الأكاديمية العربية للعلوم والتكنولوجيا والنقل البحري
Arab Academy for Science, Technology & Maritime Transport

Smart Village Campus

Mechatronics Engineering

PID Optimization Problem (12th)

Optimum Design (ME554)

Created by:

Adham Eid 20107315
Farida Ahmed Abdelsalam 20107458

Prepared For:

Dr Yasser El-shaer

Table of contents

I. Problem formulation

Project description

The objective of this optimization problem is to work out the best values for a pid controller of a hydraulic position control system. The pid controller will regulate the 3-way promotional valve that controls the main and loading cylinder as shown in Figure 1. A nonlinear model of the 3-way valve has been derived from the actual physical model. This model can be used to predict the response of the physical system when given a step input. Without a controller, the system exhibits overshoot and some oscillation before it settles. A pid controller with its corresponding (kp, ki, kd) values is needed to minimize the overshoot, and to achieve a steady state error less than 5%, a settling time between 1 second and 1.5 seconds, and a rise time greater than 0.5 seconds. Minimizing the overshoot is essential because any overshoot in the response of the system will cause damage to the system. The rise time and settling time are chosen based on the characteristics of the 3-way proportional valve. Also, a low steady state error is needed to improve the repeatability of the system.

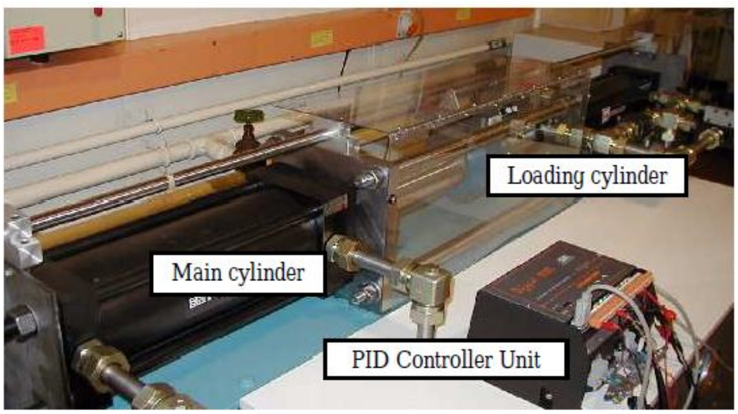


Figure 1

Data and information collection

Equation 1, is the transfer function of the hydraulic system which is from a research paper [1]. Equation 2, is the transfer function of a standard PID controller. MATLAB has been used to calculate all the data that is needed like: Overshoot, rise time, settling time, and steady state error. Figure 2 shows the block diagram of the PID controller and the transfer function.

$$G_p(s) = \frac{7.84}{3s^2 + 5.04s + 7.84} \quad (1)$$

$$PID(s) = K_p + \frac{K_i}{s} + K_d s \quad (2)$$

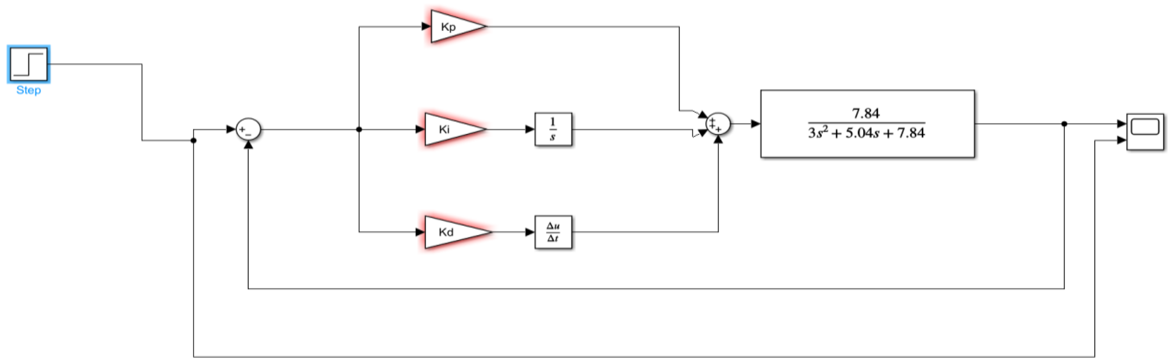


Figure 2

Definition of design variables

The values of the PID are identified as the design variables for this problem.

1. K_p = Proportional gain
2. K_i = Integral gain
3. K_d = Derivative component

Optimization Criterion

The objective is to design a PID controller that minimizes overshoot.

$$f(K_p, K_i, K_d) = Minimize\ (Overshoot)$$

Formulation of constraints

Inequality constraints:

1. Steady-state error $\leq$ 5%
2. Settling time $\geq 1\ Second$
3. Settling time $\leq 1.5\ Seconds$
4. Rise time $\geq 0.5\ Seconds$

II. Solution

Method 1: (Adham)

Genetic Algorithm Method

Advantages and limitations

Genetic algorithm is a search technique that tries to mimic natural selection and evolution. It can be used to solve both constrained and unconstrained problems. Some advantages include: its robustness because it is able to find global minimum or global maximum points. It can also take advantage of parallel processing. This means it can run more effectively on machines with many CPU cores. Genetic algorithms are also not affected by starting positions. It is also effective in searching in large solution spaces and lastly it doesn't need any derivative information. Some limitations include: long convergence times, sometimes premature convergence occurs because there isn't a lot of diversity in evolution, so it gets stuck, and it won't work well with complex fitness functions since it needs to calculate it for every population.

Justification

With all these limitations in mind, Genetic algorithm is still a suitable search method for this problem because, the fitness function is not complex, convergence times for this problem is not high, and lastly the search area is not large.

Settings

The following settings were used for the genetic algorithm: A population size of 800, Max generations of 300, Elite count of 8, Crossover fraction of 0.8, and mutation and parallel processing were turned on. The global optimization toolbox needed to use the genetic algorithm function. The parallel computing toolbox was needed to use the parallel feature in the ga function.

Results

The genetic algorithm only needed 4 generations to find a suitable solution. Info about the generations is shown in the Table 1. The algorithm took **269.33** seconds (With parallel computing enabled) and it took **1019.40** seconds (With parallel computing disabled) to find the solution.

