PID Gain Optimization for an Internal Combustion Engine at Idle

Final Project for Optimization Theory & Practice

Mark Ditsworth & Carlos Caicedo-Narvaez

December 1, 2017

Contents

1. Overview **1**

2. Particle Swarm Optimization Intuition **2**

3. Particle Swarm Optimization Algorithm **3**

4. Results **4**
4.1. Primary Optimization 4
4.2. Secondary Optimization 5

5. Conclusions **5**

Appendix A. MATLAB Code: PSO **i**

List of Figures

1. Simulink model of an internal combustion engine. 1
2. Open-loop response. 2
3. Closed-loop response with $K_p = 0.5$, $K_i = 0.001$, $K_d = 5$. 2
4. Engine speed the idling engine with each set of PID gains. 4
5. Particle positions (K_p and K_i only for viewing simplicity) during the primary optimization routine. 5
6. Cost function during the primary optimization routine. 6
7. Particle positions (K_p and K_i only for viewing simplicity) during the secondary optimization routine. 7
8. Cost function during the secondary optimization routine. 7
9. PSO Code (1) i
10. PSO Code (2) ii
11. PSO Code (3) iii
12. PSO Code (4) iv

List of Tables

1. PID gain values and associated costs. 6

1. Overview

The most common method for regulating the output of a controllable system is a PID controller. With this method, the output error, the integration of the error, and the derivative of the error are summed together and fed back as a control value. The weight each component carries into the control value is defined by the gain of each measurement (proportional, integral, and derivative). Only a small subset of gains will be able to effectively control the output of the system, and it is usually up to the designer to manually tune the gains to achieve an ideal output response. This project implements an optimization routine in place of manual tuning to select the ideal proportional, integral, and derivative gains (K_p, K_i, and K_d, respectively) for idle speed control of an internal combustion engine using the Simulink model shown in Figure 1.

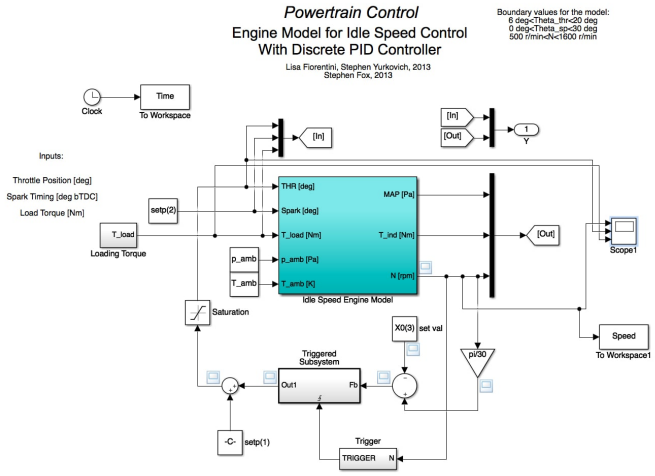

Figure 1: Simulink model of an internal combustion engine.

Without a PID controller (or any controller for that matter) the engine will stall, as shown in the open-loop response in Figure 2. By manually tuning the PID gains using common heuristics, the controller can regulate the engine speed, as depicted in Figure 3. The hand-tuned gains are not always optimal though. Optimal gains are determined by the minimization a cost function that relates to the total error observed in the simulation. The cost function used in this project is the summed squared error, in order to focus on the reduction of overshoot and undershoot. There are numerous optimization algorithms available to attempt the optimization, but the highly non-linear nature of the internal combustion engine model incentivizes the use of global search algorithms. The global search algorithm employed in this project is the Particle Swarm Optimization (PSO) algorithm. Sections 2 and 3 introduce the PSO algorithm. The results of the optimization are discussed in Section 4, and Section 5 summarizes the project's findings.

