率地图的定位

基于概率地图的定位技术，明确地辨识了可能的机器人位姿概率。通常，移动机器人定位的概率问题一直被认为是：处理从传感器获取的数据受测量误差影响的问题。所以，我们将计算机器人处在给定方位的概率称为 “概率机器人学”。概率机器人学的关键性思想是用概率论来表示不确定性，换句话来说，不是给出当前机器人方位的一个单独的最好估计，而是将机器人方位表示为对所有可能的机器人姿态的一个概率分布。当前，有两类较为流行的基于概率地图的定位方法：第一类是马尔可夫（Markov）定位，对所有可能的机器人位置，使用一个明确地指定的概率分布；第二个方法是卡尔曼滤波（Kalman Filter）定位，使用机器人位置的高斯概率密度的表示。与马尔可夫定位不同，卡尔曼滤波定位在机器人的方位空间中不独立考虑各个可能的姿态。有趣的是，如果机器人位置的不确定性被假定为具有高斯形式，则卡尔曼滤波定位过程由马尔可夫定位的公理产生。

2 状态空间模型与离散时间隐 Markov 模型

我们已在《现代控制理论》中学过系统的状态空间模型，这里我们简单地复习一下。以连续时不变系统为例，其状态空间由以下常微分方程表示：

$$\dot{x} = Ax + Bu, \quad y = Cx, x(0) = x_0 \tag{1}$$

其中 $x \in \mathbb{R}^n$ 为系统的状态，$u \in \mathbb{R}^m$ 为系统的外部输入，$y \in \mathbb{R}^p$ 是系统的输出，x_0 为系统的初始状态。我们不能直接对系统内部的状态 x_t 进行测量，而只能对其所呈现出来的输出 y_t 进行观测。

Example 1 考虑以下电路系统：

由基尔霍夫电压定律可得：

$$-u(t) + u_C(t) + u_L(t) + y(t) = 0,$$

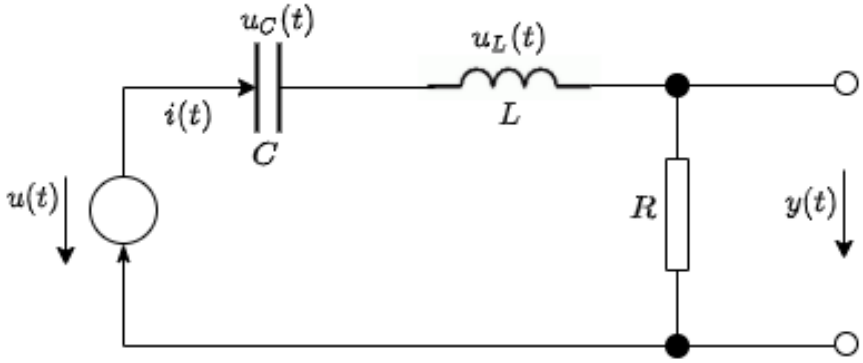

Figure 1: LCR 振荡电路

将电容、电感的模型 $C\frac{d}{dt}u_C(t) = i(t)$, $L\frac{d}{dt}i(t) = -u_L(t)$ 代入上式，并令 $x(t) := [x_1(t)^T, x_2(t)^T]^T$, $x_1(t) := u_C(t)$，$x_2(t) := i(t)$ 可得该电路系统的状态空间模型：

$$\dot{x} := \frac{d}{dt} = \begin{bmatrix} 0 & \frac{1}{C} \\ -\frac{1}{L} & -\frac{R}{L} \end{bmatrix} x + \begin{bmatrix} 0 \\ \frac{1}{L} \end{bmatrix} u$$

实际中系统的状态空间模型往往要比线性时不变系统(1)更复杂，其状态空间模型由以下常微分方程表示：

$$\dot{x} = \hat{f}(x, u), \quad y = \hat{h}(x) \tag{2}$$

而由于机器人通常通过计算机控制，其控制周期受到计算机指令周期的限制，所以在实际应用中，我们经常将系统的状态空间模型在时域上离散化，获得如下的离散时间状态空间模型：

$$x_{t+1} = f(x_t, u_t), \quad y_t = h(x_t), \quad t = 1, 2, 3, \ldots$$

在本课程中，我们将主要与离散时间的状态空间模型打交道。直观上，离散时间状态空间模型可以被理解为：给定 t 时刻的系统状态 x_t 和外部输入 u_t，系统处于 $t+1$ 时刻的系统状态 x_{t+1} 就能够由上式计算，也即系统处于 t 时刻的状态衍化到了 $t+1$ 时刻的状态；而在每个时刻 t，我们都能够观测到系统的输出 y_t。如果该系统在每个时刻的衍化与对该系统输出的观测均受到环境中随机噪声的影响，则我们可以得到如下状态空间模型：

$$x_{t+1} = f(x_t, u_t) + w_t, \quad y_t = h(x_t) + v_t, \quad t = 1, 2, 3, \ldots, \tag{3}$$

其中 $w_t \in \mathbb{R}^n, t = 1, 2, \ldots$ 被称为 “过程噪声”，$v_t \in \mathbb{R}^p, t = 1, 2, \ldots$ 被称为 “观测噪声”，均为随机向量。系统(3)的观测和衍化过程可由 Fig. 2表示。从 Fig. 2中可以看出：

