# CMU 16-745 最优控制笔记

吕翔宇 (Xiangyu Lyu)

lvxiangyu11@gmail.com

TU Darmstadt, Germany

GitHub: lvxiangyu11

2025 年 5 月 8 日

# 目录

# 1 引言

这里是 CMU 16-745 最优控制课程的笔记。我将涵盖课程中的重要概念、定理和例子。课程目标

1. 分析动力系统的稳定性。

2. 设计稳定平衡和轨迹的LQR控制器。

3. 使用离线轨迹优化设计非线性系统的轨迹。

4. 使用在线凸优化实现模型预测控制。

5. 了解随机性和模型不稳定性的影响。

6. 无最优模型时直接优化反馈策略。

写作风格备注：（TODO：终稿注释掉!）额外的公式推导使用浅绿色背景框额外的例子使用浅蓝色背景框代码使用浅灰色背景

# 2 动力学简介

这一节面向不了解控制学、动力学的读者，以一个单摆系统为例，快速的介绍连续和离散动力学系统的基础，包括稳定性分析、线性系统和非线性系统。

## 2.1 连续时间的动力学系统

Continuous-Time Dynamics System (CTDS) 是一个描述系统状态随时间变化的数学模型。它通常由以下方程表示：

$$\dot{x} = f(x, u) \tag{1}$$

其中，$x(t) \in \mathbb{R}^n$ 是系统状态，$u(t) \in \mathbb{R}^m$ 是控制输入，$f$ 是一个描述系统的动态函数，展示了系统如何根据$u$而演化。

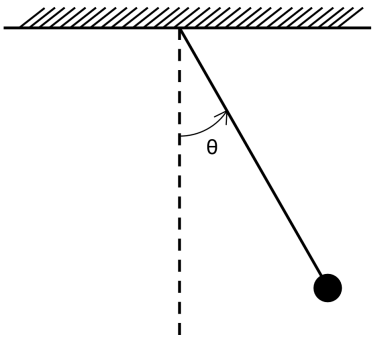


图 1: 简单单摆的物理模型。图中展示了摆球的受力情况，包括重力 $mg$ 和拉力 $\vec{T}$，以及摆球的运动方向和角度 $\theta$ 的关系。

对于1中的单摆系统，其状态x可以用角度q和速度v来表示。我们可以将其状态表示为：

$$x = \begin{bmatrix} q \\ v \end{bmatrix} \tag{2}$$

q并不总是一个vector。configuration并不必须是一个vector，并且速度并不必须是configuration的导数。并且，仅在对系统的动力学描述是平滑的时候，才可以说这个系统的描述是有效的。

**连续动态系统的简单单摆例子** 考虑1中的单摆系统。我们可以用以下方程来描述它的动力学：

$$ml^2\ddot{\theta} + mglsin(\theta) = \tau \tag{3}$$

其推导过程如下：

**从拉格朗日方程推导动力学方程推导*** 在这一节中，我们使用拉格朗日方法推导单摆系统的动力学方程。首先，单摆的动能为：

$$T = \frac{1}{2}ml^2\dot{\theta}^2$$

势能为：

$$V = mgl(1 - \cos\theta)$$

其中，$m$是质量，$l$是摆长，$\theta$是角度，$\dot{\theta}$是角速度，$g$是重力加速度。拉格朗日量$L$为动能$V$与势能$L$之差，即：

$$L = T - V = \frac{1}{2}ml^2\dot{\theta}^2 - mgl(1 - \cos\theta)$$

拉格朗日方程的一般形式为：

$$\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{\theta}}\right) - \frac{\partial L}{\partial \theta} = 0$$

我们计算拉格朗日量的偏导数：

1. 对于$\frac{\partial L}{\partial \dot{\theta}}$，有：

$$\frac{\partial L}{\partial \dot{\theta}} = ml^2\dot{\theta}$$

2. 对于$\frac{\partial L}{\partial \theta}$，有：

$$\frac{\partial L}{\partial \theta} = mgl\sin\theta$$

将这些代入拉格朗日方程，得到：

$$\frac{d}{dt}\left(ml^2\dot{\theta}\right) - mgl\sin\theta = 0$$

即：

$$ml^2\ddot{\theta} + mgl\sin\theta = 0$$

如果引入控制输入$\tau$，方程变为：

$$ml^2\ddot{\theta} + mgl\sin\theta = \tau$$

这个方程描述了单摆的动力学行为，其中$\tau$是控制输入，能够影响单摆的角加速度。最终，我们得到了单摆的动力学方程：

$$ml^2\ddot{\theta} + mgl\sin\theta = \tau$$

则系统的配置$q$可以通过单摆的角度$\theta$来表示，速度$v$可以通过角速度$\dot{\theta}$来表示。并且考虑到控制量$u$，通过施加到系统中的力矩(torque) $\tau$ 来控制单摆的运动。则状态可

以表示为：

$$x = \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix} \tag{4}$$

则其速度$\dot{x}$可以表示为：

$$\dot{x} = \begin{bmatrix} \dot{\theta} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} \dot{\theta} \\ -\frac{g}{l} sin(\theta) + \frac{\tau}{ml^2} \end{bmatrix} \tag{5}$$

则系统动态方程可以写为$f(x, u)$：

$$\dot{x} = f(x, u) = \begin{bmatrix} \dot{\theta} \\ -\frac{g}{l} sin(\theta) + \frac{\tau}{ml^2} \end{bmatrix} \tag{6}$$

则这里状态是一个流形(manifold)的函数 $x \in \mathbb{S}^1 x \mathbb{R}$ 是一个圆柱，表示了系统的状态随时间的变化。

### 2.1.1 控制映射系统

很多系统可以定义为一个控制映射系统。这是一个上述动力系统的特殊形式，其中控制输入通过一个映射矩阵$G$来影响系统。

$$\dot{x} = f_0(x) + G(x)u \tag{7}$$

对于上述简单单摆模型，可以将其表示为：

$$\dot{x} = \begin{bmatrix} \dot{\theta} \\ -\frac{g}{l} sin(\theta) + \frac{\tau}{ml^2} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \theta \\ \dot{\theta} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{1}{ml^2} \end{bmatrix} \tau \tag{8}$$