Table 1

Generation	Func-count	Best f(x)	Max Constraint	Stall Generations
1	42015	0.024591	0	0
2	83230	0.0244363	0	0
3	124445	0.0237628	0	0
4	165645	0	0.001	0

Figure 3 shows the convergence graph.

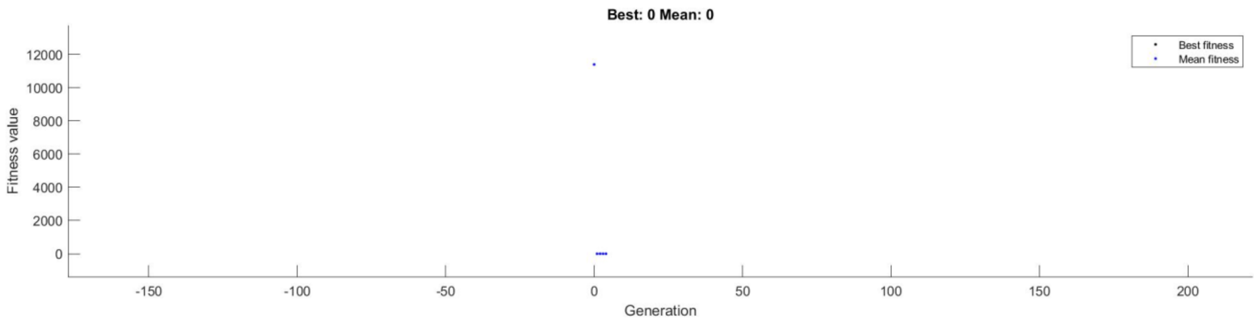


Figure 3

The solution from the genetic algorithm is:

- Kp = 2.4956
- Ki = 3.8180
- Kd = 1.4552

Table 2 shows the Rise time, settling time, Steady state error, and overshoot of the transfer function with and without the PID controller.

Table 2

Parameters	Open-Loop (no PID)	Closed-Loop (with PID)
Rise time	1.0383 s	0.581 s
Settling time	4.8291 s	0.9990 s
Overshoot	14.79%	0.00%
Steady-State error	0.50%	0.00%

Figure 4 shows a graph of the response of the transfer function to a step input with and without the PID controller

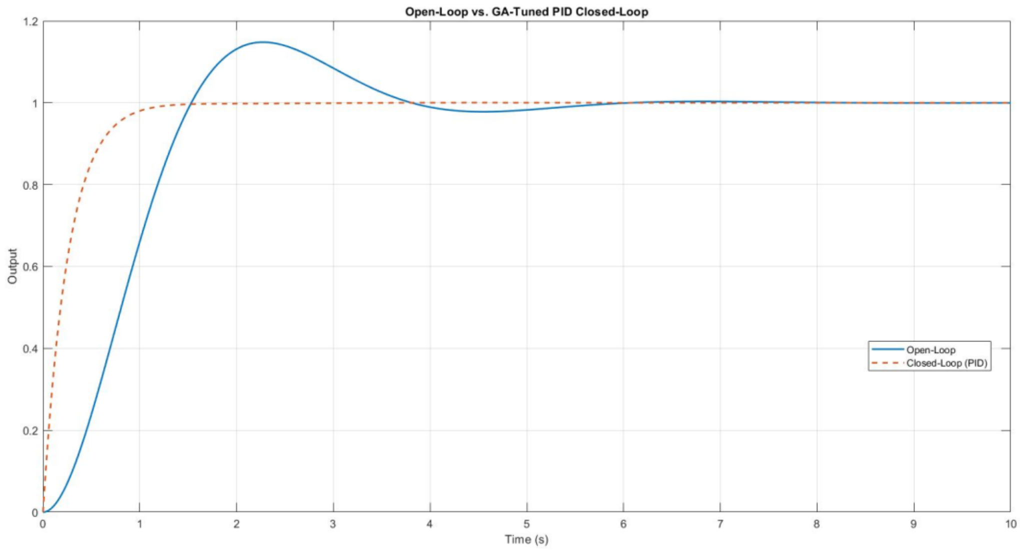


Figure 4

Analysis

The transfer function without any controller experiences nearly 15% overshoot and has very poor rise and settling time. After using the genetic algorithm to find the optimal values of the PID controller, it found that (Kp = 2.4956, Ki = 3.8180, Kd = 1.4552) these pid values minimizes the overshoot to 0 % and satisfies the rise time (with a rise time of 0.581 s), settling time (with a settling time of 0.999 s), and steady state error constraints (with a steady state error of 0%). It took 269.33 seconds to find the solution.

Genetic Algorithm + local search hybrid Method

For this problem, the genetic algorithm alone, without any modifications found a suitable result that minimizes overshoot completely, and satisfies all the constraints. A hybrid method was still tested to check if there are any differences.

For this hybrid method, Genetic algorithm is initially used, when it finds its suitable solution, the point is then given to a function called fmincon which stands for "Function minimization with constraints". This function starts from where the genetic algorithm stops and tries to find a point that is more suitable.

It generated the following PID parameters:

- Kp = 2.2386
- Ki = 3.2746

- Kd = 1.1735

Table 3 shows the results:

Table 3

Parameters	Open-Loop (no PID)	Closed-Loop (with PID) (GA)	Closed-Loop (with PID) (Hybrid)
Rise time	1.0383 s	0.6309 s	0.6310 s
Settling time	4.8291 s	0.9990s	1.0000 s
Overshoot	14.79%	0.00%	0.00%
Steady-State error	0.50%	0.00%	0.00%

Table 4

PID values	GA	GA+LS
Kp	2.4956	2.2386
Ki	3.818	3.2746
Kd	1.4552	1.1735

Analysis

Results show very slight improvement in the settling time, but the difference is negligible. Table 4 shows the PID values for the genetic algorithm methods and genetic algorithm + local search method

