浙江大學
ZHEJIANG UNIVERSITY

智能控制技术
实验报告

实验名称	永磁同步电机调速
实验地点	教 7 304
姓　名	PhilFan
学　号	19260817
实验日期	January 10, 2025
指导老师	刘山 刘禹骏

Contents

1 背景介绍

电机是新能源汽车的三大核心零部件之一，被誉为新能源汽车的“心脏”。而由于电动汽车的空间限制和使用环境的要求，传统的电力电子与电机技术已经难以满足其高性能、小尺寸和严格的环境温度要求。因此永磁电机作为电动汽车电机驱动系统的关键技术之一，受到广泛关注。

近年来，政府出台了多项政策支持国内纯电动汽车及插电式混合动力汽车在内的新能源汽车的发展。如《新能源汽车产业发展规划（2021-2035)》中，十分明确提出 2025 年我国新能源汽车新车销售量达到汽车新车销售总量的 20% 左右。规划的出台极大地鼓舞了自主品牌汽车新能源产业的上下游，产业呈现爆发式增长势头。

永磁同步电机 (PMSM) 是一种在转子上采用永磁体作为励磁源的同步电机。相比传统同步电机，PMSM 具有结构简单、效率高、功率密度大、调速性能好等优点，在伺服驱动、电动汽车等领域有广泛应用 [?]。

在秋冬学期，我学习了另一门《运动控制》的课程。课上老师提到了有关电机调速的问题我比较感兴趣，所以本次大作业我选择了永磁同步电机调速作为研究对象。（另外在运动控制的大作业中，我做的是 PID 倒立摆平衡小车，把智能控制中学到的专家控制和模糊控制算法做了应用，感觉还是很有意思的）

1.1 永磁同步电机数学模型

永磁同步电机的数学模型可以通过以下方程描述 [?]：

d 轴电流方程

$$\frac{dI_d}{dt} = \frac{1}{L_d}(V_d - R_s I_d + \omega L_q I_q) \tag{1}$$

其中：

- I_d 是 d 轴电流
- V_d 是 d 轴电压
- R_s 是定子电阻
- ω 是角速度
- L_q 和 L_d 分别是 q 轴和 d 轴的电感
- I_q 是 q 轴电流

q 轴电流方程

$$\frac{dI_q}{dt} = \frac{1}{L_q}(V_q - R_s I_q - \omega L_d I_d - \omega \psi_f) \tag{2}$$

其中：

- I_q 是 q 轴电流
- V_q 是 q 轴电压
- ψ_f 是永磁体磁链

转速方程

$$\frac{d\omega}{dt} = \frac{1}{J}(T_e - T_L - B\omega) \tag{3}$$

其中：

- ω 是角速度
- J 是转动惯量
- T_e 是电磁转矩
- T_L 是负载转矩
- B 是阻尼系数

电磁转矩方程

$$T_e = \frac{3}{2} p(\psi_f I_q + (L_d - L_q) I_d I_q) \tag{4}$$

其中：

- T_e 是电磁转矩
- p 是极对数
- ψ_f 是永磁体磁链
- L_d 和 L_q 分别是 d 轴和 q 轴的电感

2 问题分析

根据永磁同步电机的数学模型 [?]，我们可以将系统的输入、输出和状态变量进行分析。系统的输入变量包括 d 轴电压 V_d、q 轴电压 V_q 以及负载转矩 T_L。其中 V_d 和 V_q 是控制器的输出量，可以通过控制器进行调节；而 T_L 是外部负载带来的扰动，在实际

应用中往往难以准确测量。系统的输出变量是电机转速 ω，这也是我们最终需要控制的目标量。状态变量包括 d 轴电流 I_d、q 轴电流 I_q 和电机转速 ω，这些变量完整地描述了系统在任意时刻的状态。通过建立这些变量之间的关系，我们可以设计相应的控制策略来实现对电机转速的精确控制。

- **输入变量:**

  (1). V_d: d 轴电压（控制器输出）

  (2). V_q: q 轴电压（控制器输出）

  (3). T_L: 负载转矩（扰动）

- **输出变量:**

  (1). ω: 电机转速（rad/s）

- **状态变量:**

  (1). I_d: d 轴电流

  (2). I_q: q 轴电流

  (3). ω: 电机转速

S-function 建立

在本实验中，我使用 S-function 来建立永磁同步电机的 simulink 模型。

- **输入输出定义:**
  - 输入 (u): 包括 d 轴电压 V_d、q 轴电压 V_q 和负载转矩 T_L
  - 输出 (y): 电机转速 ω
  - 状态变量 (x): d 轴电流 I_d、q 轴电流 I_q 和转速 ω
- **初始化函数 (mdlInitializeSizes):** 定义系统的基本参数，包括连续状态数量 (3 个)、输入数量 (3 个) 和输出数量 (1 个)
- **状态方程 (mdlDerivatives):** 实现电机的数学模型，包括 d 轴电流方程、q 轴电流方程和转速方程

在使用时需要注意以下几点:

- 确保输入参数的单位统一性
- 采样时间设为连续 (ts = [0 0]) 以保证仿真精度

Listing 1: S-function 建立

```
function [sys, x0, str, ts, simStateCompliance] = motor(t, x, u, flag, R_s, L_d, L_q, psi_f, p, J, B)
    switch flag
        case 0
            [sys, x0, str, ts, simStateCompliance] = mdlInitializeSizes();
        case 1
            sys = mdlDerivatives(t, x, u, R_s, L_d, L_q, psi_f, p, J, B);
        case 2
            sys = mdlUpdate(t, x, u);
        case 3
            sys = mdlOutputs(t, x, u);
        case 9
            sys = mdlTerminate(t, x, u);
        otherwise
            DAStudio.error('Simulink:blocks:unhandledFlag', num2str(flag));
    end
end

% 初始化函数
function [sys, x0, str, ts, simStateCompliance] = mdlInitializeSizes()
    % 定义状态、输入、输出的大小
    sizes = simsizes;

    sizes.NumContStates = 3; % 连续状态: I_d, I_q, omega
    sizes.NumDiscStates = 0; % 无离散状态
    sizes.NumOutputs    = 1; % 输出: omega
    sizes.NumInputs     = 3; % 输入: V_d, V_q, T_L
    sizes.DirFeedthrough = 0; % 无直接传递
    sizes.NumSampleTimes = 1; % 一个采样时间

    sys = simsizes(sizes);

    x0 = [0; 0; 0]; % 初始状态: I_d, I_q, omega
    str = [];
    ts = [0 0]; % 连续时间系统
    simStateCompliance = 'UnknownSimState'; % 状态兼容性
```

```
36  end
37
38  % 更新函数（此处为空，因为是连续系统）
39  function sys = mdlUpdate(~, ~, ~)
40      sys = [];
41  end
42
43  % 输出函数
44  function sys = mdlOutputs(~, x, ~)
45      sys = [x(3)];
46  end
47
48  % 终止函数
49  function sys = mdlTerminate(~, ~, ~)
50      % 终止时执行清理操作（如果需要）
51      sys = [];
52  end
53
54  % 动态方程求解
55  function sys = mdlDerivatives(~, x, u, R_s, L_d, L_q, psi_f, p, J, B)
56      % 参数定义
57      % R_s = 0.01;   % 定子电阻
58      % L_d = 0.001;  % d 轴电感
59      % L_q = 0.0015; % q 轴电感
60      % psi_f = 0.1;  % 永磁体磁链
61      % p = 4;        % 极对数
62      % J = 0.01;     % 转动惯量
63      % B = 0.001;    % 阻尼系数
64
65      % 输入提取
66      V_d = u(1); % d 轴电压
67      V_q = u(2); % q 轴电压
68      T_L = u(3); % 负载转矩
69
70      % 状态提取
71      I_d = x(1); % d 轴电流
72      I_q = x(2); % q 轴电流
73      omega = x(3); % 转速
```

```
74
75      % 电磁转矩计算
76      T_e = (3/2) * p * (psi_f * I_q + (L_d - L_q) * I_d * I_q);
77
78      % d 轴电流微分方程
79      dId_dt = (1 / L_d) * (V_d - R_s * I_d + omega * L_q * I_q);
80
81      % q 轴电流微分方程
82      dIq_dt = (1 / L_q) * (V_q - R_s * I_q - omega * L_d * I_d - omega *
            psi_f);
83
84      % 转速微分方程
85      domega_dt = (1 / J) * (T_e - T_L - B * omega);
86
87      % 返回微分方程结果
88      sys = [dId_dt; dIq_dt; domega_dt];
89  end
```

建立好的电机模型在 simulink 中如下图所示

Figure 1: 建立的电机模型

这里我使用了 S-function 的模型封装 MASK 功能，将参数设置为常数使用输入框给定，这样就可以让我便于调整参数，进行修改。

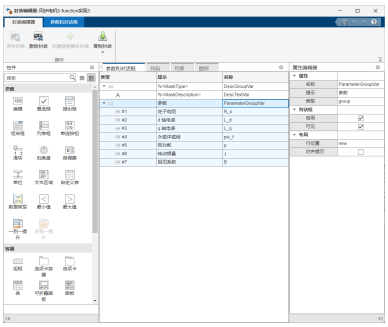

Figure 2: S-function 模型封装

Figure 3: S-function 模型封装 MASK

3 PID 建模控制

3.1 PID 控制器设计

PID 控制器是一种经典的反馈控制器，由比例 (P)、积分 (I) 和微分 (D) 三部分组成。对于永磁同步电机的速度控制，PID 控制器的输出可以表示为:

$$u(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{de(t)}{dt} \tag{5}$$

其中:

- $u(t)$ 是控制器输出
- $e(t)$ 是误差信号, 即参考速度与实际速度之差
- K_p 是比例增益
- K_i 是积分增益
- K_d 是微分增益

使用 simulink 中的 PID 模块，我们把电机模型和 PID 控制器连接起来，如下图所示

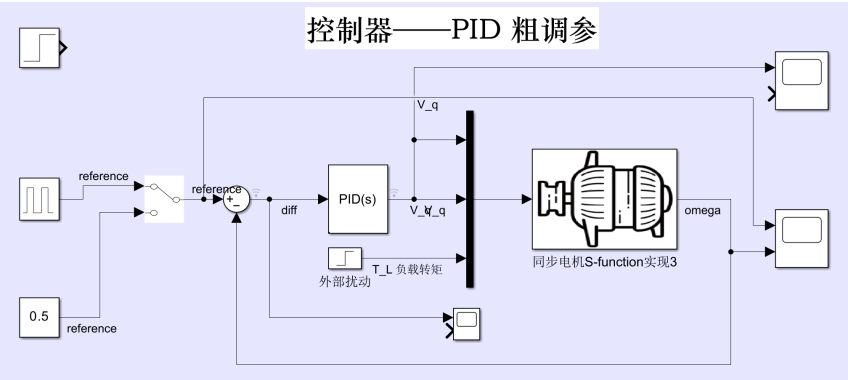

Figure 4: PID 控制模型

3.2 PID 调参

在 PID 控制器的调参过程中，我采用了以下步骤:

(1). 首先调整比例参数 K_p

- 将 K_i 和 K_d 设为 0，仅使用 P 控制

- 从小到大逐步增加 K_p，直到系统响应速度合适
- 此时系统会存在稳态误差，但响应较快

(2). 调整积分参数 K_i

- 在合适的 K_p 基础上，逐步增加 K_i
- K_i 的作用是消除稳态误差
- K_i 太大会导致超调和震荡，太小则收敛太慢
- 通过反复测试找到合适的 K_i 值，使系统既能消除稳态误差，又不会产生过大的超调

(3). 最后微调微分参数 K_d

- K_d 的作用是抑制超调和震荡
- K_d 值不宜过大，否则会引入高频噪声，使系统不稳定

通过以上调参过程，最终得到了一组 PID 参数，系统能够快速跟踪阶跃和方波信号。

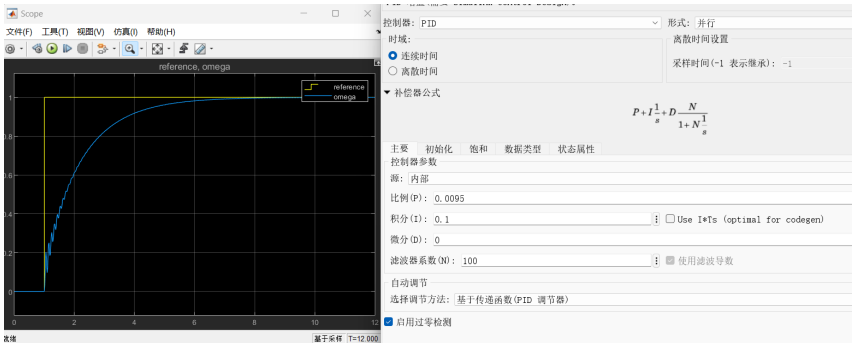

Figure 5: 跟踪阶跃信号

图6是 PID 调参结果，从左到右依次是响应与给定信号，误差，输出电压

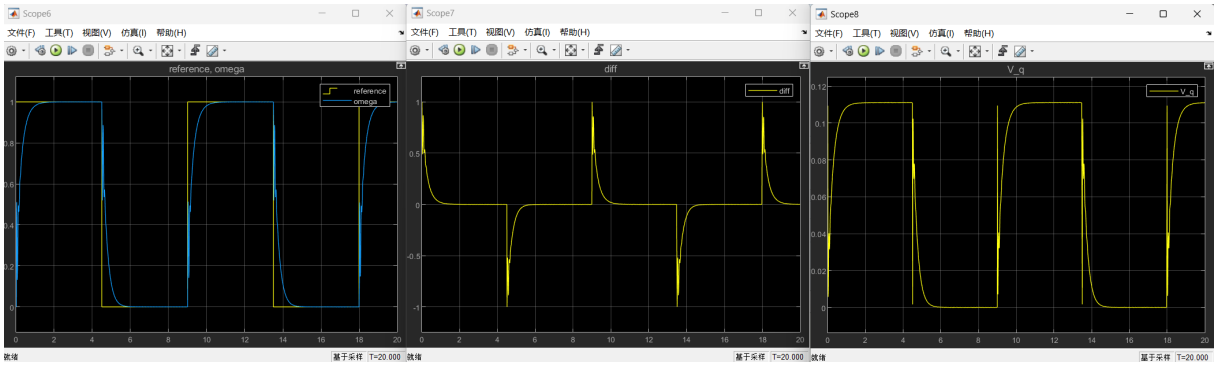

Figure 6: PID 调参结果，跟踪方波信号

Figure 7: PID 参数

3.3 Control System Designer 进行自动调参

手动调参基于试错法，需要反复调整参数，耗时耗力。

使用 Control System Designer 进行自动调参，可以大大提高调参效率。

这里我学习了 Control System Designer 的使用方法，并使用它对 PID 参数进行了自动调参。

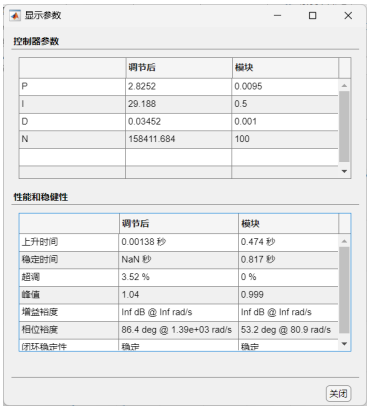

Figure 8: Control System Designer 自动调参

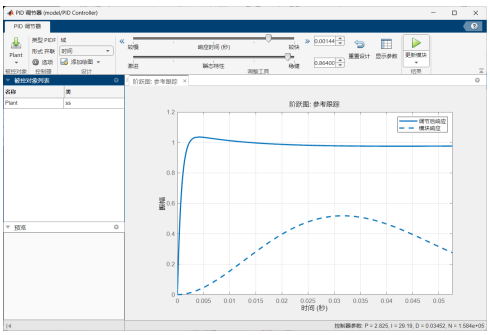

Figure 9: Control System Designer 自动调参结果

我发现自动调参中，使用了一些 PID 控制器的改进方法，比如积分限幅，使用低通滤波器，这些方法可以有效抑制超调和震荡，提高系统的稳定性。

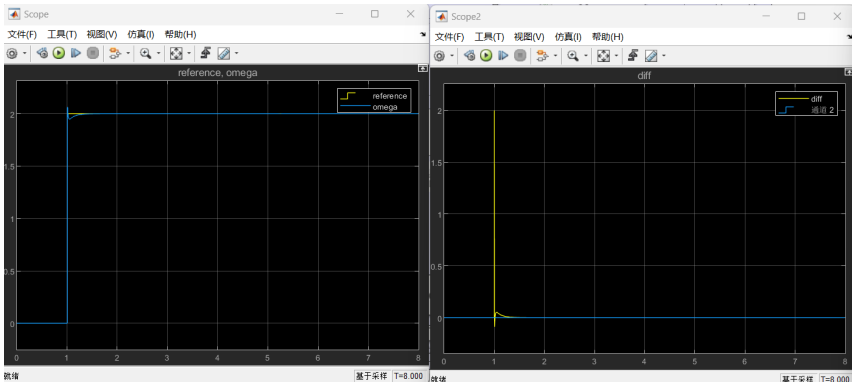

Figure 10: 更改后的波形

但是也需要注意的是，自动调参虽然可以调节出一个比较好的曲线，但是有时候自动调参调节出的参数的输出值会超出输入电压的范围，导致系统无法正常工作。也就是没有办法物理实现，所以在自动调参的时候，我们需要对输出值进行限幅，确保输出值在输入电压的范围之内。

4 MPC 模型预测控制

4.1 MPC 控制器简介

在秋学期的《空中机器人》课程当中，高飞老师也简要介绍了一下 MPC 控制器，于是，在本次大作业的过程当中，借助 matlab 的 Control System Designer 工具，我尝试设计了一个 MPC 控制器，并进行了仿真。

模型预测控制 (Model Predictive Control, MPC) 是一种先进的控制策略，它通过在线优化来预测和控制系统的未来行为。MPC 的核心思想是利用系统模型在有限时域内预测系统的输出响应，并通过求解优化问题来获得最优控制序列。

MPC 控制器的主要特点包括：

- **固定预测时域**: 在每个采样时刻，MPC 都会基于当前状态，预测未来固定长度 N 个时间步内的系统行为。这个 N 称为预测时域 (Prediction Horizon)。
- **滚动优化**: 虽然 MPC 会计算未来 N 个时间步的控制序列，但只执行第一个控制量，然后在下一个采样时刻重新进行优化计算。这种方式也被称为” 开环最优反馈” 或” 反应式规划”。
- **在线优化**: MPC 通过求解一个实时优化问题来获得控制序列，优化目标通常包括跟踪误差最小化，控制输入的平滑性，能量消耗

MPC 控制器具有多方面的优势：它能够显式考虑未来时域内的系统行为，可以有效处理多变量耦合系统，能够自然地处理系统约束，同时还可以处理非线性系统和时变系统。这些特点使得 MPC 控制器在实际工程应用中具有很强的适用性和优越性。

