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 17 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。

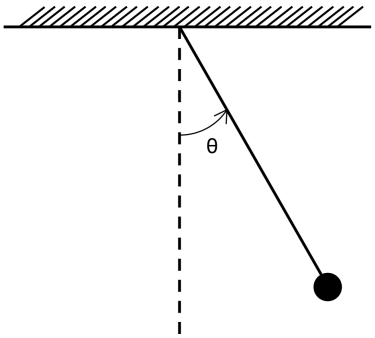


图 1: 简单单摆的物理模型。图中展示了摆球的受力情况，包括重力 mg 和拉力 $\vec{T}$，以及摆球的运动方向和角度 θ 的关系。

对于**??**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**??**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，θ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τ，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τ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θ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) τ 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。

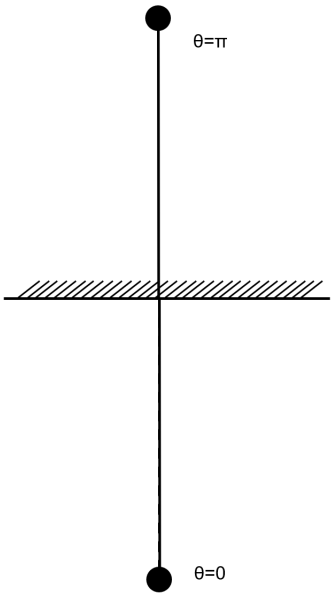


图 2: 图中展示了单摆系统的平衡点。平衡点是指系统在没有外部扰动的情况下，能够保持静止或匀速运动的状态。

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)。

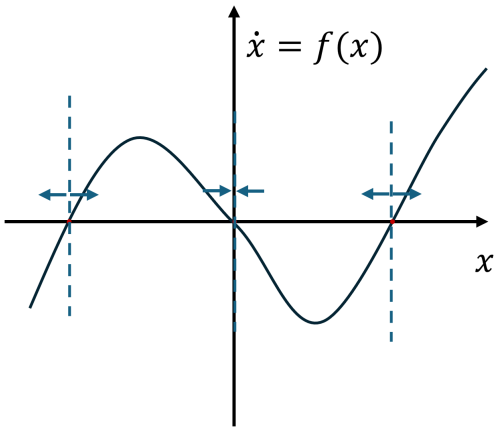


图 3: 图中展示了1维系统的稳定性分析。图中对于$\dot{x} = 0$有三个解，也就是说有三个平衡点，但是只有中间的点满足$\frac{\partial f}{\partial x} < 0$，而左右两侧$f(x)$对$x$的偏导都是大于0，系统下一步将采取正向的行动，导致系统不稳定。

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：θ 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)
```

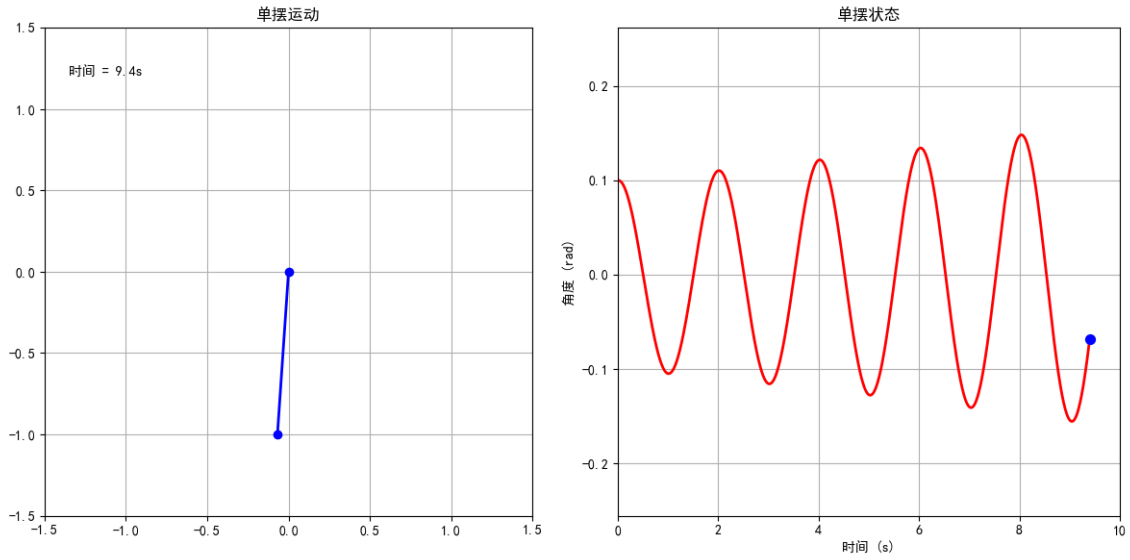


图 4: 图中展示了欧拉方法模拟的单摆系统的运动轨迹。可以看到，随着时间的推移，单摆逐渐blow up，说明该方法是数值不稳定的

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。

:

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

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

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)
```

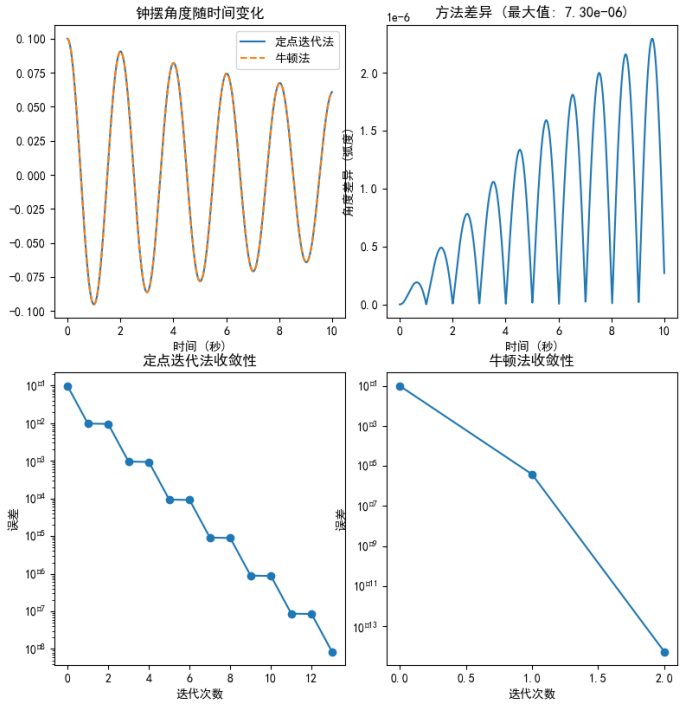


图 5: 图中展了简单单摆系统在欧拉法离散化后的稳定性分析，通过观察不同时间步长下特征值的变化来判断系统的数值稳定性。

可以从??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

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：

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

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)
```

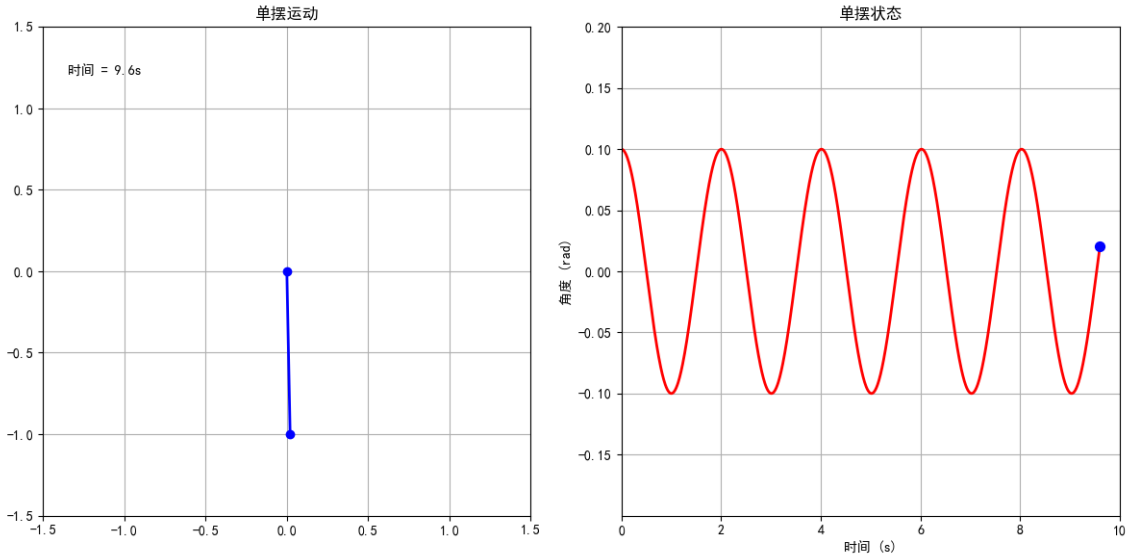


图 6: 图中展示了RK4模拟的单摆系统的运动轨迹。可以看到，随着时间的推移，单摆系统的运动轨迹更加平滑且稳定。

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：

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：

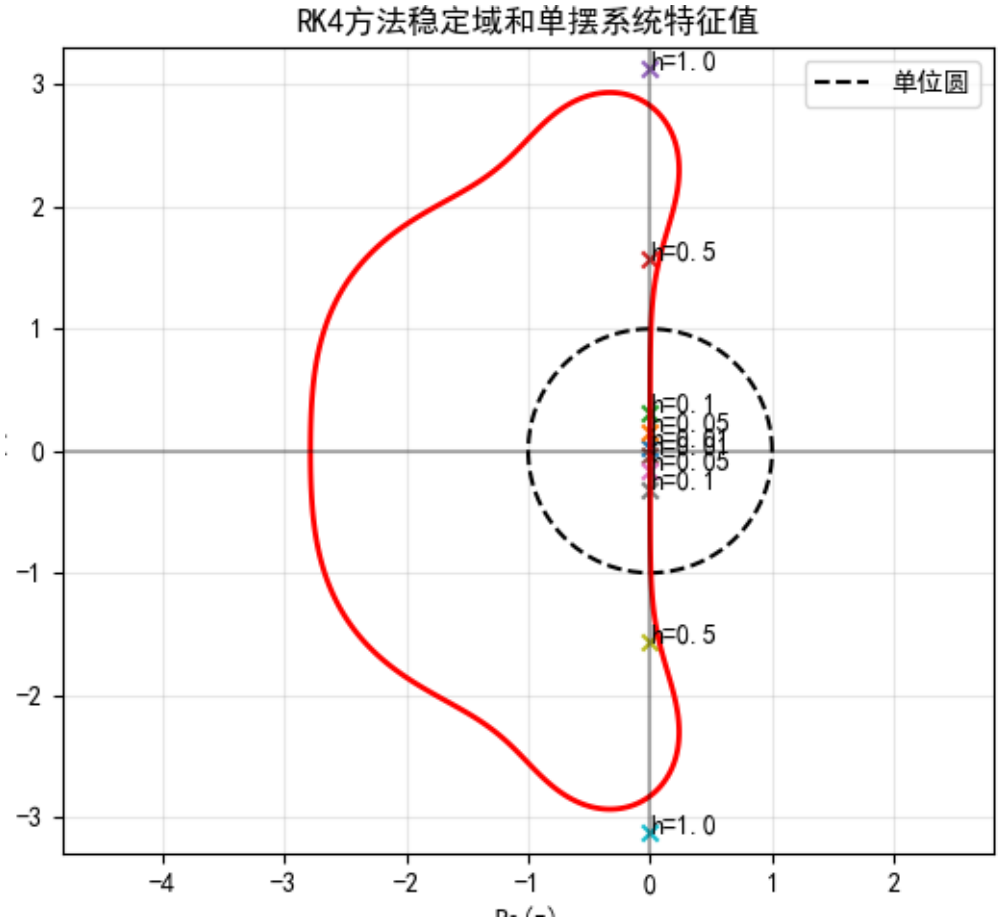


图 7: 图中展示了不同步长下，RK4方法的特征值，可以看到落在单位圆中的量。

$h = 0.01$

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

$h = 0.05$

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

$h = 0.1$

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

$h = 0.5$

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

$h = 1.0$

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

符号说明

- λ：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；
- λ_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
```

```
25            amplification_factors.append(np.abs(R))
26            eigenvalues.append(R)
27
28        print(f"步长 h = h下RK4方法的雅可比矩阵特征值:", eigenvalues)
29        print(f"放大因子(特征值绝对值):", amplification_factors)
30
31        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 λ_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 λ_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 λ_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？

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。

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

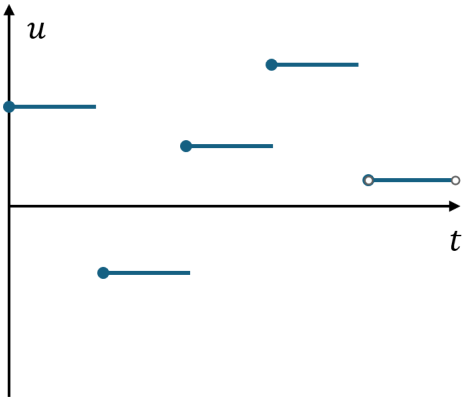


图 8: 这是一个zero-order hold的示意图。它表示在每个时间步内，控制输入$u(t)$保持不变，直到下一个时间步到来。这个方法在数字控制系统中非常常见，因为它可以简化控制器的设计和实现。

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。

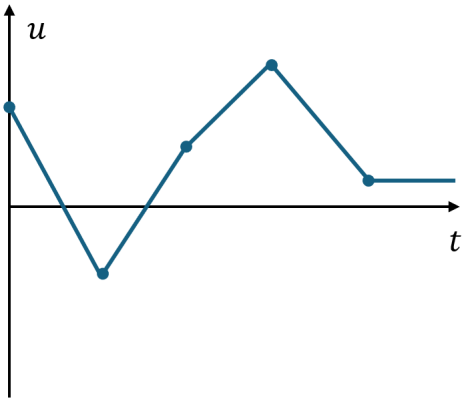


图 9: 这是一个first-order hold的示意图。它表示在每个时间步内，控制输入$u(t)$是一个线性函数，直到下一个时间步到来。这个方法在数字控制系统中也很常见，因为它可以更好地捕捉系统的动态行为。通过使用一阶保持方法，我们可以更好地捕捉系统的动态行为，尤其是在处理快速变化的系统时。与零阶保持相比，一阶保持方法能够更准确地模拟系统的响应，并减少控制输入的突变，从而提高系统的稳定性和性能。

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。从一个 root-finding 问题开始，然后介绍

- 不动点迭代方法
- 牛顿方法
  - 最小化
  - 充分条件
  - 正则化
  - overshooting 问题

3.1 符号声明

这一节介绍多个符号规定，记住就好。

一元函数导数维度 考虑 $f(x)\colon \mathbb{R}^n \to \mathbb{R}$，则

$$\frac{\partial f}{\partial x} \in \mathbb{R}^{1\times n} \Rightarrow \text{梯度是行向量。}$$

因为 $\frac{\partial f}{\partial x}$ 是将 Δx 映射到 Δf 的线性算子。并且通过泰勒展开我们可以得到：

$$f(x+\Delta x) \approx f(x) + \frac{\partial f}{\partial x}\Delta x,$$

因为 Δx 为列向量，所以如果要让$f(x+\Delta x)$合法，则$\frac{\partial f}{\partial x}$必然是行向量，

多元函数偏导（雅可比）维度 类似地，设 $g(y)\colon \mathbb{R}^m \to \mathbb{R}^n$，其雅可比为

$$J := \frac{\partial g(y)}{\partial y} \in \mathbb{R}^{n\times m}, \quad J = \begin{pmatrix} \frac{\partial g_1}{\partial y_1} & \frac{\partial g_1}{\partial y_2} & \cdots & \frac{\partial g_1}{\partial y_m} \\ \frac{\partial g_2}{\partial y_1} & \frac{\partial g_2}{\partial y_2} & \cdots & \frac{\partial g_2}{\partial y_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial g_n}{\partial y_1} & \frac{\partial g_n}{\partial y_2} & \cdots & \frac{\partial g_n}{\partial y_m} \end{pmatrix}$$

因为：

$$g(y+\Delta y) \approx g(y) + \frac{\partial g}{\partial y}\Delta y.$$

中Δy是$m*1$，所以雅可比必然是$n*m$。

链式法则（Chain Rule） ：下面公式演示了复合函数的一阶泰勒展开

$$f(g(y+\Delta y)) \approx f(g(y)) + \frac{\partial f}{\partial x}\cdot\frac{\partial g}{\partial y}\cdot\Delta y.$$

nabla 记号 ∇

$$\nabla f(x) = \left(\frac{\partial f}{\partial x}\right)^{\top} \in \mathbb{R}^{n\times 1} \quad \text{列向量},$$

$$\nabla^2 f(x) = \frac{\partial}{\partial x}(\nabla f(x)) = \frac{\partial^2 f}{\partial x^2} \in \mathbb{R}^{n\times n}.$$

3.2 Root Finding问题

考虑根问题（Root Finding）：给定函数 $f(x)$，希望找到 x^* 使得

$$f(x^*) = 0 \quad \Rightarrow \quad x^* \text{ 是根。}$$

例如：连续时间动力系统的平衡点

$$\dot{x} = f(x), \quad \text{令 } f(x^*) = 0 \Rightarrow x^* \text{ 是平衡。}$$

这个问题与不动点问题密切相关，即求解满足

$$f(x^*) = x^*$$

的 x^*，例如离散时间动力系统的平衡点。

不动点迭代（Fixed-Point Iteration） 最朴素的做法是不断迭代：

$$x_{k+1} = f(x_k),$$

若动态系统是稳定的，则不断前向模拟该系统，将会收敛到某个平衡点。

不动点收敛性分析 考虑映射 $x_{k+1} = f(x_k)$，若存在不动点 x^* 满足 $f(x^*) = x^*$，我们希望该迭代能收敛到 x^*。

定理（Banach 不动点定理）： 设 $f : \mathbb{R}^n \to \mathbb{R}^n$ 是压缩映射，即存在 $0 < L < 1$，使得对任意 x, y，有

$$\|f(x) - f(y)\| \le L\|x - y\|,$$

则 f 有唯一不动点 x^*，且对任意初始点 x_0，迭代 $x_{k+1} = f(x_k)$ 收敛于 x^*。

收敛性证明： 记 x^* 为不动点，考虑误差 $e_k := x_k - x^*$，则

$$e_{k+1} = x_{k+1} - x^* = f(x_k) - f(x^*) \Rightarrow \|e_{k+1}\| \le L\|e_k\| \le L^2\|e_{k-1}\| \le \cdots \le L^k\|e_0\|.$$

因此有：

$$\|x_k - x^*\| \le L^k\|x_0 - x^*\| \to 0 \quad \text{当 } k \to \infty.$$

这说明只要映射满足 Lipschitz 常数 $L < 1$，就能保证收敛性，且为线性收敛。

梯度意义下的不动点 在梯度下降的意义下，若将梯度流离散化：

1. **前向欧拉：**

$$x_{k+1} = x_k - h\nabla f(x_k),$$

这就是梯度下降（Gradient Descent）的离散形式。

注：可添加动画或直观演示。

2. **后向欧拉：**类似于近似梯度法，也可看作是一种近似的「梯度流」方法（continuous limit）。

欧拉法中的根寻找问题：以非线性动力系统为例 我们考虑如下非线性常微分方程（ODE）：

$$\frac{dx}{dt} = f(x) = 0.5x\left(1 - \frac{x}{3}\right), \quad x(0) = 0.5.$$

该系统的平衡点满足 $f(x^*) = 0$，即 $x^* = 0$ 和 $x^* = 3$，这也正是一个 ‘根寻找（Root Finding）问题‘。

数值方法一：前向欧拉法（Forward Euler）

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

收敛性分析：

若 f 在邻域内满足 Lipschitz 条件，即存在 $L > 0$ 使得

$$|f(x) - f(y)| \leq L|x - y|,$$

则前向欧拉法收敛。当 $hL < 1$ 时，具有稳定性。

对应 Python 代码： 关键部分的代码