MATLAB code for (Genetic algorithm method):

```
%% PID Tuning with GA (Minimize Overshoot)

% 1. Define plant transfer function
num = 7.84;
den = [3, 5.04, 7.84];
Gp = tf(num, den);

% 2. GA options & bounds
nVars   = 3;                  % [Kp, Ki, Kd]
lb      = [0,    0,    0];    % lower bounds
ub      = [100, 100, 10];     % upper bounds

options = optimoptions('ga', ...
    'Display', 'iter', ...
    'PopulationSize', 800, ...
    'MaxGenerations', 300, ...
    'EliteCount', 8, ...
    'CrossoverFraction', 0.8, ...
    'MutationFcn', {@mutationadaptfeasible}, ...
    'UseParallel', true, ...
    'PlotFcn', {@gaplotbestf, @gaplotscorediversity});

% 3. Run GA to tune PID
```

```
tic;    % start timer
[x_opt, ~] = ga( ...
    @(x) pidObjective(x, Gp), ...
    nVars, [], [], [], [], lb, ub, ...
    @(x) pidConstraints(x, Gp), ...
    options);
elapsedGA = toc;  % stop timer and store elapsed time

fprintf('GA optimization took %.2f seconds.\n\n', elapsedGA);

Kp_opt = x_opt(1);
Ki_opt = x_opt(2);
Kd_opt = x_opt(3);

fprintf('\nOptimized PID gains:\n');
fprintf('  Kp = %.4f\n', Kp_opt);
fprintf('  Ki = %.4f\n', Ki_opt);
fprintf('  Kd = %.4f\n\n', Kd_opt);

% 4. Prepare responses for plotting
t          = 0:0.01:10;
[y_ol, ~] = step(Gp, t);
C_opt      = pid(Kp_opt, Ki_opt, Kd_opt);
sys_cl     = feedback(C_opt*Gp,1);
[y_cl, ~] = step(sys_cl, t);
ess        = abs(1 - y_cl(end));  % Steady-state error

% 5. Plot comparison
figure;
plot(t, y_ol,  'LineWidth',1.5); hold on;
plot(t, y_cl, '--','LineWidth',1.5); hold off;
grid on; xlabel('Time (s)'); ylabel('Output');
title('Open-Loop vs. GA-Tuned PID Closed-Loop');
legend('Open-Loop','Closed-Loop (PID)','Location','Best');

% 6. Print performance metrics
info_ol = stepinfo(Gp);
info_cl = stepinfo(sys_cl);

Kp_ol = dcgain(Gp);  % evaluates Gp(s) as s ? 0
ess_ol = 1 / (1 + Kp_ol);

fprintf('--- Open-Loop (no PID) ---\n');
fprintf(' Rise Time:     %.4f s\n', info_ol.RiseTime);
fprintf(' Settling Time: %.4f s\n', info_ol.SettlingTime);
fprintf(' Overshoot:     %.2f%%\n', info_ol.Overshoot);
fprintf(' Steady-State Error (ess): %.2f%%\n\n', ess_ol);

fprintf('--- Closed-Loop (with PID) ---\n');
fprintf(' Rise Time:     %.4f s\n', info_cl.RiseTime);
fprintf(' Settling Time: %.4f s\n', info_cl.SettlingTime);
fprintf(' Overshoot:     %.2f%%\n', info_cl.Overshoot);
fprintf(' Steady-State Error (ess): %.2f%%\n', ess);


%% --- Objective function: minimize overshoot (penalize infeasible) ---
function cost = pidObjective(x, Gp)
    C   = pid(x(1), x(2), x(3));
    sys = feedback(C*Gp,1);
    info = stepinfo(sys);
    %Stability check
    if isempty(info) || isnan(info.Overshoot) || isinf(info.Overshoot)
        cost = 1e6;
    else
        cost = info.Overshoot;
    end
end

%% --- Constraints:
function [c, ceq] = pidConstraints(x, Gp)
    C   = pid(x(1), x(2), x(3));
    sys = feedback(C*Gp,1);
    info = stepinfo(sys);
```

```
    [y, ~] = step(sys, 0:0.01:10);
    ess = abs(1 - y(end));  % steady-state error

    c1 = ess - 0.05;               % ess ? 0.05
    c2 = info.SettlingTime - 1.5;  % Ts ? 1.5
    c3 = 1 - info.SettlingTime;    % Ts ? 1
    c4 = 0.5 - info.RiseTime;      % Rt ? 0.5

    %Check if any response returns incorrent data
    if isempty(info) || any(isnan([info.SettlingTime, info.RiseTime]))
        c = [1; 1; 1; 1];
    else
        c = [c1; c2; c3; c4];
    end
    ceq = [];
end
```

Method 2: (Farida)

Particle swarm Optimization

Advantages and limitations

Particle Swarm Optimization (PSO) is a stochastic optimization technique developed by James Kennedy and Russell Eberhart in 1995. It involves a group of particles moving through a solution space to find the optimum of an objective function. PSO balances exploration and exploitation, making it useful in engineering, machine learning, and control systems for solving nonlinear, multidimensional optimization problems. This method is an evolutionary algorithm that is simple to implement, with fewer parameters to adjust compared to other algorithms like Genetic Algorithms. It requires only a few control parameters, often converges faster than other optimization algorithms, and can be applied to a wide range of optimization problems. It also has parallelization capability, improving computation speed for large problems.
However, such method can get stuck at local minima, especially in complex or multimodal search spaces. Also, Performance of PSO depends on proper tuning of inertia weight and learning factors, with poor settings leading to instability or poor results. Scalability issues may degrade performance as problem dimensionality increases. Lack of diversity can reduce exploration ability and limit the algorithm's global optimum. pso does not guarantee convergence, especially without escape mechanisms.

Justification

Nevertheless, using PSO is quite suitable for solving this optimization problem as it can handle nonlinear and complex systems, eliminating the need for mathematical linearization or model derivatives. It is derivative-free, making it ideal for black-box system optimization. pso can be easily customized to handle multi-objective optimization, and it has global optimization capability, allowing for better performance in noisy systems. pso is also effective for systems with time delays due to fluid dynamics, as it doesn't rely on system time constants or frequency responses. Overall,pso offers a flexible and efficient solution for optimizing hydraulic systems.

