控制理论实用指南

Skywalker Liu
ltx2499323877@outlook.com
github.com/liuskywalkerjskd

2025 年 10 月

目录

1 引言 **6**

2 数字信号处理 **7**
2.1 理解频域 . . . 7
2.2 拉普拉斯变换与傅里叶变换 . . . 7
2.2.1 拉普拉斯变换 . . . 7
2.2.2 傅里叶变换 . . . 8
2.2.3 频谱 . . . 8
2.3 基于频域的滤波器设计 . . . 10
2.3.1 低通滤波器 . . . 10
2.3.2 带通滤波器 . . . 12
2.3.3 高通滤波器 . . . 12
2.3.4 滤波器类型选择与参数整定 . . . 13
2.4 Z 变换 . . . 13

3 系统描述 **15**
3.1 系统的数学模型 . . . 15
3.1.1 微分方程（Differential Equations） . . . 15
3.1.2 传递函数（Transfer Functions） . . . 15
3.1.3 状态空间表示（State-Space Representation） . . . 16
3.2 基于传递函数的系统分析 . . . 17
3.3 基于状态空间的系统分析 . . . 20
3.4 电机建模：最常见的被控对象 . . . 20

4 经典控制理论 **25**
4.1 反馈控制的基本概念 . . . 25
4.2 系统性能指标 . . . 26
4.2.1 稳定裕度 . . . 26
4.3 基于传递函数的控制器设计 . . . 28
4.4 PID 控制器 . . . 28
4.4.1 PID 控制器——连续形式 . . . 29
4.4.2 PID 控制器——离散形式 . . . 31
4.4.3 改进的 PID 控制器 . . . 31
4.5 PID 进阶改进 . . . 32
4.5.1 积分分离（Integral Separation） . . . 32
4.5.2 微分位置：PID vs PI-D vs I-PD . . . 33
4.5.3 滤波 PID：频域证明 . . . 34
4.5.4 二自由度 PID（2-DOF PID） . . . 36
4.5.5 模糊 PID（Fuzzy PID） . . . 37
4.6 PID 整定方法 . . . 38
4.6.1 经验整定 . . . 38
4.6.2 使用 MATLAB Simulink 整定 . . . 39

4.7 系统辨识：搞清楚你的被控对象究竟如何工作 . . . 39
4.7.1 阶跃响应法（时域） . . . 39
4.7.2 频率响应法（Bode 辨识） . . . 40
4.7.3 最小二乘辨识（自动化） . . . 40
4.7.4 竞赛机器人的实用系统辨识流程 . . . 41
4.8 前馈控制（Feedforward Control） . . . 41
4.9 PID 架构：单级、串级与并联串级 . . . 43
4.9.1 架构一：单级 PID . . . 43
4.9.2 架构二：串级（级联）PID . . . 43
4.9.3 架构三：并联串级 PID（多轴协同） . . . 44
4.9.4 架构对比总结 . . . 45
4.9.5 并联 PID 的输出冲突问题 . . . 46
4.9.6 思考题：串级 PID 的内环 K_p 等价于外环 K_d 吗? . . . 46

5 离散化与实现 48

5.1 从连续到离散 . . . 48
5.1.1 零阶保持（Zero-Order Hold，ZOH） . . . 48
5.1.2 双线性变换（Tustin Transform，也称 Bilinear Transform） . . . 48
5.1.3 极零点匹配法（Matched Pole-Zero Method） . . . 49
5.2 数字滤波器结构：FIR 与 IIR . . . 49
5.2.1 FIR（有限冲激响应，Finite Impulse Response） . . . 49
5.2.2 IIR（无限冲激响应，Infinite Impulse Response） . . . 50
5.2.3 FIR vs. IIR：正面对比 . . . 51
5.2.4 实用选择指南与 Biquad 实现 . . . 51
5.3 定点与浮点运算 . . . 52
5.4 采样率选择 . . . 53
5.5 嵌入式系统实现 . . . 53
5.5.1 定时器与中断配置 . . . 53
5.5.2 计算延迟与抖动 . . . 54
5.5.3 实践中的积分抗饱和 . . . 54

6 现代控制理论 57

6.1 状态空间与传递函数的关系 . . . 57
6.2 系统性能指标 . . . 57
6.2.1 可控性与可观性 . . . 57
6.2.2 闭环系统稳定性 . . . 60
6.3 基于状态空间的控制器设计 . . . 60
6.3.1 贝尔曼方程与动态规划：LQR 和 MPC 的核心引擎 . . . 60
6.3.2 LQR（线性二次调节器） . . . 62
6.3.3 优化与凸性简介 . . . 63
6.3.4 MPC（模型预测控制） . . . 64
6.3.5 TinyMPC：面向嵌入式系统的 Riccati 加速 MPC . . . 65
6.4 基于状态空间的观测器设计 . . . 67

6.4.1 龙伯格观测器（Luenberger Observer） 67
6.4.2 深刻的对偶性：控制器与观测器互为镜像 68
6.4.3 卡尔曼滤波器（Kalman Filter） 69
6.4.4 扩展卡尔曼滤波器（EKF） 70
6.4.5 基于四元数的 EKF：避免万向锁 72
6.4.6 自适应 EKF（Adaptive EKF）：在线自动调整 Q 和 R 73
6.4.7 龙伯格观测器与卡尔曼滤波器：全面对比 76
6.4.8 LQG（线性二次高斯控制器） 76
6.5 轨迹生成（Trajectory Generation）：告诉电机去哪里 81
6.5.1 梯形速度曲线（Trapezoidal Velocity Profile） 81
6.5.2 S 形速度曲线（S-Curve Profile） 81
6.5.3 Bézier 曲线与样条插值 83

7 展望：数据驱动与基于学习的控制 85
7.1 数据驱动控制 85
7.1.1 动机：从“先建模再控制”到“直接从数据控制” 85
7.1.2 Willems 基本引理 85
7.1.3 数据驱动 LQR 85
7.1.4 数据驱动 MPC（DeePC） 85
7.2 强化学习与控制 85
7.2.1 核心概念 86
7.2.2 两种范式 86
7.2.3 Sim-to-Real 迁移 86
7.2.4 成功案例与现状 86
7.3 轨迹优化与最优控制 86
7.3.1 两类求解策略 87
7.3.2 iLQR 与 DDP 87
7.4 神经网络与控制的融合 87
7.4.1 神经网络作为控制器组件 87
7.4.2 安全约束与可验证性 87
7.5 方法全景对比 87
7.6 给读者的建议 88

8 附录：即插即用 C++ 模块 89
8.1 低通滤波器（Low-Pass Filter） 89
8.2 PID 控制器（PID Controller） 90
8.3 给定值斜坡（Setpoint Ramp，LPF 软启动） 92
8.4 互补滤波器（Complementary Filter，IMU 倾斜估计） 93
8.5 轻量矩阵库（Lightweight Matrix Library） 94
8.6 卡尔曼滤波器（Kalman Filter） 96
8.7 LQR 增益（离线计算辅助工具） 98
8.8 串级 PID（Cascaded PID） 99
8.9 高级 PID（积分分离 + PI-D + 2 自由度） 101

8.10 6 自由度 IMU 扩展卡尔曼滤波器（EKF for 6-DOF IMU，姿态估计） 103
8.11 TinyMPC 求解器（TinyMPC Solver） 106
8.12 三次贝塞尔曲线轨迹（Cubic Bézier Trajectory） 110
8.13 模块依赖关系图（Module Dependency Map） 112

1 引言

这份文档**不是**教材。教材当然好——严谨、系统，催眠效果更是一流，凌晨两点百试百灵。这是一份学习笔记，记笔记的人学控制走的是”先把机器人搞坏，再翻书找原因”的野路子。

这份文档的目标很简单：给你够用的理论功底，让你能在真实硬件上**设计、调试、排查控制器**。数学该讲还是要讲，但每讲一个概念我们都会追问一句：”机器人上场的时候，这玩意到底有什么用？”

这份笔记写给谁看？

大一到大三的本科生。前置知识：学过（或正在学）微积分和基础线性代数。只要你知道导数是什么、矩阵乘法不会让你当场崩溃，就够了。

这份笔记的结构有什么特别之处？

目前市面上几乎所有控制相关教材都没有类似的内容编排。

先从**数字信号处理（DSP）**讲起——控制任何东西之前，你得先能从传感器拿到干净数据。然后学如何用数学**描述系统**，说白了就是”把你的电机到底在干什么用公式写下来”。接下来是**经典控制理论**，PID 的主场——竞赛机器人的看家本领。之后是**离散化与工程实现**，在优雅的数学和跑在固定时钟频率单片机上的残酷现实之间搭一座桥。最后走进**现代控制理论**，学会那些把”能用”的机器人变成”好用”的机器人的工具：LQR、MPC 和卡尔曼滤波器。

开始吧。

2 数字信号处理

你可能纳闷：”说好的控制理论呢，怎么先讲信号处理？”问得好。一句话：**垃圾进，垃圾出**。

控制器再牛，吃进去的数据是垃圾也白搭。你的 IMU（惯性测量单元）读数在那里乱蹦，再怎么调 PID 参数都是白费力气。所以在学控制之前，先学怎么把信号清理干净。

2.1 理解频域

日常中我们习惯在**时域（time domain）**里看信号——电压随时间变化，位置随时间变化，很直觉。但换一个视角来看，很多问题会简单得多。这个视角就是**频域（frequency domain）**。

核心思想一句话就能说清：*任何信号都可以拆成一堆不同频率的正弦波叠在一起*。你那嘈杂的陀螺仪读数？其实就是一个干净的低频旋转信号，上面糊了一层高频噪声。只要分辨出哪些频率是”有用信号”、哪些是”噪声”，就能精准地把噪声切掉。

打个比方：音乐播放器上的均衡器（EQ）。低音、中音、高音全搅在时域波形里，但均衡器能让你单独拉高或压低某个频段。控制系统里的滤波器干的就是这事。

一条关键直觉：系统里慢悠悠的、有意义的变化（比如底盘转向）发生在低频；快速的、没意义的毛刺（比如电气噪声、振动）发生在高频。频域让我们能把它们分开。

2.2 拉普拉斯变换与傅里叶变换

这两种变换是你从时域进入频域的通行证。它们密切相关，但用途略有不同。

2.2.1 拉普拉斯变换

拉普拉斯变换（Laplace Transform）将时域函数 $f(t)$ 转换为复变量 s 的函数：

$$F(s) = \mathcal{L}\{f(t)\} = \int_0^\infty f(t)e^{-st}\,dt \tag{1}$$

其中 $s = \sigma + j\omega$ 是复数。实部 σ 描述指数增长或衰减，虚部 $j\omega$ 描述振荡。

为什么要学它？因为拉普拉斯变换能把微分方程”降级”成代数方程。与其解

$$m\ddot{x} + b\dot{x} + kx = F(t),$$

（需要微积分），不如解

$$(ms^2 + bs + k)X(s) = F(s),$$

（纯代数）。导数变成乘以 s，积分变成除以 s，一下子就轻松了。

你每天都会用到的最重要的拉普拉斯变换对：

时域	**s 域**	**出现场合**
1（阶跃输入）	$\frac{1}{s}$	指令一个新设定值
e^{-at}	$\frac{1}{s+a}$	一阶系统响应
$\sin(\omega t)$	$\frac{\omega}{s^2+\omega^2}$	振荡、振动
$\dot{f}(t)$	$sF(s) - f(0)$	控制器中的微分
$\int f(t)\,dt$	$\frac{F(s)}{s}$	控制器中的积分

实用建议：积分公式不用死记。你只需要记住一件事：s 就是" 求导"，$1/s$ 就是" 积分"。光这一条，就能撑过 80% 的控制理论。

e^{-st} **到底在干什么？**核函数 $e^{-st} = e^{-\sigma t}e^{-j\omega t}$ 相当于一根" 探针"，同时问两个问题：$e^{-\sigma t}$——" 信号是不是以速率 σ 在增长或衰减？"；$e^{-j\omega t}$——" 频率 ω 的成分有多少？" 把 $f(t)\cdot e^{-st}$ 在全时间上积分，就是在算信号和各种指数正弦波之间的" 相关性"。结果 $F(s)$ 是一张覆盖整个复平面的地图，精确标出了信号中每种指数振荡模态有多少成分。

常用拉普拉斯变换对——推导过程：

单位阶跃（Unit Step）$u(t) = 1$，$t \geq 0$：

$$\mathcal{L}\{1\} = \int_0^\infty e^{-st}\,dt = \left[-\frac{e^{-st}}{s}\right]_0^\infty = \frac{1}{s}$$

指数衰减（Exponential Decay）e^{-at}：

$$\mathcal{L}\{e^{-at}\} = \int_0^\infty e^{-at}e^{-st}\,dt = \int_0^\infty e^{-(s+a)t}\,dt = \frac{1}{s+a}$$

这就是为什么极点在 $s = -a$ 的一阶系统，其时域响应是 e^{-at}——极点位置就是衰减速率。

斜坡（Ramp）$t \cdot u(t)$：

$$\mathcal{L}\{t\} = \frac{1}{s^2}$$

注意：时域乘以 t 对应于对 s 求导后取反号：$\mathcal{L}\{t \cdot f(t)\} = -\frac{d}{ds}F(s)$。由于 $\mathcal{L}\{1\} = 1/s$，所以 $\mathcal{L}\{t\} = -\frac{d}{ds}\frac{1}{s} = \frac{1}{s^2}$。

2.2.2 傅里叶变换

傅里叶变换（Fourier Transform）是拉普拉斯变换的" 安分" 版本——让 $s = j\omega$（也就是把实部 σ 设为 0）就得到了：

$$F(j\omega) = \int_{-\infty}^{\infty} f(t)e^{-j\omega t}\,dt \tag{2}$$

拉普拉斯变换覆盖整个复平面（连不稳定的发散信号也能应付），而傅里叶变换只活在虚轴上。它回答的是一个单纯的问题：" 这个信号里每个频率各有多少？"

什么时候用哪个：

- **拉普拉斯变换**：用于分析和设计控制系统（传递函数、稳定性）。
- **傅里叶变换**：用于分析信号（噪声、振动、频率成分）。

实际工程中，数字系统用的是**离散傅里叶变换（DFT）**，通过**快速傅里叶变换（FFT）**算法高效计算。想知道传感器数据里藏着什么频率？扔进 FFT 就行。MATLAB 的 `fft()` 或 Python 的 `numpy.fft.fft()` 一行搞定。

2.2.3 频谱

信号经过傅里叶变换后，得到的就是它的**频谱（Frequency Spectrum）**——一幅标出了各频率分量大小（和相位）的图。

看频谱就像给信号拍 X 光片：

- 50 Hz 处有个高峰？那大概是工频干扰（电网噪声）。

- 10 Hz 以下有个宽包？那可能是你真正感兴趣的信号。
- 所有频率上都有随机的模糊？那是白噪声（White Noise）。

实际案例：假设你的云台编码器给出一个嘈杂的速度读数。你记录了 1 秒数据，计算 FFT，看到 3 Hz 处有个清晰的峰（云台正在以 3 Hz 跟踪目标），加上 50 Hz 以上一片噪声乱麻。现在你心里有数了：设计一个截止频率在 20–30 Hz 的低通滤波器，信号就会变干净。

频谱包含两部分：

- **幅度谱**（Magnitude Spectrum）$|F(j\omega)|$：每个频率分量的强度。
- **相位谱**（Phase Spectrum）$\angle F(j\omega)$：每个频率分量的时间偏移。

在滤波器设计中，我们主要关心幅度谱。相位在后面分析控制回路稳定性（相位裕度）时才变得重要。

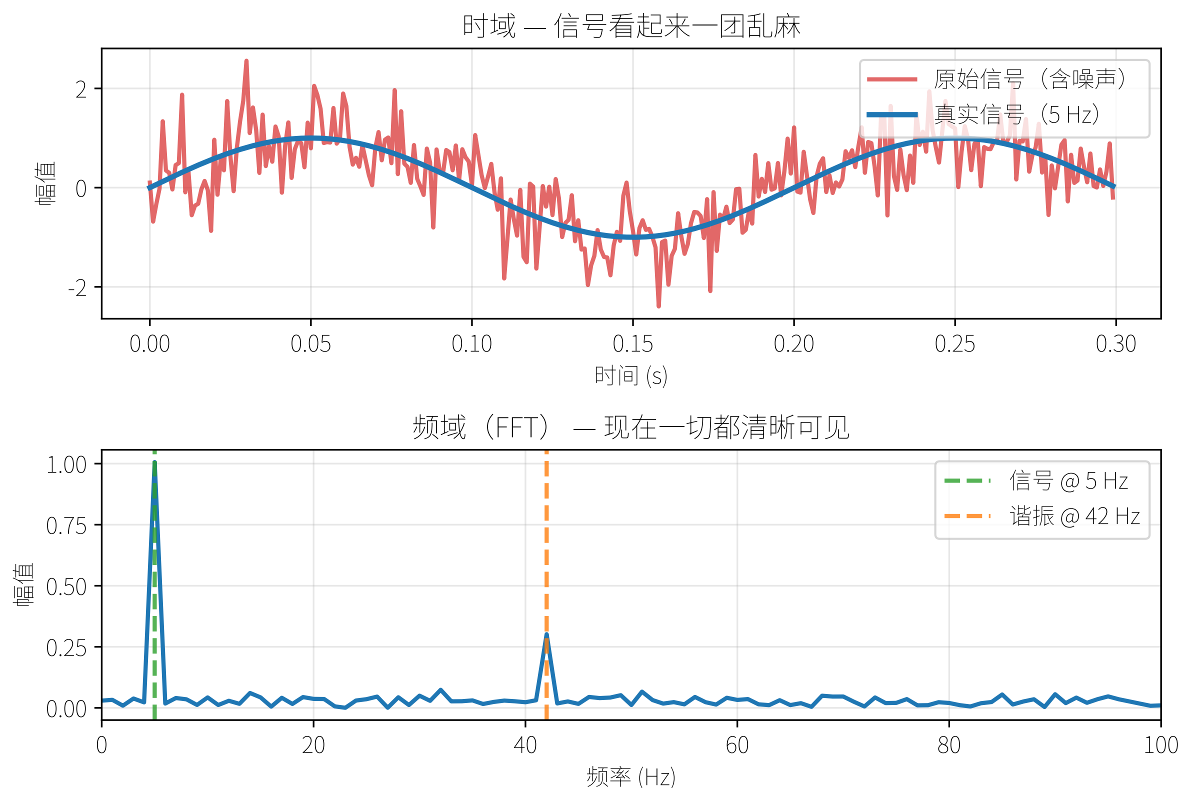

图 1: 上图：时域中的噪声信号看起来毫无希望。下图：FFT 立刻揭示出隐藏在噪声中的 5 Hz 信号和 42 Hz 谐振。

案例分析：用 FFT 排查底盘振动

竞赛机器人的经典翻车现场：麦克纳姆轮底盘低速好好的，一到高速就振动，整个车架嗡嗡响。大家怀疑螺丝松了，拧紧所有螺丝。还是振。

诊断方法：在出问题的速度下行驶时，记录 5 秒 IMU 加速度计数据。用 Python 跑一遍 FFT：

```
import numpy as np
```

```
freqs = np.fft.rfftfreq(len(data), d=1.0/sample_rate)
spectrum = np.abs(np.fft.rfft(data))
plt.plot(freqs, spectrum)
```

你看到 42 Hz 处有个巨大的尖峰。有意思。你算了一下：高速行驶时，麦轮辊子接触地面的频率是……大约 42 Hz。辊子接触频率恰好与车架结构谐振频率吻合。

解决方案：

- 在控制回路中加一个 42 Hz 的**陷波滤波器（Notch Filter，带阻滤波器）**，防止控制器激励谐振。
- 加物理阻尼（橡胶减振垫），把谐振频率移开。
- 加固车架，把谐振频率推到工作频率范围以上。

教训：不用 FFT 的话，你可能会花好几个小时拧螺丝。用了 FFT，10 秒内根本原因一目了然。**频域是一种调试神器**。只要有什么东西在振荡而你不知道为什么，记录数据，看频谱。

2.3 基于频域的滤波器设计

既然我们能在频域中”看见”信号，就可以设计滤波器——保留我们想要的频率，剔除不想要的。

滤波器本质上是一个在不同频率下增益不同的系统。我们用滤波器的**频率响应（Frequency Response）**来描述它——一张增益对频率的图（称为**波特幅度图，Bode Magnitude Plot**）。

2.3.1 低通滤波器

机器人中最常用的滤波器。通过低频，衰减高频。

一阶低通滤波器（Low-Pass Filter，LPF）：

$$H(s) = \frac{\omega_c}{s + \omega_c} \tag{3}$$

其中 $\omega_c = 2\pi f_c$ 是截止角频率（rad/s）。低于 ω_c 的频率增益约为 1（信号通过）；高于 ω_c 的频率以 -20 dB/十倍频的斜率衰减（信号被压制）。

在时域中，这等价于一个指数移动平均：

$$y[n] = \alpha \cdot x[n] + (1 - \alpha) \cdot y[n-1] \tag{4}$$

其中 $\alpha = \frac{T_s}{T_s + \frac{1}{2\pi f_c}}$，$T_s$ 是采样周期。

适用场景：平滑带噪声的传感器读数（陀螺仪、加速度计、电机电流）。这是你 90% 的情况下会用到的滤波器。

注意：低通滤波器会引入**相位滞后（Phase Lag）**——输出相对于输入有延迟。滤波越激进（截止频率越低），引入的滞后越大。在快速控制回路中，这种滞后可能使系统失稳。平滑信号和快速响应之间永远存在权衡。

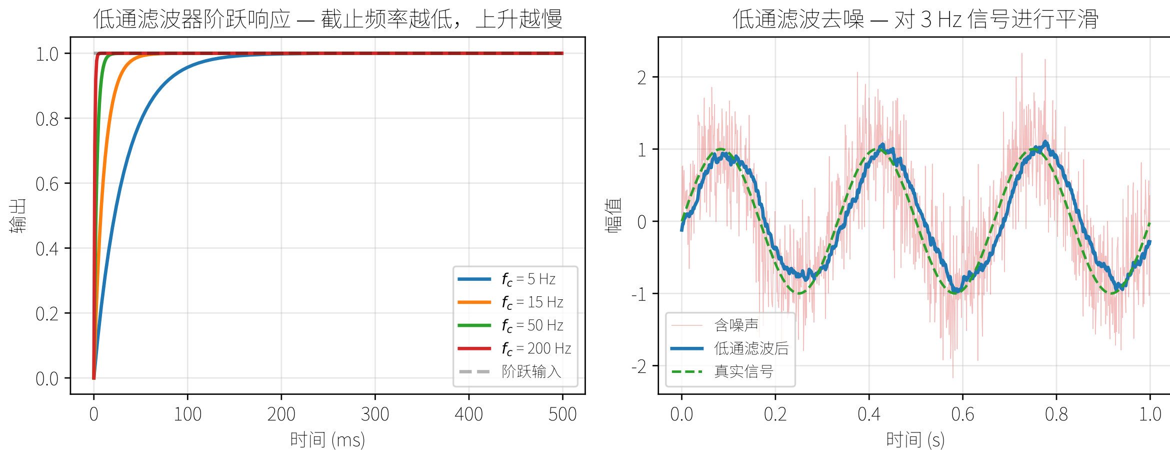

图 2: 左图：LPF 阶跃响应——截止频率越低，上升越慢，但输出越平滑。右图：对含噪声的 3 Hz 信号滤波——LPF 恢复出真实信号，同时去除了高频噪声。

案例分析：为什么低通滤波器能免费送你”软启动”

假设你命令电机从 0 瞬间跳到 5000 RPM（阶跃输入）。没有任何滤波器的话，设定值突变——电机驱动器看到一个突如其来的全电压指令，齿轮猛地咬合，底盘猛地一窜，你的队友把咖啡洒了一地。

现在在设定值上串一个一阶低通滤波器：$r_{\text{filtered}}(s) = \frac{\omega_c}{s+\omega_c} \cdot R(s)$。会发生什么？

s 域中的阶跃输入是 $R(s) = \frac{5000}{s}$。经过滤波器后：

$$R_{\text{filtered}}(s) = \frac{5000\,\omega_c}{s(s+\omega_c)}$$

做拉普拉斯逆变换：

$$r_{\text{filtered}}(t) = 5000\left(1 - e^{-\omega_c t}\right)$$

阶跃变成了一个**平滑的指数斜坡**。时间常数是 $\tau = 1/\omega_c$。经过 3τ，输出达到目标的 95%；经过 5τ，达到 99%。

这对你的机器人意味着什么：

- **机械保护：**齿轮和皮带有扭矩上限。阶跃指令可能崩掉齿轮或断掉皮带。LPF 限制了变化速率，让力保持在安全范围内。
- **限制涌入电流：**突然施加电压会引起巨大的涌入电流（反电动势为零时 $I = V/R$）。LPF 让电压缓慢爬升，电流保持在可控范围。
- **可调节的激进程度：**想要更快的斜坡？增大 ω_c（提高截止频率）。想要更柔和的斜坡？减小 ω_c。一个参数控制全部行为。

与线性斜坡的对比：你也可以让设定值线性爬升（比如每秒增加 1000 RPM）。但 LPF 更好，因为它能平滑任何设定值的突变——不只是初始阶跃，运行过程中的突然变化也一样。这是一个”设好就不用管”的方案。

代码极其简单：

```
setpoint_filtered += alpha * (setpoint_raw - setpoint_filtered);
```

一行代码。没有查找表，没有状态机。这就是 LPF 作为软启动机制的魅力所在。

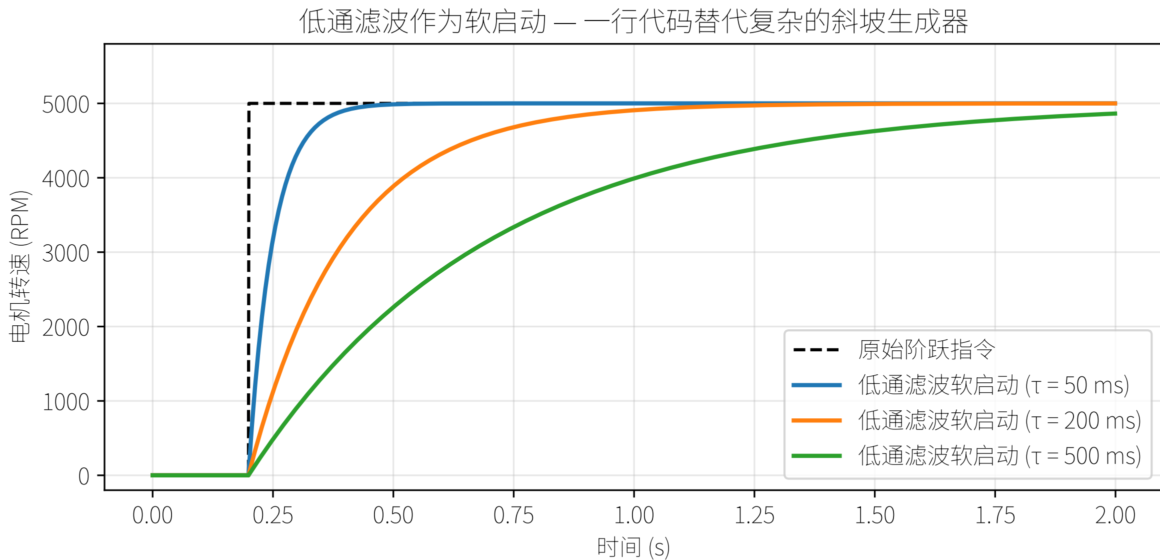

图 3: LPF 软启动：单个参数 τ 控制电机加速的激进程度。阶跃指令变成了平滑的指数曲线。

2.3.2 带通滤波器

带通滤波器（Band-Pass Filter）通过特定频率范围，衰减其他所有频率。可以看作低通和高通滤波器的组合。

$$H(s) = \frac{\frac{\omega_0}{Q} s}{s^2 + \frac{\omega_0}{Q} s + \omega_0^2} \tag{5}$$

其中 ω_0 是中心频率，Q 是品质因数（Q 越高，通带越窄）。

适用场景： 从信号中提取特定频率分量。例如，如果你知道目标在某个特定频率下振荡，想跟踪它同时剔除其他一切。

在机器人中，带通滤波器没有低通滤波器常见，但在振动分析和谐振检测方面很有用。

2.3.3 高通滤波器

高通滤波器（High-Pass Filter）通过高频，衰减低频。它是低通滤波器的反面。

一阶高通滤波器：

$$H(s) = \frac{s}{s + \omega_c} \tag{6}$$

适用场景： 去除信号中的直流偏置或缓慢漂移。例如，如果你的加速度计有缓慢漂移的偏置，高通滤波器可以去除它，同时保留快速加速度数据。

实用提示： 互补滤波器（Complementary Filter，IMU 传感器融合中常用）其实就是对一个传感器做低通、对另一个传感器做高通，两者截止频率匹配使增益之和为 1。加速度计低通处理（信任它的慢速/静态姿态信息），陀螺仪高通处理（信任它的快速旋转数据）。简单、优雅，效果出奇地好。

2.3.4 滤波器类型选择与参数整定

选对滤波器，就是回答三个问题：

1. **我想保留哪些频率？**——决定滤波器类型（低通、高通、带通）。
2. **边界在哪里？**——决定截止频率 f_c。
3. **过渡要多陡？**——决定滤波器阶数。

经验法则：

- 从一阶低通滤波器开始。简单、稳定，大多数场合够用。
- 截止频率设为你感兴趣信号带宽的 5–10 倍。太低会损失信号，太高会保留噪声。
- 如果一阶不够，试试二阶（通带平坦用巴特沃斯（Butterworth），滚降更陡用切比雪夫（Chebyshev））。
- **绝对不要**在没看过实际数据频谱的情况下拍脑袋设截止频率。靠猜截止频率是挫败感的源泉。
- 拿不定主意时，把滤波后的输出和原始信号画在一起，检查结果是否符合物理直觉。

2.4 Z 变换

我们之前用拉普拉斯变换讨论的一切，都活在连续世界里。但你的单片机是数字的——它处理的是离散采样，而不是连续信号。**Z 变换（Z-Transform）**是拉普拉斯变换在离散时间中的对应物。

离散序列 $x[n]$ 的 Z 变换是：

$$X(z) = \sum_{n=0}^{\infty} x[n] z^{-n} \tag{7}$$

关键对应关系：

连续域（s 域）	**离散域（z 域）**
s（导数）	$\frac{z-1}{T_s}$（前向差分）
$\frac{1}{s}$（积分）	$\frac{T_s}{z-1}$（累加器）
稳定条件 $\mathrm{Re}(s) < 0$	稳定条件 $\lvert z \rvert < 1$
e^{-aT_s} 在 s 域	z^{-1} 表示一个采样延迟

为什么要关心它？在单片机上实现滤波器或控制器的那一刻，你就已经在 z 域里了，不管你自己有没有意识到。那行简单的代码：

```
y = alpha * x + (1 - alpha) * y_prev;
```

实际上是一个 z 域传递函数：

$$H(z) = \frac{\alpha}{1 - (1-\alpha) z^{-1}}$$

理解 Z 变换能帮你：

- 把连续时间滤波器/控制器转换成代码（见第 5 节）。
- 分析你的离散时间系统的稳定性。

- 理解为什么你的控制器在不同采样频率下表现不同。

现在，只要记住一件事：z^{-1} **表示”上一个时刻”**。这是 Z 变换中最重要的一个认知。

常用 Z 变换对：

离散信号	z **域**	**出现场合**
$\delta[n]$（单位冲激）	1	初始冲击
$u[n]$（单位阶跃）	$\frac{z}{z-1}$	阶跃指令
$a^n u[n]$（等比衰减）	$\frac{z}{z-a}$	离散一阶系统响应
$x[n-1]$（一步延迟）	$z^{-1}X(z)$	代码中的上一个采样值
$x[n]-x[n-1]$（一阶差分）	$(1-z^{-1})X(z)$	离散导数
$\sum_{k=0}^{n} x[k]$（累加和）	$\frac{1}{1-z^{-1}}X(z)$	离散积分

s **与** z **之间的联系：**精确的映射关系是 $z = e^{sT_s}$。连续极点 $s = -a$ 映射到离散极点 $z = e^{-aT_s}$。这意味着：

- 稳定的连续极点（$\mathrm{Re}(s) < 0$）映射到 $|z| < 1$（单位圆内）。稳定性得以保持。
- 更快的连续极点（s 更负）映射到更小的 $|z|$（更靠近原点）。极快的极点映射到 $z \approx 0$ 附近。
- 振荡的连续极点（$s = \sigma \pm j\omega$）映射到 $z = e^{\sigma T_s} e^{\pm j\omega T_s}$，在单位圆内（稳定）或单位圆外（不稳定）盘旋。

读懂 z **域传递函数：**任何 z 域传递函数都可以直接转换成代码，只要把 z^{-1} 替换成”上一个值”。例如：

$$H(z) = \frac{b_0 + b_1 z^{-1}}{1 + a_1 z^{-1}}$$

对应代码：

```
y = b0 * x + b1 * x_prev - a1 * y_prev;
x_prev = x;  y_prev = y;
```

这被称为**差分方程（Difference Equation）**，是任何数字滤波器或控制器的最终形式。

3 系统描述

要控制一样东西，得先搞明白它在干什么。系统（System）就是接受输入、产生输出的任何东西。直流有刷电机吃电压、吐角速度；底盘吃轮速、吐运动轨迹；云台吃力矩、吐角度。

控制工程的第一步，是把这些输入-输出关系写成数学式子。这叫**建模（Modeling）**——整个流程中最重要的一步，也是最容易被轻视的一步。

一句忠告：平庸的控制器配上好模型，永远比完美的控制器配上烂模型强。花时间把你的系统吃透，不亏。

系统分两大类：**SISO**（单输入单输出）和 **MIMO**（多输入多输出）。一个电机是 SISO；四轮全向底盘是 MIMO。我们从简单的 SISO 入手，之后再推广。

3.1 系统的数学模型

描述系统有三种主要的数学手段，各有各的长处。

3.1.1 微分方程（Differential Equations）

最基础的描述方式。我们利用物理定律（牛顿定律、基尔霍夫定律等）写出输入、输出及其各阶导数之间的方程。

示例：直流有刷电机

直流有刷电机是机器人竞赛中最常见的执行器。以下是其简化模型：

电气方程（基尔霍夫电压定律）：

$$V = L\frac{di}{dt} + Ri + K_e\omega \tag{8}$$

机械方程（旋转运动的牛顿第二定律）：

$$J\frac{d\omega}{dt} = K_t i - b\omega - \tau_{\text{load}} \tag{9}$$

其中：

- V：输入电压，i：电枢电流
- L：电感，R：电阻
- K_e：反电动势常数，K_t：力矩常数
- J：转动惯量，b：粘性摩擦系数
- ω：角速度，τ_{load}：负载力矩

两个耦合的一阶微分方程。实际中，电气动态（$L\frac{di}{dt}$）比机械动态快得多，所以常令 $L \approx 0$ 来简化，得到：

$$J\frac{d\omega}{dt} = \frac{K_t}{R}(V - K_e\omega) - b\omega - \tau_{\text{load}} \tag{10}$$

一个把输入电压 V 和输出转速 ω 联系起来的一阶常微分方程——有了它就可以分析和做控制了。

3.1.2 传递函数（Transfer Functions）

对微分方程取拉普拉斯变换（假设零初始条件），即可得到**传递函数（Transfer Function）**：输出与输入在 s 域中的比值。

对于简化后的直流有刷电机：

$$G(s) = \frac{\Omega(s)}{V(s)} = \frac{K_t/R}{Js + (b + K_t K_e/R)} \tag{11}$$

可改写为标准一阶形式：

$$G(s) = \frac{K}{\tau s + 1} \tag{12}$$

其中 K 是直流增益（每伏特对应的稳态转速），τ 是时间常数（电机达到稳态的速度）。

传递函数的优点：

- 形式紧凑——一个分式描述整个系统。
- 级联系统只需相乘：$G_{\text{total}}(s) = G_1(s) \cdot G_2(s)$。
- 稳定性分析直观：检查分母的根（极点）即可。
- 令 $s = j\omega$ 即可得到频率响应。

传递函数的局限：

- 仅适用于线性时不变（LTI）系统。
- 假设零初始条件。
- 天然是 SISO 的——推广到 MIMO 需要传递函数矩阵，处理起来很繁琐。
- 隐藏了系统的内部状态。

3.1.3 状态空间表示（State-Space Representation）

状态空间表示将系统描述为一组矩阵形式的一阶微分方程：

$$\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u} \quad \text{（状态方程）} \tag{13}$$

$$\mathbf{y} = C\mathbf{x} + D\mathbf{u} \quad \text{（输出方程）} \tag{14}$$

其中：

- $\mathbf{x}$：状态向量（描述系统当前状况的内部变量）
- $\mathbf{u}$：输入向量
- $\mathbf{y}$：输出向量
- A：系统矩阵，B：输入矩阵，C：输出矩阵，D：直通矩阵

各矩阵的物理含义：

- A（系统/动态矩阵）：描述状态在无输入时如何自发演化。若断开电机（$\mathbf{u} = 0$），系统按 $\dot{\mathbf{x}} = A\mathbf{x}$ 演化。A 的特征值决定了系统的自然行为：稳定衰减、振荡或发散。元素 A_{ij} 表示状态 j 对状态 i 变化率的影响。
- B（输入矩阵）：描述输入如何影响各状态。元素 B_{ij} 表示输入 j 对状态 i 变化率的影响。若 B 的某一行全为零，则该状态无法被任何输入直接驱动——只能通过 A 中的耦合间接影响。
- C（输出矩阵）：描述哪些状态可被测量。元素 C_{ij} 表示状态 j 对测量量 i 的贡献。若 $C = [1\ 0\ \ldots\ 0]$，则只测量第一个状态。