```python
# [optimization\forward_backward.py]
def forward_euler(x0, t_end, h):
    t = np.arange(0, t_end, h)
    x = np.zeros_like(t)
    x[0] = x0

    for i in range(1, len(t)):
        x[i] = x[i-1] + h * f(x[i-1])  # Forward Euler update

    return t, x
```

数值方法二：后向欧拉法（Backward Euler）

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

我们需要在每一步中解非线性方程

$$x = x_k + h \cdot 0.5x\left(1 - \frac{x}{3}\right) \Rightarrow \frac{h}{6}x^2 + \left(1 - \frac{h}{2}\right)x - x_k = 0$$

收敛性分析：

设 $g(x) = x_k + hf(x)$，可视为不动点迭代 $x = g(x)$。

若 f 的导数有界，满足

$$|g'(x)| = |hf'(x)| < 1,$$

则根据 Banach 不动点定理，迭代过程收敛。

对应 Python 代码： 关键部分的代码

```python
# [optimization\forward_backward.py]
def backward_euler(x0, t_end, h):
    t = np.arange(0, t_end, h)
    x = np.zeros_like(t)
    x[0] = x0

    for i in range(1, len(t)):
        # Solve: x = x_prev + h * 0.5 * x * (1 - x / 3)
```

```
            a = h / 6
            b = (1 - h / 2)
            c = -x[i-1]
            discriminant = b**2 - 4*a*c
            if discriminant >= 0:
                x[i] = (-b + np.sqrt(discriminant)) / (2*a)  # Take positive root
            else:
                x[i] = x[i-1]  # No real root, fallback

        return t, x
```

方法比较总结：

- **前向欧拉法：** 显式方法，简单高效，但稳定性对步长敏感
- **后向欧拉法：** 隐式方法，稳定性强，每步需解根问题
- **根问题体现：** 后向欧拉需解方程 g(x) = x，可视为不动点问题，若满足收敛条件可用不动点迭代或解析求解

为什么梯度下降在大规模问题中更受欢迎？ 在大规模机器学习中，梯度下降（Gradient Descent）相比牛顿方法（Newton's Method）更常用，原因包括：

- **计算更便宜：** 牛顿法需要计算 Hessian 并求逆，代价为 $\mathcal{O}(n^3)$，而梯度下降只需计算一阶导数。（将在后面的公式中看到这个复杂度）
- **高维不可接受：** 在高维情况下，牛顿方法由于计算复杂度过高，不适合使用。
- 此外，对于非凸函数（non-convex function）：
  - **多个平衡点（equilibria）：** 若希望找到全局最小值，需要在多个平衡点之间选择，等价于 NP 难问题。
  - **几乎需要穷举搜索（exhaustive search）：** 没有启发式或凸性结构支持时，只有遍历所有不动点才能确定最优解。
  - **仅适用于稳定的不动点：** 牛顿迭代法仅在初始点足够靠近稳定的不动点时才有效；否则会发散。
  - **收敛速度慢：** 特别在函数曲率变化大或存在鞍点时，牛顿法的收敛性无法保证。

3.2.1 Newton 方法

Newton 方法的基本思想是对 $f(x)$ 作线性近似，并设为零从而求解 Δx，用于迭代更新。

由**线性近似**，在当前点 x 附近，采用一阶泰勒展开：

$$f(x+\Delta x) \approx f(x) + \frac{\partial f}{\partial x}\Delta x.$$

然后立即可以得到牛顿方法的两个步骤

- **令近似为零，转为 root-finding 问题，得到第一步，计算**

$$f(x) + \frac{\partial f}{\partial x}\Delta x = 0 \quad \Rightarrow \quad \Delta x = -\left(\frac{\partial f}{\partial x}\right)^{-1} f(x)$$

注：这里求雅可比的逆是$O(n^3)$

- **第二步，更新变量：**

$$x \leftarrow x + \Delta x,$$

- **反复迭代直到收敛。**

注： 若导数矩阵不可逆或病态，可采用正则化策略：

如添加零平方项以求伪逆（pseudo-inverse）： 即 Tikhonov regularization。

Backward Euler 步的固定点迭代示例 考虑使用 backward Euler 步来构造一个固定点迭代过程。目标是求解如下形式的不动点：

$$x_n = x_0 + hf(x_n),$$

其中 h 是步长参数。

- 初始化：$x_n = x_0$
- 误差定义：

$$\text{err} = \|x_0 + hf(x_n) - x_n\|_2$$

- 迭代至收敛：

$$\begin{aligned} \text{while err} > \epsilon: \quad & x_n \leftarrow x_0 + hf(x_n) \\ & \text{err} = \|x_0 + hf(x_n) - x_n\|_2^2 \end{aligned}$$

该迭代形式对应于隐式欧拉法或非线性方程的固定点迭代形式，可看作是牛顿方法的近似替代方案。

Backward Euler + Newton 方法联合迭代（backward_euler_step_newton_method）
目标：求解隐式方程

$$x_n = x_0 + hf(x_n)$$

这是 Backward Euler 步的形式，我们使用 Newton 方法求解该不动点方程。

- **定义残差函数：**

$$r(x_n) := x_0 + hf(x_n) - x_n$$

- **迭代步骤：**

$$\begin{aligned} & x_n \leftarrow x_0 \\ & \text{while } \|r(x_n)\| > \epsilon \text{ do:} \\ & \quad \text{计算残差 } r = x_0 + hf(x_n) - x_n \\ & \quad \text{计算雅可比 } \partial r = \frac{\partial r}{\partial x_n} = h\frac{\partial f}{\partial x}(x_n) - I \\ & \quad \text{牛顿更新：} \quad x_n \leftarrow x_n - (\partial r)^{-1} r \end{aligned}$$

这个过程将 Backward Euler 构造的非线性方程，通过 Newton 方法迭代求解，适用于刚性系统或更稳定收敛的时间积分方法。

Backward Euler 与 Newton 方法的比较

- Backward Euler 方法虽然具有 **人工阻尼（artificial damping）** 效果，但是收敛很慢（—Re(eig(R))—可以看出来），较稳定。
- 固定点迭代方法（如 $x_{k+1} = f(x_k)$）通常 **不能直接找到极小值**，其收敛速度为 **线性收敛（linear convergence）**，如图所示呈指数衰减但较慢。
- 相比之下，Newton 方法可在 **约 3 步迭代内** 快速接近最优解，具有 **二次收敛率（quadratic convergence）**，即误差下降速度为：

$$\|x_{k+1} - x^*\| \leq C\|x_k - x^*\|^2.$$

Newton 方法的总结与优势

- Newton 方法配合 backward Euler 等技术，**收敛非常快**。
- 拥有 **二次收敛（Quadratic Convergence）**，即误差下降速度极快。
- 可以达到 **机器精度（machine precision）**，误差极小。
- **主要代价**在于对 Jacobian 的因式分解 / 求逆操作，复杂度约为：

$$\mathcal{O}(n^3)$$

- 若能 **利用问题结构（problem structure）**，可进一步降低复杂度，后续会展开说明。

隐式欧拉在非线性动力系统中的求根演示（单摆） 我们考虑经典非线性系统——单摆，其动力学形式为：

$$\dot{\theta} = \omega, \quad \dot{\omega} = -\frac{g}{l}\sin(\theta) \Rightarrow \dot{x} = f(x), \quad x = \begin{bmatrix}\theta \\ \omega\end{bmatrix}$$

使用 Backward Euler 离散化后，每一步需要求解如下形式的不动点问题：

$$x_{n+1} = x_n + hf(x_{n+1}) \quad \Rightarrow \quad \text{Root-Finding:} \quad r(x) = x_n + hf(x) - x = 0$$

方法一：固定点迭代 设 $g(x) = x_n + hf(x)$，则为不动点迭代：

$$x_{k+1} = g(x_k)$$

收敛条件： 若 f 是 Lipschitz 连续，且 h 足够小使得 $|g'(x)| = |hf'(x)| < 1$，则根据 Banach 不动点定理，该迭代可收敛。

关键代码：

```
def backward_euler_step_fixed_point(fun, x0, h):
    xn = x0.copy()
    while np.linalg.norm(x0 + h * fun(xn) - xn) > tol:
        xn = x0 + h * fun(xn)  # Fixed-point iteration
    return xn
```

方法二：Newton 法 将方程 $r(x) := x_n + hf(x) - x = 0$ 应用 Newton 方法：

$$x_{k+1} = x_k - \left(\frac{\partial r}{\partial x}(x_k)\right)^{-1} r(x_k)$$

其中：

$$\frac{\partial r}{\partial x} = h\frac{\partial f}{\partial x} - I$$

我们使用 `autograd` 自动求导，构造残差雅可比并执行线性求解。

关键代码：

```python
def backward_euler_step_newton(fun, x0, h):
    def residual(x): return x0 + h * fun(x) - x
    J = jacobian(residual)  # via autograd
    x = x0.copy()
    while np.linalg.norm(residual(x)) > tol:
        x -= np.linalg.solve(J(x), residual(x))  # Newton step
    return x
```

实验比较与结论： 我们对上述两种方法在单摆动力系统中进行比较。结果显示：

- **固定点法**：收敛缓慢，对步长敏感，迭代次数较多；
- **Newton 法**：收敛快，仅需少数迭代步，适用于刚性系统；
- **误差曲线：** Newton 法呈现典型的二次收敛，误差指数下降；
- **可视化：** 使用 PendulumVisualizer 工具包对系统动态进行了动画演示。

如图所示（略），两种方法在同一初值和时间步长下收敛路径基本一致，但效率差异明显。

（注：完整代码见 optimization/pendulum_forward_backward_newton.py ）

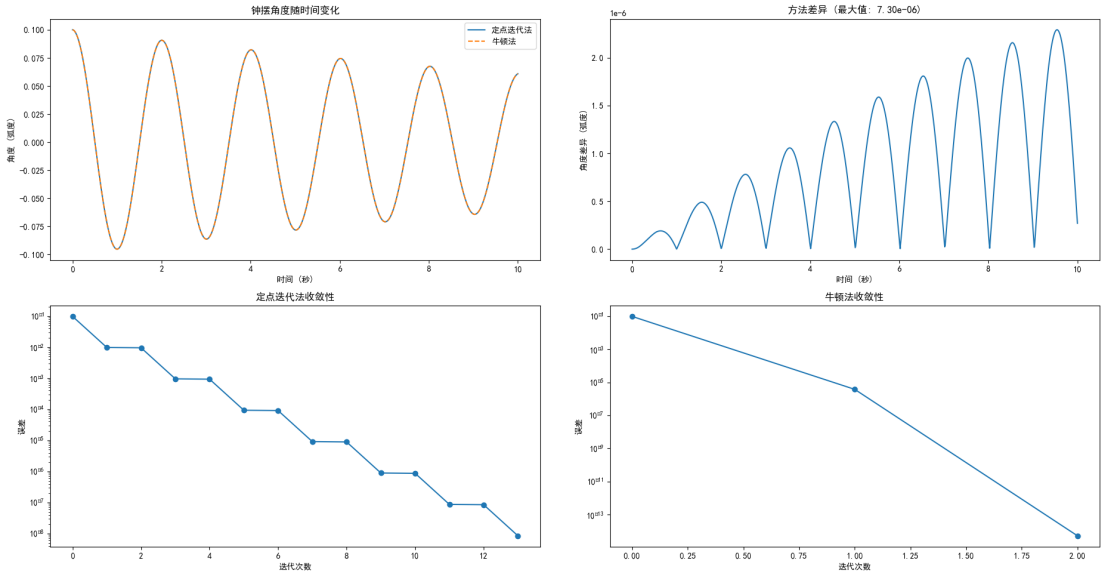


图 10: 单摆系统中两种求根方法的比较：Fixed-Point 与 Newton 法

如图 ?? 所示，我们比较了 fixed-point 与 Newton 方法在隐式欧拉时间积分中的表现：

- **左上角：** 显示两种方法所模拟出的摆角随时间的演化曲线。两条曲线几乎重合，说明在相同步长和初始条件下，两者均可得到稳定且精确的系统轨迹。
- **右上角：** 显示两种方法得到的角度差异（误差），其最大值约为 7.3×10^{-6}，属于数值误差范围，说明精度相当。
- **左下角：** 为固定点方法在某一步迭代中的误差衰减过程（以 0.1 步长模拟）。可以看出误差呈线性下降，符合固定点方法的线性收敛理论。
- **右下角：** 为牛顿方法的误差下降过程，仅用了 3 步就达到机器精度。呈现二次收敛特性，即误差以平方速度下降。

结论： 若系统非线性较强，或对精度/收敛速度有要求，建议使用 Newton 法；而在结构简单、实时性要求高时，fixed-point 方法仍具备价值。

3.2.2 牛顿方法在最小化问题中的应用

考虑优化问题

$$\min_x f(x), \quad f(x)\colon \mathbb{R}^n \to \mathbb{R}$$

- 若 f是光滑的，则要满足 $\frac{\partial f}{\partial x}\big|_{x^*} = 0$在局部最小值成立（KKT）。
- 可将该问题转化为 root-finding 问题：$\nabla f(x) = 0$

应用 Newton 方法：

1. 计算更新量

$$\begin{aligned} &\nabla f(x + \Delta x) \approx \nabla f(x) + \nabla^2 f(x)\Delta x = 0 \\ \Rightarrow \quad &\Delta x = -[\nabla^2 f(x)]^{-1}\nabla f(x) \end{aligned}$$

2. 更新

$$x \leftarrow x + \Delta x$$

解释：牛顿方法 = 二次近似最小化 牛顿方法可以理解为在当前点 x 对 $f(x)$ 作局部二次近似：

$$f(x + \Delta x) \approx f(x) + \nabla f(x)^\top \Delta x + \frac{1}{2}\Delta x^\top \nabla^2 f(x)\Delta x$$

然后最小化该二次函数，得到的最优 Δx 就是 Newton 步。

例子：

$$f(x) = x^4 + x^3 - x^2 - x, \quad \nabla f(x) = 4x^3 + 3x^2 - 2x - 1, \quad \nabla^2 f(x) = 12x^2 + 6x - 2$$

例子：一维多项式函数的牛顿法最小化

我们考虑如下的实值一元多项式函数

$$f(x) = x^4 + x^3 - x^2 - x$$

其一阶导数（梯度）为

$$\nabla f(x) = 4x^3 + 3x^2 - 2x - 1$$

二阶导数（Hessian，在一维情况下即为标量）为

$$\nabla^2 f(x) = 12x^2 + 6x - 2$$

此函数为四次多项式，具有多个临界点。令 $\nabla f(x) = 0$，即对应牛顿法中所需求解的 root-finding 问题，其解可能为极小值点、极大值点或鞍点。Hessian 的符号决定了该点的曲率性质，若 $\nabla^2 f(x^*) > 0$，则该点为局部极小值。

图像分析表明该函数在 $x \approx -1.5, 0, 1.0$ 附近存在多个极值点，但牛顿法对初始点敏感，不同初值可能收敛至不同点。

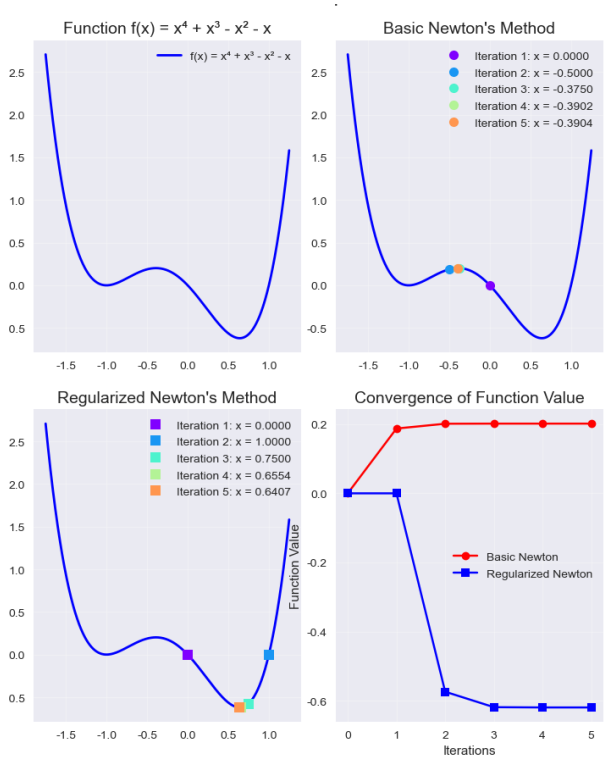
核心代码实现（Python） 我们对该函数进行数值实现，仅保留牛顿更新部分，具体如下：