1. 对每个时刻 t 的状态 x_t 来说，若上一时刻的状态 x_{t-1} 和系统输入 u_{t-1} 已知，则 x_t 的分布与之前历史的状态 $x_1, \ldots, x_{t-2}$ 无关；
2. 对每个时刻 t 的观测 y_t 来说，若当前时刻的状态 x_t 已知，则 y_t 的分布与与之前历史的状态 $x_1, \ldots, x_{t-2}$ 无关。

这事实上符合 “离散时间隐 Markov 模型” 的定义；(3)定义了一种离散时间隐 Markov 模型。下面我们给出离散时间隐 Markov 模型严谨的数学定义：

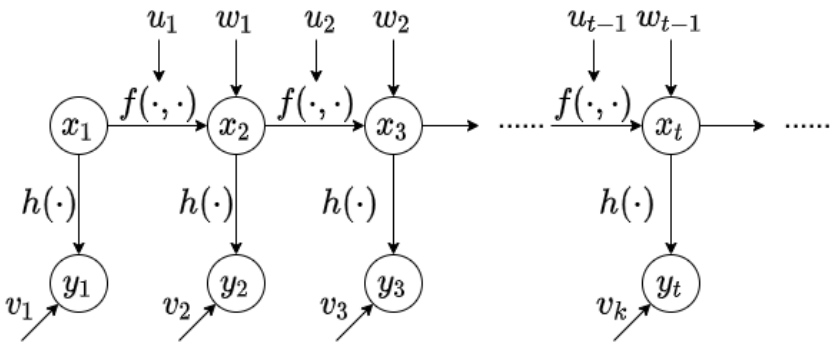

Figure 2: 离散时间隐 Markov 模型的观测和衍化过程图示

Definition 1 令 $\{X_t\}$ 和 $\{Y_t\}$ 为两组离散时间的随机过程，其中 $t \geq 1$。若

- 对于任意的 $t \geq 1$，X_t 不能被直接观测，且 $\mathbb{P}(x_{t+1} = x \mid X_1 = x_1, X_2 = x_2, \ldots, X_t = x_t) = \mathbb{P}(x_{t+1} \in \mathcal{A} \mid x_t)$*(假设这两个条件概率分布是良定义的)*;
- 对于任意的 $t \geq 1$, $x_1, \ldots, x_t$ 和 *Borel* 集 $\mathcal{A}$, 均有 $\mathbb{P}(Y_t \in \mathcal{A} \mid X_1 = x_1, \ldots, X_t = x_t) = \mathbb{P}(Y_t \in \mathcal{A} \mid X_t = x_t)$ 成立，

则 $\{X_t\}$ 和 $\{Y_t\}$ 是一个离散时间隐 *Markov* 模型。

3 概率论基础知识与多元高斯分布

本节将回顾一些本课程中会用到的概率论基础知识。

1. 随机向量 $\xi = [\xi_1, \ldots, \xi_n]^T$ 是一个从 "结果空间" Ω 到 $\mathbb{R}^n$ 的映射。我们用 $x = [x_1, \ldots, x_n]^T \in \mathbb{R}^n$ 表示随机向量 X 的 "结果" 或 "实现"。在本课程中，概率分布函数 $F_\xi(x)$ 和概率密度函数 $p_\xi(x)$ 可被大致地理解为:

$$F_\xi(x) = \mathbb{P}(\xi_1 \leq x_1, \xi_2 \leq x_2, \ldots, \xi_n \leq x_n), \quad p_\xi(x) = \frac{dF_\xi(x)}{dx_1 dx_2 \ldots dx_n};$$

   概率分布函数 $F_\xi(x)$ 与概率密度函数 $p_\xi(x)$ 具有以下性质:

   - $F_\xi(x) = \int_{-\infty}^{x} p_\xi(x)dx$;
   - $\int_{-\infty}^{\infty} p_\xi(x)dx = 1$;
   - 若 $X \in \mathbb{R}^n, Y \in \mathbb{R}^m$ 的联合概率密度函数为 $p_{X,Y}(x, y)$，则 $p_X(x) = \int_{-\infty}^{\infty} p_{X,Y}(x, y)dy$, $p_Y(x) = \int_{-\infty}^{\infty} p_{X,Y}(x, y)dx$。

2. 在本课程中，随机向量 X 的数学期望可被大致地理解为:

$$\mathbb{E}[X] = \int_{-\infty}^{\infty} x p_X(x)dx;$$

而 X 的函数 $f(X)$ 的数学期望为

$$\mathbb{E}[f(x)] = \int_{-\infty}^{\infty} f(x) p_X(x) dx.$$

对于任意的随机向量 $X, Y \in \mathbb{R}^n$ 和常数 $\alpha \in \mathbb{R}$，都有以下性质：

$$\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y] \quad \mathbb{E}[\alpha X] = \alpha \mathbb{E}[X];$$

X 的协方差矩阵的定义为

$$\begin{aligned}
\operatorname{cov}(X) &= \mathbb{E}\left[(X - \mathbb{E}[X])(X - \mathbb{E}[X])^T\right] = \mathbb{E}\left[XX^T - \mathbb{E}[X]X^T - X\mathbb{E}[X]^T + \mathbb{E}[X]\mathbb{E}[X]^T\right] \\
&= \mathbb{E}[XX^T] - \mathbb{E}[X]\mathbb{E}[X]^T - \mathbb{E}[X]\mathbb{E}[X]^T + \mathbb{E}[X]\mathbb{E}[X]^T \\
&= \mathbb{E}[XX^T] - \mathbb{E}[X]\mathbb{E}[X]^T
\end{aligned}$$