### 2.1.2 Manipulator动力学

Manipulator是一个可以在空间中移动的物体。它的动力学可以用以下方程表示（参考《现代机器人学机构、规划与控制》的第八章，读者感兴趣可以自行阅读，此处略）：

$$M(q)\dot{v} + C(q, v) = B(q)u \tag{9}$$

其中，$M(q)$是质量矩阵，$C(q, v)$是科里奥利力矩阵和重力项，$B(q)$是控制输入矩阵。这个方程描述了Manipulator的动力学行为，其中$u$是控制输入，能够影响Manipulator的运动。则configuration的变化可以表示为：

$$\dot{q} = G(q)v \tag{10}$$

其中，$G(q)$是一个映射矩阵，将速度$v$映射到configuration的变化$\dot{q}$。则系统的运动学可以描述为：

$$\dot{x} = f(x, y) = \begin{bmatrix} G(q)v \\ -M(q)^{-1}(B(q) - C) \end{bmatrix} \tag{11}$$

在简单单摆系统中 $M(q) = ml^2$, $C(q, v) = mgl\sin(\theta)$, $B = I$, $G = I$。

所有机械系统都可以表述为这个形式，这是因为这个形式是一个描述Euler-Lagerange方程的不同形式(Kinetic energy - potential energy $L = T - V$)。

$$L = \frac{v^T M(q) v}{2} - V(q) \tag{12}$$

### 2.1.3 线性系统

线性系统（Linear System）是表述控制问题和设计控制器的通用表述方法。我们知道可以相对简单的解决一个线性系统，线性系统描述为：

$$\dot{x} = A(t)x + B(t)u \tag{13}$$

其中，$A(t)$和$B(t)$是时间变化的矩阵。线性系统的解可以通过求解常微分方程来获得。线性系统的稳定性分析通常使用特征值和特征向量的方法。线性系统的控制器设计通常使用状态反馈和观测器设计的方法。

我们通常通过在现在状态附近的线性化来近似线性化非线性系统：

$$\dot{x} = f(x, u) \tag{14}$$

其中 $A = \frac{\partial f}{\partial x}, B = \frac{\partial f}{\partial u}$

### 2.1.4 平衡点 equilibria

equilibria指的是系统将会保持的位置，即一阶稳态。

$$\dot{x} = f(x, u) = 0 \tag{15}$$

在代数层面，这表示动力方程的根。对于简单单摆系统，平衡点可以表示为：

$$\dot{x} = f(x, u) = \begin{bmatrix} \dot{\theta} \\ -\frac{g}{l} sin(\theta) + \frac{\tau}{ml^2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{16}$$

**平衡点的求解** 考虑一个简单的问题，我们如何找到一个平衡点？我们可以通过求解以下方程来找到平衡点：

$$\dot{x} = f(x, u) = \begin{bmatrix} \dot{\theta} \\ -\frac{g}{l} sin(\theta) + \frac{\tau}{ml^2} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \Rightarrow \frac{1}{ml^2} u = \frac{g}{l} sin(\theta) \Rightarrow u = mgl\sin(\theta) \tag{17}$$

带入$\theta = \pi/2$, 得到$u = mgl$ 一般来说，可以将寻找平衡点的问题转化为求解一个非线性方程组的问题。

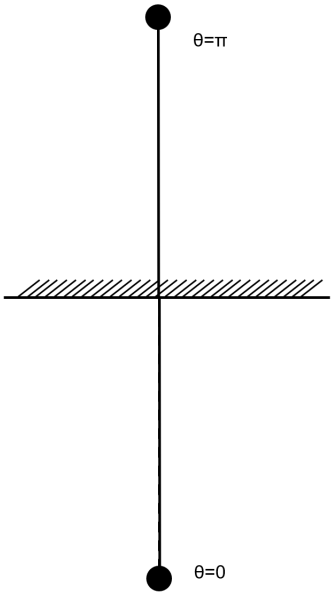


图 2: 图中展示了单摆系统的平衡点。平衡点是指系统在没有外部扰动的情况下，能够保持静止或匀速运动的状态。

### 2.1.5 平衡点的稳定性分析

在知道了如何求平衡点后，我们需要知道，如果对系统施加一个小的扰动，系统是否会回到平衡点。考虑一个1维系统，在这个例子中，当$\frac{\partial f}{\partial x} < 0$时系统是稳定的。当$\frac{\partial f}{\partial x} > 0$时系统是不稳定的。我们可以通过线性化来分析系统的稳定性。对于一个线性系统，我们可以通过求解特征值来判断系统的稳定性。对于一个非线性系统，我们可以通过求解雅可比矩阵的特征值来判断系统的稳定性，因为从这个点越推越远。这个$\frac{\partial f}{\partial x} < 0$的区域是一个吸引子(Attractor)又称basin of attraction of system(系统吸引盆地)，而$\frac{\partial f}{\partial x} > 0$的区域是一个排斥子(Repeller)。

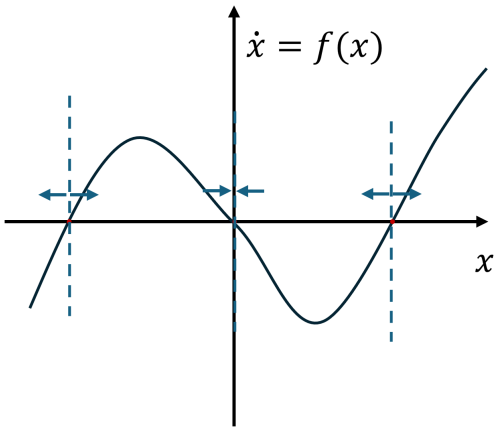


图 3: 图中展示了1维系统的稳定性分析。图中对于$\dot{x} = 0$有三个解，也就是说有三个平衡点，但是只有中间的点满足$\frac{\partial f}{\partial x} < 0$，而左右两侧$f(x)$对$x$的偏导都是大于0，系统下一步将采取正向的行动，导致系统不稳定。

推广到高维系统，这个$\frac{\partial f}{\partial x} < 0$则为是一个Jacobian matrix。为了研究系统的稳定性，我们可以对Jacobian matrix进行特征值分解。对于一个线性系统，我们可以通过求解特征值来判断系统的稳定性。即，如果每个特征值的实部都小于0，则系统是渐近稳定的；如果有一个特征值的实部大于0，则系统是不稳定的；如果所有特征值的实部都等于0，则系统是边界稳定的。可以用公式描述为：

$$Re(eig(\frac{\partial f}{\partial x})) = \begin{cases} < 0 & \text{渐近稳定} \\ > 0 & \text{不稳定} \\ = 0 & \text{边界稳定} \end{cases} \tag{18}$$