```
def f(x):
  return x**4 + x**3 - x**2 - x
def grad_f(x):
  return 4*x**3 + 3*x**2 - 2*x - 1
def hessian_f(x):
  return 12*x**2 + 6*x - 2

# basic newton method
def newton_step(x0):
    return x0 - grad_f(x0) / hessian_f(x0)

# normalized newton method, to avoid Hessian matrix is negative
def regularized_newton_step(x0):
    beta = 1.0
    H = hessian_f(x0)
    while H <= 0:
        H += beta
        beta *= 2
    return x0 - grad_f(x0) / H
```

图像分析与比较 我们从初始点 $x_0 = 0$ 开始，分别使用基本牛顿法与正则化牛顿法进行 5 次迭代。可视化结果如下图所示：



并且我们绘制出此处的Hessian：

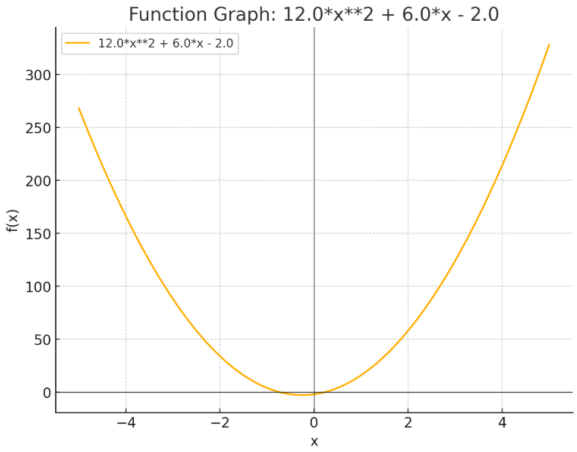


图 11: 单元多次函数的Hessian可视化，可以看到有负的情况，如果是矩阵则添加I的倍数，此处因为是单变量，添加1的倍数即可。

数值结果：

$$\text{基本牛顿法：} \quad x^* \approx -0.3904, \quad f(x^*) \approx 0.2017$$

$$\text{正则化牛顿法：} \quad x^* \approx 0.6404, \quad f(x^*) \approx -0.6197$$

结论：

- 函数 $f(x)$ 存在多个临界点，且 Hessian 可为负，说明基本牛顿法存在收敛到非极小值点的风险；
- 正则化牛顿法通过调整 Hessian，避免非正定性所带来的不稳定收敛问题；

- 在该例中，正则化方法显著提高了迭代稳定性与最终函数值质量。

结论： Newton 方法是 **局部 root-finding 方法**，对 $\nabla f(x) = 0$ 求解，其收敛结果取决于初始点：

- 可能收敛到极小值、极大值或鞍点。
- 要求初始点足够接近目标点。

3.3 最小值存在的充分条件

- 一阶必要条件：$\nabla f(x^*) = 0$ 是最小值的必要条件，但**不是充分条件!**
- 看一个标量情况：

  $$\Delta x = -(\nabla^2 f)^{-1} \cdot \nabla f$$

  梯度方向决定前进方向，Hessian 决定“步长”或“学习率”。
- 若 Hessian 为正：

  $$\nabla^2 f > 0 \Rightarrow \text{descent（最小化）}$$

  若 Hessian 为负：

  $$\nabla^2 f < 0 \Rightarrow \text{ascent（最大化）}$$
- 更进一步，在 $\mathbb{R}^n$ 中，若

  $$\nabla^2 f \succ 0 \quad \text{（严格正定）} \Rightarrow \nabla^2 f \in \mathcal{S}_{++}$$

  [1] 即所有特征值均为正，是局部极小的**充分条件**。因此，**局部极小的二阶充分条件**是：

  在极小值点，Hessian 的所有特征值为正，即

  $$\Rightarrow \text{方向为下降方向（descent）}$$

3.4 正则化（Regularization）与强凸性

- 若 $\nabla^2 f > 0$ 在整体上成立 $\Leftrightarrow$ $f(x)$ 是**强凸函数（strongly convex）**，则可用牛顿法全局求解。
- 但这对非凸或非线性问题通常不成立，因此不能总是依赖牛顿法。
- **正则化的动机：** 为了让 Hessian 始终正定，从而确保下降方向，可以进行如下处理：
  - 设 $H = \nabla^2 f$，若 $H \not\succ 0$，则执行：

    $$\text{while } H \not\succ 0: \quad H \leftarrow H + \beta I$$

    其中 $\beta > 0$，I 为单位矩阵，β 可逐步增大。

[1]关于矩阵正定的定义与性质请参考《矩阵分析》相关书籍与课程，此处不赘述。

– 更新步长：

$$\Delta x = -H^{-1}\nabla f, \quad x \leftarrow x + \Delta x$$

– 这种方法称为 **Tikhonov 正则化（Tikhonov regularization）**，在最小二乘（least squares）等问题中也常见，体现为给权重加上二次惩罚项。

- **damped Newton** 方法，通过调节步长或对 Hessian 添加扰动来控制更新方向，使其总是一个下降步（descent step）：

  – 它**保证**是下降方向，同时还能让步长变得更小（make step size smaller）。

  – 给 Hessian 加上单位阵（positive identity）本质上等价于在优化目标中添加一个二次惩罚项（quadratic penalty）：

  例如 $\beta\Delta x^\top \Delta x$ 被加到目标函数上 $\Rightarrow$ 惩罚步长过大，避免 overshoot

为什么给 Hessian 加上 βI 等价于加惩罚项？ 在牛顿方法中，我们在 x 附近用二阶泰勒展开近似目标函数：

$$f(x+\Delta x) \approx f(x) + \nabla f(x)^\top \Delta x + \frac{1}{2}\Delta x^\top \nabla^2 f(x)\Delta x$$

此时求解的是使该近似最小的 Δx。
为了增强数值稳定性，或者避免 Hessian 非正定带来的问题，我们会对 Hessian 做如下正则化：

$$\tilde{H} = \nabla^2 f(x) + \beta I$$

此时对应的优化问题变为最小化：

$$f(x) + \nabla f(x)^\top \Delta x + \frac{1}{2}\Delta x^\top (\nabla^2 f(x) + \beta I)\Delta x$$

$$= f(x) + \nabla f(x)^\top \Delta x + \frac{1}{2}\Delta x^\top \nabla^2 f(x)\Delta x + \frac{1}{2}\beta\|\Delta x\|^2$$

因此，**给 Hessian 加 βI 等价于在原函数上添加一项二次惩罚项：**

$$\frac{1}{2}\beta\|\Delta x\|^2$$

这样做的意义是：

* 控制 Δx 的幅度，防止步长过大（overshooting）
* 保证 Hessian 正定，从而确保下降方向
* 提高病态 Hessian 的数值稳定性

```python
# aim function
def f(x): return x**4 + x**3 - x**2 - x
def grad_f(x): return 4*x**3 + 3*x**2 - 2*x - 1
def hessian_f(x): return 12*x**2 + 6*x - 2

# Damped Newton
def damped_newton_step(x0, beta=5.0):
    H = hessian_f(x0)
```

```
    # normalized Hessian`
    H_damped = H + beta
    return x0 - grad_f(x0) / H_damped

# compare to different newton path
x0 = 0.0
path_beta_1 = [x0]
path_beta_5 = [x0]

for _ in range(5):
    x0 = damped_newton_step(path_beta_1[-1], beta=1.0)
    path_beta_1.append(x0)

x0 = 0.0
for _ in range(5):
    x0 = damped_newton_step(path_beta_5[-1], beta=5.0)
    path_beta_5.append(x0)
```

- 总结：这种方法的效果是 **抑制步长过大**，尤其适合 Hessian 非正定或发散风险高的情形。

3.5 线性搜索（Line Search） Backtracking方法

为了解决牛顿法中可能的 overshooting（步长过大）问题，引入线性搜索（Line Search）策略。

基本思想： 沿当前方向 Δx，不直接采用全量更新，而是寻找一个缩放因子 $\alpha \in (0, 1]$，使得目标函数有足够的下降。

下降条件： 基于一阶泰勒展开，有如下下降近似：

$$f(x + \Delta x) \approx f(x) + \nabla f(x)^\top \Delta x$$

我们要求实际下降量不小于这个估计值的一定比例，即：

$$f(x + \alpha \Delta x) \leq f(x) + \beta \nabla f(x)^\top \Delta x$$

其中 $\beta \in (0, 1)$ 是容差系数（通常取 $\beta \approx 10^{-4}$），该条件称为 Armijo 条件。

算法伪代码： 下面展示了Armijo算法伪代码回溯线搜索是一种用于解决无约束优化问题的步长选择技术。该算法的主要步骤如下：

1. **初始化：**设置初始步长 $\alpha = 1$，以及减小系数 $\tau \in (0, 1)$（通常为0.5）和Armijo条件参数 $\beta \in (0, 1)$（通常为0.1）。
2. **迭代过程：**重复检查Armijo条件：$f(x + \alpha \Delta x) > f(x) + \beta \alpha \nabla f(x)^\top \Delta x$。如果条件成立，表示当前步长过大，需要减小步长：$\alpha \leftarrow \tau \cdot \alpha$。
3. **终止条件：**当找到合适的步长（即不满足上述条件）时，返回新的解 $x_{\text{new}} = x + \alpha \Delta x$。

在优化过程中，Δx 通常是负梯度方向 $-\nabla f(x)$ 或其他搜索方向。Armijo条件保证了函数值有足够的下降，从而确保算法的收敛性。参数 β 控制了对下降幅度的要求，而 τ 控制了步长减小的速度。

Backtracking Line Search Algorithm

Input : x, Δx, $f(x)$, $\nabla f(x)$
Output: $x_{\text{new}} = x + \alpha \Delta x$
$\alpha \leftarrow 1$; $\tau \in (0,1)$; $\beta \in (0,1)$;
while $f(x + \alpha \Delta x) > f(x) + \beta \alpha \nabla f(x)^\top \Delta x$ **do**
  $\alpha \leftarrow \tau \cdot \alpha$;
return $x + \alpha \Delta x$

简写记号： 也可写成如下形式，强调目标函数下降满足一阶下降近似的比例容差：

$$f(x + \alpha \Delta x) \leq f(x) + \beta \nabla f(x)^\top \Delta x \quad \text{（下降容忍）}$$

或者在笔记中常见的简写方式：

$$f(x + \Delta x) \leq f(x) + \beta \nabla f(x)^\top \Delta x$$

该条件是基于一阶导数的估计下降，也称作：
$f(x + x) \leq f(x) + \beta \nabla f(x)^T \Delta x$

4 约束优化问题

因为有约束的优化问题是一个复杂的问题，我们这里独立作为一章进行讲解。在**??**中，我们讨论了无约束的优化问题，然而现实中大多数是有约束的优化问题，比如说机械臂的速度不能过快、奇异位置限制等等。所以这一章，我们主要讲述有约束的优化问题。

4.1 等式约束问题

考虑以下最优化问题，受等式约束的限制：

$$\min_x f(x) : f(x) : \mathbb{R}^n \to \mathbb{R}$$

$$\text{s.t.} \quad c(x) = 0 : \mathbb{R}^n \to \mathbb{R}^m$$

4.1.1 一阶必要条件

在函数达到最小值的点x处的一阶必要条件：

- 需要 $\nabla f(x) = 0$ 在无约束的/自由方向上。
- 需要 $c(x) = 0$。

如果$c(x)$是直线，那么它肯定与$c(x)$的切线相切，$f(x)$ 在该点达到了最小值（图**??**）。与此相同的是，如果$c(x)$是N维的，则最小化函数的切线意味着它们的切向量（导数）必须共线。因此，$\nabla f(x)$的任何非零分量都必须垂直于约束面（constraint surface）/流形（mainfold）。

$$\nabla f + \lambda \nabla c = 0 \quad \text{对于某个 } \lambda \in \mathbb{R}$$

其中λ是拉格朗日乘子或“对偶变量”。通常，考虑到多维情况，我们有：

$$\frac{\partial f}{\partial x} + \lambda^T \frac{\partial c}{\partial x} = 0 \quad \lambda \in \mathbb{R}^m$$

基于该梯度条件，我们定义拉格朗日函数：

$$L(x, \lambda) = f(x) + \lambda^T c(x)$$

该函数受到以下约束：

$$\text{s.t.} \quad \begin{cases} \nabla_x L(x, \lambda) = \nabla f + \lambda^T \dfrac{\partial c}{\partial x} = 0 \\ \nabla_x L(x, \lambda) = c(x) = 0 \end{cases}$$

这些条件共同表示了KKT条件。我们现在可以使用牛顿法联合求解x和λ作为根求解问题：

$$\nabla_x L(x + \Delta x, \lambda + \Delta\lambda) \approx \nabla_x L(x, \lambda) + \frac{\partial^2 L}{\partial x^2}\Delta x + \frac{\partial^2 L}{\partial x \partial \lambda}\Delta\lambda = 0$$

$$\nabla_x L(x + \Delta x, \lambda + \Delta\lambda) \approx c(x) + \frac{\partial c}{\partial x}\Delta x = 0 \Rightarrow \frac{\partial c}{\partial x}\Delta x = -c(x)$$

这可以写成以下矩阵系统：

$$\begin{bmatrix} \frac{\partial^2 L}{\partial x^2} & \frac{\partial^2 L}{\partial x \partial \lambda} \\ -\nabla_x L(x,\lambda) & -c(x) \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \end{bmatrix} = 0$$

其中第一矩阵是拉格朗日函数的Hessian矩阵，描述了KKT系统。

梯度计算与可视化 为了进行梯度可视化，我们首先计算目标函数和约束条件的梯度。在优化问题中，目标函数的梯度表示为：

$$\nabla f(x_1, x_2) = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2} \right] = [2(x_1 - 2), 2(x_2 - 1)]$$

约束条件的梯度为：

$$\nabla c(x_1, x_2) = \left[\frac{\partial c}{\partial x_1}, \frac{\partial c}{\partial x_2} \right] = [1, 1]$$

接下来，我们用Python代码计算这些梯度，并通过箭头将其可视化。

