钟显博

辨识过程数学推导

已知控制系统是由**一阶惯性环节和延迟环节**组成，可以写出他的传递函数：

$$G(s)=\frac{K}{1+Ts}e^{-\theta s}$$

将传递函数转换为时域的表达方式：

- 闭环传递函数

$$\begin{aligned}G(s)&=\frac{Y(s)}{U(s)}\\&=\frac{K}{1+Ts}e^{-\theta s}\end{aligned}$$

- 求出$Y(s)$

$$\begin{aligned}Y(s)&=G(s)U(s)\\&=\frac{KU}{s(1+Ts)}e^{-\theta s}\end{aligned}$$

- 求出$Y(t)$，此时$t>\theta$

$$\begin{aligned}y(t)&=KU(1-e^{-\frac{(t-\theta)}{T}})u(t-\theta)\\Y(t)&=y(t)+y_0\\&=KU(1-e^{-\frac{(t-\theta)}{T}})u(t-\theta)+y_0\end{aligned}$$

下面是使用两点法进行参数辨识的过程：

- 参数K辨识

$$\begin{aligned}\lim_{t\to\infty}Y(t)&=KU+y_0\\K&=\frac{Y(\infty)-y_0}{U}\end{aligned}$$

- 参数T、θ辨识

$$\begin{aligned}Y(t_{39})&=KU(1-e^{-\frac{(t-\theta)}{T}})+y_0\\&=y_0+0.393(Y(\infty)-y_0)\\Y(t_{63})&=KU(1-e^{-\frac{(t-\theta)}{T}})+y_0\\&=y_0+0.632(Y(\infty)-y_0)\end{aligned}$$

$$\begin{aligned}t_{39}&=\theta-T\ln(1-0.393)\\&=\theta+0.5T\\t_{63}&=\theta-T\ln(1-0.632)\\&=\theta+T\end{aligned}$$

$$\begin{aligned}T&=2(t_{63}-t_{39})\\\theta&=t_{63}-T\end{aligned}$$

智能优化算法(GWO——灰狼优化算法)

参数定义

- 最优解：α（一阶狼）
- 第二解：β（二阶狼）
- 第三解：δ（三阶狼）
- 候选解：ω（四阶狼）

探索阶段

- 随机数：A
  - A的绝对值大于1迫使群狼偏离猎物，远离领头狼
  - A的绝对值小于1迫使群狼靠近猎物，向领头狼靠近
- 随机数：C
  - 范围为$[0,2]$
  - C大于1**加强**猎物位置对灰狼下一个位置的影响
  - C小于1**降低**猎物位置对灰狼下一个位置的影响

开发阶段

$$D = |C \cdot X_p(t) - X(t)|$$
$$X(t+1) = X_p(t) - A \cdot D$$

其中，t表示当前迭代次数，A和C为系数，X_p为猎物的位置（即当前最优解的位置），$X(t)$为灰狼个体第t代的位置

A和C的计算方法

$$A = 2a \cdot r_1 - a$$
$$C = 2 \cdot r_2$$

r_1, r_2是$[0,1]$的随机值，为了模拟逼近猎物，A是区间$[-a,a]$的一个随机值，a在迭代过程从2减少到0

攻击猎物

- **探索阶段**($|A|>1$)：ω狼远离领导者
- **开发阶段**($|A|<1$)：ω狼靠近领导者

$$D_\alpha = |C_1 \cdot X_\alpha - X| \tag{5}$$
$$D_\beta = |C_2 \cdot X_\beta - X| \tag{6}$$
$$D_\delta = |C_3 \cdot X_\delta - X| \tag{7}$$

其中，(5)，(6)，(7)分别表示α狼，β狼，δ狼和其他个体之间的距离；X_α，X_β，X_δ分别表示α狼，β狼，δ狼的当前位置，C_1，C_2，C_3是随机数，X是灰狼个体当前位置

$$X_1 = |X_\alpha - A_1 \cdot (D_\alpha)| \tag{8}$$
$$X_2 = |X_\beta - A_2 \cdot (D_\beta)| \tag{9}$$
$$X_3 = |X_\delta - A_3 \cdot (D_\delta)| \tag{10}$$