4.2 MPC 控制器设计

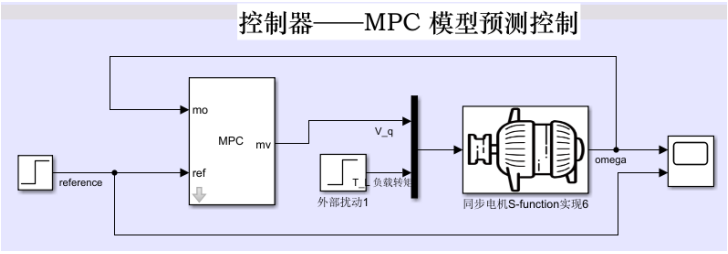

Figure 11: Simulink 模型

(1). 安装 MPC Designer 工具箱

(2). 在 Simulink 中建立系统模型

(3). 打开设计界面, 点击 MPC Structure, 设置输入输出通道, 打开 I/O Attributes, 设置输入输出名称

(4). 点击 Update and Simulate 进行仿真

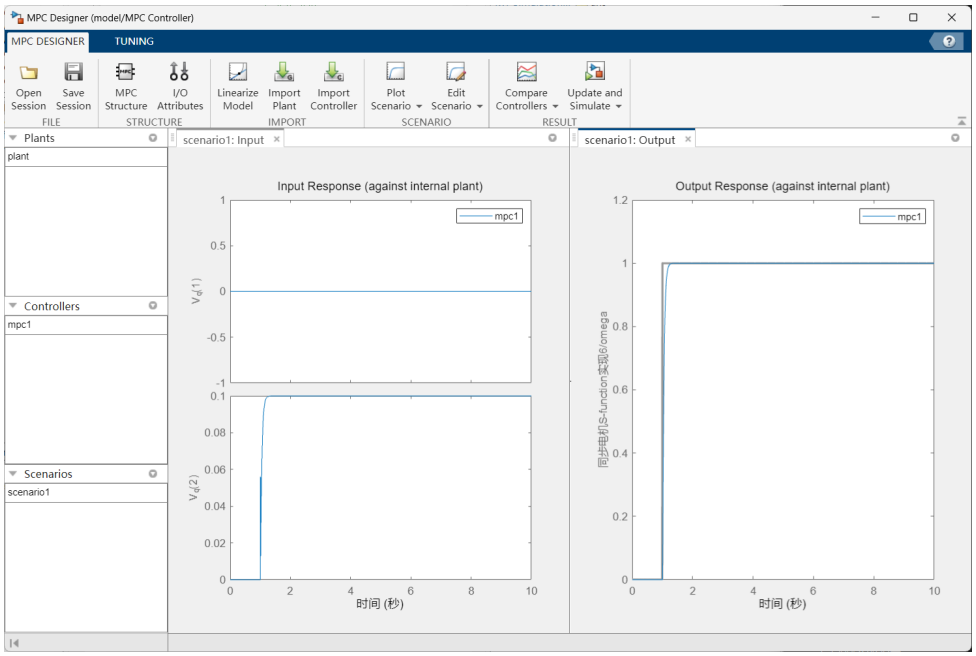

Figure 12: MPC Designer 设置界面

在这一部分我主要参考了以下资料:

- Matlab 官方 MPC Toolbox 文档[1]
- CSDN 博客《模型预测控制-Matlab 自带 MPC Designer 工具》[2]

Listing 2: MPC 控制器设计

```
load('MPC_plant.mat'); % 加载模型文件
load('MPC_RefSignal.mat');
load('MPC_MDSignal.mat');
%% create MPC controller object with sample time
mpc1 = mpc(plant_C, 0.01);
%% specify prediction horizon
mpc1.PredictionHorizon = 10;
%% specify control horizon
mpc1.ControlHorizon = 2;
%% specify nominal values for inputs and outputs
```

[1] https://ww2.mathworks.cn/help/mpc/gs/introduction.html
[2] https://blog.csdn.net/weixin_43470383/article/details/134227287

```
mpc1.Model.Nominal.U = [0;0];
mpc1.Model.Nominal.Y = 0;
%% specify weights
mpc1.Weights.MV = [0 0];
mpc1.Weights.MVRate = [0.1 0.1];
mpc1.Weights.OV = 1;
mpc1.Weights.ECR = 100000;
%% specify simulation options
options = mpcsimopt();
options.RefLookAhead = 'off';
options.MDLookAhead = 'off';
options.Constraints = 'on';
options.OpenLoop = 'off';
%% run simulation
sim(mpc1, 1001, mpc1_RefSignal, mpc1_MDSignal, options);
```

导入到 simulink 中，发现其控制效果还是很好的。

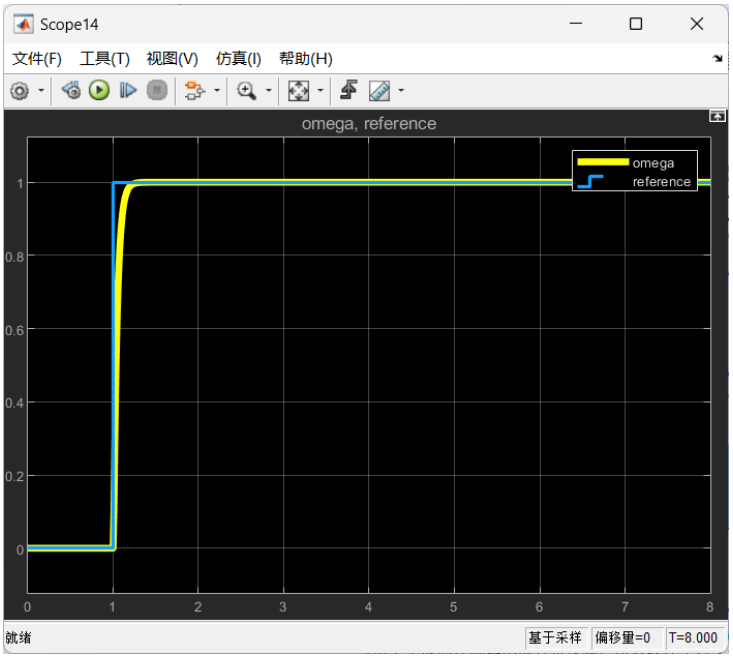

Figure 13: MPC 控制器仿真结果

5 模糊控制

模糊控制的定义

- 是将模糊数学理论应用于自动控制领域的控制方法
- 基于模糊集合理论、模糊语言变量及模糊逻辑推理

工作原理

- 通过观察过程输出精确量转化为模糊量
- 经过人脑思维与逻辑推理进行模糊判决
- 将判决结果的模糊量转化为精确量

控制特性

- 属于非线性控制
- 不依赖于控制对象的精确模型
- 仅依靠少量控制规则
- 具有较强的鲁棒性
- 适用于数学模型未知、复杂的非线性系统控制

5.1 模糊控制器设计

根据系统逻辑，将系统 slx 模型搭建如下

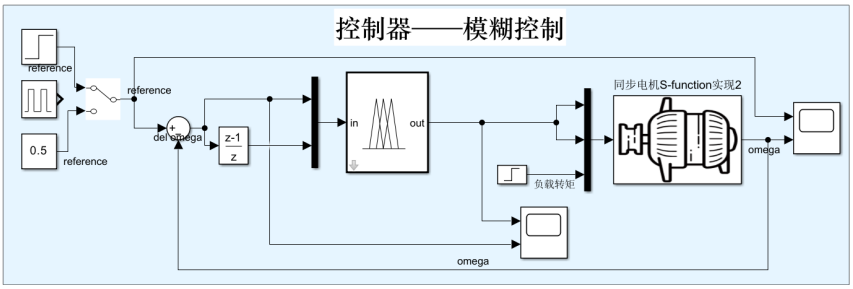

Figure 14: 模糊控制器 Simulink 结构

使用代码建立规则和隶属度函数。

x,v 和 U 我都设置的是三角函数

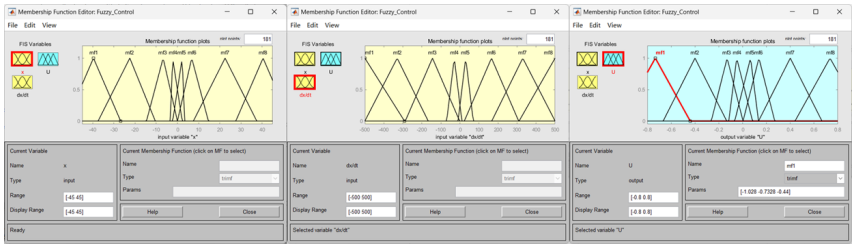

Figure 15: 输入输出论域的范围

```
clear;
close all;

% 创建模糊推理系统，使用 newfis
fuzzyController = mamfis( ...
    'NumInputs',1,'NumInputMFs',2,...
    'NumOutputs',1,'NumOutputMFS',2,...
    'AddRule','none');

% 定义输入变量x（位置误差）及其隶属度函数，范围 [-0.04, 0.04]
fuzzyController.Inputs(1).Name = 'x';
fuzzyController.Inputs(1).Range = [-0.04 0.04];

fuzzyController = addMF(fuzzyController, 'input', 1, 'EN', 'trimf',
    [-0.05 -0.04 -0.03]); % EN: Extremely Negative
fuzzyController = addMF(fuzzyController, 'input', 1, 'VN', 'trimf',
    [-0.04 -0.03 -0.02]); % VN: Very Negative
fuzzyController = addMF(fuzzyController, 'input', 1, 'N', 'trimf', [-0.03
     -0.02 -0.01]); % N: Negative
fuzzyController = addMF(fuzzyController, 'input', 1, 'NZ', 'trimf',
    [-0.02 -0.01 0]); % Z: Zero
fuzzyController = addMF(fuzzyController, 'input', 1, 'PZ', 'trimf', [0
    0.01 0.02]); % Z: Zero
fuzzyController = addMF(fuzzyController, 'input', 1, 'P', 'trimf', [0.01
    0.02 0.03]); % P: Positive
fuzzyController = addMF(fuzzyController, 'input', 1, 'VP', 'trimf', [0.02
     0.03 0.04]); % VP: Very Positive
fuzzyController = addMF(fuzzyController, 'input', 1, 'EP', 'trimf', [0.03
     0.04 0.05]); % EP: Extremely Positive

