Department of Computer and Systems Engineering

2024/2025

CSE 325 **Process Control**

Dr. Ahmed Osman Mahmoud

PID Tuning

Bayesian Optimization Method

رقم الجلوس	الاسم
6358	محمد محمود محمد غانم
6349	كريم وائل طلبه محمد

Goal of PID Tuning:

In a PID controller, we try to find the **best values** for:

- Kp : Proportional gain
- Ki : Integral gain
- Kd : Derivative gain

So that the system:

- Responds quickly
- Has minimal overshoot
- Reaches steady state with low error
- Remains stable

What is Bayesian Optimization (BO)?

Bayesian Optimization is a statistical method used to optimize black-box functions that are expensive to evaluate. In the context of PID controller tuning, BO helps to efficiently find optimal values for the controller's proportional, integral, and derivative gains by iterating through potential settings and learning from each iteration's feedback. This method is particularly advantageous in scenarios where real-time feedback from controllers is necessary and where traditional tuning methods may be time-consuming or risky.

It’s great when:

- Simulations are slow
- The system is a "black box" (you don’t know the math model)
- You want the **best result with few evaluations**

How BO PID Tuning Works (Conceptually)

1. Define an Objective Function

- You choose a **performance metric** to minimize — for example:
    - **ITAE** (Integral of Time-weighted Absolute Error)
    - Rise time
    - Overshoot
    - A combination of these

This becomes the "**score**" for each set of PID gains.

2. Tell It What Values to Try

- You define the range of possible values for Kp, Ki, and Kd.
- For example: "Try Kp between 0.1 and 10".

3. Start With a Few Random Tests

- BO tries a few random combinations of gains.
- It simulates the system and records the performance scores.

4. Build a Model

- BO uses these results to build a **surrogate model** — a prediction of how different PID gains will perform across the whole range.

5. Predict & Decide What to Try Next

- BO uses the model to guess which new gain values are most promising.
- It **balances**:
    - Trying areas that might be better (**exploration**)
    - Trying areas known to be good (**exploitation**)

6. Run a New Test

- It tests a new set of gains and observes the result.
- Then it updates its model with the new information.

7. Repeat

- This cycle of **predicting** → **testing** → **learning** continues for a set number of rounds (e.g**., 20 iterations**).

8. Return the Best Gains

- After all **iterations**, BO reports the **PID settings that gave the best performance** during the testing.

Why Use Bayesian Optimization?

Advantage	Explanation
Efficient	Finds good solutions with fewer tests than random or grid search.
Smart Search	Uses previous results to make better decisions.
No Need for Gradients	Works even if the system is non-linear or black-box (like a Simulink model).
Customizable	You can define your own "what makes a good system" rule.

Conclusion

Bayesian Optimization provides a **modern** and **efficient** method to auto-**tune** PID controllers in **MATLAB**. By utilizing its **iterative**, probabilistic framework, users can achieve **optimal** system performance while **minimizing resource** use, making it an essential tool for systems requiring precision control. This approach is particularly beneficial in industrial applications where **tuning** traditional PID parameters **manually** is **impractical** and **time-consuming**.

Bayesian Optimization in MATLAB

Bayesian_Optimization.m

/MATLAB Drive/PID tuning/Bayesian_Optimization.m

```matlab
%% Define the plant model (symbolic - also used in Simulink)
s = tf('s');
plant = 1 / (s^2 + 3*s + 2);  % Second-order transfer function of the plant

%% Bayesian Optimization Setup

% Define the objective function to minimize: ITAE calculated from Simulink
objective = @(x) pid_objective_simulink(x.Kp, x.Ki, x.Kd);

% Define the PID gain variables with their bounds
vars = [
    optimizableVariable('Kp',[0.1, 10]);   % Proportional gain range
    optimizableVariable('Ki',[0.1, 10]);   % Integral gain range
    optimizableVariable('Kd',[0.01, 5]);   % Derivative gain range
];

% Run Bayesian optimization to minimize the ITAE cost function
results = bayesopt(objective, vars, ...
    'MaxObjectiveEvaluations', 20, ...              % Number of iterations
    'IsObjectiveDeterministic', true, ...           % Same output each run
    'AcquisitionFunctionName', 'expected-improvement-plus', ...  % BO strategy
    'Verbose', 0);                                  % Suppress detailed output

% Extract the best PID gains found
bestParams = results.XAtMinObjective;
Kp = bestParams.Kp;
Ki = bestParams.Ki;
Kd = bestParams.Kd;

% Display the optimized PID parameters
fprintf("Best PID gains:\nKp = %.3f\nKi = %.3f\nKd = %.3f\n", Kp, Ki, Kd);
```