Jacobian矩阵定义为：

$$\frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{bmatrix} \tag{19}$$

在上述简单单摆系统中

$$f(x) = \begin{bmatrix} \dot{\theta} \\ -\frac{g}{l}sin(\theta) \end{bmatrix} \Rightarrow \frac{\partial f}{\partial x} = \begin{bmatrix} 0 & 1 \\ -\frac{g}{l}cos(\theta) & 0 \end{bmatrix} \tag{20}$$

考虑平衡点$\theta = 0$，我们可以得到Jacobian矩阵的特征值：

$$\frac{\partial f}{\partial x} = \begin{bmatrix} 0 & 1 \\ -\frac{g}{l} & 0 \end{bmatrix} \Rightarrow \lambda^2 + \frac{g}{l} = 0 \Rightarrow \lambda = \pm i\sqrt{\frac{g}{l}} \tag{21}$$

由于有一个特征值的实部大于0，表示系统在该点是一个**鞍点**，即系统在该点附近的运动是发散的。

考虑平衡点$\theta = \pi$，我们可以得到Jacobian矩阵的特征值：

$$\frac{\partial f}{\partial x} = \begin{bmatrix} 0 & 1 \\ -\frac{g}{l} & 0 \end{bmatrix} \Rightarrow \lambda^2 - \frac{g}{l} = 0 \Rightarrow \lambda = 0 \pm \sqrt{\frac{g}{l}} \tag{22}$$

该点特征值实部为0，表示系统在该点是一个**中心点**，即系统在该点附近的运动是周期性的（不考虑空气摩擦力）。对于纯虚数特征值系统，这个系统叫做边缘稳定（marginally stable）。可以添加damping来使得系统稳定，如$u = -K_d\dot{\theta}$（摩擦力）。

## 2.2 离散时间的动力学系统

在这一节中，我们将介绍

· 连续的常微分方程（Ordinary Differential Equations, ODEs）

· 离散动力系统稳定性分析

在通常情况下，我们无法直接从$\dot{x} = f(x)$中求出$x(t)$的解析解，则无法直接求出系统的状态随时间的变化。我们可以通过在离散时间内通过数值的方法近似表示$x(t)$。并且，离散时间模型可以捕获连续ODEs无法捕获的动态行为（比如，离散时间模型可以捕获系统的周期性行为，接触）。

### 2.2.1 正向欧拉法

离散时间动力学系统（Discrete-Time Dynamics System, DTDS）是一个描述系统状态随时间变化的数学模型。通常用于模拟计算，如模拟器等。它通常由以下方程表示：

$$x_{k+1} = f_{discrete}(x_k, u_k) \tag{23}$$

**正向欧拉法*** 这里将介绍正向欧拉法，并举一个例子。
用一个最简单的离散化表示，即正向欧拉法（Forward Euler Method）：

$$x_{k+1} = x_k + h f_{continuous}(x_k, u_k) \tag{24}$$

其中，$h$是离散化的时间步长。这个方程表示在时间步长$h$内，系统状态$x_k$和控制输入$u_k$的关系。$f_{continuous} = \frac{\partial f(x,u)}{\partial t}$
**手算例子：** 考虑一个简单的线性系统 $\dot{x} = -2x$，其离散化形式为：

$$x_{k+1} = x_k + h(-2x_k) = (1 - 2h)x_k \tag{25}$$

假设初始条件为 $x_0 = 1$ 且步长 $h = 0.1$，则计算前几个离散时间点的状态：

- 对于 $k = 0$，$x_0 = 1$；
- 对于 $k = 1$，$x_1 = (1 - 2 \times 0.1) \times 1 = 0.8$；
- 对于 $k = 2$，$x_2 = (1 - 2 \times 0.1) \times 0.8 = 0.64$；
- 对于 $k = 3$，$x_3 = (1 - 2 \times 0.1) \times 0.64 = 0.512$。

我们可以看到，系统状态随着时间步的增加逐渐减小。

用一个最简单的离散化表示，即正向欧拉法（Forward Euler Method）：

$$x_{k+1} = x_k + h f_{continuous}(x_k, u_k) \tag{26}$$

其中，$h$是离散化的时间步长。这个方程表示在时间步长$h$内，系统状态$x_k$和控制输入$u_k$的关系。$f_{continuous} = \frac{\partial f(x,u)}{\partial t}$

**离散时间动力学系统的简单单摆例子*** 继续单摆系统

单摆的连续时间动力学系统可以通过以下方程描述：

$$\dot{x} = f(x,u) = \begin{bmatrix} \dot{\theta} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} \dot{\theta} \\ -\frac{g}{l}\sin(\theta) + \frac{\tau}{ml^2} \end{bmatrix}$$

其中：$\theta$ 为单摆的角度（配置变量），$\dot{\theta}$ 为角速度（速度变量），$\ddot{\theta}$ 为角加速度。对于离散时间动力学系统，我们采用正向欧拉法进行离散化。设定离散时间步长为 $h$，那么在每个时间步内，系统的状态 $x_k = [\theta_k, \dot{\theta}_k]$ 通过以下公式更新：

$$x_{k+1} = x_k + h \cdot f(x_k, u_k)$$

将连续时间的动力学方程 $f(x,u)$ 代入上式，我们得到：

$$x_{k+1} = \begin{bmatrix} \theta_k \\ \dot{\theta}_k \end{bmatrix} + h \cdot \begin{bmatrix} \dot{\theta}_k \\ -\frac{g}{l}\sin(\theta_k) + \frac{\tau_k}{ml^2} \end{bmatrix}$$

这个递推公式可以用来计算每个时间步的状态。

具体而言，我们可以分别更新角度和角速度：

$$\theta_{k+1} = \theta_k + h \cdot \dot{\theta}_k$$

$$\dot{\theta}_{k+1} = \dot{\theta}_k + h\left(-\frac{g}{l}\sin(\theta_k) + \frac{\tau_k}{ml^2}\right)$$