- D（直通矩阵）：描述输入绕过动态环节直接影响输出的路径。大多数物理系统中 $D=0$，因为输入必须经过动态环节才能影响输出。当存在从输入到输出的直接代数通路时（例如纯电阻分压器），$D \neq 0$。

示例：直流有刷电机的状态空间表示
选取状态 $\mathbf{x} = \begin{bmatrix} \omega \\ i \end{bmatrix}$（角速度和电枢电流）：

$$\begin{bmatrix} \dot{\omega} \\ \dot{i} \end{bmatrix} = \begin{bmatrix} -b/J & K_t/J \\ -K_e/L & -R/L \end{bmatrix} \begin{bmatrix} \omega \\ i \end{bmatrix} + \begin{bmatrix} 0 \\ 1/L \end{bmatrix} V \tag{15}$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} \omega \\ i \end{bmatrix} \tag{16}$$

状态空间的强大之处：

- 天然处理 MIMO 系统——只需把矩阵扩大即可。
- 暴露内部状态，便于我们进行状态估计（观测器）和状态反馈控制。
- 与现代控制技术（LQR、MPC、卡尔曼滤波）无缝衔接。
- 易于推广到非线性系统（将 $A\mathbf{x}$ 替换为 $f(\mathbf{x})$）。

权衡：状态空间更强大，但也更抽象；传递函数对 SISO 系统直觉更好。好的工程师两手都得硬。

3.2 基于传递函数的系统分析

拿到传递函数 $G(s)$，就能从中挖出大量信息。
极点与零点。将 $G(s) = \frac{N(s)}{D(s)}$，$N(s)$ 的根称为**零点（Zeros）**，$D(s)$ 的根称为**极点（Poles）**。

- **极点决定稳定性**。若所有极点的实部均为负（即位于复平面左半部分），系统稳定。若任何极点实部为正，系统不稳定，将发散。极点在虚轴上意味着系统临界稳定（持续振荡）。
- **极点决定响应速度**。极点越靠左，动态越快。*主导极点*（最靠近虚轴的极点）决定系统的整体响应速度。
- **复数极点意味着振荡**。一对共轭复数极点 $s = -\sigma \pm j\omega_d$ 产生振荡行为。实部 σ 决定振荡衰减的速度，虚部 ω_d 决定振荡频率。
- **零点塑造响应**。零点可以抵消极点、改变瞬态响应，或导致非最小相位行为（系统先朝错误方向运动）。

快速判断系统阶数的技巧：数积分次数。在深入数学之前，先介绍一个直观的判断方法。系统的阶数等于从输入变量到输出变量在物理意义上需要经过的积分次数。以电压控制的直流有刷电机为例：

- 电压 → 电流（通过电感的一次积分：$L\frac{di}{dt} = V - \ldots$）→ 第一个积分器。
- 电流 → 角速度（通过惯量的一次积分：$J\frac{d\omega}{dt} = K_t i - \ldots$）→ 第二个积分器。

从电压到角速度经过两次积分 ⇒ 二阶系统。若令 $L \approx 0$（电流瞬时响应），则去掉一次积分，变为一阶系统。若输出是位置而非速度，则再增加一次积分（$\theta = \int \omega \, dt$），系统变为三阶（或在 $L \approx 0$ 简化下为二阶）。这一计数技巧适用于任何物理系统：弹簧、热系统、液位系统——数一数独立的储能元件（电感、电容、质量、弹簧），得到的就是系统阶数。

标准二阶系统。许多物理系统可近似为：

$$G(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2} \tag{17}$$

其中 ω_n 是自然频率（Natural Frequency），ζ 是阻尼比（Damping Ratio）。阻尼比决定了瞬态响应的全部特征：

ζ	**行为**
$\zeta > 1$	过阻尼：响应慢，无振荡
$\zeta = 1$	临界阻尼：最快的无振荡响应
$0 < \zeta < 1$	欠阻尼：振荡并衰减
$\zeta = 0$	无阻尼：持续振荡
$\zeta < 0$	不稳定：振荡发散

对于大多数机器人应用，ζ 取 0.5 到 0.8 之间最为合适——响应快，超调可接受。

欠阻尼情形（$0 < \zeta < 1$）的常用公式：

阶跃响应为：

$$y(t) = 1 - \frac{e^{-\zeta\omega_n t}}{\sqrt{1-\zeta^2}} \sin(\omega_d t + \phi), \quad \omega_d = \omega_n\sqrt{1-\zeta^2}, \quad \phi = \arccos(\zeta) \tag{18}$$

其中 ω_d 是**阻尼自然频率（Damped Natural Frequency）**——系统实际振荡的频率（始终小于 ω_n）。

由此可推导出各性能指标的解析表达式：

$$\text{超调量:} \quad M_p = e^{-\pi\zeta/\sqrt{1-\zeta^2}} \times 100\% \tag{19}$$

$$\text{峰值时间:} \quad t_p = \frac{\pi}{\omega_d} = \frac{\pi}{\omega_n\sqrt{1-\zeta^2}} \tag{20}$$

$$\text{调节时间（2\%）:} \quad t_s \approx \frac{4}{\zeta\omega_n} \tag{21}$$

$$\text{上升时间（近似）:} \quad t_r \approx \frac{1.8}{\omega_n} \tag{22}$$

这些公式告诉你什么：

- 超调量只取决于 ζ，与 ω_n 无关。更快的系统（ω_n 更大）超调百分比相同——只是发生得更快。
- 调节时间取决于乘积 $\zeta\omega_n$——即极点的实部。更快的调节要求极点在复平面上更靠左。
- 存在直接权衡：ζ 越小，上升时间越快但超调越大；ζ 越大，超调越小但响应越慢。
- 取 $\zeta = 0.707$（即 $1/\sqrt{2}$）时：超调量 $\approx 4.3\%$，通常被称为速度与平稳性之间的“最优”阻尼。

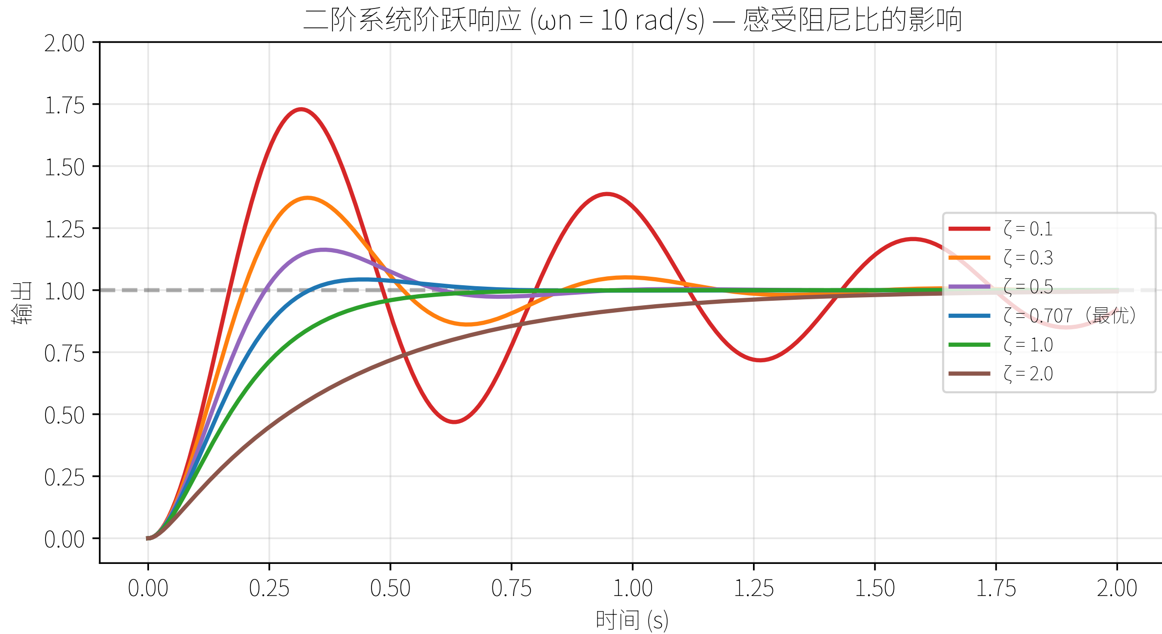

图 4: 不同阻尼比下的二阶系统阶跃响应。注意 $\zeta = 0.707$（蓝色曲线）在无明显超调的前提下上升最快——这是大多数控制应用的“甜蜜点”。

案例分析：弹簧-质量-阻尼器——一个你能亲身感受的系统

在钻进更抽象的数学之前，先用一个你坐过的二阶系统来建立直觉：汽车悬挂。

汽车过坑，就是教科书级别的弹簧-质量-阻尼器：车身（质量 m）在弹簧（刚度 k）上弹，减震器（阻尼系数 b）负责耗散能量。

$$m\ddot{x} + b\dot{x} + kx = F(t)$$

传递函数：$G(s) = \frac{1}{ms^2+bs+k} = \frac{1/m}{s^2+\frac{b}{m}s+\frac{k}{m}}$

自然频率：$\omega_n = \sqrt{k/m}$。 阻尼比：$\zeta = \frac{b}{2\sqrt{km}}$。

现在用身体感受数学：

- **没有减震器**（$\zeta = 0$）：过坑后永远弹跳不停。这是蹦床，不是汽车。无阻尼。
- **减震器磨损**（$\zeta = 0.2$）：过坑后弹跳 5–6 次才平稳。乘客晕车。欠阻尼。
- **良好减震器**（$\zeta = 0.5$–0.7）：过坑后轻微弹跳一两次，迅速稳定。舒适。这是甜蜜点。
- **越野卡车减震器**（$\zeta = 1$）：过坑后缓慢回到中位，完全不弹跳。安全但偏硬。临界阻尼。
- **减震器灌满混凝土**（$\zeta = 3$）：过坑后极慢才能回位。每一颗小石头都能感觉到。过阻尼。

跟你的机器人有什么关系？云台、底盘、机械臂——都有自然频率和阻尼。云台收到阶跃指令后振荡？欠阻尼，ζ 太低。响应迟钝？过阻尼，ζ 太高。PID 控制器（第 4 节）的本质，就是给系统外挂虚拟弹簧（P）、虚拟阻尼器（D）和记忆（I），从而改变等效的 ζ 和 ω_n。

为什么 ω_n 很重要：云台又重、电机又弱，ω_n 天生就低——物理上它就快不起来。PID 参数调到天上去也突破不了自然频率的天花板。要更快？换更大的电机（提高 k），或者减轻负载（减小 m）。硬件设计和控制设计从来就是一件事。

3.3 基于状态空间的系统分析

对于状态空间模型 $\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}$，系统行为由 A **的特征值**决定。

这里有个漂亮的对应关系：*A 的特征值就是传递函数的极点*。前面关于极点说的那些分析，在这里全部成立。特征值分析的好处是天然适用于 MIMO 系统，传递函数在那里就显得笨重了。

矩阵指数（Matrix Exponential）。状态空间系统的自由响应（无输入）为：

$$\mathbf{x}(t) = e^{At}\mathbf{x}(0) \tag{23}$$

矩阵指数 e^{At} 编码了所有动态信息。若对 A 进行对角化（在可能的情况下），得到：

$$e^{At} = P \begin{bmatrix} e^{\lambda_1 t} & & \\ & \ddots & \\ & & e^{\lambda_n t} \end{bmatrix} P^{-1} \tag{24}$$

每个特征值 λ_i 贡献一个指数模态。实部为负的特征值衰减，实部为正的发散，复数特征值振荡。这是状态空间语言中"极点决定一切"的表达方式。

3.4 电机建模：最常见的被控对象

控制器离不开被控对象（Plant）模型。比赛机器人里，95% 的被控对象都是电机——驱动轮子、旋转飞轮、控制炮塔，全靠它。把电机模型搞明白，是学任何控制算法之前最值得投资的技能。

直流有刷电机模型

直流有刷电机（DC Brushed Motor）由两个耦合子系统组成：电气部分和机械部分。

电气动态（电枢回路的 KVL）：

$$V = L\frac{di}{dt} + Ri + K_e\omega \tag{25}$$

机械动态（转子的牛顿第二定律）：

$$J\frac{d\omega}{dt} = K_t i - b\omega - \tau_{\text{load}} \tag{26}$$

其中 V 为输入电压，i 为电枢电流，ω 为角速度，L 为电感，R 为电阻，K_e 为反电动势常数，K_t 为力矩常数，J 为转子转动惯量，b 为粘性摩擦系数，τ_{load} 为外部负载力矩。

取拉普拉斯变换（令 $\tau_{\text{load}} = 0$）并消去 $I(s)$，得到**完整的二阶传递函数**：

$$\frac{\Omega(s)}{V(s)} = \frac{K_t}{JLs^2 + (JR + bL)s + (bR + K_tK_e)} \tag{27}$$

一阶简化。对于机器人竞赛中使用的大多数电机，电气时间常数 $\tau_e = L/R$ 在毫秒量级——远快于任何机械响应。令 $L \approx 0$，模型简化为更易处理的一阶系统：

$$\frac{\Omega(s)}{V(s)} = \frac{K_m}{\tau_m s + 1} \tag{28}$$

其中**电机增益（Motor Gain）**和**机械时间常数（Mechanical Time Constant）**为：

$$K_m = \frac{K_t}{bR + K_tK_e}, \qquad \tau_m = \frac{JR}{bR + K_tK_e} \tag{29}$$

这个一阶模型是你进行 PID 调参、前馈设计以及大多数实际工程工作时所用的模型。

深入理解时间常数 τ

时间常数 τ 是控制理论中最基本也最常见的参数之一，然而初学者往往对其缺乏直观的认识。本节对此作详细阐释。

τ 的物理意义。对于任意一阶系统 $G(s) = \frac{K}{\tau s+1}$，其阶跃响应为：

$$y(t) = K\left(1 - e^{-t/\tau}\right)$$

在 $t = \tau$ 时，输出达到终值的 $1 - e^{-1} \approx 63.2\%$，即所谓 “63% 法则”。为何恰好是 63.2%？原因在于一阶系统具有如下特性：*任意时刻的变化速率与当前状态到目标之间的距离成正比*。在 $t = 0$ 时系统偏差最大，因而变化速率最快；随着逐渐趋近目标值，速率相应减缓。这一过程可类比为一名短跑运动员，起跑阶段全力加速，而随着接近终点逐渐放慢脚步。经过一个 τ 后，系统完成 63.2% 的过渡；随后每经过一个 τ，再完成剩余距离的 63.2%。具体数值如下表所示：

时间	终值百分比	工程含义
1τ	63.2%	过渡过程尚在进行，响应仍在明显变化
2τ	86.5%	已接近终值，但仍有可观偏差
3τ	95.0%	满足大多数工程场景的精度要求
4τ	98.2%	进入 2% 误差带，对应调节时间 t_s
5τ	99.3%	可视为已达到稳态

物理类比：漏桶注水模型。考虑一个底部开有小孔的水桶。桶空时无水压，注入的水全部留在桶中，水位上升最快。随着水位升高，底部水压增大，漏出量随之增加。最终，注入流量与漏出流量达到平衡，水位不再变化。从空桶到达平衡水位 63% 所需的时间即为该系统的时间常数 τ，其大小由桶的容积（对应惯量）和孔径（对应阻尼/摩擦）共同决定。

电机系统的行为与此完全类似：输入电压驱动电流流入绕组，而反电动势与摩擦则阻碍转速的提升。机械时间常数 τ_m 刻画的正是电机转速从零过渡到稳态值的快慢。

τ 与其他关键参数的内在联系：

- **调节时间：**$t_s \approx 4\tau$ (2% 准则)。以 $\tau_m = 50$ ms 的电机为例，其开环调节时间约为 200 ms。若此响应速度不能满足需求，则需要选用更大功率的电机或引入反馈控制器。
- **带宽：**一阶系统的 -3 dB 带宽为 $\omega_{bw} = 1/\tau$。$\tau_m = 50$ ms 对应带宽 20 rad/s $\approx$ 3.2 Hz，这意味着在开环条件下，该电机在物理上无法跟踪频率高于 3.2 Hz 的指令信号。
- **极点位置：**$G(s) = K/(\tau s + 1)$ 的极点位于 $s = -1/\tau$。τ 越小，极点在 s 平面上越偏左，系统动态响应越快。换言之，“更快的电机对应更小的时间常数” 在传递函数语言中等价于 “极点距虚轴更远”。
- **采样率：**数字控制器的采样周期一般不应超过 $\tau/10$。对于 $\tau_m = 50$ ms 的系统，建议采样周期不大于 5 ms，即采样频率至少为 200 Hz。

直流增益 K 是时间常数的互补参数。时间常数描述系统响应的*速度*，而直流增益描述系统输出的*幅值*。以电机为例，K_m 表示单位电压对应的稳态转速。若 $K_m = 10$ rad/s/V，则 1 V 输入最终产生 10 rad/s 的转速。当应用场景要求 100 rad/s 时，至少需要 10 V 的驱动电压。值得注意的是，反馈控制器无法改变被控对象的直流增益，它只能改变系统到达稳态的速度。

电气时间常数与机械时间常数的分离。电机实际上具有两个时间常数：电气时间常数 $\tau_e = L/R$（通常在毫秒量级）与机械时间常数 $\tau_m = JR/(bR + K_tK_e)$（通常在 20–200 ms 量级）。由于 $\tau_e \ll \tau_m$，电气过渡过程在机械响应尚未显著变化之前便已基本完成。这正是将二阶电机模型简化为一阶模型的物理依据：当转速开始发生可观测变化时，电流早已达到其准稳态值。从传递函数的角度来看，快速的电气极点在 s 平面上远离虚轴，对系统总体响应的贡献可以忽略——即该极点不构成*主导极点*（*Dominant Pole*）。

主导极点近似。当系统具有多个时间常数时，最大的时间常数（即最靠近虚轴的极点）对系统响应起决定性作用。这一事实使得高阶系统往往可以用低阶模型进行有效近似——那些对应较小时间常数的快速极点仅在极短暂的初始阶段产生影响，随即迅速衰减。这并非是不严谨的简化，而是基于充分物理依据的合理近似，也是一阶模型在工程实践中广泛适用的根本原因。

无刷电机与 FOC

无刷直流电机（BLDC Motors，Brushless DC Motors）是三相电机，但运行**磁场定向控制（FOC，Field Oriented Control）**的现代电调在内部处理了三相的复杂性。FOC 利用 Park 变换和 Clarke 变换将三相电流投影到旋转的 d-q 参考系，从而在 q 轴上等效出一台直流有刷电机。从控制器的角度看，电调暴露的是一台*虚拟直流有刷电机*——输入电压（或电流），观测角速度，同样的传递函数直接适用，无需考虑换相问题。

理解 FOC：原理与意义

无刷电机有三组定子绕组，彼此间隔 120°。要驱动转子旋转，这三组绕组必须按特定序列通电，使磁场始终“拉着”转子前进。难点在于：所需电流是正弦的，且其相位取决于转子的角位置。同时控制三个随时间和位置变化的正弦信号，处理起来十分复杂。

FOC 的核心思想：通过坐标变换，将问题化为等效的直流有刷电机控制。

第一步：Clarke 变换（$abc \to \alpha\beta$）。将三相电流 i_a, i_b, i_c 投影到固定的二维坐标系 (α, β)。由于 $i_a + i_b + i_c = 0$，一个变量是冗余的，可在不丢失信息的前提下降维。结果：三个正弦变为两个正弦。

第二步：Park 变换（$\alpha\beta \to dq$）。利用实测转子角度 θ_e，将固定坐标系旋转至与转子磁极对齐的**旋转参考系**。在该参考系下：

- q 轴电流（i_q）产生力矩——与直流有刷电机中的电枢电流完全等效。
- d 轴电流（i_d）控制磁场强度。大多数应用中令 $i_d = 0$（无需弱磁或增磁）。

经过 Park 变换后，三个时变正弦量在稳态下变为两个*常数*（i_d 和 i_q）。控制常数比控制正弦容易得多——只需使用 PI 控制器。

第三步：dq 轴 PI 控制。两个简单的 PI 控制器：

- d 轴 PI：保持 $i_d = 0$（或在高速弱磁时跟踪参考值）。
- q 轴 PI：跟踪期望力矩电流 $i_q^* = \tau_{\text{desired}}/K_t$。

第四步：逆变换。将 dq 电压指令经逆 Park、逆 Clarke 变换还原为三相电压，通过 PWM 施加到电机绕组。

FOC 对机器人的意义：

- **精确的力矩控制**。i_q 直接决定力矩，因此 FOC 能够实现精确的力矩（即电流）控制——这对力控应用和低速平稳运行至关重要。
- **最大效率**。令 $i_d = 0$，则所有电流都用于产生力矩，没有能量浪费在无用的磁场分量上。

结果：发热更少，电池续航更长。

- **全速域平稳运行**。与较简单的换相方式（梯形波/六步换相）不同，FOC 产生平滑的正弦电流，无力矩脉动。云台在低速时不会出现“齿槽感”。
- **简化上层控制**。电调在内部以 10–40 kHz 完成全部 FOC 运算。上层控制器看到的是干净的电流（或电压）接口——一台虚拟直流有刷电机。只需设计速度环和位置环即可。

FOC 需要什么：FOC 依赖精确的转子位置反馈，通常使用磁编码器（如 AS5048）或霍尔传感器。没有位置反馈时，需采用无传感器 FOC（通过反电动势估算位置）——在无人机电调中常见，但精度不及有传感器方案。

状态空间形式

进行观测器设计、LQR 或 MPC 时，你需要完整的状态空间表示。取 $\mathbf{x} = \begin{bmatrix} i & \omega \end{bmatrix}^T$，$u = V$：

$$\dot{\mathbf{x}} = A\mathbf{x} + Bu, \qquad A = \begin{bmatrix} -R/L & -K_e/L \\ K_t/J & -b/J \end{bmatrix}, \quad B = \begin{bmatrix} 1/L \\ 0 \end{bmatrix} \tag{30}$$

当 $L \approx 0$ 时，电流状态 i 成为代数变量，状态简化为 $\mathbf{x} = [\omega]$，回到一阶模型。当有电流传感器时，完整的两状态形式才是你输入 LQR 或卡尔曼滤波器的模型。

实用技巧：如何获取电机参数

你很少能拿到包含 J、b、L 的数据手册。自己动手测量：

- **堵转测试**——施加已知的低电压 V_s，锁住转轴，测量堵转电流 I_s。则 $R = V_s/I_s$。
- **空载测试**——无负载，施加额定电压 V_0，等待稳态后测量 ω_{free}。则 $K_e = (V_0 - I_{\text{free}}R)/\omega_{\text{free}}$（SI 单位：V·s/rad）。在 SI 单位下，$K_t = K_e$ 严格成立。
- **阶跃响应**——施加电压阶跃，记录 $\omega(t)$。达到最终值 63% 的时间即为 τ_m。
- **转动惯量**——由 $\tau_m = JR/(bR + K_tK_e)$ 反解 J；或用*摆动测试*：在已知力臂上挂已知质量，在已知力矩下测量角加速度 α，计算 $J = \tau/\alpha$。

使用带电流控制器的 FOC 电调时的捷径：大多数现代 FOC 电调以极高带宽闭合内部电流环。若该环激活，电流 i 几乎瞬间跟踪指令。等效被控对象简化为*纯机械动态*：

$$\frac{\Omega(s)}{I_{\text{ref}}(s)} = \frac{K_t}{Js + b}$$

你只需要 K_t、J 和 b——而 b 通常可以忽略不计。这就是你用于调试外部速度环的被控对象。

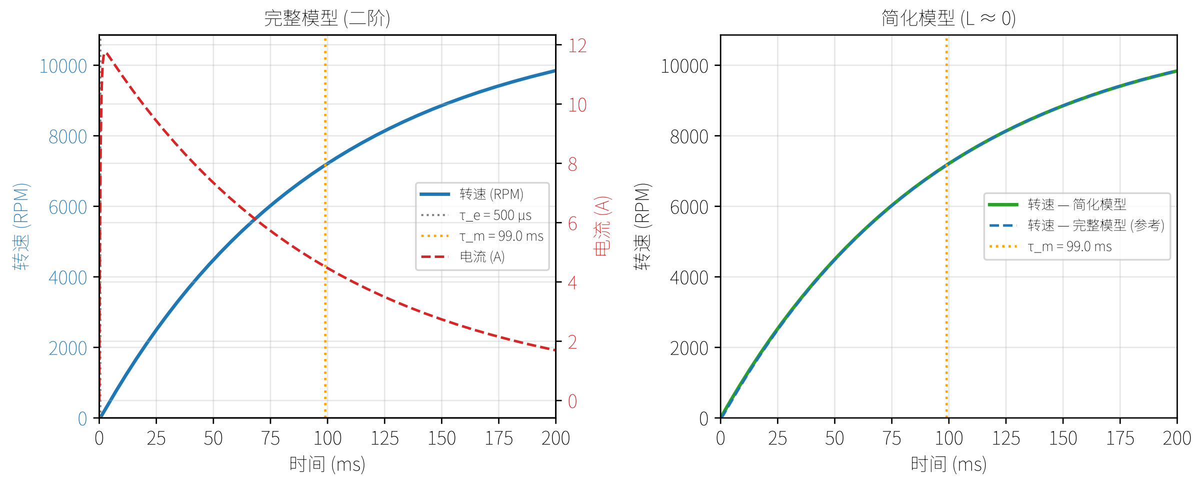

图 5: 直流有刷电机阶跃响应：从电压到角速度。左图：完整二阶模型，展示了快速的电气瞬态（$\tau_e \approx 1$ ms）和缓慢的机械上升过程（$\tau_m \approx 50$ ms）。右图：一阶近似（$L \approx 0$）几乎完美地还原了机械响应——电气极点响应太快，根本无关紧要。

4 经典控制理论

终于到了你最想看的章节。经典控制理论讲的就是怎么让系统听话。而经典控制的绝对主角——95% 的比赛机器人都在用的工具——就是 PID 控制器。

不过在讲 PID 之前，得先搞清楚反馈控制为什么管用。

4.1 反馈控制的基本概念

开环控制就像闷头做菜不尝味道。严格按菜谱操作，祈祷结果没问题。烤箱温度偏了 10 度？蛋糕废了。

闭环（反馈）控制就像边做边尝。尝一口，调味，再尝，再调。烤箱温度不准也没关系，因为你在实时补偿。

用数学语言描述，一个基本的反馈回路如下：

1. 计算**误差（error）**：$e(t) = r(t) - y(t)$，其中 r 是期望输出（设定值），y 是实际输出。
2. 将误差输入**控制器（controller）** $C(s)$，产生控制输入 $u(t)$。
3. 控制输入驱动**被控对象（plant）** $G(s)$，产生输出 $y(t)$。
4. 测量 $y(t)$，返回第 1 步。

从设定值 R 到输出 Y 的闭环传递函数（Closed-loop Transfer Function）为：

$$\frac{Y(s)}{R(s)} = \frac{C(s)G(s)}{1 + C(s)G(s)} \tag{31}$$

经典控制里最重要的一个公式。多看两眼，记住它。

解析闭环传递函数：

令 $L(s) = C(s)G(s)$ 为**回路传递函数（loop transfer function）**（也叫开环传递函数）。则：

- **互补灵敏度函数（Complementary Sensitivity，设定值跟踪）**：$T(s) = \frac{L(s)}{1+L(s)}$。这就是上面写的那个——它告诉你输出跟随参考的好坏程度。理想情况下，$T(s) \approx 1$（低频处跟踪良好），$T(s) \approx 0$（高频处抑制噪声）。
- **灵敏度函数（Sensitivity Function）**：$S(s) = \frac{1}{1+L(s)}$。它告诉你输出端的扰动对输出的影响程度。S 越小 = 抗扰性越好。注意 $T(s) + S(s) = 1$ 始终成立——你不可能在同一频率下同时实现完美跟踪和完美抗扰。这是反馈控制的基本局限。
- **输入灵敏度（Input Sensitivity）**：$C(s) \cdot S(s) = \frac{C(s)}{1+L(s)}$。它告诉你需要多大的控制量。如果这个量在高频处很大，你的执行器就会没日没夜地放大噪声。

一句话抓住要害： 回路增益大（$|L(j\omega)| \gg 1$）的频段内，$T \approx 1$，$S \approx 0$——反馈在干活。回路增益小（$|L(j\omega)| \ll 1$）的频段内，$T \approx 0$，$S \approx 1$——反馈摆烂。**带宽（bandwidth）**就是 $|L(j\omega)| = 1$（穿越频率）对应的频率，是" 反馈管用" 和" 反馈不管用" 的分界线。

反馈凭什么这么强：

- **不怕模型不准**。$G(s)$ 和实际有出入？反馈能兜底。
- **自动抗扰**。摩擦、风、碰撞——来一个纠一个。

- **重塑动态响应**。选对 $C(s)$，系统可以更快、阻尼可以更足。

代价：反馈也可能把系统搞炸。控制器太猛，纠正过头，误差反向，再纠正再过头，一来二去就振荡甚至失稳。所以我们需要分析稳定性的工具。

案例分析：淋浴水温问题

反馈控制失稳这件事，你在自家浴室就体验过：

打开淋浴，水是凉的。热水旋钮拧到底（K_p 拉满）。两秒钟啥反应都没有（管道延迟——这就是控制理论里的**传输滞后**，transport lag）。等不及了，继续加。突然，滚烫的水喷上来。赶紧拧向冷水。两秒后，冰水。在滚烫和冰冷之间反复横跳 30 秒，放弃。

出了什么问题（用控制理论来说）：

- **被控对象（plant）**是热水器加管道，有显著的**时间延迟（time delay）**（约 2 秒）。
- 你（控制器）使用了非常高的 K_p——激进的纠正动作。
- 时间延迟引入了相位滞后（phase lag）。在你调节旋钮的频率上，总相位滞后超过 180°。你的相位裕度（phase margin）为负。系统**失稳**。

解决方法：慢慢拧，然后**等**。这相当于降低 K_p，把带宽压到远低于 $1/(2 \times 延迟)$。用速度换稳定性——你在机器人上每个控制回路里都会遇到同样的取舍。

教训：有延迟的系统天生难控制。机器人上有通信延迟（CAN 总线、视觉处理），你就必须降低控制器的激进程度，或者上 Smith 预测器（一种补偿已知延迟的技术）。

4.2 系统性能指标

设计控制器时，得先定义”好”长什么样。对阶跃响应（设定值突变），标准指标如下：

- **上升时间（Rise time）**（t_r）：输出从终值的 10% 上升到 90% 所需的时间。通常越快越好。
- **超调量（Overshoot）**（M_p）：输出超过设定值的幅度，以百分比表示。通常越小越好。
- **调节时间（Settling time）**（t_s）：输出保持在终值 ±2%（或 5%）范围内所需的时间。
- **稳态误差（Steady-state error）**（e_{ss}）：系统稳定后剩余的误差。理想情况下为零。

这几个指标之间经常打架：上升时间越快，超调往往越大。控制设计就是在你的场景下找到合适的折中。跟踪目标的云台——超调要小，不能在目标前后抖。底盘冲刺到最高速——超调无所谓，越快到速越好。

4.2.1 稳定裕度

系统稳定了还不够，我们想知道它有多稳定——离翻车还有多远。两个数字告诉你答案：

增益裕度（Gain Margin，GM）：在系统失稳之前，回路增益还能增加多少。以 dB 为单位。增益裕度 6 dB 意味着增益可以翻倍。典型目标：> 6 dB。

相位裕度（Phase Margin，PM）：在系统失稳之前，系统还能承受多少额外的相位滞后。以度为单位。相位裕度 45° 意味着系统还能承受额外 45° 的滞后。典型目标：30°–60°。

你可以从开环传递函数 $C(s)G(s)$ 的 **Bode 图（Bode Plot）**上直接读取这两个裕度。

什么是 Bode 图？Bode 图是两张上下叠放的图，横轴都是频率（对数坐标）：

- **上图：幅频图**。显示增益 $|G(j\omega)|$（dB）与频率的关系。0 dB 表示增益为 1（输出等于输入）。+20 dB 表示增益为 10。-20 dB 表示增益为 0.1。换算公式：$\text{dB} = 20\log_{10}(|G|)$。

- **下图：相频图**。显示相角 $\angle G(j\omega)$（度）与频率的关系。相位 $-90°$ 意味着输出比输入滞后四分之一个周期。

如何从 Bode 图读取稳定裕度：

- **相位穿越频率（Phase crossover frequency）**（ω_{pc}）：相位穿越 $-180°$ 的频率。在此频率读取幅值——幅值曲线到 0 dB 的距离就是**增益裕度**。如果在 ω_{pc} 处幅值低于 0 dB，系统稳定，增益裕度为正。
- **增益穿越频率（Gain crossover frequency）**（ω_{gc}）：幅值穿越 0 dB 的频率。在此频率读取相位——相位曲线到 $-180°$ 的距离就是**相位裕度**。如果在 ω_{gc} 处相位高于 $-180°$，系统稳定，相位裕度为正。

Bode 图速绘技巧： 对于传递函数 $G(s)$：

- 位于 $s = -a$ 处的极点（pole）贡献从 $\omega = a$ 开始的 -20 dB/decade 衰减，以及以 $\omega = a$ 为中心的 $-45°$/decade 相位变化。
- 位于 $s = -a$ 处的零点（zero）贡献 $+20$ dB/decade 和 $+45°$/decade（与极点相反）。
- 积分器（$1/s$）在所有频率贡献 -20 dB/decade，以及恒定的 $-90°$ 相位。
- 直流增益决定幅频图的纵向偏移。

掌握这些规则，你可以在一分钟内手绘近似的 Bode 图——在调试时需要快速验证思路时非常管用。

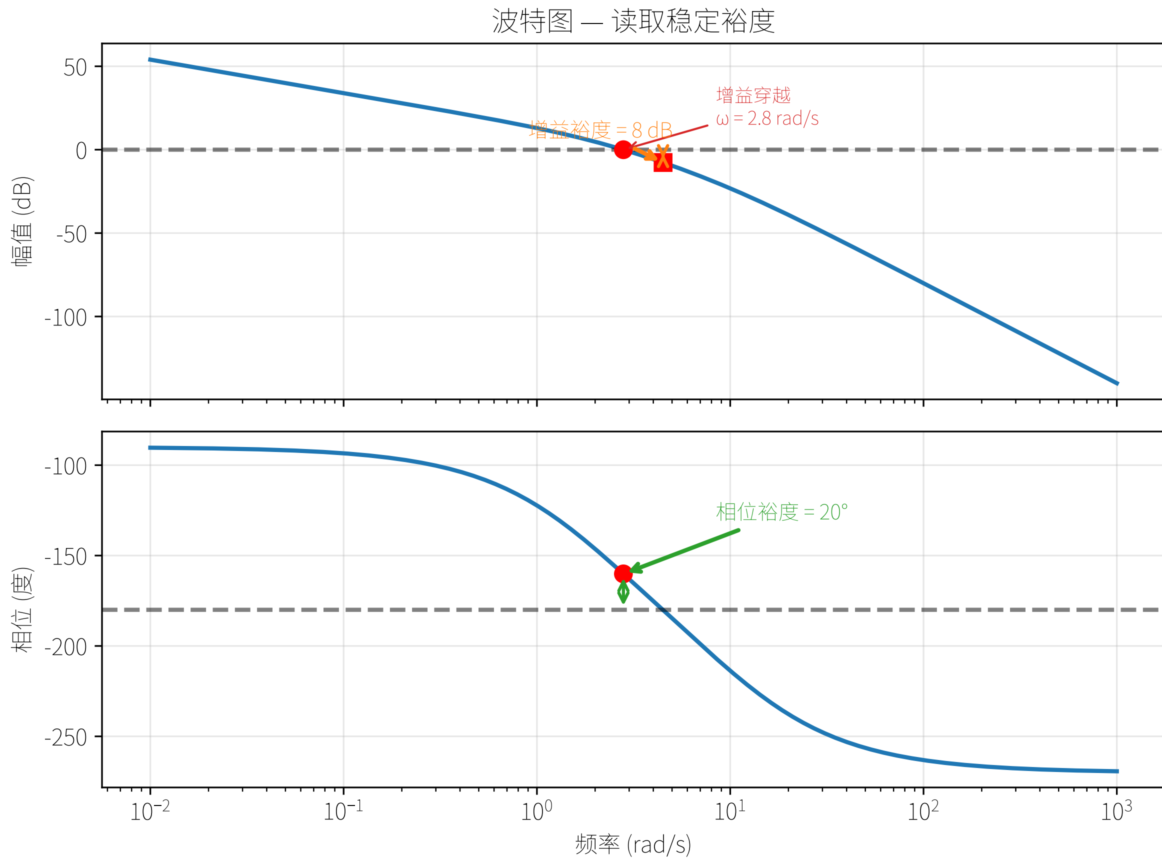

图 6: 开环系统 $L(s) = \frac{100}{s(s+2)(s+10)}$ 的 Bode 图。增益裕度（橙色）和相位裕度（绿色）可直接从图中读取。两者均为正值，故系统稳定。