Bayesian_Optimization.m

/MATLAB Drive/PID tuning/Bayesian_Optimization.m

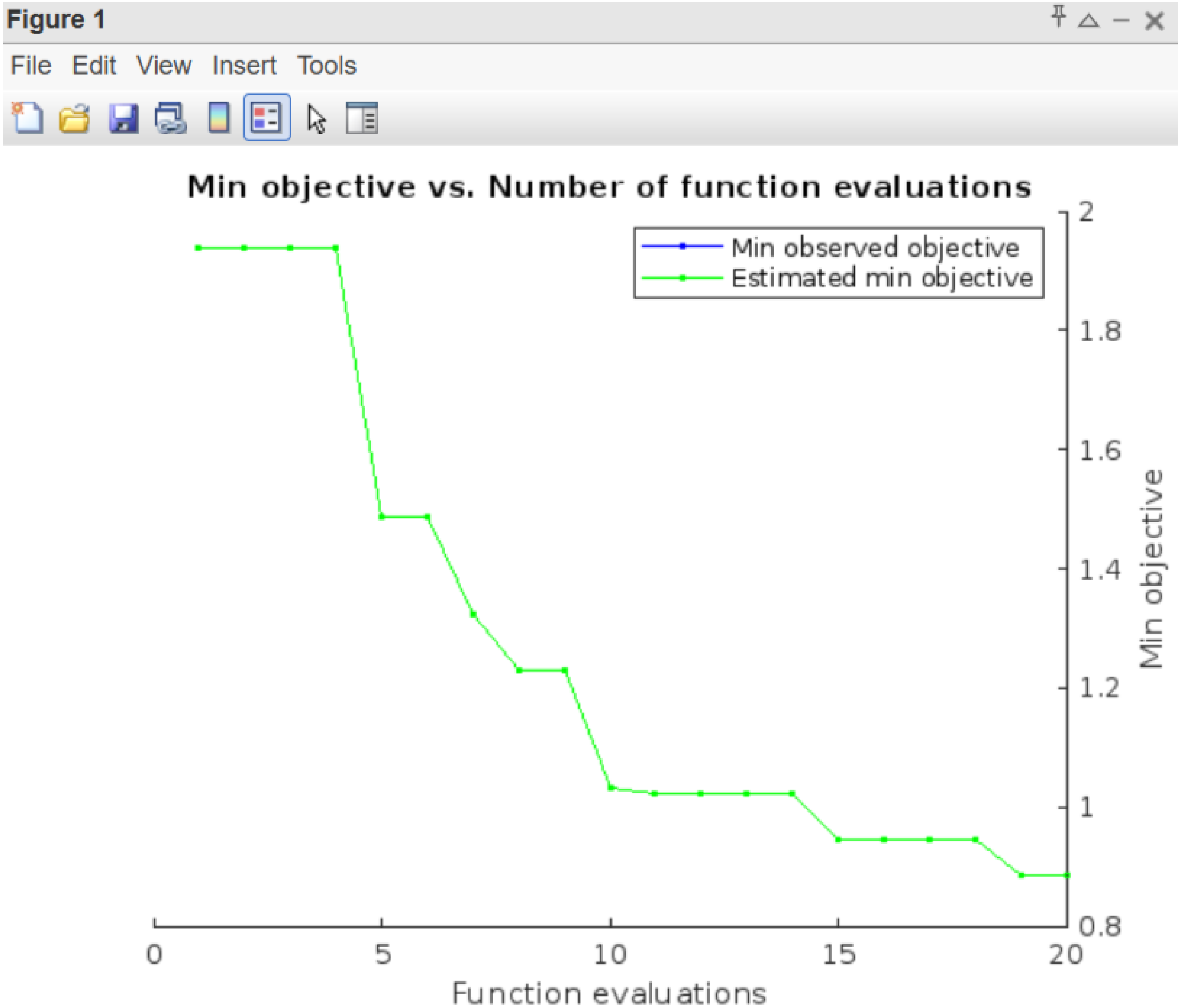
```matlab

%% Run Simulink model with best PID gains
simOut = sim('pid_sim_model');  % Simulate model with Kp, Ki, Kd from base workspace

% Extract output data from simulation
y = simOut.yout{1}.Values.Data;   % System response
t = simOut.yout{1}.Values.Time;   % Time vector
u = ones(size(t));                % Step input signal

%% Plot Step Input and System Response
figure;
plot(t, u, 'r--', 'LineWidth', 1.5); hold on;   % Plot step input in red dashed line
plot(t, y, 'b', 'LineWidth', 2);                % Plot system response in blue
title('PID Response By Bayesian-Optimization');
xlabel('Time (seconds)');
ylabel('Amplitude');
legend('Step Input', 'System Output');
grid on;

%% Objective Function Used by Bayesian Optimization
function cost = pid_objective_simulink(Kp, Ki, Kd)
    % Assign PID gains to base workspace for use in Simulink
    assignin('base', 'Kp', Kp);
    assignin('base', 'Ki', Ki);
    assignin('base', 'Kd', Kd);

    try
        % Run the Simulink model and return outputs to workspace
        simOut = sim('pid_sim_model', 'ReturnWorkspaceOutputs', 'on');

        % Extract system output and time
        y = simOut.yout{1}.Values.Data;
```