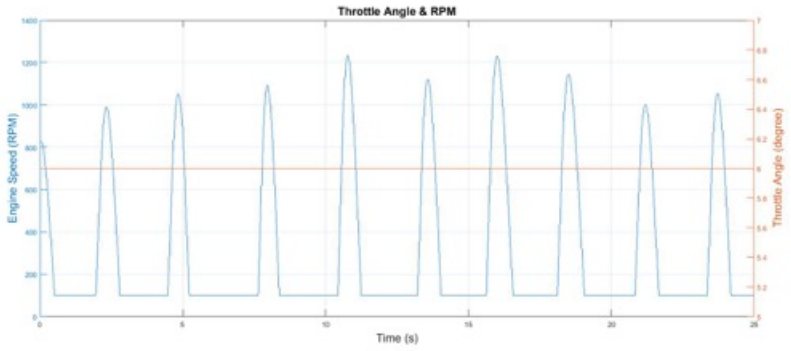

Figure 2: Open-loop response.

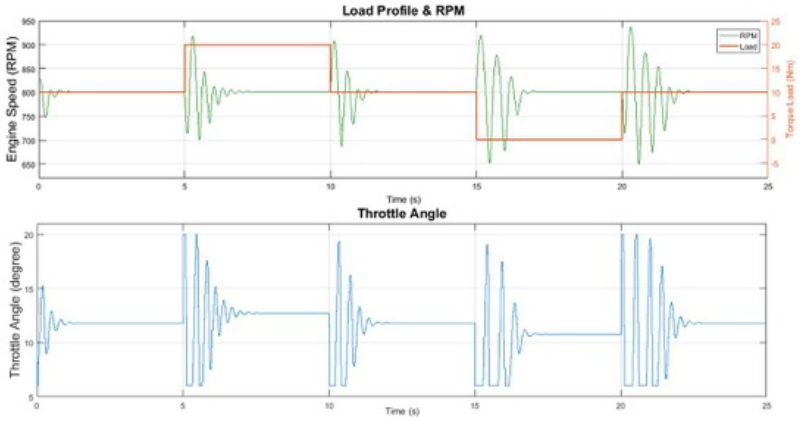

Figure 3: Closed-loop response with $K_p = 0.5$, $K_i = 0.001$, $K_d = 5$.

2. Particle Swarm Optimization Intuition

Particle Swarm Optimization (PSO) is a global search algorithm that uses numerous "particles" and their respective costs to find the optimal policy. In this project's case, the "particles" are points in 3-D space corresponding to the K_p, K_i, and K_d values. The swarm of particles interact with each other to drive the swarm to the best location in the defined space, thus resulting in the optimal policy.

As PSO algorithm iterates, each particle keeps track of its own best-so-far position as well as the swarm's best-so-far position, also called the "global-best" position. The particle updates by adding a velocity vector to the particle. This velocity vector has an element of randomness to it, to allow the particle to explore new values. The closer the particle is to it's best-so-far position and/or the global-best position, however, the less randomness is introduced to the velocity vector; the velocity will more closely resemble the velocity that brought the particle to this position.

How much the particles learn from their own history versus the swarm's history is governed by hyperparameters c_1 and c_2, known as the "cognitive" and "social" components, respectively. If c_1 is larger than c_2, the movement of each particle is controlled

more by it's best-so-far position. Comparatively, if c_2 is larger than c_1, the movement of each particle is controlled more by the global-best position.

Because of the randomness of particle movement, it is common for the global-best to remain unchanged for several iterations before a new global-best position to be discovered. Due to this behavior, the stopping criteria for PSO should not be a tolerance in the change in cost. PSO should stop once the cost reaches a set threshold value, or after a set number of iterations. In this project, we are not seeking a specific cost. Rather we are seeking a desired system response. Thus, the number of iterations was fixed at 10.

3. Particle Swarm Optimization Algorithm

To begin, the initial PID values are used to construct a set of n particles. A random number from an equal distribution of [-1,1] is added to each particle's components to create the initial position of the swarm. The first particle is left unchanged to account for the possibility that the initial PID values are the optimal values. For each particle, an initial velocity vector, $v_0^{(i)}$, is generated randomly from an equal distribution of [-1,1]. The cognitive and social components c_1 and c_2 are defined.