```
# 计算目标函数的梯度
gradient_x1 = 2 * (opt_x1 - 2)  # df/dx1'
gradient_x2 = 2 * (opt_x2 - 1)  # df/dx2
constraint_grad_x1 = 1  # dc/dx1
constraint_grad_x2 = 1  # dc/dx2
```

图像与说明 在下方的图像中，我们可以看到目标函数、约束条件以及梯度向量的可视化。图像由三个子图组成：
1. 目标函数图：左侧的图展示了目标函数$f(x_1, x_2) = (x_1 - 2)^2 + (x_2 - 1)^2$的三维图形（自行运行代码，可以用鼠标拖动方向）。在目标函数的表面上，最优点被标记为红色圆点，表示最小化的解。
2. 等高线与约束条件图：中间的图展示了目标函数的等高线以及约束条件$x_1 + x_2 = 3$的线。最优点同样被标记在图中。
3. 梯度与约束图：右侧的图展示了目标函数的梯度（蓝色箭头）和约束条件的梯度（红色箭头）。最优点位于两条梯度相互平衡的位置，符合KKT条件。

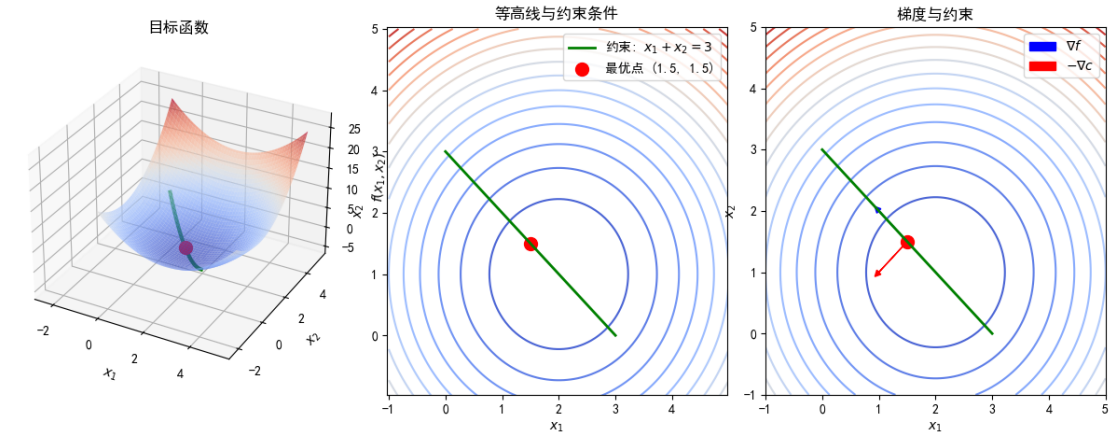


图 12: 目标函数、等高线与约束条件以及梯度可视化图。

4.1.2 Gauss-Newton 方法的应用

Gauss-Newton原本是在牛顿法基础上修改，仅用于解决非线性最小二乘法问题，其不用计算二阶导数矩阵。

Gauss-Newton 方法在实际问题中广泛应用，特别是在优化问题和约束最优化中。相比于标准的牛顿法，Gauss-Newton 方法通过忽略二阶导数的影响（即忽略“约束曲率”）来简化计算，从而大大减少了每次迭代的计算量。尽管如此，Gauss-Newton 方法通常会导致稍慢的收敛速度，但每次迭代所需的计算量远小于标准牛顿法。

$$\frac{\partial^2 L}{\partial x^2} = \nabla^2 f + \frac{\partial}{\partial x}\left[\left(\frac{\partial c}{\partial x}\right)^T \lambda\right]$$ 后面这一项是Hessian矩阵，计算复杂度太高!

应用示例： 我们考虑一个优化问题，目标函数为 $f(x)$，约束条件为 $c(x)$，Gauss-Newton 方法通过如下步骤来求解最优解：

$$\nabla_x L(x, \lambda) = \nabla f + \lambda^T \frac{\partial c}{\partial x} = 0$$

其中 $L(x, \lambda)$ 是拉格朗日乘子法的拉格朗日函数，λ 是拉格朗日乘子。

标准牛顿法与 Gauss-Newton 方法的对比 这里用公式对比两种方法的区别

标准牛顿法

目标函数和约束的数学公式：

$$\frac{\partial^2 L}{\partial x^2} = \nabla^2 f(x) + \frac{\partial}{\partial x}\left[\left(\frac{\partial c(x)}{\partial x}\right)^T \lambda\right]$$

更新步的 KKT 条件：

$$\begin{bmatrix} H & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \end{bmatrix} = \begin{bmatrix} -\nabla f(x) - C^T \lambda \\ -c(x) \end{bmatrix}$$

Hessian 矩阵：

$$H = \nabla^2 f(x) + \frac{\partial}{\partial x}\left[\left(\frac{\partial c(x)}{\partial x}\right)^T \lambda\right]$$

更新公式：

$$\begin{bmatrix} x_{k+1} \\ \lambda_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ \lambda_k \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta \lambda \end{bmatrix}$$

Gauss-Newton 方法

目标函数和约束的数学公式：

$$\frac{\partial^2 L}{\partial x^2} = \nabla^2 f(x)$$

更新步的 KKT 条件：

$$\begin{bmatrix} H & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \end{bmatrix} = \begin{bmatrix} -\nabla f(x) - C^T \lambda \\ -c(x) \end{bmatrix}$$

Hessian 矩阵：

$$H = \nabla^2 f(x)$$

更新公式：

$$\begin{bmatrix} x_{k+1} \\ \lambda_{k+1} \end{bmatrix} = \begin{bmatrix} x_k \\ \lambda_k \end{bmatrix} + \begin{bmatrix} \Delta x \\ \Delta \lambda \end{bmatrix}$$

```python
# Newton方法实现 - 计算完整的Hessian矩阵
def newton_step(x0, lambda0):
    # 确保x0是一维数组
    x0 = np.ravel(x0)
    lambda0 = np.array([lambda0]).ravel()

    # 计算目标函数的Hessian矩阵 (包括二阶导数)
    H = hessian_f(x0)

    # 计算约束的雅可比矩阵
    C = grad_c(x0).reshape(1, -1)
```

```
    # 构建完整的KKT矩阵
    KKT_matrix = np.block([
        [H, C.T],
        [C, np.array([[0]])]
    ])

    # 构建右侧向量
    rhs = np.concatenate([
        -grad_f(x0) - C.T @ lambda0,
        -np.array([c(x0)])
    ])

    # 求解线性方程组
    delta_z = np.linalg.solve(KKT_matrix, rhs)

    delta_x = delta_z[:2]
    delta_lambda = delta_z[2]

    return x0 + delta_x, lambda0[0] + delta_lambda
```

```
# Gauss-Newton方法实现 - 忽略约束曲率（二阶导数）
def gauss_newton_step(x0, lambda0):
    # 确保x0是一维数组
    x0 = np.ravel(x0)
    lambda0 = np.array([lambda0]).ravel()

    # 计算目标函数的Hessian矩阵 (仅使用一阶项)
    H = hessian_f(x0)

    # 计算约束的雅可比矩阵
    C = grad_c(x0).reshape(1, -1)

    # Gauss-Newton简化：忽略约束的二阶导数影响
    # 直接使用目标函数的Hessian而不添加约束的二阶导数影响

    # 构建简化的KKT矩阵
    KKT_matrix = np.block([
        [H, C.T],
        [C, np.array([[0]])]
    ])

    # 构建右侧向量
    rhs = np.concatenate([
        -grad_f(x0) - C.T @ lambda0,
        -np.array([c(x0)])
    ])

    # 求解线性方程组
    delta_z = np.linalg.solve(KKT_matrix, rhs)

    delta_x = delta_z[:2]
    delta_lambda = delta_z[2]

    return x0 + delta_x, lambda0[0] + delta_lambda
```

实验结果与比较 我们设置初始猜测为 $[-1,-1]$ 和 $[-3,2]$，分别使用牛顿法和 Gauss-Newton 方法求解。实验结果表明，Gauss-Newton 方法在每次迭代中计算量较小，但收敛速度略慢于标准牛顿法。我们比较了标准牛顿法和Gauss-Newton方法在约束优化问题中的收敛速度和表现。

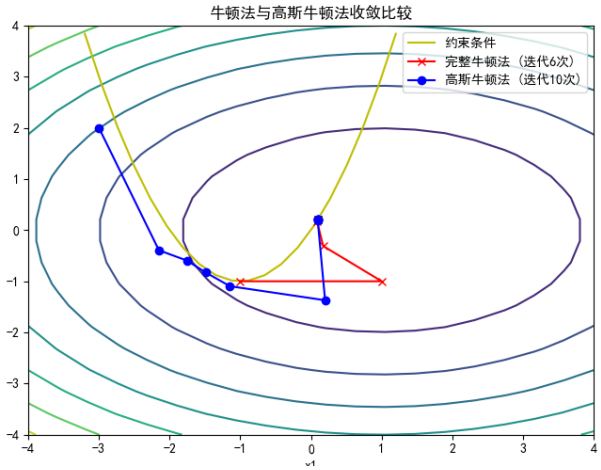


图 13: 牛顿法与高斯-牛顿法的优化路径对比（初始猜测为[-1, -1]）

从图??中可以看出，牛顿法的收敛速度较快，而Gauss-Newton方法则在迭代过程中收敛较慢，但计算量显著较低。

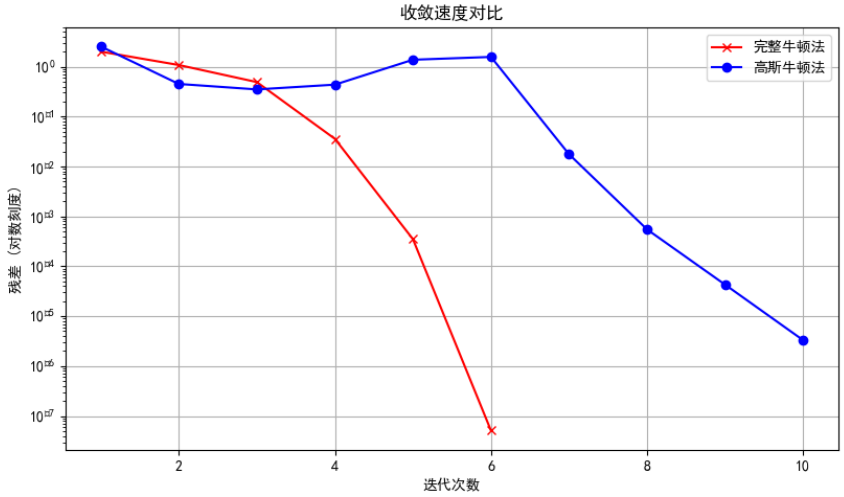


图 14: 标准牛顿法与Gauss-Newton方法的收敛速度对比

图??展示了标准牛顿法与Gauss-Newton方法在不同迭代次数下的收敛速度对比。可以看到，标准牛顿法在较少的迭代次数内达到了收敛，而Gauss-Newton方法则需要更多的迭代。

Take-Away Message:

- Gauss-Newton 方法在处理大规模优化问题时非常有效。
- 在一些情况下，可能需要对 $\frac{\partial^2 L}{\partial x^2}$ 进行正则化。
- 尽管 Gauss-Newton 方法的收敛速度较慢，但其每次迭代的计算量显著降低。
- 约束的曲率越大，由于忽略二次项而导致的偏差越大，从而性能越不好。

4.2 不等式约束

更一般地，我们必须调和不等式约束。考虑一个带有不等式约束的最小化问题：

$$\min_x f(x) \quad \text{s.t.} \quad c(x) \geq 0$$

让我们先看一下纯不等式约束的情况。通常，这些方法结合了等式约束方法，以处理在同一问题中同时包含不等式/等式约束的情况。

4.2.1 一阶必要条件

和之前一样，一阶必要条件要求：

- 在无约束/自由方向上，需要 $\nabla f(x) = 0$。
- 需要 $c(x) \geq 0$，注意这里的不等式。

从数学上讲，

$$\begin{cases} \nabla f(x) - (\frac{\partial c}{\partial x})^T \lambda = 0 & \text{“Stationarity”} \quad \text{（驻点条件）} \\ c(x) \leq 0 & \text{“Primal Feasibility”} \quad \text{（原始可行性）} \\ \lambda \geq 0 & \text{“Dual Feasibility”} \quad \text{（对偶可行性）} \\ \lambda^T c(x) = 0 & \text{“Complementarity”} \quad \text{（互补条件）} \end{cases}$$

$(\frac{\partial c}{\partial x})^T$是惩罚项，确保若$c(x)$被违反，则对目标函数的惩罚项为正。这些条件一起指定了完整的KKT条件。注意，无论 $c(x)$ 或 λ 的符号如何，它都依赖于你习惯的方式，一条原则是它总是防止目标函数的变化。

4.2.2 直观理解

这些条件的直观理解如下：

- 如果约束是活动的（我们处于约束流形上），我们有 $c(x) = 0 \Rightarrow \lambda = 0$。这与等式约束的情况相同。
- 如果约束不活跃，我们有 $c(x) > 0 \Rightarrow \lambda = 0$，这与无约束的情况相同。
- 本质上，互补性确保了 λ 或 $c(x)$ 其中之一为零，或者约束的“开/关”切换。

4.3 算法

这里实现优化方法比等式约束问题更加复杂；我们不能直接将牛顿法应用到KKT条件中。我们有许多可供选择的方法，不同方法有不同的折衷。

4.3.1 活动集方法 (Active-Set Method)

使用场合： 当你知道哪些约束是活动的/不活动时，可以使用此方法。

适用问题： 解决等式约束问题（当它是活动时）。

优点： 如果你有一个好的启发式方法，它可以非常快速。

缺点： 如果你不知道哪些约束是活动的，它可能会非常复杂（组合性问题）。

活动约束的判定 判断一个约束是否是活动的，可以使用以下定义：

定义： 如果约束 $c_i(x)$ 在某点 x^* 满足 $c_i(x^*) = 0$，则称该约束在点 x^* 是**活动的**。对于不等式约束 $c_i(x) \geq 0$，这意味着约束在边界上被严格满足。

定理（活动集判定）： 在一个约束优化问题的最优解 x^* 处：

$$\min_x f(x) \quad \text{满足} \quad c_i(x) \geq 0,\, i = 1, \ldots, m,$$

活动约束的集合 $\mathcal{A}$ 定义为：

$$\mathcal{A} = \{i \mid c_i(x^*) = 0\}.$$

对于所有 $i \notin \mathcal{A}$，有 $c_i(x^*) > 0$，并且对应的拉格朗日乘子 $\lambda_i = 0$。

例子： 考虑以下优化问题：

$$\min_{x_1, x_2} f(x_1, x_2) = (x_1 - 1)^2 + (x_2 - 2)^2$$

$$\text{满足} \quad c_1(x) = x_1 + x_2 - 3 \geq 0, \quad c_2(x) = x_1 \geq 0.$$

在候选解 $x^* = (1, 2)$ 处：

- $c_1(x^*) = 1 + 2 - 3 = 0$：c_1 是活动的。
- $c_2(x^*) = 1 \geq 0$：c_2 是非活动的。

因此，活动集为 $\mathcal{A} = \{1\}$。

4.3.2 障碍法 / 内点法 (Barrier / Interior-Point Method)

$$\min_x f(x) \quad \text{s.t.} \quad Cx \leq 0 \quad \Rightarrow \quad \min_x \left(f(x) - \frac{1}{\rho} \sum_{i=1}^{m} \log(-c_i(x)) \right)$$

因为添加了一个在边界处blow-up的项，所以不会越界。我们可以通过绘制$f(x) = -log(-x)$看到，越到边界0，函数值越大。我们可以通过调整ρ来控制惩罚的强度，ρ越大，半径越大，约束越强，在实际中，常常从大到小来调整ρ，直到收敛为止

方法描述： 我们用一个“障碍函数”替换不等式约束，目标函数在约束边界处爆炸，见图14。

适用问题： 这是中等规模凸问题的标准方法。对MPC问题非常常用

缺点： 对于非凸问题，需要大量的技巧和技巧来使其工作。

4.4 惩罚方法 / 外点法 (Penalty / Exterior-Point Method)

外点法（Penalty Method）通过用惩罚项来替代约束条件，得到一个无约束的优化问题。具体来说，它通过惩罚违反约束的行为来进行优化。

4.4.1 外部惩罚方法

首先我们替换约束条件，使用惩罚项来代替，得到如下的优化问题：

$$\min_{\mathbf{x}} f(\mathbf{x}) \quad \text{s.t.} \quad c(\mathbf{x}) \leq 0$$

转化为

$$\min_{\mathbf{x}} f(\mathbf{x}) + \rho \left[\max(0, c(\mathbf{x}))\right]^2$$

其中，ρ是惩罚系数，随着迭代的进行，ρ通常逐渐增大，以增强对违反约束的惩罚。

惩罚方法易于实现，但存在一些问题，例如数值不稳定（由于ρ增大），并且无法达到高精度。特别是在ρ较大时，算法可能会变得非常不稳定。

$$\min_{\mathbf{x}} f(\mathbf{x}) + \rho \left[\max(0, c(\mathbf{x}))\right]^2$$

- **优点：**
  - 实现简单。
  - 适用于一般的约束优化问题。
- **缺点：**
  - 数值不稳定。
  - 无法达到高精度。
  - 需要增加ρ来处理约束条件，这会导致问题的复杂度增加。
  - 因为惩罚只是pull-back，所以会导致越界。

4.4.2 增广拉格朗日方法 (Augmented Lagrangian Method)

我们向惩罚方法中添加拉格朗日乘子估计，以解决惩罚方法所遇到的问题。具体而言，增广拉格朗日函数可以通过如下表达式表示：

$$\min_{x} f(x) + \tilde{\lambda}^T c(x) + \frac{\rho}{2} \left[\min(0, c(x))\right]^2$$

其中，$\tilde{\lambda}^T c(x)$称之为拉格朗日乘子估计值（lagerangian multipliers estimation）用于吸收约束，最终这个估计值$\tilde{\lambda}$收敛于λ。$\frac{\rho}{2} \left[\min(0, c(x))\right]^2$表示二次惩罚项，$\rho$是惩罚参数。这两项结合在一起，形成了增广拉格朗日函数$L_\rho(x, \lambda)$。首先，我们在固定的 λ 下对 x 进行最小化，然后每次迭代时通过“卸载”惩罚项来更新 λ。

为了进一步更新拉格朗日乘子，我们知道如果是最小值，则其偏导一定为零，则有：

$$\begin{aligned} & \frac{\partial f}{\partial x} - \tilde{\lambda} \frac{\partial c}{\partial x} + \rho c(x)^T \frac{\partial c}{\partial x} = 0 \\ \Leftrightarrow & \frac{\partial f}{\partial x} - \left[\tilde{\lambda} - \rho c(x)\right]^T \frac{\partial c}{\partial x} = 0 \\ & \Rightarrow \quad \tilde{\lambda} \leftarrow \tilde{\lambda} - \rho c(x) \end{aligned}$$

这说明，$\tilde{\lambda}$ 将会更新为 $\tilde{\lambda} - \rho c(x)$，对于那些活动约束，越来越多的约束将会吸入到lagrangian multipliers中。

在算法上，我们可以将增广拉格朗日方法表示为如下：

Algorithm 1: 增广拉格朗日方法

Input: 初始值 x_0, λ_0, ρ_0
Output: 最优解 x，拉格朗日乘子 λ
while 未收敛 **do**
  最小化 $L_\rho(x, \tilde{\lambda})$ 以更新 x ;
  更新拉格朗日乘子：$\tilde{\lambda} \leftarrow \tilde{\lambda} - \max(0, \tilde{\lambda} - \rho c(x))$;
    /* 更新拉格朗日乘子，确保非负性。 */
  增大惩罚参数：$\rho \leftarrow \alpha\rho$;
    /* 增加惩罚参数，一般 $\alpha \approx 10$。 */
end

该方法的优点在于：

- 修复了惩罚方法中可能出现的病态问题。
- 收敛速度较快（超线性），但精度适中。
- 在非凸问题中同样有效。

不等式约束的优化方法，牛顿增广拉格朗日法例
目标函数：

$$f(x) = \frac{1}{2}\left(x - \begin{bmatrix}1\\0\end{bmatrix}\right)^T Q \left(x - \begin{bmatrix}1\\0\end{bmatrix}\right)$$

其中，Q 是对角矩阵。
目标函数的梯度：

$$\nabla f(x) = Q\left(x - \begin{bmatrix}1\\0\end{bmatrix}\right)$$

目标函数的 Hessian 矩阵：

$$\nabla^2 f(x) = Q$$

约束条件：

$$c(x) = Ax - b$$

其中，$A = \begin{bmatrix}1\\-1\end{bmatrix}$ 和 $b = -1$。
增广拉格朗日函数：

$$\mathcal{L}_\rho(x, \lambda) = f(x) + \lambda \cdot c(x) + \frac{\rho}{2} \cdot (c(x))^2$$

其中，λ 是拉格朗日乘子，ρ 是惩罚参数。
算法流程： 牛顿法求解增广拉格朗日函数的最小值点

Algorithm 2: 牛顿法求解增广拉格朗日函数的最小值点

Input: 初始猜测 x_0, 惩罚参数 ρ, 拉格朗日乘子 λ, 容忍度 ϵ
Output: 最优解 x
初始化 $x \leftarrow x_0$;
$p \leftarrow \max(0, c(x))$;
while 满足收敛条件且迭代次数 $<$ 最大迭代次数 **do**
  计算梯度 $g = \nabla f(x) + (\lambda + \rho p) \cdot \nabla c(x)$;
  计算Hessian矩阵 $H = \nabla^2 f(x) + \rho \cdot (\nabla c(x))^T \cdot \nabla c(x)$;
  解线性方程组 $H\Delta x = -g$ 得到 Δx;
  更新 $x \leftarrow x + \Delta x$;
  更新 $p \leftarrow \max(0, c(x))$;
  更新拉格朗日乘子 $\lambda \leftarrow \lambda + \rho \cdot c(x)$;
return 最优解 x

代码如下