这个离散化公式描述了如何在每个时间步内，根据当前的角度和角速度，通过正向欧拉法更新单摆的状态。

针对简单单摆系统，设置$l = m = 1, h = 0.1 or 0.01$将会爆炸（blow up）。

以下实现了单摆系统的离散时间模拟代码，缩短模拟步长可以看到blow up被延缓了，但是仍会blow up。

```python
# [discrete_time_dynamic_forward_euler.py]
def pendulum_dynamics(x):
    l = 1.0
    g = 9.81

    theta = x[0] # 角度
    theta_dot = x[1] # 角速度
    theta_ddot = -(g/l) * np.sin(theta) # 角加速度，由重力引起的

    return np.array([theta_dot, theta_ddot])

def pendulum_forward_euler(fun, x0, Tf, h):
    t = np.arange(0, Tf + h, h)
```

```
    x_hist = np.zeros((len(x0), len(t))) # 状态历史记录
    x_hist[:, 0] = x0    # 初始状态

    for k in range(len(t) - 1): # 迭代计算每个时间步的状态
        x_hist[:, k+1] = x_hist[:, k] + h * fun(x_hist[:, k]) # 使用欧拉法更新状态
    return x_hist, t

if __name__ == "__main__":
    # 初始条件和参数
    x0 = np.array([0.1, 0])  # 初始角度和角速度
    Tf = 10.0  # 模拟总时间(秒)'
    h = 0.01  # 时间步长'

    # 计算单摆运动'
    x_hist, t_hist = pendulum_forward_euler(pendulum_dynamics, x0, Tf, h)

    # 创建可视化对象并显示动画'
    visualizer = PendulumVisualizer(pendulum_length=1.0)
    visualizer.visualize(x_hist, t_hist)
```

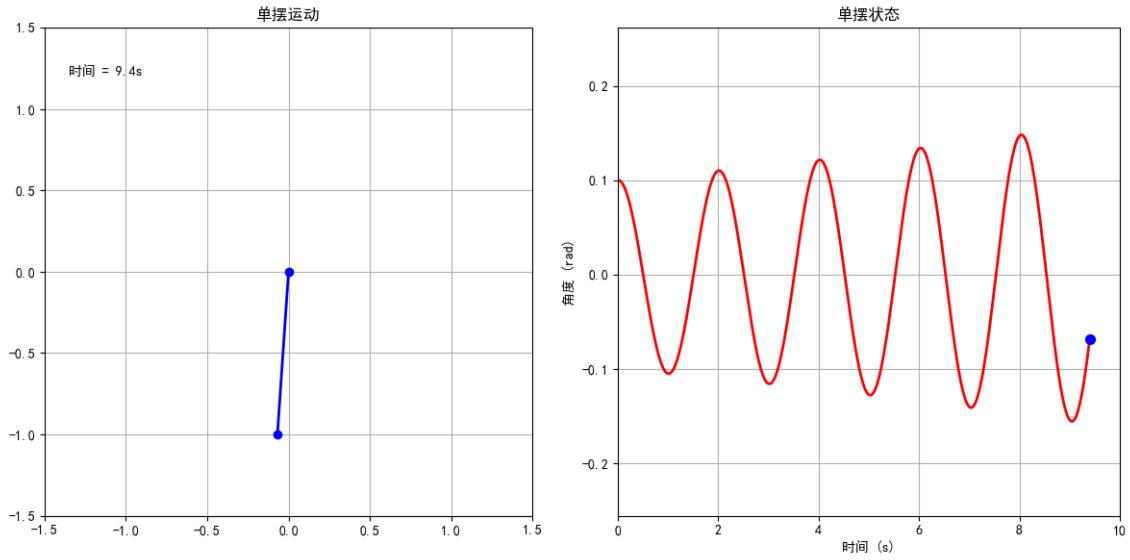


图 4: 图中展示了欧拉方法模拟的单摆系统的运动轨迹。可以看到，随着时间的推移，单摆逐渐blow up，说明该方法是数值不稳定的

**正向欧拉法的稳定性分析** 正向欧拉法的稳定性分析可以通过考虑离散时间系统的特征值来进行。对于线性系统，正向欧拉法的稳定性取决于离散化步长 $h$ 和系统矩阵 $A$ 的特征值。具体而言，如果所有特征值的模长小于1，则系统是渐近稳定的；如果有一个特征值的模长大于1，则系统是不稳定的。回顾之前判断系统是否稳定，$Re(eig(\frac{\partial f}{\partial x}))$与0的关系。对于离散时间，动力学系统可以迭代映射为：

$$x_N = f_d(f_d(f_d(...f_d(x_0))...)) \tag{27}$$

考虑线性和chain rule，有：

$$\frac{\partial x_N}{\partial x_0} = \frac{\partial f_d}{\partial x_0}\frac{\partial f_d}{\partial x_1}\cdots\frac{\partial f_d}{\partial x_{N-1}}\bigg|_{x_0} = A_d^N \tag{28}$$

其中 $A_d$ 表示线性化的点的偏导 $\frac{\partial f_d}{\partial x}$，并且在迭代中，每个点的偏导数都是相同的。

稳定点是$x=0$（可以通过修改坐标系达成），系统的稳定性意味着：

$$\lim_{N\to\infty} A_d^N x0 = 0, \forall x_0 \in \mathbb{R}^n \Rightarrow \lim_{N\to\infty} A_d^N = 0 \Rightarrow |eig(A_d)| < 1 \tag{29}$$

如果$|eig(A_d)| > 1$，则系统不稳定，迭代之后那些大于1的特征值将会占据主导地位，导致系统不稳定。

等价的，$eig(A_d)$必须在复数平面上位于单位圆内。

下面开始分析简单单摆系统在欧拉方法离散化后的稳定性，h取0.1, g=9.81, l=1.0。:

$$A_d = \frac{f_d}{x_k} = I + hA_{\text{continuous}} = I + h\begin{bmatrix} 0 & 1 \\ -\frac{g}{l}\cos(\theta) & 0 \end{bmatrix} \tag{30}$$

$$\Rightarrow \text{eig}(A_d\big|_{\theta=0}) = 1 \pm 0.313i \Rightarrow |\text{eig}(A_d)| = 1.313 > 1 \Rightarrow \text{不稳定} \tag{31}$$

我们可以将h和eigenvalue的关系绘制成图表，其代码如下：

```python
# [euler_eigenvalue_stability.py]
# 此处仅展示关键代码

h_values = np.linspace(0.01, 0.5, 100)
eigenvalues = []

for h in h_values:
    A_d = A_discrete(h, theta)
    eigs = eigvals(A_d)
    eigenvalues.append(eigs)

# 转换为numpy数组便于处理
eigenvalues = np.array(eigenvalues)

# 计算特征值的模
magnitudes = np.abs(eigenvalues)
```

可以从5中看出

- 欧拉法对简单单摆系统的离散化在任何正的时间步长下都是不稳定的，因为特征值的模总是大于1。
- 时间步长越大，特征值的模越大，系统越不稳定。
- 系统的两个特征值在实部上逐渐分离
- 所有点都在单位圆外，表明系统在所有正的h值下都是不稳定的