经验法则：如果系统振荡，说明相位裕度太小。如果系统响应迟钝，说明增益裕度太大（你过于保守了）。相位裕度约 45° 通常能取得不错的平衡。

4.3 基于传递函数的控制器设计

经典控制器设计的目标是：给定被控对象 $G(s)$，选择控制器 $C(s)$，使得闭环系统 $\frac{C(s)G(s)}{1+C(s)G(s)}$ 满足性能指标。

方法有很多（根轨迹、频率响应整形、极点配置），但对于竞赛机器人而言，最实用的方法是：

1. 从 PID 控制器出发（下一节介绍）。
2. 用系统化的方法进行整定。
3. 如果 PID 不够，添加前馈或切换到现代方法（第 6 节）。

听起来好像不够高大上？事实就是：全世界大约 95% 的工业控制回路跑的都是 PID。简单、皮实、管用。没必要把事情搞复杂。

4.4 PID 控制器

PID 控制器把误差 $e(t)$ 拆成三项来计算控制信号：

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau) d\tau + K_d \frac{de(t)}{dt} \tag{32}$$

三项各司其职：

- **P（比例）：**”差多少就推多少”——误差大就猛推，误差小就轻推。最直觉的一项，像弹簧。
- **I（积分）：**”这误差怎么老不消？肯定有东西在拖后腿。”它一直攒力气直到误差归零。能消稳态误差，但容易超调和积分饱和（windup）。
- **D（微分）：**”误差在猛变，赶紧踩刹车！”它提供阻尼、压制超调，但也会放大噪声。

打个比方：你开车，前面有个停车标志。

- P：”还有 100 米，踩油门。””还有 10 米，松油门。”
- D：”靠近得太快了——现在就刹车，别等到跟前！”
- I：”停在标志前 1 米了，没对准。慢慢蹭，蹭到精确位置。”

4.4.1 PID 控制器——连续形式

PID 控制器的传递函数为：

$$C(s) = K_p + \frac{K_i}{s} + K_d s = K_p \left(1 + \frac{1}{T_i s} + T_d s\right) \tag{33}$$

其中 $T_i = K_p/K_i$ 是积分时间，$T_d = K_d/K_p$ 是微分时间。

从频域理解 PID：

PID 控制器的 Bode 图揭示了它为何有效：

- **极低频处**（$\omega \to 0$）：K_i/s 项主导。增益趋向无穷大（$+\infty$ dB），I 项之所以能消稳态误差，就是因为这一点——直流处增益无穷大，直流误差必须为零。
- **中频处**：K_p 主导。增益约为 $20\log_{10}(K_p)$ dB。控制器的大部分工作在这个频率范围完成。
- **高频处**（$\omega \to \infty$）：$K_d s$ 主导。增益以 +20 dB/decade 随频率增大，D 项放大噪声的原因就在这里——噪声的频率越高，D 项的增益越大。

PID 有**两个折断频率（break frequencies）**：$\omega_1 = 1/T_i$（I 项移交给 P 项）和 $\omega_2 = 1/T_d$（P 项移交给 D 项）。两者之间，控制器表现为纯比例控制。幅频图在对数坐标下看起来像”V” 形（或浴缸形）：低频处增益大（I），中间平坦（P），高频处上升（D）。

为什么另一种形式很重要：$K_p(1 + \frac{1}{T_i s} + T_d s)$ 的形式使物理含义更清晰：T_i 表示”积分项回看多久”，T_d 表示”微分项向前预测多远”。T_i 越大，积分动作越慢。T_d 越大，预测性越强（但对噪声也更敏感）。

各增益的影响：

增大	上升时间	超调量	调节时间	稳态误差
K_p	减小	增大	小幅变化	减小
K_i	减小	增大	增大	消除
K_d	小幅变化	减小	减小	无影响

调参起手式：先只用 P。有稳态误差？加 I。超调或振荡太大？加 D。千万别三个参数同时非零地往上调——你根本分不清是哪个旋钮在起作用。

案例分析：整定云台偏航电机——亲眼看 P、I、D 各自的作用

下面一步步整定 RoboMaster 云台偏航轴。电机 GM6020，CAN 总线控制，目标是跟踪角度。设定值：90° 阶跃。

第 1 步：只用 P（$K_p = 5.0$，$K_i = 0$，$K_d = 0$）

云台快速向目标摆动，但超调约 15°，振荡 3–4 次，最终以 **2° 的稳态误差**稳定。为什么有误差？因为云台接近目标时，误差变小，P 项输出也随之减小。到某个时刻，P 项输出等于摩擦力矩，云台在还未到达目标时就停下来了。单靠 P 项无法消除持续扰动（如摩擦或重力）引起的误差。

第 2 步：加入 I（$K_p = 5.0$，$K_i = 0.5$，$K_d = 0$）

稳态误差消失了！积分项缓慢积累小的残余误差并推动电机，直到误差精确为零。但现在超调更大了（约 20°），还有缓慢振荡，需要 2 秒才能衰减。积分项在" 记住" 过去的误差并施加了不必要的力。这是 I 项经典的权衡：修复了直流误差，但使瞬态响应变差了。

第 3 步：加入 D（$K_p = 5.0$，$K_i = 0.5$，$K_d = 2.0$）

超调量大幅下降至约 5°，振荡几乎消失。微分项感知到快速靠近并在云台到达目标之前施加制动力矩。就像一位经验丰富的司机，在到达路口前很早就开始刹车，而不是到最后一刻。

第 4 步：精细调整（$K_p = 6.0$，$K_i = 0.3$，$K_d = 3.0$）

略微增大 K_p 以加快上升速度，减小 K_i（我们只需要它来克服静摩擦，不需要激进的误差纠正），增大 K_d 以增加阻尼。结果：快速、平滑的跟踪，超调不到 3°，稳态误差为零。云台在 0.3 秒内锁定目标。

把每个参数调到过大会发生什么？

- K_p 过大：云台剧烈振荡——超调太猛，纠正动作又向另一方向超调。这是失稳的前兆。
- K_i 过大：缓慢、黏糊的振荡。积分" 记得" 太用力，不断过度纠正。云台像果冻一样摇摆。
- K_d 过大：电机嗡嗡震动。微分放大传感器噪声，导致快速来回的力矩指令。你的电机听起来像一只愤怒的蜜蜂。

教训：每个参数都有一个" 甜蜜点"。搞懂每一项的物理机制（弹簧、记忆、阻尼器），出了问题你就能秒定位，而不是蒙着眼乱拧旋钮。

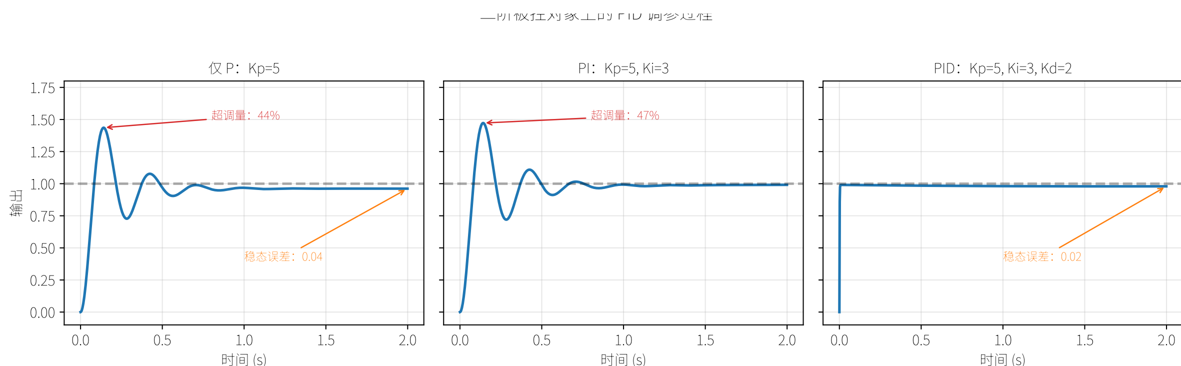

图 7: 二阶被控对象上的 PID 整定过程。纯 P 存在稳态误差；PI 消除稳态误差但超调更大；PID 在保持零稳态误差的同时抑制了超调。

4.4.2 PID 控制器——离散形式

你的微控制器无法计算连续的积分和微分，它以固定采样时间步长 T_s 处理离散样本。离散 PID 为：

$$u[n] = K_p\, e[n] + K_i\, T_s \sum_{k=0}^{n} e[k] + K_d \frac{e[n] - e[n-1]}{T_s} \tag{34}$$

或者用更便于实现的**增量式（速度式）形式（incremental / velocity form）**：

$$\Delta u[n] = K_p(e[n] - e[n-1]) + K_i\, T_s\, e[n] + K_d \frac{e[n] - 2e[n-1] + e[n-2]}{T_s} \tag{35}$$

$$u[n] = u[n-1] + \Delta u[n] \tag{36}$$

增量式的实践优势：如果控制器复位或出现故障，不会突然产生大幅输出——每次只调整一个小增量。这对你的硬件更安全。

基础离散 PID 伪代码：

```
error = setpoint - measurement
integral += error * dt
derivative = (error - prev_error) / dt
output = Kp * error + Ki * integral + Kd * derivative
prev_error = error
```

警告：这种朴素实现在实践中有三个问题会坑到你：积分饱和（integral windup）、微分冲击（derivative kick）和噪声放大。请阅读下一节。

4.4.3 改进的 PID 控制器

教科书上的 PID 有几个众所周知的问题。以下是每个竞赛机器人都应该使用的修复方案。

1. 积分饱和与抗积分饱和（Integral Windup and Anti-windup）

问题：执行器饱和了（电机已经到最大电压），积分项还在拼命攒误差。等误差方向反过来，积分值已经攒得巨大，要很久才能"退饱和"，于是大幅超调。

修复方案（限幅法）：

```
integral += error * dt
integral = clamp(integral, -max_integral, max_integral)
```

更好的修复方案（反向计算法）：当输出饱和时，按比例减小积分项：

```
output_raw = Kp * error + Ki * integral + Kd * derivative
output = clamp(output_raw, -max_output, max_output)
integral += (error + (output - output_raw) / Ki) * dt
```

2. 微分冲击（Derivative Kick）

问题：设定值突变（阶跃指令），误差 $e = r - y$ 瞬间跳一大截。微分项把它当成无限大的变化率，输出直接飙出一个巨大尖峰。

修复方案：对测量值求导，而非对误差求导：

```
derivative = -(measurement - prev_measurement) / dt
```

测量值变化是平滑的（电机又不会瞬移），自然不会有冲击。

3. 微分噪声放大（Derivative Noise Amplification）

问题：微分放大高频噪声。测量值一有噪声，微分项的输出就剧烈抖动。

修复方案：对微分项进行低通滤波：

$$D(s) = \frac{K_d s}{1 + \frac{K_d}{N K_p} s} \tag{37}$$

在代码中，这是对微分项的一阶低通滤波，N（通常为 8–20）控制滤波器截止频率。

4. 设定值加权（Setpoint Weighting）

有时你希望 P 项和 D 项对设定值变化与扰动的响应不同。设定值加权引入参数 b 和 c：

$$u = K_p(b \cdot r - y) + K_i \int (r - y)\, dt + K_d \frac{d(c \cdot r - y)}{dt} \tag{38}$$

设 $b = 1, c = 0$（仅对测量值求导）是最常见的配置。

4.5 PID 进阶改进

以上四个修复是竞赛机器人的底线。但还有更深一层的结构性改进，能让性能产生质的变化。下面从频域角度分析，配合仿真演示。

4.5.1 积分分离（Integral Separation）

问题：当误差较大时（例如刚收到阶跃指令后），积分项积极累积。这个”预充电”的积分在系统接近设定值时会引起超调——积分”记住”了大误差阶段太多的历史。

修复方案：只有当误差小于阈值 ε 时才激活积分项：

$$u_I = \begin{cases} K_i \int e\, dt & \text{如果} |e| < \varepsilon \\ 0 & \text{如果} |e| \geq \varepsilon \end{cases} \tag{39}$$

频域解释：积分分离实际上使控制器变成**非线性**——大误差时表现为 PD 控制器（无积分），小误差时切换为完整的 PID。在频域中，这意味着：

- 大瞬态期间：直流处的开环增益是有限的（仅 PD），因此不存在无穷低频增益。这防止了积分在瞬态期间的”饱和”。
- 接近稳态时：积分激活，提供无穷直流增益和零稳态误差——正是我们需要的。

控制器实际上根据误差幅度拥有**两种不同的 Bode 图**：一种没有低频处的 K_i/s 极点（类 PD），一种有（PID）。阈值 ε 控制”切换”的时机。

代码：

```
if (fabs(error) < epsilon) {
    integral += Ki * error * dt;
}
// 否则：积分保持冻结（既不增长也不复位）
```

如何选择 ε： 将其设置为纯 PD 控制器的近似稳态误差。这样，积分仅在 PD 完成了大部分工作、只需要最后一点” 微调” 时才激活。

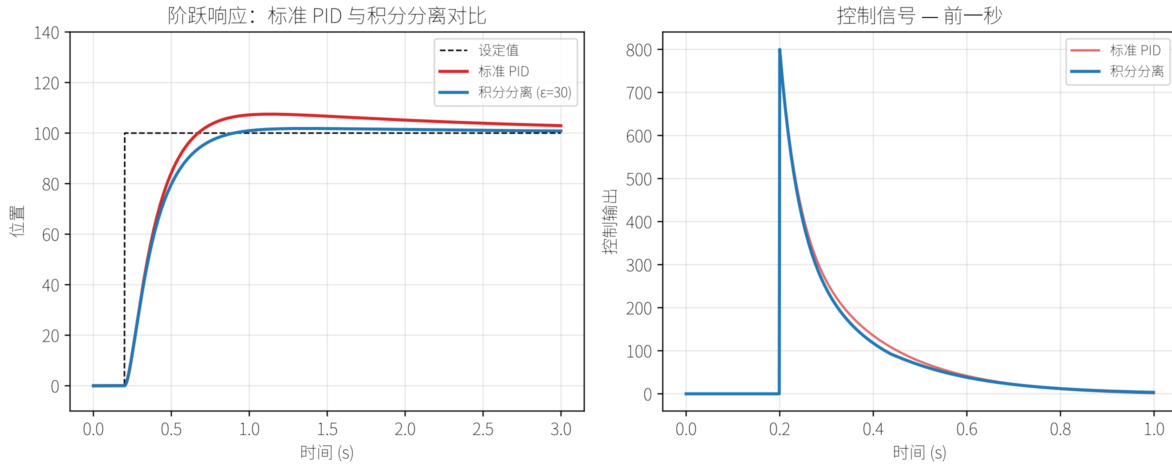

图 8: 左图：积分分离（蓝色）相比标准 PID（红色）减少了超调，因为积分不在大初始瞬态期间累积。右图：第一秒内的控制信号——注意分离版本的峰值更小。

4.5.2 微分位置：PID vs PI-D vs I-PD

标准 PID 将三项都作用于误差 $e = r - y$。但有**三种结构变体**，改变了 P 项和 D 项作用于哪个信号：

结构	P 作用于	I 作用于	D 作用于
PID	误差 $(r - y)$	误差 $(r - y)$	误差 $(r - y)$
PI-D	误差 $(r - y)$	误差 $(r - y)$	测量值 $(-y)$
I-PD	测量值 $(-y)$	误差 $(r - y)$	测量值 $(-y)$

频域分析： 三种结构具有相同的闭环抗扰传递函数（从扰动到输出）。它们仅在设定值到输出的传递函数上有所不同：

- **PID：** $\frac{Y}{R} = \frac{(K_d s^2 + K_p s + K_i)G(s)}{s + (K_d s^2 + K_p s + K_i)G(s)}$ ——分子含 $K_d s^2$ 项，在阶跃响应中产生” 微分冲击”（分子零点放大了阶跃的高频成分）。
- **PI-D：** $\frac{Y}{R} = \frac{(K_p s + K_i)G(s)}{s + (K_d s^2 + K_p s + K_i)G(s)}$ ——从设定值通路中移除了 $K_d s^2$，消除了微分冲击。分母（稳定性、抗扰性）不变。
- **I-PD：** $\frac{Y}{R} = \frac{K_i \cdot G(s)}{s + (K_d s^2 + K_p s + K_i)G(s)}$ ——从设定值通路中同时移除 $K_d s^2$ 和 $K_p s$。阶跃响应最平滑（无 P 或 D 引起的超调），但也最慢。

要点： 这些结构让你可以把设定值跟踪（分子控制）和抗扰性能（分母控制）拆开来分别调。分母一样 = 稳定性和抗扰性一样；分子不同 = 设定值跟踪行为不同。

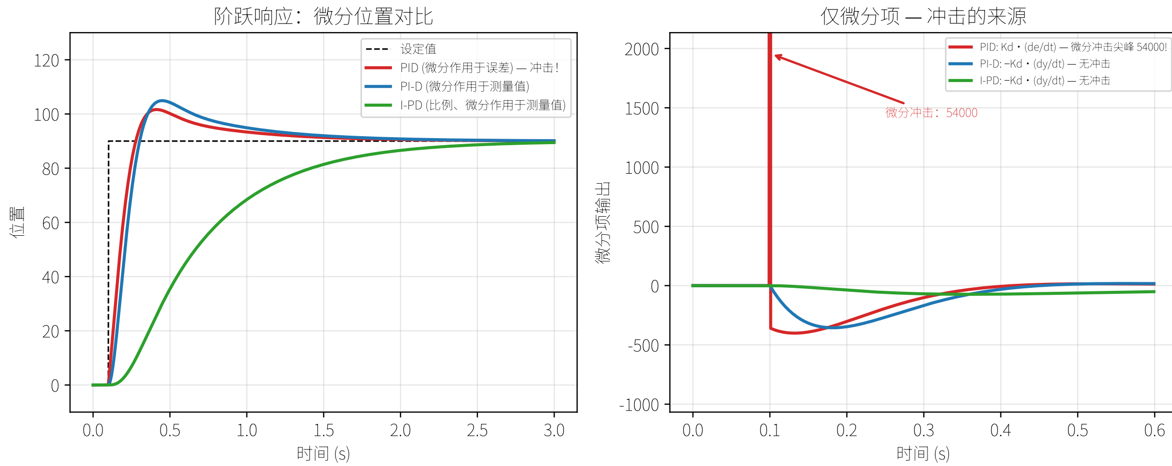

图 9: 左图：PID（红色）显示微分冲击尖峰和超调。PI-D（蓝色）消除了冲击。I-PD（绿色）最平滑但最慢。右图：控制信号揭示了标准 PID 中的剧烈冲击——这个尖峰会磨坏齿轮。

4.5.3 滤波 PID：频域证明

理想 PID 控制器 $C(s) = K_p + K_i/s + K_d s$ 有一个根本性问题：$K_d s$ 项在高频处具有**无界增益**。在 Bode 图上，$K_d s$ 的幅值以 +20 dB/decade 永远上升。这意味着：

1. 传感器噪声（存在于高频）被无限放大。
2. 控制器要求执行器响应速度无限快，这在物理上是不可能的。
3. 在离散时间中，微分项将高频噪声混叠进控制带。

解决方案——滤波微分（filtered derivative）：

$$C_{\text{filtered}}(s) = K_p + \frac{K_i}{s} + \frac{K_d N s}{s + N} \tag{40}$$

$\frac{Ns}{s+N}$ 项是截止频率为 $\omega = N$ rad/s 的一阶高通滤波器。在 N 以下，它表现为纯微分（$\approx s$）。在 N 以上，它趋于常数增益 N。在 Bode 图上：

- $\omega < N$ 处：滤波 PID 与理想 PID 完全相同，性能一致。
- $\omega > N$ 处：幅值**趋于平坦**，而不是永远上升。最大微分增益为 $K_d \cdot N$，噪声放大有界。

如何选择 N： 典型值为 $N = 8$–20。规则：N 要足够大，使微分在控制器带宽处仍有效工作；但也要足够小以衰减噪声。具体而言：$N > 5 \times \omega_{\text{bandwidth}}$（避免滤波器吃掉有用的微分作用）且 $N < \omega_{\text{noise}}/3$（充分衰减噪声）。

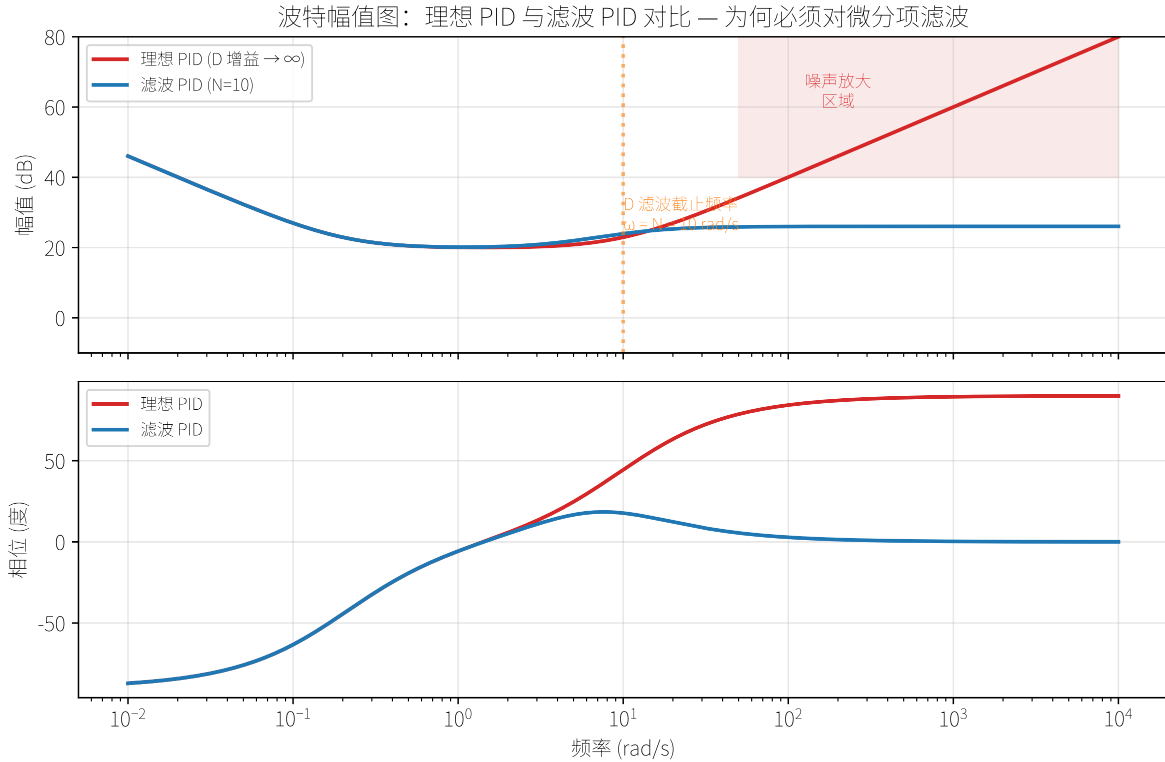

图 10: Bode 图：理想 PID（红色）在高频处增益永远上升——它会放大每一点传感器噪声。滤波 PID（蓝色）在 $\omega = N$ 以上趋于平坦，限制了噪声放大。在滤波器截止频率以下，两者完全相同。

输出级低通滤波器（Output-stage LPF）：另一种方法是在将 PID 输出发送到执行器前，对整个 PID 输出进行低通滤波：

$$U(s) = \frac{\omega_f}{s + \omega_f} \cdot C(s) \cdot E(s) \tag{41}$$

这种方法实现更简单（PID 后加一个低通滤波器），但它同时滤掉了 P 项和 I 项，可能减慢响应速度。仅对微分项滤波的方案更受青睐，因为它精确地针对问题所在。

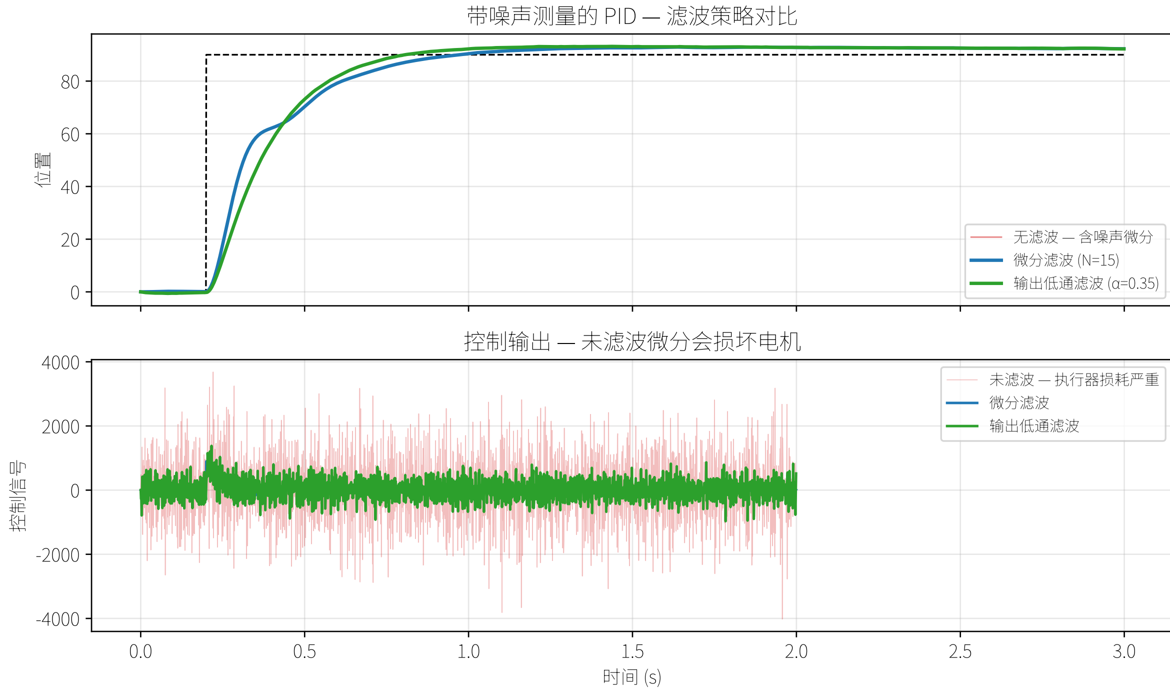

图 11: 上图：带噪声测量的位置响应。未滤波 PID（红色）因噪声放大的微分项而振荡。滤波微分（蓝色）和输出低通滤波（绿色）都能平滑响应。下图：未滤波的控制信号非常剧烈——会损坏真实的电机。

4.5.4 二自由度 PID（2-DOF PID）

标准 PID 有**一个自由度**：你选择 K_p、K_i、K_d 来满足你的指标。但存在一个根本性冲突：能给出良好设定值跟踪（快速、低超调的阶跃响应）的增益，与能给出良好抗扰性（受扰后快速恢复）的增益不同。

二自由度 PID 通过引入设定值加权参数 b（作用于 P 项）和 c（作用于 D 项）来解决这一矛盾：

$$u = K_p(b \cdot r - y) + K_i \int (r - y)\, dt + K_d \frac{d(c \cdot r - y)}{dt} \tag{42}$$

频域分析： 闭环传递函数变为：

$$\frac{Y}{R} = \frac{(bK_p + cK_d s + K_i/s) \cdot G(s)}{1 + C(s)G(s)} \quad \text{（设定值跟踪——由 } b\text{、}c \text{ 调节）} \tag{43}$$

$$\frac{Y}{D} = \frac{G(s)}{1 + C(s)G(s)} \quad \text{（抗扰性——与 } b\text{、}c \text{ 无关！）} \tag{44}$$

神奇之处： b 和 c 只影响设定值传递函数。抗扰性完全由 K_p、K_i、K_d 控制。现在你可以：

1. 调整 K_p、K_i、K_d 以实现激进的抗扰性。
2. 独立地降低 b（通常为 0.5–0.8）来柔化设定值响应并减少超调。
3. 设 $c = 0$ 以消除微分冲击（这就是 PI-D 结构）。

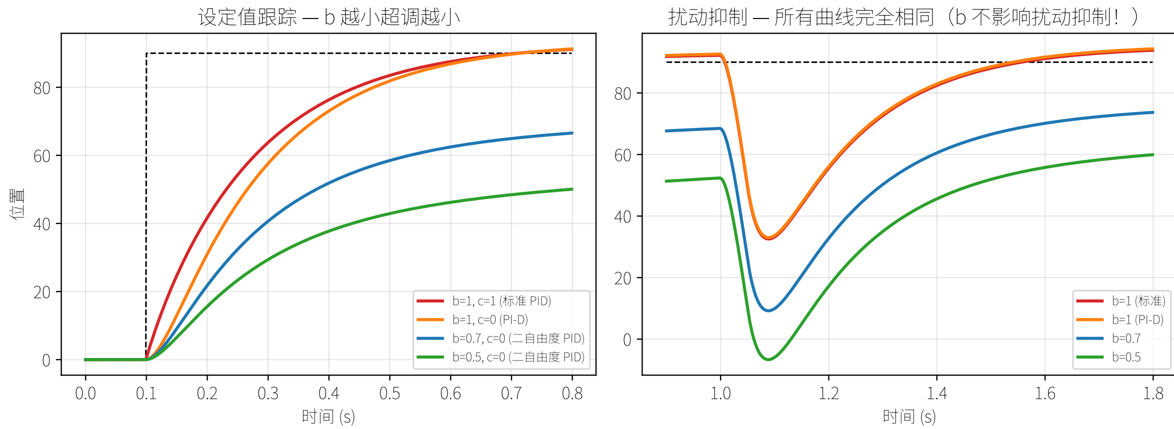

图 12: 左图：不同的 b 值独立控制设定值超调。右图：$t = 1.0\text{s}$ 时的扰动抑制对所有 b 值**完全相同**——证明了二自由度 PID 真正实现了两个目标的解耦。

4.5.5 模糊 PID（Fuzzy PID）

动机。标准 PID 控制器使用固定增益 K_p、K_i、K_d。然而最优增益取决于工作状态：误差较大时需要激进的响应，误差较小时则需要精细控制。模糊 PID 根据当前误差 e 及其变化率 $\dot{e}$ 实时调整 K_p、K_i、K_d。

工作原理。

1. **模糊化（Fuzzification）**。为误差 e 及其导数 $\dot{e}$ 定义模糊集合。常用的划分包含五个语言变量：NB（负大）、NS（负小）、ZO（零）、PS（正小）、PB（正大）。隶属度函数（通常为三角形或梯形）将 e 和 $\dot{e}$ 的精确值映射为各模糊集合的隶属度。

2. **规则表**。一张规则表(例如 5×5 或 7×7)将 $(e, \dot{e})$ 的每种组合映射为增益调整量 $(\Delta K_p, \Delta K_i, \Delta K_d)$。规则以自然语言描述，例如：
   - "若 e 为 PB 且 $\dot{e}$ 为 PB ⇒ 大幅增大 K_p，减小 K_i（防止积分饱和），增大 K_d。"
   - "若 e 为 ZO 且 $\dot{e}$ 为 NS ⇒ 保持 K_p 适中，启用 K_i 以消除稳态误差，减小 K_d。"

3. **解模糊（Defuzzification）**。将触发的规则进行聚合（常用重心法），得到 ΔK_p、ΔK_i、ΔK_d 的精确值。

4. **增益更新**。实际 PID 增益为：

$$K_p = K_{p0} + \Delta K_p, \qquad K_i = K_{i0} + \Delta K_i, \qquad K_d = K_{d0} + \Delta K_d$$

   其中 K_{p0}、K_{i0}、K_{d0} 为基准增益。

ΔK_p 简化规则表示例（3×3）。

$e \setminus \dot{e}$	N	Z	P
N	PB	PS	ZO
Z	PS	ZO	NS
P	ZO	NS	NB

解读：当误差为负且持续增大（$\dot{e}=\mathrm{N}$）时，大幅增加 K_p（PB）以驱动系统回到目标；当两者均接近零时，无需调整（ZO）。

实践评估。

- **优势**：模糊 PID 能够适应变化的工况——负载变化、非线性工作区域——而无需显式系统模型。
- **不足**：规则表的设计主要依赖经验。实质上是将"整定 3 个增益"转化为"整定一张规则表"，且更难以严格验证。
- **适用场景**：在竞赛机器人中，精心整定的固定 PID 或二自由度 PID 通常已经足够。当系统需要在差异显著的工况间切换时（例如底盘需同时适应空载和满载状态），模糊 PID 值得考虑。
- **与增益调度的对比**：增益调度在离散工作点之间手动切换预整定的 PID 参数集，切换边界处可能产生暂态。模糊 PID 由于隶属度函数在规则之间自然插值，能够提供更平滑的过渡。

4.6 PID 整定方法

你已经设计好了 PID 控制器。现在来回答那个永恒的问题：K_p、K_i、K_d 应该取什么值?