3. 对于 $x \in \mathbb{R}$，高斯积分

$$\int_{-\infty}^{\infty} e^{-x^2} dx = \sqrt{\pi}.$$

Proof: 我们首先需证明该积分值存在。令

$$I(a) = \int_{-a}^{a} e^{-x^2} dx,$$

我们需证明 $\lim_{a\to\infty} I(a) < \infty$。注意到由于 $-xe^{-x^2} > e^{-x^2}, \forall x \in (-\infty, -1]$ 以及 $xe^{-x^2} > e^{-x^2}, \forall x \in [1, \infty)$，下式一定成立：

$$\begin{aligned}
&\int_{-\infty}^{\infty} e^{-x^2} dx = \int_{-\infty}^{-1} e^{-x^2} dx + \int_{-1}^{1} e^{-x^2} dx + \int_{-1}^{\infty} e^{-x^2} dx \\
&< \int_{-\infty}^{-1} -xe^{-x^2} dx + \int_{-1}^{1} e^{-x^2} dx + \int_{1}^{\infty} xe^{-x^2} dx \\
&= -\frac{1}{2}\int_{-\infty}^{-1} e^{-x^2} dx^2 + \int_{-1}^{1} e^{-x^2} dx + \frac{1}{2}\int_{1}^{\infty} e^{-x^2} dx^2 \\
&= \frac{1}{2}\int_{1}^{\infty} e^{-y} dy + \int_{-1}^{1} e^{-x^2} dx + \frac{1}{2}\int_{1}^{\infty} e^{-y} dy \\
&= e^{-1} + \int_{-1}^{1} e^{-x^2} dx < \infty.
\end{aligned}$$

所以 $\lim_{a\to\infty} I(a)$ 存在。

$$I^2(a) = \left(\int_{-a}^{a} e^{-x^2} dx\right)\left(\int_{-a}^{a} e^{-y^2} dy\right) = \int_{-a}^{a}\int_{-a}^{a} e^{-(x^2+y^2)} dxdy$$

把上述重积分转到极坐标系中，可得

$$\left(1 - e^{-a^2}\right)\pi = \int_0^{2\pi}\int_0^{a} re^{-r^2} drd\theta \le I^2(a) \le \int_0^{2\pi}\int_0^{\sqrt{2}a} re^{-r^2} drd\theta = \left(1 - e^{-2a^2}\right)\pi$$

对上述不等式取极限，可得 $\pi \le \lim_{a\to\infty} I^2(a) \le \pi$，所以 $\int_{-\infty}^{\infty} e^{-x^2} dx = \sqrt{\pi}$. □

4. 如果一个随机向量 $X \in \mathbb{R}^n$ 的概率密度函数 $p_X(x)$ 为

$$p_X(x) = \frac{1}{(2\pi)^{n/2}\det(P)^{1/2}} e^{-\frac{1}{2}(x-m)^T P^{-1}(x-m)},$$

则 X 服从多元高斯分布，记为 $X \sim \mathcal{N}(m, P)$。

若 $X \sim \mathcal{N}(m, P)$，$A \in \mathbb{R}^{m\times n}$，且 A 行满秩，则 $Y = AX + b \sim \mathcal{N}(Am + b, APA^T)$.

5. 连续随机向量的贝叶斯公式为

$$p_{X|Y}(x) = \frac{p_{X,Y}(x,y)}{p_Y(y)} = \frac{p_{Y|X}(y)p_X(x)}{p_Y(y)},$$

其中 $p_{X|Y}(x)$ 为给定 Y 条件下 X 的后验概率密度函数，$p_{X,Y}(x,y)$ 为 X 和 Y 的联合分布的概率密度函数，$p_Y(y)$ 为 Y 的边缘分布的概率密度函数，$p_{Y|X}(y)$ 是给定 X 条件下 Y 的似然函数，$p_X(x)$ 是 X 的先验概率密度函数。

后文为了简便起见，将使用下列写法：$p(x|y) := p_{X|Y}(x)$，$p(x,y) := p_{X,Y}(x,y)$，$p(y) := p_Y(y)$，$p(x) := p_X(x)$，$p(y|x) := p_{Y|X}(y)$。

Example 2

$$p(x,y|z) = \frac{p(x,y,z)}{p(z)} = \frac{p(x|y,z)p(y,z)}{p(z)} = p(x|y,z)p(y|z).$$

Example 3 假设 $\{x_t\}$ 和 $\{y_t\}$ 是一个离散时间隐 *Markov* 模型。$x_1, \ldots, x_n, y_1, \ldots y_n$ 的联合分布的概率密度函数满足

$$p(x_{1:n}, y_{1:n}) = p(y_1|x_1)p(x_1)\prod_{t=2}^{n} p(y_t|x_t)p(x_t|x_{t-1}).$$

Proof:

$$\begin{aligned}
p(x_{1:n}, y_{1:n}) &= p(y_n|x_n, \underbrace{\cancel{x_{1:n-1}, y_{1:n-1}}}_{\textit{Markov 特性}})p(x_{1:n}, y_{1:n-1}) \\
&= p(y_n|x_n)p(x_n|x_{n-1}, \underbrace{\cancel{x_{1:n-2}, y_{1:n-1}}}_{\textit{Markov 特性}})p(x_{1:n-1}, y_{1:n-1}) \\
&= p(y_1|x_1)p(x_1)\prod_{t=2}^{n} p(y_t|x_t)p(x_t|x_{t-1})
\end{aligned}$$