```

```
% 可视化x的隶属度函数
figure;
subplot(311);
plotmf(fuzzyController, 'input', 1);
title('位置误差 (x) 的隶属度函数');

% 定义输入变量dx/dt (位置误差变化率) 及其隶属度函数, 范围 [-0.5, 0.5]
fuzzyController.Inputs(2).Name = 'dx';
fuzzyController.Inputs(2).Range = [-0.5 0.5];
fuzzyController = addMF(fuzzyController, 'input', 2, 'EN', 'trimf', [-0.6 -0.5 -0.4]); % EN: Extremely Negative
fuzzyController = addMF(fuzzyController, 'input', 2, 'VN', 'trimf', [-0.5 -0.4 -0.3]); % VN: Very Negative
fuzzyController = addMF(fuzzyController, 'input', 2, 'N', 'trimf', [-0.4 -0.3 -0.2]); % N: Negative
fuzzyController = addMF(fuzzyController, 'input', 2, 'NZ', 'trimf', [-0.2 -0.1 0]); % Z: Zero
fuzzyController = addMF(fuzzyController, 'input', 2, 'PZ', 'trimf', [0 0.1 0.2]); % Z: Zero
fuzzyController = addMF(fuzzyController, 'input', 2, 'P', 'trimf', [0.2 0.3 0.4]); % P: Positive
fuzzyController = addMF(fuzzyController, 'input', 2, 'VP', 'trimf', [0.3 0.4 0.5]); % VP: Very Positive
fuzzyController = addMF(fuzzyController, 'input', 2, 'EP', 'trimf', [0.4 0.5 0.6]); % EP: Extremely Positive

% 可视化dx/dt的隶属度函数
subplot(312);
plotmf(fuzzyController, 'input', 2);
title('位置误差变化率 (dx/dt) 的隶属度函数');

% 定义输出变量U (控制电压) 的隶属度函数, 范围 [-10, 10]
fuzzyController = addvar(fuzzyController, 'output', 'U', [-10 10]);
fuzzyController = addMF(fuzzyController, 'output', 1, 'EL', 'trimf', [-10 -10 -7]); % EL: Extremely Low
fuzzyController = addMF(fuzzyController, 'output', 1, 'VL', 'trimf', [-10 -7 -4]); % VL: Very Low
fuzzyController = addMF(fuzzyController, 'output', 1, 'L', 'trimf', [-7
```

```
    -4 -1]); % L: Low
51  fuzzyController = addMF(fuzzyController, 'output', 1, 'NZ', 'trimf', [-3
    -1 1]); % NZ:negetive ZERO
52  fuzzyController = addMF(fuzzyController, 'output', 1, 'PZ', 'trimf', [-1
    1 3]); % M: positive ZERO
53  fuzzyController = addMF(fuzzyController, 'output', 1, 'H', 'trimf', [1 4
    7]); % H: High
54  fuzzyController = addMF(fuzzyController, 'output', 1, 'VH', 'trimf', [4 7
     10]); % VH: Very High
55  fuzzyController = addMF(fuzzyController, 'output', 1, 'EH', 'trimf', [7
    10 10]); % EH: Extremely High
```

但是值得注意的是，根据调试信息可以看出，这种方法已经逐步被淘汰了，在接下来的版本当中将全被替换成为 ui 界面操作

5.1.1 模糊规则的设计

其实使用代码生成模糊矩阵比手动一个个点还是要快一些的，这也是我为什么选择这种方法的原因。

这里我使用了另一个课本中给出的一个参考 8x8 矩阵作为调参的 backbone，之后我在这个的基础上进行了对于我这个系统的调参和修改。

Listing 3: 生成规则矩阵

```
1   fuzzyController = readfis('Fuzzy_Control.fis');
2
3   table = [
4       [1, 1, 1, 1, 6, 5, 5, 5],
5       [1, 2, 2, 1, 7, 5, 5, 5],
6       [2, 2, 2, 1, 2, 6 ,6, 6],
7       [2, 3, 3, 2, 7, 6, 6, 6],
8       [3, 3, 3, 2, 8, 7, 6, 7],
9       [3, 3, 4, 2,8, 7, 7, 7],
10      [3, 4, 4, 2, 8, 8, 7, 8],
11      [4, 4, 4, 4, 8, 8, 8, 8]];
12  % 生成规则矩阵
13  rules = [];
14  for i = 1:8
15      for j = 1:8
16          disp(table(i,j));
```

```
        output = table(i,j);
        rules = [rules; i j output 1 1]; % 1 1表示使用 'min' 合成和 '
            centroid' 解模糊
    end
end

% disp(rules)

% 添加规则到模糊系统
fuzzyController = addrule(fuzzyController, rules);
%showrule(fuzzyController,'Format','symbolic');
```

Listing 4: 可视化规则

```
% 可视化规则
ruleview(fuzzyController);

%figure;
%plotfis(fuzzyController);

% 输出 surface
figure;
gensurf(fuzzyController,[1,2],1);
saveas(gcf, 'surf.jpg');

% 保存为.fis文件
writefis(fuzzyController, 'Fuzzy_Control.fis');
```

建立了初步的规则，但在系统上进行尝试的时候发现效果并不好，经常会出现不收敛、不稳定、震荡大等问题。

所以经过反复多次的修改和尝试，最后得到了一版规则，可以达到给定的效果。

5.1.2 模糊控制器调参效果

从上面的仿真结果可以看出：

(1). 系统响应性能良好。从图中可以看出，系统能够快速跟踪给定的参考输入信号，位置误差在短时间内就能收敛到参考位置附近。

(2). 稳态精度合理。在稳态时，位置 x 能够很好地跟随参考输入，稳态误差比较小，满足控制要求。

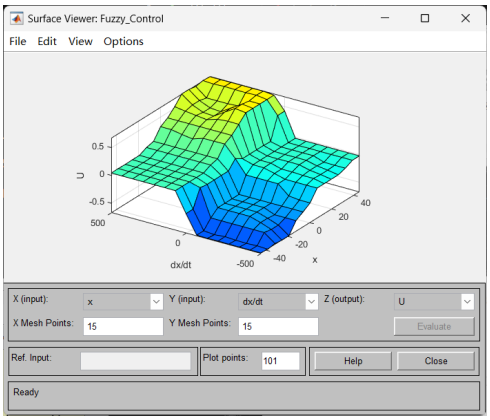

Figure 16: 模糊控制 surface 的形状

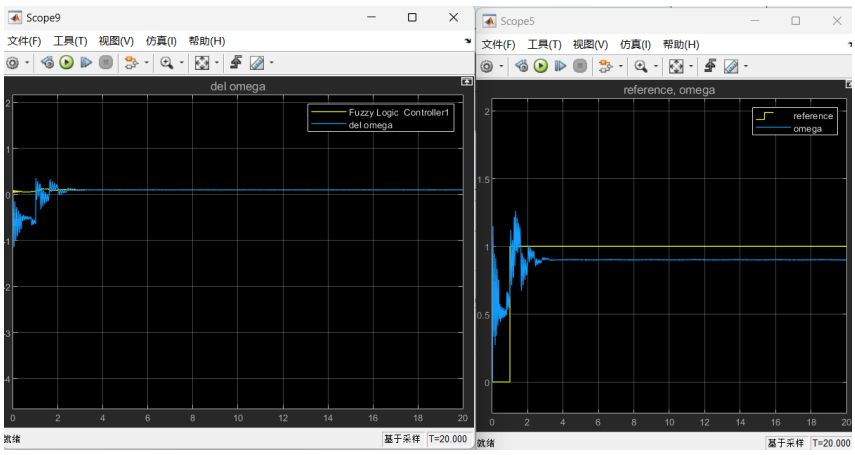

Figure 17: 第一次调参后，不能稳定在参考输入上

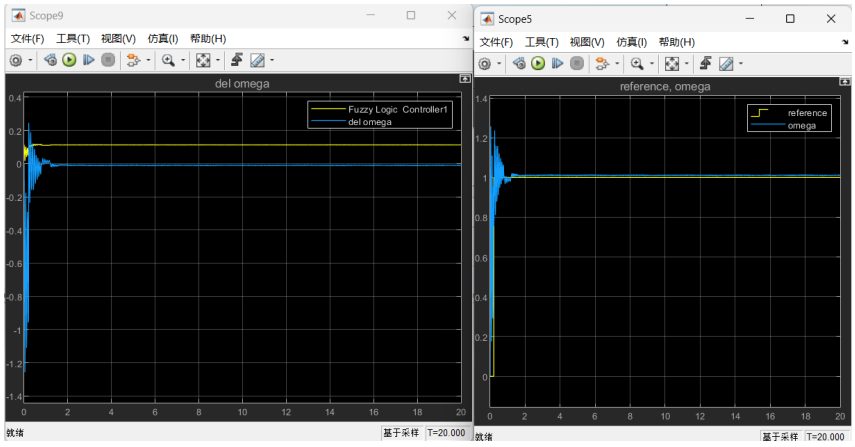

Figure 18: 第二次调参后，稳定在阶跃输入上

5.2 模糊 PID 控制

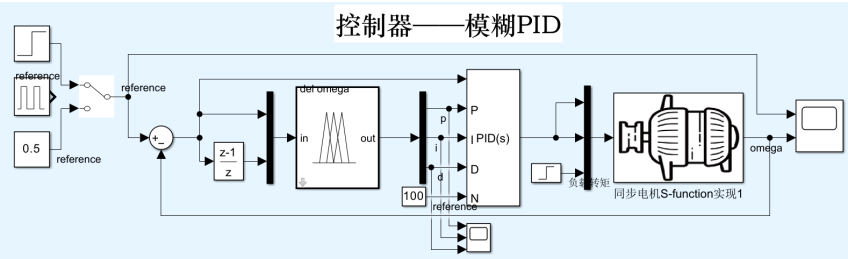

Figure 19: 模糊 PID Simulink 结构

模糊 PID 控制器是将模糊控制和 PID 控制相结合的一种智能控制方法。它利用模糊控制的专家经验来在线调整 PID 控制器的参数, 从而提高系统的控制性能。

在永磁同步电机的速度控制中, 模糊 PID 控制器以速度误差 e 和误差变化率 ec 作为输入, 通过模糊推理得到 PID 参数的调整量 ΔK_p、ΔK_i 和 ΔK_d。具体工作过程如下:

(1). 首先将速度误差 e 和误差变化率 ec 进行模糊化, 即将精确值映射到相应的模糊集合中。

(2). 根据专家经验建立模糊规则库, 描述 e 和 ec 与 PID 参数调整量之间的关系。

(3). 采用模糊推理得到 ΔK_p、ΔK_i 和 ΔK_d 的模糊值。

(4). 对模糊推理结果进行解模糊化, 得到 PID 参数的具体调整量。

所以，根据以下的论域范围和隶属度函数，我设计了一个模糊控制的 PID

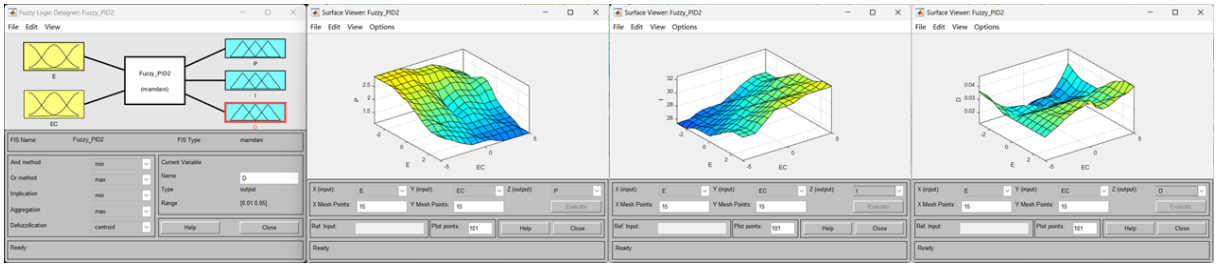

Figure 20: 模糊 PID 论域范围与决策平面

这个初版的结果并不理想，所以我对论域范围和规则进行了调整，得到了以下结果

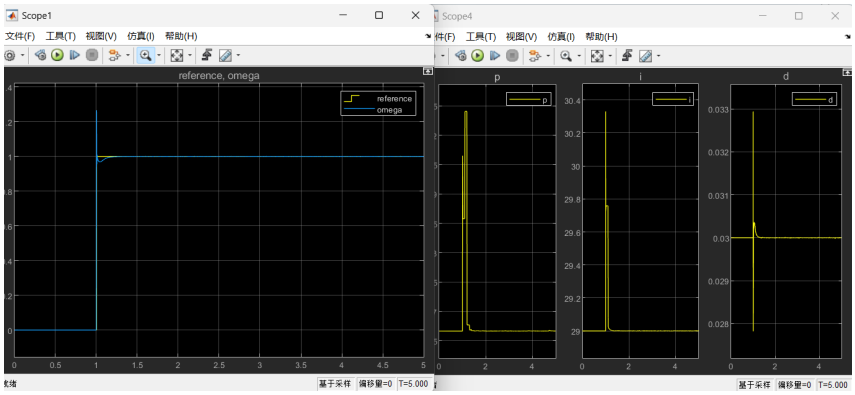

Figure 21: 模糊 PID 控制器

6 神经网络建模控制

6.1 BP-PID 控制

第二种方法是基于 BP 神经网络的 PID 控制经典的增量式数字 PID 控制算法的公式为：

$$u(k) = u(k-1) + K_p[e(k) - e(k-1)] + K_i e(k) + K_d[e(k) - 2e(k-1) + e(k-2)] \quad (6)$$

其中：

- $u(k)$ 是当前采样时刻的控制量
- $e(k)$ 是当前采样时刻的期望输出与实际输出之差
- K_p、K_i、K_d 分别是比例、积分、微分系数

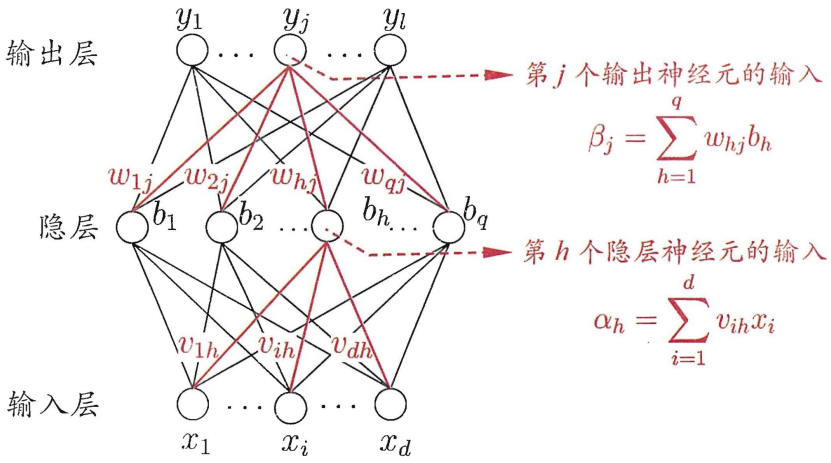

Figure 22: BP-PID 控制器结构

将 K_p、K_i、K_d 视为依赖于系统运行状态的可调参数时，可以将上述描述表示为：

$$u(k) = f[u(k-1), K_p, K_i, K_d, e(k), e(k-1), e(k-2)] \quad (7)$$

其中，$f(\cdot)$ 是一个与 K_p、K_i、K_d、$u(k-1)$、$e(k)$ 等有关的非线性函数。这个函数可以通过训练和学习来找到最佳的控制规律，通常使用 BP 神经网络（反向传播神经网络）来实现。

包括输入层、隐含层和输出层。输出层的三个输出分别对应 PID 控制器的三个可调参数 K_p、K_i 和 K_d。由于这些参数不能为负，所以输出层神经元的变换函数选择非负的 Sigmoid 函数。

输入： 训练集 $D = \{(\boldsymbol{x}_k, \boldsymbol{y}_k)\}_{k=1}^{m}$;
学习率 η.

过程：

在$(0,1)$范围内随机初始化网络中所有连接权和阈值
repeat
  for all $(\boldsymbol{x}_k, \boldsymbol{y}_k) \in D$ **do**
    根据当前参数和式(5.3) 计算当前样本的输出 $\hat{\boldsymbol{y}}_k$;
    根据式(5.10) 计算输出层神经元的梯度项 g_j;
    根据式(5.15) 计算隐层神经元的梯度项 e_h;
    根据式(5.11)-(5.14) 更新连接权 w_{hj}, v_{ih} 与阈值 θ_j, γ_h
  end for
until 达到停止条件

输出： 连接权与阈值确定的多层前馈神经网络

Figure 23: BP 算法流程——图源《机器学习周志华》

所以，根据这样的结构，我设计了 BP-PID 控制器，并使用 simulink 进行了仿真。

Listing 5: BPD-PID 控制器实现

```
function [sys,x0,str,ts,simStateCompliance] = bppid(t,x,u,flag,T,nh,
    xite,alfa)
% 输入参数:
% nh: 隐含层神经元数量 (hidden_neurons)
% xite: 学习率 (learning_rate)
% alfa: 动量因子 (momentumfactor)
% T: 采样时间 (sampling_time)
% 这些参数已经使用mask方法在simulink中设置
switch flag
    case 0
        [sys,x0,str,ts,simStateCompliance] = mdlInitializeSizes(T,nh);
            % 初始化函数
    case 1
        sys = mdlDerivatives(t, x, u);
    case 2
        sys = mdlUpdate(t, x, u);
    case 3
        sys = mdlOutputs(t,x,u,nh,xite,alfa); % 输出函数
    case 9
        sys = mdlTerminate(t, x, u);
    otherwise
        DAStudio.error('Simulink:blocks:unhandledFlag', num2str(flag))
            ;
```

```
    end
end

function [sys,x0,str,ts,simStateCompliance] = mdlInitializeSizes(T,nh)
    % 调用初始化函数
    % 输入参数:
    %   T  - 采样时间
    %   nh - 隐含层神经元个数
    sizes = simsizes;
    sizes.NumContStates = 0;
    sizes.NumDiscStates = 0;
    sizes.NumOutputs   = 4+6*nh; % 输出维数:控制量u,PID参数(Kp,Ki,Kd),隐含层和输出层权值
    sizes.NumInputs    = 7+12*nh; % 输入维数:[e(k),e(k-1),e(k-2),y(k),y(k-1),r(k),u(k-1)],前后两次权值
    sizes.DirFeedthrough = 1;
    sizes.NumSampleTimes = 1;
    sys = simsizes(sizes);
    x0  = [];
    str = [];
    ts  = [T 0];
    simStateCompliance = 'UnknownSimState';
end

function sys = mdlOutputs(~,~,u,nh,xite,alfa)
    % 如果权值未初始化,则随机初始化
    if any(isnan(u(8:7+12*nh)))
        hiddenWeights_prev2 = 0.5 * randn(nh,3) - 0.25; % 隐含层前两次权值(k-2)
        outputWeights_prev2 = 0.5 * randn(3,nh) - 0.25; % 输出层前两次权值(k-2)
        hiddenWeights_prev1 = 0.5 * randn(nh,3) - 0.25; % 隐含层前一次权值(k-1)
        outputWeights_prev1 = 0.5 * randn(3,nh) - 0.25; % 输出层前一次权值(k-1)
    else
        hiddenWeights_prev2 = reshape(u(8:7+3*nh),nh,3); % 隐含层前两次权值矩阵(nh×3)
```

```
52          outputWeights_prev2 = reshape(u(8+3*nh:7+6*nh),3,nh); % 输出层前两
                次权值矩阵(3×nh)
53          hiddenWeights_prev1 = reshape(u(8+6*nh:7+9*nh),nh,3); % 隐含层前一
                次权值矩阵(nh×3)
54          outputWeights_prev1 = reshape(u(8+9*nh:7+12*nh),3,nh);% 输出层前一
                次权值矩阵(3×nh)
55      end
56
57      % 神经网络输入
58      networkInput = [u(6),u(4),u(1)]; % [r(k),y(k),e(k)]
59      pidInput = [u(1)-u(2);u(1);u(1)+u(3)-2*u(2)]; % [e(k)-e(k-1);e(k);e(k
            )+e(k-2)-2*e(k-1)]
60
61      % 计算隐含层输入
62      hiddenInput = networkInput * hiddenWeights_prev1'; % 维数:1×nh
63
64      % 限制隐含层输入范围,防止溢出
65      for i = 1:length(hiddenInput)
66          hiddenInput(i) = min(max(hiddenInput(i), -500), 500);
67      end
68
69      % 隐含层输出(使用Sigmoid激活函数)
70      hiddenOutput = exp(hiddenInput)./(exp(hiddenInput)+exp(-hiddenInput))
            ;
71      % hiddenOutput = max(0, hiddenInput); % 可选用ReLU激活函数
72
73      % 输出层计算
74      outputLayerInput = outputWeights_prev1 * hiddenOutput'; % 维数:3×1
75
76      % PID参数计算(使用Sigmoid导数作为激活函数)
77      pidParams = 2./(exp(outputLayerInput)+exp(-outputLayerInput)).^2;
78      pidParams = pidParams/100; % 缩放PID参数
79
80      % 计算控制量
81      controlOutput = u(7) + pidParams' * pidInput;
82
83      % 计算梯度
84      gradientSign = sign((u(4)-u(5))/(controlOutput-u(7)+0.0000001));
```

```
85      pidParamsGradient = 2./(exp(pidParams)+exp(-pidParams)).^2;
86
87      % 下面是BP神经网络的权值更新部分
88      % 计算delta项
89      delta = u(1) * gradientSign * pidInput .* pidParamsGradient;
90
91      % 更新权值
92      outputWeights = outputWeights_prev1 + xite*delta*hiddenOutput + alfa*(
            outputWeights_prev1-outputWeights_prev2);
93
94      % 计算隐含层梯度
95      hiddenGradient = 2./(exp(hiddenOutput)+exp(-hiddenOutput)).^2;
96      % hiddenGradient = double(hiddenInput > 0); % ReLU导数
97
98      % 更新隐含层权值
99      hiddenWeights = hiddenWeights_prev1 + xite*(hiddenGradient.*(delta'*
            outputWeights))'*networkInput + alfa*(hiddenWeights_prev1-
            hiddenWeights_prev2);
100
101     % 整合输出
102     sys = [controlOutput;pidParams(:);hiddenWeights(:);outputWeights(:)];
103 end
104
105 function sys = mdlDerivatives(~,~,~)
106     sys = [];
107 end
108
109 function sys = mdlUpdate(~, ~, ~)
110     sys = [];
111 end
112
113 function sys = mdlTerminate(~, ~, ~)
114     sys = [];
115 end
```

