tcb/box tcb/boxSect

tcb/box/begin0 tcb/box/begindefault

tag=tcb/box tcb/box/begindefault

tcb/box/end0 tcb/box/enddefault

tcb/box/enddefault

tcb/box/title tcb/box/titleCaption

tcb/box/title0 tcb/box/titledefault para/semantictcb/box/title tcb/box/titledefault

tcb/box/draw2 tcb/box/drawdefault 2

tcb/box/drawdefault

tcb/box/init0 tcb/box/initdefault minipage/beforenoopminipage/afternoop tcb/box/initdefault

tcb/box/upper0 tcb/box/upperdefault minipage/beforetag/dfltminipage/aftertag/dflt tcb/box/upperdefault

tcb/box/lower0 tcb/box/lowerdefault minipage/beforetag/dfltminipage/aftertag/dflt tcb/box/lowerdefault

tcb/drawing/init0tcb/drawing/initdefaulttcb/drawing/initdefault

# 实验配置与控制策略调参指南

如何使用 `Week 4_me/Code` 调配仿真实验

Andrea Zhang — MATH3060/1 Group Project

2026 年 2 月

## 目录

# Part I

# 基础概念：代码架构与实验流程

## 1 文件夹结构总览

本实验工具箱已自包含在 `Week 4_me/` 文件夹下，不依赖上级目录。

**文件夹结构**

```
Week 4_me/
  Code/                          # 自包含代码副本
    models/
      pde_2d_model.py            # 核心：2D 热方程求解器
      ode_model.py               # ODE 模型（0D）
      pde_1d_model.py            # 1D PDE 模型
    controllers/
      bang_bang.py               # Bang-Bang 控制（基线）
      pid.py                     # PID 控制（连续反馈）
      lqr.py                     # LQR 最优控制
      pontryagin.py              # Pontryagin 开环最优
    utils/
      parameters.py              # 统一物理参数
      metrics.py                 # 评价指标（RMSE, 能耗等）
      plotting.py                # 绘图工具
    tests/                       # 单元测试
    experiments/                 # 已有的实验脚本
  scenarios.py                   # 6 个场景的配置工厂
  run_scenarios.py               # 统一实验运行脚本
  Scenario_Exploration.ipynb     # 教学展示 Notebook
  results/                       # 输出图表
  research_note/                 # 本文档所在目录
```

**设计哲学：模型–控制器–场景三层解耦**

整个代码的设计遵循**三层分离**原则：

1. **模型层**（`models/`）：定义物理方程和求解方法，不知道控制器是什么
2. **控制层**（`controllers/`）：定义控制策略 $u(t, T)$，不知道空间在几维
3. **场景层**（`scenarios.py`）：定义边界条件参数，不知道用哪种控制器

三者通过统一接口 `u_func(t, T_therm)` → `float` 连接。更换控制器只需**改一行代码**。

## 2 运行一次实验的完整流程

**从零到图的四步流程**

**Step 1 选择场景**：调用 `scenarios.py` 中的工厂函数，获取配置字典 `cfg`

**Step 2 构建模型**：用 `cfg` 构建 `HeatEquation2D` 实例

**Step 3 选择控制器**：实例化控制器，包装为 `u_func(t, T_therm)`

**Step 4 运行仿真**：调用 `model.simulate(u_func)`，获取 $(t, T_{\text{field}}, T_{\text{therm}})$

最小可运行示例（在 `Week 4_me/` 目录下）：

```python
import sys; sys.path.insert(0, './Code')
import numpy as np
from models.pde_2d_model import HeatEquation2D
from controllers.bang_bang import BangBangController
from scenarios import make_baseline

# Step 1: 选场景
cfg = make_baseline()

# Step 2: 建模型
model = HeatEquation2D(
    Lx=cfg['Lx'], Ly=cfg['Ly'],
    nx=cfg.get('nx', 51), ny=cfg.get('ny', 41),
    heater_pos=cfg['heater_pos'],
    thermostat_pos=cfg['thermostat_pos'],
    wall_h=cfg.get('wall_h'),
    domain_mask=cfg.get('domain_mask'),
    h_updater=cfg.get('h_updater'),
)

# Step 3: 选控制器
ctrl = BangBangController()
def u_func(t, T_therm):
    u = ctrl.get_u(t, T_therm)
    if ctrl._on and T_therm > ctrl.T_set + ctrl.delta:
        ctrl.switch(t, T_therm)
    elif not ctrl._on and T_therm < ctrl.T_set - ctrl.delta:
        ctrl.switch(t, T_therm)
    return u

# Step 4: 跑仿真
t, T_field, T_therm = model.simulate(u_func, t_end=120)
print(f"Final thermostat: {T_therm[-1]:.1f} C")
```