Bayesian_Optimization.m

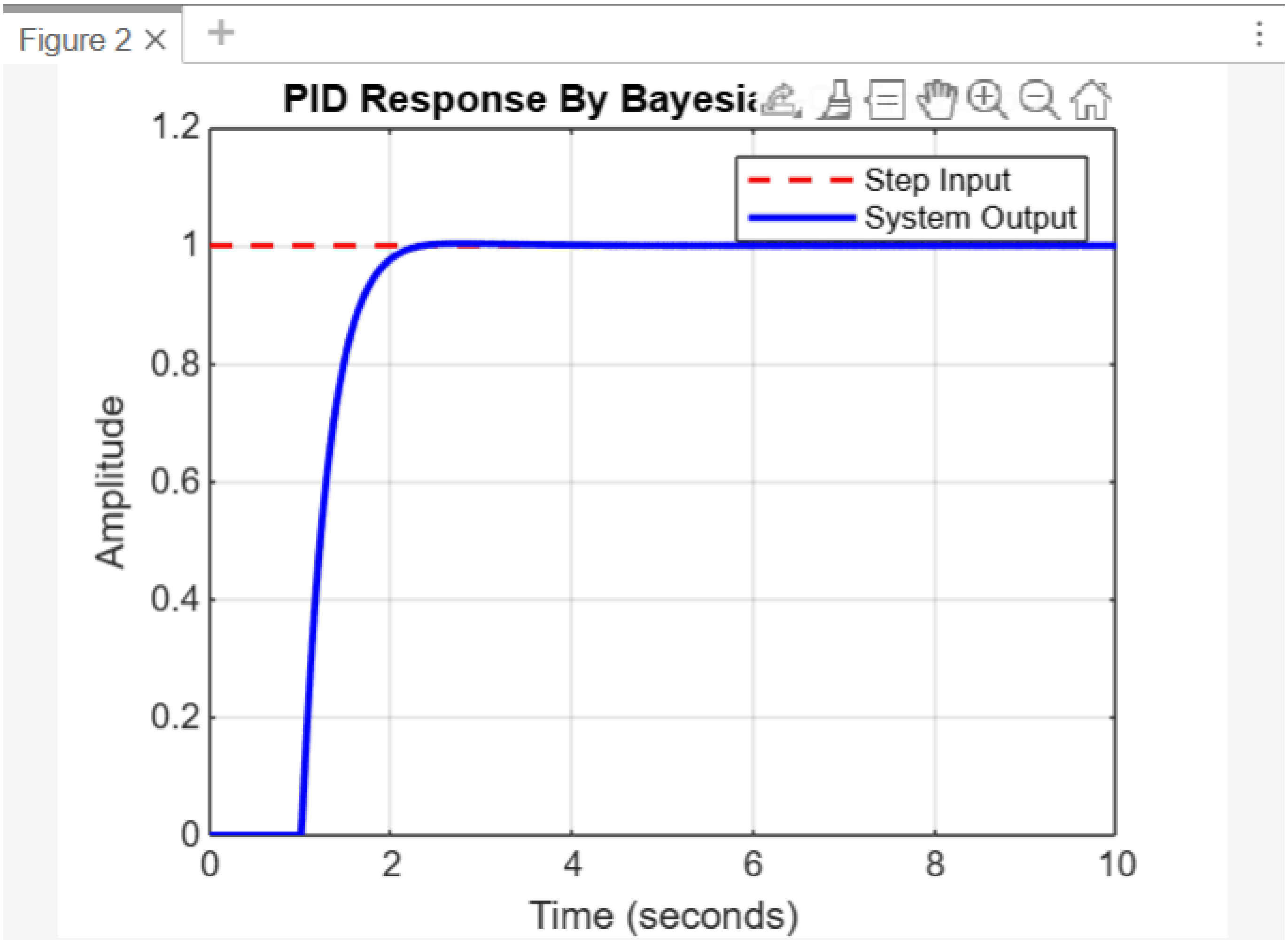
/MATLAB Drive/PID tuning/Bayesian_Optimization.m

```matlab

        % Extract system output and time
        y = simOut.yout{1}.Values.Data;
        t = simOut.yout{1}.Values.Time;

        % Compute the error (for step input = 1)
        error = 1 - y;

        % Calculate ITAE: Integral of Time-weighted Absolute Error
        cost = trapz(t, t .* abs(error));
    catch
        % If simulation fails (e.g., unstable parameters), return high cost
        cost = inf;
    end
end

```

Min objective vs. Number of function evaluations



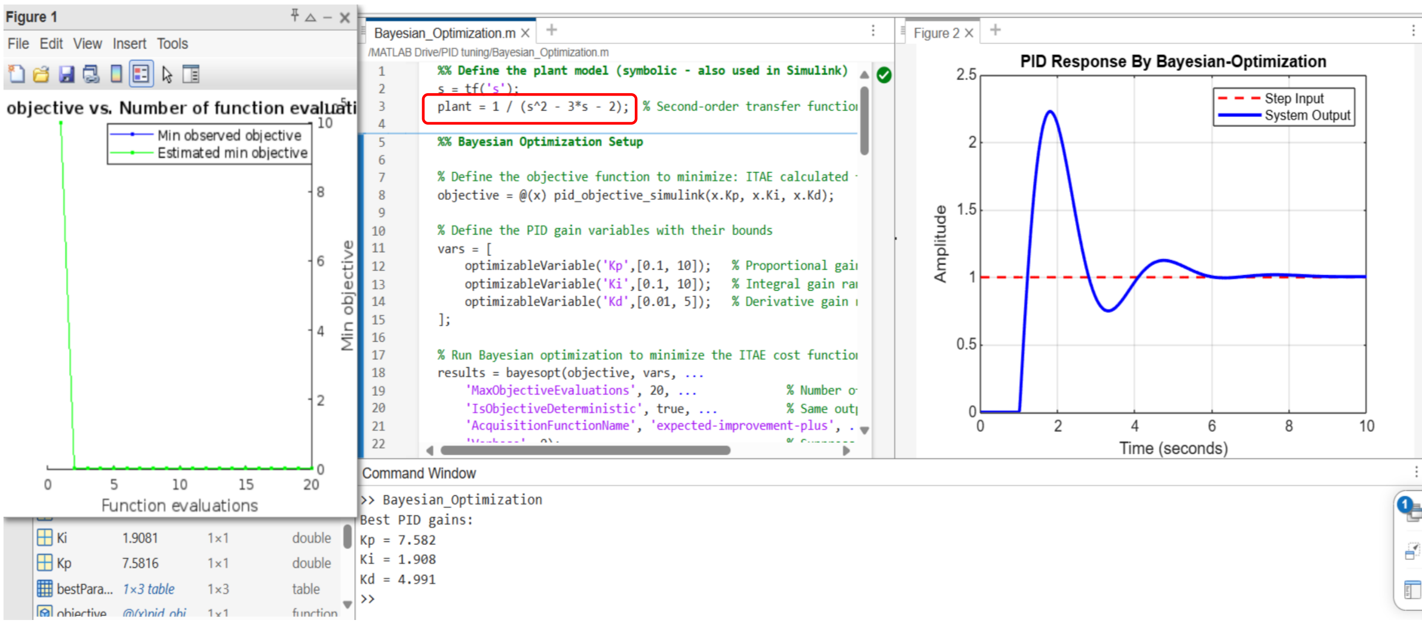
PID Response By Bayesian-Optimization



Command Window

```
>> Bayesian_Optimization
Best PID gains:
Kp = 9.991
Ki = 6.975
Kd = 3.095
>>
```