其中，X_1，X_2，X_3分别表示受α狼，β狼，δ狼影响，ω狼调整后的位置。这里取平均值，即

$$X_{(t+1)} = \frac{X_1 + X_2 + X_3}{3}$$

适应度(ITAE)

$$ITAE = \int_0^{t_{sim}} t \cdot |e(t)| = \sum_{k=1}^{N} t_k \cdot |r - y(t_k)|$$

- 通过**时间加权**自然平衡快速性和稳定性
- 可以满足实际工程对动态性能的要求

算法工作流程

1. 给定**狼群数量**，一头狼有三个参数，分别是K_p, K_i, K_d
2. **随机**初始化参数，计算适应度$(ITAE)$，根据适应度选出初始的α狼，β狼，δ狼
3. 更新参数A、C，计算群狼受α狼，β狼，δ狼**影响后的位置**并取**平均值**
4. 计算适应度并更新α狼信息
5. 循环往复直至迭代次数达到上限

个人理解

- α狼是根据自适应度选出的，也就是当前几轮迭代可以选出的最优解，其是**最**靠近猎物的
- 算法通过**随机数**来决定群狼如何移动
  - 当群狼向猎物靠近，也就是向头狼靠近，有利于寻找更优解
  - 当群狼远离猎物，也就是远离头狼，有利于寻找其它方向的最优解，防止**陷入局部最优解**
- 算法是全局和局部搜索相结合，能够**减小这两种搜索单独使用的局限性**

代码剖析

两点法

- 可参考论文：https://www.doc88.com/p-3854861288561.html
- 数学推导过程见上文

```
# 系统辨识函数 - 两点法
def two_point_method(t, y):
    y_0 = y[0]
    y_ss = np.mean(y[-50:])
    y_39 = y_0 + 0.39347 * (y_ss - y_0)
    y_63 = y_0 + 0.63212 * (y_ss - y_0)

    t_39 = np.interp(y_39, y, t)
    t_63 = np.interp(y_63, y, t)

    tau =  2* (t_63 - t_39)
    theta = 2 * t_39 - t_63

    u_step = volte[0]
    K = (y_ss - y_0) / u_step
```

```
    return K, tau, theta
```

PID控制器

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

- **积分饱和**：当系统输出因物理限制(如执行器饱和)无法达到控制器要求的输出时，误差会持续存在，导致积分项不断累积变得非常大，**导致系统响应迟缓并且引起超调**
- 解决方法就是使用**条件积分**，进行积分钳位，减少不必要的积分

1. 初始化
    - 首先对K_p, K_i, K_d参数进行初始化，对**给定目标值**进行初始化，对积分项，误差项进行初始化
    - 给定积分上限，防止积分饱和
2. 更新
    - **条件积分**: 只在误差小于5时才积分，这是为了防止系统远离设定点时积分项过大
    - **积分限幅**: 使用`clip`函数限制积分项的范围，防止积分饱和

```
# PID控制器类
class PIDController:
    def __init__(self, Kp, Ki, Kd, setpoint):
        self.Kp = Kp
        self.Ki = Ki
        self.Kd = Kd
        self.setpoint = setpoint
        self.prev_error = 0
        self.integral = 0
        self.integral_limit = 50  # 降低积分限制，减少积分饱和

    def update(self, current_value, dt):
        error = self.setpoint - current_value

        # 增加比例限制减少过冲
        proportional = self.Kp * error

        # 仅在误差范围内积分
        if abs(error) < 5:  # 只在误差小于5℃时积分
            self.integral += error * dt

        # 限制积分项
        self.integral = np.clip(self.integral, -self.integral_limit, self.integral_limit)

        derivative = (error - self.prev_error) / dt
        self.prev_error = error

        return proportional + self.Ki * self.integral + self.Kd * derivative
```