**快速运行命令**

也可以直接用命令行：

```
# 跑所有 6 个场景（完整 120 min，约需 5-10 分钟）
python run_scenarios.py

# 只跑 S1 和 S2
python run_scenarios.py S1 S2

# 快速测试模式（30 min，约需 1 分钟）
python run_scenarios.py --quick
```

# Part II

# 场景配置：如何修改物理参数

## 3 场景配置字典的结构

每个场景由一个 Python 字典（dict）描述，包含以下键：

**场景配置字典的键说明**

| 键 | 类型 | 含义 | 是否必须 |
|---|---|---|---|
| `name` | str | 场景标识名 | 是 |
| `Lx`, `Ly` | float | 房间长、宽 (m) | 是 |
| `nx`, `ny` | int | $x$, $y$ 方向网格点数 | 否 (默认 51, 41) |
| `wall_h` | dict | 四面墙的 $h$ 数组 | 否 (默认全 0.5) |
| `heater_pos` | (x, y) | 加热器位置 | 是 |
| `thermostat_pos` | (x, y) | 恒温器位置 | 是 |
| `domain_mask` | ndarray (bool) | 域 mask (L 形等) | 否 (默认全 True) |
| `h_updater` | callable | 时变 BC 回调 | 否 (默认无) |

## 4 修改场景参数的常见操作

### 4.1 操作 1：改变窗户大小和材质

打开 `scenarios.py`，找到 `make_window()` 函数：

```
def make_window():
```

```
    h_south = np.full(NX, H_WALL)  # 先全填 0.5
    x = np.linspace(0, 5, NX)
    # 在 x in [1.5, 3.5] 设为窗户 h=2.5
    h_south[(x >= 1.5) & (x <= 3.5)] = 2.5
    # ^^^^^^^^  改这两个数字调窗户位置
    # ^^^^^^^^^^^^^^^^^^^^^^^^^^^^  改这个值调窗户材质
    ...
```

**常用窗户 $h$ 值参考**

| 窗户类型 | $h$ ($\mathrm{m}^{-1}$) | 来源 |
|---|---|---|
| 单层玻璃 | 2.5 | MEP Academy, 2024 |
| 双层中空玻璃 | 1.0 | Incropera et al., 2007 |
| 三层 Low-E 玻璃 | 0.6 | 估计值 |
| 普通外墙 (基准) | 0.5 | 项目设定 |

### 4.2 操作 2：改变房间形状

房间形状由 Lx, Ly 和 domain_mask 共同决定。

```
# 例：创建一个 10x3 米的走廊
def make_corridor():
    nx, ny = 101, 31
    wall_h = {
        'south': np.full(nx, 0.5),  # 外墙
        'north': np.full(nx, 0.5),  # 外墙 (两面朝外)
        'west':  np.full(ny, 0.0),
        'east':  np.full(ny, 0.0),
    }
    return dict(name="Corridor_10x3", Lx=10, Ly=3,
                nx=nx, ny=ny, wall_h=wall_h,
                heater_pos=(5.0, 1.5),
                thermostat_pos=(5.0, 1.5))
```

**网格密度一致性**

改变房间尺寸时，务必同步调整 nx, ny 以保持网格密度 $\Delta x \approx 0.1$ m：

$$n_x = \frac{L_x}{0.1} + 1, \qquad n_y = \frac{L_y}{0.1} + 1 \tag{1}$$

否则可能因网格太粗导致数值不稳定，或因网格太密导致运行时间过长。

### 4.3 操作 3：调整加热器和恒温器位置

```
# 在 make_baseline() 或任何场景工厂中修改：
heater_pos=(1.0, 0.5),      # 加热器：(x, y) 坐标
thermostat_pos=(4.0, 4.0),  # 恒温器：(x, y) 坐标
```