4.6.1 经验整定

比赛中最常用的方法，因为不需要数学推导，效果却出人意料地好。

Ziegler-Nichols 极限增益法（ultimate gain method）：

1. 令 $K_i=0$，$K_d=0$。
2. 逐渐增大 K_p，直到系统以恒定幅值振荡。此时的 K_p 即为**极限增益（ultimate gain）** K_u，振荡周期为 T_u。
3. 按下表设置 PID 增益：

控制器	K_p	K_i	K_d
P	$0.5K_u$	—	—
PI	$0.45K_u$	$0.54K_u/T_u$	—
PID	$0.6K_u$	$1.2K_u/T_u$	$0.075K_uT_u$

警告：Ziegler-Nichols 给出的整定结果比较激进，超调量约 25%。对于许多竞赛应用，你需要在此基础上适当减小 K_p 和 K_i。

手动整定（实用方法）：

1. 先只用 K_p。增大到响应快但略有振荡。
2. 加入 K_d 以抑制振荡。增大到振荡消失。
3. 如果有稳态误差，加入少量 K_i。缓慢增大——过大会引起缓慢振荡。
4. 精细调整。用不同设定值和扰动进行测试。

竞赛建议：始终在机器人实际负载下整定，而不是让电机空转。装了摄像头的云台与裸电机在测试台上的表现差异悬殊。

4.6.2 使用 MATLAB Simulink 整定

如果你能用 MATLAB，可以对系统建模，并使用 **PID 整定器（PID Tuner）**应用程序或 **Simulink Control Design** 自动找到最优增益。

基本流程：

1. 在 Simulink 中建立被控对象模型（电机 + 负载）。
2. 添加 PID 控制器模块。
3. 在 PID 模块中点击"Tune" 按钮，打开 PID 整定器。
4. 调整响应时间和鲁棒性滑块。
5. 将增益导出到你的嵌入式代码中。

在硬件上精调之前拿到一个起始点，很好用。但前提是被控对象模型得准——模型错了，整定出来的参数也是错的。

4.7 系统辨识：搞清楚你的被控对象究竟如何工作

到目前为止，我们假设你知道被控对象模型——传递函数 $G(s)$ 或状态空间矩阵 A、B、C、D。但这些从哪里来？你有两个选择：

1. **第一性原理建模（First-principles Modeling）：**写出物理方程（牛顿定律、基尔霍夫定律等），解析推导模型。第 3 节里对直流有刷电机做的就是这个。物理过程简单、理解充分的时候效果最好。
2. **系统辨识（System Identification，Sysid）：**向真实系统施加已知输入，测量输出，然后拟合数学模型到输入-输出数据。当物理过程过于复杂、未知，或者你想捕捉第一性原理模型忽略的效应（非线性摩擦、缆线刚度、齿轮间隙）时使用。

实际做的时候，几乎总是**两手都用**：第一性原理定结构（" 这是一个带延迟的二阶系统"），系统辨识定参数（K、τ、ζ 等）。

4.7.1 阶跃响应法（时域）

最简单的系统辨识技术。施加阶跃输入并记录输出。

对于一阶系统 $G(s) = \frac{K}{\tau s+1}$：

1. 向电机电压施加幅值为 A 的阶跃。
2. 记录速度响应 $y(t)$。
3. 读取：
    - $K =$ （最终稳态值）/ $A =$ 直流增益
    - $\tau =$ 到达终值 63.2% 所需的时间 = 时间常数

对于二阶系统 $G(s) = \frac{K\omega_n^2}{s^2+2\zeta\omega_n s+\omega_n^2}$：

1. 施加阶跃并记录。

2. 读取：
   - $K =$ （终值）/（阶跃幅值）
   - $M_p =$ 百分比超调量 $\Rightarrow \zeta = \frac{-\ln(M_p/100)}{\sqrt{\pi^2+\ln^2(M_p/100)}}$
   - $t_p =$ 第一个峰值出现的时间 $\Rightarrow \omega_n = \frac{\pi}{t_p\sqrt{1-\zeta^2}}$

实践建议：

- 以传感器支持的最高速率记录数据（电机 1 kHz 以上）。
- 测试运行 3–5 次取平均以减小噪声影响。
- 用不同阶跃幅值测试以检验线性性。若响应形状改变，系统是非线性的。
- 同时记录上升阶跃（0 → V）和下降阶跃（V → 0）。若两者不同，说明存在不对称动态（如正反转摩擦不同）。

4.7.2 频率响应法（Bode 辨识）

在不同频率施加正弦输入，在每个频率测量增益和相移。这直接构造出系统的 Bode 图。

步骤：

1. 向系统施加 $u(t) = A\sin(\omega t)$。
2. 等待瞬态消失（几个周期）。
3. 测量稳态输出幅值 B 和相位滞后 ϕ。
4. 计算增益：$|G(j\omega)| = B/A$，相位：$\angle G(j\omega) = \phi$。
5. 在期望带宽的 0.1× 至 10× 范围内重复上述步骤。

这方法的厉害之处： Bode 图能直接告诉你系统阶次（数斜率变化的段数）、共振频率（幅值图上的峰）和时间延迟（相位线性下降）。有了 Bode 图就可以拟合传递函数。

更快的替代——扫频信号（Chirp）： 一个个频率挨个测太慢了。直接给一个 **chirp**——频率在几秒内从低扫到高。然后对输入输出分别做 FFT，一除：$G(j\omega) = \frac{Y(j\omega)}{U(j\omega)}$。一次实验拿到完整 Bode 图。

4.7.3 最小二乘辨识（自动化）

对于数字系统，系统辨识可以完全自动化。给定输入-输出数据 $\{u[k], y[k]\}$，拟合离散传递函数：

$$y[k] = -a_1 y[k-1] - \cdots - a_n y[k-n] + b_0 u[k-1] + \cdots + b_m u[k-m-1] \tag{45}$$

一个标准的**线性回归**问题。把所有样本堆成矩阵方程 $\mathbf{y} = \Phi\boldsymbol{\theta}$，最小二乘求解：$\boldsymbol{\theta} = (\Phi^T\Phi)^{-1}\Phi^T\mathbf{y}$。

用 Python（3 行代码）：

```
from scipy.signal import dlsim
from scipy.optimize import curve_fit
# 或者使用专用库：
import control
sys_id = control.tf(num, den, dt)  # 拟合到数据
```

```
# MATLAB中更简单:
% data = iddata(y, u, Ts);
% sys = tfest(data, 2);  % 拟合2阶传递函数
% bode(sys);  % 查看结果
```

MATLAB 的 System Identification Toolbox 是这方面的黄金标准。函数 `tfest(data, n)` 可自动将 n 阶传递函数拟合到记录的数据上。从 $n = 1$ 开始，逐步增加直到拟合效果满意（对电机而言通常 $n = 2$ 就足够了）。

4.7.4 竞赛机器人的实用系统辨识流程

1. **开环阶跃测试：**禁用所有控制器。向电机发送已知的电压阶跃。以最高速率记录编码器/IMU 数据。
2. **快速估计：**目测阶跃响应，估算 K 和 τ（一阶）或 K、ζ、ω_n（二阶）。
3. **验证：**用相同的阶跃输入仿真辨识出的模型，将仿真结果叠加到真实数据上。如果吻合，就完成了。
4. **必要时改进：**如果一阶模型不够精确，尝试二阶。如果有明显延迟，添加时间延迟项 $e^{-\tau_d s}$。
5. **用于控制器设计：**将辨识出的模型用于 PID 整定（MATLAB PID 整定器）、LQR 设计（A、B 矩阵）或前馈计算。

常见陷阱：

- **摩擦：**静摩擦（stiction）使系统在低速附近非线性。你的线性模型在速度接近零时会不准确。考虑分别辨识”运动中”和”启动时”的模型。
- **间隙（Backlash）：**齿轮间隙产生死区。系统在齿轮咬合之前不响应小输入。这在阶跃响应中表现为一段平坦区域。
- **饱和：**如果阶跃幅值将电机推入饱和，辨识出的模型将是错误的。使用线性工作范围内的阶跃幅值（通常为最大电压的 30–70%）。
- **耦合：**在多轴系统（云台、底盘）上，移动一个轴可能激励另一个轴。如有可能，锁定其他轴后分别辨识每个轴。

为什么系统辨识很重要：基于 20% 参数误差模型设计的控制器，性能会比基于真实模型的差 20%。5 分钟的数据记录加系统辨识，能省去数小时的手动 PID 整定。这对于高度依赖精确 A、B 矩阵的 LQR 和 MPC 尤为如此。

4.8 前馈控制（Feedforward Control）

反馈是被动的：等误差冒出来才纠正。前馈是**主动的**：提前预判系统需要什么，直接给到位。

示例：你想让云台跟踪一个以恒定速度运动的目标。纯反馈控制下，控制器始终在追赶目标，产生跟踪误差。有了前馈，你计算维持该速度所需的电机电压并加到反馈输出上：

$$u(t) = u_{\text{前馈}}(t) + u_{\text{反馈}}(t) \tag{46}$$

前馈项处理运动的”预期”部分；反馈处理”意外”部分（扰动、模型误差）。

重力补偿（gravity compensation）是机械臂和云台常见的前馈项：

$$u_{\text{ff}} = \tau_{\text{重力}}(\theta) = mgl\sin(\theta) \tag{47}$$

少了这一项，反馈控制器就得把力气浪费在跟重力较劲上，顾不上跟踪参考了。

前馈黄金法则：前馈靠模型吃饭，模型有多准它就有多准。反馈不靠模型，天生鲁棒。**前馈拿来提速，反馈留着兜底。**

案例分析：云台级联 PID——为什么两个回路胜过一个

你想让云台指向某个角度。最朴素的做法：一个 PID 回路直接从角度误差控制到电机电压。能跑，但效果一般。下面说为什么，以及高手怎么做。

单回路控制的问题：

从电压到角度的传递函数是二阶的（电机动态 + 速度对位置的积分）。单个 PID 得同时伺候快速的电流动态和慢悠悠的机械动态，相当于让一个人边开车边导航——不是不行，就是累。

级联（嵌套）回路方案：

用两个串联的 PID 控制器：

1. **内环（速度环）：**以 1 kHz 运行。输入：期望速度。输出：电机电压指令。该回路速度快，能迅速抑制扰动（摩擦、负载变化）。
2. **外环（位置环）：**以 200 Hz 运行。输入：期望角度。输出：期望速度（成为内环的设定值）。该回路处理较慢的位置跟踪。

为什么级联更好：

- 每个回路处理一”层”动态，调整更简单——先调内环（忽略外环），再调外环。
- 内环充当”屏障”——在外环感知到扰动之前，内环就已快速纠正。有人碰了你的云台，速度环在毫秒内就完成了补偿。
- 各层次都有限幅：外环输出的速度指令天然有界，防止突然的大力矩。

加入前馈：

再从轨迹规划器加入速度前馈：

$$\omega_{\text{cmd}} = \underbrace{K_p(\theta_{\text{ref}} - \theta)}_{\text{外环 PID}} + \underbrace{\dot{\theta}_{\text{ref}}}_{\text{前馈}}$$

前馈直接告诉内环”跟上参考轨迹要多快”，外环 PID 只负责补小差。效果：快速运动时也能平滑、精确地跟踪。RoboMaster 强队的云台基本都是这个架构。

级联回路的经验法则：

- 内环应比外环快 5–10 倍。
- 从内到外调整：先调内环，再调外环。
- 每个回路在上一层看来应像一个简单的一阶系统。

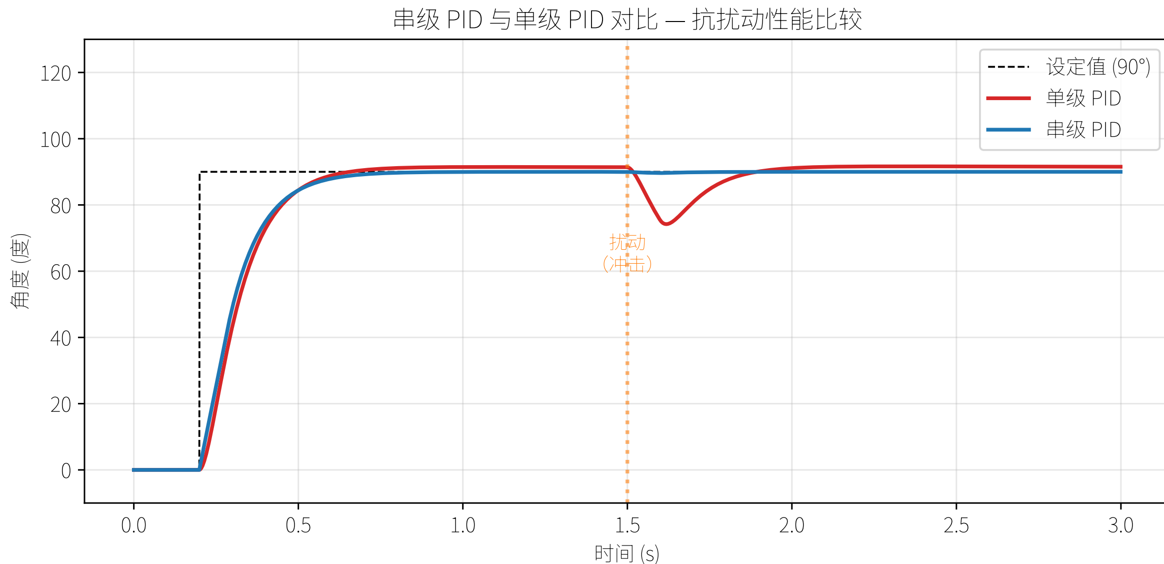

图 13: 级联 PID（蓝色）与单回路 PID（红色）在云台电机上的对比。两者都跟踪 90° 设定值，但当扰动在 $t = 1.5\text{s}$ 发生时，级联架构恢复得更快，因为内速度环承受了冲击。

4.9 PID 架构：单级、串级与并联串级

上述案例分析展示了串级 PID 在单轴上相比单回路 PID 的优势。在实际比赛中，机器人通常需要多轴协调控制（例如云台偏航 + 俯仰，底盘 X + Y + 旋转）。本节系统比较三种 PID 架构，并提供相应的代码实现。

4.9.1 架构一：单级 PID

最简单的架构：每个轴一个 PID 控制器，从位置误差直接输出电机电压。

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)\, d\tau + K_d \frac{de(t)}{dt}, \qquad e(t) = r(t) - y(t)$$

优点： 实现简单，每个轴只需整定一组增益，计算开销最小。

局限： 单个控制器需同时处理快速动态（电气/电流）和慢速动态（机械/位置）。这种耦合使得整定困难——为加快跟踪而增大增益往往会引发振荡。抗扰动能力较弱，因为控制器只有在扰动传播到位置输出后才能做出反应。

4.9.2 架构二：串级（级联）PID

每个轴两个嵌套回路：外环产生速度（或电流）设定值，输入内环。

$$\omega_{\text{cmd}} = \text{PID}_{\text{外环}}\big(\theta_{\text{ref}} - \theta\big), \qquad u = \text{PID}_{\text{内环}}\big(\omega_{\text{cmd}} - \omega\big)$$

优点：

- 每个回路只处理一层动态，使整定变得简单（先内环，后外环）。
- 内环在扰动传递到外环之前便将其抑制，抗扰动性能显著提升。
- 天然的限幅保护：外环输出有界的速度指令，避免突然的大力矩。

设计准则：

- 内环带宽应为外环的 5–10 倍。
- 从外环的视角看，内环应表现为一个简单的一阶系统（即整定良好、响应快、无超调）。
- 由内而外整定：先在外环断开的条件下稳定并优化内环，再闭合外环。

4.9.3 架构三：并联串级 PID（多轴协同）

在多轴系统（如偏航 + 俯仰的云台）中，每个轴各运行一个串级 PID 控制器，各轴并行运行。各轴共享传感器数据，并可选择性地加入轴间耦合补偿。

耦合问题的重要性： 在云台上，俯仰运动会改变重心位置，对偏航轴产生力矩扰动。若不加以补偿，偏航控制器会将其视为未知扰动，反应迟缓。通过加入耦合前馈项，偏航控制器可预判俯仰运动并主动补偿：

$$u_{\text{偏航}} = \text{串级 PID}_{\text{偏航}}(\cdot) + k_{\text{耦合}} \cdot \dot{\theta}_{\text{俯仰}}$$

```
// 含轴间耦合补偿的并联串级 PID
struct GimbalController {
    CascadedPID yaw;
    CascadedPID pitch;
    float cross_yaw_from_pitch = 0.0f;  // 耦合增益

    struct Output { float yaw_voltage; float pitch_voltage; };

    Output update(float yaw_ref, float yaw_pos, float yaw_vel,
                  float pitch_ref, float pitch_pos, float pitch_vel,
                  float yaw_vel_ff = 0, float pitch_vel_ff = 0) {
        // 耦合补偿：俯仰角速度对偏航的扰动
        float cross_comp = cross_yaw_from_pitch * pitch_vel;

        float u_pitch = pitch.update(pitch_ref, pitch_pos,
                                     pitch_vel, pitch_vel_ff);
        float u_yaw   = yaw.update(yaw_ref, yaw_pos,
                                   yaw_vel, yaw_vel_ff)
                      + cross_comp;
        return {u_yaw, u_pitch};
    }

    void reset() { yaw.reset(); pitch.reset(); }
};
```

4.9.4 架构对比总结

	单级 PID	串级 PID	并联串级
每轴回路数	1	2	2
整定难度	中等	低（分层整定）	低（分层整定）
抗扰动能力	慢	快	快
轴间耦合处理	无	无	可补偿
代码复杂度	最低	适中	适中
典型应用	简单执行器	单轴云台	多轴云台

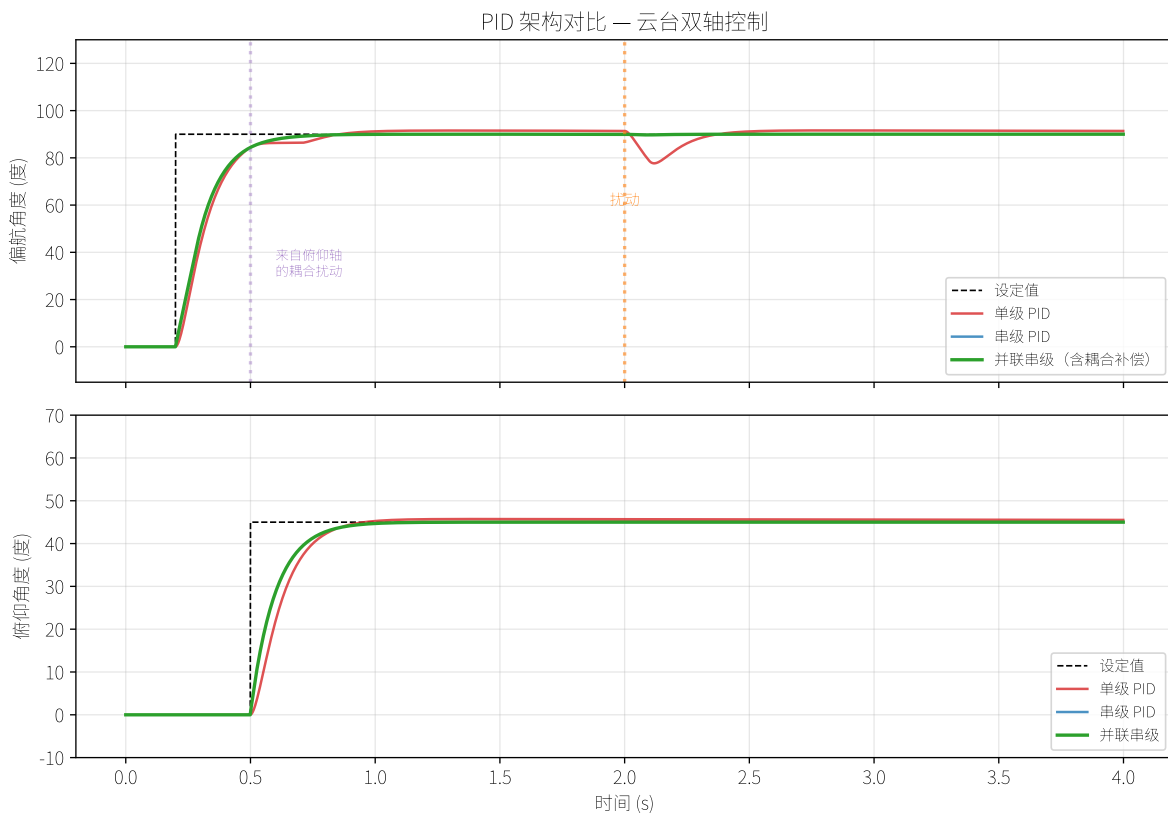

图 14: 双轴云台上的 PID 架构对比。上图：偏航轴，在 $t = 2\text{s}$ 处受到外部扰动，在 $t = 0.5\text{s}$ 处受到来自俯仰运动的耦合扰动。下图：俯仰轴。含耦合补偿的并联串级架构（绿色）在扰动和耦合同时存在时提供了最平稳的偏航响应。

实践建议：对于任何具有多个受控轴的系统，建议采用并联串级架构。额外的代码复杂度极小（仅需一个结构体封装两个 `CascadedPID` 对象），而分层整定和耦合补偿带来的性能提升却十分显著。

实践案例：两轮自平衡车（并联 PID）

两轮自平衡机器人（如 Segway）是经典的并联多环控制问题。至少需要同时控制三个目标：平衡（倾斜角）、速度和偏航（转向）。每个目标各有一个控制器，它们的输出在电机层面相加。

- **平衡环（快）：** 对 IMU 测量的倾斜角 ϕ 进行 PD 控制。这是最关键的环——必须以 $\geq 500\,\text{Hz}$ 运行，毫秒级响应。输出是施加给两个轮子的同向力矩。
- **速度环（中）：** 对轮速（编码器）进行 PI 控制。参考值来自用户操纵杆。输出叠加到平衡力矩上。该环防止机器人在保持平衡的同时漂移。
- **偏航环（慢）：** 对偏航角速度（陀螺仪）进行 P 控制。输出是*差动*力矩——左轮正、右轮负（或反之）——用于转向。

电机指令融合：

$$u_{\text{left}} = \underbrace{\tau_{\text{balance}}}_{\text{PD on }\phi} + \underbrace{\tau_{\text{velocity}}}_{\text{PI on }v} + \underbrace{\tau_{\text{yaw}}}_{\text{P on }\dot{\psi}}$$

$$u_{\text{right}} = \tau_{\text{balance}} + \tau_{\text{velocity}} - \tau_{\text{yaw}}$$

τ_{yaw} 的符号翻转实现了差速转向。每个环独立整定：先调平衡（机器人得先站住才能动），再调速度，最后调偏航。

4.9.5 并联 PID 的输出冲突问题

当多个 PID 控制器并联运行、输出相加到同一个执行器时，它们可能*互相打架*。这是并联架构的根本风险。

平衡车中的冲突。考虑平衡环命令“向前倾以保持直立”，而速度环同时命令“向后倾以减速”。两者在各自的视角下都是正确的，但叠加后的输出相互抵消，给电机留下一个微弱、混乱的指令。最坏情况下，两个环互相振荡：速度环制造了一个倾斜扰动，平衡环纠正它，进而改变了速度，又触发速度环——形成环间极限环振荡。

原因分析： 在串级架构中，外环*只*和内环对话，内环*只*和执行器对话。层级结构从设计上就避免了冲突。而并联架构中，所有环都直接和执行器对话，因此它们的控制目标必须*天然兼容*。当目标不兼容时，叠加输出就会变得混乱。

缓解策略：

1. **带宽分离**。确保每个环工作在不同的频率带：最快的环（平衡）在高频主导，最慢的环（偏航）只在低频起作用。冲突指令在时间尺度上分离，因此不会干扰。
2. **优先级限幅**。对较慢环的输出进行限幅，使其不能压过较快的安全关键环。例如，限制 $|\tau_{\text{velocity}}|$ 使其不超过 $|\tau_{\text{balance}}|$ 的某个比例。
3. **输出额度分配**。为每个环预留固定的执行器输出范围：例如 60% 给平衡，30% 给速度，10% 给偏航。这保证了任何一个环都不会“饿死”另一个环。
4. **转化为串级**。如果冲突持续存在，重构控制器，使速度环输出*倾斜角给定值*给平衡环，而不是直接输出与之竞争的力矩。这从设计上消除了冲突——代价是增加了环间耦合。

经验法则： 当被控变量在*物理上正交*时（例如平衡车上的平衡 vs. 偏航——一个是矢状面，另一个是侧向），并联 PID 效果良好。当两个环试图控制共享同一执行器自由度的变量时（例如平衡和速度都需要矢状面力矩），并联 PID 就会力不从心。

4.9.6 思考题：串级 PID 的内环 K_p 等价于外环 K_d 吗？

考虑一个位置–速度串级系统。外环输出的速度给定值正比于位置误差：$v_{\text{ref}} = K_p^{\text{outer}}(r - y)$。内环跟踪该速度。再看单级位置 PID：微分项产生 $K_d \cdot \dot{e} \approx -K_d \cdot \dot{y}$，这是一个正比于速度的反馈。

它们等价吗？*几乎等价，但不完全*。在稳态线性分析中，外环 K_p 通过理想内环速度回路产生的效果类似于位置误差上的 K_d。具体来说，如果内环增益为 1 且无动态延迟（即瞬时速度跟踪），

则:

$$u = K_p^{\text{inner}}\big(K_p^{\text{outer}}(r - y) - \dot{y}\big)$$

展开后包含类似 $K_p \cdot e + K_d \cdot (-\dot{y})$ 的项，其中 $K_d \approx K_p^{\text{inner}}$。

但串级在实践中更优，因为:

1. 内环有自己的积分项，可以消除速度层面的扰动（摩擦、负载变化），而无需依赖外环。
2. 内环对外环指令进行带宽限制，防止外环要求物理上不可能的加速度。裸露的 K_d 没有这种保护——它放大高频噪声，可能导致执行器饱和。
3. 内环使用的是测量的速度信号，而非数值微分。微分放大噪声；直接测量不会。

结论: 微分项和内环比例增益在概念上是相关的——两者都注入了与速度相关的阻尼。但串级架构提供的噪声抑制、速度层面的扰动抑制以及隐式速率限制，是裸露的 K_d 项无法匹敌的。

5 离散化与实现

纸上（或 MATLAB 里）的连续时间控制器已经设计好了。接下来要把它塞进 STM32 的 1 kHz 定时器中断里跑起来。本节就是在理论和嵌入式现实之间架桥。

5.1 从连续到离散

连续时间传递函数 $C(s)$ 得先转成离散时间的 $C(z)$，才能写进代码。方法有好几种，各有各的取舍。

5.1.1 零阶保持（Zero-Order Hold，ZOH）

零阶保持（ZOH）假设输入在两次采样之间保持不变——DAC 就是这么工作的。物理上最精确的离散化方法。

核心思想： 在采样时刻之间，控制信号保持不变。我们精确计算连续系统在这种分段常数输入下的响应。

对于一阶系统 $G(s) = \frac{a}{s+a}$，采样周期为 T_s：

$$G(z) = \frac{1 - e^{-aT_s}}{z - e^{-aT_s}} \tag{48}$$

适用场景： 当你希望离散系统在采样时刻与连续系统完全吻合时。MATLAB 的 `c2d(sys, Ts, 'zoh')` 使用的就是这种方法。

缺点： 对高阶系统，转换公式可能相当复杂，通常借助软件工具完成，不适合手算。

5.1.2 双线性变换（Tustin Transform，也称 Bilinear Transform）

双线性变换（Tustin 变换）将 s 替换为：

$$s \leftarrow \frac{2}{T_s} \cdot \frac{z-1}{z+1} \tag{49}$$

纯粹的代数代换：把连续传递函数里的 s 全换成上面的表达式，化简就完了。

示例： 一阶低通滤波器 $H(s) = \frac{\omega_c}{s+\omega_c}$ 变换后为：

$$H(z) = \frac{\omega_c T_s (z+1)}{(2 + \omega_c T_s) z + (\omega_c T_s - 2)} \tag{50}$$

双线性变换流行的原因：

- 对低阶系统可以手算，计算简单。
- 保持稳定性：稳定的连续系统映射到稳定的离散系统。
- 低频段的频域匹配效果好。

注意事项： 在高频处（接近奈奎斯特频率），双线性变换会引入**频率翘曲（frequency warping）**——离散滤波器的截止频率相对于连续设计发生偏移。使用**预翘曲（pre-warping）**来修正这一问题：

$$\omega_{\text{pre-warped}} = \frac{2}{T_s} \tan\left(\frac{\omega_c T_s}{2}\right) \tag{51}$$

在连续设计中用 $\omega_{\text{pre-warped}}$ 替换 ω_c，再应用双线性变换。

5.1.3 极零点匹配法（Matched Pole-Zero Method）

该方法使用 $z = e^{sT_s}$ 将连续系统的极点和零点映射到 z 域。它能较好地保留时域响应特性。

适用场景： 当匹配瞬态响应比匹配频率响应更重要时。在实践中不如 ZOH 或双线性变换常用。

小结： 对于大多数竞赛应用，**双线性变换是首选方法**。简单、稳定、精度足够。需要精确的采样时刻匹配时用 ZOH。如果你使用 MATLAB，直接用 `c2d()` 让它处理数学细节即可。

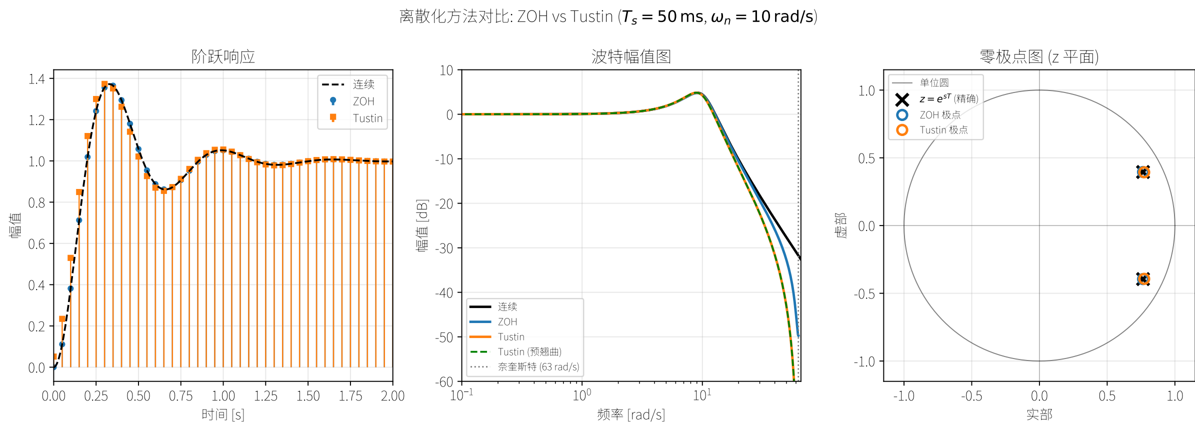

图 15: ZOH 与双线性变换对欠阻尼二阶系统的离散化对比（$\omega_n = 10$ rad/s，$\zeta = 0.3$，$T_s = 50$ ms）。左图：阶跃响应——ZOH 在采样时刻与连续系统完全吻合。中图：Bode 幅频图——双线性变换在奈奎斯特频率附近出现频率翘曲；预翘曲（绿色）修正了这一问题。右图：极零点图，展示了各方法如何将连续极点映射到 z 平面。

5.2 数字滤波器结构：FIR 与 IIR

学会了如何将连续时间设计转换为离散时间传递函数之后，自然而然的下一个问题是：得到的数字滤波器在代码中到底长什么样？有两种基本结构：**FIR** 和 **IIR**。理解二者的区别，对于在单片机上选择正确的实现方式至关重要。

5.2.1 FIR（有限冲激响应，Finite Impulse Response）

FIR 滤波器的输出仅由当前和过去的输入采样的加权和决定——没有反馈：

$$y[n] = \sum_{k=0}^{M} b_k \, x[n-k] \tag{52}$$

z 域表示：

$$H(z) = \sum_{k=0}^{M} b_k \, z^{-k} \tag{53}$$

FIR 滤波器**只有零点**（除了 $z = 0$ 处的平凡极点外没有极点），这带来两个重要性质：

- **永远稳定**——没有反馈就不存在失控振荡的风险。
- **可以实现线性相位**——只要系数对称（$b_k = b_{M-k}$），每个频率分量的延迟相同，信号波形被完美保持。

代价是**滤波器阶数**：要实现陡峭的截止，可能需要 $M = 50$–200 个抽头（tap），这意味着要存储那么多过去的采样值，每个输出采样要做那么多次乘法。

常用 FIR 设计方法：

- **窗函数法（Window Method）**——截断理想冲激响应，乘以窗函数（汉明窗 Hamming、布莱克曼窗 Blackman、凯泽窗 Kaiser）以减少频谱泄漏。
- **频率采样法（Frequency Sampling）**——在 DFT 采样点上指定期望的幅度响应，再做逆变换。
- **Parks–McClellan（Remez）算法**——一种最优算法，最小化最大逼近误差（等纹波设计）。

5.2.2 IIR（无限冲激响应，Infinite Impulse Response）

IIR 滤波器同时使用过去的输入和过去的输出——存在反馈通路：

$$y[n] = \sum_{k=0}^{M} b_k\, x[n-k] \;-\; \sum_{k=1}^{N} a_k\, y[n-k] \tag{54}$$

z 域表示：

$$H(z) = \frac{\displaystyle\sum_{k=0}^{M} b_k\, z^{-k}}{1 + \displaystyle\sum_{k=1}^{N} a_k\, z^{-k}} = \frac{B(z)}{A(z)} \tag{55}$$

IIR 滤波器**既有极点又有零点**。极点使得用很低的阶数就能实现很尖锐的频率选择性（一个 4 阶 IIR 可以超过一个 50 阶 FIR），但也带来风险：

- **可能不稳定**——如果任何极点位于单位圆外（$|z| > 1$），输出将无界增长。
- **非线性相位**——不同频率的延迟不同，信号波形会失真。
- **系数敏感**——在定点单片机上，系数的舍入可能将极点推出单位圆，导致不稳定。

经典 IIR 设计从模拟原型出发，使用我们刚讲过的方法（双线性变换、ZOH 等）映射到离散时间：

- **巴特沃斯（Butterworth）**——通带最大平坦，单调滚降。”香草味”选项；听起来平淡无奇，但 80% 的情况下它就是正确的默认选择。
- **切比雪夫 I 型（Chebyshev Type I）**——通带等纹波，阻带单调。同阶数下比巴特沃斯过渡更陡，代价是通带有纹波。
- **切比雪夫 II 型（Chebyshev Type II）**——通带单调，阻带等纹波。不太常见，但在通带平坦度更重要时很有用。
- **椭圆滤波器（Elliptic / Cauer）**——通带和阻带都是等纹波。在给定阶数下过渡带最窄，但到处都有纹波。

注意：上一小节介绍的双线性变换和冲激不变法，正是将这些模拟原型转换为离散 IIR 滤波器的工具。

5.2.3 FIR vs. IIR：正面对比

特性	FIR	IIR
稳定性	永远稳定	需验证（极点在单位圆内）
相位	可实现线性相位	非线性相位
实现陡峭截止的阶数	高（50–200+）	低（典型 4–10）
每个采样的计算量	$M+1$ 次乘法	$M+N+1$ 次，但 M,N 很小
延迟	较高（滤波器更长）	较低
能否从模拟原型设计	否	能（巴特沃斯、切比雪夫、椭圆）
系数量化敏感度	低	高

5.2.4 实用选择指南与 Biquad 实现

- 需要**线性相位**（音频、对称脉冲整形、通信）？用 **FIR**。
- 需要**陡峭截止且计算量小**（实时控制回路、资源受限的 MCU）？用 **IIR**。
- 需要**保证稳定**且不想仔细分析？用 **FIR**。
- 对于高阶 IIR 滤波器，**一定**要用级联二阶段（biquad）实现，以降低系数敏感度：

$$H(z) = \prod_{i=1}^{L} \frac{b_{0i} + b_{1i}\, z^{-1} + b_{2i}\, z^{-2}}{1 + a_{1i}\, z^{-1} + a_{2i}\, z^{-2}} \tag{56}$$

每一段是一个”biquad”——一个二阶 IIR 滤波器，实现简单、数值稳定。L 个 biquad 级联得到 $2L$ 阶滤波器，同时将系数量化效应限制在每一段内部。

示例：C 语言中的 biquad（转置直接 II 型）

```
// 一个 biquad 段——两个状态变量，数值稳定
float biquad(float x, float *state,
             float b0, float b1, float b2,
             float a1, float a2) {
    float y = b0*x + state[0];
    state[0] = b1*x - a1*y + state[1];
    state[1] = b2*x - a2*y;
    return y;
}
```

快速决策流程

1. 应用对相位敏感吗（例如波形对称性很重要）？→ **FIR**，使用对称系数。
2. 实时计算很紧张吗（MCU 运行快速控制回路）？→ **IIR**（巴特沃斯或切比雪夫），用级联 biquad 实现。
3. 滤波器阶数适中且平台内存充足？→ 两者皆可；默认选 **FIR**，更简单更鲁棒。
4. 需要在给定阶数下实现最陡的过渡带？→ **IIR**，使用椭圆原型。

机器人领域的经验法则：嵌入式系统上大多数传感器平滑滤波器都是低阶 IIR（通常只是

1 阶或 2 阶巴特沃斯）。FIR 滤波器在音频处理、通信以及任何线性相位不可妥协的应用中大放异彩。

5.3 定点与浮点运算

竞赛中常用的微控制器（STM32F1、STM32F4）通常配有硬件浮点单元（FPU）。如果你的芯片有 FPU，**就用浮点**。更简单、更不容易出错，对大多数控制循环来说也足够快。

如果你不得不使用没有 FPU 的微控制器，或者需要追求极致速度，就需要用到**定点运算（fixed-point arithmetic）**。其思路是用缩放整数来表示小数：

$$\text{fixed_value} = \text{float_value} \times 2^Q \tag{57}$$

其中 Q 是小数位数。例如，在 Q16 格式中，数值 3.14 存储为 $3.14 \times 65536 = 205{,}888$。

常见陷阱：

- 溢出：两个 Q16 数相乘会产生 Q32 结果，必须右移回位。
- 范围与精度的取舍：小数位越多，精度越高，但表示范围越小。
- 积分累积：PID 中的积分项在定点运算中很容易溢出。

建议： 除非有充分理由，否则使用 32 位浮点。STM32F4 及以上可以在一个时钟周期内完成单精度浮点运算。

计算速度为何直接决定控制性能

许多初学者认为，只要算法“正确”，控制就能正常工作。这忽略了一个关键事实：*所有控制算法都必须在采样周期 T_s 内完成执行*。如果计算时间超过 T_s，控制器将错过截止时间，下一周期被延迟，等效控制频率降低。最坏情况下，系统可能变得不稳定。

具体示例。假设速度环以 1 kHz 运行（$T_s = 1$ ms），主控为 168 MHz 的 STM32F407。不同算法的计算耗时如下：

算法	复杂度	大致耗时	能否在 1 ms 内完成？
PID（3 次乘加）	$O(1)$	$\sim$1 μs	可以（余量 1000 倍）
二阶 IIR 滤波器	$O(1)$	$\sim$1 μs	可以
卡尔曼滤波器（4×4）	$O(n^3)$，$n = 4$	$\sim$20 μs	可以（余量 50 倍）
LQR（增益矩阵乘法，4×1）	$O(n \cdot m)$	$\sim$2 μs	可以
MPC（$N = 10$，$n = 4$，$m = 2$）	$O(N \cdot n^3)$	$\sim$200–500 μs	勉强
EKF（含三角函数）	$O(n^3) + \text{trig}$	$\sim$50–100 μs	可以，但需注意

计算超时的后果：

- **丢失更新：** 执行器保持上一周期的指令多运行一个周期。系统将此视为增加的时间延迟。回忆淋浴水温的案例（第 4.1 节）：延迟越大，相位裕度越小，系统越接近失稳。
- dt **不一致：** 如果超时是间歇性的（有时快，有时慢），等效 dt 会波动。假定 dt 恒定的积分项和微分项将累积误差。这就是*抖动*（*jitter*），其危害往往大于恒定延迟，因为它不可预测。
- **级联失效：** 在串级架构中，如果内环超时，外环收到的是过期的速度数据。外环的响应品

质下降，抗扰动能力——采用串级控制的核心动机——将大打折扣。

实际诊断方法。在 ISR 的开始和结束处翻转一个 GPIO 引脚，用示波器观察脉宽。若脉宽接近 T_s，说明已处于危险区域。若脉宽曾超过 T_s，则存在必须在调参之前解决的截止时间违规。

计算过慢时的应对措施：

1. **降低算法复杂度**。在约束不活跃时用 LQR 替代 MPC；在非线性不严重时用互补滤波器替代 EKF。
2. **离线预计算**。LQR 增益、三角函数查找表、MPC 增益序列均可在 MATLAB/Python 中计算后存为常量。
3. **降低控制频率**。如果系统带宽允许，将频率从 1 kHz 降至 500 Hz 可使计算预算翻倍。需检查系统时间常数是否允许（回顾：$T_s \leq \tau/10$）。
4. **分层分频**。将耗时的外环以较低频率运行（如 MPC 以 100 Hz），同时保持简单的内环（PID 以 1 kHz）。内环在 MPC 更新间隔内维持控制。

核心认识：最好的算法不是最复杂的算法，而是能在 T_s 内可靠完成且满足性能要求的算法。一个精心调好的 PID 以 1 kHz 运行，永远优于一个在 200 Hz 下超时的 MPC。

5.4 采样率选择

控制循环应该跑多快？太慢则无法响应快速扰动；太快则浪费 CPU 时间（且微分项会放大量化噪声）。

经验法则：采样率应为闭环系统带宽的 10–30 倍。

应用场景	带宽	采样率
底盘速度	∼5 Hz	50–200 Hz
云台位置	∼20 Hz	200–1000 Hz
电机电流环	∼500 Hz	5–20 kHz

实际约束：你的采样率受以下因素限制：

- 传感器更新率（采样速度超过编码器/IMU 的更新速率毫无意义）。
- 计算时间（所有控制运算必须在一个采样周期内完成）。
- 通信延迟（CAN 总线、SPI、I2C 可能成为瓶颈）。