```python
# 设置对角矩阵 Q
Q = np.diag([0.5, 1.0])

# 目标函数
def f(x):
    """计算目标函数值"""
    diff = x - np.array([1.0, 0.0])
    return 0.5 * diff.T @ Q @ diff

# 目标函数的梯度
def grad_f(x):
    """计算目标函数梯度"""
    return Q @ (x - np.array([1.0, 0.0]))

# 目标函数的Hessian矩阵
def hessian_f(x):
    """计算目标函数的Hessian矩阵"""
    return Q

# 约束条件
A = np.array([1.0, -1.0])
b = -1.0

def c(x):
    """计算约束条件值"""
    return np.dot(A, x) - b

def grad_c(x):
    """计算约束条件梯度"""
    return A

# 增广拉格朗日函数
def La(x, lam, rho):
    """计算增广拉格朗日函数值"""
    p = max(0, c(x))
    return f(x) + lam * p + (rho/2) * (p**2)

# 牛顿法求解
def newton_solve(x0, lam, rho, tol=1e-8):
    """使用牛顿法求解增广拉格朗日函数的最小值点"""
    x = x0.copy()
    p = max(0, c(x))

    # 初始化约束梯度矩阵
    C = np.zeros((1, 2))
    if c(x) >= 0:
```

```
        C = grad_c(x).reshape(1, 2)

    # 计算目标函数梯度
    g = grad_f(x) + (lam + rho * p) * C.T.flatten()

    # 牛顿法迭代
    iter_count = 0
    max_iter = 100
    while np.linalg.norm(g) >= tol and iter_count < max_iter:
        # 计算Hessian矩阵
        H = hessian_f(x) + rho * C.T @ C

        # 计算牛顿方向
        delta_x = np.linalg.solve(H, -g)

        # 更新x
        x = x + delta_x

        # 更新约束相关量
        p = max(0, c(x))
        C = np.zeros((1, 2))
        if c(x) >= 0:
            C = grad_c(x).reshape(1, 2)

        # 更新梯度
        g = grad_f(x) + (lam + rho * p) * C.T.flatten()
        iter_count += 1

    return x
```

4.4.3 二次规划示例 (Quadratic Program Example)

考虑以下问题：

$$\min_x \frac{1}{2}x^T Q x + q^T x \quad \text{s.t.} \quad Ax \leq b, \quad Cx = d$$

这是一个二次规划问题，包含二次项和线性约束。这类问题在控制领域中非常常见并且有广泛的应用，通常可以快速求解。

机器人学中的二次规划示例 考虑一个机器人末端执行器的运动规划问题，我们希望最小化其关节速度的平方和，同时满足末端执行器的速度约束。问题可以表述为以下二次规划问题：

$$\min_{\dot{q}} \frac{1}{2}\dot{q}^T W \dot{q}$$

$$\text{s.t.} \quad J\dot{q} = v, \quad \dot{q}_{\min} \leq \dot{q} \leq \dot{q}_{\max}$$

其中：

- $\dot{q}$ 是关节速度向量。
- W 是权重矩阵，通常为对角矩阵，用于平衡不同关节的速度优先级。
- J 是机器人雅可比矩阵，描述了关节速度与末端执行器速度之间的关系。
- v 是末端执行器的期望速度。
- $\dot{q}_{\min}$ 和 $\dot{q}_{\max}$ 是关节速度的上下限。

求解方法 该问题可以通过二次规划求解器来解决。以下是一个简单的 Python 实现：

```python
import numpy as np
from scipy.optimize import minimize

# 定义问题参数
W = np.diag([1.0, 1.0])  # 权重矩阵
J = np.array([[1.0, 0.5], [0.5, 1.0]])  # 雅可比矩阵
v = np.array([0.5, 0.5])  # 期望末端速度
q_min = np.array([-1.0, -1.0])  # 关节速度下限
q_max = np.array([1.0, 1.0])  # 关节速度上限

# 定义目标函数
def objective(q_dot):
    return 0.5 * q_dot.T @ W @ q_dot

# 定义约束
def equality_constraint(q_dot):
    return J @ q_dot - v

constraints = {'type': 'eq', 'fun': equality_constraint}
bounds = [(q_min[i], q_max[i]) for i in range(len(q_min))]

# 求解二次规划问题
result = minimize(
    objective,
    x0=np.zeros(len(q_min)),
    bounds=bounds,
    constraints=constraints)

# 输出结果
if result.success:
    print("Optimal joint velocities:", result.x)
else:
    print("Optimization failed.")
```

结果与分析 通过上述代码，我们可以得到满足末端执行器速度约束的最优关节速度，同时保证关节速度的平方和最小。这种方法在机器人运动规划中非常常用，尤其是在实时控制中。

4.5 正规化和对偶

考虑以下问题：

$$\min_x f(x) \quad \text{s.t.} \quad c(x) = 0$$

我们可以将其表示为：

$$\min_x f(x) + P_\infty(c(x))$$

其中，$P_\infty(x)$ 定义为：

$$P_\infty(x) = \begin{cases} 0, & x = 0 \quad (\text{满足约束}) \\ +\infty, & x \neq 0 \quad (\text{不满足约束}) \end{cases}$$

从实践角度看，这样的表达式非常糟糕，但我们可以通过求解以下问题得到相同的效果：

$$\min_x \max_\lambda f(x) + \lambda^T c(x)$$

当 $c(x) \neq 0$ 时，内问题的结果为 $+\infty$。对于不等式约束的类似表达：

$$\min_x f(x) \quad \text{s.t.} \quad c(x) \geq 0 \Rightarrow$$

$$\min_x f(x) + P_\infty^+(c(x))$$

此时，$P_\infty^+(x)$ 定义为：

$$P_\infty^+(x) = \begin{cases} 0, & x \geq 0 \\ +\infty, & x < 0 \end{cases}$$

我们可以将其表示为：

$$\min_x \max_{\lambda \geq 0} f(x) - \lambda^T c(x)$$

其中，$f(x) - \lambda c(x)$ 为拉格朗日函数 $L(x, \lambda)$。

对于凸问题，可以交换最小值和最大值的顺序，解不会改变；这就是对偶问题的概念！然而，在一般情况下（如非凸问题），这种交换不成立。

该表达式的解释是，KKT 条件定义了 (x, λ) 空间中的鞍点。

KKT 系统在最优点时应该有 $dim(x)$ 个正特征值和 $dim(\lambda)$ 个负特征值，最优解下的系统称为“准定的”线性系统。quasi-definite是指，一个具有已知正负特征值的鞍点系统。

总结： 当正规化 KKT 系统时，左下方的块应该是负的！

$$\begin{bmatrix} H + \beta I & C^T \\ C & -\beta I \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta \lambda \end{bmatrix} = \begin{bmatrix} -\nabla_x L \\ -c(x) \end{bmatrix} \quad \text{with} \quad \beta > 0$$

这里左上角要正的，右下角要负的。

这使得系统成为准定的系统。

约束不等式牛顿法及其正则化 在处理约束优化问题时，牛顿法和正则化牛顿法可以有效地求解问题。牛顿法通过计算目标函数的Hessian矩阵和约束的雅可比矩阵，使用KKT条件构造线性系统并进行求解。正则化牛顿法通过引入正则化参数来确保解满足鞍点条件，从而避免计算中的奇异问题。

牛顿法 牛顿法的核心思想是通过最小化拉格朗日函数，计算目标函数和约束条件的梯度，利用以下的 KKT 条件来更新参数 x 和拉格朗日乘子 λ。

首先，构造KKT矩阵 K，包含目标函数的Hessian矩阵 H 和约束函数的雅可比矩阵 C：

$$K = \begin{bmatrix} H & C^T \\ C & 0 \end{bmatrix}$$

然后解以下线性系统，得到增量 Δx 和 $\Delta \lambda$：

$$K \cdot \begin{bmatrix} \Delta x \\ \Delta \lambda \end{bmatrix} = \begin{bmatrix} -\nabla f(x) - C^T \lambda \\ -c(x) \end{bmatrix}$$

牛顿法的迭代过程为：

$$x^{(k+1)} = x^{(k)} + \Delta x$$
$$\lambda^{(k+1)} = \lambda^{(k)} + \Delta \lambda$$

正则化牛顿法 正则化牛顿法通过引入正则化矩阵来保证系统的稳定性。通过修改KKT矩阵并引入正则化参数 β，正则化后的KKT矩阵为：

$$K_{\text{reg}} = K + \beta \cdot \text{diag}(\mathbf{I}, -\mathbf{I})$$

其中，$\beta > 0$ 是正则化系数，确保矩阵的特征值满足准定条件。最后，求解以下线性系统得到更新增量：

$$K_{\text{reg}} \cdot \begin{bmatrix} \Delta x \\ \Delta \lambda \end{bmatrix} = \begin{bmatrix} -\nabla f(x) - C^T \lambda \\ -c(x) \end{bmatrix}$$

正则化牛顿法的迭代过程为：

$$x^{(k+1)} = x^{(k)} + \Delta x$$
$$\lambda^{(k+1)} = \lambda^{(k)} + \Delta \lambda$$

Python 代码展示 以下是实现牛顿法和正则化牛顿法的关键 Python 代码：

```python
#[optimization_inequality_constraints_regularized.py]
# 牛顿步骤函数
def newton_step(x, lam):
    H = hess_f(x) + lam * np.array([[2, 0], [0, 0]])
    C = jacobian_c(x)

    # 构建KKT矩阵
    K = np.block([
        [H, C.T],
        [C, np.zeros((1, 1))]
    ])

    # 构建右侧向量
    rhs = np.concatenate([-grad_f(x) - C.T @ lam, -np.array([c(x)])])

    # 求解线性系统
    delta_z = np.linalg.solve(K, rhs)

    delta_x = delta_z[:2]
    delta_lam = delta_z[2:]

    return delta_x, delta_lam

# 正则化牛顿步骤函数
def regularized_newton_step(x, lam):
    beta = 1.0
    H = hess_f(x) + lam * np.array([[2, 0], [0, 0]])
    C = jacobian_c(x)