□

在移动机器人定位和建图中，贝叶斯公式将扮演非常重要的角色。移动机器人上配备了激光雷达、毫米波雷达、声呐、IMU、GPS 等传感器。传感器返回的数据带有测量误差；而机器人本身的移动会由于控制算法的精度、环境的变化、地面打滑等，同样会引入误差，所以机器人总体是可被理解为一个隐 Markov 模型，包含机器人在 t 时刻的位姿 $\{x_t\}$，传感器在 t 时刻的观测值 $\{y_t\}$ 以及地图 M。我们可以进一步将移动机器人的定位、建图和 SLAM 大致地理解为：

- 安装在机器人上的传感器对当前环境（地图）的观测，总是基于机器人当前的位姿，其在每一时刻 t 观测值的概率密度函数为 $p(y_t|x_t, M)$；
- 离线建图：基于全部的传感器信息和机器人位姿信息，建立地图，即从 $p(M|x_{1:N}, y_{1:N})$ 中进行采样 $1 \leq t \leq N$；
- 定位：基于传感器历史信息与地图，推算机器人当前时刻 t 的位姿信息，即从 $p(x_t|y_{1:t}, M)$ 中进行采样；
- SLAM：基于传感器历史信息，推算机器人当前时刻 t 的位姿信息和地图信息 M_t，即从 $p(x_t, M_t|y_{1:t})$ 中进行采样。

4 质点动力学模型

4.1 质点动力学模型建立

如果将机器人简化为一个质点，考虑这种简化的情况，如 figure3 所示：当在某一时刻，对现实环境中位于点 P 的质点施加特定方向的力 F 时，可以得出以下关于质点状态的等式：

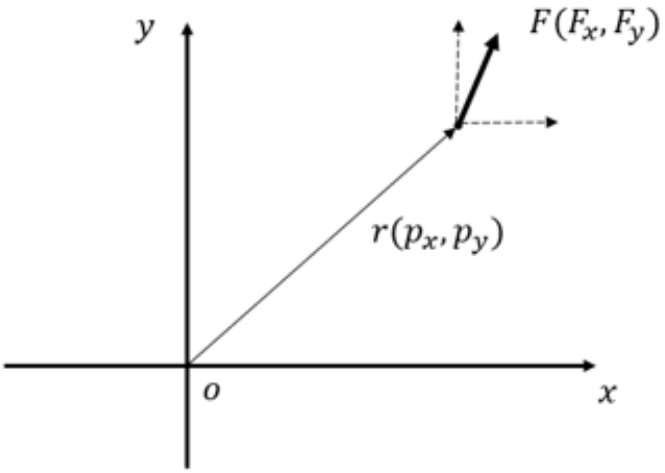

Figure 3: 质点动力学模型

$$
\begin{aligned}
state: x &= [v_x, p_x, v_y, p_y]^T \\
\dot{v}_x &= F_x/m + w_1 \\
\dot{p}_x &= v_x + w_2 \\
\dot{v}_y &= F_y/m + w_3 \\
\dot{p}_y &= v_y + w_4
\end{aligned}
$$

考虑到机器人在运动过程中，可能会出现打滑等不可预知的现象，所以我们在上述方程中加入了白噪声 $w_1(t)$, $w_2(t)$,$w_3(t)$ 和 $w_4(t)$。我们把它写成向量形式：$w(t) = [w_1(t), w_2(t), w_3(t), w_4(t)]^T$。白

噪声模型是对现实中噪声的理想化处理，其期望和方差在时序上相互不相关，并且其期望为 0，即

$$\mathbb{E}[w] = 0, \qquad \mathrm{cov}(w(\tau_1), w(\tau_2)) = \delta(\tau_1 - \tau_2)\tilde{Q}.$$

其中，$\tilde{Q}$ 为半正定矩阵。若我们假定，w_1,w_2,w_3,w_4 之间，两两互相独立，那么 $\tilde{Q}$ 是对角阵，即 $\tilde{Q} = diag(q_1, q_2, q_3, q_4)$。

另外，$\delta(\tau_1 - \tau_2)$ 为 Dirac-delta 函数，满足

$$\delta(\tau_1 - \tau_2) = \begin{cases} +\infty, & \text{如果}\tau_1 = \tau_2, \\ 0, & \text{其他.} \end{cases}.$$

对于支撑集为紧集的连续函数 $f(x)$ 积分，Dirac-delta 函数具有如下性质：

$$\int_{-\infty}^{+\infty} f(t)\delta(t - \tau)dt = f(\tau).$$

这也是以后推导卡尔曼滤波器时噪声模型的基础。

将质点状态方程写为矩阵形式：

$$\begin{bmatrix} \dot{v}_x \\ \dot{p}_x \\ \dot{v}_y \\ \dot{p}_y \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} v_x \\ p_x \\ v_y \\ p_y \end{bmatrix} + \begin{bmatrix} 1/m & 0 \\ 0 & 0 \\ 0 & 1/m \\ 0 & 0 \end{bmatrix} \begin{bmatrix} F_x \\ F_y \end{bmatrix} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix}$$