For each particle, the simulation is run and the cost is calculated. The ith cost is assigned to $C_p^{(i)}$, the ith particle's best-so-far cost ,and the ith position (vector of PID gains) is assigned to P_i, the particle's best-so-far position. The global-best cost is assigned to C_g and the corresponding position is assigned to G.

The iteration then begins.

Step 1. Vectors r and s are randomly generated with values coming from an equal distribution of [0,1].

Step 2. The ith particle's velocity, $v_k^{(i)}$, is updated with by Equation 1.

$$v_{k+1}^{(i)} = \omega v_k^{(i)} + c_1 r(P_i - X_k^{(i)}) + c_2 s(G - X_k^{(i)}) \tag{1}$$

where $X_k^{(i)}$ is the ith particle's current location. ω is a scaling term that must be tuned, similar to c_1 and c_2.

Step 3. The ith particle's position is then updated by Equation 2.

$$X_{k+1}^{(i)} = X_k^{(i)} + v_{k+1}^{(i)} \tag{2}$$

Step 4. With new particle positions, the simulation is run for each particle and the corresponding costs are recorded.

Step 5. If the ith particle's new cost is less than, P_i, $C_p^{(i)}$ and P_i are updated accordingly.

Step 6. The minimum of the swarm's costs is compared to the C_g. If the new minimum is better, C_g and G are updated accordingly.

Step 7. If the stopping criteria has been reached, stop. Otherwise, reiterate.

4. Results

4.1. Primary Optimization

The PID gains were first manually tuned to achieve a reasonable, though clearly sub-optimal, response. With K_p, K_i, and K_d set to 0.5, 0.04, and 5, respectively, the response displayed in Figure 4 was achieved.

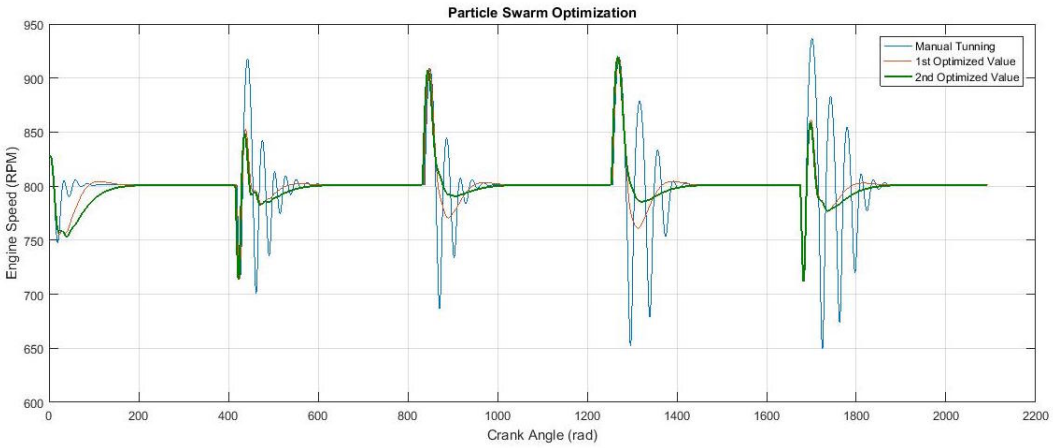

Figure 4: Engine speed the idling engine with each set of PID gains.