灰狼优化算法

- 需要通过多次调试得出三个参数的搜索范围，否则结果会不尽人意
  - 辨识出来的时间参数很大，因此K_i要很小
  - 后来我发现**振荡严重**，因此增大K_d的搜索范围
- 同时在计算$ITAE$中加入了惩罚机制，如果超调过大会进行**严重惩罚**

```
class GreyWolfOptimizer:
    # 初始化参数
    def __init__(self, sys_params, setpoint=35, wolves=5, iterations=25,room_temp=16.8):
        self.K, self.tau, self.theta = sys_params
        self.setpoint = setpoint
        self.n_wolves = wolves
        self.max_iter = iterations
        self.room_temp = room_temp

        # 参数搜索范围
        self.bounds = np.array([
            [1.0, 50.0],  # Kp范围
            [0.01, 10.0],  # Ki范围
            [5.0, 200.0]  # Kd范围
        ])

        # 初始化狼群
        # 三个维度是因为有三个参数要调整，矩阵是5行3列
        self.wolves = np.zeros((wolves, 3))
        self.fitness = np.full(wolves, 1e6)

        # 随机初始化位置
        for i in range(wolves):
            for j in range(3):
                self.wolves[i, j] = np.random.uniform(*self.bounds[j])

        # 记录最佳方案，第一行是最优参数
        self.alpha = self.wolves[0].copy()
        self.alpha_fitness = 1e6
        self.fitness_history = []

    def evaluate(self, params):
        """评估PID参数的性能"""
        Kp, Ki, Kd = params

        # 仿真时间与步长
        sim_time = 30000
        points = 300  # 减少点数量加快速度，每100s为一个时间步
        time = np.linspace(0, sim_time, points)
        dt = time[1] - time[0] # dt这个时候为100

        # 初始化
        temp = np.ones_like(time) * self.room_temp
```

```
        pid = PIDController(Kp, Ki, Kd, self.setpoint)

        # 跟踪最大超调量
        max_overshoot = 0

        for i in range(1, len(time)):
            # 延迟处理简化
            delay_idx = max(0, i - int(self.theta / dt)) if self.theta / dt > 0 else i
            current_temp = temp[delay_idx]

            u = pid.update(current_temp, dt)
            u = np.clip(u, 0, 10)  # 电压限制

            # 系统响应计算 - 简化和稳定化
            previous_temp = temp[i - 1] - self.room_temp
            new_temp_value = previous_temp + (self.K * u - previous_temp) * dt / self.tau

            # 检查超调
            current_temp_val = new_temp_value + self.room_temp
            if current_temp_val > self.setpoint:
                overshoot = current_temp_val - self.setpoint
                if overshoot > max_overshoot:
                    max_overshoot = overshoot

            # 更新温度
            temp[i] = np.clip(new_temp_value + self.room_temp, 0, 100)

        # 性能指标 - 增加对超调的惩罚权重
        error = np.abs(temp - self.setpoint)
        itae = np.sum(error * time)

        # 大幅增加超调惩罚权重，加快收敛
        if max_overshoot > 0.1:
            itae += max_overshoot * 2000
        elif max_overshoot > 0.05:
            itae += max_overshoot * 1000

        return itae

    def optimize(self):
        print("开始灰狼优化...")

        # 计算初始适应度
        best_score = float('inf')
        for i in range(self.n_wolves):
            self.fitness[i] = self.evaluate(self.wolves[i])
            if self.fitness[i] < best_score:
                best_score = self.fitness[i]
                self.alpha = self.wolves[i].copy()
                self.alpha_fitness = best_score

        print(f"初始最佳适应度: {self.alpha_fitness:.2f}")
        self.fitness_history.append(self.alpha_fitness)
```

```
        # 计算必要参数 计算狼群移动距离
        for it in range(self.max_iter):
            a = 2 - it * (2 / self.max_iter)  # 线性递减系数

            for i in range(self.n_wolves):
                r1, r2 = np.random.rand(3), np.random.rand(3)
                A = 2 * a * r1 - a
                C = 2 * r2

                # 向Alpha移动
                for j in range(3):
                    D = np.abs(C[j] * self.alpha[j] - self.wolves[i, j])
                    self.wolves[i, j] = np.clip(self.alpha[j] - A[j] * D,
                                                self.bounds[j][0],
                                                self.bounds[j][1])

                # 评估新位置
                new_fitness = self.evaluate(self.wolves[i])

                # 更新Alpha
                if new_fitness < self.alpha_fitness:
                    self.alpha = self.wolves[i].copy()
                    self.alpha_fitness = new_fitness
                    print(f"迭代 {it + 1}: 新Alpha, ITAE={new_fitness:.2f}")

            # 记录最佳适应度
            self.fitness_history.append(self.alpha_fitness)

        print("优化完成!")
        return self.alpha
```