以状态空间模型写出，其形如：

$$\dot{x} = \tilde{A}x + \tilde{B}u + w$$

即为质点动力学模型的连续形式。

4.2 质点动力学模型离散化

在使用计算机进行仿真和设计对应的控制系统时，因为传感器采集到的信息和控制器发出的信息在时域上不可能是连续的，只能以特定的频率接收和发出信息，因此不可避免的需要对该动力学模型进行离散化。为了在计算机中实现上述机器人运动学模型，我们假设第 t_k 时刻与 t_{k+1} 时刻之间的机器人状态保持不变，即：$f(X(\tau), u(\tau)) \approx f(X(t_k), u(t_k))$，可得

$$\begin{aligned} X(t_{k+1}) &= X(t_k) + \int_{t_k}^{t_{k+1}} f(X(\tau), u(\tau))d\tau + \int_{t_k}^{t_{k+1}} w(\tau)d\tau \\ &\approx X(t_k) + \int_{t_k}^{t_{k+1}} f(X(t_k), u(t_k))d\tau + \underbrace{\int_{t_k}^{t_{k+1}} w(\tau)d\tau}_{w_k} \\ &= X(t_k) + (t_{k+1} - t_k)f(X(t_k), u(t_k)) + w_k, \end{aligned}$$

其中 $\mathbb{E}[w_k]=\int_{t_k}^{t_{k+1}}\mathbb{E}[w(\tau)]d\tau=0$ 并且

$$\begin{aligned}
&\operatorname{cov}(w_{k_1},w_{k_2})=\mathbb{E}\left[\int_{t_{k_1}}^{t_{k_1+1}}w(\tau_1)d\tau_1\int_{t_{k_2}}^{t_{k_2+1}}w(\tau_2)d\tau_2\right]\\
&=\left[\int_{t_{k_2}}^{t_{k_2+1}}\int_{t_{k_1}}^{t_{k_1+1}}\mathbb{E}[w(\tau_1)w(\tau_2)]d\tau_1d\tau_2\right]=\left[\int_{t_{k_2}}^{t_{k_2+1}}\int_{t_{k_1}}^{t_{k_1+1}}\delta(\tau_1-\tau_2)\tilde{Q}d\tau_1d\tau_2\right]\\
&=\begin{cases}(t_{k+1}-t_k)\tilde{Q}:=Q_k, & \text{当}k_1=k_2=k\\ 0, & \text{其他}\end{cases}.
\end{aligned}$$

对于线性系统而言，我们可以有更精确的离散化方程:

$$\begin{aligned}
&\dot{x}=\tilde{A}x+\tilde{B}u+w\\
&x(t_{k+1})=e^{\tilde{A}(t_{k+1}-t_k)}x(t_k)+\int_{t_k}^{t_{k+1}}e^{\tilde{A}(t_{k+1}-s)}\tilde{B}u(s)ds+\int_{t_k}^{t_{k+1}}e^{\tilde{A}(t_{k+1}-s)}w(s)ds\\
&x(t_{k+1})\approx\underbrace{e^{\tilde{A}(t_{k+1}-t_k)}}_{A}x(t_k)+\underbrace{\int_0^{t_{k+1}-t_k}e^{\tilde{A}\tau}d\tau\tilde{B}}_{B}u(t_k)+\underbrace{\int_{t_k}^{t_{k+1}}e^{\tilde{A}(t_{k+1}-s)}w(s)ds}_{w_k},
\end{aligned}$$

其中，$\mathbb{E}[w_k]=\int_{t_k}^{t_{k+1}}e^{\tilde{A}(t_{k+1}-s)}\mathbb{E}[w(s)]ds=0$，且有

$$\begin{aligned}
&\operatorname{cov}(w_{k_1},w_{k_2})=\mathbb{E}\left[\int_{t_{k_1}}^{t_{k_1+1}}e^{\tilde{A}(t_{k_1+1}-\tau_1)}w(\tau_1)d\tau_1\int_{t_{k_2}}^{t_{k_2+1}}e^{\tilde{A}(t_{k_2+1}-\tau_2)}w(\tau_2)d\tau_2\right]\\
&=\left[\int_{t_{k_2}}^{t_{k_2+1}}\int_{t_{k_1}}^{t_{k_1+1}}e^{\tilde{A}(t_{k_1+1}-\tau_1)}\mathbb{E}[w(\tau_1)w(\tau_2)]e^{\tilde{A}^T(t_{k_2+1}-\tau_2)}d\tau_1d\tau_2\right]\\
&=\left[\int_{t_{k_2}}^{t_{k_2+1}}\int_{t_{k_1}}^{t_{k_1+1}}e^{\tilde{A}(t_{k_1+1}-\tau_1)}\delta(\tau_1-\tau_2)\tilde{Q}e^{\tilde{A}^T(t_{k_2+1}-\tau_2)}d\tau_1d\tau_2\right]\\
&=\begin{cases}\int_0^{t_{k+1}-t_k}e^{\tilde{A}s}\tilde{Q}e^{\tilde{A}^Ts}ds:=Q_k, & \text{如果}k_1=k_2=k,\\ 0, & \text{其他.}\end{cases}
\end{aligned}$$