**加热器位置的物理含义**

heater_pos 指定了一个 2D Gaussian 热源的中心位置。热源的空间分布为

$$S(x,y) = \frac{u(t) \cdot G(x,y)}{L_x L_y}, \qquad G(x,y) = \exp\left(-\frac{(x-x_h)^2 + (y-y_h)^2}{2\sigma^2}\right) \tag{2}$$

其中 $\sigma = 0.5$ m（默认半径）。加热器 “辐射” 范围约 $2\sigma = 1$ m。

**位置选择原则**：

- 加热器靠近外墙 → 直接对抗热损失，但加热器附近过热
- 加热器在房间中心 → 温度更均匀，但外墙附近偏冷
- 恒温器应代表 “人体感知温度”，通常放在房间中部

### 4.4 操作 4：创建新的不规则房间

```
# 例：T 形房间
def make_T_shape():
    nx, ny = 51, 51
    x = np.linspace(0, 5, nx)
    y = np.linspace(0, 5, ny)
    X, Y = np.meshgrid(x, y, indexing='ij')

    mask = np.zeros((nx, ny), dtype=bool)
    # 下部横条：x in [0,5], y in [0,2]
    mask[(Y >= 0) & (Y <= 2)] = True
    # 上部竖条：x in [1.5,3.5], y in [2,5]
    mask[(X >= 1.5) & (X <= 3.5) & (Y >= 2)] = True

    wall_h = {
        'south': np.full(nx, 0.5),
        'north': np.full(nx, 0.0),
        'west':  np.full(ny, 0.0),
        'east':  np.full(ny, 0.0),
    }
    return dict(name="T_Shape", Lx=5, Ly=5, nx=nx, ny=ny,
                wall_h=wall_h, domain_mask=mask,
                heater_pos=(2.5, 1.0),
                thermostat_pos=(2.5, 1.0))
```

### 4.5 操作 5：添加时变边界条件

```python
# 例：东墙的通风口，在 t=50~70 min 打开
def make_east_vent(t_open=50, t_close=70):
    def h_updater(t, model):
        h_east = np.full(model.ny, 0.0)
        if t_open <= t <= t_close:
            y = np.linspace(0, model.Ly, model.ny)
            h_east[(y >= 1.0) & (y <= 3.0)] = 5.0
        model.h_east = h_east
    ...
    return dict(..., h_updater=h_updater)
```

**h_updater 的注意事项**

1. Callback 在**每次 RHS 求值时调用**（一次仿真可能调用数万次），应尽量轻量
2. 修改的是 `model.h_east` 等属性（直接赋值），不需要返回值
3. $h$ 在 $t_{\text{open}}$ 处的突变会让求解器自动缩小步长，这是正确行为

# Part III
# 控制策略切换与调参

## 5 四种控制器的数学定义

**四种控制策略一览**

| 策略 | 控制律 $u(t,T)$ | 关键参数 | 定位 |
|---|---|---|---|
| Bang-Bang | $u=\begin{cases}U_{\max} & T<T_{\text{set}}-\delta\\ 0 & T>T_{\text{set}}+\delta\end{cases}$ | $\delta$ (滞回带宽) | 基线 |
| PID | $u=K_pe+K_i\int e\,\mathrm{d}t+K_d\dot{e}$ | $K_p,K_i,K_d$ | 工程标准 |
| LQR | $u=-K(T-T_{\text{set}})+u_{\text{ss}}$ | $Q,R$ (代价权重) | 数学最优 |
| Pontryagin | $u^*(t)=\text{clip}\left(-\frac{\lambda(t)}{2R},0,U_{\max}\right)$ | $Q,R,t_f$ | 理论极限 |

**统一接口：get_u(t, T) → float**

所有控制器都实现了相同的方法签名：

```
class SomeController:
    def get_u(self, t, T) -> float:
        """输入：当前时间 t，恒温器温度 T
           输出：加热功率 u in [0, U_max]"""
```