These PID gains were served as the initial particle positions for the PSO algorithm as described in the above section. Chong and Zak[1] state that ω should be slightly less than 1, and that c_1 and c_2 should be close to 2. These values, however, resulted in velocity vectors that grew the distribution of the swarm with each iteration. Using smaller values for these hyperparameters was hypothesized to perform better because the smaller velocity vectors would contain the particle distribution and allow for the swarm to converge. Through manual tuning, desirable results were achieved by using $\omega = 0.4$, $c_1 = 0.6$, and $c_2 = 0.3$. The convergence of the swarm using these parameters is illustrated in Figure 5. The global-best cost was tracked during the PSO process, and is displayed in Figure 6. The optimal particle position was discovered by the fourth iteration. However, as described before, termination at the fifth iteration would be unwise due to the randomness of the search. Table 1 depicts the optimized PID gains determined by the PSO algorithm. These optimized PID gains resulted in a 66.3% decrease in the cost.

This optimized response has less overshoot, significantly less oscillation, and a smaller settling time than the hand-tuned response of Figure 4.

[1]E. Chong, S. Zak. An Introduction to Optimization. Fourth Edition. 2013. pp 284.

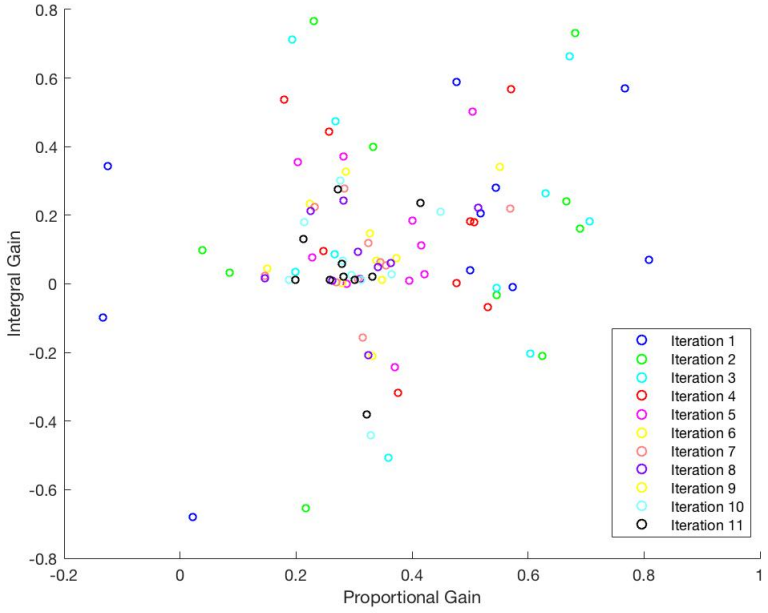

Figure 5: Particle positions (K_p and K_i only for viewing simplicity) during the primary optimization routine.

4.2. Secondary Optimization

The optimized PID gains found by the PSO algorithm in the primary optimization routine were used to generate the initial particle positions for a secondary round of optimization. This secondary PSO round used different hyperparameter values to restrict the search space to a smaller area: $\omega = 0.2$, $c_1 = 0.3$, $c_2 = 0.15$. The initial particle positions were generated in a similar method described in Section 3, but the multipliers came from a uniform distribution of [-0.01,0.01]. The convergence of the particles on the optimal position is shown in Figure 7, and the further decrease in cost is shown in Figure 8.

The secondary optimization routine refined the PID gains to the values given in Table 2. These new PID gains resulted in a 4% improvement in the cost from the primary optimization routine. Figure 4 illustrates the engine response with the refined PID gains. Note the improvement in undershoot near 900 and 1300 radians. The settling angle in the optimized engine response is below 200 radians.