When system or plant is unstable



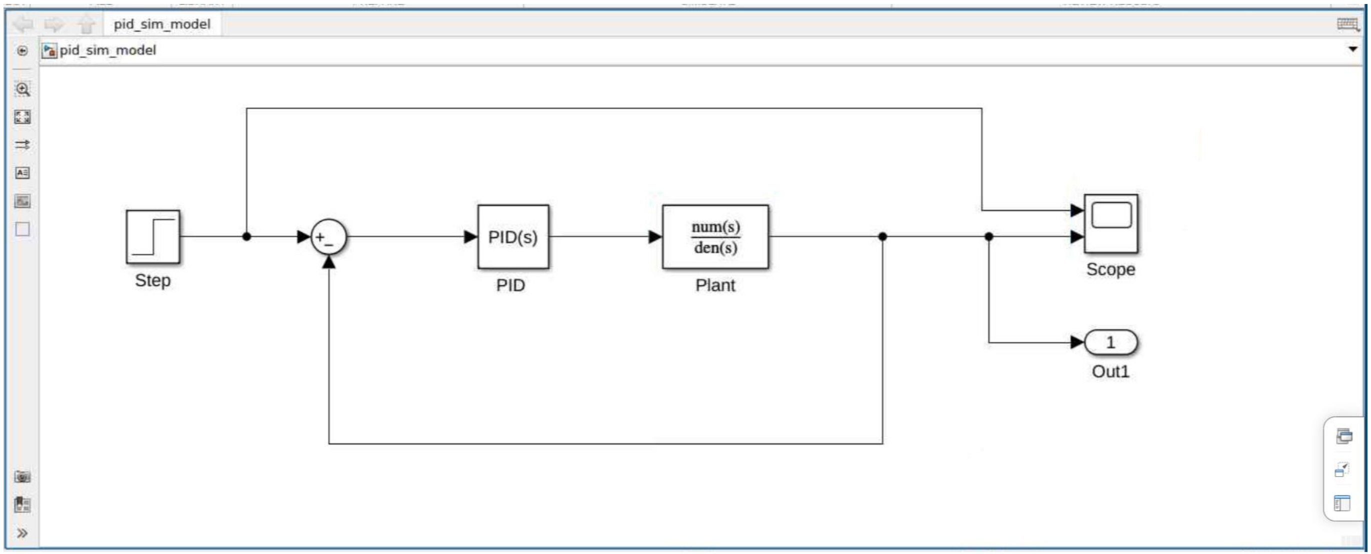
Change Function cost to handle unstable system