因此切换控制器**只需改一行代码**，模型层和场景层完全不用动。

# 6 如何切换控制器

## 6.1 在 run_scenarios.py 中切换

打开 run_scenarios.py，找到 run_one() 函数中的控制器实例化：

```
def run_one(cfg, t_end=T_END):
    model = build_model(cfg)
    # ===== 在这里切换控制器 =====
    ctrl = BangBangController()           # 当前：Bang-Bang
    # ctrl = PIDController(Kp=2.0, Ki=0.1, Kd=0.5, dt=0.5)  # PID
    # ctrl = LQRController(Q=1.0, R=0.01)                    # LQR
    ...
```

**Bang-Bang 需要额外的滞回逻辑**

Bang-Bang 控制器使用事件检测的滞回机制，在 run_one() 中 u_func 包含了手动的 switch 逻辑。如果切换到 PID 或 LQR，u_func 可以简化为：

```
def u_func(t, T_therm):
    return ctrl.get_u(t, T_therm)  # PID/LQR 无需手动 switch
```

## 6.2 在 Notebook 中切换

Scenario_Exploration.ipynb 中的 run_scenario() 函数与 run_one() 逻辑相同，改同一处即可。

## 6.3 同时对比多个控制器

```
# 对同一个场景用不同控制器跑，然后对比
from controllers.bang_bang import BangBangController
from controllers.pid import PIDController
from controllers.lqr import LQRController

cfg = make_baseline()
controllers = {
```

```
    'Bang-Bang': BangBangController(),
    'PID':       PIDController(Kp=2.0, Ki=0.1, Kd=0.5, dt=0.5),
    'LQR':       LQRController(Q=1.0, R=0.01),
}

for name, ctrl in controllers.items():
    model = build_model(cfg)
    def u_func(t, T, _ctrl=ctrl):  # 闭包捕获 ctrl
        return _ctrl.get_u(t, T)
    t, T_field, T_therm = model.simulate(u_func, t_end=120)
    print(f"{name}: Final T_therm = {T_therm[-1]:.1f} C")
```

# 7 Bang-Bang 控制器调参

## 7.1 数学模型

带滞回的 Bang-Bang 控制器是一个**混合动力系统** Goebel et al., 2009：

$$u(t) = \begin{cases} U_{\max} & \text{if heater is ON} \\ 0 & \text{if heater is OFF} \end{cases} \tag{3}$$

切换条件：

$$\text{ON} \to \text{OFF}: \quad T > T_{\text{set}} + \delta \tag{4}$$

$$\text{OFF} \to \text{ON}: \quad T < T_{\text{set}} - \delta \tag{5}$$

## 7.2 可调参数

**Bang-Bang 参数调整指南**

| 参数 | 默认值 | 调大的效果 | 调小的效果 |
|---|---|---|---|
| `delta` ($\delta$) | 0.5°C | 切换少，温度波动大 | 切换频繁，接近 Zeno |
| `U_max` ($U_{\max}$) | 15 | 升温快，超调大 | 升温慢，可能达不到设定 |
| `T_set` ($T_{\text{set}}$) | 20°C | 更高目标温度 | 更低目标温度 |

```
# 调参示例
ctrl = BangBangController(
    T_set=22.0,    # 提高设定温度
    U_max=15,      # 保持默认功率
    delta=1.0,     # 加宽滞回带 -> 减少开关次数
)
```

**滞回带宽 $\delta$ 的权衡**

$\delta$ 控制了**舒适度 vs 设备磨损**的权衡：

- $\delta = 0$: 温度精确跟踪 $T_{\text{set}}$，但切换无穷频繁（**Zeno 效应**，见 Zhang et al., 2001）
- $\delta = 0.5$: 温度在 $[T_{\text{set}} - 0.5, T_{\text{set}} + 0.5]$ 范围内波动，切换次数适中
- $\delta = 2.0$: 温度在 $[18, 22]$°C 范围内大幅波动，切换很少

实际恒温器通常取 $\delta \approx 0.5$–$1.0$°C。

# 8 PID 控制器调参

## 8.1 数学模型

离散 PID 控制律 Åström and Murray, 2021：

$$u(t) = \underbrace{K_p \cdot e(t)}_{\text{比例}} + \underbrace{K_i \int_0^t e(\tau)\,\mathrm{d}\tau}_{\text{积分}} + \underbrace{K_d \frac{\mathrm{d}e}{\mathrm{d}t}}_{\text{微分}}, \quad e(t) = T_{\text{set}} - T(t) \tag{6}$$