若采样周期为 $\Delta t=t_{k+1}-t_k$，离散化后的过程噪声的协方差 $\operatorname{cov}(w_t,w_\tau)=\Sigma\delta(t-\tau)$；其中，

$$\begin{aligned}
\Sigma&=\int_0^{\Delta t}e^{\tilde{A}s}\tilde{Q}e^{\tilde{A}^Ts}ds\\
&=\int_0^{\Delta t}\begin{bmatrix}1&0&0&0\\ s&1&0&0\\ 0&0&1&0\\ 0&0&s&1\end{bmatrix}\begin{bmatrix}q_1&0&0&0\\ 0&q_2&0&0\\ 0&0&q_3&0\\ 0&0&0&q_4\end{bmatrix}\begin{bmatrix}1&s&0&0\\ 0&1&0&0\\ 0&0&1&s\\ 0&0&0&1\end{bmatrix}ds\\
&=\begin{bmatrix}\Delta tq_1&\frac{\Delta t^2}{2}q_1&0&0\\ \frac{\Delta t^2}{2}q_1&\Delta tq_2+\frac{\Delta t^3}{3}q_1&0&0\\ 0&0&\Delta tq_3&\frac{\Delta t^2}{2}q_3\\ 0&0&\frac{\Delta t^2}{2}q_3&\Delta tq_4+\frac{\Delta t^3}{3}q_3\end{bmatrix}
\end{aligned}$$

计算可得离散系统参数：

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ \Delta t & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \Delta t & 1 \end{bmatrix}, B = \begin{bmatrix} \frac{\Delta t}{m} & 0 \\ \frac{\Delta t^2}{2m} & 0 \\ 0 & \frac{\Delta t}{m} \\ 0 & \frac{\Delta t^2}{2m} \end{bmatrix}, \omega_k \sim N(0, \Sigma)$$

5 Numpy、Scipy 使用初探

numpy 是 python 编程中常用的扩展包，可以方便的处理数组，并提供了简单的矩阵运算的功能。scipy 是基于 numpy 开发的数学工具包，支持更复杂的矩阵运算。它们将在接下来的课程学习中大量使用。

5.1 版本和安装

课程使用的镜像预装了 python2.7.12 版本，并已经安装了对应的 numpy1.11.0 和 scipy1.2.2。由于 python2 现已停止支持，最新版本的 numpy 和 scipy 在 python2 中不可用，查阅文档时请注意版本区别。

5.2 创建数组对象

numpy 的基本数据结构是 array。下面是几种常用的创建 array 方法：

```
>>> import numpy as np
>>> import scipy.linalg as lnr
>>> A_list = [
...     [1,2,3],
...     [4,5,6]
... ]
>>> A = np.array(A_list)          # 从列表创建array
>>> print A
[[1 2 3]
 [4 5 6]]
>>> # 指定array数据类型为浮点型
>>> A = np.array(A_list, dtype = np.float)
>>> print A
[[ 1.  2.  3.]
 [ 4.  5.  6.]]
>>> np.zeros((2,3))      # 创建2行3列的零矩阵
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>> np.ones((2,3))       # 创建全1矩阵
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> np.eye(3)            # 创建3维单位方阵
array([[ 1.,  0.,  0.],
```

```
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
>>> # 创建随机数矩阵，每个元素独立服从U[0,1]
>>> np.random.random((3,2))
array([[ 0.00783261,  0.9901787 ],
       [ 0.67345818,  0.56650474],
       [ 0.61189143,  0.53739241]])
>>> np.random.random((3,2,2)) # 创建3层2行2列高维数组
array([[[ 0.63486732,  0.01954174],
        [ 0.73169491,  0.92889407]],

       [[ 0.19185609,  0.13109882],
        [ 0.48347947,  0.15575861]],

       [[ 0.51188658,  0.53638946],
        [ 0.54709691,  0.11810446]]])
>>> ls = [1,2,3,4]
>>> np.diag(ls) # 创建对角阵
array([[1, 0, 0, 0],
       [0, 2, 0, 0],
       [0, 0, 3, 0],
       [0, 0, 0, 4]])
>>> lnr.block_diag(np.ones((2,2)), 2*np.ones((3,2)))    # 创建分块对角阵
array([[ 1.,  1.,  0.,  0.],
       [ 1.,  1.,  0.,  0.],
       [ 0.,  0.,  2.,  2.],
       [ 0.,  0.,  2.,  2.],
       [ 0.,  0.,  2.,  2.]])
>>> A.repeat((2,3), axis=0)  # 矩阵元素重复
array([[1, 2, 3],
       [1, 2, 3],
       [4, 5, 6],
       [4, 5, 6],
       [4, 5, 6]])
>>> np.tile(A,[2,3])    # 矩阵重复
array([[1, 2, 3, 1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6, 4, 5, 6],
       [1, 2, 3, 1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6, 4, 5, 6]])
```

5.3 array 常用的成员变量

array 的成员变量记录了一些基本信息。

```
>>> A = np.random.random((4,3,2))
```

```
>>> A.dtype # array数据类型
dtype('float64')
>>> A.ndim  # array维数
3
>>> A.size  # array元素个数
24
>>> A.shape # array形状
(4, 3, 2)
>>> A.data  # array存放数据的地址（类似C中的指针）
<read-write buffer for 0x7f2096e61990, size 192, offset ...>
```