Settings

The following settings were used for the PSO: Number of particles are 30, Max iterations of 100, Inertia weight that decays by 0.99 per iteration ,Cognitive coefficient (self-confidence) of 1.5 and Social coefficient (swarm confidence) of 1.5 .

Matlab code

```
% Standard PSO-PID Optimization for DC Motor Control
% Plant: Gp(s) = 0.067 / (0.00113s^2 + 0.0078854s + 0.0171)
clc;
clear;
close all;

%% DC Motor Transfer Function
num = 0.067;
den = [0.00113, 0.0078854, 0.0171];
motor_tf = tf(num, den);

%% PSO Parameters
n_particles = 30;       % Number of particles
max_iter = 100;          % Maximum iterations
w = 0.9;                % Inertia weight
c1 = 1.5;               % Cognitive coefficient
c2 = 1.5;               % Social coefficient
Kp_range = [0, 50];     % Range for Kp
Ki_range = [0, 50];     % Range for Ki
Kd_range = [0, 10];     % Range for Kd

%% Initialize particles
particles = struct('position', [], 'velocity', [], 'cost', [], 'best_position',
[], 'best_cost', []);
global_best.cost = inf;
global_best.position = [];

for i = 1:n_particles
    % Random initialization within bounds
    Kp = Kp_range(1) + (Kp_range(2)-Kp_range(1))*rand();
    Ki = Ki_range(1) + (Ki_range(2)-Ki_range(1))*rand();
    Kd = Kd_range(1) + (Kd_range(2)-Kd_range(1))*rand();

    particles(i).position = [Kp, Ki, Kd];
    particles(i).velocity = zeros(1, 3);

    % Evaluate initial cost
```

```
    [cost, ~, ~, steady_state_error] = evaluate_pid([Kp, Ki, Kd], motor_tf);
    particles(i).cost = cost;
    particles(i).best_position = particles(i).position;
    particles(i).best_cost = cost;

    % Update global best
    if cost < global_best.cost
        global_best.cost = cost;
        global_best.position = particles(i).position;
    end
end

%% PSO Optimization Loop
for iter = 1:max_iter
    for i = 1:n_particles
        % Update velocity
        r1 = rand(1,3);
        r2 = rand(1,3);

        cognitive = c1 * r1 .* (particles(i).best_position - 
particles(i).position);
        social = c2 * r2 .* (global_best.position - particles(i).position);

        particles(i).velocity = w * particles(i).velocity + cognitive + social;

        % Update position
        particles(i).position = particles(i).position + particles(i).velocity;

        % Apply bounds
        particles(i).position(1) = max(Kp_range(1), min(Kp_range(2), 
particles(i).position(1)));
        particles(i).position(2) = max(Ki_range(1), min(Ki_range(2), 
particles(i).position(2)));
        particles(i).position(3) = max(Kd_range(1), min(Kd_range(2), 
particles(i).position(3)));

        % Evaluate new position
        [cost, step_info, ~, steady_state_error] = 
evaluate_pid(particles(i).position, motor_tf);

        % Update personal best
        if cost < particles(i).best_cost
            particles(i).best_cost = cost;
            particles(i).best_position = particles(i).position;

            % Update global best
            if cost < global_best.cost
                global_best.cost = cost;
                global_best.position = particles(i).position;
```

```
                best_step_info = step_info;
                best_steady_state_error = steady_state_error;
            end
        end
    end

    % Display progress
    fprintf('Iteration  %d:  Global  Best  score  =  %.4f,  Kp=%.3f,  Ki=%.3f,
Kd=%.3f\n', ...
            iter,         global_best.cost,          global_best.position(1),
global_best.position(2), global_best.position(3));

    % Optional: Adaptive inertia weight
    w = w * 0.99;
end

%% Display Results
fprintf('\nOptimization Results:\n');
fprintf('Best PID Parameters: Kp = %.3f, Ki = %.3f, Kd = %.3f\n', ...
        global_best.position(1),                        global_best.position(2),
global_best.position(3));

% Get open-loop response metrics
[y_open, t_open] = step(motor_tf);
open_info = stepinfo(y_open, t_open);
ss_error_open = abs(1 - y_open(end)) * 100;

fprintf('\n--- Scaled Open-Loop (no PID) ---\n');
fprintf('Rise Time:    %.4f s\n', open_info.RiseTime);
fprintf('Settling Time:%.4f s\n', open_info.SettlingTime);
fprintf('Overshoot:    %.2f%%\n', open_info.Overshoot);
fprintf('Steady-State Error: %.2f%%\n', ss_error_open);

fprintf('\n--- Closed-Loop (with PID) ---\n');
fprintf('Rise Time:    %.4f s\n', best_step_info.RiseTime);
fprintf('Settling Time:%.4f s\n', best_step_info.SettlingTime);
fprintf('Overshoot:    %.2f%%\n', best_step_info.Overshoot);
fprintf('Steady-State Error: %.2f%%\n', best_steady_state_error);

% ... (previous code remains the same until the comparative plot section)

%% Comparative Plot: Open-Loop vs PID-Controlled
[~, ~, sys_cl] = evaluate_pid(global_best.position, motor_tf);

% Calculate DC gain of open-loop system
DC_gain = dcgain(motor_tf);

% Create scaled open-loop system that settles at 1
scaled_open_loop = motor_tf/DC_gain;
```

```
figure;
set(gcf, 'Position', [100, 100, 800, 500]);

% Simulate responses with consistent time vector
t = 0:0.001:2;
[y_open, t_open] = step(scaled_open_loop, t);  % Using scaled open-loop
[y_pid, t_pid] = step(sys_cl, t);

% Plot both responses
plot(t_open, y_open, 'b', 'LineWidth', 2);
hold on;
plot(t_pid, y_pid, 'r', 'LineWidth', 2);
plot([0 t(end)], [1 1], 'k--', 'LineWidth', 1); % Reference line
hold off;

% Add labels and title
title('System Response Comparison: Open-Loop vs PSO-Optimized PID', 'FontSize',
14);
xlabel('Time (seconds)', 'FontSize', 12);
ylabel('Amplitude', 'FontSize', 12);
legend('Scaled Open-Loop Response', 'PSO-Optimized PID', 'Reference',
'Location', 'southeast');
grid on;

% Set consistent axes
xlim([0 2]);
ylim([0 max(1.2, 1.2*max([y_open; y_pid]))]);

% Recalculate open-loop metrics with scaled system
[y_open_scaled, t_open_scaled] = step(scaled_open_loop);
open_info_scaled = stepinfo(y_open_scaled, t_open_scaled);
ss_error_open_scaled = abs(1 - y_open_scaled(end)) * 100;

%% Cost Function with Penalty Method
function [cost, step_info, sys_cl, steady_state_error] =
evaluate_pid(pid_params, motor_tf)
    Kp = pid_params(1);
    Ki = pid_params(2);
    Kd = pid_params(3);

    % Create PID controller
    pid_tf = pid(Kp, Ki, Kd);

    % Closed-loop system
    sys_cl = feedback(pid_tf * motor_tf, 1);

    % Get step response information
    try
```

```
    [y, t] = step(sys_cl);
    step_info = stepinfo(sys_cl);

    % Calculate steady-state error properly
    steady_state_value = y(end);
    steady_state_error = abs(1 - steady_state_value) * 100; % in percentage

    % Extract performance metrics
    overshoot = step_info.Overshoot;
    if isempty(overshoot)
        overshoot = 0;
    end

    rise_time = step_info.RiseTime;
    settling_time = step_info.SettlingTime;

    % Objective: Minimize overshoot
    base_cost = overshoot;

    % Constraints with penalty functions
    penalty = 0;

    % Steady-state error < 5%
    if steady_state_error >= 5
        penalty = penalty + 1000 + (steady_state_error - 5)^2;
    end

    % Settling time between 1 and 1.5 sec
    if settling_time < 1
        penalty = penalty + 1000 + (1 - settling_time)^2;
    elseif settling_time > 1.5
        penalty = penalty + 1000 + (settling_time - 1.5)^2;
    end

    % Rise time at least 0.5 sec
    if rise_time < 0.5
        penalty = penalty + 1000 + (0.5 - rise_time)^2;
    end

    % Total cost
    cost = base_cost + penalty;

catch
    % If simulation fails (unstable system), assign high cost
    cost = 1e6;
    step_info.Overshoot = 100;
    step_info.RiseTime = 0;
    step_info.SettlingTime = 100;
    steady_state_error = 100;
```