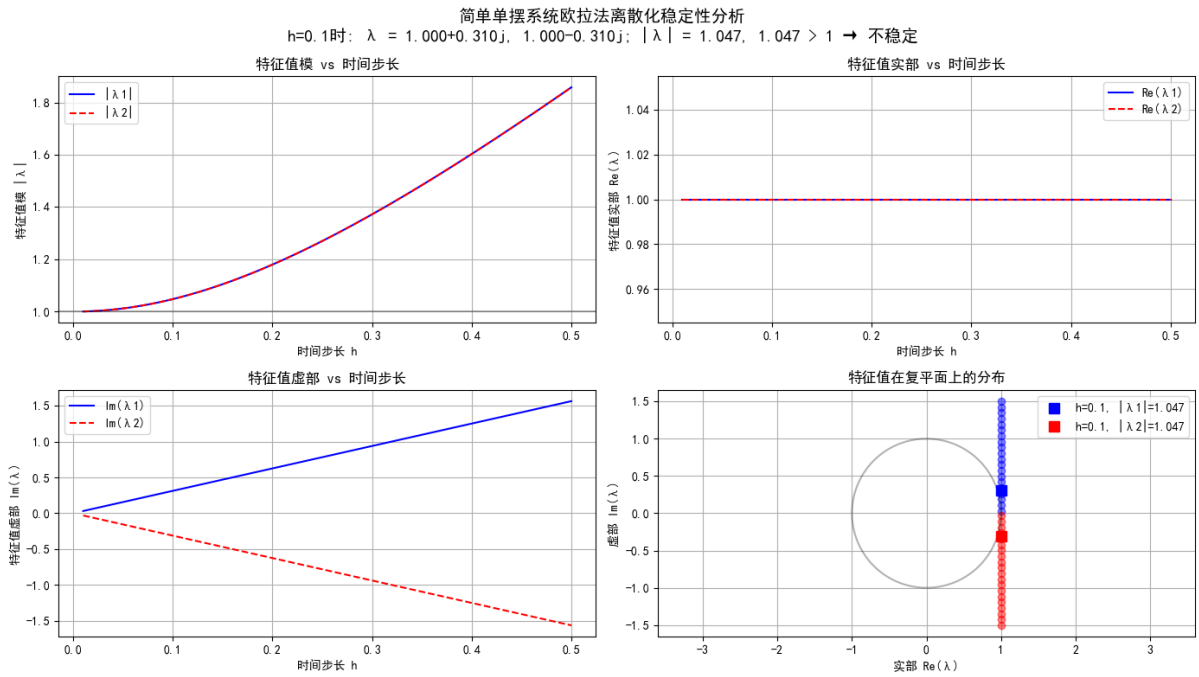


图 5: 图中展了简单单摆系统在欧拉法离散化后的稳定性分析，通过观察不同时间步长下特征值的变化来判断系统的数值稳定性。

**结论**

- 谨慎使用正向欧拉法，尤其是对于非线性系统。正向欧拉法在某些情况下可能会导致数值不稳定，尤其是在系统具有强非线性或刚性时。对于这些情况，可以考虑使用更高级的数值积分方法，如RK4方法或隐式欧拉法等。
- 在稳定点时谨慎检测的能量行为，是否是保守系统，是否有耗散。
- 因为欧拉方法说到底是在$t$时刻的导数$f(x_k, u_k)$，相对于$x$总会有一个overshoot导致系统越来越blow up。所以避免使用正向欧拉法来模拟非线性系统的动力学行为。

### 2.2.2 RK4方法

4th order Runge-Kutta method（RK4）是一种常用的数值积分方法，用于求解常微分方程（ODEs）。它通过在每个时间步内计算多个斜率来提高精度。RK4方法的基本思想是通过四个斜率的加权平均来近似ODE的解。

**RK4方法*** 这里将介绍RK4方法，并举一个例子。
RK4方法（四阶龙格-库塔法）是一种常用的数值积分方法，用于求解常微分方程。它的基本思想是通过在每个时间步内计算四个斜率来估算系统状态的变化，

进而得到更精确的解。

RK4方法的基本步骤如下：

1. 计算四个斜率：

$$
\begin{aligned}
k_1 &= hf(x_k, u_k) \quad \text{(evaluate at beginning)} \\
k_2 &= hf\left(x_k + \frac{1}{2}k_1, u_k\right) \quad \text{(evaluate at midpoint 1)} \\
k_3 &= hf\left(x_k + \frac{1}{2}k_2, u_k\right) \quad \text{(evaluate at midpoint 2)} \\
k_4 &= hf(x_k + k_3, u_k) \quad \text{(evaluate at end)}
\end{aligned}
$$

   其中，$h$是时间步长，$f(x_k, u_k)$是系统的动力学方程，$x_k$和$u_k$分别表示在时刻$k$时的状态和控制变量。

2. 更新状态：

$$x_{k+1} = x_k + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

   通过这一步，RK4方法将四个斜率加权平均，得到更精确的状态更新公式。

RK4方法通过计算四个斜率来提高精度，相比于正向欧拉法，RK4方法在每个时间步内计算了更多的信息，从而能够更准确地逼近常微分方程的解。RK4方法的误差是$O(h^4)$，比欧拉法的$O(h^2)$更高效，适合用于高精度要求的数值计算。

**手算例子：** 现在，我们通过一个简单的例子来手动计算RK4方法的应用。假设我们要求解以下简单的常微分方程：

$$\frac{dx}{dt} = -2x$$

初始条件为$x(0) = 1$，我们选择时间步长$h = 0.1$。我们将使用RK4方法计算$x$在$t = 0.1$时刻的值。

1. 计算四个斜率：

$$
\begin{aligned}
k_1 &= 0.1 \cdot (-2 \cdot 1) = -0.2 \\
k_2 &= 0.1 \cdot (-2 \cdot (1 + \frac{1}{2} \cdot (-0.2))) = -0.19 \\
k_3 &= 0.1 \cdot (-2 \cdot (1 + \frac{1}{2} \cdot (-0.19))) = -0.19 \\
k_4 &= 0.1 \cdot (-2 \cdot (1 + (-0.19))) = -0.162
\end{aligned}
$$