BP-PID 控制器计算流程

BP-PID 控制器的核心思想是使用神经网络来动态调整 PID 控制器的参数。整个控制器的计算过程可以分为以下几个步骤：

神经网络结构

- 输入层接收系统状态，包括参考输入 $r(k)$、系统输出 $y(k)$ 和控制误差 $e(k)$
- 隐含层包含 nh 个神经元，使用 Sigmoid 函数作为激活函数
- 输出层有 3 个神经元，分别对应 PID 控制器的三个参数 K_p、K_i、K_d

前向计算过程

网络的前向计算分为以下步骤：

1. 隐含层计算：

$$\begin{aligned} &\text{隐含层输入：}\ I = \mathbf{x}_{\text{input}} \cdot W_i^T \\ &\text{隐含层输出：}\ O_h = \frac{\exp(I)}{\exp(I) + \exp(-I)} \end{aligned} \tag{8}$$

2. 输出层计算：

$$\begin{aligned} &\text{输出层输入：}\ O = W_o \cdot O_h^T \\ &\text{PID 参数：}\ \mathbf{K} = \frac{2}{(\exp(O) + \exp(-O))^2} \end{aligned} \tag{9}$$

3. 控制量计算：

$$u(k) = u(k-1) + K_p[e(k) - e(k-1)] + K_i e(k) + K_d[e(k) - 2e(k-1) + e(k-2)] \tag{10}$$

参数使用 BP 算法更新

1. 计算输出层的误差梯度：

$$\delta_o = \text{sign}\left(\frac{e(k) - e(k-1)}{u(k) - u(k-1)}\right) \cdot \frac{\partial O}{\partial K} \tag{11}$$

2. 更新输出层权重：

$$W_o^{new} = W_o + \eta \cdot \delta_o \cdot O_h + \alpha(W_o - W_o^{old}) \tag{12}$$

其中 η 是学习率，α 是动量因子。

3. 更新隐含层权重：

$$W_i^{new} = W_i + \eta \cdot (\delta_o \cdot W_o)^T \cdot \mathbf{x}_{\text{input}} + \alpha(W_i - W_i^{old}) \tag{13}$$

控制器特点

- 通过神经网络的学习能力，可以自适应调整 PID 参数

- 采用增量式 PID 算法，避免了积分饱和问题
- 引入动量项加快收敛速度，提高控制性能
- 使用 Sigmoid 函数保证 PID 参数始终为正值

设计了 BP-PID 控制器之后，我先在一阶传递函数的被控对象上进行了尝试，发现可以很好的控制。

根据图25 ，可以发现子经过调整之后，找到了比较合适的 PID 参数。

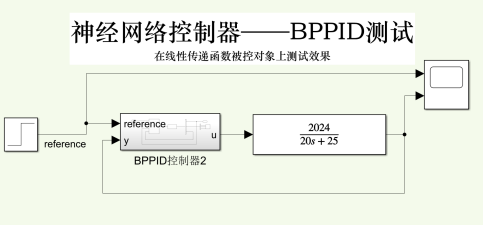

Figure 24: 一阶传递函数被控对象

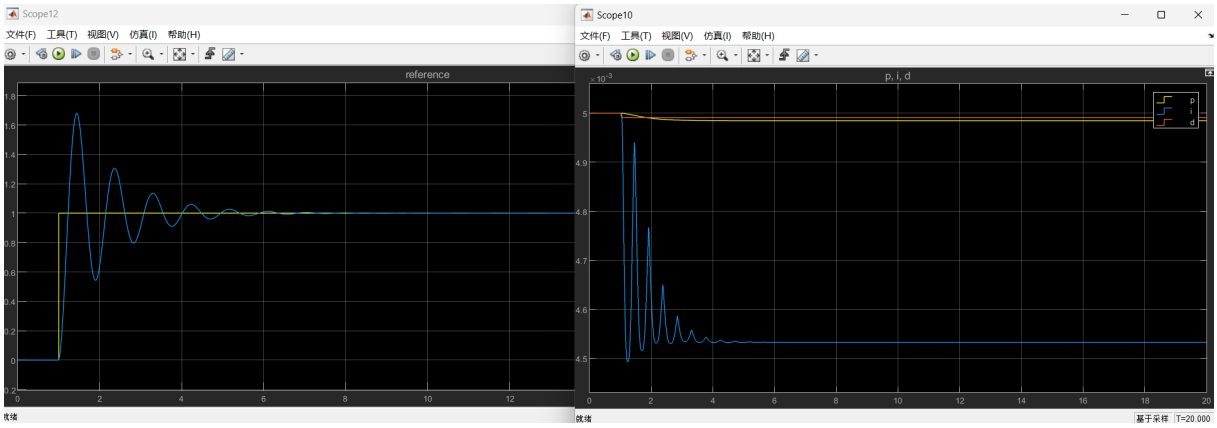

Figure 25: 一阶传递函数被控对象仿真结果

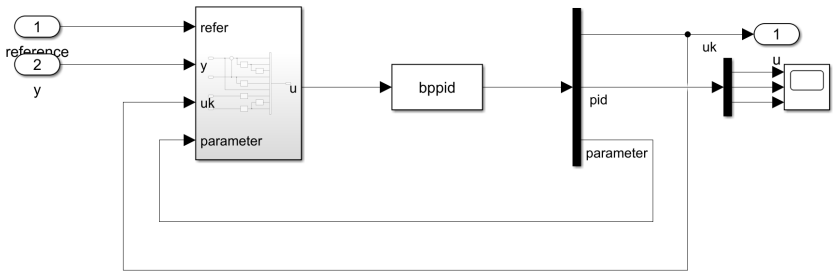

Figure 26: 控制器内部结构

在一阶传递函数仿真结束后，我尝试将 BP-PID 控制器应用到永磁同步电机的速度控制中，在经过多次调整学习率和动量因子之后，得到了以下结果。

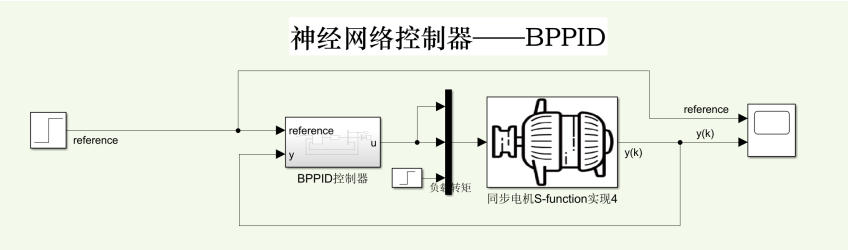

Figure 27: BP-PID 控制器

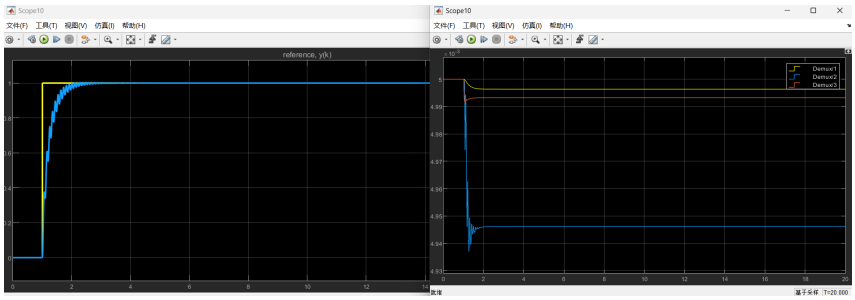

Figure 28: 永磁同步电机速度控制仿真结果

在 PID 控制器的设计过程中，要注意初始值的范围不能太离谱。因为系统不是静态的，所以初始值不能太离谱。

同时，为了加快计算速度，我尝试使用 ReLU 作为隐藏层的激活函数。ReLU 的数学表达式为:

$$f(x) = \max(0, x) = \begin{cases} x, & x > 0 \\ 0, & x \leq 0 \end{cases} \tag{14}$$

其导数为:

$$f'(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases} \tag{15}$$

ReLU 的主要优点是:

- 计算简单, 导数计算也很简单
- 能缓解梯度消失问题
- 具有稀疏性, 可以让一部分神经元的输出为 0

但在本实验中使用 ReLU 效果并不理想, 所以最终没有采用这个方案。

6.2 RBF 网络

在 BP-PID 控制器之外，我还尝试使用了 RBF 网络来实现神经网络控制。

RBF（Radial Basis Function）网络是一种前馈神经网络，其核心思想是利用高斯核函数计算输入的非线性映射，主要用于逼近非线性函数和分类任务。更新过程的核心在于调整权值 w 和其他参数，使网络输出逐渐逼近目标值。