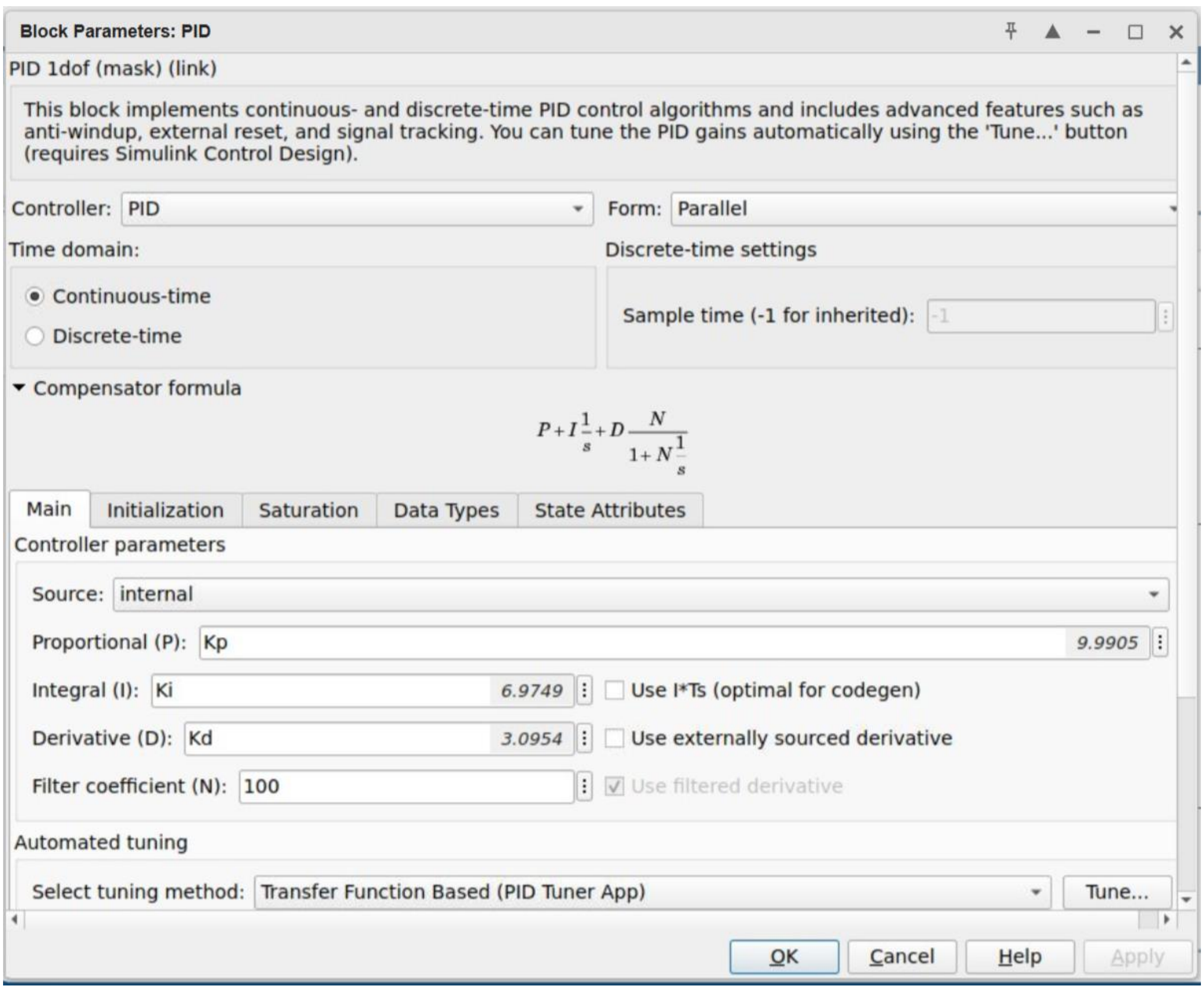
Bayesian_Optimization.m

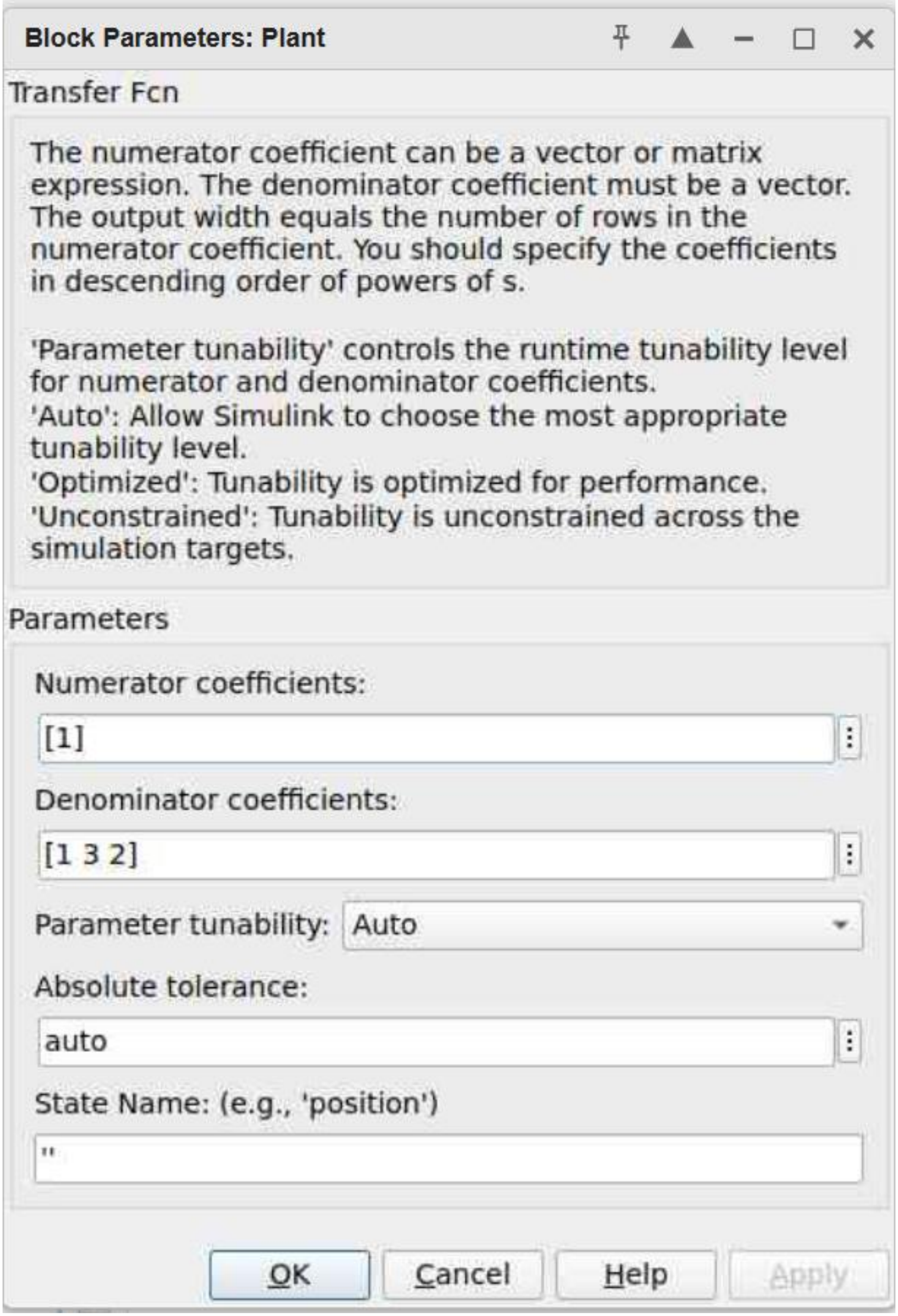
/MATLAB Drive/PID tuning/Bayesian_Optimization.m

```
%% Objective Function Used by Bayesian Optimization
function cost = pid_objective_simulink(Kp, Ki, Kd)
    % Assign PID gains to base workspace for use in Simulink
    assignin('base', 'Kp', Kp);
    assignin('base', 'Ki', Ki);
    assignin('base', 'Kd', Kd);

    try
        % Run the Simulink model and return outputs to workspace
        simOut = sim('pid_sim_model', 'ReturnWorkspaceOutputs', 'on');

        % Extract system output and time
        y = simOut.yout{1}.Values.Data;
        t = simOut.yout{1}.Values.Time;

        % Unstable detection threshold
        if any(isnan(y)) || any(isinf(y)) || max(abs(y)) > 1e3
            cost = 1e6;  % Assign a large penalty cost
            return;
        end

        % Compute the error (for unit step input)
        error = 1 - y;

        % Calculate ITAE: Integral of Time-weighted Absolute Error
        cost = trapz(t, t .* abs(error));

    catch
        % If simulation fails (e.g., divergence or runtime error), assign high cost
        cost = 1e6;
    end
end
```