2. 更新状态：

$$x_1 = 1 + \frac{1}{6}(-0.2 + 2(-0.19) + 2(-0.19) + (-0.162)) = 0.818$$

因此，使用RK4方法得到的$x(0.1)$的近似值为0.818。

通过这个手算例子，我们可以看到RK4方法比简单的欧拉法能够提供更精确的结果，尤其是在步长较大的情况下。

```python
# [discrete_time_dynamic_RK4.py]
import numpy as np
from src.PendulumVisualizer import *
def pendulum_dynamics(x):
    l = 1.0
    g = 9.81

    theta = x[0]
    theta_dot = x[1]
    theta_ddot = -(g/l) * np.sin(theta)

    return np.array([theta_dot, theta_ddot])

def pendulum_rk4(fun, x0, Tf, h):
    t = np.arange(0, Tf + h, h)

    x_hist = np.zeros((len(x0), len(t)))
    x_hist[:, 0] = x0

    for k in range(len(t) - 1):
        x = x_hist[:, k]

        k1 = fun(x)
        k2 = fun(x + h/2 * k1)
        k3 = fun(x + h/2 * k2)
        k4 = fun(x + h * k3)

        x_hist[:, k+1] = x + h/6 * (k1 + 2*k2 + 2*k3 + k4)

    return x_hist, t

if __name__ == "__main__":
    x0 = np.array([0.1, 0])
    Tf = 10.0
    h = 0.01

    x_hist, t_hist = pendulum_rk4(pendulum_dynamics, x0, Tf, h)

    visualizer = PendulumVisualizer(pendulum_length=1.0)
    visualizer.visualize(x_hist, t_hist)
```

**RK4稳定性分析** 通过计算RK4的雅可比矩阵的特征值我们可以得到：

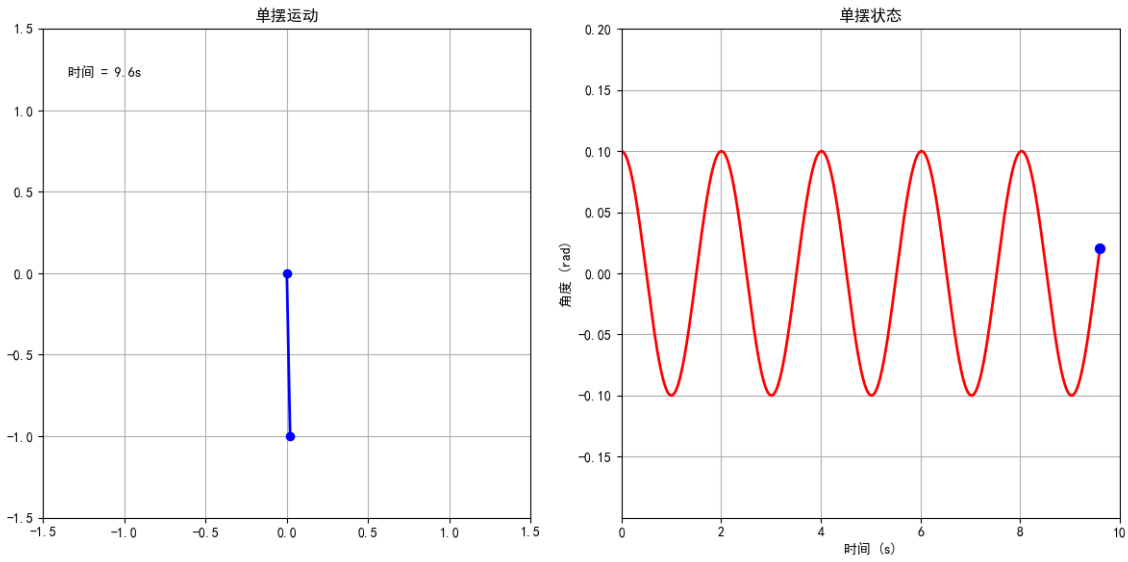


图 6: 图中展示了RK4模拟的单摆系统的运动轨迹。可以看到，随着时间的推移，单摆系统的运动轨迹更加平滑且稳定。

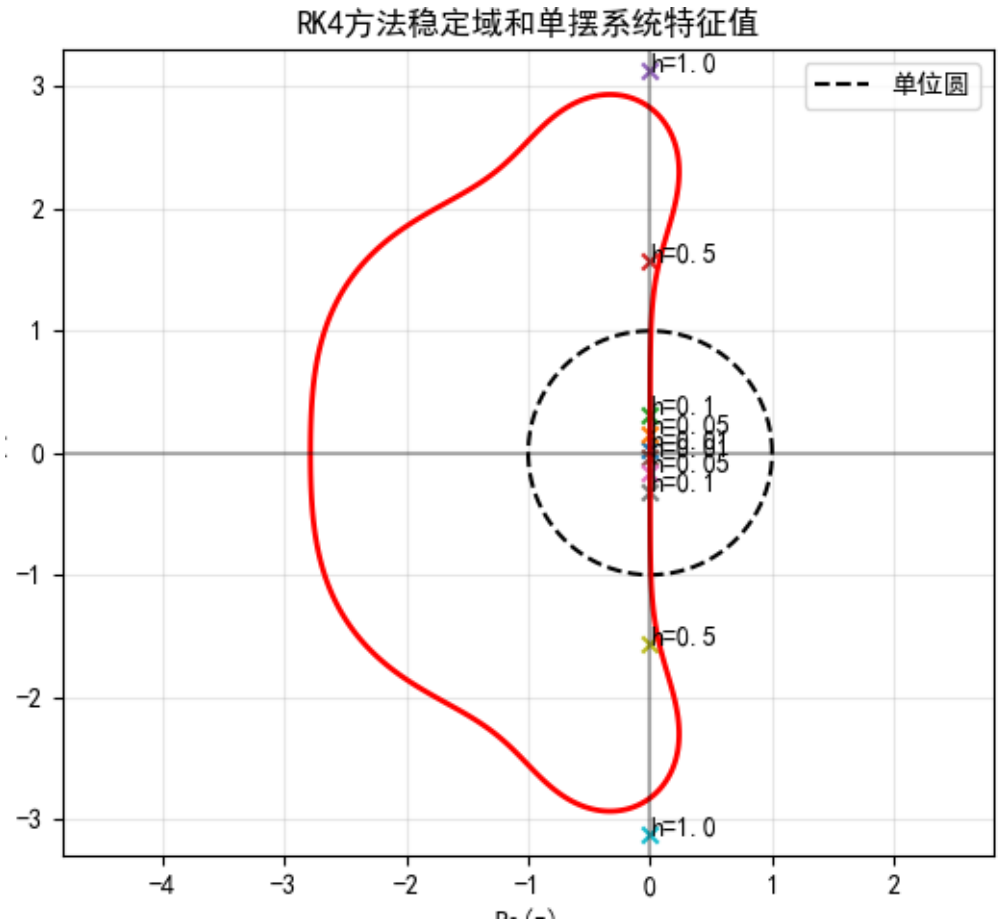


图 7: 图中展示了不同步长下，RK4方法的特征值，可以看到落在单位圆中的量。

RK4 方法稳定性分析

**RK4 方法稳定性分析实验结果**

- 单摆线性化系统的特征值：

$$\lambda = \begin{bmatrix} 0 + 3.1321j \\ 0 - 3.1321j \end{bmatrix}$$

- 步长 $h = 0.01$ 下，RK4 方法的雅可比矩阵特征值：

$$\lambda_J = \begin{bmatrix} 0.9995 + 0.0313j \\ 0.9995 - 0.0313j \end{bmatrix}$$