输出限幅至 $[0, U_{\max}]$，带 **anti-windup**（积分饱和时停止积分累积）。

## 8.2 三个增益参数的物理含义

**PID 三参数的作用**

| 参数 | 作用 | 过大 | 过小 |
|---|---|---|---|
| $K_p$ | 放大当前误差 | 响应快但振荡 | 响应慢、稳态误差大 |
| $K_i$ | 消除稳态误差 | 超调严重、积分饱和 | 仍有残余误差 |
| $K_d$ | 抑制变化率 (阻尼) | 对噪声敏感 | 振荡不易消除 |

## 8.3 调参方法

**PID 调参三步法**

1. **先调 $K_p$**：令 $K_i = K_d = 0$，逐渐增大 $K_p$ 直到系统快速响应但不振荡
2. **再加 $K_i$**：逐渐增大 $K_i$，消除稳态误差，但不要太大以免超调
3. **最后加 $K_d$**：增大 $K_d$ 以抑制超调和振荡

```
# 方法 1: 手动调参
ctrl = PIDController(Kp=3.0, Ki=0.05, Kd=1.0, dt=0.5)

```

```python
# 方法 2: Ziegler-Nichols 自动估计
from controllers.pid import ziegler_nichols_tuning
Kp, Ki, Kd = ziegler_nichols_tuning(k_cool=0.1)
print(f"Z-N: Kp={Kp:.2f}, Ki={Ki:.2f}, Kd={Kd:.2f}")
ctrl = PIDController(Kp=Kp, Ki=Ki, Kd=Kd, dt=0.5)
```

**Ziegler-Nichols 调参的原理**

代码中的 `ziegler_nichols_tuning(k_cool)` 基于房间 ODE 模型 $\mathrm{d}T/\mathrm{d}t = -k(T - T_\mathrm{a}) + u$ 的时间常数 $\tau = 1/k$：

$$K_p = 0.6 \cdot U_\mathrm{max}/15 \approx 0.6 \tag{7}$$

$$K_i = K_p/(0.5\tau) = K_p \cdot 2k \tag{8}$$

$$K_d = K_p \cdot 0.125\tau \tag{9}$$

这只是起始估计值，通常还需要根据 2D PDE 模型的响应做微调。

**PID 用于 2D PDE 时的注意事项**

PID 是为 ODE（0D 模型）设计的。在 2D PDE 中，恒温器只读取**一个点**的温度，但控制器的输出影响整个温度场。这意味着：

- 恒温器位置对 PID 性能影响巨大（比 Bang-Bang 更敏感）
- $K_d$ 的微分作用可能被 2D 扩散效应 “稀释”，需要适当增大
- `dt` 参数应与仿真的时间步一致（默认 0.5 min = `t_eval` 的间隔）

# 9 LQR 控制器调参

## 9.1 数学模型

LQR 最小化二次代价泛函 Anderson and Moore, 1990：

$$J = \int_0^\infty \left[ Q \cdot (T - T_\mathrm{set})^2 + R \cdot u^2 \right] \mathrm{d}t \tag{10}$$

最优反馈律：

$$\boxed{u^* = -K \cdot (T - T_\mathrm{set}) + u_\mathrm{ss}} \tag{11}$$

其中 $K$ 由 Riccati 方程求得，$u_\mathrm{ss} = k(T_\mathrm{set} - T_\mathrm{a})$ 是稳态前馈项。

## 9.2 两个权重参数的物理含义

**$Q$ 和 $R$ 的直觉理解**

- $Q$ = **"温度偏差有多贵"**：$Q$ 越大，控制器越不能容忍偏离设定值
- $R$ = **"用电有多贵"**：$R$ 越大，控制器越 "省电" 但允许温度偏差
- **核心权衡**：$Q/R$ 比值决定了 "舒适 vs 节能" 的平衡点