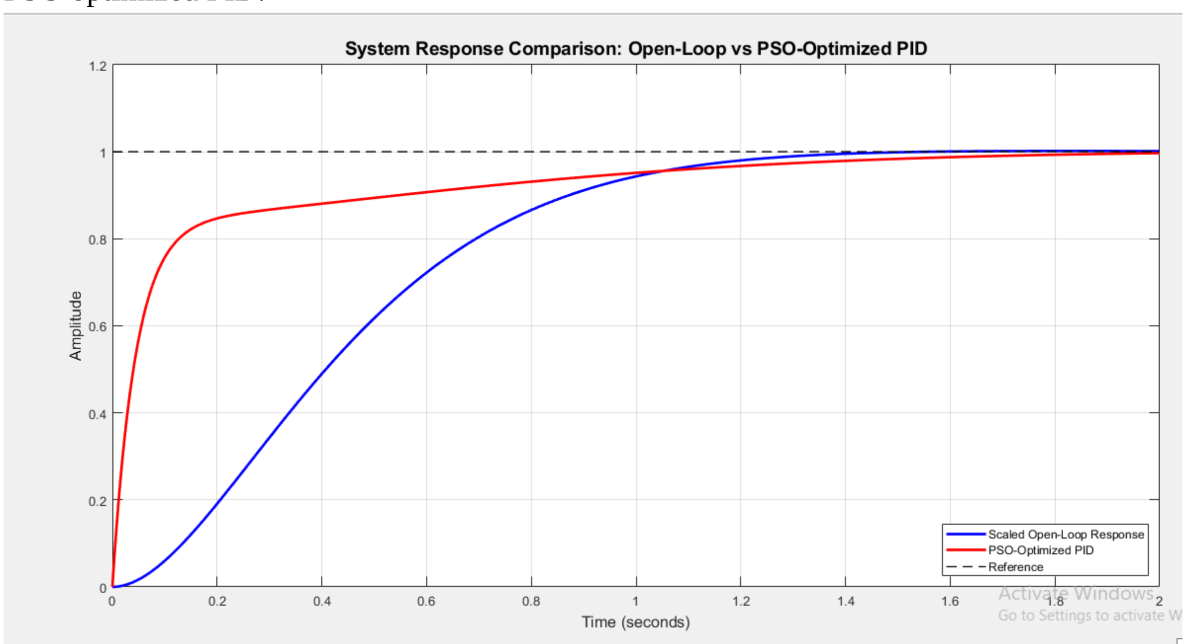
```
        sys_cl = tf(1,[1 1]); % Dummy stable system
    end
end
```

Results

The output for the PSO optimization was as follows
PID Gains: Kp = 1.124, Ki = 1.572, Kd = 0.318
The table below shows the difference between scaled open loop , standard pso and hybrid pso results
The figure below shows the graph of the response of the transfer function without PID ,with PSO and with PSO-SA PID.

parameters	Open loop (no PID)	Standard PSO
Rise time:	1.0407s	0.5404s
Settling time:	4.9542s	1.4189s
Overshoot:	14.58%	0.00%
Steady state error (ess):	0.18%	0.34%

The figure below shows the graph of the response of the transfer function without PID and with PSO optimized PID.



Analysis

The transfer function without any controller experiences 14.58% overshoot and has very poor rise and settling time. After using the standard PSO to find the optimal values of the PID controller, it found that (Kp = 1.124, Ki = 1.572, Kd = 0.318) these pid values minimizes the overshoot to 0 % , minimizes the rise time and settling time and satisfies steady state error constraint.