    # 构建KKT矩阵
    K = np.block([
        [H, C.T],
        [C, np.zeros((1, 1))]
    ])
```

```
35
36      # 计算特征值
37      e = np.linalg.eigvals(K)
38
39      # 正则化矩阵直到满足鞍点条件
40      while not (np.sum(e > 0) == len(x) and np.sum(e < 0) == len(lam)):
41          reg_matrix = np.zeros_like(K)
42          reg_matrix[:2, :2] = beta * np.eye(2)
43          reg_matrix[2:, 2:] = -beta * np.eye(1)
44          K = K + reg_matrix
45          e = np.linalg.eigvals(K)
46
47      # 求解线性系统
48      rhs = np.concatenate([-grad_f(x) - C.T @ lam, -np.array([c(x)])])
49      delta_z = np.linalg.solve(K, rhs)
50
51      delta_x = delta_z[:2]
52      delta_lam = delta_z[2:]
53
54      return delta_x, delta_lam
55
56  # 迭代方法
57  def iterate_newton_method(x_init, lam_init, max_iter=10, tol=1e-6):
58      x = x_init
59      lam = lam_init
60      for k in range(max_iter):
61          delta_x, delta_lam = newton_step(x, lam)
62          x += delta_x
63          lam += delta_lam
64
65          if np.linalg.norm(delta_x) < tol:
66              break
67      return x, lam
68
69  def iterate_regularized_newton_method(x_init, lam_init, max_iter=10, tol=1e-6):
70      x = x_init
71      lam = lam_init
72      for k in range(max_iter):
73          delta_x, delta_lam = regularized_newton_step(x, lam)
74          x += delta_x
75          lam += delta_lam
76
77          if np.linalg.norm(delta_x) < tol:
78              break
79      return x, lam
```

4.6 Merit Functions (for line search)解决超调问题

如何进行一个根寻找问题的线搜索？考虑如下问题：

$$\text{find}\, x^* \quad \text{s.t.} \quad c(x^*) = 0$$

首先，我们定义一个标量“Merit 函数” $P(x)$，它度量距离解的距离。常用的 Merit 函数包括：

$$P(x) = \frac{1}{2} c(x)^T c(x) = \frac{1}{2} \|c(x)\|_2^2$$

$$\text{或} \quad P(x) = \|c(x)\|_1$$

注意：我们可以使用任何范数。现在我们可以对 $P(x)$ 使用阿米约规则（Armijo Rule）：

Armijo Rule on $P(x)$:

Algorithm 3: Armijo Rule

Input: 初始步长 $\alpha = 1$
Output: 优化的步长 α
while $p(x + \alpha\Delta x) > p(x) + b\nabla p(x)^T \Delta x$ **do**
  $\alpha \leftarrow \alpha/2$;
end
$x \leftarrow x + \alpha\Delta x$

Step Length: $\alpha\nabla p(x)^T \Delta x$ 是期望的减少量，用于松弛防止来回反弹。
β 是一个小的标量，通常 $0 < \beta < 1$。

4.6.1 约束最小化的线搜索

对于约束最小化问题，我们希望提出一个方案，指定我们违反约束的程度，以及我们距离最优解的远近。考虑问题：

$$\min_x f(x)$$

$$\begin{cases} c(x) \leq 0 \\ d(x) = 0 \end{cases}$$

我们可以定义拉格朗日函数为：

$$L(x, \lambda, \mu) = f(x) + \lambda^T c(x) + \mu^T d(x)$$

此时，我们有多种 Merit 函数的选项。其中一个是：

$$P(x, \lambda, \mu) = \frac{1}{2} \| r_{KKT} L(x, \lambda, \mu) \|_2^2$$

这里，$r_{KKT}(x, \lambda, \mu)$ 是 KKT 残差：

$$r_{KKT}(x, \lambda, \mu) = \begin{bmatrix} \nabla_x L(x, \lambda, \mu) \\ \min(0, c(x)) \\ d(x) \end{bmatrix}$$

然而，这不是最佳选择，因为评估 KKT 条件的梯度和牛顿步长的求解一样昂贵。
另一个选择是：

$$P(x, \lambda, \mu) = f(x) + \rho \left\| \begin{bmatrix} \min(0, c(x)) \\ d(x) \end{bmatrix} \right\|_1$$

这里，ρ 是目标函数最小化和约束满足之间的标量折衷。请记住，任何范数都可以用于此（图中使用的是 1 范数），但使用 1 范数是最常见的。这种方法提供了灵活性，因为我们可以在接近最优解时调整 ρ。
又一个选择是：

$$P(x, \lambda, \mu) = f(x) - \lambda^T c(x) + \mu^T d(x) + \frac{\rho}{2} \| \min(0, c(x)) \|_2^2$$

这实际上就是增广拉格朗日方法本身。可以看到merrit-function.ipynb中，未添加KKT residual的情况下，牛顿法将有非常大的过充问题。增加了KKT residual后将采取更保

守的步长。我们可以看到，牛顿法在没有 KKT 残差的情况下会有非常大的过冲问题。添加 KKT 残差后，牛顿法将采取更保守的步长。

```julia
function gauss_newton_step(x, λ)
    H = ∇²f(x)                  # 目标函数 Hessian
    C = ∂c(x)                   # 约束雅可比（行向量）
    # 解 KKT 线性系统 [H  C'; C  0]·[Δx; Δλ] = [-∇f - C'λ; -c]
    Δz = [H  C'; C  0] \ ([-∇f(x) - C' * λ; -c(x)])
    Δx = Δz[1:2];  Δλ = Δz[3]
    return Δx, Δλ
end