主要代码如下

Listing 6: RBF 神经网络控制器实现

```
function [sys,x0,str,ts,simStateCompliance] = rbfpid(t,x,u,flag)
% RBF神经网络PID控制器
% 网络结构：3-6-1
% 输入：误差e、误差积分、误差微分
% 输出：控制量u

% 采样时间
Ts = 0.001;

switch flag
    case 0
        [sys,x0,str,ts,simStateCompliance] = mdlInitializeSizes(Ts);
    case 2
        sys = mdlUpdate(x,u,Ts);
    case 3
        sys = mdlOutputs(t,x,u);
    case {1,4,9}
        sys = [];
    otherwise
        DAStudio.error('Simulink:blocks:unhandledFlag', num2str(flag))
            ;
end

function [sys,x0,str,ts,simStateCompliance] = mdlInitializeSizes(Ts)
    sizes = simsizes;

    sizes.NumContStates = 0;    % 连续状态个数
    sizes.NumDiscStates = 3;    % 离散状态个数
    sizes.NumOutputs   = 5;     % 输出个数
    sizes.NumInputs    = 4;     % 输入个数
```

```
30      sizes.DirFeedthrough = 1;   % 直通标志
31      sizes.NumSampleTimes = 1;   % 采样时间个数
32
33      sys = simsizes(sizes);
34      x0  = [0;0;0];              % 初始状态
35      str = [];
36      ts  = [Ts 0];               % 采样时间
37      simStateCompliance = 'UnknownSimState';
38
39  function sys = mdlUpdate(x,u,Ts)
40      % 更新离散状态
41      sys = [u(1);                % 误差e(k)
42            x(2) + u(1)*Ts;       % 误差积分
43            (u(1) - u(2))];       % 误差微分e(k)-e(k-1)
44
45  function sys = mdlOutputs(t,x,u)
46      persistent weights weights_prev weights_prev2 hidden_output centers
           pid_params
47
48      % 学习参数设置
49      learning_rate_pid = [0.01 0.01 1]; % PID参数学习率
50      learning_rate_rbf = 0.06;          % RBF网络学习率
51      momentum = 0.3;                    % 动量因子
52      gaussian_width = 5;                % 高斯函数宽度
53
54      if t == 0
55          % 初始化参数
56          pid_params = [0.1 0.2 0.1];    % PID参数初值[Kp Ki Kd]
57          % 初始化高斯基函数中心
58          centers = [linspace(-1,1,6);  % 误差范围
59                     linspace(0,1,6);    % 积分范围
60                     linspace(0,1,6)];   % 微分范围
61          hidden_output = zeros(6,1);    % 隐层输出
62          weights = zeros(6,1);          % 网络权值
63          weights_prev = weights;        % 上一次权值
64          weights_prev2 = weights_prev; % 上上次权值
65      end
66
```

```
67      % 计算PID控制器输出
68      control_output = pid_params*x;
69
70      % RBF网络输入
71      rbf_input = [control_output u(3) u(4)]';
72
73      % 计算隐层输出
74      for j = 1:6
75          hidden_output(j) = exp(-norm(rbf_input - centers(:,j))^2/(2*
                gaussian_width^2));
76      end
77
78      % 计算网络输出
79      network_output = weights'*hidden_output;
80
81      % 更新网络权值
82      delta_weights = learning_rate_rbf*(u(3) - network_output)*
            hidden_output;
83      weights = weights_prev + delta_weights + momentum*(weights_prev -
            weights_prev2);
84
85      % 计算雅可比矩阵
86      jacobian = weights.*hidden_output.*(-rbf_input(1) + centers(1,:))'/
            gaussian_width^2;
87      delta_output = sum(jacobian);
88
89      % 更新PID参数(限制为非负)
90      pid_params = max(pid_params + u(1) * delta_output * x' .*
            learning_rate_pid, 0);
91
92      % 保存历史权值
93      weights_prev2 = weights_prev;
94      weights_prev = weights;
95
96      % 输出结果
97      sys = [control_output;network_output;pid_params(:)];
```

RBF 的计算流程是这样的

1. **输入层计算**

$$\begin{aligned} u_{PID} &= \mathbf{K}_{PID} \cdot \mathbf{x} \\ \mathbf{x}_{RBF} &= [u_{PID}, e(k), \dot{e}(k)]^T \end{aligned} \tag{16}$$

其中 $\mathbf{K}_{PID}$ 为 PID 参数向量,$\mathbf{x}$ 为 PID 控制器的输入向量。

2. **隐含层高斯激活函数计算**

$$h_j = \exp\left(-\frac{\|\mathbf{x}_{RBF} - \mathbf{c}_j\|^2}{2b^2}\right), \quad j = 1, \ldots, 6 \tag{17}$$

其中 $\mathbf{c}_j$ 为第 j 个高斯基函数的中心向量,b 为高斯函数的宽度参数。

3. **输出层计算**

$$y_{out} = \mathbf{w}^T \mathbf{h} \tag{18}$$

其中 $\mathbf{w}$ 为输出层权值向量,$\mathbf{h}$ 为隐含层输出向量。

4. **权值更新**

$$\begin{aligned} \Delta \mathbf{w} &= \eta (y_d - y_{out}) \mathbf{h} \\ \mathbf{w}(k) &= \mathbf{w}(k-1) + \Delta \mathbf{w} + \alpha [\mathbf{w}(k-1) - \mathbf{w}(k-2)] \end{aligned} \tag{19}$$

其中 η 为学习率,α 为动量因子,y_d 为期望输出。

5. **Jacobian 矩阵计算**

$$\mathbf{J} = \mathbf{w} \odot \mathbf{h} \odot \frac{-\mathbf{x}_{RBF}(1) + \mathbf{c}_1}{b^2} \tag{20}$$

其中 $\odot$ 表示 Hadamard 积 (逐元素相乘)。

通过以上计算流程,RBF 网络可以实现对非线性系统的在线辨识和控制。其中高斯基函数提供了输入空间的局部响应特性, 而权值的在线调整则保证了网络的自适应学习能力。

了解 RBF 网络的计算流程之后，我尝试在 simulink 中搭建 RBF 神经网络，并同样首先在一阶传递函数被控对象上进行了尝试，发现可以很好的控制。

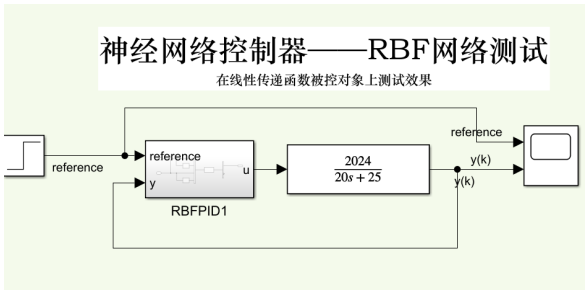

Figure 29: 一阶传递函数被控对象

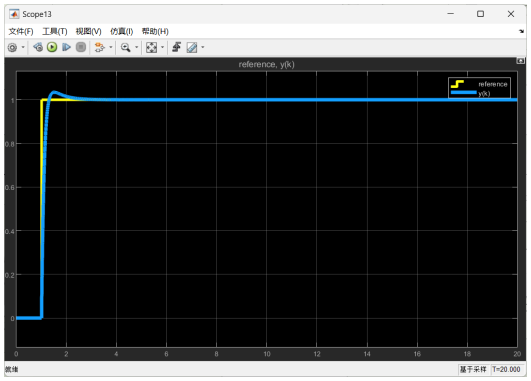

Figure 30: 一阶传递函数被控对象仿真结果

在验证了 RBF 网络的控制效果之后，我尝试将 RBF 网络应用到永磁同步电机的速度控制中，结果如图33所示

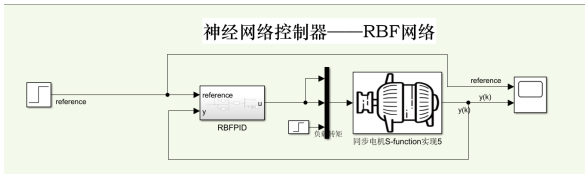

Figure 31: RBF 神经网络结构

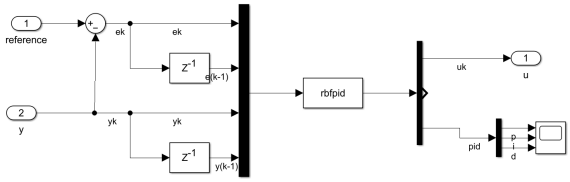

Figure 32: RBF 控制器内部结构

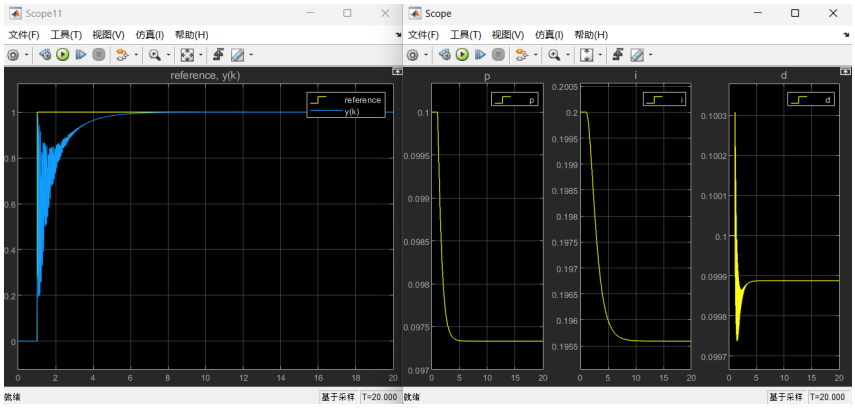

Figure 33: 永磁同步电机速度控制仿真结果

7 结果比较

在比较了各个控制器之后，我得到了如下的结果：

从仿真结果中，虽然各个控制器的表现都很好，但是可以对四种控制器的性能进行对比分析：

Table 1: 各控制器性能指标对比

控制器类型	稳定时间	超调量	稳态误差
模糊 PID 控制器	最短	较大	最小
BPPID 控制器	较短	最小	最小
RBF 神经网络控制器	较慢	较小	较小
模糊控制器	较慢	较大	较大