重点：级联控制环（位置 → 速度 → 电流）里，内环要比外环快 5–10 倍。电流环 10 kHz、速度环 1 kHz、位置环 200 Hz，是一种典型配置。

5.5 嵌入式系统实现

5.5.1 定时器与中断配置

在大多数微控制器（STM32 等）上，控制循环运行在**定时器中断**中。定时器以固定周期 T_s 触发，中断服务例程（ISR）执行控制算法。

最佳实践：

- 为每个控制环使用专用硬件定时器。

- 通过预分频器和自动重装寄存器设置所需的定时器周期 T_s。
- ISR **越短越好**：读传感器、计算输出、设置执行器。不要打印、不要 UART、不要动态内存分配。
- ISR 与主循环共享的变量使用 `volatile` 修饰。
- 正确设置中断优先级：电流环 > 速度环 > 位置环。

示例（STM32 HAL）：

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim) {
    if (htim->Instance == TIM6) {  // 1 kHz control loop
        float measurement = read_encoder();
        float error = setpoint - measurement;
        float output = pid_compute(&pid, error);
        set_motor_pwm(output);
    }
}
```

5.5.2 计算延迟与抖动

延迟（Latency）是从读取传感器到施加输出之间的时间。越小越好。超过 $T_s/2$ 时，性能会显著下降。

抖动（Jitter）是延迟在每个周期间的变化量。如果你的控制循环通常耗时 10 μs，但偶尔因被其他中断抢占而耗时 500 μs，行为就会不一致。抖动往往比固定延迟危害更大。

如何减小抖动：

- 给控制定时器分配最高中断优先级。
- 避免长时间全局关闭中断。
- 在任何 ISR 中不要使用阻塞等待（delay、轮询）。
- 对 ISR 执行时间进行剖析（profiling），确保其远小于 T_s。

5.5.3 实践中的积分抗饱和

我们在 PID 章节中讨论过抗饱和（Anti-Windup），但在这里值得特别强调，因为它是真实硬件上 PID “莫名其妙” 异常的头号根源。

实践中何时会发生积分饱和？

- 电机电流达到限制（你在命令 20A，但 ESC 上限为 10A）。
- PWM 饱和（占空比已到 100%，无法再增加）。
- 机械硬限位（云台碰到限位，轮子被堵转）。
- 通信失败（命令发出但未被接收——输出实际上为零）。

实现检查清单：

1. 将积分项限幅到合理范围。
2. 将总输出限幅到执行器的实际限制。
3. 输出饱和时停止积分累积（条件积分）。
4. 切换控制器或模式时，重置或平滑过渡积分项。
5. 明确测试积分饱和恢复：命令一个大阶跃，让其饱和，然后反向。

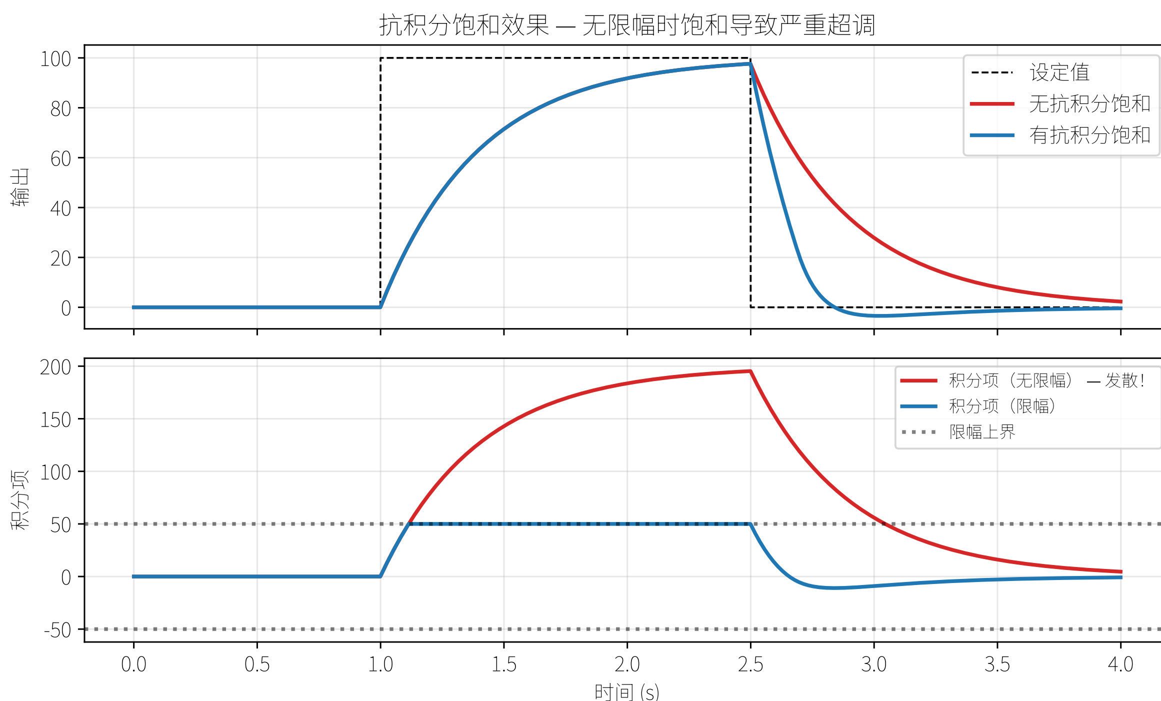

图 16: 无抗饱和时（红色），积分项在饱和期间急剧增大，导致设定值切换后出现巨大超调。加入限幅（蓝色）后，积分项保持有界，响应干净利落。

一个调试故事：你的机器人轮毂电机在从堵转中恢复后剧烈振荡。你检查了 K_p、K_d——看起来没问题。你尝试降低增益——有所改善，但系统变得迟钝。真正的原因呢？积分项在堵转期间累积到了一个巨大的值，现在正在朝相反方向猛烈驱动电机。加入积分限幅后立竿见影。这种事情每个人都会遇到，你也会遇到。现在你知道该看哪里了。

从连续 PID 到 STM32 代码：完整流程

本例演示完整流程：从连续时间 PID 出发，离散化，最终在 STM32 定时器中断中实现。

第一步：连续 PID。教科书上的 PID 传递函数：

$$C(s) = K_p + \frac{K_i}{s} + K_d s$$

取 $K_p = 6$，$K_i = 0.3$，$K_d = 3$，$T_s = 1\,\text{ms}$（1 kHz 控制频率）。

第二步：Tustin 离散化。对每一项施加双线性变换 $s \leftarrow \frac{2}{T_s}\frac{z-1}{z+1}$：

- **积分项：**$\frac{K_i}{s}$ → `integral += Ki * Ts * 0.5 * (e + e_prev)`（梯形法则）
- **微分项：**$K_d s$ → `deriv = Kd * (e - e_prev) / Ts`（后向差分，更好的做法是作用于

测量值）

第三步：STM32 实现。

```
// In TIM6 interrupt handler, every 1ms:
void TIM6_DAC_IRQHandler(void) {
    HAL_TIM_IRQHandler(&htim6);

    float y = read_encoder();        // measured output
    float r = get_setpoint();        // reference
    float e = r - y;

    // Tustin integral (trapezoidal)
    integral += Ki * Ts * 0.5f * (e + e_prev);
    integral = clamp(integral, -IMAX, IMAX);  // anti-windup

    // Derivative on measurement (avoids derivative kick)
    float dy = (y - y_prev) / Ts;

    // PID output
    float u = Kp * e + integral - Kd * dy;
    u = clamp(u, -UMAX, UMAX);

    set_pwm(u);  // actuator output

    e_prev = e;
    y_prev = y;
}
```

关键实现细节：

1. **微分作用于测量值**（$-K_d\dot{y}$），而非误差（$K_d\dot{e}$）。设定值 r 的阶跃变化会使 $\dot{e}$ 产生无穷大尖峰，猛击执行器。使用 $\dot{y}$ 可避免"微分冲击"。
2. **积分限幅**防止执行器饱和时的积分饱和。
3. **输出限幅**确保 PWM 在硬件极限范围内。
4. 在**无 FPU 的 MCU** 上避免浮点运算——改用定点 Q16 或 Q24 算术。
5. **所有状态变量**（`integral`、`e_prev`、`y_prev`）为 `static` 或文件作用域全局变量，在 ISR 调用间持久保存。

6 现代控制理论

经典控制（PID、频率响应、根轨迹）已经能走很远了，大多数 SISO 系统它都能漂亮地搞定。但有些问题它够不着：

- 你的全向底盘有四个电机，必须同时协调控制（MIMO，多输入多输出）。
- 你想要最优性能，而不仅仅是”差不多够用”（最优控制）。
- 你需要估计无法直接测量的量，比如从位置数据中估计速度，或从有噪声的 IMU 中估计倾斜角（状态估计）。
- 你想明确地处理约束条件，如最大电机电流或关节角度限制（约束优化）。

这就是现代控制理论的用武之地。它直接在状态空间上干活，线性代数是它的主要武器。

6.1 状态空间与传递函数的关系

同一个系统的两种写法，可以互相转换。

状态空间转传递函数：

$$G(s) = C(sI - A)^{-1}B + D \tag{58}$$

传递函数转状态空间：转法不唯一——同一个传递函数可以对应多种状态空间实现（可控标准型、可观标准型等）。

关键区别：传递函数看的是输入-输出行为；状态空间看的是内部动力学。两个系统可以传递函数一模一样，内部结构完全不同——这些额外信息只有状态空间能捕捉到。

为什么这很重要：有些内部模态可能不稳定，但传递函数里看不到（零点把它们抵消了）。状态空间能把这些”隐藏炸弹”揪出来。所以现代控制偏爱状态空间——它看得见全貌。

6.2 系统性能指标

6.2.1 可控性与可观性

设计控制器或观测器之前，先问两个根本问题。

可控性（Controllability）：「我能否利用现有输入，将每个状态驱动到任意期望值?」

若系统 (A, B) 是可控的，则**可控性矩阵**（controllability matrix）满秩：

$$\mathcal{C} = \begin{bmatrix} B & AB & A^2B & \cdots & A^{n-1}B \end{bmatrix}, \quad \text{rank}(\mathcal{C}) = n \tag{59}$$

推导：可控性矩阵为什么长这样

$\mathcal{C} = [B \;\; AB \;\; \cdots \;\; A^{n-1}B]$ 不是凭空变出来的，它从状态递推方程一步步推出来。

第一步：逐步展开状态转移。离散时间系统 $\mathbf{x}[k+1] = A\mathbf{x}[k] + B\mathbf{u}[k]$，从 $\mathbf{x}[0] = \mathbf{0}$ 出发往

前推：

$$\mathbf{x}[1] = B\mathbf{u}[0] \tag{60}$$
$$\mathbf{x}[2] = A\mathbf{x}[1] + B\mathbf{u}[1] = AB\mathbf{u}[0] + B\mathbf{u}[1] \tag{61}$$
$$\mathbf{x}[3] = A^2B\mathbf{u}[0] + AB\mathbf{u}[1] + B\mathbf{u}[2] \tag{62}$$
$$\vdots$$
$$\mathbf{x}[n] = A^{n-1}B\mathbf{u}[0] + A^{n-2}B\mathbf{u}[1] + \cdots + B\mathbf{u}[n-1] \tag{63}$$

第二步：写成矩阵形式。把输入序列摞成向量：

$$\mathbf{x}[n] = \begin{bmatrix} A^{n-1}B & A^{n-2}B & \cdots & AB & B \end{bmatrix} \begin{bmatrix} \mathbf{u}[0] \\ \mathbf{u}[1] \\ \vdots \\ \mathbf{u}[n-1] \end{bmatrix} = \mathcal{C} \cdot \mathbf{u}_{\text{seq}} \tag{64}$$

（$\mathcal{C}$ 的列只是换了个顺序，秩不变。）

第三步：秩条件。我们想从零出发到达任意目标 $\mathbf{x}[n] \in \mathbb{R}^n$。方程 $\mathcal{C} \cdot \mathbf{u}_{\text{seq}} = \mathbf{x}[n]$ 对任意右端都有解，当且仅当 $\text{rank}(\mathcal{C}) = n$。这就是可控性判据。

为什么只要 n 列？ Cayley–Hamilton 定理告诉我们，A^n 可以表示为 $I, A, A^2, \ldots, A^{n-1}$ 的线性组合。所以 A^nB 已经落在 $\text{span}\{B, AB, \ldots, A^{n-1}B\}$ 里了，再往后算也不会增加新信息。

连续时间版本： 对 $\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}$，从 $\mathbf{x}(0) = \mathbf{0}$ 出发的可达集合为：

$$\mathbf{x}(t) = \int_0^t e^{A(t-\tau)} B\mathbf{u}(\tau)\, d\tau$$

把矩阵指数 $e^{A(t-\tau)} = \sum_{k=0}^{\infty} \frac{(t-\tau)^k}{k!} A^k$ 展开，再用 Cayley–Hamilton 把 n 阶及以上的项折叠掉，可达集合还是 $\text{span}\{B, AB, \ldots, A^{n-1}B\}$——同一个可控性矩阵。

推导：可观性矩阵为什么长这样

可观性矩阵来自对偶的问题：给定输出序列 $\mathbf{y}[0], \mathbf{y}[1], \ldots$，能否反推初始状态 $\mathbf{x}[0]$？

第一步：写出输出方程。令 $\mathbf{u} = \mathbf{0}$（已知的输入贡献可以事先减掉）。自由响应下：

$$\mathbf{y}[0] = C\mathbf{x}[0] \tag{65}$$
$$\mathbf{y}[1] = C\mathbf{x}[1] = CA\mathbf{x}[0] \tag{66}$$
$$\mathbf{y}[2] = CA^2\mathbf{x}[0] \tag{67}$$
$$\vdots$$
$$\mathbf{y}[n-1] = CA^{n-1}\mathbf{x}[0] \tag{68}$$

第二步：堆成矩阵方程。

$$\begin{bmatrix} \mathbf{y}[0] \\ \mathbf{y}[1] \\ \vdots \\ \mathbf{y}[n-1] \end{bmatrix} = \underbrace{\begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix}}_{\mathcal{O}} \mathbf{x}[0] \tag{69}$$

第三步：秩条件。要从输出序列唯一地恢复 $\mathbf{x}[0]$，方程 $\mathcal{O} \cdot \mathbf{x}[0] = \mathbf{y}_{\text{seq}}$ 必须有唯一解。这要求 $\text{rank}(\mathcal{O}) = n$（列满秩）。

为什么只要 n 行？跟可控性同理——Cayley–Hamilton 使得 CA^n 是 $C, CA, \ldots, CA^{n-1}$ 的线性组合，多测几步也不会提供新信息。

对偶性一目了然：可控性问的是"$\mathcal{C}$ 能否把输入空间映满状态空间"；可观性问的是"$\mathcal{O}$ 能否把状态空间区分成不同的输出"。把一个转置，就得到另一个——这正是对偶性的数学含义：(A, B) 可控 $\Leftrightarrow$ (A^T, B^T) 可观（其中 B^T 扮演 C 的角色）。

物理含义：若某个状态是不可控的，则任何输入都无法影响它。例如，若你的机器人底盘有一个电机坏了，你就在某些方向上失去了可控性。

工作示例：考虑一个简化的二维底盘，状态为 $\mathbf{x} = \begin{bmatrix} v_x \\ v_y \end{bmatrix}$（$x$ 和 y 方向的速度），只有一个朝前的电机：

$$A = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

可控性矩阵：$\mathcal{C} = \begin{bmatrix} B & AB \end{bmatrix} = \begin{bmatrix} 1 & -1 \\ 0 & 0 \end{bmatrix}$。秩为 1，但 $n = 2$。**不可控！**该电机只能驱动 v_x；没有任何输入能改变 v_y。这在物理上很直观：汽车不能横向移动。若添加一个侧向电机（$B = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$），可控性便恢复了。

可观性（Observability）：「我能否从现有输出（测量值）中确定每个状态?」

若系统 (A, C) 是可观的，则**可观性矩阵**（observability matrix）满秩：

$$\mathcal{O} = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix}, \quad \text{rank}(\mathcal{O}) = n \tag{70}$$

物理含义：若某个状态不可观，则无法从测量值中推断出它。例如，若你只测量电机位置而不测量电流，则电流状态可能是不可观的（取决于模型）。

可控性-可观性对偶性：系统 (A, B, C) 可控，当且仅当对偶系统 (A^T, C^T, B^T) 可观。这一优美的对称性意味着控制器设计技术（极点配置）可以直接移植到观测器设计中。

总之：设计控制器之前查可控性，设计观测器之前查可观性。条件不满足，数学再花哨也白搭——得改硬件（加传感器、修执行器）。

6.2.2 闭环系统稳定性

对于具有状态反馈 $\mathbf{u} = -K\mathbf{x}$ 的闭环状态空间系统：

$$\dot{\mathbf{x}} = (A - BK)\mathbf{x} \tag{71}$$

当且仅当 $(A - BK)$ 的所有特征值都具有负实部时，系统稳定。增益矩阵 K 正是我们设计用来将这些特征值配置到期望位置的。

李雅普诺夫稳定性（Lyapunov stability）：一种更通用的稳定性分析工具。若存在一个正定函数 $V(\mathbf{x})$（类似”能量”的函数），且该函数沿轨迹始终递减（$\dot{V} < 0$），则系统稳定。找到这样的 V 就证明了稳定性；找不到并不能证明不稳定（也许只是你没往正确方向找）。

对于线性系统，李雅普诺夫方程为：

$$A^T P + PA = -Q \tag{72}$$

若 $Q > 0$ 且解 $P > 0$，则系统稳定。MATLAB 的 `lyap()` 可直接求解。

6.3 基于状态空间的控制器设计

6.3.1 贝尔曼方程与动态规划：LQR 和 MPC 的核心引擎

在讲 LQR 和 MPC 之前，得先认识它们背后的数学引擎：**动态规划（DP）**和**贝尔曼方程**。如果你打过算法竞赛（USACO、Codeforces、LeetCode），DP 你肯定不陌生——控制理论里的 DP 跟那个是同一个东西，只是换了身衣服。

贝尔曼最优性原理（Bellman Principle of Optimality）：一个最优策略具有如下性质：无论初始状态和初始决策如何，后续决策对于由第一个决策产生的状态而言，必须构成最优策略。

说人话：只要你知道从每个中间状态往后的最优策略，就可以枚举所有选项，挑出”这一步的代价 + 之后最优代价”之和最小的那个——最优的第一步就这么定了。

热身：网格世界最短路径（你已经熟悉的 DP）

考虑一道经典 DP 问题：你在一个 $n \times n$ 的网格上。从 $(0, 0)$ 出发，目标是到达 $(n-1, n-1)$。每个格子 (i, j) 有代价 $c(i, j)$。你只能向右或向下移动。求最小代价路径。

状态：你的位置 (i, j)。

决策（控制）：向右移动或向下移动。

代价函数（cost-to-go）： $V(i, j) =$ 从 (i, j) 到达终点的最小总代价。

贝尔曼方程：

$$V(i, j) = c(i, j) + \min\{V(i + 1, j),\ V(i, j + 1)\}$$

边界条件： $V(n-1, n-1) = c(n-1, n-1)$。

求解：从终点向后填表，直到起点。在每个格子处，最优决策是朝代价函数值更小的那个邻居走。总复杂度：$O(n^2)$。

你从第一道 DP 题就一直在这样做了。现在看看换一套词汇会发生什么：

DP（动态规划）	最优控制	含义
状态 (i,j)	状态 $\mathbf{x}_k$	当前所在位置
决策（右/下）	控制输入 $\mathbf{u}_k$	下一步做什么
转移 $(i,j) \to (i+1,j)$	$\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k)$	物理/动力学规律
步骤代价 $c(i,j)$	阶段代价 $\ell(\mathbf{x}_k, \mathbf{u}_k)$	该步的惩罚
代价函数 $V(i,j)$	值函数 $V_k(\mathbf{x})$	最优未来代价
贝尔曼递推	贝尔曼方程	DP 更新规则
向后填表	向后 Riccati 递推	求解方式
(i,j) 处的最优动作	最优策略 $\mathbf{u}_k^* = \pi(\mathbf{x}_k)$	每个状态该做什么

一模一样的算法。唯一的区别：控制里的「网格」是连续的（状态是实数而不是整数），「决策」是实值的控制输入而不是离散的左/右。

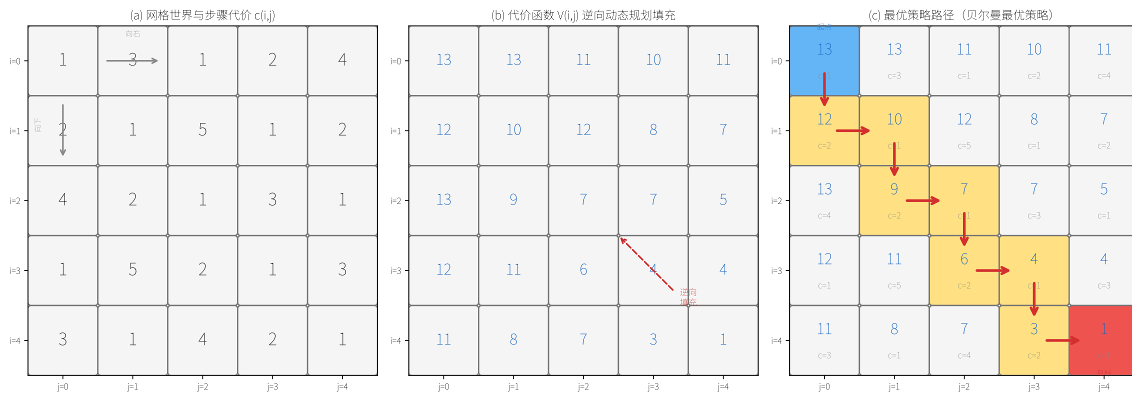

图 17: 网格世界 DP 示例（5×5，移动方式：向右或向下）。(a) 每个格子有步骤代价 $c(i,j)$。(b) 代价函数 $V(i,j)$ 从终点向后填充——这就是 DP 表（等价于控制中的 Riccati 递推）。(c) 最优路径（总代价 $= 13$）遵循贝尔曼策略：在每个格子处，朝 V 值更小的邻居走。将「格子」换成「状态」、「右/下」换成「控制输入」，你就得到了 LQR。

最优控制的贝尔曼方程（离散时间）：

给定系统 $\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k)$，阶段代价 $\ell(\mathbf{x}_k, \mathbf{u}_k)$ 和终端代价 $\ell_f(\mathbf{x}_N)$，最优代价函数（值函数）满足：

$$V_k(\mathbf{x}) = \min_{\mathbf{u}} \left[\ell(\mathbf{x}, \mathbf{u}) + V_{k+1}\left(f(\mathbf{x}, \mathbf{u})\right)\right] \tag{73}$$

边界条件为 $V_N(\mathbf{x}) = \ell_f(\mathbf{x})$。

第 k 步的最优控制为：

$$\mathbf{u}_k^*(\mathbf{x}) = \arg\min_{\mathbf{u}} \left[\ell(\mathbf{x}, \mathbf{u}) + V_{k+1}\left(f(\mathbf{x}, \mathbf{u})\right)\right] \tag{74}$$

这是从时间 $k = N$ 到 $k = 0$ 向后求解的，就像从终点向起点填 DP 表一样。

从网格世界到 LQR：「连续网格」问题

现在想象网格世界变成了连续的。「位置」不再是格子，而是实值状态 $\mathbf{x} \in \mathbb{R}^n$。「移动」不再是右/下，而是实值控制 $\mathbf{u} \in \mathbb{R}^m$。转移是线性的：$\mathbf{x}_{k+1} = A\mathbf{x}_k + B\mathbf{u}_k$。步骤代价是二次的：

$\ell(\mathbf{x},\mathbf{u}) = \mathbf{x}^T Q\mathbf{x} + \mathbf{u}^T R\mathbf{u}$。

代入贝尔曼方程：

$$V_k(\mathbf{x}) = \min_{\mathbf{u}} \left[\mathbf{x}^T Q\mathbf{x} + \mathbf{u}^T R\mathbf{u} + V_{k+1}(A\mathbf{x} + B\mathbf{u})\right]$$

神奇之处来了：由于动力学是线性的、代价是二次的，值函数**也是二次的**：$V_k(\mathbf{x}) = \mathbf{x}^T P_k \mathbf{x}$，其中 P_k 是某个矩阵。将此假设代入并对 $\mathbf{u}$ 求导：

$$\frac{\partial}{\partial \mathbf{u}} \left[\mathbf{u}^T R\mathbf{u} + (A\mathbf{x} + B\mathbf{u})^T P_{k+1}(A\mathbf{x} + B\mathbf{u})\right] = 0$$

求解得：

$$\mathbf{u}_k^* = -\underbrace{(R + B^T P_{k+1} B)^{-1} B^T P_{k+1} A}_{K_k} \mathbf{x}_k \tag{75}$$

$$P_k = Q + A^T P_{k+1} A - A^T P_{k+1} B (R + B^T P_{k+1} B)^{-1} B^T P_{k+1} A \tag{76}$$

P_k 的这个向后递推正是**离散 Riccati 方程**——就是「向后填 DP 表」这一步。当时域 $N \to \infty$ 时，P_k 收敛到常数 P，得到稳态的 **LQR** 增益 $K = (R + B^T PB)^{-1} B^T PA$。

对应网格世界的类比：

- P_k 是代价矩阵——它告诉你从任意状态出发的「剩余代价」，就像网格中的 $V(i,j)$。
- K_k 是第 k 步的最优策略——它告诉你在每个状态该做什么，就像「若 $V(i+1,j) < V(i,j+1)$ 则向右走」。
- Riccati 递推就是填 DP 表——同样是 $O(N)$ 的向后遍历。

与 MPC 的联系： MPC 使用的是同一套贝尔曼/动态规划框架，只是额外增加了两点：

1. 对状态和输入的**约束**（就像在网格世界中加了墙——某些格子被禁止进入）。
2. **滚动时域**：不是从 $k = 0$ 到 N 只求解一次，而是在每个时间步用最新测量值作为新的「起点」重新求解。

约束使贝尔曼方程更难求解（min 变成了约束优化问题），这就是为什么 MPC 需要二次规划（QP）求解器，或 TinyMPC 的 ADMM 方法。但其底层原理完全相同：向后递推求最优代价函数，向前应用最优策略。

与强化学习的联系：如果你熟悉强化学习（RL），贝尔曼方程与 RL 中的**贝尔曼最优性方程**完全一致（$V^*(s) = \max_a[r(s,a) + \gamma V^*(s')]$）。LQR 是线性二次问题的「闭式 RL」。当动力学未知或非线性时，RL 方法（Q-learning、策略梯度）对 LQR 精确计算的解进行近似。

6.3.2 LQR（线性二次调节器）

有了贝尔曼/动态规划这套框架，LQR 就水到渠成了——它就是线性二次贝尔曼方程的稳态解。不用手动配极点，只需给一个代价函数，告诉它” 性能和控制力度之间怎么平衡”：

$$J = \int_0^\infty \left(\mathbf{x}^T Q\mathbf{x} + \mathbf{u}^T R\mathbf{u}\right) dt \tag{77}$$

- Q：惩罚状态偏差（Q 大 = 激进跟踪、快速响应）。
- R：惩罚控制代价（R 大 = 温和输入、节省能量）。

最优状态反馈增益为：

$$K = R^{-1}B^T P \tag{78}$$

其中 P 是**连续代数 Riccati 方程（Continuous Algebraic Riccati Equation，CARE）**的解：

$$A^T P + PA - PBR^{-1}B^T P + Q = 0 \tag{79}$$

不要手动求解这个方程。在 MATLAB 中用 `lqr(A, B, Q, R)`，在 Python 中用 `scipy.linalg.solve_continuo`

Riccati 方程到底在算什么？ P 是**最优代价地图**：P_{ij} 编码了状态 i 和 j 交互带来的「未来代价」。哪个状态难搞（拉回零点要花大力气），P 里对应的元素就大。最优增益 $K = R^{-1}B^T P$ 的意思就是：「代价高的状态猛纠，代价低的状态轻拿轻放。」

LQR 调参： 关键就在 Q 和 R 怎么选。

- 从 $Q = I$（单位矩阵）和 $R = \rho I$（ρ 为标量）开始。
- ρ 小：激进、快速、消耗大量执行器能量。
- ρ 大：保守、缓慢、对执行器温和。
- 若要对某些状态施加更大惩罚，增大 Q 对应的对角元素。例如，若你更关心位置而非速度，令 $Q_{11} > Q_{22}$。

LQR 在比赛中的优势：

- 保证稳定性（如果系统可控）。
- 保证鲁棒性裕度（对 SISO 系统，至少 6 dB 增益裕度和 60° 相位裕度）。
- 轻松处理 MIMO 系统。
- 对多轴系统而言，Q/R 调参比 PID 的三个增益更直观。

示例： 平衡两轮机器人。状态为 $\mathbf{x} = [\theta, \dot{\theta}, x, \dot{x}]^T$（倾斜角、倾斜角速度、位置、速度）。PID 方法需要多个嵌套环路；LQR 只需一个 K 矩阵便可搞定。

局限性： LQR 需要完整的状态反馈（$\mathbf{u} = -K\mathbf{x}$），这意味着你需要知道所有状态。如果无法测量所有状态，则需要观测器（见第 6.4 节）。

6.3.3 优化与凸性简介

MPC 在每个时间步都要解一道优化题。在深入之前，先建立必要的优化概念。

一般优化问题：

$$\min_{\mathbf{x}} \; f(\mathbf{x}) \quad \text{subject to} \quad g_i(\mathbf{x}) \le 0, \;\; h_j(\mathbf{x}) = 0$$

其中 f 是**目标函数**（代价），g_i 是**不等式约束**，h_j 是**等式约束**。优化器在满足所有约束的前提下，寻找使 f 最小的决策变量 $\mathbf{x}$。

什么使问题成为凸问题？满足以下条件：

1. 目标函数 f 是**凸函数**：对任意两点 $\mathbf{a}, \mathbf{b}$ 和 $0 \le \lambda \le 1$，

$$f\big(\lambda\mathbf{a} + (1-\lambda)\mathbf{b}\big) \le \lambda f(\mathbf{a}) + (1-\lambda) f(\mathbf{b})$$

   几何含义：函数曲面位于任意弦的下方。二次函数 $f(\mathbf{x}) = \mathbf{x}^T Q\mathbf{x}$，当 $Q \succeq 0$（半正定）时为凸函数。

2. **可行域**(由约束定义)是凸集:两个可行点之间的任意线段仍在可行域内。线性不等式($A\mathbf{x} \le \mathbf{b}$)始终定义凸集。

为什么凸性重要:凸问题只有一个全局最小值——不存在局部最小值陷阱。任何求解器找到的解就是最优解。非凸问题可能有多个局部最小值,求解器可能陷入较差的解。

二次规划(QP):控制中最常见的优化问题。目标函数是二次的,约束是线性的:

$$\min_{\mathbf{x}} \ \frac{1}{2}\mathbf{x}^T H\mathbf{x} + \mathbf{c}^T\mathbf{x} \quad \text{s.t.} \quad A\mathbf{x} \le \mathbf{b}$$

当 $H \succeq 0$ 时,问题为凸问题,即使在嵌入式系统上也能高效求解(TinyMPC 正是利用了这一结构)。无约束 LQR 是无约束 QP,其解就是 Riccati 方程。带有输入和状态盒式约束的 MPC 是有约束 QP。

与控制的联系:本章涉及的每一个最优控制问题——LQR、MPC、卡尔曼滤波器(LQR 的对偶)——都是凸 QP。这并非巧合:选择线性动力学和二次代价正是因为它们使问题成为凸问题,从而可以实时求解。

6.3.4 MPC(模型预测控制)

MPC 是现代控制工具箱里最强、最灵活的一把。它在每个时间步都解一道优化题:

「当前状态已知,未来 N 步内,什么输入序列能在满足所有约束的前提下把代价压到最低?」

每个时间步执行:

1. 测量(或估计)当前状态 $\mathbf{x}[k]$。
2. 求解优化问题:

$$\min_{\mathbf{u}[k],\ldots,\mathbf{u}[k+N-1]} \sum_{i=0}^{N-1} \left(\mathbf{x}[k+i]^T Q\mathbf{x}[k+i] + \mathbf{u}[k+i]^T R\mathbf{u}[k+i]\right) + \mathbf{x}[k+N]^T P_f \mathbf{x}[k+N] \tag{80}$$

约束条件:

$$\mathbf{x}[k+i+1] = A_d\mathbf{x}[k+i] + B_d\mathbf{u}[k+i] \tag{81}$$

$$\mathbf{u}_{\min} \le \mathbf{u}[k+i] \le \mathbf{u}_{\max} \tag{82}$$

$$\mathbf{x}_{\min} \le \mathbf{x}[k+i] \le \mathbf{x}_{\max} \tag{83}$$

3. 仅施加第一个输入 $\mathbf{u}[k]$。
4. 在下一个时间步重复(滚动时域)。

理解代价函数的各项:

- $\mathbf{x}^T Q\mathbf{x}$:**状态阶段代价**。惩罚每一步偏离期望状态的程度。Q_{ii} 越大,意味着「我越在乎将状态 i 保持在零点(或参考值)附近。」
- $\mathbf{u}^T R\mathbf{u}$:**输入阶段代价**。惩罚控制力度。R 越大,意味着「对执行器温柔一些。」这防止优化器找到数学上最优但物理上剧烈的解。
- $\mathbf{x}[k+N]^T P_f\mathbf{x}[k+N]$:**终端代价**。惩罚预测时域末端的状态。没有这一项,MPC 可能会「作弊」,把所有误差推到时域末尾(就像学生一直拖到最后一天才做事)。将 P_f 设为 LQR Riccati 方程的解是保证稳定性的常见选择。

- N（**预测时域**）：MPC 向前看多少步。时域越长 = 预判能力越强，但计算量越大。太短 = 近视行为（MPC「看不到」即将到来的转弯或障碍物）。一个好的起点：$N = 10$–30 步。

滚动时域： MPC 算出整条最优轨迹，但只走第一步，然后重新规划。看起来很浪费，但恰恰是这一点让 MPC 鲁棒：每一步都拿最新测量值修正模型误差和扰动。就像下棋提前算 10 步，但对手每走一步你就重新评估一次局面。

MPC 为何强大：

- **显式处理约束**。电机电流限制、关节角度限制、速度边界——这些都是优化问题的一部分。
- **自然处理 MIMO 系统**。多个输入和输出只是优化中的更多变量。
- **预瞄能力**。如果你知道未来的参考轨迹（例如规划好的路径），MPC 可以提前看到并主动采取行动。
- **调参直观**。与 LQR 类似，调整 Q 和 R。预测时域 N 是主要的额外参数。

代价： MPC 需要在每个时间步求解优化问题。对于具有二次代价的线性系统（线性 MPC），这是一个二次规划（QP）问题，可以高效求解。对于非线性系统（非线性 MPC），则困难得多。

比赛机器人的实用考量：

- 对于简单系统（底盘速度控制），PID 或 LQR 通常已经足够。MPC 有些大材小用。
- 对于带约束的轨迹跟踪（自主导航、机械臂），MPC 值得投入。
- 像 OSQP、qpOASES 或 acados 这样的库可以快速求解 QP，足以满足嵌入式实时应用。
- 如果实时计算代价太高，可以考虑显式 *MPC*——将解预先离线计算为查找表。

MPC 滚动时域：向前规划，执行一步，重新规划

图 18: MPC 滚动时域在位置跟踪问题上的示意图。在每个时间步，MPC 规划未来 $N = 10$ 步的最优轨迹（绿色），但只施加**第一个**控制动作（圆圈标出）。然后重新测量、重新规划、反复循环。预测窗口随每步向前滑动——这种「向前规划、执行一步、再规划」的循环，正是 MPC 对模型误差和干扰具有鲁棒性的原因。

6.3.5 TinyMPC：面向嵌入式系统的 Riccati 加速 MPC

标准 MPC 在每个时间步都求解一个完整的二次规划（QP）。对于一个有 n 个状态、m 个输入、时域为 N 的系统，这意味着求解一个有 $N(n+m)$ 个决策变量的问题——在笔记本电脑上可

行，但对微控制器来说极为繁重。**TinyMPC**（Nguyen 等，2024）是一个突破性成果，它通过利用 Riccati 方程的结构，使 MPC 能够在资源受限的嵌入式系统（Teensy、STM32，甚至 Arduino）上运行。

核心思路： 对于无约束的线性 MPC，其解就是一个时变的 LQR——可以通过向后 Riccati 递推计算得到。TinyMPC 将这个 Riccati 递推**离线**预计算完成，然后用轻量的 ADMM（交替方向乘子法，Alternating Direction Method of Multipliers）循环**在线**处理约束。

算法结构：

第一步——离线预计算（一次性，或当模型改变时）：

从终端代价 $P_N = Q_f$ 出发，向后运行 Riccati 递推，计算每步的最优反馈增益：

$$S_k = R + B_d^T P_{k+1} B_d \tag{84}$$

$$K_k = S_k^{-1} B_d^T P_{k+1} A_d \tag{85}$$

$$P_k = Q + A_d^T P_{k+1} A_d - A_d^T P_{k+1} B_d K_k \tag{86}$$

这给出 N 个反馈增益矩阵 $K_0, K_1, \ldots, K_{N-1}$ 和 $N+1$ 个代价矩阵 $P_0, P_1, \ldots, P_N$。这些存储在内存中，不再重复计算（除非模型或代价矩阵发生变化）。

第二步——在线求解（每个控制周期）：

给定当前状态 $\mathbf{x}_0$：

1. **前向展开：** 使用预计算的增益计算无约束最优轨迹：$\mathbf{u}_k = -K_k(\mathbf{x}_k - \mathbf{x}_k^{\text{ref}})$，然后 $\mathbf{x}_{k+1} = A_d\mathbf{x}_k + B_d\mathbf{u}_k$。
2. **ADMM 投影：** 若任意 $\mathbf{u}_k$ 或 $\mathbf{x}_k$ 违反约束，运行几次 ADMM 迭代（通常 5–20 次），找到最近的可行解。每次 ADMM 迭代只需：
   - 一次类 Riccati 的向后遍历（使用预计算的 K_k，因此代价低廉）。
   - 一次投影步：$z_k = \text{clamp}(u_k + \lambda_k, u_{\min}, u_{\max})$——逐元素截断。
   - 一次对偶更新：$\lambda_k \leftarrow \lambda_k + u_k - z_k$。
3. **施加第一个输入：** 将 $\mathbf{u}_0$ 发送给执行器。丢弃其余部分（滚动时域）。

为何速度如此之快：

- Riccati 递推（计算量最大的部分）已在**离线**完成。在线工作只是矩阵乘法和截断操作。
- 无需 QP 求解器库。在线无需矩阵分解。只需乘加和截断。
- 每次 ADMM 迭代的复杂度为 $O(N \cdot n^2)$——与时域成线性关系，与状态维度成二次关系。对于 4 状态系统、$N = 10$：每次迭代约 1600 次乘加。10 次迭代共 16000 次乘加——STM32F4 在 168 MHz 下可在 $< 100\,\mu$s 内完成。
- 内存占用：$N \times (n \times n + m \times n)$ 个浮点数用于预计算增益。对于 4 状态、2 输入、$N = 10$：约 240 个浮点数 = 960 字节。

与标准基于 QP 的 MPC 对比：

	标准 MPC（OSQP）	TinyMPC
在线计算	完整 QP 求解	Riccati 展开 + ADMM
典型求解时间（STM32）	1–10 ms	50–200 μs
内存	数 KB（稀疏矩阵）	小型系统 < 1 KB
依赖项	QP 求解器库	无（仅矩阵运算）
约束处理	精确（到容差）	近似（ADMM）
热启动	可行	自然（用上一次解）

何时使用 TinyMPC：

- 需要在 RAM/CPU 有限的微控制器上运行 MPC。
- 模型是线性的（或可以在每步线性化用于非线性系统）。
- 对输入的硬约束很重要（电机电流限制、关节角度限制）。
- 时域较短到中等（$N \leq 30$）。

何时不用 TinyMPC：

- 复杂非线性模型（在功能更强的计算机上使用 acados 或 CasADi）。
- 时域很长（$N > 50$）且有大量状态约束。
- 如果问题无约束——直接用 LQR，它更简单。

参考： TinyMPC 的完整论文和开源实现见 `tinympc.org`。C 实现不超过 1000 行，可在 ARM Cortex-M 微控制器上运行。附录中的 C++ 模块遵循同样的 Riccati + ADMM 结构。

6.4 基于状态空间的观测器设计

LQR（以及所有状态反馈）都要求你知道全部状态。但实际中往往只能测到一部分。**观测器**（Observer，也叫估计器）的作用就是从能拿到的测量值里把完整状态”猜”出来。

6.4.1 龙伯格观测器（Luenberger Observer）

龙伯格观测器的思路很直接：复制一份系统模型，再加一个修正项，用测量误差把估计值往真实状态方向拽。

$$\dot{\hat{\mathbf{x}}} = A\hat{\mathbf{x}} + B\mathbf{u} + L(\mathbf{y} - C\hat{\mathbf{x}}) \tag{87}$$

其中：

- $\hat{\mathbf{x}}$ 是估计状态。
- $\mathbf{y} = C\mathbf{x}$ 是实际测量值。
- L 是观测器增益矩阵（由我们设计）。

估计误差 $\mathbf{e} = \mathbf{x} - \hat{\mathbf{x}}$ 的演化方程为：

$$\dot{\mathbf{e}} = (A - LC)\mathbf{e} \tag{88}$$

若选择 L 使 $(A - LC)$ 的所有特征值都有负实部，估计误差将衰减到零。观测器「追上」真实状态。

设计经验法则：将观测器的极点配置得比控制器极点快 2–5 倍。观测器的收敛速度应比控制器动作快，这样控制器始终有一个良好的状态估计。

分离原理（Separation Principle）：你可以独立设计控制器增益 K 和观测器增益 L。只要每个部分单独稳定，组合的控制器-观测器系统就是稳定的。这是一个优美的理论结果，极大地简化了设计过程。

何时使用龙伯格观测器：当你的模型准确且噪声特性不很清楚时。它简单、确定性强、易于调试。

6.4.2 深刻的对偶性：控制器与观测器互为镜像

这是控制理论里最漂亮的结论之一。搞懂它，你对控制系统的理解会上一个台阶。下面把这种对称结构摆出来。

控制器问题：给定系统 $\dot{\mathbf{x}} = A\mathbf{x}+B\mathbf{u}$, 找到增益 K 使得 $\mathbf{u} = -K\mathbf{x}$ 能让闭环系统 $\dot{\mathbf{x}} = (A-BK)\mathbf{x}$ 稳定且表现良好。

观测器问题：给定系统 $\dot{\mathbf{x}} = A\mathbf{x} + B\mathbf{u}$，$\mathbf{y} = C\mathbf{x}$，找到增益 L 使得估计误差 $\dot{\mathbf{e}} = (A - LC)\mathbf{e}$ 衰减到零。

现在并排对比两个闭环矩阵：

	控制器	**观测器**
闭环矩阵	$A - BK$	$A - LC$
待设计的增益	K	L
必要条件	可控性 (A, B)	可观性 (A, C)
设计目标	配置 $A - BK$ 的特征值	配置 $A - LC$ 的特征值
信息流向	输入 → 状态	测量 → 状态估计
Riccati 方程	$A^TP + PA - PBR^{-1}B^TP + Q = 0$	$AP_o + P_oA^T - P_oC^TR_n^{-1}CP_o + Q_n = 0$

数学对偶性：若将观测器 Riccati 方程转置，令 $A \to A^T$、$C^T \to B$、$Q_n \to Q$、$R_n \to R$，你会得到与控制器 Riccati 方程完全相同的形式。它们字面上是同一个方程，只是矩阵转置了。这意味着：

- 任何设计控制器的算法都可以设计观测器（只需转置输入）。
- 若你能用 `place(A, B, poles)` 为控制器配置极点，就能用 `place(A', C', poles)'` 配置观测器极点——注意转置。
- LQR 控制器（来自控制 Riccati 的 K）有一个对偶：**卡尔曼滤波器（Kalman Filter）**（来自观测器 Riccati 的 L）。LQR 是最优控制器；卡尔曼是最优观测器。它们是对偶问题。

打个比方：控制器是*扬声器*，观测器是*麦克风*。扬声器（控制器）通过输入 B 向系统注入能量，塑造状态轨迹。麦克风（观测器）通过输出 C 监听系统，重建内部状况。可控性问的是：”扬声器能覆盖房间的每个角落吗？”可观性问的是：”麦克风能听到房间的每个角落吗？”同一个房间，对称的两个问题。

实际价值：你调状态反馈控制器时积累的直觉——极点越快响应越快但越怕噪声，极点越慢估计越稳但跟踪越慢——设计观测器时*完全通用*。你在拧同一个旋钮，只不过拧的是镜子的另一面。

6.4.3 卡尔曼滤波器（Kalman Filter）

卡尔曼滤波器是线性高斯系统下的最优观测器，可以看作龙伯格观测器的” 概率升级版”——不用手动配极点，而是根据噪声统计特性自动算出最优增益 L。

带噪声的系统模型：

$$\mathbf{x}[k+1] = A_d\mathbf{x}[k] + B_d\mathbf{u}[k] + \mathbf{w}[k], \quad \mathbf{w} \sim \mathcal{N}(0, Q_n) \tag{89}$$

$$\mathbf{y}[k] = C\mathbf{x}[k] + \mathbf{v}[k], \quad \mathbf{v} \sim \mathcal{N}(0, R_n) \tag{90}$$

其中 $\mathbf{w}$ 是过程噪声（模型不确定性），$\mathbf{v}$ 是测量噪声。Q_n 和 R_n 是它们各自的协方差矩阵。

卡尔曼滤波器算法（离散时间）：

预测（*Predict*）：

$$\hat{\mathbf{x}}[k|k-1] = A_d\hat{\mathbf{x}}[k-1|k-1] + B_d\mathbf{u}[k-1] \tag{91}$$

$$P[k|k-1] = A_dP[k-1|k-1]A_d^T + Q_n \tag{92}$$

更新（*Update*）：

$$K_k = P[k|k-1]C^T(CP[k|k-1]C^T + R_n)^{-1} \tag{93}$$

$$\hat{\mathbf{x}}[k|k] = \hat{\mathbf{x}}[k|k-1] + K_k(\mathbf{y}[k] - C\hat{\mathbf{x}}[k|k-1]) \tag{94}$$

$$P[k|k] = (I - K_kC)P[k|k-1] \tag{95}$$

直觉：卡尔曼增益 K_k 在模型预测和测量之间自动分配信任度：

- 若过程噪声大（Q_n 大）而测量噪声小（R_n 小），滤波器更信任测量值。
- 若测量噪声大而过程噪声小，滤波器更信任模型。

卡尔曼滤波器调参归结于选择 Q_n 和 R_n：

- R_n 通常可以直接测量（在系统静止时记录传感器数据，计算方差）。
- Q_n 更难确定——它代表模型不确定性。从较小的值开始，若滤波器太迟钝（跟踪不够快），则增大。
- 一个常用技巧：将 Q_n 和 R_n 当作调节旋钮。增大 Q_n/R_n 的比值以获得更快的跟踪（但噪声更多）。减小以获得更平滑的估计（但有更多滞后）。听起来很熟悉？这与低通滤波器截止频率的权衡相同，只是用概率语言表达。

为什么机器人圈子到处都是卡尔曼：

- **传感器融合：**将 IMU、编码器和视觉数据融合成一个连贯的状态估计。
- **速度估计：**从有噪声的位置测量中获得平滑的速度，而不使用纯微分器。
- **在明确意义下最优：**对于线性高斯系统，它最小化均方估计误差。

实用说明：卡尔曼滤波器每步需要一次矩阵求逆（K_k 的计算）。对于小状态维度（$n \leq 6$），这微不足道。对于更大的系统，使用高效实现（如 Cholesky 分解）。在 STM32F4 上，一个 6 状态的 EKF 可以在 1 kHz 下舒适地运行。

6.4.4 扩展卡尔曼滤波器（EKF）

标准卡尔曼滤波器假设线性动力学和线性测量。但真实机器人系统是**非线性的**：欧拉角和角速度之间是三角函数关系，加速度计的测量模型跟倾斜角的正弦余弦有关，等等。

扩展卡尔曼滤波器（EKF）的处理方式很粗暴也很有效：每一步都在当前估计值附近把系统线性化，当成线性的来跑标准卡尔曼，然后在新估计值处重新线性化。

非线性系统模型：

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k, \mathbf{u}_k) + \mathbf{w}_k \quad \text{（非线性动力学）} \tag{96}$$

$$\mathbf{y}_k = h(\mathbf{x}_k) + \mathbf{v}_k \quad \text{（非线性测量）} \tag{97}$$

EKF 算法：

预测：

$$\hat{\mathbf{x}}_{k|k-1} = f(\hat{\mathbf{x}}_{k-1|k-1}, \mathbf{u}_{k-1}) \tag{98}$$

$$F_k = \left.\frac{\partial f}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_{k-1|k-1}} \quad \text{（动力学雅可比矩阵）} \tag{99}$$

$$P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q \tag{100}$$

更新：

$$H_k = \left.\frac{\partial h}{\partial \mathbf{x}}\right|_{\hat{\mathbf{x}}_{k|k-1}} \quad \text{（测量雅可比矩阵）} \tag{101}$$

$$K_k = P_{k|k-1} H_k^T (H_k P_{k|k-1} H_k^T + R)^{-1} \tag{102}$$

$$\hat{\mathbf{x}}_{k|k} = \hat{\mathbf{x}}_{k|k-1} + K_k \left(\mathbf{y}_k - h(\hat{\mathbf{x}}_{k|k-1})\right) \tag{103}$$

$$P_{k|k} = (I - K_k H_k) P_{k|k-1} \tag{104}$$

与标准卡尔曼滤波器的唯一区别：

1. 预测使用非线性函数 $f(\cdot)$ 代替矩阵 A。
2. 测量残差使用非线性函数 $h(\cdot)$ 代替矩阵 C。
3. 协方差传播使用（在当前估计处求值的）雅可比矩阵 F_k 和 H_k，代替常数矩阵 A 和 C。

其余所有部分——卡尔曼增益公式、协方差更新、预测-更新循环——都完全相同。

案例研究：基于 EKF 的 6-DOF IMU 位姿估计

比赛机器人里最常见的 EKF 应用：用六轴 IMU（三轴加速度计 + 三轴陀螺仪）估计机器人的三维姿态（横滚、俯仰、偏航）。

状态向量： $\mathbf{x} = [\phi, \theta, \psi, b_x, b_y, b_z]^T$

- ϕ：横滚，θ：俯仰，ψ：偏航（弧度制欧拉角）
- b_x, b_y, b_z：陀螺仪偏差（rad/s）——EKF 自动估计这些偏差

动力学模型 $f(\mathbf{x},\mathbf{u})$**：** 经过偏差补偿后的陀螺仪角速度通过运动学方程驱动欧拉角：

$$\omega_x = \text{gyro}_x - b_x, \quad \omega_y = \text{gyro}_y - b_y, \quad \omega_z = \text{gyro}_z - b_z \tag{105}$$

$$\dot{\phi} = \omega_x + \sin\phi\tan\theta\cdot\omega_y + \cos\phi\tan\theta\cdot\omega_z \tag{106}$$

$$\dot{\theta} = \cos\phi\cdot\omega_y - \sin\phi\cdot\omega_z \tag{107}$$

$$\dot{\psi} = \frac{\sin\phi}{\cos\theta}\cdot\omega_y + \frac{\cos\phi}{\cos\theta}\cdot\omega_z \tag{108}$$

$$\dot{b}_x = 0, \quad \dot{b}_y = 0, \quad \dot{b}_z = 0 \quad \text{（偏差建模为随机游走）} \tag{109}$$

雅可比矩阵 $F = \frac{\partial f}{\partial \mathbf{x}}$ 是一个含三角函数元素的 6×6 矩阵，必须在每个时间步重新计算（这就是「扩展」的部分）。

测量模型 $h(\mathbf{x})$**：** 加速度计测量重力方向，从而给出横滚和俯仰（但**不能**给出偏航）：

$$h(\mathbf{x}) = \begin{bmatrix}\phi\\ \theta\end{bmatrix} = \begin{bmatrix}\arctan 2(a_y, a_z)\\ \arctan 2(-a_x, \sqrt{a_y^2 + a_z^2})\end{bmatrix} \tag{110}$$

雅可比矩阵很简单：$H = \begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 & 0 & 0\end{bmatrix}$。

关键设计决策：

- **偏航不可观测**——单靠六轴 IMU，重力无法告诉你哪个方向是北。偏航完全依赖陀螺仪。要纠正偏航漂移，需要添加磁力计（九轴 IMU）或外部航向参考。
- Q **矩阵调参：** 角度对应的对角元素（Q_{00}, Q_{11}, Q_{22}）代表每步陀螺仪积分的漂移程度。越大 = 越不信任陀螺仪。偏差项（Q_{33}, Q_{44}, Q_{55}）代表偏差随机游走速率——通常很小（10^{-4} 到 10^{-6}）。
- R **矩阵调参：** 代表加速度计噪声。在振动或线性加速期间，加速度计读数会被污染。在高加速度机动时增大 R（自适应 EKF），避免信任不良的加速度计数据。
- **万向锁（Gimbal Lock）：** 当俯仰角 $= \pm 90^\circ$ 时，运动学方程中的 $\tan\theta$ 和 $1/\cos\theta$ 项趋于无穷大。对于可能倾斜超过 $\pm 60^\circ$ 的机器人，使用**四元数表示**代替欧拉角（EKF 结构相同，状态参数化不同）。

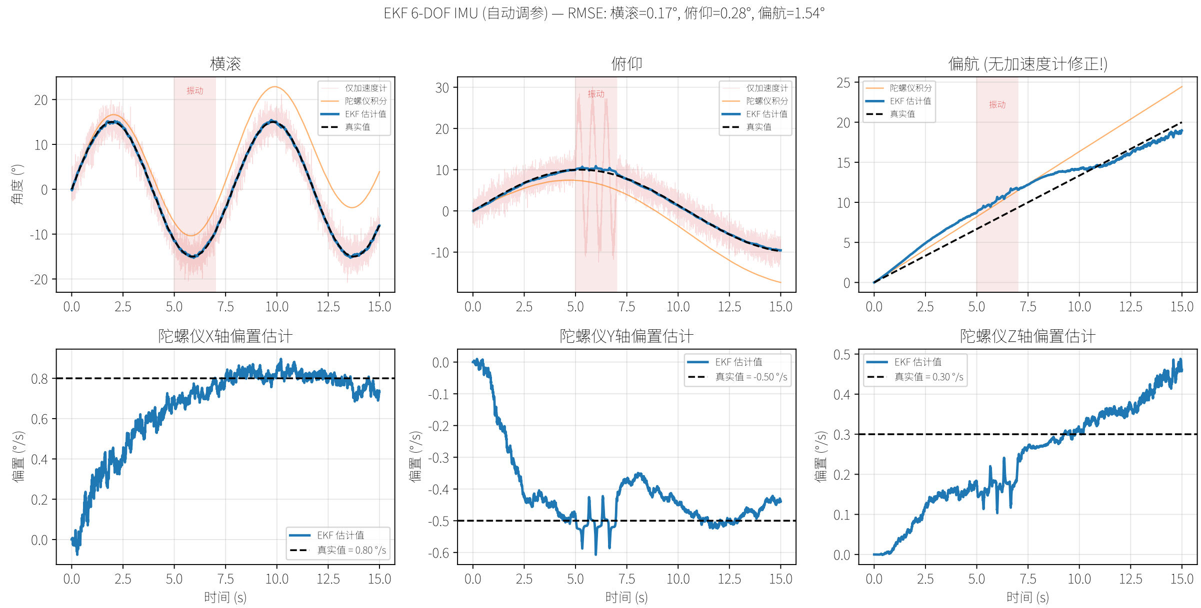

图 19: EKF 6-DOF IMU 位姿估计，配合自动调参的连续自适应 R。使用平滑指数缩放 $R = R_{\text{base}} \cdot (1 + \alpha\, e^{\beta \cdot \Delta a})$，横滚和俯仰分别有独立的 R_{base}，并加入俯仰专用的 x 轴加速度偏差检测。参数通过离线网格搜索确定，最小化加权 RMSE。顶行：横滚（RMSE 0.17°）、俯仰（0.28°）、偏航（1.54°）。振动期间（$t = 5$–7s，阴影区域），自适应 R 指数增大，拒绝被污染的加速度计数据。偏航漂移仅靠偏差估计得以限制（无航向参考）。底行：陀螺仪偏差平稳收敛至真实值。

EKF 之外：当状态不确定性较大或非线性较强时，EKF 的线性化可能不准确。替代方案：

- **无迹卡尔曼滤波器（Unscented Kalman Filter，UKF）：**不进行线性化，而是将「sigma 点」（一组精心选取的样本状态）通过非线性函数传播。对高度非线性系统比 EKF 更准确，但略微昂贵（每步需要 $2n + 1$ 次函数求值，n 为状态维度）。
- **误差状态卡尔曼滤波器（Error-State Kalman Filter，ESKF）：**估计围绕名义轨迹的小误差，而不是完整状态。在基于 IMU 的系统中很流行，因为误差状态保持较小（线性化准确），且避免了四元数奇异性。这是大多数生产级 IMU 融合库内部所使用的方法。
- **互补滤波器（Complementary Filter）：**对于计算资源极其有限的简单横滚/俯仰估计，互补滤波器（第 2 节）仍然是一个可行的单行代码替代方案。

6.4.5 基于四元数的 EKF：避免万向锁

前面的欧拉角 EKF 在俯仰角 ±60° 以内跑得挺好。但俯仰角接近 ±90° 时，运动学方程直接炸了：$\tan\theta \to \infty$，$1/\cos\theta \to \infty$。这就是**万向锁（Gimbal Lock）**——数学上的奇异性，不是物理上的。机器人好好的，是你的公式坏了。

解决方法是用**单位四元数** $\mathbf{q} = \begin{bmatrix} q_w & q_x & q_y & q_z \end{bmatrix}^T$ 代替欧拉角来表示姿态。四元数没有奇异性，结构紧凑（4 个数，相比旋转矩阵的 9 个），并且可以高效地合成旋转。

状态向量： $\mathbf{x} = \begin{bmatrix} \mathbf{q}^T & \boldsymbol{\beta}^T \end{bmatrix}^T \in \mathbb{R}^7$，其中 $\boldsymbol{\beta} = \begin{bmatrix} \beta_x & \beta_y & \beta_z \end{bmatrix}^T$ 为陀螺仪偏差。

预测步骤：给定陀螺仪测量值 $\boldsymbol{\omega}_m$，经偏差修正后的角速度为 $\boldsymbol{\omega} = \boldsymbol{\omega}_m - \boldsymbol{\beta}$。四元数运动学方程为：

$$\dot{\mathbf{q}} = \frac{1}{2}\mathbf{q} \otimes \begin{bmatrix} 0 \\ \boldsymbol{\omega} \end{bmatrix} = \frac{1}{2}\Omega(\boldsymbol{\omega})\,\mathbf{q} \tag{111}$$

其中 $\otimes$ 是四元数乘法，$\Omega(\boldsymbol{\omega})$ 是 4×4 斜对称矩阵：

$$\Omega(\boldsymbol{\omega}) = \begin{bmatrix} 0 & -\omega_x & -\omega_y & -\omega_z \\ \omega_x & 0 & \omega_z & -\omega_y \\ \omega_y & -\omega_z & 0 & \omega_x \\ \omega_z & \omega_y & -\omega_x & 0 \end{bmatrix} \tag{112}$$

离散时间预测（一阶近似）：

$$\mathbf{q}_{k+1} = \left(I + \frac{\Delta t}{2}\Omega(\boldsymbol{\omega}_k)\right)\mathbf{q}_k, \qquad \boldsymbol{\beta}_{k+1} = \boldsymbol{\beta}_k \tag{113}$$

预测后，将 $\mathbf{q}$ **归一化**为单位长度：$\mathbf{q} \leftarrow \mathbf{q}/\|\mathbf{q}\|$。

测量模型：加速度计测量体坐标系中的重力方向。给定四元数 $\mathbf{q}$，体坐标系中预测的重力方向为：

$$\mathbf{g}_{\text{pred}} = R(\mathbf{q})^T \begin{bmatrix} 0 \\ 0 \\ g \end{bmatrix} = g \begin{bmatrix} 2(q_x q_z - q_w q_y) \\ 2(q_y q_z + q_w q_x) \\ q_w^2 - q_x^2 - q_y^2 + q_z^2 \end{bmatrix} \tag{114}$$

测量雅可比矩阵 $H = \partial \mathbf{g}_{\text{pred}}/\partial \mathbf{x}$ 是一个 3×7 矩阵（3 个加速度计分量，4 个四元数 + 3 个偏差状态）。偏差列为零，因为加速度计读数与陀螺仪偏差不直接相关。

更新步骤：标准 EKF 更新照常进行。残差为 $\boldsymbol{\nu} = \mathbf{a}_{\text{meas}} - \mathbf{g}_{\text{pred}}$，更新后再次将 $\mathbf{q}$ 归一化。

欧拉角与四元数：何时用哪个

- **欧拉角**：若你的机器人在俯仰角 $\pm60°$ 范围内工作（地面车辆、大多数炮塔、正常飞行的大多数无人机），欧拉角完全可行。数学更直观，也更容易调试。
- **四元数**：对于可能翻转、翻滚或垂直指向的机器人（特技无人机、水下 ROV、机械臂腕部附近的奇异点），四元数是必要的。
- **提示：**你可以在内部使用四元数进行所有计算，仅在显示和人类可读的日志中转换为欧拉角：$\phi = \text{atan2}(2(q_w q_x + q_y q_z),\ 1 - 2(q_x^2 + q_y^2))$，等等。
- **误差状态方法（ESKF）**避免在 EKF 状态中直接使用 4 元素四元数。它改为估计 3 元素旋转误差 $\delta\boldsymbol{\theta}$，并在每次更新后将其应用到四元数上。这使协方差矩阵 P 保持 6×6 而非 7×7，并避免了单位范数约束问题。大多数生产级 IMU 库（如 PX4 的 EKF2、VINS-Mono）内部使用 ESKF。

6.4.6 自适应 EKF（Adaptive EKF）：在线自动调整 Q 和 R

EKF 效果好不好，很大程度上取决于噪声协方差 Q（过程噪声）和 R（测量噪声）。实际中这两个矩阵通常靠手调——一个磨人的试错过程。更麻烦的是，真实噪声在运行中会变：装在振动底盘上的 IMU 和静止时的噪声特性根本不一回事。固定的 Q、R 应付不了这种情况。

自适应 EKF（Adaptive EKF）方法能够在线估计或调整 Q 和 R，自动进行。以下是几种方法，从简单到复杂。

方法一：基于残差的自适应估计（Innovation-Based Adaptive Estimation，IAE）

最简单也最实用的一种方法。核心思路：**残差序列** $\boldsymbol{\nu}_k = \mathbf{y}_k - h(\hat{\mathbf{x}}_{k|k-1})$（测量值减去预测值）里就藏着 Q 和 R 到底对不对的信息。

判断依据：滤波器调对了的话，残差应该是均值为零的白噪声，协方差 $S_k = H_k P_{k|k-1} H_k^T + R$。残差老是比预期大？要么 Q 太小（太信模型了），要么 R 太小（太信传感器了）。

算法： 使用最近 N_w 个残差的滑动窗口估计实际残差协方差：

$$\hat{S}_k = \frac{1}{N_w} \sum_{j=k-N_w+1}^{k} \boldsymbol{\nu}_j \boldsymbol{\nu}_j^T \tag{115}$$

然后更新 R 使其匹配：

$$\hat{R}_k = \hat{S}_k - H_k P_{k|k-1} H_k^T \tag{116}$$

若 $\hat{R}_k$ 出现负特征值(在 N_w 较小时可能发生),将其截断到最小值。典型窗口大小：$N_w = 20\text{–}100$ 个样本。

代码：