# Merit 函数（增广拉格朗日形式）
ρ = 1.0
function P(x, λ)
    f(x) + λ' * c(x) + 0.5 * ρ * dot(c(x), c(x))
end

# Merit 梯度
function ∇P(x, λ)
    g = ∇f(x) + ∂c(x)' * (λ .+ ρ * c(x))
    return [g; c(x)]
end

# 带 Armijo 线搜索的一步更新
function update!(x, λ)
    Δx, Δλ = gauss_newton_step(x, λ)
    α = 1.0
    # Armijo 判据
    while P(x + α * Δx, λ + α * Δλ) > P(x, λ) + 0.01 * α * dot(∇P(x, λ), [Δx; Δλ])
        α *= 0.5
    end
    x_new = x + α * Δx
    λ_new = λ + α * Δλ
    return x_new, λ_new
end
```

图 15: 更新后的不等式约束Merit函数可视化

结论

- 基于 KKT 残差的 $P(x)$ 计算代价较高。
- 过大的罚项权重可能导致问题。
- 增广拉格朗日方法可以与 Merit 函数一起使用。因此，如果我们使用增广拉格朗日来解决问题，只需将其作为 Merit 函数即可。

5 最优控制

5.1 最优控制的历史回顾

TODO: 完成这一部分

5.2 变分法 Calculus of Variations

这里我们将首先介绍变分法，因为最优控制的很多方法都可以看作是变分法的推广，例如动态规划（dynamic programming）和最优控制（optimal control）等。

我们希望寻找一个函数 $y(x)$，使得泛函

$$J[y] = \int_a^b L(x, y(x), y'(x))\, dx$$

取得极值，其中 $L(x, y, y')$ 为已知函数，称为拉格朗日量（Lagrangian），$y(x)$ 是待求函数，满足边界条件 $y(a) = y_a$，$y(b) = y_b$。

为此，引入扰动函数

$$y_\varepsilon(x) = y(x) + \varepsilon\eta(x),$$

其中 ε 为无穷小参数，$\eta(x)$ 为任意光滑函数，且满足端点条件 $\eta(a) = \eta(b) = 0$。

考虑变分：

$$\delta J = \frac{d}{d\varepsilon} J[y_\varepsilon]\bigg|_{\varepsilon=0} = \frac{d}{d\varepsilon}\int_a^b L(x, y+\varepsilon\eta, y'+\varepsilon\eta')\, dx\bigg|_{\varepsilon=0} = \int_a^b \left(\frac{\partial L}{\partial y}\eta + \frac{\partial L}{\partial y'}\eta'\right) dx.$$

对第二项积分分部，得：

$$\int_a^b \frac{\partial L}{\partial y'}\eta'\, dx = \left[\frac{\partial L}{\partial y'}\eta\right]_a^b - \int_a^b \frac{d}{dx}\left(\frac{\partial L}{\partial y'}\right)\eta\, dx.$$

因为 $\eta(a) = \eta(b) = 0$，边界项为零，故变分为：

$$\delta J = \int_a^b \left(\frac{\partial L}{\partial y} - \frac{d}{dx}\left(\frac{\partial L}{\partial y'}\right)\right)\eta(x)\, dx.$$

由于 $\eta(x)$ 是任意满足端点条件的光滑函数，根据变分法基本引理，必有：

$$\frac{\partial L}{\partial y} - \frac{d}{dx}\left(\frac{\partial L}{\partial y'}\right) = 0.$$

这就是著名的 Euler-Lagrange 方程。

例1：最短路径问题 考虑使下式最小的曲线：

$$J[y] = \int_a^b \sqrt{1+y'^2}\, dx.$$

此时 $L = \sqrt{1+y'^2}$，则

$$\frac{\partial L}{\partial y} = 0, \quad \frac{\partial L}{\partial y'} = \frac{y'}{\sqrt{1+y'^2}},$$

代入 Euler-Lagrange 方程：

$$\frac{d}{dx}\left(\frac{y'}{\sqrt{1+y'^2}}\right)=0\Rightarrow\frac{y'}{\sqrt{1+y'^2}}=\text{常数}\Rightarrow y'=\text{常数},$$

因此 $y(x)$ 是直线：$y(x)=mx+c$。

例2：最速降线问题（Brachistochrone Problem） 设一个质点从原点 $(0,0)$ 滑落至点 (x_1,y_1)，在重力作用下沿某曲线运动。我们希望寻找使滑动时间最短的曲线 $y(x)$，其中 $y(x)>0$ 表示向下。

根据能量守恒，质点在位置 y 处的速度为：

$$v=\sqrt{2gy}.$$

滑动时间为：

$$T[y]=\int_0^{x_1}\frac{ds}{v}=\int_0^{x_1}\frac{\sqrt{1+y'^2}}{\sqrt{2gy}}dx.$$

因此泛函为：

$$J[y]=\int_0^{x_1}L(y,y')\,dx,\quad\text{其中}\quad L(y,y')=\frac{\sqrt{1+y'^2}}{\sqrt{2gy}}.$$

接下来我们使用 Euler-Lagrange 方程进行推导：

$$\frac{d}{dx}\left(\frac{\partial L}{\partial y'}\right)-\frac{\partial L}{\partial y}=0.$$

首先计算各个偏导数：

1. 对 y' 的偏导：

$$\frac{\partial L}{\partial y'}=\frac{1}{\sqrt{2gy}}\cdot\frac{y'}{\sqrt{1+y'^2}}.$$

2. 对 y 的偏导：

$$\frac{\partial L}{\partial y}=-\frac{1}{2}\cdot\frac{\sqrt{1+y'^2}}{\sqrt{2g}\cdot y^{3/2}}.$$

然后我们来求导数项：

$$\frac{d}{dx}\left(\frac{\partial L}{\partial y'}\right)=\frac{d}{dx}\left(\frac{y'}{\sqrt{2gy}\cdot\sqrt{1+y'^2}}\right).$$

将其视为乘积微分：

$$\frac{d}{dx}\left(\frac{y'}{\sqrt{y}\sqrt{1+y'^2}}\right)=\frac{1}{\sqrt{1+y'^2}}\cdot\frac{d}{dx}\left(\frac{y'}{\sqrt{y}}\right)+y'\cdot\frac{d}{dx}\left(\frac{1}{\sqrt{1+y'^2}}\right)\cdot\frac{1}{\sqrt{y}}.$$

计算这项非常繁琐，因此我们转而采用简便的方法——**Beltrami 恒等式**，它适用于 L 不显含 x 的情况。

Beltrami 恒等式：

若 L 不显含 x，则：

$$L - y' \frac{\partial L}{\partial y'} = \text{常数}.$$

代入我们的问题：
先计算：

$$\frac{\partial L}{\partial y'} = \frac{y'}{\sqrt{2gy} \cdot \sqrt{1+y'^2}},$$

所以

$$L - y' \frac{\partial L}{\partial y'} = \frac{\sqrt{1+y'^2}}{\sqrt{2gy}} - y' \cdot \frac{y'}{\sqrt{2gy} \cdot \sqrt{1+y'^2}}.$$

合并为一个式子：

$$L - y' \frac{\partial L}{\partial y'} = \frac{1}{\sqrt{2gy}} \left(\sqrt{1+y'^2} - \frac{y'^2}{\sqrt{1+y'^2}} \right) = \frac{1}{\sqrt{2gy}} \cdot \frac{1+y'^2-y'^2}{\sqrt{1+y'^2}} = \frac{1}{\sqrt{2gy} \cdot \sqrt{1+y'^2}}.$$

因此：

$$\frac{1}{\sqrt{2gy(1+y'^2)}} = C.$$

两边取倒数并平方：

$$2gy(1+y'^2) = \frac{1}{C^2}.$$

整理得：

$$1 + y'^2 = \frac{1}{2gC^2 y} \Rightarrow y'^2 = \frac{1}{2gC^2 y} - 1.$$

对这个微分方程进行变换求解。令：

$$\frac{1}{2gC^2} = A \Rightarrow y'^2 = \frac{A}{y} - 1.$$

—

参数化解法：令

$$y = \frac{A}{2}(1 - \cos\theta), \quad x = \frac{A}{2}(\theta - \sin\theta),$$

则此摆线满足上述微分关系，即为最速降线解。

5.3 确定性最优控制 Deterministic Optimal Control

这一节将介绍确定性最优控制，Pontryagin's Maximun Principle（庞特里亚金最大值原理）和线性二次调节器（Linear Quadratic Regulator, LQR）问题。

首先我们考虑这样的一个控制问题：

$$\begin{aligned} \min_{x(t),u(t)} \quad & J(x(t),u(t)) = \int_{t_0}^{t_f} L(x(t),u(t))\,dt + L_F(x(t_f)) \\ \text{s.t.} \quad & \dot{x}(t) = f(x(t),u(t)) \\ & \text{and 其他约束条件...} \end{aligned} \tag{36}$$

这里我们通过最小化cost functional J (是一个泛函)来求解控制问题。$x(t)$ 是状态变量，$u(t)$ 是控制变量， $L(x(t),u(t))$ 是状态成本（stage cost），$L_F(x(t_f))$ 是终端成本（terminal cost），$f(x(t),u(t))$ 是系统动力学方程（动力学的限制）。这是一个“无限维”问题，即需要无限多个离散时间的控制点来完全描述要施加的控制，也就是说

$$u(t)$$

是一个函数而不是一个数值。但对于计算机控制来说，我们只能在有限个离散时间点上施加控制，因此我们需要将这个问题离散化。我们可以通过控制时间间隔来无限逼近这个无限时间问题。

$$u(t) = \lim_{N\to\infty} u_n(t) = \lim_{N\to\infty} u_n(t_0 + n\Delta t), \quad n = 0, 1, \ldots, N \tag{37}$$

- 该问题的解是开环（open-loop）轨迹。
- 目前只有极少数控制问题在连续时间下有解析解。
- 我们重点关注离散时间下的情形，在这种情况下可以采用可行的算法。

5.3.1 离散时间 Discrete Time

考虑该问题的离散时间版本：

$$\begin{aligned}
&\min_{x_{1:N},\, u_{1:N-1}} \quad J(x_{1:N}, u_{1:N-1}) = \sum_{k=1}^{N-1} L(x_k, u_k) + L_F(x_N) \\
&\text{s.t.} \quad x_{n+1} = f(x_n, u_n) \\
&\qquad u_{\min} \le u_k \le u_{\max} \quad \text{力矩约束（Torque limits）} \\
&\qquad c(x_k) \le 0 \quad \forall k \quad \text{障碍/碰撞约束（Obstacle / collision constraints）}
\end{aligned}$$

- 这个问题的离散时间版本现在是一个有限维问题。
- 采样点 x_k, u_k 常被称为“节点”（knot points）。
- 我们可以使用如 Runge-Kutta 方法等积分方法，将连续系统转换为离散时间问题。(见**??**节)
- 最后，也可以通过插值法将离散时间问题转回连续时间问题。

5.3.2 Pontryagin 最小值原理

- Pontryagin 最小值原理（Pontryagin's Minimum Principle），如果我们最大化奖励函数则又称为“最大值原理”。
- 本质上为确定性最优控制问题提供了**一阶必要条件**。
- 在离散时间下，它其实是 KKT 条件的一个特例。

考虑我们之前的问题：

$$\min_{x_{1:N},\, u_{1:N-1}} \quad J(x_{1:N}, u_{1:N-1}) = \sum_{k=1}^{N-1} L(x_k, u_k) + L_F(x_N)$$
$$\text{s.t.} \quad x_{n+1} = f(x_n, u_n)$$

在本设置中，我们主要考虑控制约束（如力矩限制），但难以处理状态约束（如碰撞约束）。

我们可以写出该问题的 Lagrangian 如下：

$$L = \sum_{k=1}^{N-1} \left[l(x_k, u_k) + \lambda_{k+1}^T (f(x_k, u_k) - x_{k+1}) \right] + L_F(x_N)$$

这通常以“哈密顿量”形式表达： [2] [3]

$$H(x, u, \lambda) = l(x, u) + \lambda^T f(x, u)$$

将 H 带入 L 中，得

$$L = H(x_1, u_1, \lambda_2) + \left[\sum_{k=2}^{N-1} H(x_k, u_k, \lambda_{k+1}) - \lambda_k^T x_k \right] + L_F(x_N) - \lambda_N^T x_N$$

对 x 和 λ 求偏导数（必要条件）：

$$\frac{\partial L}{\partial x_k} = \frac{\partial H}{\partial x_k} + \frac{\partial f}{\partial x_k}^T \lambda_{k+1} - \lambda_k = 0$$

$$\frac{\partial L}{\partial \lambda_k} = f(x_k, u_k) - x_{k+1} = 0$$

$$\frac{\partial L}{\partial x_N} = \frac{\partial L_F}{\partial x_N} - \lambda_N = 0$$

对控制量 u，我们显式写出极小化过程（考虑控制约束）：

$$u_k = \arg\min_{\tilde{u}} H(x_k, \tilde{u}, \lambda_{k+1}), \quad \text{s.t. } \tilde{u} \in \mathcal{U}$$

其中 $\mathcal{U}$ 是可行控制集合，比如 $u_{\min} \le \tilde{u} \le u_{\max}$。

---

[2] Hamiltonian 是一个物理学术语，通常用于描述系统的总能量。在最优控制中指的是“当前成本+未来影响的总和”，有点像RL中的“价值函数”的意思。公式中的

- $l(x,u)$表示当前时刻的损失，如u^2能量耗散。
- $f(x,u)$是系统方程描述x如何变化，
- λ是协态变量（co-state variable），表示对未来影响的权重。
- $\lambda^T f(x,u)$是“当前状态变化对未来目标的影响”——像是提前把未来的损失加进来。

[3] 协态变量（co-state variable），也叫伴随变量（adjoint variable）在最优化问题里，有约束时我们会引入拉格朗日乘子，衡量目标函数对约束的“敏感度”。协态变量其实就是动态系统里的“拉格朗日乘子”，用来处理动力学约束 $\dot{x} = f(x,u)$。直观来说，协态变量告诉你“如果当前状态发生微小变化，对最终目标的影响有多大”。

总结如下：

$$\begin{cases} x_{k+1} = \nabla_\lambda H(x_k, u_k, \lambda_{k+1}) = f(x_k, u_k) \\ \lambda_k = \nabla_x H(x_k, u_k, \lambda_{k+1}) = \nabla_x l(x_k, u_k) + \left(\dfrac{\partial f}{\partial x}\right)^T \lambda_{k+1} \\ u_k = \arg\min\limits_{\tilde{u}} H(x_k, \tilde{u}, \lambda_{k+1}), \quad \text{s.t. } \tilde{u} \in \mathcal{U} \\ \lambda_N = \dfrac{\partial L_F}{\partial x_N} \end{cases}$$

连续时间形式：

$$\dot{x} = \nabla_\lambda H(x, u, \lambda) = f_{continuous}(x, u)$$

$$\dot{\lambda} = \nabla_x H(x, u, \lambda) = \nabla_x l(x, u) + \left(\frac{\partial f_{continuous}}{\partial x}\right)^T \lambda$$

$$u = \arg\min_{\tilde{u}} H(x, \tilde{u}, \lambda), \quad \text{s.t. } \tilde{u} \in \mathcal{U}$$

$$\lambda_N = \frac{\partial L_F}{\partial x_N}$$

Pontryagin's 实际例子 下面用一个经典最优控制问题，详细说明 Pontryagin 最小值原理的求解过程。

1. **问题表述（离散形式）**

   控制一维质量点的运动，在 N 个时间步内从已知初态 (x_1, v_1) 移动到已知终态 (x_N, v_N)，使控制能量最小。

$$\min_{x_{1:N},\, v_{1:N},\, u_{1:N-1}} \quad J = \sum_{k=1}^{N-1} u_k^2$$

$$\text{s.t. } \begin{cases} x_{k+1} = x_k + h v_k \\ v_{k+1} = v_k + h u_k \end{cases}$$

   其中 h 是步长，u_k 是控制输入（加速度），x_k 是位置，v_k 是速度。

2. **写出Lagrangian**

   为了引入动力学约束，引入协态变量 λ_{k+1}^x, λ_{k+1}^v，Lagrangian为：

$$L = \sum_{k=1}^{N-1} \left[u_k^2 + \lambda_{k+1}^x (x_k + h v_k - x_{k+1}) + \lambda_{k+1}^v (v_k + h u_k - v_{k+1})\right] + 0$$

   （这里认为final cost=0）

3. **写出Hamiltonian**

$$H(x_k, v_k, u_k, \lambda_{k+1}^x, \lambda_{k+1}^v) = u_k^2 + \lambda_{k+1}^x (v_k) + \lambda_{k+1}^v u_k$$

   其中，$l(x_k, u_k) = u_k^2$，$f(x_k, v_k, u_k) = (v_k, u_k)$。

4. **将Hamiltonian带入Lagrangian**

结合模板公式，Lagrangian可以表达为：

$$L = H(x_1, v_1, u_1, \lambda_2^x, \lambda_2^v) + \sum_{k=2}^{N-1} \left[H(x_k, v_k, u_k, \lambda_{k+1}^x, \lambda_{k+1}^v) - (\lambda_k^x x_k + \lambda_k^v v_k) \right] - (\lambda_N^x x_N + \lambda_N^v v_N)$$

5. **对 x_k, v_k, λ_k 求偏导（必要条件）**

- 对 x_k 求导：

$$\frac{\partial L}{\partial x_k} = \lambda_{k+1}^x - \lambda_k^x = 0 \implies \lambda_{k+1}^x = \lambda_k^x$$

- 对 v_k 求导：

$$\frac{\partial L}{\partial v_k} = h\lambda_{k+1}^x + \lambda_{k+1}^v - \lambda_k^v = 0 \implies \lambda_k^v = \lambda_{k+1}^v + h\lambda_{k+1}^x$$

- 对 $\lambda_{k+1}^x, \lambda_{k+1}^v$ 求导得到动力学约束：

$$x_{k+1} = x_k + hv_k$$

$$v_{k+1} = v_k + hu_k$$

6. **对控制 u_k 求极小（Pontryagin条件）**

$$u_k = \arg\min_u H(x_k, v_k, u, \lambda_{k+1}^x, \lambda_{k+1}^v)$$

具体为：

$$\frac{\partial H}{\partial u_k} = 2u_k + \lambda_{k+1}^v = 0 \implies u_k^* = -\frac{1}{2}\lambda_{k+1}^v$$

7. **配合边界条件解联立方程**

$$x_1 = x_0, \quad v_1 = v_0, \quad x_N = x_f, \quad v_N = v_f$$

λ_N^x, λ_N^v 可由终点条件确定

联立以上递推关系与边界条件，即可数值求解 $x_k, v_k, \lambda_k^x, \lambda_k^v, u_k$ 的最优轨迹。

7. Python 数值解法（简要代码说明）

1. 定义微分方程，将 λ_1 和 λ_2 作为扩展变量，一起求解。
2. 将 u 用 λ_2 表达，代入方程。
3. 用 `scipy.integrate.solve_bvp` 设置微分方程和边界条件，数值求解。
4. 画出 $x(t), v(t), u(t)$ 的最优轨迹。

8. 代码完整见[OptimalControl/PMP.py] 本代码实现的离散最优控制问题数学描述如下：

1. **优化目标：**

$$\min_{u_0,\ldots,u_{N-2}} J = \sum_{k=0}^{N-2} u_k^2$$

2. **系统动力学（离散）：**

$$\begin{cases} x_{k+1} = x_k + hv_k \\ v_{k+1} = v_k + hu_k \end{cases} \qquad \text{for } k = 0, 1, \ldots, N-2$$

3. **边界条件：**

$$x_0 = x_{\text{init}}, \quad v_0 = v_{\text{init}}, \quad x_{N-1} = x_{\text{final}}, \quad v_{N-1} = v_{\text{final}}$$

4. **协态变量递推（伴随方程，反向递推）：**

$$\lambda_k^x = \lambda_{k+1}^x$$

$$\lambda_k^v = \lambda_{k+1}^v + h\lambda_{k+1}^x$$