- 放大因子（特征值绝对值）：

$$|\lambda_J| = \begin{bmatrix} 0.999999999993 \\ 0.999999999993 \end{bmatrix}$$

**不同步长下的稳定性分析**

- 对于不同的步长 $h$，RK4 方法的稳定性如下所示：

$h = 0.01$

$$\lambda = \begin{bmatrix} 0 + 3.1321j \\ 0 - 3.1321j \end{bmatrix}, \quad \lambda_J = \begin{bmatrix} 0.9995 + 0.0313j \\ 0.9995 - 0.0313j \end{bmatrix}, \quad |\lambda_J| = \begin{bmatrix} 0.999999999993 \\ 0.999999999993 \end{bmatrix}$$

$h = 0.05$

$$\lambda = \begin{bmatrix} 0 + 3.1321j \\ 0 - 3.1321j \end{bmatrix}, \quad \lambda_J = \begin{bmatrix} 0.9878 + 0.1560j \\ 0.9878 - 0.1560j \end{bmatrix}, \quad |\lambda_J| = \begin{bmatrix} 0.999999897875 \\ 0.999999897875 \end{bmatrix}$$

$h = 0.1$

$$\lambda = \begin{bmatrix} 0 + 3.1321j \\ 0 - 3.1321j \end{bmatrix}, \quad \lambda_J = \begin{bmatrix} 0.9514 + 0.3081j \\ 0.9514 - 0.3081j \end{bmatrix}, \quad |\lambda_J| = \begin{bmatrix} 0.999993524289 \\ 0.999993524289 \end{bmatrix}$$

$h = 0.5$

$$\lambda = \begin{bmatrix} 0 + 3.1321j \\ 0 - 3.1321j \end{bmatrix}, \quad \lambda_J = \begin{bmatrix} 0.0244 + 0.9259j \\ 0.0244 - 0.9259j \end{bmatrix}, \quad |\lambda_J| = \begin{bmatrix} 0.9262 \\ 0.9262 \end{bmatrix}$$

$h = 1.0$

$$\lambda = \begin{bmatrix} 0 + 3.1321j \\ 0 - 3.1321j \end{bmatrix}, \quad \lambda_J = \begin{bmatrix} 0.1048 \pm 1.9889j \end{bmatrix}, \quad |\lambda_J| = \begin{bmatrix} 1.9916 \\ 1.9916 \end{bmatrix}$$

**符号说明**

- $\lambda$：线性化系统的特征值，反映系统本身的动态特性；
- $J$：RK4 离散系统下的雅可比矩阵，用于描述线性近似；
- $\lambda_J$：RK4 方法下的雅可比矩阵特征值；
- $|\lambda_J|$：特征值的模（放大因子），表示数值解是否随时间增长或衰减。

```python
    # [discrete_time_dynamic_RK4.py]
    # 此处仅展示关键代码
def compute_rk4_jacobian_eigenvalues(h):
    # 线性系统 x' = Ax 的情况下，A的特征值为 lambda
    # 对于单摆在小角度近似下，线性化后的A矩阵为
    l = 1.0
    g = 9.81
    A = np.array([[0, 1], [-g/l, 0]])  # 线性化的单摆动力学

    # 计算A的特征值
    eigen_A = np.linalg.eigvals(A)
    print("单摆线性化系统的特征值:", eigen_A)

    # 计算RK4方法的放大矩阵特征值
    # R(z) = 1 + z + z^2/2 + z^3/6 + z^4/24

    # 计算不同特征值的放大因子
    amplification_factors = []
    eigenvalues = []

    # 对每个特征值计算放大因子
    for lam in eigen_A:
        z = h * lam
        R = 1 + z + z**2/2 + z**3/6 + z**4/24
        amplification_factors.append(np.abs(R))
        eigenvalues.append(R)

    print(f"步长 h = h下RK4方法的雅可比矩阵特征值:", eigenvalues)
    print(f"放大因子(特征值绝对值):", amplification_factors)

    return eigen_A, eigenvalues, amplification_factors
```

**稳定性判据与特征值分析** 设系统矩阵 $A$ 的特征值为 $\lambda_i$，步长为 $h$，则有：

$$z_i = h \cdot \lambda_i$$

RK4 迭代中相应的放大因子为：

$$\rho_i = R(z_i)$$

其模长：

$$|\rho_i| = |R(z_i)|$$

判断稳定性标准：

- 若 $|\rho_i| < 1$，则该模态衰减（数值稳定）；
- 若 $|\rho_i| = 1$，则该模态保持（边界稳定）；
- 若 $|\rho_i| > 1$，则该模态发散（不稳定）。