```
// After computing innovation nu = z - h(x_pred):
// Accumulate outer product in sliding window
innov_buffer[buf_idx] = nu;
buf_idx = (buf_idx + 1) % N_WINDOW;

if (samples_seen >= N_WINDOW) {
    // Estimate innovation covariance
    Mat<NZ,NZ> S_hat = Mat<NZ,NZ>::zeros();
    for (int j = 0; j < N_WINDOW; j++) {
        S_hat = S_hat + innov_buffer[j] * innov_buffer[j].T();
    }
    S_hat = S_hat * (1.0f / N_WINDOW);

    // Adapt R
    R = S_hat - H * P_pred * H.T();
    // Clamp diagonal to minimum (prevent negative/zero variance)
    for (int i = 0; i < NZ; i++)
        if (R(i,i) < R_min) R(i,i) = R_min;
}
```

方法二：基于状态残差的 Q 自适应

IAE 方法用于自适应 R，而过程噪声 Q 可以使用**状态残差**（预测状态与更新状态之差）来自适应：

$$\hat{Q}_k = \frac{1}{N_w} \sum_j (K_j \boldsymbol{\nu}_j)(K_j \boldsymbol{\nu}_j)^T + P_{k|k} - F_k P_{k-1|k-1} F_k^T \tag{117}$$

这比 IAE 更复杂，数值稳定性也更差。在实践中，许多系统只自适应 R 而保持 Q 固定，因为 R 变化更大（传感器噪声随环境变化），而 Q 主要由模型不确定性决定（更为恒定）。

方法三：衰减记忆（指数遗忘）

最简单的「自适应」技巧：将预测协方差乘以**遗忘因子** $\alpha > 1$：

$$P_{k|k-1} = \alpha \cdot (F_k P_{k-1|k-1} F_k^T + Q) \tag{118}$$

这在每步人为地增大不确定性，使滤波器「忘记」旧数据，更信任新测量值。这等效于在不显式改变 Q 的情况下让 Q 变大。

- $\alpha = 1.0$：标准 EKF（不遗忘）。
- $\alpha = 1.01$–1.05：轻度自适应——滤波器对突然变化反应更快。
- $\alpha > 1.1$：非常激进——滤波器几乎忽略模型，跟着测量值走。仅在已知扰动期间使用。

何时使用： 当你预期会有偶发的突然模型变化时（例如，机械臂拾取了未知负载——惯性瞬间改变）。衰减记忆防止滤波器「锁定」在过时的动力学上。

代码（在标准 EKF 中增加一行）：

```
P_pred = F * P * F.T() + Q;
P_pred = P_pred * alpha;   // <-- fading memory, alpha = 1.01
```

方法四：多模式自适应 EKF（适用于 IMU）

对于比赛机器人，最实用的方法是简单的**模式切换**策略，而不是连续自适应：

```
// Compute acceleration magnitude
float accel_mag = sqrt(ax*ax + ay*ay + az*az);
float accel_err = fabs(accel_mag - 9.81f);