(终点条件：$\lambda_{N-1}^x, \lambda_{N-1}^v$ 用射击法确定)

5. **最优性条件（Pontryagin极小）：**

$$u_k^* = \arg\min_u \left(u^2 + \lambda_{k+1}^v u\right) \implies u_k^* = -\frac{1}{2}\lambda_{k+1}^v$$

6. **整体递推流程（数值实现）：**

已知 x_0, v_0, 猜测 $\lambda_{N-1}^x, \lambda_{N-1}^v$

$$\text{for } k = N-2 \to 0: \quad \begin{cases} \lambda_k^x = \lambda_{k+1}^x \\ \lambda_k^v = \lambda_{k+1}^v + h\lambda_{k+1}^x \end{cases}$$

$$\text{for } k = 0 \to N-2: \quad \begin{cases} u_k = -\frac{1}{2}\lambda_{k+1}^v \\ x_{k+1} = x_k + hv_k \\ v_{k+1} = v_k + hu_k \end{cases}$$

调整 $\lambda_{N-1}^x, \lambda_{N-1}^v$ 使得 $x_{N-1} = x_{\text{final}}, v_{N-1} = v_{\text{final}}$

8. shoting法理解 通俗理解：射击法就像打靶：你站在起点（已知初始状态），目标是命中远处的靶心（终点约束），但你不知道子弹出膛时该瞄准哪个方向（未知的初始/终点“协态”参数）。所以你“先猜一下方向”，打出去，看结果离靶心有多远。然后不断调整你的“瞄准角度”，直到正中靶心。

数学理解：在两点边值问题里，只有初值（如 x_0, v_0）是已知的，协态变量的终点值（如 λ_N^x, λ_N^v）是未知的。我们将协态变量的终点值当作“待定参数”来猜测。用这些猜测值，正向/反向递推系统状态和协态，看看末端 x_N, v_N 距离目标还有多远（残差）。用数值优化器（比如 root）不断调整协态的终点猜测，使残差收敛为0，就得到了完整的最优轨迹。

7. **9. 数值结果**

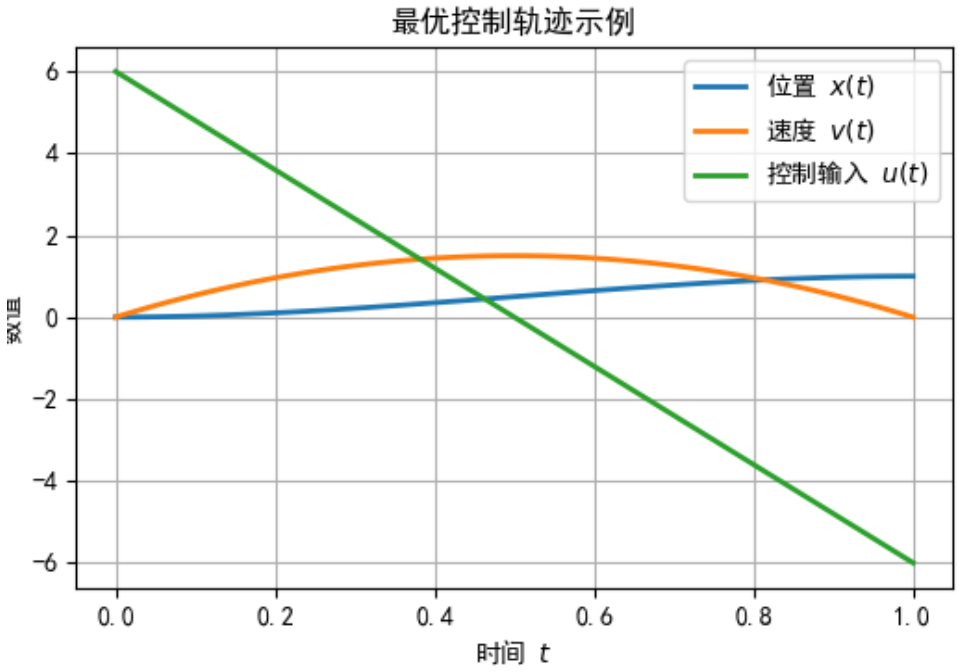


图 16: 离散最优控制问题的数值解与控制轨迹示例。图中显示了位置 $x(t)$、速度 $v(t)$ 和控制输入 $u(t)$ 随时间的变化。

由图 ?? 可见：

- 位置 $x(t)$ 从初始点 0 平滑地到达终点 1，轨迹为一条凸二次曲线；
- 速度 $v(t)$ 在中途达到最大值，末端回到 0，实现了“平滑启动和平滑停止”；
- 控制输入 $u(t)$ 随时间线性减小，为一条斜率为负的直线，这符合 $u(t) = at + b$ 的结构，是能量最优问题的典型特征。

这种结果显示了在能量最小的约束下，最优控制策略是** “温和施加加速度” **，而不是突变或极值切换。控制量 $u(t)$ 的线性变化，保证了既满足动态约束，又能使能量消耗最小。整体轨迹平滑，末端严格达到指定位置与速度。

5.4 线性二次调节器 Linear Quadratic Regulator (LQR)

LQR（线性二次调节器）是一类非常重要的最优控制方法，广泛应用于线性系统。LQR问题的目标是在满足线性动力学的约束下，使得系统的某一二次型性能指标最小化。数学表达如下：

$$\min_{x_{1:N},\, u_{1:N-1}} \sum_{k=1}^{N-1} \left[\frac{1}{2} x_k^T Q x_k + \frac{1}{2} u_k^T R u_k \right] + \frac{1}{2} x_N^T Q_N x_N \tag{38}$$

$$\text{s.t. } x_{k+1} = A_k x_k + B_k u_k \tag{39}$$

其中：

- $Q \succeq 0$ 用于惩罚系统状态，$R \succ 0$ 用于惩罚控制输入。
- Q_N 用于终端状态惩罚。

- A_k, B_k 分别为系统的离散时间动力学矩阵，若对所有时刻均相同，则称为时不变LQR，否则为时变LQR。

LQR的一些性质：

- LQR的目标通常是将系统状态驱动至原点（$x = 0$）。
- 当 A_k, B_k, Q_k, R_k 全部为常数时，称为“时不变LQR”；否则为“时变LQR”（TVLQR）。
- LQR常用于平衡点的稳定，TVLQR适用于轨迹跟踪问题。
- LQR理论可以（在局部）近似描述许多非线性问题，因此非常常用。
- LQR还有很多扩展形式，如无限时域LQR、随机LQR等。
- LQR被称为“控制理论皇冠上的明珠”。

5.4.1 间接射击法下的LQR（不推荐）

根据哈密顿量理论，LQR问题可用如下递推公式求解：

$$x_{k+1} = A_k x_k + B_k u_k \tag{31}$$
$$\lambda_k = Q x_k + A_k^T \lambda_{k+1}, \quad \lambda_N = Q_N x_N \tag{32}$$
$$u_k = -R^{-1} B_k^T \lambda_{k+1} \tag{33}$$

上述公式给出了：

- x_k 的正向递推（动力学方程）；
- λ_k 的反向递推（伴随方程）；
- u_k 的解析最优解（最优性条件）。

间接射击法求解LQR的基本流程：

1. 初始猜测 $u_{1:N-1}$ 及 x_0，用系统动力学正向递推出所有 $x_{1:N}$。
2. 反向递推伴随变量 λ_k，即 $\lambda_k = Q x_k + A_k^T \lambda_{k+1}$，终点 $\lambda_N = Q_N x_N$。
3. 用 $\Delta u_k = (u_k - u_{\text{old}})$ 修正输入，并用 $u_{\text{new}} = u_{\text{old}} + \alpha \Delta u$ 做线搜索，更新控制律。
4. 重复步骤2-3，直至 Δu 收敛。

5.4.2 例子：双积分器系统

以双积分器（double integrator）为例，其动力学连续时间表达式为：

$$\dot{x} = \begin{bmatrix} q \\ \dot{q} \end{bmatrix}, \quad \begin{bmatrix} \dot{q} \\ \ddot{q} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} q \\ \dot{q} \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} u$$

将其离散化，得到：

$$x_{k+1} = \begin{bmatrix} 1 & h \\ 0 & 1 \end{bmatrix} x_k + \begin{bmatrix} \frac{1}{2}h^2 \\ h \end{bmatrix} u_k$$

其中，$\begin{bmatrix} 1 & h \\ 0 & 1 \end{bmatrix}$ 为系统矩阵 A，$\begin{bmatrix} \frac{1}{2}h^2 \\ h \end{bmatrix}$ 为控制矩阵 B，h 为离散时间步长。

5.5 LQR 作为二次规划（QP）问题

假设初始状态 x_1 已知（不是优化变量）。定义决策变量 z 为：

$$z = \begin{bmatrix} u_1 \\ x_2 \\ u_2 \\ \vdots \\ x_N \end{bmatrix}$$

再定义 Hessian 矩阵 H：

$$H = \begin{bmatrix} R_1 & 0 & \cdots & 0 \\ 0 & Q_2 & \cdots & 0 \\ 0 & 0 & \cdots & Q_N \end{bmatrix}$$

于是目标函数可写为 $J = \frac{1}{2} z^T H z$。再定义约束矩阵 C 和向量 d：

$$C = \begin{bmatrix} B_1 & (-I) & 0 & \cdots & 0 & & \\ A & B & (-I) & \cdots & 0 & & \\ 0 & A_2 & B_2 & \cdots & (-I) & & \\ \vdots & \vdots & \vdots & \ddots & \vdots & & \\ 0 & 0 & 0 & \cdots & A_{N-1} & B_{N-1} & (-I) \end{bmatrix}$$

$$d = \begin{bmatrix} A_1 x_1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

于是，LQR 问题可以写成标准的二次规划问题（QP）：

$$\min_z \quad \frac{1}{2} z^T H z \tag{40}$$

$$\text{s.t.} \quad Cz = d \tag{41}$$

其 Lagrangian 为

$$L(z, \lambda) = \frac{1}{2} z^T H z + \lambda^T [Cz - d]$$

KKT 条件为

$$\nabla_z L = Hz + C^T \lambda = 0 \tag{42}$$

$$\nabla_\lambda L = Cz - d = 0 \tag{43}$$

可整理为一个大的线性系统

$$\begin{bmatrix} H & C^T \\ C & 0 \end{bmatrix} \begin{bmatrix} z \\ \lambda \end{bmatrix} = \begin{bmatrix} 0 \\ d \end{bmatrix}$$

只需解这个线性方程组即可得到LQR的精确解!

5.5.1 例子

同样以双积分器（sliding brick）为例，可以直接比较 QP 解和间接射击法的效果，QP解往往收敛性更好、效率更高。

5.5.2 Riccati 递推：LQR QP 的结构

LQR的QP KKT线性系统很稀疏，结构高度规则。以如下块状矩阵为例：

$$\left[\begin{array}{cccc|cccc} R & 0 & 0 & 0 & B^T & 0 & 0 & 0 \\ 0 & Q & 0 & 0 & -I & A^T & 0 & 0 \\ 0 & 0 & Q & 0 & 0 & -I & A^T & 0 \\ 0 & 0 & 0 & Q_N & 0 & 0 & -I & 0 \\ \hline B & -I & 0 & 0 & 0 & 0 & 0 & 0 \\ A & B & -I & 0 & 0 & 0 & 0 & 0 \\ 0 & A & B & -I & 0 & 0 & 0 & 0 \\ 0 & 0 & A & B & 0 & 0 & 0 & 0 \end{array}\right] \begin{bmatrix} u_1 \\ x_2 \\ u_2 \\ x_3 \\ \lambda_2 \\ \lambda_3 \\ \lambda_4 \\ \lambda_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ -A_1 x_1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

考虑上三角的最后一行（虚线以上部分）：

$$Q_N x_4 - \lambda_4 = 0 \implies \lambda_4 = Q_N x_4$$

再看倒数第二行：

$$R u_3 + B^T \lambda_4 = 0 \implies u_3 = -(R + B^T Q_N B)^{-1} B^T Q_N A x_3$$

递推出一个反馈增益 $K_3 = (R + B^T Q_N B)^{-1} B^T Q_N A$。

再看倒数第三行：

$$Q x_3 - \lambda_3 + A^T \lambda_4 = 0$$

代入 λ_4 和 u_3 后，可以写成

$$Q x_3 - \lambda_3 + A^T Q_N (A x_3 + B u_3) = 0$$

进一步递推，可以定义

$$P_3 = Q + A^T Q_N (A - B K_3)$$

最终，得到 Riccati 方程递推形式：

$$P_N = Q_N \tag{44}$$

$$K_n = (R + B^T P_{n+1} B)^{-1} B^T P_{n+1} A \tag{45}$$

$$P_n = Q + A^T P_{n+1} (A - B K_n) \tag{46}$$

要点总结：

- 这就是著名的 Riccati 方程或 Riccati 递推。
- 由于 QP 结构稀疏，递推可高效实现，复杂度 $O(Nn^3)$ 而不是 $O(N^3(n+m)^3)$。
- 这让我们可以得到一个反馈律 $u = -Kx$，而不只是开环最优轨迹。

5.6 无限时域 LQR (Infinite Horizon LQR)

下面考虑无限时域的LQR问题。

- 对于时不变LQR，K 增益矩阵在无限时域下会收敛到一个常数值。
- 对于稳定化问题，几乎总是使用这个常数 K。
- 对于 P 的递推，可以用不动点迭代法或求根法数值求解。Julia/Matlab/Python 中的 `dare` 函数都可以直接求解。

5.6.1 有限时域 LQR

$$\min_{x_{1:N},u_{1:N-1}} \sum_{k=1}^{N-1} \left[\frac{1}{2}x_k^T Q x_k + \frac{1}{2}u_k^T R u_k\right] + \frac{1}{2}x_N^T Q_N x_N \tag{47}$$

$$\text{s.t. } x_{k+1} = A_k x_k + B_k u_k \tag{48}$$

其中 A_k, B_k 可以随时间变化。

5.6.2 无限时域 LQR

$$\min_{x_{1:\infty},u_{1:\infty}} \sum_{k=1}^{\infty} \left[\frac{1}{2}x_k^T Q x_k + \frac{1}{2}u_k^T R u_k\right] \tag{49}$$

$$\text{s.t. } x_{k+1} = A x_k + B u_k \tag{50}$$

其中 A 和 B 是时不变的。

5.7 如何求解LQR控制律（How to solve the policy）

有两种常见求解策略：

- $\pi(x)$：每次在线求解QP问题（太慢）。
- $\pi(x) = -Kx$：对于无限时域LQR，只需一次性计算出反馈增益 K，即可通过线性反馈生成所有的 U_k。

如果我们想让机器人到达任意目标点，该如何做？已知目标点 x_{goal}，有

$$x_{\text{goal}} = A x_{\text{goal}} + B u_{\text{goal}} \tag{51}$$

$$\tilde{x} = x - x_g, \quad \tilde{U} = U - U_g \tag{52}$$

则有

$$x_{k+1} - x_g = A x_k + B U_k - (A x_g + B U_g) \tag{53}$$

$$x_{k+1} - x_g = A(x_k - x_g) + B(U_k - U_g) \tag{54}$$

这样，只需对 $\tilde{x}, \tilde{U}$ 求解LQR，得到反馈增益

$$K = \mathrm{dLQR}(A, B, Q, R) \tag{55}$$
$$\tilde{U} = -K\tilde{x} \tag{56}$$
$$U - U_g = -K(x - x_g) \tag{57}$$
$$U = -K(x - x_g) + U_g \tag{58}$$

因此，我们可以把机器人驱动到任意目标。如果A, B, Q, R都是常数，K也是常数。如果A, B是时变的，就需要每个时刻都重新计算K（可离线提前算好）。

5.7.1 6.4.1 非线性系统的扩展

对某个定点进行泰勒展开，设 $\bar{x} = f(\bar{x}, \bar{u})$，则动态函数可线性化为：

$$x_{k+1} \approx f(\bar{x}, \bar{u}) + \left.\frac{\partial f}{\partial x}\right|_{\bar{x},\bar{u}} (x - \bar{x}) + \left.\frac{\partial f}{\partial u}\right|_{\bar{x},\bar{u}} (u - \bar{u}) \tag{59}$$
$$\bar{x} + \Delta x_{k+1} = f(\bar{x}, \bar{u}) + A\Delta x + B\Delta u \tag{60}$$
$$\Delta x_{k+1} = \bar{A}\Delta x + \bar{B}\Delta u \tag{61}$$

5.8 可控性（Controllability）

如何判断 LQR 能否工作？对于时不变系统，有一个简单的答案。对任意初始状态 x_0，x_n 的表达式为：

$$x_n = Ax_{n-1} + Bu_{n-1} \tag{62}$$
$$= A(Ax_{n-2} + Bu_{n-2}) + Bu_{n-1} \tag{63}$$
$$= A^n x_0 + A^{n-1}Bu_0 + \ldots + Bu_{n-1} \tag{64}$$
$$= [B\ AB\ A^2B\ \ldots\ A^{n-1}B] \begin{bmatrix} u_{n-1} \\ u_{n-2} \\ \vdots \\ u_0 \end{bmatrix} + A^n x_0 \tag{65}$$

我们的目标是让 $x_n = 0$。可以定义可控性矩阵 C：

$$C = [B\ AB\ A^2B\ \ldots\ A^{n-1}B]$$

要想从任意 x_0 驱动到任意 x_n，可控性矩阵必须满秩：

$$\mathrm{rank}(C) = n, \quad n = \dim(x)$$

这等价于对 $u_0, \ldots, u_{n-1}$ 求解如下最小二乘问题：

$$\begin{bmatrix} u_{n-1} \\ u_{n-2} \\ \vdots \\ u_0 \end{bmatrix} = \left[C^T(CC^T)^{-1}\right][x_n - A^n x_0]$$

其中 $[C^T(CC^T)^{-1}]$ 是 C 的伪逆，需要 CC^T 可逆。由于 Cayley-Hamilton 定理，A^n 可以写成更小幂次 A 的线性组合：

$$A^n = \sum_{k=0}^{n-1} \alpha_k A^k$$

因此，继续增加步数/增加 C 的列数，并不能提高 C 的秩。在时变系统情况下，用QP方法比上述解法更好。

6 动态规划

6.1 贝尔曼原理

- 最优控制问题本质上具有顺序结构。
- 过去的控制输入会影响未来的状态，但未来的控制输入不能影响过去的状态。
- 贝尔曼原理（即最优性原理）指出了这一点对最优轨迹的影响。
- 最优轨迹的子轨迹对于相应定义的子问题也必须是最优的。

6.2 动态规划

- 贝尔曼原理建议我们从轨迹的终点开始倒推。
- 在Riccati方程以及Pontryagin的协态/乘子方程中已经隐约看到这一点。
- 定义“最优代价-to-go”，即“价值函数”$V_N(x)$。
- 它表示如果我们最优地行动，从时刻k的状态x开始所产生的累计代价。
- 对于LQR问题，有：

$$V_N(x) = \frac{1}{2}x^T Q_N x = \frac{1}{2}x^T P_N x \tag{1}$$

- 现在我们可以回退一步，计算$V_{N-1}(x)$：

$$\begin{aligned}&\min_u \frac{1}{2}x_{N-1}^T Q_{N-1}x_{N-1} + \frac{1}{2}u^T R u + V_N(A_{N-1}x_{N-1} + B_{N-1}u)\\ &= \min_u \frac{1}{2}u^T R u + \frac{1}{2}(A_{N-1}x_{N-1} + B_{N-1}u)^T Q_N (A_{N-1}x_{N-1} + B_{N-1}u)\end{aligned}$$

- 对于最优动作，有∇_u该代价为0，即

$$u^T R_{N-1} + (A_{N-1}x_{N-1} + B_{N-1}u)^T Q_B B_{N-1} = 0 \tag{4}$$

$$\implies u_{N-1} = -(R_{N-1} + B_{N-1}^T Q_N B_{N-1})^{-1} B_{N-1}^T Q_N A_{N-1} x_{N-1} \tag{5}$$

- 可以定义$K_{N-1} = (R_{N-1} + B_{N-1}^T Q_N B_{N-1})^{-1} B_{N-1}^T Q_N A_{N-1}$以便书写。
- 将$u = -Kx$代入$V_{N-1}(x)$的表达式，有：

$$V_{N-1}(x) = \frac{1}{2}x^T (Q_{N-1} + K_{N-1}^T R_{N-1} K + (A_{N-1} - B_{N-1}K_{N-1})^T Q_N (A_{N-1} - B_{N-1}K_{N-1}))x \tag{6}$$

- 可以定义$P_{N-1} = (Q_{N-1}+K_{N-1}^T R_{N-1} K+(A_{N-1}-B_{N-1}K_{N-1})^T Q_N (A_{N-1}-B_{N-1}K_{N-1}))$，使得：

$$V_{N-1}(x) = \frac{1}{2} x^T P_{N-1} x \tag{7}$$

- 现在我们得到了K和P的递推关系，可以一直迭代到$k=1$。这其实就是Riccati方程。K与前面提到的Riccati方程中的一样，而P与之前的不同。在数值计算上，这个P是对称且更优的。

6.2.1 动态规划算法

反向DP算法

1: $V_N(x) \leftarrow l_N(x)$
2: $K \leftarrow N$
3: **while** $K > 1$ **do**
4: $V_{k-1} = \min_u [l(x,u) + V_k(f(x,u))]$ ▷ 贝尔曼方程
5: $k \leftarrow k-1$

- 如果我们知道$V_k(x)$，最优反馈策略为：

$$u_k(x) = \arg\min_u [l(x_k,u) + V_{k+1}(f(x_k,u))] \tag{8}$$

- DP方程也可以用行动-价值（action-value）或Q函数等价地表示：

$$S_k(x,u) = l(x,u) + V_{k+1}(f(x,u)) \tag{9}$$

$$u_k(x_k) = \arg\min_u S_k(x_k,u) \tag{10}$$

- 这些通常记为$Q(x,u)$，这里我们用S，因为Q在LQR状态代价中也被使用。

6.2.2 维数灾难 (The Curse)

- 对于全局最优，DP方法是充分的。
- 但它只对简单问题（如LQR或低维问题）可行。
- $V(x)$在LQR问题中保持二次型，但对一般非线性问题则难以解析表达。
- 即使可以表达，$\min_u S(x,u)$通常是非凸的，单独求解也可能很困难。
- 随着状态维数增加，DP的计算量急剧上升，因为$V(x)$难以表示。

6.2.3 我们为什么关心这些？

- 近似DP（用函数逼近来表示$V(x)$或$S(x,u)$）非常有效。
- 是现代强化学习方法的基础。
- DP很好地推广到随机问题（只需将所有内容包裹在期望算子下），而Pontryagin方法则不能。

6.2.4 什么是拉格朗日乘子？

- 回忆QP中的Riccati推导：

$$\lambda_n = P_n x_n \tag{11}$$

$$P_n = Q + A^T P_{n+1}(A - BK) \tag{12}$$

$$= Q + K^T R K + (A - BK)^T P_{n+1}(A - BK)$$

$$V_n(x) = \frac{1}{2} x^T P_n x \tag{14}$$

$$\lambda_n = \nabla_x V_n(x) \tag{15}$$

- 动力学的乘子就是代价对状态的梯度。
- 这种思想可推广到一般非线性情形，不仅限于LQR。

6.3 例子

请记得查看课程视频中的例子，其中QP中的λ_n与DP中的$\nabla_x V_n(x)$是对应的。

Appendix