5. Conclusions

The end result of the PSO algorithm resulted in a 67.7% reduction in cost compared to the hand-tuned PID controller. More practically speaking, the optimized PID controller had a worst-case engine perturbation response of just 120 RPM in overshoot and a

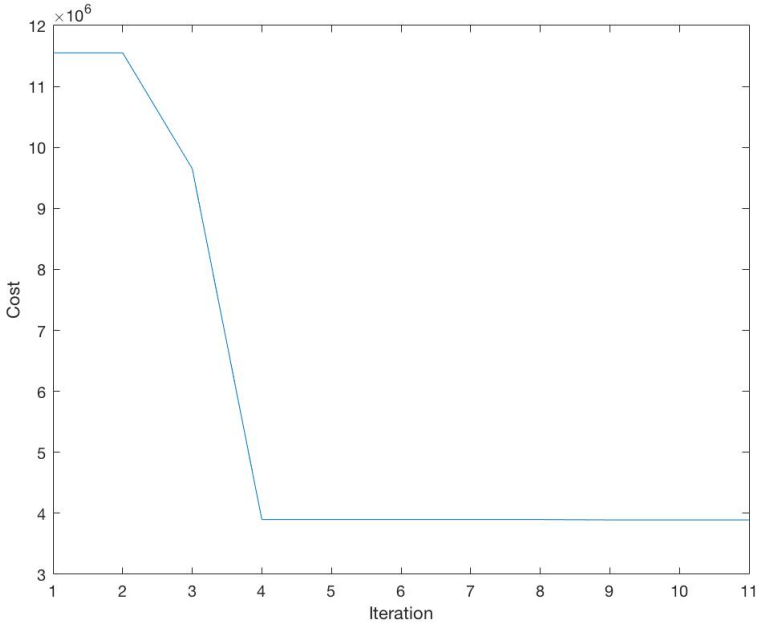

Figure 6: Cost function during the primary optimization routine.

	Initial Value	First Optimized Value	Second Optimized Value
K_p	0.5000	0.2590	0.2630
K_i	0.0400	0.0112	0.0061
K_d	5.0000	4.8771	4.8531
Cost	$1.1552 \cdot 10^7$	$3.892 \cdot 10^6$	$3.7301 \cdot 10^6$

Table 1: PID gain values and associated costs.

settling crank angle of roughly 150 radians, compared to the hand-tuned response of 150 RPM overshoot and 200 radian settling angle. It is also clear that the optimized PID gains result in much less oscillation during the settling period.

The PSO algorithm required some initial manual tuning of the PID gains to get a decent starting point for the swarm. However, once a mediocre response is achieved, the PSO algorithm will improve the gains without further tuning. Because the PSO algorithm must run the simulation for each particle at each iteration, the computational requirements can be very high for a large number of particles leading to a long optimization time. However, the speed can be improved with the use of parallel computing, where the cost for each particle is calculated simultaneously.

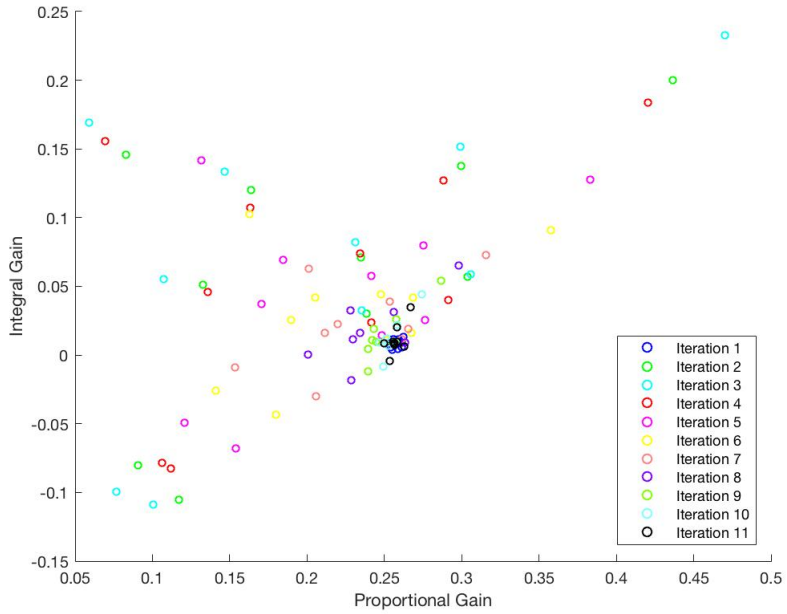

Figure 7: Particle positions (K_p and K_i only for viewing simplicity) during the secondary optimization routine.