if (accel_err < 0.5f) {
    // Stationary or slow motion: trust accelerometer
    ekf.R(0,0) = 0.05f;  ekf.R(1,1) = 0.05f;
} else if (accel_err < 2.0f) {
    // Moderate motion: reduce trust
    ekf.R(0,0) = 0.5f;   ekf.R(1,1) = 0.5f;
} else {
    // High acceleration / vibration: mostly ignore accelerometer
    ekf.R(0,0) = 10.0f;  ekf.R(1,1) = 10.0f;
}
```

逻辑： 当 $|\mathbf{a}| \approx g$（$9.81\ \mathrm{m/s^2}$）时，机器人没有线性加速度，加速度计确实在测量重力——信任它（R 小）。当 $|\mathbf{a}|$ 偏离 g 时，存在线性加速度或振动污染读数——不信任它（R 大）。

数学上不优雅？无所谓。实战中**极其好使**。不少冠军机器人用的就是这套。鲁棒、好调（就三个阈值），计算量几乎为零。

总结：用哪种方法？

方法	**自适应对象**	**复杂度**	**最适用场景**
多模式切换	R（离散）	极简	移动机器人上的 IMU
衰减记忆	等效 Q	一行代码	突发模型变化
IAE（基于残差）	R（连续）	中等	传感器噪声变化
基于状态残差	Q 和 R	高	科研/离线场合

一句警告： 若实现不当，自适应方法可能使滤波器发散。务必将自适应后的 Q 和 R 截断到合理范围，使用最小窗口大小（$N_w \geq 20$），并在部署到硬件之前在记录的数据上进行验证。

6.4.7 龙伯格观测器与卡尔曼滤波器：全面对比

两种观测器都学了，放在一起比一下，看看什么场景用哪个。

方面	龙伯格观测器	卡尔曼滤波器
设计输入	期望极点位置（你来选择）	噪声协方差 Q_n、R_n（来自物理或调参）
增益计算	一次性极点配置	递推（每步更新 P 和 K_k）或稳态
最优性	非最优——你选择「合理」的极点	最优（对线性高斯系统最小化均方误差）
噪声处理	无显式噪声模型——增益固定	显式建模过程噪声和测量噪声
自适应性	增益 L 固定——在所有条件下相同	增益 K_k 随不确定性 P 的演化而自适应
计算代价	极低（仅矩阵乘法）	较高（每步矩阵乘法 + 求逆）
调参难度	中等（选择极点位置）	中等（选择 Q_n、R_n）
置信度输出	无	P 矩阵告诉你估计的置信程度
传感器融合	可行但需手动	自然——只需扩展 C 矩阵

它们之间的深层联系：到了稳态（$k \to \infty$），卡尔曼增益 K_k 收敛到常数 K_∞。这时候卡尔曼滤波器就是一个龙伯格观测器，取 $L = K_\infty$。差别在于：卡尔曼是从噪声统计特性里最优地算出这个 L 的，而龙伯格需要你凭直觉或极点配置来选。

再深挖一层：稳态卡尔曼增益 K_∞ 是观测器 Riccati 方程的解——恰好是 LQR 那个控制器 Riccati 方程的对偶。对称性到此闭合：

	控制（输入）侧	估计（输出）侧
手动设计	配置 K 的极点	龙伯格：配置 L 的极点
最优设计	LQR：最小化 $\int(\mathbf{x}^TQ\mathbf{x} + \mathbf{u}^TR\mathbf{u})\,dt$	卡尔曼：最小化 $E[\|\mathbf{x} - \hat{\mathbf{x}}\|^2]$
Riccati 变量	P（控制代价函数）	P（估计不确定性）
调参旋钮	Q 对比 R（状态惩罚对比控制代价）	Q_n 对比 R_n（模型信任对比传感器信任）

6.4.8 LQG（线性二次高斯控制器）

LQR（最优控制器）和卡尔曼滤波器（最优观测器）都有了，把它们拼在一起就是 LQG。**LQG（Linear Quadratic Gaussian，线性二次型高斯控制器）**就是将两者结合的结果：LQR 负责控制，卡尔曼滤波器负责估计，而分离原理（separation principle）保证了这种组合的稳定性。

LQG 架构：

$$\text{卡尔曼滤波器：} \quad \dot{\hat{\mathbf{x}}} = A\hat{\mathbf{x}} + B\mathbf{u} + L(\mathbf{y} - C\hat{\mathbf{x}}) \tag{119}$$

$$\text{LQR 控制器：} \quad \mathbf{u} = -K\hat{\mathbf{x}} \tag{120}$$

其中 K 来自控制 Riccati 方程（最小化状态代价），L 来自估计 Riccati 方程（最小化估计误差）。控制器从不接触真实状态——它作用于卡尔曼估计值。

为什么 LQG 在 MIMO/噪声系统中优于 PID：

1. **最优噪声处理：**PID 通过微分项放大噪声。LQG 的卡尔曼滤波器提供最小方差状态估计——为控制器提供尽可能干净的信号。

2. **天然适合 MIMO：** PID 本质上是 SISO 的。对于 4 状态平衡机器人，PID 需要层叠的多个回路和手工调整的交叉耦合。LQG 用一个 K 矩阵处理所有状态。

3. **状态估计：** PID 无法估计未测量的状态。LQG 的卡尔曼滤波器可从部分测量（例如仅位置传感器）重建完整状态（包括速度）。

4. **系统化整定：** PID 每个回路有 3 个旋钮。LQG 有 Q、R（控制代价）和 Q_n、R_n（噪声模型）——物理含义明确的矩阵。

LQG 的鲁棒性警告： 纯 LQR 具有保证的稳定裕度（增益裕度 ≥ 6 dB，相位裕度 $\geq 60°$）。当你加入卡尔曼滤波器构成 LQG 时，这些保证消失了。LQG 控制器可以有任意小的稳定裕度。这就是著名的 **Doyle 反例**（1978 年）——它推动了 H_∞ 鲁棒控制的发展。

实践含义： 当模型准确且噪声特性明确时，LQG 效果良好。若存在显著的模型不确定性（未知摩擦、变化载荷），可考虑：

- **LQG/LTR（回路传递恢复，Loop Transfer Recovery）：** 修改卡尔曼滤波器整定，在被控对象输入处恢复 LQR 的鲁棒性裕度。

- **H_∞ 控制：** 针对最坏情况的模型不确定性进行设计，而非平均噪声。

- **实用做法：** 在 ±30% 参数变化的仿真中验证 LQG。如果保持稳定，通常足够在比赛中使用。

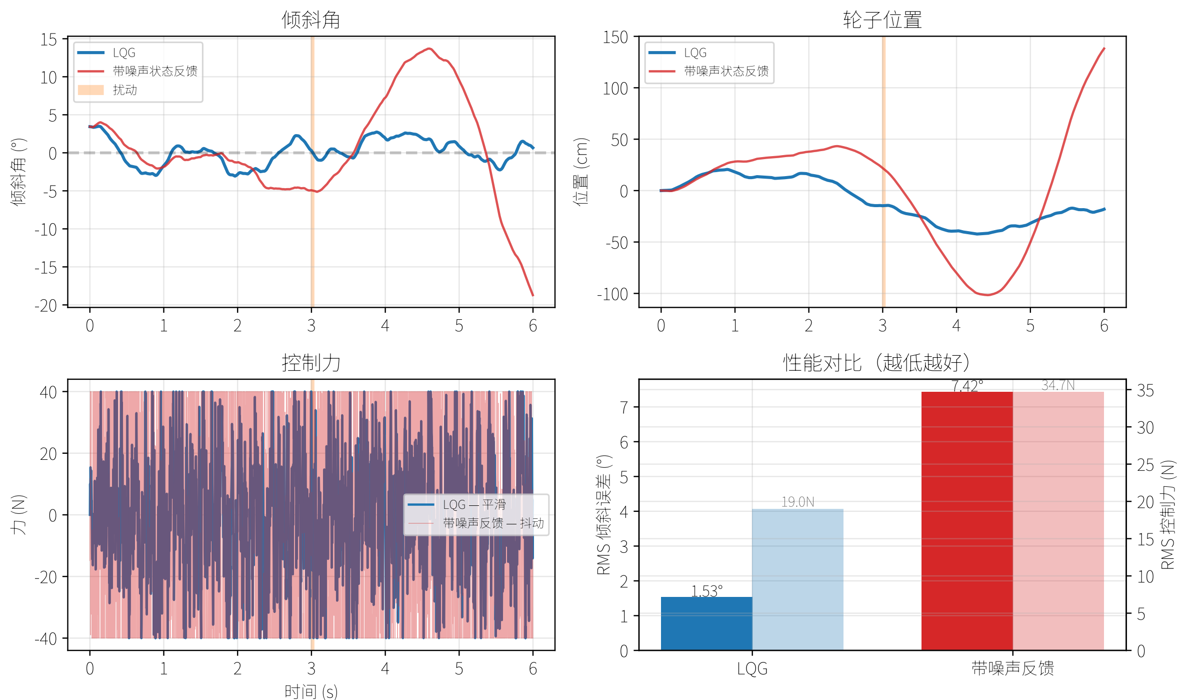

图 20: LQG（蓝色）与直接带噪反馈（红色）在平衡机器人上的对比（$t = 2$s 处有扰动）。两者使用相同的 LQR 增益，但红色使用原始噪声测量加数值微分，蓝色使用卡尔曼滤波器。LQG 恢复更快（左上），位置维持更好（右上），控制量更平滑（左下），RMS 误差更低（右下）。这展示了卡尔曼滤波器的价值：相同增益、相同结构，性能却天壤之别。

实用决策树：

1. **你根本需要状态估计吗？**若你能直接测量所有状态（例如，编码器同时给出位置和速度），则完全跳过观测器。
2. **你的模型准确，且噪声不是主要问题？**使用龙伯格观测器。它更简单，实现和调试更容易。
3. **传感器噪声显著，或需要传感器融合？**使用卡尔曼滤波器。它最优地处理噪声，并自然地融合多个传感器。
4. **你的系统是非线性的？**使用扩展卡尔曼滤波器（EKF）。它在每步线性化，对「轻度」非线性系统（如 IMU 姿态估计）效果良好。
5. **系统高度非线性或 EKF 发散？**使用无迹卡尔曼滤波器（UKF），或考虑粒子滤波器（超出本文范围）。

案例研究：IMU 倾斜角估计——卡尔曼滤波器与互补滤波器

你在做一个自平衡机器人，需要实时、准确地知道倾斜角 θ。手头两个传感器：

- **加速度计：**测量重力方向 → 给出 $\theta_{\text{accel}} = \arctan(a_x/a_z)$。平均而言准确，但有噪声，并且会被任何线性加速度污染（机器人移动时，它会「看到」虚假的倾斜）。
- **陀螺仪：**测量角速度 $\dot{\theta}_{\text{gyro}}$。短期内非常干净，但随时间漂移，因为你在积分：$\theta_{\text{gyro}}(t) = \theta_0 + \int \dot{\theta}_{\text{gyro}}\, dt$。0.1°/s 的微小偏差一分钟后会积累为 6° 的误差。

关键：加速度计低频好（静态倾斜），高频差（振动、加速度干扰）。陀螺仪高频好（快速旋转），低频差（漂移）。天然互补。

方案一：互补滤波器（快速而粗糙的方法）

$$\theta[n] = \alpha \cdot (\theta[n-1] + \dot{\theta}_{\text{gyro}} \cdot T_s) + (1-\alpha) \cdot \theta_{\text{accel}}$$

这字面上就是对陀螺仪施加高通滤波，对加速度计施加低通滤波，取 $\alpha \approx 0.98$。对于一行代码而言，效果出奇地好。

方案二：卡尔曼滤波器（正规方法）

状态：$\mathbf{x} = \begin{bmatrix} \theta \\ \dot{\theta}_{\text{bias}} \end{bmatrix}$（角度和陀螺仪偏差）。

模型：角度由（经偏差修正的）陀螺仪角速度驱动：

$$\begin{bmatrix} \theta \\ \dot{\theta}_{\text{bias}} \end{bmatrix}_{k+1} = \begin{bmatrix} 1 & -T_s \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \theta \\ \dot{\theta}_{\text{bias}} \end{bmatrix}_k + \begin{bmatrix} T_s \\ 0 \end{bmatrix} \dot{\theta}_{\text{gyro},k}$$

测量：加速度计给出 θ 的有噪声观测：

$$y_k = \begin{bmatrix} 1 & 0 \end{bmatrix} \mathbf{x}_k + v_k$$

卡尔曼滤波器能做到而互补滤波器做不到的事：

- 自动**估计并修正陀螺仪偏差**（$\dot{\theta}_{\text{bias}}$）。无需手动标定即可消除漂移。
- 根据噪声统计特性**动态调整信任比例**，而不是使用固定的 α。
- 提供状态的**置信度估计**（P 矩阵），让你知道对输出的信任程度。
- 通过扩展状态向量，可轻松推广到三维姿态（横滚、俯仰、偏航）。

互补滤波器何时「足够好」？对于简单的平衡机器人或机器人不会大幅加速的云台。互补滤波器只有一行代码，不需要矩阵运算。

何时需要卡尔曼滤波器？当机器人有显著的线性加速度（快速行驶的哨兵机器人、无人机）时，当需要陀螺仪偏差估计时，或当融合两个以上传感器（添加磁力计用于偏航、添加车轮编码器用于速度）时。

比赛建议：从互补滤波器开始。如果在激进机动时观察到漂移或性能不佳，升级到卡尔曼滤波器。互补滤波器也是一个很好的「健全性检查」——如果你的卡尔曼滤波器给出了完全不同的答案，说明你的模型有问题。

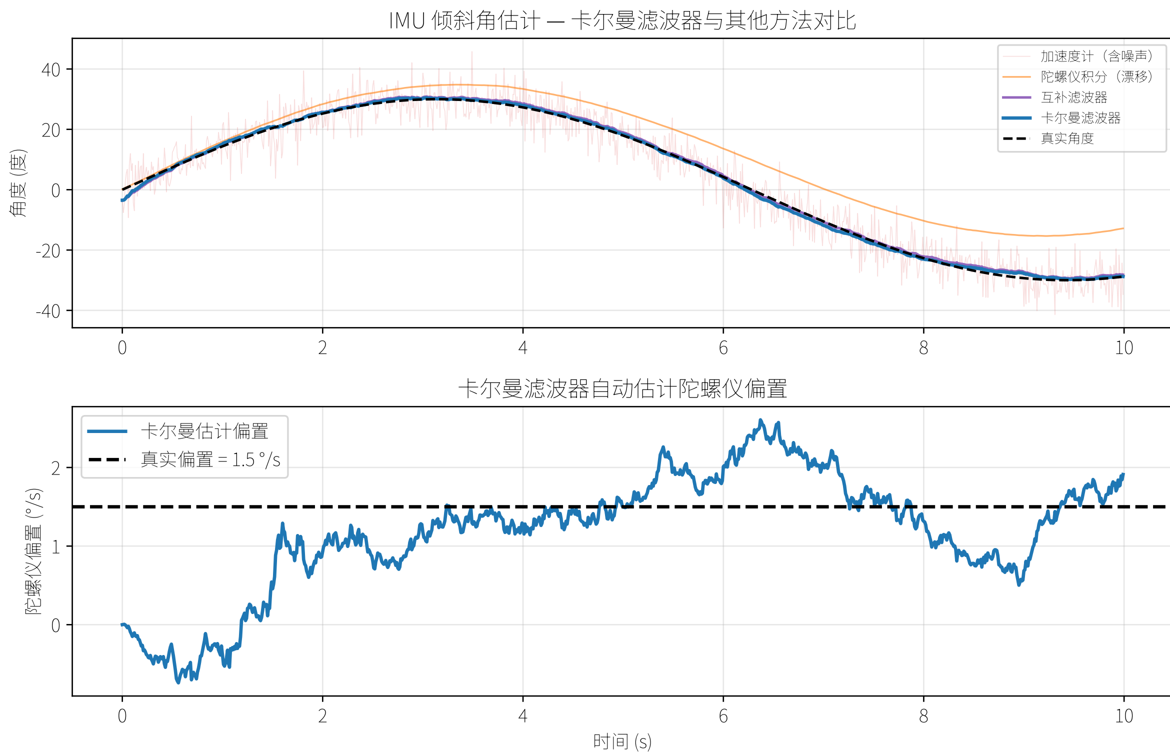

图 21: 顶部：IMU 倾斜角估计对比。纯陀螺仪积分（橙色）严重漂移。加速度计（红色，浅）有噪声。互补滤波器（紫色）和卡尔曼滤波器（蓝色）都能跟踪真实角度（黑色虚线），但卡尔曼更平滑。底部：卡尔曼滤波器自动估计并收敛到 1.5°/s 的真实陀螺仪偏差。

案例研究：LQR 平衡机器人——从方程到实际驾驭

给一个两轮自平衡机器人（类似 Segway）设计控制器。这是现代控制的"Hello World"，能很直观地展示为什么 MIMO 系统用 LQR + 卡尔曼比 PID 更顺手。

系统：一个两轮平台，有一个高耸的机身，必须在保持直立的同时跟踪期望位置。

状态： $\mathbf{x} = \begin{bmatrix} \theta & \dot{\theta} & x & \dot{x} \end{bmatrix}^T$，其中 θ 是倾斜角，x 是车轮位置。

MIMO 的麻烦：要往前走，机器人得先向前倾（就像人要先弯腰才能迈步）。但倾斜恰恰是我们要避免的！位置控制和平衡**耦合且矛盾**。PID 至少要两个嵌套环路，还得小心处理耦合。LQR 一个增益矩阵 $K = [k_1, k_2, k_3, k_4]$ 全部搞定：

$$u = -k_1\theta - k_2\dot{\theta} - k_3(x - x_{\text{ref}}) - k_4\dot{x}$$

通过 Q 和 R 调参：

- 想要激进的平衡？增大 Q_{11}（惩罚倾斜角）。
- 想要精确的位置跟踪？增大 Q_{33}（惩罚位置误差）。
- 电机功率不足？增大 R（惩罚控制代价）。

一个典型的起始点：$Q = \mathrm{diag}(100, 1, 10, 1)$，$R = 1$。意思是：" 倾斜角我比速度在乎 100 倍，位置比速度在乎 10 倍。"MATLAB 里跑一行 `K = lqr(A, B, Q, R)`，四个增益往下灌，机器人就能立起来了。

完整系统：

1. **卡尔曼滤波器**从 IMU + 车轮编码器估计 $[\theta, \dot{\theta}, x, \dot{x}]$。
2. **LQR 控制器**从估计状态计算电机力矩。
3. 两者均以 500–1000 Hz 在 STM32 上运行。

卡尔曼 + LQR，这个组合就是平衡机器人、无人机姿态控制以及一切多状态耦合系统的标配。

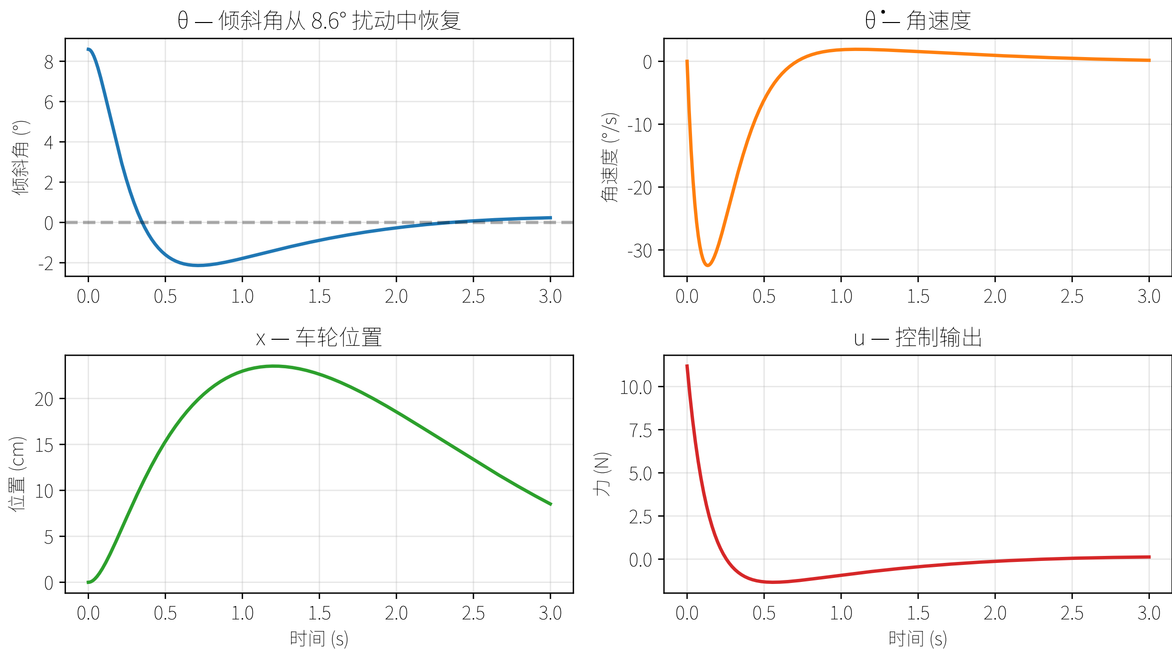

图 22: LQR 平衡机器人仿真。从 8.6° 的初始倾斜开始，控制器将四个状态全部恢复到零。左上：倾斜角在约 1 秒内收敛。右上：角速度。左下：车轮向后移动以承接下落的机身，然后归位。右下：控制力——注意初始的大幅脉冲，随后平稳衰减。

6.5 轨迹生成（Trajectory Generation）：告诉电机去哪里

直接把指令从 0° 跳到 90°，别指望有好结果——电机又不会瞬移。你需要一条平滑的**参考轨迹**，位置、速度、加速度都得规划好，而且要尊重执行器的物理极限。没有轨迹规划，PID 上来就饱和，积分往死里攒，等电机好不容易靠近 90°，控制器已经在跟自己打架了。轨迹生成，就是” 我的炮塔能用” 和” 我的炮塔跟强队一个水平” 之间的分水岭。

6.5.1 梯形速度曲线（Trapezoidal Velocity Profile）

梯形曲线是运动控制的主力。给定移动距离 d、最大速度 $v_{\max}$ 和最大加速度 $a_{\max}$，产生三个阶段：

1. 从静止以 $a_{\max}$ **加速**至 $v_{\max}$
2. 以 $v_{\max}$ **匀速**行驶
3. 以 $-a_{\max}$ **减速**回到静止

时间与距离：

$$t_{\text{加速}} = \frac{v_{\max}}{a_{\max}}, \qquad d_{\text{加速}} = \frac{1}{2} a_{\max} t_{\text{加速}}^2, \qquad d_{\text{匀速}} = d - 2\, d_{\text{加速}} \tag{121}$$

如果 $d_{\text{匀速}} < 0$，距离不足以达到 $v_{\max}$。曲线变为**三角形**：电机加速至峰值速度 $v_{\text{peak}} = \sqrt{a_{\max} d}$ 后立即减速，没有匀速阶段。

时刻 t 的参考信号：

令 $t_1 = t_{\text{加速}}$，$t_2 = t_1 + d_{\text{匀速}}/v_{\max}$，$t_3 = t_2 + t_{\text{加速}}$。

$$\theta_{\text{ref}}(t) = \begin{cases} \frac{1}{2} a_{\max} t^2 & 0 \le t < t_1 \quad \text{（加速段）} \\ d_{\text{加速}} + v_{\max}(t - t_1) & t_1 \le t < t_2 \quad \text{（匀速段）} \\ d - \frac{1}{2} a_{\max}(t_3 - t)^2 & t_2 \le t \le t_3 \quad \text{（减速段）} \end{cases} \tag{122}$$

$$\dot{\theta}_{\text{ref}}(t) = \begin{cases} a_{\max}\, t & 0 \le t < t_1 \\ v_{\max} & t_1 \le t < t_2 \\ a_{\max}(t_3 - t) & t_2 \le t \le t_3 \end{cases} \qquad \ddot{\theta}_{\text{ref}}(t) = \begin{cases} +a_{\max} & 0 \le t < t_1 \\ 0 & t_1 \le t < t_2 \\ -a_{\max} & t_2 \le t \le t_3 \end{cases} \tag{123}$$

这三个信号是实现完整前馈增强控制器所需的全部内容（更多内容见下面的实践框）。

6.5.2 S 形速度曲线（S-Curve Profile）

梯形曲线的加速度是不连续的——瞬间从 0 跳变到 $a_{\max}$。这种突然的冲击（jerk）会激励机械共振，磨损齿轮，引起振动。**S 形曲线**（或限制冲击的曲线）增加了第七个约束：加速度的变化率（冲击，$j_{\max}$）受到限制。

这将三段式梯形曲线变为七段式曲线：

1. 冲击增加（加速度从 0 增加到 $a_{\max}$）
2. 恒定加速度 $a_{\max}$
3. 冲击减小（加速度从 $a_{\max}$ 减小到 0）
4. 以 $v_{\max}$ 匀速行驶
5. 冲击减小（加速度变为 $-a_{\max}$）
6. 恒定减速度 $-a_{\max}$

7. 冲击增加（回到零加速度）

结果是一个处处二阶连续可微的位置曲线——平滑到云台和精密机构几乎感觉不到运动开始。数学推导较为复杂，但实践中你很少需要从头实现：运动控制库（WPILib 的 `TrapezoidProfile`、REV 的 Motion Magic 等）已经帮你处理好了。

实践：如何选择曲线类型

梯形曲线适合 90% 的竞赛应用：

- 驱动轮、飞轮、炮塔旋转
- 实现和调试简单——三段、分段公式
- 足够好用，除非你听到齿轮震颤或看到快速运动时的振荡

S 形曲线适用于对平滑性要求严苛的场合：

- 云台和双轴稳定平台
- 精密定位机构（相机滑轨、基于编码器的机械臂）
- 任何担心齿轮磨损或振动的场合

跟踪轨迹炮塔的完整前馈链：

在每个控制时间步，轨迹规划器输出三个数：θ_{ref}、$\dot{\theta}_{\text{ref}}$、$\ddot{\theta}_{\text{ref}}$。全部利用起来：

$$u = \underbrace{K_p\big(\theta_{\text{ref}} - \theta\big) + K_i \int (\cdot)\, dt + K_d \dot{e}}_{\text{外位置 PID}} + \underbrace{k_v\, \dot{\theta}_{\text{ref}}}_{\text{速度前馈}} + \underbrace{k_a\, \ddot{\theta}_{\text{ref}}}_{\text{加速度前馈}}$$

加速度前馈来自旋转的牛顿第二定律：$\tau = J\alpha$，因此 $k_a \approx J/k_\tau$，其中 k_τ 是电机的力矩常数。位置 PID 只需消除小的跟踪误差；前馈承担主要工作。

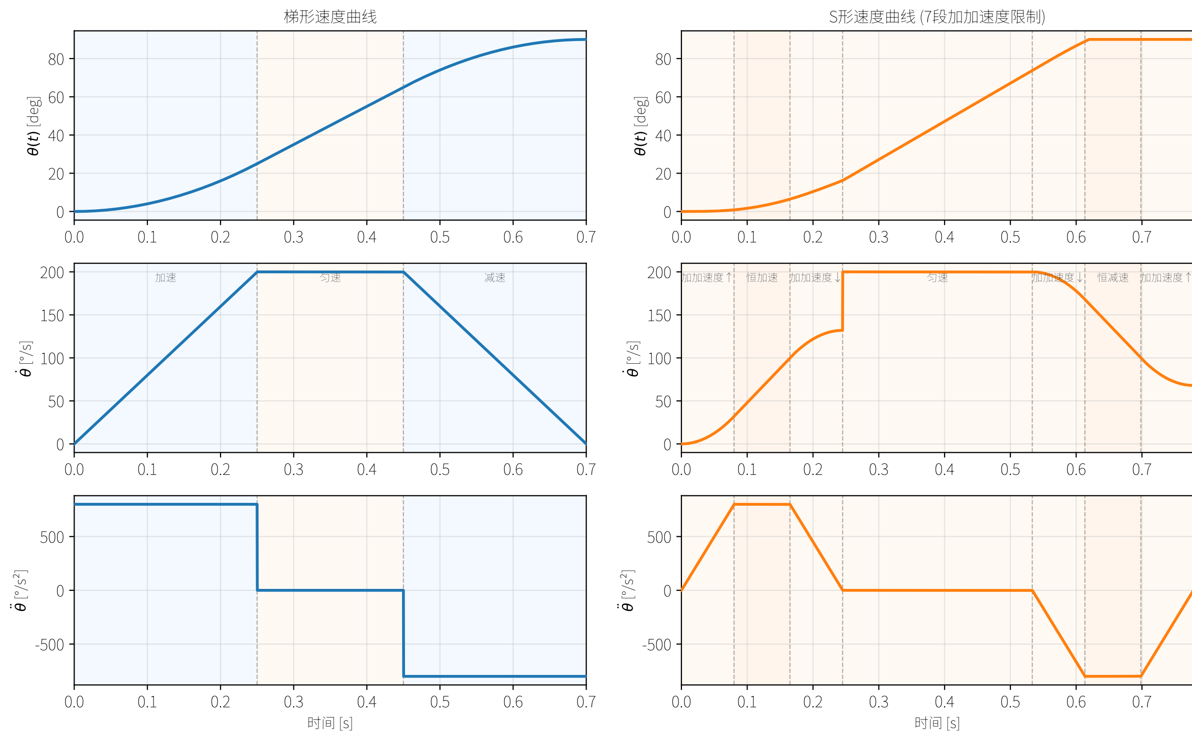

图 23: 90° 运动的梯形曲线（左）与 S 形曲线（右）对比。梯形曲线的加速度不连续（突然跳变），而 S 形曲线限制冲击以实现更平滑的运动。两者都生成用于前馈控制的位置、速度和加速度参考信号。

6.5.3 Bézier 曲线与样条插值

梯形曲线和 S 形曲线适用于单段点到点运动，但许多应用场景需要在二维或三维空间中沿多个路径点生成平滑轨迹。**Bézier 曲线**和**三次样条**是实现该目标的两种常用工具。

Bézier 曲线。 n 次 Bézier 曲线由 $n+1$ 个**控制点** $\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_n$ 定义，其参数化形式为 ($t \in [0,1]$)：

$$\mathbf{B}(t) = \sum_{i=0}^{n} \binom{n}{i} (1-t)^{n-i}\, t^i\, \mathbf{P}_i$$

最常用的是**三次 Bézier 曲线**（$n = 3$），使用四个控制点。曲线经过 $\mathbf{P}_0$（$t = 0$）和 $\mathbf{P}_3$（$t = 1$），而 $\mathbf{P}_1$ 和 $\mathbf{P}_2$ 控制曲线形状但不在曲线上。其关键性质在于：曲线在 $\mathbf{P}_0$ 处的切线指向 $\mathbf{P}_1$，在 $\mathbf{P}_3$ 处的切线从 $\mathbf{P}_2$ 方向出发——这为进出方向提供了直观的控制手段。Bézier 曲线适用于需要平滑曲线且可控进出方向的**路径点规划**场景（如全向驱动路径、机械臂末端执行器轨迹等）。

三次样条插值。 若需经过一组路径点 $\{q_0, q_1, \ldots, q_N\}$，可采用在路径点处拼接的**分段三次多项式**。每段的形式为：

$$q_i(t) = a_i + b_i\, t + c_i\, t^2 + d_i\, t^3$$

在各拼接点处，**连续性条件**要求相邻段的位置、速度和加速度相匹配，从而获得 C^2 连续性（直到二阶导数连续）。常用的边界条件有两种：

- **自然样条**：两端点处的二阶导数为零。

- **固定样条**：在两端点处指定速度（一阶导数）。

与 Bézier 曲线相比，三次样条**保证经过所有路径点**——Bézier 曲线仅经过首尾两个控制点。与梯形曲线相比，三次样条在全程提供**光滑的速度和加速度**，不存在不连续点。

适用场景对比。

- **梯形/S 形曲线**：具有明确速度和加速度限制的单段点到点运动。
- **Bézier 曲线**：需要形状控制的平滑二维/三维路径（如全向驱动路径、末端执行器轨迹）。
- **三次样条**：需要经过每个路径点的多点轨迹（如自主导航序列）。

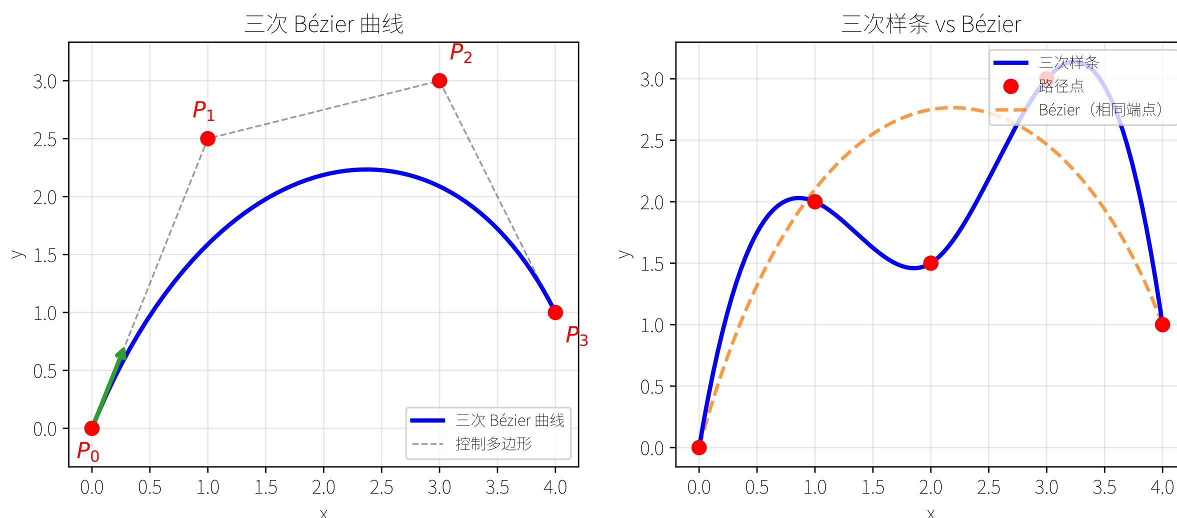

图 24: 左图：三次 Bézier 曲线由四个控制点 P_0–P_3 定义，曲线经过 P_0 和 P_3，P_1 和 P_2 控制曲线形状。绿色和红色箭头表示端点处的切线方向。右图：三次样条插值保证经过所有路径点（红点），而 Bézier 曲线仅经过首尾两个控制点。

7 展望：数据驱动与基于学习的控制

前述所有方法——PID、状态空间、LQR、卡尔曼滤波、MPC——均假设已知被控对象的模型。然而在实际工程中，获取精确模型往往是最困难的环节。新一代技术试图绕过或增强建模步骤，直接从数据中学习控制策略。本章概览这些前沿方向，并讨论它们与经典控制理论的关系。

7.1 数据驱动控制

7.1.1 动机：从“先建模再控制”到“直接从数据控制”

传统流程是：采集数据 $\rightarrow$ 系统辨识 $\rightarrow$ 建立模型 $\rightarrow$ 设计控制器。每一步都会引入误差，误差会逐级放大。一个自然的问题是：*能否跳过建模，直接从数据设计控制器？*

7.1.2 Willems 基本引理

这一切的理论基础是 **Willems 基本引理（Fundamental Lemma, 2005）**：对于线性时不变系统，若一条长度为 T 的输入–输出轨迹具有足够阶数的*持续激励*（*persistent excitation*）特性，则系统的所有可能轨迹均可表示为该数据 Hankel 矩阵各列的线性组合：

$$\begin{bmatrix} U_- \\ X_- \\ U_+ \\ X_+ \end{bmatrix} g = \begin{bmatrix} u_{\text{ini}} \\ x_{\text{ini}} \\ u \\ x \end{bmatrix},$$

其中 U_-, X_- 为历史数据构成的 Hankel 矩阵，g 为待求的系数向量。

直觉解读： 一条足够丰富的轨迹“编码”了系统的全部动力学信息，与知道 (A, B) 矩阵等价。

7.1.3 数据驱动 LQR

利用 Willems 引理，LQR 问题

$$\min_K \sum_{k=0}^{\infty} \bigl(x_k^\top Q\, x_k + u_k^\top R\, u_k\bigr)$$

可以直接基于 Hankel 矩阵构建的数据一致性约束来求解，*无需辨识* A 或 B。

局限性：

- 数据必须满足持续激励条件——随机噪声不够，输入信号需足够丰富。
- 对测量噪声较为敏感。
- 目前仅适用于线性系统（非线性扩展是活跃的研究方向）。

7.1.4 数据驱动 MPC（DeePC）

DeePC（Data-Enabled Predictive Control）将 Willems 引理与 MPC 结合：用数据 Hankel 矩阵替代预测模型，直接在数据空间中求解预测控制问题。这使得 MPC 的约束处理能力得以保留，同时无需显式系统辨识。

7.2 强化学习与控制

强化学习（RL）将最优控制推广到具有任意代价函数和约束的非线性系统。

7.2.1 核心概念

- **策略** $\pi(s) \to a$：从状态到动作的映射——即控制器。在线性二次情形下，$\pi(x) = -Kx$ 退化为 LQR。
- **值函数** $V^\pi(s) = \mathbb{E}\big[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s\big]$：从某状态出发的期望累积奖励，类比动态规划中的最优代价函数（cost-to-go）。
- **策略梯度**：沿 $\nabla_\theta J(\theta)$ 方向更新策略参数 θ，使期望奖励最大化。

7.2.2 两种范式

1. **无模型 RL**（PPO、SAC 等）：通过试错直接学习策略，不需要动力学模型。通用性强，但样本效率极低——通常需要数百万次交互，仅在仿真中可行，部署时需进行 sim-to-real 迁移。
2. **基于模型的 RL**（MBPO、Dreamer 等）：先从数据中学习动力学模型，再利用该模型进行规划或策略优化。样本效率更高，更接近传统控制的思路。

7.2.3 Sim-to-Real 迁移

RL 策略通常在仿真中训练，但仿真与真实系统之间存在差距（sim-to-real gap）。常用的弥合策略：

- **域随机化（Domain Randomization）**：在训练时随机化物理参数（质量、摩擦系数、延迟等），迫使策略学习对参数变化具有鲁棒性的行为。
- **系统辨识与高保真仿真**：精确标定仿真参数，缩小仿真与真实之间的差距。
- **微调（Fine-tuning）**：先在仿真中预训练，再在真实系统上用少量数据进行微调。

7.2.4 成功案例与现状

RL 在科研领域取得了显著成果：

- **足式运动**：MIT Mini Cheetah、ANYmal 等四足机器人通过 RL 学会了在崎岖地形上行走和奔跑。
- **灵巧操控**：OpenAI 的魔方求解、谷歌的机器人抓取。
- **无人机竞速**：苏黎世联邦理工 Swift 系统在无人机竞速中超越人类冠军。

但在工程竞赛和工业场景中 RL 仍然罕见，原因包括：

- 训练需要高保真仿真环境，开发成本高。
- 学习到的策略是"黑箱"，调试极其困难。
- 竞赛周期通常不足以完成 RL 的开发迭代。
- 安全性和可解释性保证不足。

7.3 轨迹优化与最优控制

最优控制的目标是寻找 $u(t)$ 使代价泛函最小化：

$$J = \int_0^T L\big(x(t), u(t)\big)\,\mathrm{d}t + \Phi\big(x(T)\big),$$

约束为动力学方程 $\dot{x} = f(x, u)$。

7.3.1 两类求解策略

- **间接法**（庞特里亚金极大值原理）：推导伴随方程得到必要条件，求解两点边值问题。数学上优雅，但对复杂约束系统的实现难度大。
- **直接法**（配点法、多重打靶法）：将轨迹离散化为决策变量，转化为大规模非线性规划（NLP）。更具实用性——现代轨迹优化工具普遍采用此策略。

7.3.2 iLQR 与 DDP

迭代 LQR（iLQR）与**微分动态规划（DDP）**是现代机器人轨迹优化的主力算法：在标称轨迹处线性化非线性动力学，求解对应的 LQR 子问题获得修正量，更新轨迹后重复迭代。

与 MPC 的联系：模型预测控制本质上是在每个时间步以滚动时域方式重复求解轨迹优化。当动力学为非线性时，MPC 内部常使用 iLQR 或 NLP 求解器。

常用工具：CasADi、Drake、ALTRO、Crocoddyl、MATLAB `fmincon`。

7.4 神经网络与控制的融合

7.4.1 神经网络作为控制器组件

神经网络不一定要替代整个控制器，更实用的方式是将其嵌入经典控制框架的特定模块中：

- **学习动力学模型**：用神经网络拟合 $\dot{x} = f_\theta(x,u)$，然后在 MPC 或 iLQR 中使用该模型。
- **学习残差动力学**：$\dot{x} = f_{\text{phys}}(x,u) + f_\theta(x,u)$，让神经网络补偿物理模型未能捕捉的部分。
- **学习代价函数**：从人类演示中学习任务目标（逆强化学习）。

7.4.2 安全约束与可验证性

将学习方法与安全保证结合是当前的热点研究方向：

- **控制屏障函数（CBF）**：为学习到的策略添加安全约束层，确保系统状态不离开安全集合。
- **Lyapunov 神经网络**：训练神经网络同时学习控制器和 Lyapunov 函数，从而提供稳定性证明。
- **可达性分析**：计算系统在学习策略下的可达集合，验证其是否满足安全要求。

7.5 方法全景对比

方法	需要模型？	处理非线性？	实时？	成熟度
PID	否（经验调参）	局部	是	工业标准
LQR	是（线性）	否	是	成熟
MPC（线性）	是（线性）	否	是	成熟
NMPC	是（非线性）	是	视情况	初步工业应用
数据驱动 LQR	否（仅数据）	否	是	学术前沿
DeePC	否（仅数据）	有限	视情况	学术前沿
基于模型的 RL	学习所得	是	离线训练	科研验证
无模型 RL	否	是	离线训练	科研验证
iLQR / DDP	是（非线性）	是	准实时	科研/初步工业
神经网络 + MPC	混合	是	视情况	学术前沿

7.6 给读者的建议

全局视角。控制理论与机器学习之间的边界正在消融。数据驱动方法带来了自适应能力，基于学习的方法带来了处理复杂性的能力。但本文所涵盖的基础内容——状态空间建模、稳定性分析、观测器设计、最优控制——仍然是这一切的根基。

实用路线图：

1. **先掌握经典方法**。PID、LQR、MPC、卡尔曼滤波——这些工具在绝大多数机器人应用中足够使用，且可靠、可调试、有理论保证。
2. **当经典方法遇到瓶颈时，引入数据驱动技术**。系统难以建模？尝试数据驱动 LQR/MPC。环境变化大？考虑自适应控制。
3. **RL 用于“经典方法做不到的事”**。极端敏捷运动、高维灵巧操控、复杂接触交互——这些场景下 RL 的优势才真正显现。
4. **永远保留安全层**。无论控制器多么“智能”，添加控制屏障函数或硬件安全限位，确保系统不会做出危险动作。

深入理解经典基础，将使你成为更优秀的高级方法使用者，而非被取代者。

8 附录：即插即用 C++ 模块

本附录提供了笔记中所有主要算法的生产级、仅头文件（header-only）C++ 实现。每个模块均自成一体——只需将头文件放入项目中即可使用。所有模块均针对嵌入式系统设计（无动态内存分配、无异常、无 STL 容器）。

设计理念（Design Philosophy）：

- **仅头文件（Header-only）**：无需 `.cpp` 文件。直接包含即用。
- **无堆分配（No heap allocation）**：所有数据均存于栈上，或通过模板进行静态尺寸确定。
- **极少依赖（Minimal dependencies）**：仅依赖 `<cmath>` 与 `<cstring>`，无 STL。
- **中断安全（ISR-safe）**：所有 `update()` 方法均可在定时器中断中安全调用。
- **基于模板（Template-based）**：矩阵维度为编译期常量，实现零开销抽象。

8.1 低通滤波器（Low-Pass Filter）

一阶 IIR 低通滤波器。一行构造，一行使用。

```
// lpf.hpp
#pragma once
#include <cmath>

class LowPassFilter {
public:
    LowPassFilter() : alpha_(1.0f), y_prev_(0.0f), initialized_(false) {}

    /// 用截止频率 (Hz) 和采样周期 (s) 进行配置
    void configure(float cutoff_hz, float dt) {
        float rc = 1.0f / (2.0f * M_PI * cutoff_hz);
        alpha_ = dt / (rc + dt);
    }

    /// 直接设置平滑因子 (0 < alpha <= 1)
    void set_alpha(float alpha) { alpha_ = alpha; }

    /// 处理一个采样点，以固定采样率调用
    float update(float x) {
        if (!initialized_) {
            y_prev_ = x;
            initialized_ = true;
            return x;
        }
        y_prev_ = alpha_ * x + (1.0f - alpha_) * y_prev_;
        return y_prev_;
```

```
    }

    /// 重置滤波器状态
    void reset() { initialized_ = false; y_prev_ = 0.0f; }

    /// 获取当前滤波值，不输入新采样点
    float value() const { return y_prev_; }

private:
    float alpha_;
    float y_prev_;
    bool  initialized_;
};
```

用法（Usage）：