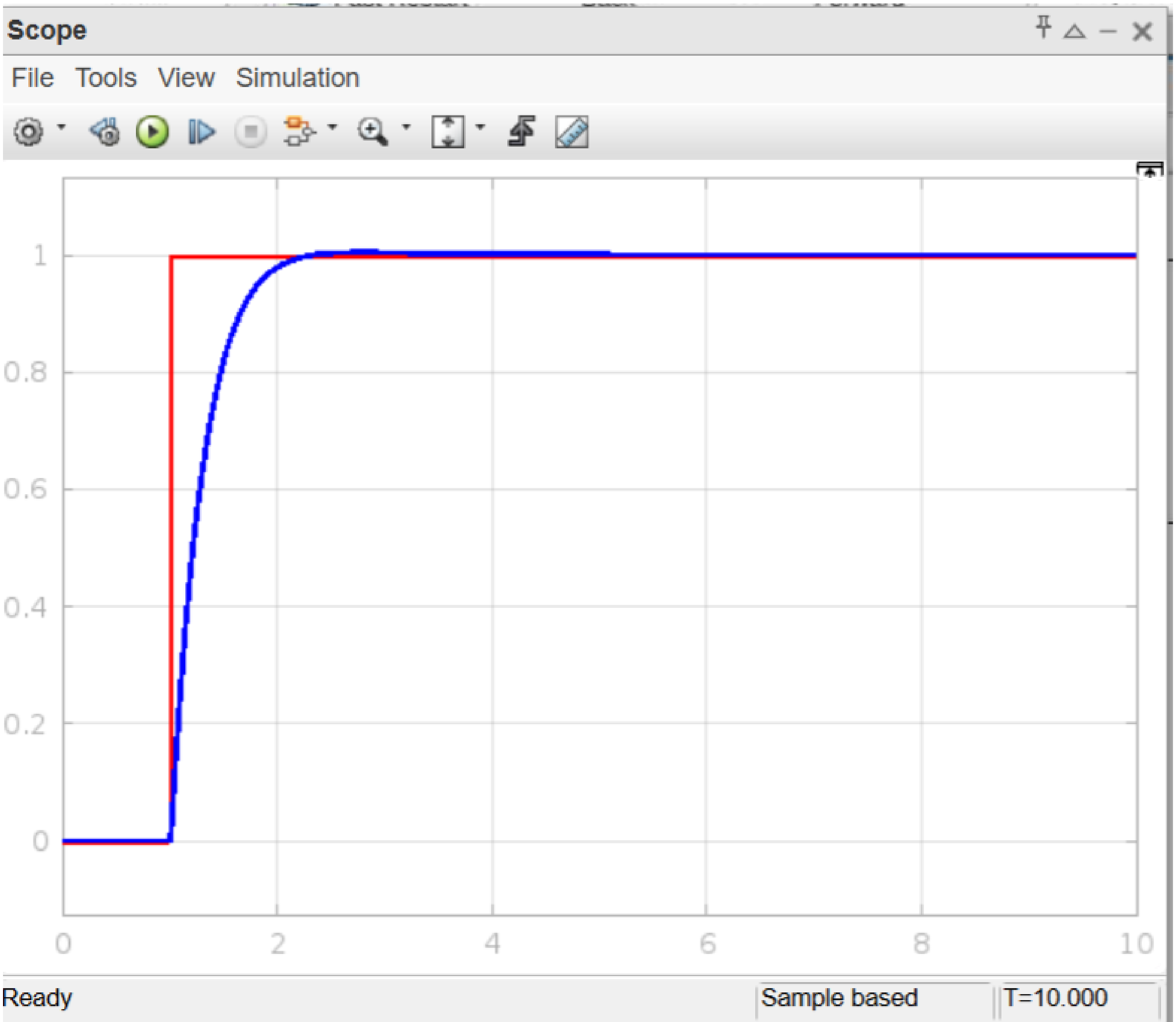
Bayesian Optimization in Simulink






Block Parameters: Plant
Transfer Fcn
The numerator coefficient can be a vector or matrix expression. The denominator coefficient must be a vector. The output width equals the number of rows in the numerator coefficient. You should specify the coefficients in descending order of powers of s.
'Parameter tunability' controls the runtime tunability level for numerator and denominator coefficients.
'Auto': Allow Simulink to choose the most appropriate tunability level.
'Optimized': Tunability is optimized for performance.
'Unconstrained': Tunability is unconstrained across the simulation targets.
Parameters
Numerator coefficients:
[1]
Denominator coefficients:
[1 3 2]
Parameter tunability:
Auto
Absolute tolerance:
auto
State Name: (e.g., 'position')
''
OK
Cancel
Help
Apply

Scope output



Good gain tuning (Another method)

"Good Gain" tuning is a practical method for tuning PID controllers, especially suitable for industrial applications. It focuses primarily on tuning the **proportional gain (Kp)** to achieve a desired response with minimal overshoot and then refining the response with integral (Ki) and derivative (Kd) actions. Here's a quick breakdown:

Steps in Good Gain Tuning:

1. **Start with Ki = 0 and Kd = 0**.
2. **Increase Kp** gradually until the system responds quickly but without oscillation or too much overshoot.
    - This point is referred to as the "good gain" — a value that gives a fast yet stable response.
3. **Add integral action (Ki)** to eliminate steady-state error.
    - Increase Ki carefully to avoid introducing oscillations or instability.
4. **Add derivative action (Kd)** to dampen oscillations and improve transient response, if needed.

Advantages:

- Simple and intuitive.
- Doesn't require complex modeling.
- Effective for systems with slow dynamics or where overshoot must be minimized.

Limitations:

- Might not be optimal for fast or unstable systems.
- Manual and iterative — requires engineering intuition.

pid_good_gain_gui in MATLAB

pid_good_gain_gui.m

/MATLAB Drive/PID tuning/pid_good_gain_gui.m

```
function pid_good_gain_gui
    % Create the Main Figure
    fig = figure('NumberTitle', 'off', ...
                 'Color', [0.95 0.95 0.95], ...
                 'Position', [300 100 800 600]);

    % Define Plant
    s = tf('s');
    G = 1 / (s^2 + 10*s + 20); % Example Plant

    % Axes for Plot
    ax = axes('Parent', fig, 'Position', [0.08 0.45 0.88 0.5]);
    grid on;
    hold on;
    xlabel('Time (s)');
    ylabel('Amplitude');
    set(gca, 'FontSize', 12);

    % -------- Sliders and Labels --------

    % Kp Slider and Label
    uicontrol('Style', 'text', 'Position', [100 200 60 20], ...
              'String', 'Kp:', 'FontSize', 12, ...
              'BackgroundColor', [0.95 0.95 0.95]);
    kp_slider = uicontrol('Style', 'slider', ...
        'Min', 0, 'Max', 1000, 'Value', 5, ...
        'Position', [160 200 500 20], ...
        'SliderStep', [0.001 0.05], ...
        'Callback', @update_plot);

    % Kp Value Display
    kp_value_display = uicontrol('Style', 'text', ...
```