**LQR $Q/R$ 参数效果**

| $Q$ | $R$ | 行为 | 适用场景 |
|---|---|---|---|
| 1.0 | 0.01 | 激进追踪，高能耗 | 手术室、精密实验室 |
| 1.0 | 0.1 | 平衡模式 | 普通办公室 |
| 1.0 | 1.0 | 节能模式，温度波动大 | 仓库、无人区域 |

```python
# 基础用法
ctrl = LQRController(Q=1.0, R=0.01)  # 默认：激进追踪

# Pareto 参数扫描：探索 Q/R 空间
from controllers.lqr import pareto_scan
Q_vals = [0.1, 1.0, 10.0]
R_vals = [0.001, 0.01, 0.1, 1.0]
ctrls, params = pareto_scan(Q_vals, R_vals)
# ctrls[i] 对应 params[i] = (Q, R)
```

**Riccati 方程的标量形式**

对于一阶 ODE 模型 $\dot{x} = -kx + u$（$x = T - T_{\text{set}}$），代数 Riccati 方程为

$$-kP + P(-k) - \frac{P^2}{R} + Q = 0 \tag{12}$$

解为

$$P = R\left(-k + \sqrt{k^2 + Q/R}\right), \qquad K = P/R = -k + \sqrt{k^2 + Q/R} \tag{13}$$

当 $Q/R \gg k^2$ 时，$K \approx \sqrt{Q/R}$（高增益反馈）。

**注意**：这个 $K$ 是基于 ODE 模型计算的。将其用于 2D PDE 时，控制器只看恒温器一点的温度，与 ODE 中 "看整间房温度" 不同，因此效果会有差异。

## 10 Pontryagin 最优控制（开环）

### 10.1 与其他控制器的本质区别

**Pontryagin: 开环 vs 闭环**

Pontryagin 控制器是**预计算的开环控制**：

- 求解两点边值问题（TPBVP），得到整条最优轨迹 $u^*(t)$
- 运行时直接**插值查表**，不读取当前温度
- 对模型扰动（开门、窗户变化）**没有反馈能力**

因此 Pontryagin 适合作为 “理论最优基准”，不适合实际控制。

```python
from controllers.pontryagin import PontryaginController

# 默认参数（基于 ODE 模型）
ctrl = PontryaginController(Q=1.0, R=0.01, t_end=120)
# 内部自动求解 TPBVP，可能需要几秒

# 使用方式与其他控制器完全相同
u = ctrl.get_u(t=30, T=18.5)  # 但实际上忽略了 T，直接插值
```

# Part IV
# 评价指标与结果分析

## 11 六项评价指标

所有控制策略用统一指标对比（定义在 `Code/utils/metrics.py`）：

**评价指标体系**

| 指标 | 公式 | 含义 |
|---|---|---|
| 能耗 $E$ | $\int_0^{t_f} u(t)\,\mathrm{d}t$ | 总加热能量 |
| 温度偏差 RMSE | $\sqrt{\frac{1}{t_f}\int_0^{t_f}(T-T_{\text{set}})^2\,\mathrm{d}t}$ | 平均偏离程度 |
| 最大超调 | $\max(T(t)-T_{\text{set}},0)$ | 峰值过热 |
| 稳定时间 $t_s$ | 首次进入 $T_{\text{set}}\pm0.5$°C 并保持 | 响应速度 |
| 切换次数 $N_s$ | 加热器 ON↔OFF 的次数 | 设备磨损 |
| 统一代价 $J$ | $\int_0^{t_f}\left[Q(T-T_{\text{set}})^2+Ru^2\right]\,\mathrm{d}t$ | 综合权衡 |

```python
from utils.metrics import compute_all_metrics
from utils.parameters import T_SET

metrics = compute_all_metrics(t, T_therm, u_arr, T_SET)
print(f"Energy:     {metrics['energy']:.1f}")
print(f"RMSE:       {metrics['rmse']:.2f} C")
print(f"Overshoot:  {metrics['max_overshoot']:.2f} C")
print(f"Settle:     {metrics['settling_time']:.1f} min")
print(f"Switches:   {metrics['switching_count']}")
print(f"Cost J:     {metrics['unified_cost']:.1f}")
```