```
LowPassFilter gyro_filter;
gyro_filter.configure(30.0f, 0.001f);  // 截止频率 30 Hz，采样频率 1 kHz

// 在控制循环 ISR 中：
float gyro_clean = gyro_filter.update(gyro_raw);
```

8.2 PID 控制器（PID Controller）

功能完整的 PID，包含反计算抗积分饱和（back-calculation anti-windup）、测量值微分（derivative-on-measurement）、微分滤波和输出限幅。

```
// pid.hpp
#pragma once
#include <cmath>

struct PIDConfig {
    float kp = 0.0f;
    float ki = 0.0f;
    float kd = 0.0f;
    float dt = 0.001f;         // 采样周期 (s)
    float out_min = -1e6f;     // 输出下限
    float out_max =  1e6f;     // 输出上限
    float integral_max = 1e6f; // 积分项限幅
    float d_filter_N = 10.0f;  // 微分滤波系数（越大滤波越弱）
};

class PID {
public:
    PID() = default;
```

```
explicit PID(const PIDConfig& cfg) : cfg_(cfg) {}

void configure(const PIDConfig& cfg) { cfg_ = cfg; }

/// 计算 PID 输出，每个控制周期调用一次
/// setpoint: 期望值，measurement: 实测值
float update(float setpoint, float measurement) {
    float error = setpoint - measurement;

    // --- 比例项 ---
    float p_term = cfg_.kp * error;

    // --- 带反计算抗积分饱和的积分项 ---
    integral_ += cfg_.ki * error * cfg_.dt + anti_windup_;
    integral_ = clamp(integral_, -cfg_.integral_max, cfg_.integral_max);

    // --- 测量值微分（避免微分冲击）---
    //     带一阶低通滤波器
    float d_raw = -(measurement - meas_prev_) / cfg_.dt;
    float alpha_d = cfg_.dt * cfg_.d_filter_N
                  / (1.0f + cfg_.dt * cfg_.d_filter_N);
    d_filtered_ = alpha_d * d_raw + (1.0f - alpha_d) * d_filtered_;
    float d_term = cfg_.kd * d_filtered_;

    meas_prev_ = measurement;

    // --- 总输出 ---
    float output_raw = p_term + integral_ + d_term;
    float output = clamp(output_raw, cfg_.out_min, cfg_.out_max);

    // --- 反计算抗积分饱和 ---
    if (cfg_.ki != 0.0f) {
        anti_windup_ = (output - output_raw) * (1.0f / cfg_.ki)
                     * cfg_.dt * 0.5f;
    } else {
        anti_windup_ = 0.0f;
    }

    return output;
}

void reset() {
```

```
        integral_ = 0.0f;
        d_filtered_ = 0.0f;
        meas_prev_ = 0.0f;
        anti_windup_ = 0.0f;
    }

    // 访问内部状态用于调试
    float get_integral() const { return integral_; }

private:
    PIDConfig cfg_;
    float integral_    = 0.0f;
    float d_filtered_  = 0.0f;
    float meas_prev_   = 0.0f;
    float anti_windup_ = 0.0f;

    static float clamp(float v, float lo, float hi) {
        return v < lo ? lo : (v > hi ? hi : v);
    }
};
```

用法（Usage）：

```
PIDConfig cfg;
cfg.kp = 6.0f;  cfg.ki = 0.3f;  cfg.kd = 3.0f;
cfg.dt = 0.001f;
cfg.out_min = -30000.0f;  cfg.out_max = 30000.0f;  // GM6020 输出范围
cfg.integral_max = 10000.0f;

PID yaw_pid(cfg);

// 在 ISR 中：
float torque_cmd = yaw_pid.update(target_angle, current_angle);
```

8.3 给定值斜坡（Setpoint Ramp，LPF 软启动）

专用给定值平滑器——封装低通滤波器，实现限速的给定值过渡。

```
// ramp.hpp
#pragma once
#include "lpf.hpp"

class SetpointRamp {
public:
    /// ramp_time: 阶跃变化到达 95% 所需的近似时间（秒）
```

```
    void configure(float ramp_time, float dt) {
        float cutoff = 3.0f / (2.0f * M_PI * ramp_time);  // 3*tau ~ 95%
        filter_.configure(cutoff, dt);
    }

    float update(float raw_setpoint) {
        return filter_.update(raw_setpoint);
    }

    void reset() { filter_.reset(); }

private:
    LowPassFilter filter_;
};
```

8.4 互补滤波器（Complementary Filter，IMU 倾斜估计）

融合加速度计（重力倾斜）与陀螺仪（角速率）。

```
// complementary_filter.hpp
#pragma once
#include <cmath>

class ComplementaryFilter {
public:
    /// alpha: 陀螺仪信任系数（典型值 0.95-0.99）
    void configure(float alpha, float dt) {
        alpha_ = alpha;
        dt_ = dt;
    }

    /// accel_angle: 由加速度计计算的倾斜角（rad）
    /// gyro_rate:   陀螺仪角速率（rad/s）
    /// 返回值:      融合后的角度估计（rad）
    float update(float accel_angle, float gyro_rate) {
        angle_ = alpha_ * (angle_ + gyro_rate * dt_)
               + (1.0f - alpha_) * accel_angle;
        return angle_;
    }

    void reset(float initial_angle = 0.0f) { angle_ = initial_angle; }
    float angle() const { return angle_; }

private:
```

```
    float alpha_ = 0.98f;
    float dt_    = 0.001f;
    float angle_ = 0.0f;
};
```

8.5 轻量矩阵库（Lightweight Matrix Library）

编译期固定尺寸矩阵，用于嵌入式卡尔曼滤波器/LQR。无堆分配，无 STL。

```
// mat.hpp
#pragma once
#include <cstring>
#include <cmath>

template<int ROWS, int COLS>
struct Mat {
    float data[ROWS][COLS] = {};

    float& operator()(int r, int c)       { return data[r][c]; }
    float  operator()(int r, int c) const { return data[r][c]; }

    static Mat zeros() { Mat m; return m; }

    static Mat identity() {
        static_assert(ROWS == COLS, "Identity requires square matrix");
        Mat m;
        for (int i = 0; i < ROWS; ++i) m.data[i][i] = 1.0f;
        return m;
    }

    // 矩阵加法
    Mat operator+(const Mat& rhs) const {
        Mat r;
        for (int i = 0; i < ROWS; ++i)
            for (int j = 0; j < COLS; ++j)
                r.data[i][j] = data[i][j] + rhs.data[i][j];
        return r;
    }

    // 矩阵减法
    Mat operator-(const Mat& rhs) const {
        Mat r;
        for (int i = 0; i < ROWS; ++i)
            for (int j = 0; j < COLS; ++j)
```

```
            r.data[i][j] = data[i][j] - rhs.data[i][j];
        return r;
    }

    // 矩阵乘法
    template<int COLS2>
    Mat<ROWS, COLS2> operator*(const Mat<COLS, COLS2>& rhs) const {
        Mat<ROWS, COLS2> r;
        for (int i = 0; i < ROWS; ++i)
            for (int j = 0; j < COLS2; ++j)
                for (int k = 0; k < COLS; ++k)
                    r.data[i][j] += data[i][k] * rhs.data[k][j];
        return r;
    }

    // 标量乘法
    Mat operator*(float s) const {
        Mat r;
        for (int i = 0; i < ROWS; ++i)
            for (int j = 0; j < COLS; ++j)
                r.data[i][j] = data[i][j] * s;
        return r;
    }

    // 转置
    Mat<COLS, ROWS> T() const {
        Mat<COLS, ROWS> r;
        for (int i = 0; i < ROWS; ++i)
            for (int j = 0; j < COLS; ++j)
                r.data[j][i] = data[i][j];
        return r;
    }

    // 2x2 或 3x3 逆矩阵（仅限小矩阵，供卡尔曼滤波器使用）
    Mat inverse() const;
};

// 1x1 逆矩阵
template<> inline Mat<1,1> Mat<1,1>::inverse() const {
    Mat<1,1> r;
    r.data[0][0] = 1.0f / data[0][0];
    return r;
```

```
}

// 2x2 逆矩阵
template<> inline Mat<2,2> Mat<2,2>::inverse() const {
    Mat<2,2> r;
    float det = data[0][0]*data[1][1] - data[0][1]*data[1][0];
    float inv_det = 1.0f / det;
    r.data[0][0] =  data[1][1] * inv_det;
    r.data[0][1] = -data[0][1] * inv_det;
    r.data[1][0] = -data[1][0] * inv_det;
    r.data[1][1] =  data[0][0] * inv_det;
    return r;
}

// 3x3 逆矩阵
template<> inline Mat<3,3> Mat<3,3>::inverse() const {
    Mat<3,3> r;
    float det = data[0][0]*(data[1][1]*data[2][2]-data[1][2]*data[2][1])
              - data[0][1]*(data[1][0]*data[2][2]-data[1][2]*data[2][0])
              + data[0][2]*(data[1][0]*data[2][1]-data[1][1]*data[2][0]);
    float inv = 1.0f / det;
    r(0,0)=(data[1][1]*data[2][2]-data[1][2]*data[2][1])*inv;
    r(0,1)=(data[0][2]*data[2][1]-data[0][1]*data[2][2])*inv;
    r(0,2)=(data[0][1]*data[1][2]-data[0][2]*data[1][1])*inv;
    r(1,0)=(data[1][2]*data[2][0]-data[1][0]*data[2][2])*inv;
    r(1,1)=(data[0][0]*data[2][2]-data[0][2]*data[2][0])*inv;
    r(1,2)=(data[0][2]*data[1][0]-data[0][0]*data[1][2])*inv;
    r(2,0)=(data[1][0]*data[2][1]-data[1][1]*data[2][0])*inv;
    r(2,1)=(data[0][1]*data[2][0]-data[0][0]*data[2][1])*inv;
    r(2,2)=(data[0][0]*data[1][1]-data[0][1]*data[1][0])*inv;
    return r;
}
```

8.6 卡尔曼滤波器（Kalman Filter）

通用离散时间卡尔曼滤波器。模板参数在编译期确定状态维度与测量维度。

```
// kalman.hpp
#pragma once
#include "mat.hpp"

template<int N_STATE, int N_MEAS, int N_INPUT = 1>
class KalmanFilter {
public:
```

```
using StateVec  = Mat<N_STATE, 1>;
using MeasVec   = Mat<N_MEAS, 1>;
using InputVec  = Mat<N_INPUT, 1>;
using StateMat  = Mat<N_STATE, N_STATE>;
using InputMat  = Mat<N_STATE, N_INPUT>;
using MeasMat   = Mat<N_MEAS, N_STATE>;
using GainMat   = Mat<N_STATE, N_MEAS>;
using MeasCov   = Mat<N_MEAS, N_MEAS>;

StateMat A;   // 状态转移矩阵
InputMat B;   // 输入矩阵
MeasMat  C;   // 测量矩阵（部分文献记为 H)
StateMat Q;   // 过程噪声协方差
MeasCov  R;   // 测量噪声协方差

StateVec x;   // 状态估计
StateMat P;   // 估计协方差

/// 预测步骤：向前传播状态与协方差
void predict(const InputVec& u) {
    x = A * x + B * u;
    P = A * P * A.T() + Q;
}

/// 预测步骤（无输入)
void predict() {
    x = A * x;
    P = A * P * A.T() + Q;
}

/// 更新步骤：融合测量值
void update(const MeasVec& z) {
    // 新息
    MeasVec y = z - C * x;
    // 新息协方差
    MeasCov S = C * P * C.T() + R;
    // 卡尔曼增益
    GainMat K = P * C.T() * S.inverse();
    // 状态更新
    x = x + K * y;
    // 协方差更新（Joseph 形式，提高数值稳定性)
    StateMat I_KC = StateMat::identity() - K * C;
```

```
        P = I_KC * P * I_KC.T() + K * R * K.T();
    }

    /// 预测 + 更新合并
    void step(const InputVec& u, const MeasVec& z) {
        predict(u);
        update(z);
    }
};
```

用法——IMU 倾斜估计（案例研究中的 2 状态卡尔曼滤波器）：

```
KalmanFilter<2, 1, 1> kf;
float dt = 0.001f;

// 状态：[角度，陀螺仪偏置]
kf.A(0,0) = 1.0f;  kf.A(0,1) = -dt;
kf.A(1,0) = 0.0f;  kf.A(1,1) = 1.0f;

kf.B(0,0) = dt;
kf.B(1,0) = 0.0f;

kf.C(0,0) = 1.0f;  kf.C(0,1) = 0.0f;

// 调整这些参数
kf.Q(0,0) = 0.001f;  kf.Q(1,1) = 0.003f;
kf.R(0,0) = 0.03f;

kf.P = Mat<2,2>::identity() * 1.0f;

// 在 ISR 中：
Mat<1,1> u;  u(0,0) = gyro_rate;
Mat<1,1> z;  z(0,0) = accel_angle;
kf.step(u, z);
float estimated_angle = kf.x(0, 0);
float estimated_bias  = kf.x(1, 0);
```

8.7 LQR 增益（离线计算辅助工具）

LQR 增益计算需要求解 Riccati 方程，通常在 MATLAB/Python 中离线完成。本模块在运行时应用预计算得到的增益矩阵 K。

```
// lqr.hpp
#pragma once
#include "mat.hpp"
```

```
template<int N_STATE, int N_INPUT>
class LQRController {
public:
    using StateVec = Mat<N_STATE, 1>;
    using InputVec = Mat<N_INPUT, 1>;
    using GainMat  = Mat<N_INPUT, N_STATE>;

    GainMat K;  // 由 MATLAB 计算后填入：K = lqr(A, B, Q, R)

    /// 计算最优控制输入：u = -K * (x - x_ref)
    InputVec compute(const StateVec& x, const StateVec& x_ref) const {
        StateVec error = x - x_ref;
        return (K * error) * (-1.0f);
    }

    /// 计算调节控制输入 (x_ref = 0)
    InputVec compute(const StateVec& x) const {
        return (K * x) * (-1.0f);
    }
};
```

用法——平衡小车：

```
// 增益离线计算：K = lqr(A, B, diag([100,1,10,1]), 1)
LQRController<4, 1> lqr;
lqr.K(0,0) = -31.62f;  // 角度增益
lqr.K(0,1) = -5.27f;   // 角速度增益
lqr.K(0,2) = -3.16f;   // 位置增益
lqr.K(0,3) = -4.12f;   // 速度增益

// 在 ISR 中 (状态来自卡尔曼滤波器) :
Mat<4,1> state;
state(0,0) = kf.x(0,0);  // 角度
state(1,0) = kf.x(1,0);  // 角速度
state(2,0) = encoder_pos;
state(3,0) = encoder_vel;

auto u = lqr.compute(state);
set_motor_torque(u(0,0));
```

8.8 串级 PID（Cascaded PID）

双环串级控制器（外环位置 + 内环速度），与云台案例研究中描述的结构一致。

```cpp
// cascaded_pid.hpp
#pragma once
#include "pid.hpp"

class CascadedPID {
public:
    PID outer;   // 位置环 (较慢)
    PID inner;   // 速度环 (较快)

    struct Config {
        PIDConfig outer_cfg;
        PIDConfig inner_cfg;
        float ff_gain = 0.0f;  // 速度前馈增益
    };

    void configure(const Config& cfg) {
        outer.configure(cfg.outer_cfg);
        inner.configure(cfg.inner_cfg);
        ff_gain_ = cfg.ff_gain;
    }

    /// 串级完整更新
    /// pos_setpoint: 期望位置
    /// pos_measured: 实测位置
    /// vel_measured: 实测速度
    /// vel_feedforward: 前馈参考速度 (可选)
    float update(float pos_setpoint, float pos_measured,
                 float vel_measured, float vel_feedforward = 0.0f) {
        // 外环: 位置 -> 速度指令
        float vel_cmd = outer.update(pos_setpoint, pos_measured)
                      + ff_gain_ * vel_feedforward;

        // 内环: 速度 -> 力矩/电流指令
        float output = inner.update(vel_cmd, vel_measured);
        return output;
    }

    void reset() { outer.reset(); inner.reset(); }

private:
    float ff_gain_ = 0.0f;
};
```

用法——云台 Yaw 轴：

```
CascadedPID::Config cfg;
// 外环 (200 Hz)
cfg.outer_cfg = {.kp=8.0f, .ki=0.0f, .kd=0.0f, .dt=0.005f,
                 .out_min=-300.0f, .out_max=300.0f};
// 内环 (1 kHz)
cfg.inner_cfg = {.kp=20.0f, .ki=1.0f, .kd=0.0f, .dt=0.001f,
                 .out_min=-30000.0f, .out_max=30000.0f,
                 .integral_max=10000.0f};
cfg.ff_gain = 1.0f;

CascadedPID gimbal;
gimbal.configure(cfg);

// 内环 ISR (1 kHz) ——仅更新内环
float torque = gimbal.inner.update(vel_cmd_cached, gyro_yaw);

// 外环 ISR (200 Hz) ——同时更新双环
float torque = gimbal.update(target_angle, encoder_yaw,
                             gyro_yaw, ref_velocity);
```

8.9 高级 PID（积分分离 + PI-D + 2 自由度）

在基础 PID 的基础上增加了第 4.5 节中所有改进特性。

```
// pid_advanced.hpp
#pragma once
#include <cmath>

struct AdvancedPIDConfig {
    float kp = 0.0f, ki = 0.0f, kd = 0.0f;
    float dt = 0.001f;
    float out_min = -1e6f, out_max = 1e6f;
    float integral_max = 1e6f;
    float d_filter_N = 10.0f;     // 微分低通滤波系数
    float setpoint_weight_b = 1.0f; // P 给定值权重 (0-1)
    float setpoint_weight_c = 0.0f; // D 给定值权重 (0=PI-D, 1=PID)
    float integral_sep_eps = 0.0f;  // 0 = 禁用, >0 = 误差阈值
};

class AdvancedPID {
public:
    AdvancedPID() = default;
```

```
    explicit AdvancedPID(const AdvancedPIDConfig& c) : cfg_(c) {}
    void configure(const AdvancedPIDConfig& c) { cfg_ = c; }

    float update(float setpoint, float measurement) {
        float error = setpoint - measurement;

        // --- 带给定值加权的比例项 (2 自由度) ---
        float p_term = cfg_.kp * (cfg_.setpoint_weight_b * setpoint
                                  - measurement);

        // --- 带积分分离的积分项 ---
        if (cfg_.integral_sep_eps <= 0.0f
            || fabsf(error) < cfg_.integral_sep_eps) {
            integral_ += cfg_.ki * error * cfg_.dt;
        }
        // 抗积分饱和: 反计算
        integral_ += anti_windup_;
        integral_ = clamp(integral_, -cfg_.integral_max,
                          cfg_.integral_max);

        // --- 带给定值加权 + 滤波微分的微分项 ---
        float d_input = cfg_.setpoint_weight_c * setpoint - measurement;
        float d_raw = -(d_input - d_prev_) / cfg_.dt;
        float alpha = cfg_.dt * cfg_.d_filter_N
                    / (1.0f + cfg_.dt * cfg_.d_filter_N);
        d_filtered_ = alpha * d_raw + (1.0f - alpha) * d_filtered_;
        float d_term = cfg_.kd * d_filtered_;
        d_prev_ = d_input;

        // --- 输出 ---
        float raw = p_term + integral_ + d_term;
        float out = clamp(raw, cfg_.out_min, cfg_.out_max);

        // 反计算抗积分饱和
        anti_windup_ = (cfg_.ki != 0.0f)
            ? (out - raw) / cfg_.ki * cfg_.dt * 0.5f
            : 0.0f;
        return out;
    }

    void reset() {
        integral_ = 0; d_filtered_ = 0; d_prev_ = 0; anti_windup_ = 0;
```

```
    }

private:
    AdvancedPIDConfig cfg_;
    float integral_ = 0, d_filtered_ = 0, d_prev_ = 0, anti_windup_ = 0;
    static float clamp(float v, float lo, float hi) {
        return v < lo ? lo : (v > hi ? hi : v);
    }
};
```

用法示例：

```
// 标准 PID（与 pid.hpp 相同）
AdvancedPIDConfig cfg;
cfg.kp=6; cfg.ki=0.3; cfg.kd=3; cfg.dt=0.001f;

// PI-D（微分作用于测量值，避免微分冲击）
cfg.setpoint_weight_c = 0.0f;  // 微分仅作用于测量值

// I-PD（P 和 D 均作用于测量值）
cfg.setpoint_weight_b = 0.0f;  // 比例仅作用于测量值
cfg.setpoint_weight_c = 0.0f;  // 微分仅作用于测量值

// 2 自由度：软化给定值响应，保持扰动抑制能力
cfg.setpoint_weight_b = 0.7f;  // 减小给定值超调
cfg.setpoint_weight_c = 0.0f;  // 无微分冲击

// 积分分离：仅在接近目标时积分
cfg.integral_sep_eps = 5.0f;   // 误差阈值（度）
```

8.10 6 自由度 IMU 扩展卡尔曼滤波器（EKF for 6-DOF IMU，姿态估计）

即用型 EKF，从 6 轴 IMU 估计横滚角、俯仰角、偏航角及陀螺仪偏置。

```
// ekf_imu.hpp — 基于 EKF 的 6 自由度 IMU 姿态估计
// 状态：[roll, pitch, yaw, bias_gx, bias_gy, bias_gz]
// 测量：加速度计 -> 横滚角、俯仰角
// 来源：控制理论实践指南
// 许可证：MIT
#pragma once
#include "mat.hpp"
#include <cmath>

class EKF_IMU {
public:
```

```
static constexpr int NX = 6;  // 状态数
static constexpr int NZ = 2;  // 测量数（由加速度计得到的横滚角、俯仰角）

using State  = Mat<NX, 1>;
using Cov    = Mat<NX, NX>;
using MeasV  = Mat<NZ, 1>;
using MeasC  = Mat<NZ, NZ>;
using KGain  = Mat<NX, NZ>;
using Hmtx   = Mat<NZ, NX>;

State x;   // [roll, pitch, yaw, bx, by, bz]
Cov   P;

Cov   Q;   // 过程噪声协方差
MeasC R;   // 测量噪声协方差

float dt = 0.001f;  // 采样周期

void init() {
    x = State::zeros();
    P = Cov::identity() * 0.1f;
    // 默认值（根据你的 IMU 进行调整）
    Q = Cov::zeros();
    Q(0,0)=0.001f; Q(1,1)=0.001f; Q(2,2)=0.001f;
    Q(3,3)=0.0001f; Q(4,4)=0.0001f; Q(5,5)=0.0001f;
    R(0,0) = 0.15f; R(1,1) = 0.15f;
}

/// 主更新函数，以 IMU 采样频率调用
/// gx,gy,gz: 陀螺仪读数 (rad/s)
/// ax,ay,az: 加速度计读数 (m/s^2)
void update(float gx, float gy, float gz,
            float ax, float ay, float az) {
    // === 预测步骤 ===
    float phi = x(0,0), theta = x(1,0);
    float wx = gx - x(3,0), wy = gy - x(4,0), wz = gz - x(5,0);

    float sp = sinf(phi), cp = cosf(phi);
    float tt = tanf(theta), ct = cosf(theta);

    // 欧拉运动学
    float phi_dot   = wx + sp*tt*wy + cp*tt*wz;
```

```
float theta_dot = cp*wy - sp*wz;
float psi_dot   = (sp/ct)*wy + (cp/ct)*wz;

// 预测状态
State xp = x;
xp(0,0) += phi_dot * dt;
xp(1,0) += theta_dot * dt;
xp(2,0) += psi_dot * dt;

// 雅可比矩阵 F (6x6)
Cov F = Cov::identity();
F(0,0) += (cp*tt*wy - sp*tt*wz)*dt;
F(0,1)  = (sp/(ct*ct)*wy + cp/(ct*ct)*wz)*dt;
F(0,3) = -dt;  F(0,4) = -sp*tt*dt;  F(0,5) = -cp*tt*dt;
F(1,0)  = (-sp*wy - cp*wz)*dt;
F(1,4) = -cp*dt;  F(1,5) = sp*dt;
F(2,0)  = (cp/ct*wy - sp/ct*wz)*dt;
float st = sinf(theta);
F(2,1)  = (sp*st/(ct*ct)*wy + cp*st/(ct*ct)*wz)*dt;
F(2,4) = -sp/ct*dt;  F(2,5) = -cp/ct*dt;

Cov Pp = F * P * F.T() + Q;

// === 更新步骤 (加速度计 -> 横滚角、俯仰角) ===
float z_roll  = atan2f(ay, az);
float z_pitch = atan2f(-ax, sqrtf(ay*ay + az*az));

MeasV z;  z(0,0) = z_roll;  z(1,0) = z_pitch;
MeasV h;  h(0,0) = xp(0,0); h(1,0) = xp(1,0);

MeasV innov;
innov(0,0) = wrap_pi(z(0,0) - h(0,0));
innov(1,0) = wrap_pi(z(1,0) - h(1,0));

// H = [1 0 0 0 0 0; 0 1 0 0 0 0]
Hmtx H = Hmtx::zeros();
H(0,0) = 1.0f; H(1,1) = 1.0f;

MeasC S = H * Pp * H.T() + R;
KGain Kk = Pp * H.T() * S.inverse();

x = xp + Kk * innov;
```

```
        P = (Cov::identity() - Kk * H) * Pp;
    }

    float roll()  const { return x(0,0); }
    float pitch() const { return x(1,0); }
    float yaw()   const { return x(2,0); }
    float bias_x() const { return x(3,0); }
    float bias_y() const { return x(4,0); }
    float bias_z() const { return x(5,0); }

private:
    static float wrap_pi(float a) {
        while (a >  3.14159265f) a -= 6.28318530f;
        while (a < -3.14159265f) a += 6.28318530f;
        return a;
    }
};
```

用法（Usage）：

```
EKF_IMU ekf;
ekf.dt = 0.002f;  // IMU 采样频率 500 Hz
ekf.init();

// 针对你的 IMU 进行调参（MPU6050 示例）：
ekf.Q(0,0) = 0.001f;  ekf.Q(1,1) = 0.001f;  ekf.Q(2,2) = 0.001f;
ekf.Q(3,3) = 1e-5f;   ekf.Q(4,4) = 1e-5f;   ekf.Q(5,5) = 1e-5f;
ekf.R(0,0) = 0.1f;    ekf.R(1,1) = 0.1f;

// 在 IMU 中断中（500 Hz）：
ekf.update(gyro_x, gyro_y, gyro_z, accel_x, accel_y, accel_z);

float roll_deg  = ekf.roll()  * 180.0f / M_PI;
float pitch_deg = ekf.pitch() * 180.0f / M_PI;
float yaw_deg   = ekf.yaw()   * 180.0f / M_PI;
```

8.11 TinyMPC 求解器（TinyMPC Solver）

基于 Riccati 方法的最小化嵌入式 MPC 求解器（参见第 6 节理论部分）。可在微控制器上实时求解带盒式约束的 LQR 问题，基于 TinyMPC（Nguyen 等，2024）。

```
// tiny_mpc.hpp — 适用于嵌入式系统的基于 Riccati 的 MPC
// 基于：TinyMPC (Nguyen et al., 2024)
// 离线预计算 Riccati 递推，在线用 ADMM 求解。
#pragma once
```

```
#include "mat.hpp"

template<int NX, int NU, int N_HORIZON>
class TinyMPC {
public:
    using StateVec = Mat<NX, 1>;
    using InputVec = Mat<NU, 1>;
    using StateMat = Mat<NX, NX>;
    using InputMat = Mat<NX, NU>;
    using GainMat  = Mat<NU, NX>;

    // 系统模型（离散）
    StateMat Ad;
    InputMat Bd;

    // 代价矩阵
    StateMat Q;          // 状态代价
    Mat<NU,NU> R;        // 输入代价
    StateMat Q_terminal; // 终端代价（设为 DARE 解）

    // 约束
    InputVec u_min, u_max;
    StateVec x_min, x_max;

    // ADMM 参数
    float rho = 1.0f;        // 惩罚参数
    int   max_iter = 10;      // ADMM 迭代次数（典型值 5-20）

    // 预计算的 Riccati 增益（离线调用 precompute() 一次）
    GainMat  K_gains[N_HORIZON];    // 反馈增益
    StateMat P_mats[N_HORIZON + 1]; // 代价函数矩阵

    /// 预计算 Riccati 递推（调用一次，或在模型变化时重新调用）
    void precompute() {
        P_mats[N_HORIZON] = Q_terminal;
        for (int k = N_HORIZON - 1; k >= 0; --k) {
            // P = Q + A'*P_next*A - A'*P_next*B*(R+B'*P_next*B)^-1
            //     * B'*P_next*A
            auto P_next = P_mats[k + 1];
            auto BtP = Bd.T() * P_next;
            auto S = R + BtP * Bd;  // NU x NU
            auto S_inv = S.inverse();
```

```
            K_gains[k] = S_inv * BtP * Ad;  // NU x NX
            auto AtP = Ad.T() * P_next;
            P_mats[k] = Q + AtP * Ad - AtP * Bd * K_gains[k];
        }
    }

    /// 针对当前状态求解 MPC，返回最优首步输入
    InputVec solve(const StateVec& x0,
                   const StateVec x_ref[N_HORIZON]) {
        // 用无约束 LQR 展开初始化
        StateVec x_traj[N_HORIZON + 1];
        InputVec u_traj[N_HORIZON];
        x_traj[0] = x0;

        for (int k = 0; k < N_HORIZON; ++k) {
            StateVec dx = x_traj[k] - x_ref[k];
            u_traj[k] = (K_gains[k] * dx) * (-1.0f);
            // 截断（投影步骤）
            for (int i = 0; i < NU; ++i) {
                float v = u_traj[k](i, 0);
                v = v < u_min(i,0) ? u_min(i,0) :
                    (v > u_max(i,0) ? u_max(i,0) : v);
                u_traj[k](i, 0) = v;
            }
            x_traj[k + 1] = Ad * x_traj[k] + Bd * u_traj[k];
        }

        // ADMM 精化以满足约束
        InputVec z[N_HORIZON];      // 松弛变量
        InputVec lambda[N_HORIZON]; // 对偶变量
        for (int k = 0; k < N_HORIZON; ++k) {
            z[k] = u_traj[k];
            lambda[k] = InputVec::zeros();
        }

        for (int iter = 0; iter < max_iter; ++iter) {
            // 前向传播：带惩罚的无约束求解
            x_traj[0] = x0;
            for (int k = 0; k < N_HORIZON; ++k) {
                StateVec dx = x_traj[k] - x_ref[k];
                // 含 ADMM 惩罚的改进 Riccati
                InputVec u_unc = (K_gains[k] * dx) * (-1.0f);
```

```
                // ADMM 修正
                u_traj[k] = u_unc + (z[k] - lambda[k]) * (rho * 0.5f);
                x_traj[k+1] = Ad * x_traj[k] + Bd * u_traj[k];
            }
            // z 更新：投影到约束集
            for (int k = 0; k < N_HORIZON; ++k) {
                for (int i = 0; i < NU; ++i) {
                    float v = u_traj[k](i,0) + lambda[k](i,0);
                    v = v < u_min(i,0) ? u_min(i,0) :
                        (v > u_max(i,0) ? u_max(i,0) : v);
                    z[k](i, 0) = v;
                }
            }
            // 对偶变量更新
            for (int k = 0; k < N_HORIZON; ++k) {
                lambda[k] = lambda[k] + u_traj[k] - z[k];
            }
        }
        return u_traj[0];
    }
};
```

用法——底盘轨迹跟踪：

```
// 二维位置控制：状态=[x,y,vx,vy]，输入=[ax,ay]
TinyMPC<4, 2, 10> mpc;  // 4 状态，2 输入，预测步长=10
float dt = 0.01f;

// 离散双积分器模型
mpc.Ad = Mat<4,4>::identity();
mpc.Ad(0,2) = dt; mpc.Ad(1,3) = dt;
mpc.Bd(2,0) = dt; mpc.Bd(3,1) = dt;

// 代价权重
mpc.Q = Mat<4,4>::identity() * 10.0f;
mpc.R = Mat<2,2>::identity() * 1.0f;
mpc.Q_terminal = Mat<4,4>::identity() * 100.0f;

// 约束：最大加速度 5 m/s^2
mpc.u_min(0,0) = -5; mpc.u_min(1,0) = -5;
mpc.u_max(0,0) =  5; mpc.u_max(1,0) =  5;

mpc.precompute();  // 启动时调用一次
```

```
// 在控制循环中:
Mat<4,1> x_ref[10];  // 由路径规划器填充
auto u = mpc.solve(current_state, x_ref);
set_accel(u(0,0), u(1,0));
```

8.12 三次贝塞尔曲线轨迹（Cubic Bézier Trajectory）

通过四个控制点生成平滑的二维路径。适用于全向底盘路径规划和末端执行器轨迹。

```
// bezier.hpp -- Cubic Bezier curve for 2D trajectory generation
#pragma once

struct Vec2 { float x, y; };

class CubicBezier {
public:
    Vec2 p0, p1, p2, p3;  // Control points

    /// Evaluate position at parameter t in [0, 1]
    Vec2 position(float t) const {
        float u = 1.0f - t;
        float u2 = u * u, t2 = t * t;
        float u3 = u2 * u, t3 = t2 * t;
        return {
            u3*p0.x + 3*u2*t*p1.x + 3*u*t2*p2.x + t3*p3.x,
            u3*p0.y + 3*u2*t*p1.y + 3*u*t2*p2.y + t3*p3.y
        };
    }

    /// Evaluate first derivative (tangent) at parameter t
    Vec2 tangent(float t) const {
        float u = 1.0f - t;
        return {
            3*(u*u*(p1.x-p0.x) + 2*u*t*(p2.x-p1.x) + t*t*(p3.x-p2.x)),
            3*(u*u*(p1.y-p0.y) + 2*u*t*(p2.y-p1.y) + t*t*(p3.y-p2.y))
        };
    }

    /// Evaluate second derivative (curvature direction)
    Vec2 acceleration(float t) const {
        float u = 1.0f - t;
        return {
            6*(u*(p2.x - 2*p1.x + p0.x) + t*(p3.x - 2*p2.x + p1.x)),
            6*(u*(p2.y - 2*p1.y + p0.y) + t*(p3.y - 2*p2.y + p1.y))
```

```
        };
    }

    /// Approximate arc length by summing N linear segments
    float arc_length(int N = 100) const {
        float len = 0;
        Vec2 prev = p0;
        for (int i = 1; i <= N; ++i) {
            Vec2 cur = position(static_cast<float>(i) / N);
            float dx = cur.x - prev.x, dy = cur.y - prev.y;
            len += sqrtf(dx*dx + dy*dy);
            prev = cur;
        }
        return len;
    }
};
```

使用示例——全向底盘路径：

```
CubicBezier path;
path.p0 = {0, 0};          // Start
path.p1 = {1.0f, 0};       // Exit direction: forward
path.p2 = {2.0f, 1.5f};    // Entry direction at end
path.p3 = {3.0f, 1.5f};    // End point

// Track with 50 samples
for (int i = 0; i <= 50; ++i) {
    float t = static_cast<float>(i) / 50;
    Vec2 pos = path.position(t);
    Vec2 vel = path.tangent(t);  // For feedforward heading
    // Send (pos.x, pos.y) to chassis controller
}
```

8.13 模块依赖关系图（Module Dependency Map）

模块	文件	依赖
低通滤波器	lpf.hpp	（无）
PID 控制器	pid.hpp	（无）
高级 PID	pid_advanced.hpp	（无）
给定值斜坡	ramp.hpp	lpf.hpp
互补滤波器	complementary_filter.hpp	（无）
矩阵库	mat.hpp	（无）
卡尔曼滤波器	kalman.hpp	mat.hpp
EKF IMU（6 自由度）	ekf_imu.hpp	mat.hpp
LQR 控制器	lqr.hpp	mat.hpp
TinyMPC 求解器	tiny_mpc.hpp	mat.hpp
串级 PID	cascaded_pid.hpp	pid.hpp
三次贝塞尔曲线	bezier.hpp	（无）

所有模块均采用 MIT 许可证。将其复制到项目的 `include/control/` 目录下，`#include` 后即可使用。

结语

如果你已读到这里，恭喜——你已经为设计竞赛机器人的控制系统打下了坚实的理论基础。以下是一张速查表，供你在不同场景下快速选用合适的工具：

场景	工具
传感器数据噪声大	低通滤波器（第 2 节）
电机实际特性是什么?	电机建模（第 3 节）
单电机转速/位置控制	PID（第 4 节）
云台跟踪	PID + 前馈（第 4 节）
平滑点对点运动	轨迹生成（第 6 节）
将 PID 从纸面移植到 MCU	Tustin 离散化（第 5 节）
平衡小车 / MIMO 系统	LQR（第 6 节）
带约束的轨迹跟踪	MPC（第 6 节）
噪声传感器的状态估计	卡尔曼滤波器（第 6 节）
IMU + 编码器融合	扩展卡尔曼滤波器（第 6 节）
机器人翻转/翻滚超过 $\pm 90°$	四元数 EKF（第 6 节）

请记住：理论是工具，而非目的。**最好的控制器是能在赛场上正常运行的那一个**，而不是在纸上看起来最优雅的那一个。从简单的方法（PID）开始，理解你的系统，只在必要时增加复杂度，并且始终、始终在真实硬件上进行测试。

祝你在下次比赛中好运。去造出令人惊叹的东西吧。