pid_good_gain_gui.m

/MATLAB Drive/PID tuning/pid_good_gain_gui.m

```
        kp_value_display = uicontrol('Style', 'text', ...
                                     'Position', [680 200 100 20], ...
                                     'String', num2str(kp_slider.Value, 'Kp: %.2f'), ...
                                     'FontSize', 12, ...
                                     'BackgroundColor', [0.95 0.95 0.95]);

        % Ki Slider and Label
        uicontrol('Style', 'text', 'Position', [100 160 60 20], ...
                  'String', 'Ki:', 'FontSize', 12, ...
                  'BackgroundColor', [0.95 0.95 0.95]);
        ki_slider = uicontrol('Style', 'slider', ...
            'Min', 0, 'Max', 500, 'Value', 0, ...
            'Position', [160 160 500 20], ...
            'SliderStep', [0.001 0.05], ...
            'Callback', @update_plot);

        % Ki Value Display
        ki_value_display = uicontrol('Style', 'text', ...
                                     'Position', [680 160 100 20], ...
                                     'String', num2str(ki_slider.Value, 'Ki: %.2f'), ...
                                     'FontSize', 12, ...
                                     'BackgroundColor', [0.95 0.95 0.95]);

        % Kd Slider and Label
        uicontrol('Style', 'text', 'Position', [100 120 60 20], ...
                  'String', 'Kd:', 'FontSize', 12, ...
                  'BackgroundColor', [0.95 0.95 0.95]);
        kd_slider = uicontrol('Style', 'slider', ...
            'Min', 0, 'Max', 500, 'Value', 0, ...
            'Position', [160 120 500 20], ...
            'SliderStep', [0.001 0.05], ...
            'Callback', @update_plot);
```

pid_good_gain_gui.m

/MATLAB Drive/PID tuning/pid_good_gain_gui.m

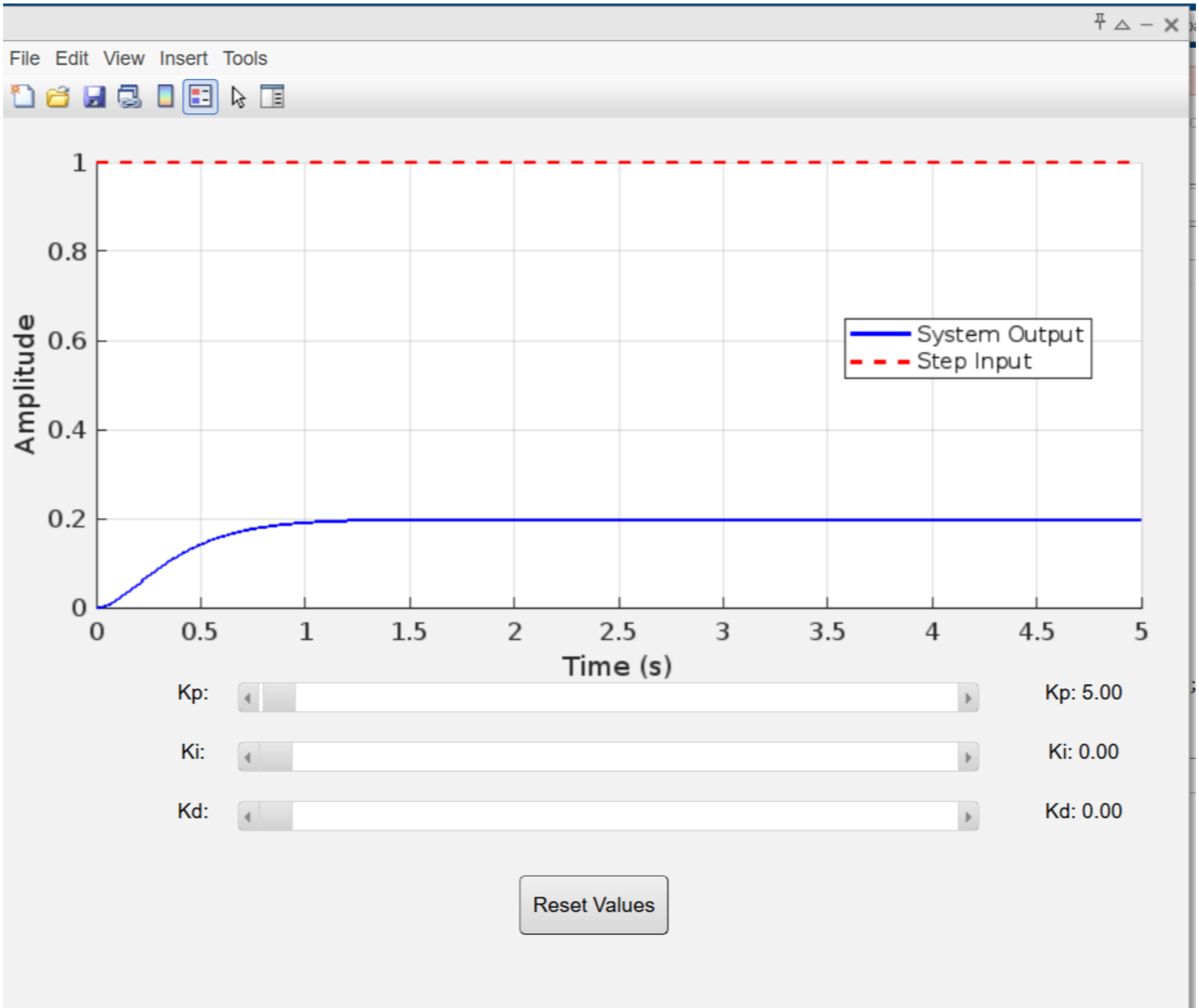
```
            'Callback', @update_plot);

        % Kd Value Display
        kd_value_display = uicontrol('Style', 'text', ...
                                      'Position', [680 120 100 20], ...
                                      'String', num2str(kd_slider.Value, 'Kd: %.2f'), ...
                                      'FontSize', 12, ...
                                      'BackgroundColor', [0.95 0.95 0.95]);

        % Reset Button
        uicontrol('Style', 'pushbutton', 'String', 'Reset Values', ...
                  'FontSize', 12, ...
                  'Position', [350 50 100 40], ...
                  'Callback', @reset_values);

        % Initial Plot
        update_plot();

        function update_plot(~, ~)
            % Get current Kp, Ki, Kd values
            Kp = kp_slider.Value;
            Ki = ki_slider.Value;
            Kd = kd_slider.Value;

            % Update value displays
            kp_value_display.String = sprintf('Kp: %.2f', Kp);
            ki_value_display.String = sprintf('Ki: %.2f', Ki);
            kd_value_display.String = sprintf('Kd: %.2f', Kd);

            % PID Controller
            C = pid(Kp, Ki, Kd);