这是因为在数值方法（如 Runge-Kutta 方法）中，我们**并不是直接求解原始的微分方程** $\dot{x} = Ax$，而是通过一种离散时间的迭代形式来逼近它。
我们逐步解释为何会有：

$$z_i = h \cdot \lambda_i, \quad \rho_i = R(z_i)$$

**1. 原始系统的连续解形式** 设系统为：

$$\dot{x}(t) = Ax(t)$$

其解析解为：

$$x(t) = e^{At}x(0)$$

也就是说，**系统演化由指数矩阵** $e^{At}$ 控制。

**2. 数值方法的离散迭代** RK4 等数值方法的本质是逼近这个指数矩阵的一种方式。对于一个离散步长 $h$，我们用某个**近似演化矩阵** $R(hA)$ 来代替 $e^{hA}$，从而实现一步迭代：

$$x_{n+1} = R(hA)x_n$$

**3. 为什么用特征值来简化？** 矩阵 $A$ 的特征值 $\lambda_i$ 描述了系统在各特征方向上的增长或衰荡速率。在分析数值稳定性时，我们关注每个模态（也就是特征值对应的方向）在一步后是否被**放大或抑制**。
由于特征值和矩阵函数的可交换性（在 $A$ 可对角化时）：

$$\text{如果 } Av_i = \lambda_i v_i, \text{ 则 } R(hA)v_i = R(h\lambda_i)v_i$$

也就是说，**数值迭代对每个特征模态的作用可以简化为一个复数放大因子**：

$$\rho_i = R(h\lambda_i)$$

其中：

- $h\lambda_i = z_i$：当前模态对应的无量纲时间步；
- $R(z_i)$：RK4 稳定性函数，在 $z_i$ 的值，给出该模态的“放大比例”；
- $|R(z_i)|$：判断该模态是否稳定（小于1表示衰减，等于1表示保持，大于1表示爆炸）。

**4. 直觉类比** 你可以将 $\lambda_i$ 理解为一个“频率”或“增长率”，而 $h$ 是我们离散模拟的“采样间隔”。那么：

$$z_i = h\lambda_i$$

用这个“频率率（时间步长）”来捕捉这个模态，数值方法能否稳定逼近它？
如果 $z_i$ 落在 RK4 的稳定区域，则该模态可以被良好模拟；否则，就会出现误差爆炸。

### 2.2.3 反向欧拉法

pybullet、mujoco使用的正是反向欧拉方法。对于隐式ODEs，例如：

$$f_d(x_{k+1}, x_k, u_k) = 0 \tag{32}$$

这表示在每个时间步内，系统的状态$x_{k+1}$和控制输入$u_k$之间的关系是隐式的。我们可以通过迭代的方法来求解这个方程。反向欧拉法（Backward Euler Method）是一种常用的数值积分方法，用于求解隐式ODEs。它的基本思想是通过在每个时间步内计算一个隐式方程来估算系统状态的变化，进而得到更精确的解。

$$x_{k+1} = x_k + hf(x_{k+1}) \tag{33}$$

相较于forward euler，这里的backward euler计算$f(x)$是用的未来的时间$x_{k+1}$而不是当前的时间$x_k$。于是我们可以通过反向欧拉法来模拟系统：

$$f_d(x_{k+1}, x_k, u_k) = x_{k+1} - x_k - hf(x_{k+1}, u_k) = 0 \tag{34}$$

可以将这个问题转换为一个$x_{k+1}$时刻的root-finding问题，这个问题将在之后介绍。类似于机器人学中计算inverse dynamic的Newton-Euler方法？（TODO：添加引用）

**反向欧拉法直觉理解** backward euler 通过离散化向系统添加了damping，即产生了undershoot（相对于euler方法的overshoot），使得系统在每个时间步内都能保持稳定。通过在每个时间步内计算一个隐式方程，反向欧拉法能够更好地捕捉系统的动态行为，尤其是在处理刚性系统时。反向欧拉法的误差是$O(h)$，比欧拉法的$O(h^2)$更高效，适合用于高精度要求的数值计算。但是这导致了由此构建的模拟器与现实存在很大的差距！值得注意的是:

- 显式方法通常比隐式方法更稳定。
- 对于forward simulation，解决隐式ODEs代价很高。
- 对于很多直接的轨迹优化方法，不再如此高代价。

### 2.2.4 离散控制输入

对于系统，我们前文论述了状态通过$x(t)$来描述，并考虑了连续与离散时间的状态。现在我们来考虑系统的控制输入$u(t)$的离散化表示。

$$u(t) = u_k, t_k \leq t < tk + 1 \tag{35}$$

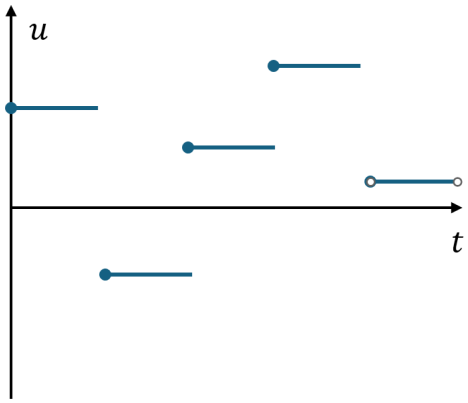


图 8: 这是一个zero-order hold的示意图。它表示在每个时间步内，控制输入$u(t)$保持不变，直到下一个时间步到来。这个方法在数字控制系统中非常常见，因为它可以简化控制器的设计和实现。

零阶保持$u(t) = u_n, t_n \leq t \leq T_{n+1}$

而一阶保持$u(t) = u_n + (\frac{u_{k+1}-u_k}{h}(t - t_n))$，其中$h$是时间步长。一阶保持是一个更好的代替方法。它表示在每个时间步内，控制输入$u(t)$是一个线性函数，直到下一个时间步到来。这个方法在数字控制系统中也很常见，因为它可以更好地捕捉系统的动态行为。

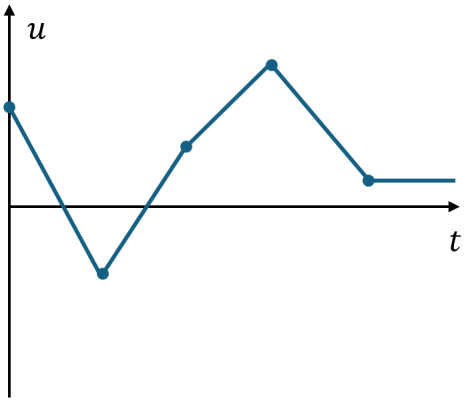


图 9: 这是一个first-order hold的示意图。它表示在每个时间步内，控制输入$u(t)$是一个线性函数，直到下一个时间步到来。这个方法在数字控制系统中也很常见，因为它可以更好地捕捉系统的动态行为。通过使用一阶保持方法，我们可以更好地捕捉系统的动态行为，尤其是在处理快速变化的系统时。与零阶保持相比，一阶保持方法能够更准确地模拟系统的响应，并减少控制输入的突变，从而提高系统的稳定性和性能。

高阶保持，使用高阶的多项式来近似控制输入。然而在Bang-Bang控制中（比如说击球的这一瞬间，球换向快速弹回），两个极端值快速变换，其最优控制率不是光滑的，此时高阶保持甚至不如零阶保持

# 3 数值优化基础

这一节将介绍常用的数值优化方法。

# Appendix