从表1中可以看出：

- **稳定时间**方面：模糊 PID 控制器表现最好，达到稳态时间最短；其次是 BPPID 控制器；RBF 和模糊控制器相对较慢。
- **超调量**方面：BPPID 神经网络控制器的超调最小；RBF 次之；而模糊 PID 和模糊控制器的超调量相对较大。
- **稳态误差**方面：模糊 PID 和 BPPID 控制器的稳态误差最小；RBF 神经网络控制器次之；模糊控制器的稳态误差相对较大。

8 Matlab 工具包学习记录

8.1 Deep Learning Toolbox 学习

首先我根据助教的视频，和 b 站上的视频，学习了神经网络工具箱的基本用法，具体用到的数据和代码放到 `learn_toolbox` 文件夹中

当然，使用工具箱自带的一些数据集也是可以的。

Listing 7: 打开神经网络工具箱

```
nftool
```

- 选择导入数据集
- 正确选择是行还是列
- 选择训练集的比例；隐藏层的个数（5or10）
- 选择训练方法：一般选择第一个方法；第二个方法和遗传算法一起用；第三个方法比较慢

提示: validation test 表示的是泛化性能，如果连续 6 个 epoch 都上不去，就停止训练

图像查看：一般看回归的图（第四个），如果都上了 0.9 差不多就可以了

模型保存

Listing 8: 保存模型

```
save('filename.mat') % 保存所有变量
save('filename.mat', 'var1', 'var2') % 只保存指定变量
```

Listing 9: 从.mat 文件中导入数据

```
load('data.mat');
```

模型预测

Listing 10: 方法 1：使用 sim 函数

```
PreY = zeros(10,1);
for i = 1:10
    PreY(i,1) = sim(net,PreX(i,:)');
    % sim函数第二个参数列数等于输入向量的个数
end
disp(PreY);
```

Listing 11: 方法 2：使用生成的函数

```
myNeuralNetworkFunction(X) % 这里需要把后两个参数去掉
```

多输入多输出

多输入多输出也是一样的操作，唯一值得注意的地方就是在训练之前需要将行还是列选择正确（特征 or 样本）

使用脚本替代 ui 操作

Listing 12: 使用脚本替代 ui 操作，并且函数化

```
function train()
    dataFile = 'data.mat';
    load(dataFile); % 从文件导入数据
    x = features'; % 转置为符合网络输入格式
    t = labels'; % 转置为符合网络目标格式

    % 选择训练函数
    trainFcn = 'trainlm'; % Levenberg-Marquardt 反向传播

    % 创建拟合网络
    hiddenLayerSize = 15;
    net = fitnet(hiddenLayerSize, trainFcn);

    % 输入输出的预处理函数
    net.input.processFcns = {'removeconstantrows', 'mapminmax'};
    net.output.processFcns = {'removeconstantrows', 'mapminmax'};

    % 设置数据的划分方式
    net.divideFcn = 'dividerand'; % 随机划分数据
    net.divideMode = 'sample'; % 划分所有样本
    net.divideParam.trainRatio = 70/100;
    net.divideParam.valRatio = 15/100;
    net.divideParam.testRatio = 15/100;

    % 选择性能函数
    net.performFcn = 'mse'; % 均方误差

    % 选择绘图函数
    net.plotFcns = {'plotperform', 'plottrainstate', 'ploterrhist', ...
```

```
        'plotregression', 'plotfit'};

    % 训练网络
    [net, tr] = train(net, x, t);

    % 测试网络
    y = net(x);
    e = gsubtract(t, y);
    performance = perform(net, t, y);

    % 重新计算训练、验证和测试性能
    trainTargets = t .* tr.trainMask{1};
    valTargets = t .* tr.valMask{1};
    testTargets = t .* tr.testMask{1};
    trainPerformance = perform(net, trainTargets, y);
    valPerformance = perform(net, valTargets, y);
    testPerformance = perform(net, testTargets, y);

    figure, plotperform(tr);% 绘制并保存训练性能图
%     figure, plottrainstate(tr);% 绘制并保存训练状态图
%     figure, ploterrhist(e);% 绘制并保存误差直方图
    figure, plotregression(t, y)
%     figure, plotfit(net, x, t);% 绘制并保存拟合图

    % 保存模型
%     modelFile = ['model_level' num2str(level) '.mat'];
%     save(modelFile, 'net'); % 保存神经网络模型
    % 生成simulink模型
    gensim(net);
    disp(['Model saved as ' modelFile]);
end
```

使用 Control System Designer 进行自动调参，可以大大提高调参效率。

界面如下，拖动两个按钮可以修改曲线参数，按下按钮两边得大于/小于号，可以拓宽，收缩调节范围。

点击 show Parameter 显示曲线信息，有曲线的超调量，增益，达到设定值的时间，曲线是否稳定等。可以边拖动曲线，边看表格。

可以看到有两条曲线，一条是上次 PID 值下得曲线，一条是目前 PID 值下得曲线。

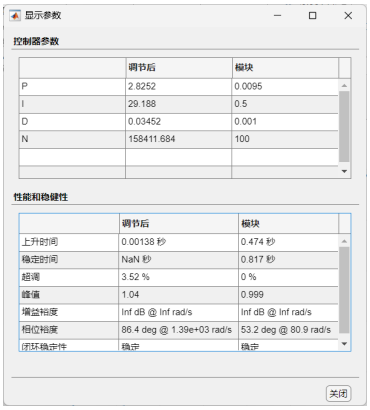

Figure 34: Control System Designer 自动调参

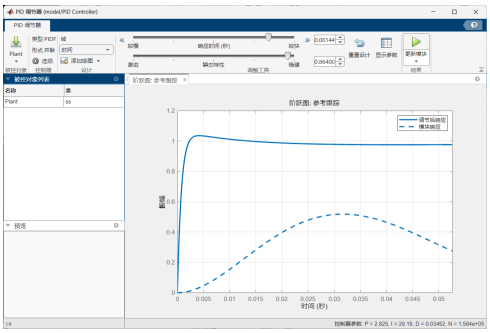

Figure 35: Control System Designer 自动调参结果

找到比较合适的曲线，更新到 simulink 中，点击 update 按钮，就可以更新到 simulink 中。

8.2 MPC Designer 学习

A 附录 1：文件夹结构

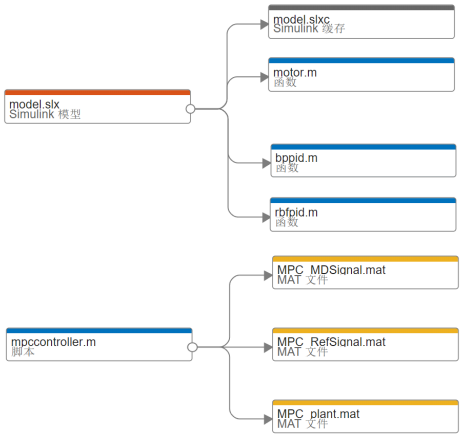

Figure 36: 文件依存关系图

```
├── README.md ..... 项目说明文件
├── assets ..... 存放图片资源的文件夹
├── report
│   ├── thesis.tex ..... 主要的 LaTeX 源文件
│   └── thesis.pdf ..... 生成的 PDF 报告
├── src ..... 源代码文件夹
│   ├── bppid.m ..... BP 神经网络 PID 控制器实现
│   ├── Fuzzy_Control.fis ..... 模糊控制器设计文件
│   ├── Fuzzy_PID.fis ..... 模糊 PID 控制器设计文件
│   ├── model.slx ..... Simulink 模型文件
│   ├── model.slxc ..... Simulink 缓存文件
│   ├── motor.m ..... 电机模型实现
│   ├── mpccontroller.m ..... MPC 控制器实现
│   ├── Project.prj ..... 项目配置文件
│   ├── rbfpid.m ..... RBF 神经网络 PID 控制器实现
│   └── test.fis ..... 测试用模糊系统文件
└── thesis ..... 参考文献文件夹
    ├── 一种永磁同步电机无模型高阶滑模控制算法 _ 赵凯辉.pdf ..... 参考论文
    ├── 空间矢量坐标变换（等幅值变换）- 知乎.html ..... 参考资料
    └── 空间矢量坐标变换（等幅值变换）- 知乎 _files ..... 参考资料附件
```

B 附录 2：版本更新记录

Table 2: 版本更新记录

时间	更新内容
12.27 上午	完成 PID 控制器设计与调参
12.27 下午	完成 Control System Designer 自动调参与实验
12.28 上午	完成 MPC 模型预测控制器设计与实现
12.29 上午	完成模糊控制器设计与模糊 PID 控制实现
12.30 上午	完成 BP-PID 神经网络控制器设计
12.30 下午	完成 RBF 神经网络控制器实现与测试
12.31 下午	完成实验结果分析与报告撰写

C 附录 3：总结与经验感悟

通过半个学期的学习，我从一个完全不会使用 simulink 的小白，现在可以在 simulink 中快速搭建一个模型进行仿真的实验。非常感谢老师和助教的指导。

智能控制这门课程算是比较硬核的，4 个小作业，每个作业都涉及到了不同的控制方法，并且需要使用不同的工具箱，而大作业则是需要将这些方法结合起来，完成一个控制的系统的搭建。学习各个工具包的用法确实是需要很多时间和精力投入的，每次作业我都要花蛮久的时间进行学习和实验，但是我觉得投入的时间确实学到了很多东西。学到的专家控制和模糊控制方法也用在了另一门课程的大作业课程设计中。

比较痛苦的是作业可能缺少一些难度的阶梯，比如在每次小作业讲解的时候，如果可以安排一个小例子或者简单的模型来验证工具包安装和使用的正确性，那么可能可以减少很多同学的疑惑。

在本门课程中，我也整理了一些我在课堂的笔记和实验上的一些心得，放在了我的个人网站上，可以访问下边的链接查看课程笔记，matlab 工具包学习记录