Particle swarm Optimization + Simulated Annealing Method

Advantages and limitations

PSO with Simulated Annealing (PSO-SA) is a hybrid optimization algorithm that combines Particle Swarm Optimization's global search capability with Simulated Annealing's local refinement ability. It guides particles towards promising regions using personal and global best experiences, while Simulated Annealing introduces controlled random jumps to help particles escape local optima. PSO-SA offers several advantages, including improved global search capability, balanced exploration and exploitation, more robust convergence, effectiveness for nonlinear and noisy systems, and adaptive behavior. It helps PSO escape local minima, allowing for better convergence to the global optimum. The hybrid method reduces the risk of premature convergence, making it suitable for real-world control systems like hydraulic systems. Additionally, SA introduces temperature-dependent randomness, allowing occasional uphill moves to avoid traps in suboptimal regions.

However, the hybrid optimization method (SA) has several drawbacks, including increased computational cost, parameter tuning complexity, potential slow convergence, and implementation complexity. SA adds extra steps per iteration, making it slower than basic PSO, especially for large-scale problems. It also requires careful tuning of both parameters, which can slow down the optimization process if not properly tuned.

Justification

But using this hybrid method will improve the results of PSO optimization as PSO-SA is a robust solution for real-world hydraulic systems, overcoming sensor noise, load fluctuations, and external disturbances due to its non-gradient information requirement and stochastic acceptance of suboptimal solutions. PSO may struggle with suboptimal solutions due to swarm stagnation, but SA's thermal escape mechanism allows occasional uphill moves, enabling better-performing PID parameters.

Settings

The following settings were used for the simulated annealing: Cooling rate of 0.95 and the Number of SA iterations per PSO iteration is 5. The setting for the pso is the same as previously stated.