5.4 常用数组操作和元素访问

```
>>> #    一元数组操作
... A = np.array([
...      [1,2,3,4],
...      [5,6,7,8],
...      [9,10,11,12]
... ])
>>> A = A.astype(np.float64)    # 更改数据类型
>>> B = A.T      # 矩阵转置
>>> B
array([[  1.,   5.,   9.],
       [  2.,   6.,  10.],
       [  3.,   7.,  11.],
       [  4.,   8.,  12.]])
>>> # 维度拆分和合并。注意:直接改变A, 没有返回值
>>> A.resize(3,2,2)
>>> A
array([[[  1.,   2.],
        [  3.,   4.]],

       [[  5.,   6.],
        [  7.,   8.]],

       [[  9.,  10.],
        [ 11.,  12.]]])
>>> C = np.transpose(A, [0,2,1])    # 高维数组交换维度
>>> C
array([[[  1.,   3.],
        [  2.,   4.]],

       [[  5.,   7.],
        [  6.,   8.]],
```

```
       [[  9.,  11.],
        [ 10.,  12.]]])
>>>
>>> #    数组拼接
... D1 = np.array([[1,2,3],[4,5,6]])
>>> D2 = np.array([[7,8,9],[10,11,12]])
>>> np.stack([D1,D2],axis=0)    # 增维拼接
array([[[ 1,  2,  3],
        [ 4,  5,  6]],

       [[ 7,  8,  9],
        [10, 11, 12]]])
>>> np.stack([D1,D2],axis=1)
array([[[ 1,  2,  3],
        [ 7,  8,  9]],

       [[ 4,  5,  6],
        [10, 11, 12]]])
>>> np.stack([D1,D2],axis=2)
array([[[ 1,  7],
        [ 2,  8],
        [ 3,  9]],

       [[ 4, 10],
        [ 5, 11],
        [ 6, 12]]])
>>> np.concatenate([D1, D2], axis = 0)  # 同维度拼接
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
>>> np.concatenate([D1, D2], axis = 1)
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
>>>
>>> #    访问数组元素
... A = np.array([range(20)])
>>> A.resize((4,5))
>>> A
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> A[3,4]       # 访问第3行第4列的元素
```

```
19
>>> A[0,:]       # 访问第0行的所有元素
array([0, 1, 2, 3, 4])
>>> A[:,-2]      # 访问倒数第2列的元素
array([ 3,  8, 13, 18])
>>> A[1,1:4]     # 访问第1行，第1-4列的元素
array([6, 7, 8])
>>> # 把A的成员copy给B。注意:A的data成员被直接赋给B,
>>> # A和B的数据实际共用同一段地址。(浅copy)
>>> B = A
>>> D = np.copy(A)  # 将A的数据copy给D，不同址(深copy)
>>> C = B.T      # C是B的转置
>>> C[0,:] = C[1,:] # 把C第1行的值赋给第0行
>>> C            # C的值改变
array([[ 1,  6, 11, 16],
       [ 1,  6, 11, 16],
       [ 2,  7, 12, 17],
       [ 3,  8, 13, 18],
       [ 4,  9, 14, 19]])
>>> # 注意，转置不改变数据地址，只是以不同顺序访问，
>>> # 因此改变C也会改变同样使用这块地址的B
>>> B
array([[ 1,  1,  2,  3,  4],
       [ 6,  6,  7,  8,  9],
       [11, 11, 12, 13, 14],
       [16, 16, 17, 18, 19]])
>>> # 直接“=”赋值得到的B与A的data成员指向相同地址，
>>> # 因此A的值也改变
>>> A
array([[ 1,  1,  2,  3,  4],
       [ 6,  6,  7,  8,  9],
       [11, 11, 12, 13, 14],
       [16, 16, 17, 18, 19]])
>>> D   # 深copy得到的D没有改变
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

5.5 矩阵基本运算

```
# 矩阵基本运算
A = np.array(range(1,13))
A.resize((4,3))
```

```
np.mean(A,axis=0)    # 计算每行的平均值
np.min(A,axis=1)     # 计算每列的最小值
np.max(A)            # 计算整个数组的最大值
B = np.array(range(1,13))
B.resize((4,3))
A+B      # 矩阵加法。减法同
A*B      # 按位相乘，同matlab中的.*。除法同
A.dot(B.T)  # 矩阵乘法。

# 一般来说位运算需要两矩阵维度完全相同，
# 但服从广播(broadcast)规则时，运算仍合法，如下例。
A = np.array(range(24))
A.resize((4,3,2))
C = np.array(range(8))
C.resize((4,1,2))
A*C # A,C维数相同，长度不同的维度，C的长为1，运算合法
D = np.array(range(6))
D.resize((3,2))
A*D # A，D维数不同，D的维度与A的最后几个维度相同，合法
C*D # 同时应用上面两个规则，运算合法。
```

5.6 进阶矩阵运算

利用 scipy 的 linalg 模块，可以实现一些略复杂的矩阵运算。如下例。另，scipy 还有 optimize、signal 等模块，具体参阅具体参阅http://scipy.github.io/devdocs/reference/index.html。

```
import scipy.linalg as lnr
A=np.random.randint(0,10,(4,4))
r = np.linalg.matrix_rank(A)    #矩阵秩
d = lnr.det(A)          # 行列式
n = lnr.norm(A)         # 矩阵范数
n = lnr.norm(A[:,1])    # 向量范数
B = lnr.pinv(A)         # 伪逆
val, vec_r = lnr.eig(A) # 特征值和特征向量
U, s, Vh = lnr.svd(A)   # 奇异值分解
C = lnr.expm(A)         # 矩阵指数运算
```

5.7 array 数据的可视化

matplotlib 是最常用的 python 数据可视化工具。它的优点是自由度高，功能强大，但缺点是接口非常底层，代码量稍大。这里举例介绍 matplotlib 的简单应用。