```

pid_good_gain_gui.m

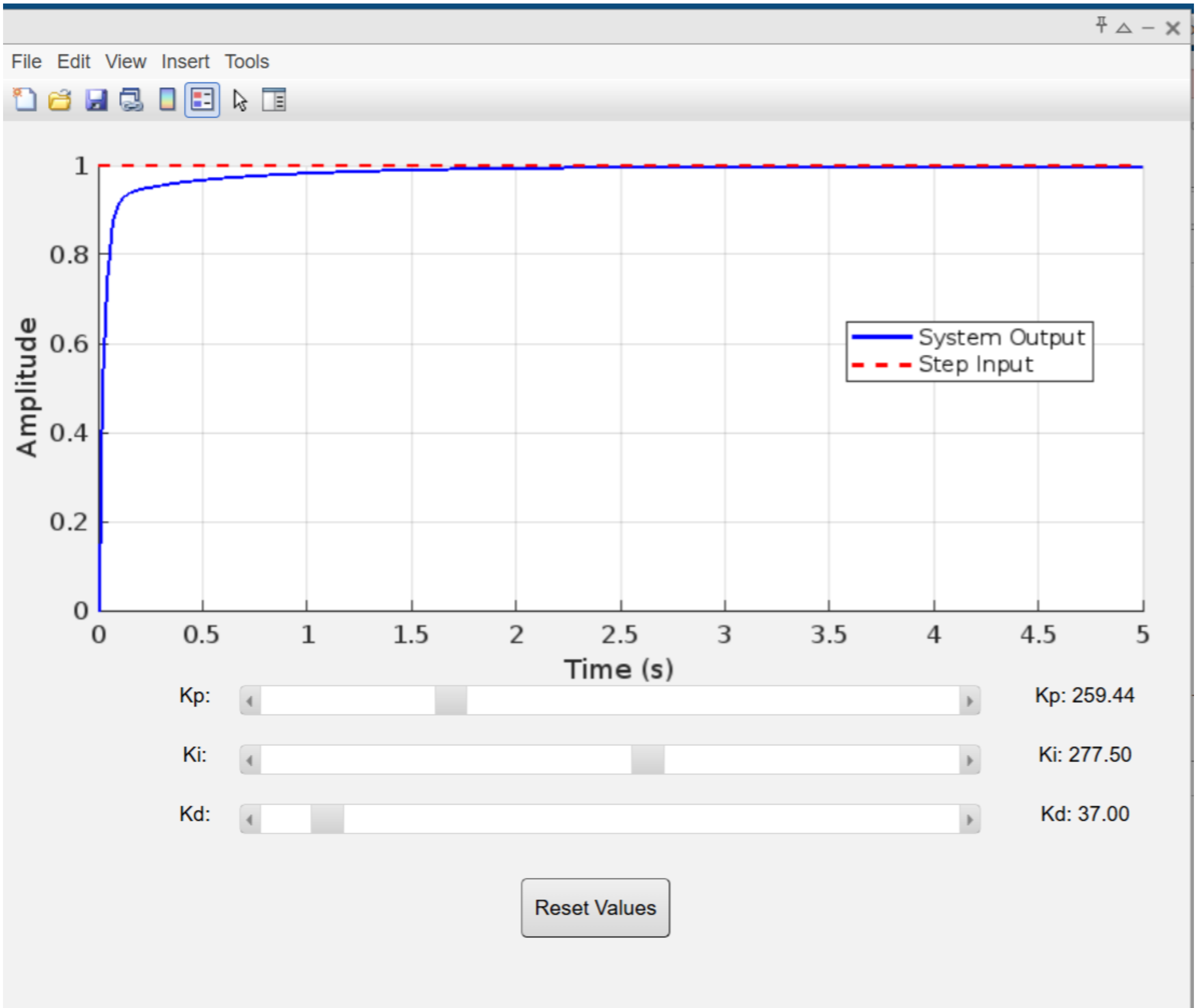
/MATLAB Drive/PID tuning/pid_good_gain_gui.m

```

        % Closed Loop System
        T = feedback(C*G, 1);

        % Time vector
        t = 0:0.01:5;

        % Step response
        [y, t_out] = step(T, t);

        % Step input
        u = ones(size(t_out));

        % Clear axes and plot
        cla(ax);
        plot(ax, t_out, y, 'b-', 'LineWidth', 2); hold on;
        plot(ax, t_out, u, 'r--', 'LineWidth', 2);
        legend(ax, {'System Output', 'Step Input'}, 'Location', 'best');
        grid(ax, 'on');
    end

    function reset_values(~, ~)
        % Reset to initial PID values
        kp_slider.Value = 5;
        ki_slider.Value = 0;
        kd_slider.Value = 0;
        update_plot();
    end
end
```

GUI , Initial State



Tune PID by Sliders of Kp ,Ki, Kd to better response



Another Example in good gain method >>The same that in Bayesian-Optimization