MATLAB code

```matlab
% Hybrid PSO-SA vs Standard PSO PID Optimization Comparison
% Plant: Gp(s) = 7.84 / (3s^2 + 5.04s + 7.84)
clc; clear; close all;

%% DC Motor Transfer Function
num = 7.84;
den = [3, 5.04, 7.84];
motor_tf = tf(num, den);

%% Common Parameters
n_particles = 30;       % Number of particles
max_iter = 100;          % Maximum iterations
Kp_range = [0, 50];     % Range for Kp
Ki_range = [0, 50];     % Range for Ki
Kd_range = [0, 10];     % Range for Kd

%% 1. First run Standard PSO Optimization
disp('Running Standard PSO Optimization...');
w = 0.9;                % Inertia weight
c1 = 1.5;               % Cognitive coefficient
c2 = 1.5;               % Social coefficient

% Initialize particles
particles = struct('position', [], 'velocity', [], 'cost', [], 'best_position',
[], 'best_cost', []);
pso_global_best.cost = inf;
pso_global_best.position = [];

for i = 1:n_particles
    Kp = Kp_range(1) + (Kp_range(2)-Kp_range(1))*rand();
    Ki = Ki_range(1) + (Ki_range(2)-Ki_range(1))*rand();
    Kd = Kd_range(1) + (Kd_range(2)-Kd_range(1))*rand();

    particles(i).position = [Kp, Ki, Kd];
    particles(i).velocity = zeros(1, 3);

    [cost, ~, ~, ~] = evaluate_pid([Kp, Ki, Kd], motor_tf);
    particles(i).cost = cost;
    particles(i).best_position = particles(i).position;
    particles(i).best_cost = cost;

    if cost < pso_global_best.cost
        pso_global_best.cost = cost;
        pso_global_best.position = particles(i).position;
    end
end

% PSO Optimization Loop
for iter = 1:max_iter
    for i = 1:n_particles
        r1 = rand(1,3);
        r2 = rand(1,3);
        particles(i).velocity = w * particles(i).velocity + ...
```

```
                                c1   *   r1   .*   (particles(i).best_position   -
particles(i).position) + ...
                                c2   *   r2   .*   (pso_global_best.position   -
particles(i).position);

        particles(i).position = particles(i).position + particles(i).velocity;

        % Apply bounds
        particles(i).position  =  max([Kp_range(1),  Ki_range(1),  Kd_range(1)],
...
                                     min([Kp_range(2),              Ki_range(2),
Kd_range(2)], particles(i).position));

        [cost, ~, ~, ~] = evaluate_pid(particles(i).position, motor_tf);

        if cost < particles(i).best_cost
            particles(i).best_cost = cost;
            particles(i).best_position = particles(i).position;

            if cost < pso_global_best.cost
                pso_global_best.cost = cost;
                pso_global_best.position = particles(i).position;
            end
        end
    end
    w = w * 0.99; % Inertia weight decay
end

% Get final PSO results
[~,        pso_step_info,        pso_sys_cl,        pso_ss_error]          =
evaluate_pid(pso_global_best.position, motor_tf);

%% 2. Then run Hybrid PSO-SA Optimization
disp('Running Hybrid PSO-SA Optimization...');
w = 0.9;                % Reset inertia weight
c1 = 1.5;               % Cognitive coefficient
c2 = 1.5;               % Social coefficient

% SA Parameters
T0 = 100;               % Initial temperature
alpha = 0.95;           % Cooling rate
sa_iter = 5;            % SA iterations per PSO iteration

% Initialize particles (same initialization as PSO)
for i = 1:n_particles
    Kp = Kp_range(1) + (Kp_range(2)-Kp_range(1))*rand();
    Ki = Ki_range(1) + (Ki_range(2)-Ki_range(1))*rand();
    Kd = Kd_range(1) + (Kd_range(2)-Kd_range(1))*rand();

    particles(i).position = [Kp, Ki, Kd];
    particles(i).velocity = zeros(1, 3);

    [cost, ~, ~, ~] = evaluate_pid([Kp, Ki, Kd], motor_tf);
    particles(i).cost = cost;
    particles(i).best_position = particles(i).position;
    particles(i).best_cost = cost;
```

```
    if cost < pso_global_best.cost
        pso_global_best.cost = cost;
        pso_global_best.position = particles(i).position;
    end
end

% Hybrid PSO-SA Optimization Loop
T = T0;  % Initial temperature
for iter = 1:max_iter
    % Standard PSO Update
    for i = 1:n_particles
        r1 = rand(1,3);
        r2 = rand(1,3);
        particles(i).velocity = w * particles(i).velocity + ...
                              c1  *  r1  .*  (particles(i).best_position  -
particles(i).position) + ...
                              c2  *  r2  .*  (pso_global_best.position  -
particles(i).position);

        particles(i).position = particles(i).position + particles(i).velocity;
        particles(i).position  = max([Kp_range(1), Ki_range(1), Kd_range(1)],
...
                               min([Kp_range(2),  Ki_range(2),  Kd_range(2)],
particles(i).position));

        [cost, ~, ~, ~] = evaluate_pid(particles(i).position, motor_tf);

        if cost < particles(i).best_cost
            particles(i).best_cost = cost;
            particles(i).best_position = particles(i).position;

            if cost < pso_global_best.cost
                pso_global_best.cost = cost;
                pso_global_best.position = particles(i).position;
            end
        end
    end

    % Simulated Annealing Local Search
    for k = 1:sa_iter
        for i = 1:n_particles
            scale  =  [(Kp_range(2)-Kp_range(1)),  (Ki_range(2)-Ki_range(1)),
(Kd_range(2)-Kd_range(1))];
neighbor = particles(i).best_position + T/T0 * randn(1,3) .* (scale / 20);
            neighbor = max([Kp_range(1), Ki_range(1), Kd_range(1)], ...
                        min([Kp_range(2),     Ki_range(2),     Kd_range(2)],
neighbor));

            [neighbor_cost, ~, ~, ~] = evaluate_pid(neighbor, motor_tf);

            delta_cost = neighbor_cost - particles(i).best_cost;
            if delta_cost < 0 || rand() < exp(-delta_cost/T)
                particles(i).best_position = neighbor;
                particles(i).best_cost = neighbor_cost;

                if neighbor_cost < pso_global_best.cost
                    pso_global_best.cost = neighbor_cost;
```

```
                    pso_global_best.position = neighbor;
                end
            end
        end
    end

    T = alpha * T; % Cool down temperature
    w = w * 0.99;  % Inertia weight decay
end

% Get final PSO-SA results
[~,      pso_sa_step_info,      pso_sa_sys_cl,      pso_sa_ss_error]      =
evaluate_pid(pso_global_best.position, motor_tf);

%% Display Comparison Results
% Scale open-loop to settle at 1
DC_gain = dcgain(motor_tf);
scaled_open_loop = motor_tf/DC_gain;

% Get open-loop response metrics
[y_open, t_open] = step(scaled_open_loop);
open_info = stepinfo(y_open, t_open);
ss_error_open = abs(1 - y_open(end)) * 100;

% Display all results
fprintf('\n=== Complete Performance Comparison ===\n');

fprintf('\n--- Scaled Open-Loop (no PID) ---\n');
fprintf('Rise Time:    %.4f s\n', open_info.RiseTime);
fprintf('Settling Time:%.4f s\n', open_info.SettlingTime);
fprintf('Overshoot:    %.2f%%\n', open_info.Overshoot);
fprintf('Steady-State Error: %.2f%%\n', ss_error_open);

fprintf('\n--- Standard PSO Results ---\n');
fprintf('PID   Gains:   Kp   =   %.4f,   Ki   =   %.4f,   Kd   =   %.4f\n',
pso_global_best.position);
fprintf('Rise Time:    %.4f s\n', pso_step_info.RiseTime);
fprintf('Settling Time:%.4f s\n', pso_step_info.SettlingTime);
fprintf('Overshoot:    %.2f%%\n', pso_step_info.Overshoot);
fprintf('Steady-State Error: %.2f%%\n', pso_ss_error);


fprintf('\n--- Hybrid PSO-SA Results ---\n');
fprintf('PID   Gains:   Kp   =   %.4f,   Ki   =   %.4f,   Kd   =   %.4f\n',
pso_global_best.position);
fprintf('Rise Time:    %.4f s\n', pso_sa_step_info.RiseTime);
fprintf('Settling Time:%.4f s\n', pso_sa_step_info.SettlingTime);
fprintf('Overshoot:    %.2f%%\n', pso_sa_step_info.Overshoot);
fprintf('Steady-State Error: %.2f%%\n', pso_sa_ss_error);


%% Triple Comparison Plot
figure;
set(gcf, 'Position', [100, 100, 1000, 700]);

% Simulate all responses with consistent time vector
t = 0:0.01:10;
```

```
[y_open_scaled, ~] = step(scaled_open_loop, t);
[y_pso, ~] = step(pso_sys_cl, t);
[y_pso_sa, ~] = step(pso_sa_sys_cl, t);

% Plot all responses
plot(t, y_open_scaled, 'Color', [0 0.5 0], 'LineWidth', 2); % Dark green for
open-loop
hold on;
plot(t, y_pso, 'b', 'LineWidth', 2); % Blue for PSO
plot(t, y_pso_sa, 'r', 'LineWidth', 2); % Red for PSO-SA
plot([0 t(end)], [1 1], 'k--', 'LineWidth', 1); % Reference line
hold off;

% Format plot
title('System Response Comparison: Open-Loop vs PSO vs PSO-SA PID', 'FontSize',
16);
xlabel('Time (seconds)', 'FontSize', 14);
ylabel('Amplitude', 'FontSize', 14);
legend('Scaled  Open-Loop',  'Standard  PSO  PID',  'Hybrid  PSO-SA  PID',
'Reference', 'Location', 'southeast');
grid on;
xlim([0 10]);
ylim([0 1.2]);
%% Cost Function with Penalty Method
function     [cost,     step_info,     sys_cl,     steady_state_error]     =
evaluate_pid(pid_params, motor_tf)
    Kp = pid_params(1);
    Ki = pid_params(2);
    Kd = pid_params(3);
    t = 0:0.01:10;

    pid_tf = pid(Kp, Ki, Kd);
    sys_cl = feedback(pid_tf * motor_tf, 1);

    try
        [y, t] = step(sys_cl);
        step_info = stepinfo(sys_cl);
        steady_state_value = y(end);
        steady_state_error = abs(1 - steady_state_value) * 100;

        overshoot = step_info.Overshoot;
        if isempty(overshoot), overshoot = 0; end

        rise_time = step_info.RiseTime;
        settling_time = step_info.SettlingTime;

        base_cost = 0.5 * overshoot + 0.8 * settling_time + 0.7 * rise_time;
        penalty = 0;

        if steady_state_error >= 5
            penalty = penalty + 1000 + (steady_state_error - 5)^2;
        end

        if settling_time < 1
            penalty = penalty + 1000 + (1 - settling_time)^2;
        elseif settling_time > 1.5
            penalty = penalty + 1000 + (settling_time - 1.5)^2;
```