```
from matplotlib import pyplot as plt
A = np.random.multivariate_normal([0,1], [[1,0.3],[0.3,1]], (1024))
fig, ax = plt.subplots(2,2)             # 绘制含2x2子图的图窗
```

```
ax[0,0].scatter(A[:,0], A[:,1])          # 散点图
ax[0,0].set_title("Scatter chart")       # 设置图标题
ax[0,1].hist2d(A[:,0], A[:,1], bins=20) # 用hist2d实现的热度图
ax[0,1].set_title("Heatmap")             #       hist可以作直方图
ax[1,0].plot(range(20), A[:20,0])        # 折线图
ax[1,0].set_title("Line chart")
ax[1,1].boxplot([A[:,0], A[:,1]])        # 箱线图
ax[1,1].set_title("Boxplot")
fig.show()                          # 展示图窗
```

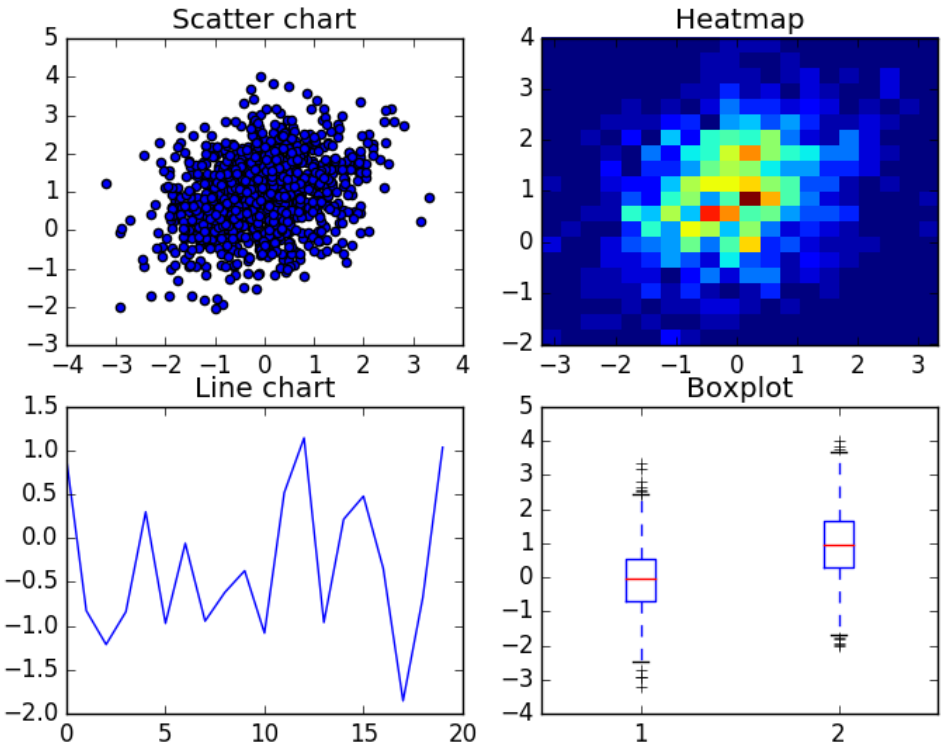

Figure 4: matplotlib 画图

6 练习：python 下质点运动学模型实现演示

以下是使用 python 模拟质点在带有噪声的环境中运动的脚本，请根据上面推导的运动模型与噪声模型将其补全

```
import numpy as np
from matplotlib import pyplot as plt
```

```
# particle's weight (kg)
m = 1
# Sampling period
dt = 0.01
# total simulation time
time = 30
# Variances of the process noise: to be completed
q1 =
q2 =
q3 =
q4 =
Sigma = np.zeros((4,4))
Sigma[0, 0] =
Sigma[0, 1] =
Sigma[1, 0] =
Sigma[1, 1] =
Sigma[2, 2] =
Sigma[2, 3] =
Sigma[3, 2] =
Sigma[3, 3] =


'''
@param X_0 initial states [vx_0, px_0, vy_0, py_0]
@param F   force
@return X  states
'''
def forward_dynamics(X_0, F):
    Len = F[:, 0].size
    Dim = X_0.size
    X = np.zeros(shape=(Len, Dim))
    X[0, :] = X_0
    # Matrix A: to be completed
    A =
    # Matrix B: to be completed
    B =
    # Simulate moving
    for i in range(1, Len):
        # Dynamic model: pending completed
        X[i, :] =
        # Add noise
        w = np.random.multivariate_normal( \
            np.zeros(Dim), Sigma)
        X[i, :] += w
    return X
```

```
# time instances
t = np.arange(0, int(time/dt)) * dt

# force input
F_x = -np.sin(t)
F_y = np.cos(t)
F = np.stack([F_x, F_y], axis=1)

# initial state X_0 = ([vx_0, px_0, vy_0, py_0])
X_0 = np.array([1, 0, 0, 0])
X = forward_dynamics(X_0, F)

# draw picture
fig, ax = plt.subplots(2, 2)
ax[0, 0].plot(t, F_x)
ax[0, 0].set_title("t-Fx")
ax[1, 0].plot(t, X[:, 1])
ax[1, 0].set_title("t-px")
ax[0, 1].plot(t, F_y)
ax[0, 1].set_title("t-Fy")
ax[1, 1].plot(t, X[:, 3])
ax[1, 1].set_title("t-py")

plt.figure(2)
plt.plot(X[:, 1], X[:, 3])
plt.title("Moving Trajactory")
plt.show()
```