## 12 空间不均匀度（PDE 特有）

对于 2D PDE 模型，温度场的空间分布不均匀度也是重要指标：

$$\sigma_T = \mathrm{std}\left(T(x,y,t_f)\right) = \sqrt{\frac{1}{|\Omega|}\int_\Omega (T-\bar{T})^2\,\mathrm{d}\Omega} \tag{14}$$

```python
# 终态温度场的空间不均匀度
T_final = T_field[:, :, -1]
mask = cfg.get('domain_mask')
if mask is not None:
    T_active = T_final[mask]
else:
    T_active = T_final.ravel()
nonuniformity = np.std(T_active)
print(f"Spatial std: {nonuniformity:.2f} C")
```

# 13 批判性分析

**控制策略选择的批判性评价**

**优势**:

- 统一接口设计使策略切换极为简单，有利于系统化对比
- 四种控制器覆盖了从最简单（Bang-Bang）到最理论（Pontryagin）的完整谱系
- 评价指标全面，涵盖了能耗、舒适度、设备寿命三个维度

**局限**:

- PID 和 LQR 的参数是基于 ODE（0D）模型设计的，直接用于 2D PDE 可能不是最优
- 所有控制器只读取**一个点**（恒温器）的温度，无法感知空间分布
- Pontryagin 是开环控制，对扰动（开门、日照变化）无反馈能力
- 未考虑执行器动态（加热器不能瞬间开关，有启动延迟）

**未解决的问题**:

- 如何为 2D PDE 模型专门设计控制器?（需要分布参数控制理论）
- 多恒温器、多加热器的协调控制问题
- 模型预测控制（MPC）是否能更好地处理时变扰动?

**对本项目的启示**：当前四种策略足以展示 “不同数学工具的应用”，但报告中应明确指出 0D 控制器用于 2D 模型的局限性，并将其作为 Discussion 部分的亮点。

# 14 快速查阅卡片

**常用命令速查**

| 操作 | 代码 |
|---|---|
| 运行全部场景 | `python run_scenarios.py` |
| 快速测试 | `python run_scenarios.py --quick` |
| 只跑 S1+S4 | `python run_scenarios.py S1 S4` |
| 切到 PID | `ctrl = PIDController(Kp=2, Ki=0.1, Kd=0.5, dt=0.5)` |
| 切到 LQR | `ctrl = LQRController(Q=1.0, R=0.01)` |
| Z-N 自动调参 | `Kp,Ki,Kd = ziegler_nichols_tuning(0.1)` |
| 计算指标 | `m = compute_all_metrics(t, T, u, T_SET)` |
| 跑测试 | `cd Code && python -m pytest tests/ -v` |

## 15 自检清单

- ☐ 能不看文档，从零运行一个场景并获得温度场图
- ☐ 能修改 `scenarios.py` 创建一个新场景（如 T 形房间）
- ☐ 能在 `run_scenarios.py` 中把 Bang-Bang 切换为 PID 并跑通
- ☐ 能解释 PID 三个参数 $K_p, K_i, K_d$ 分别控制什么
- ☐ 能解释 LQR 中 $Q/R$ 比值对控制行为的影响
- ☐ 能用 `compute_all_metrics` 计算出全部 6 项指标
- ☐ 能说出 Pontryagin 与其他三种控制器的本质区别（开环 vs 闭环）
- ☐ 能向老师解释代码的三层解耦设计

## References

Anderson, Brian D. O. and John B. Moore (1990). *Optimal Control: Linear Quadratic Methods*. Reprint: Dover, 2007. Prentice-Hall.

Åström, Karl Johan and Richard M. Murray (2021). *Feedback Systems: An Introduction for Scientists and Engineers*. 2nd ed. Princeton University Press.

Goebel, Rafal et al. (2009). "Hybrid dynamical systems". In: *IEEE Control Systems Magazine* 29.2, pp. 28–93.

Incropera, Frank P. et al. (2007). *Fundamentals of Heat and Mass Transfer*. 6th ed. John Wiley & Sons.

MEP Academy (2024). *Heat Transfer thru Walls and Windows*. https://mepacademy.com/heat-transfer-thru-walls-and-windows/.

Zhang, Jun et al. (2001). "Zeno hybrid systems". In: *International Journal of Robust and Nonlinear Control* 11.5, pp. 435–451.