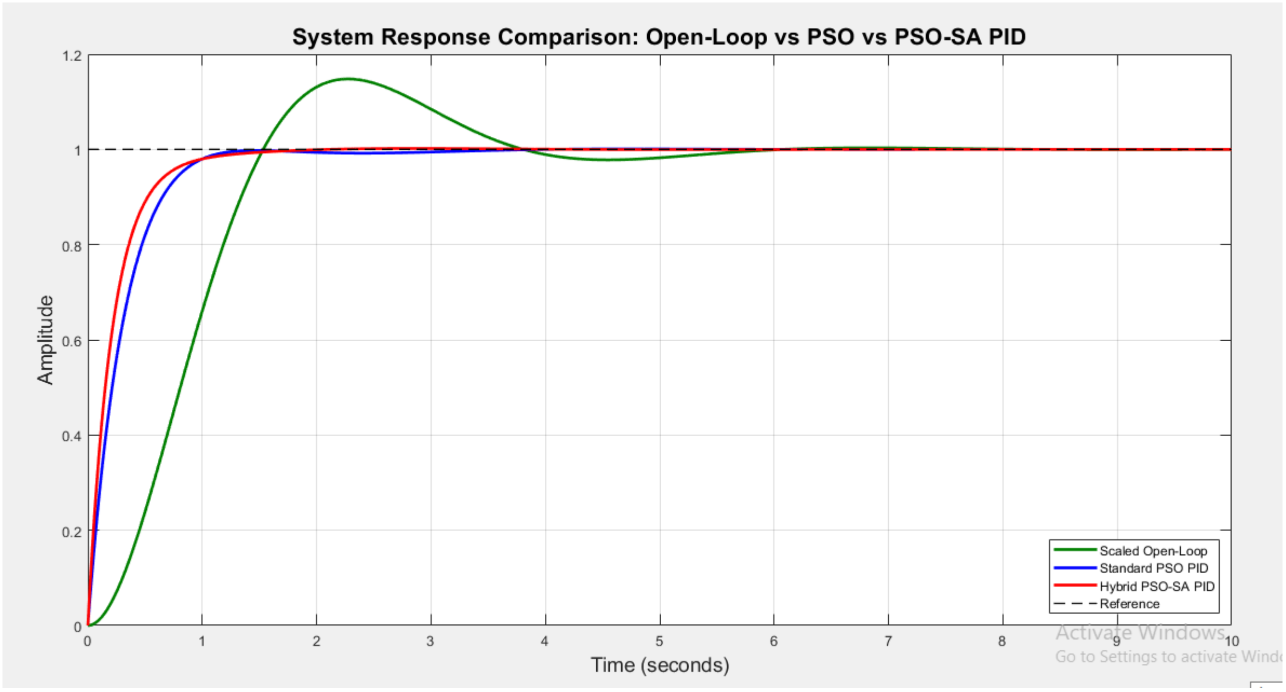
```
        end

        if rise_time < 0.5
            penalty = penalty + 5000 + 1000 * (0.5 - rise_time)^2;

        end

        cost = base_cost + penalty;

    catch
        cost = 1e6;
        step_info.Overshoot = 100;
        step_info.RiseTime = 0;
        step_info.SettlingTime = 100;
        steady_state_error = 100;
        sys_cl = tf(1,[1 1]);
    end
end
```

Results

The output for the PSO-SA optimization was as follows
PID Gains: Kp = 2.8309, Ki = 4.3907, Kd = 1.7615
The table below shows the difference between scaled open loop , standard pso and hybrid pso results

parameters	Open loop (no PID)	Standard PSO	Hybrid PSO-SA
Rise time:	1.0407s	0.6190s	0.5003s
Settling time:	4.9542s	1.0000s	1.0006s
Overshoot:	14.58%	0.00%	0.00%
Steady state error (ess):	0.18%	0.25%	0.34%

The figure below shows the graph of the response of the transfer function without PID ,with PSO and with PSO-SA PID.



Analysis

After using the hybrid PSO to find the optimal values of the PID controller, it found that (Kp = 2.8309, Ki = 4.3907, Kd = 1.7615) these pid values minimizes the overshoot to 0 % and decreases the rise time even more than the standard pso and satisfies the settling time and steady state error constraints.

Method comparison

Table 5

Parameters	Open-Loop (no PID)	Closed-Loop (with PID) (GA)	Closed-Loop (with PID) (GA+LS)	Closed-Loop (with PID) (PSO)	Closed-Loop (with PID) (PSO+SA)
Rise time	1.0383 s	0.6309 s	0.6310 s	0.5404 s	0.5003 s
Settling time	4.8291 s	0.9990s	1.0000 s	1.4189 s	1.0006 s
Overshoot	14.79%	0.00%	0.00%	0.00%	0.00%
Steady-State error	0.50%	0.00%	0.00%	0.34%	0.34%

Table 5 shows a comparison between all the methods tested. All methods were able to find a suitable solution but the genetic algorithm methods were able to completely minimize overshoot while, PSO methods still had 0.34% overshoot. It is still a very low amount of overshoot but genetic algorithm methods were still able to completely get rid of it.

Table 6

PID values	GA	GA+LS	PSO	PSO+SA
Kp	2.4956	2.2386	1.124	2.8309
Ki	3.818	3.2746	1.572	4.3907
Kd	1.4552	1.1735	0.318	1.7615

Table 6 shows the PID values of all the methods used.

References

[1] *Optimal-tuning PID controller design for hydraulic position control systems*. (2001, September 1). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/document/7076000