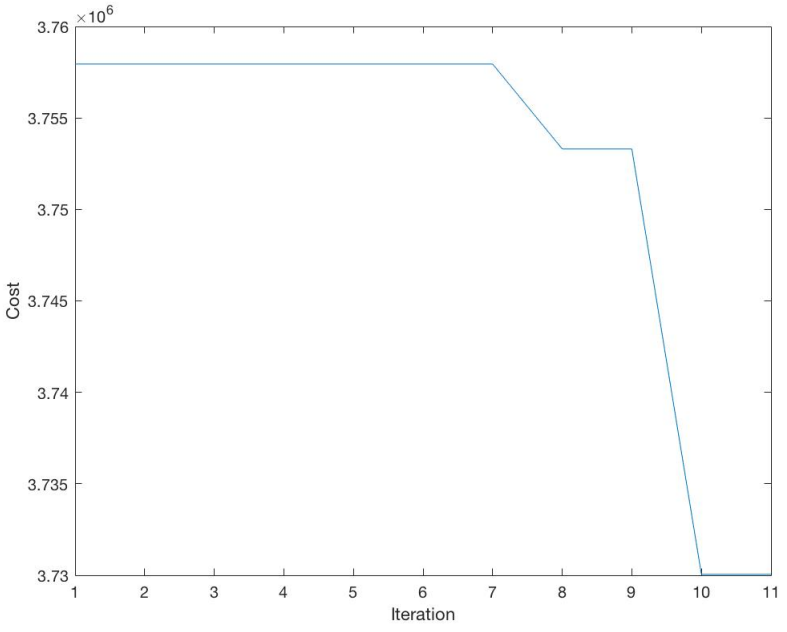

Figure 8: Cost function during the secondary optimization routine.

Appendix A. MATLAB Code: PSO

```
87
88      % Initialize PID Gains
89      %KP = 0.2590;
90 -    KP = 0.5;
91      %TI = 1;
92      %KI = 0.0112;
93 -    KI = 0.04;
94      %Td = 1;
95      %Kd = 4.8771;
96 -    Kd = 5;
97
98      %simOut = sim('idle_speed_model_project_pid');
99
100     % Colors
101 -    colors = [0 0 1; 0 1 0; 0 1 1; 1 0 0; 1 0 1; 1 1 0;
102         1 0.5 0.5; 0.5 0 1; 0.5 1 0; 0.5 1 1; 0 0 0];
103     % Set number of particles
104 -    d = 10;
105     % Define inertial constant
106     %w = 0.4;
107 -    w = 0.2;
108     % Define Cognitive component
109     %c1 = 0.6;
110 -    c1 = 0.3;
111     %define Social component
112     %c2 = 0.3;
113 -    c2 = 0.15;
114     % Define Tolerance
115 -    tol = 1e-2;
116     % Create particle vectors
117 -    Ks = rand(3,d);
118 -    for x = 1:d
119 -        Ks(:,x) = [KP;KI;Kd];
120 -    end
121 -    Ks = Ks + (rand(3,d)-rand(3,d))*0.01;
```

Figure 9: PSO Code (1)

```
116      % Create particle vectors
117 -    Ks = rand(3,d);
118 -    for x = 1:d
119 -        Ks(:,x) = [KP;KI;Kd];
120 -    end
121 -    Ks = Ks + (rand(3,d)-rand(3,d))*0.01;
122 -    Ks(:,1) = [KP;KI;Kd];
123 -    scatter(Ks(1,:), Ks(2,:),20,colors(1,:))
124      % Initialize particle best vector
125 -    P = Ks;
126      % Initialize cost vector
127 -    costs = rand(1,d);
128      % Initialize Velocities
129 -    v = rand(3,d) - rand(3,d);
130      % Perform Initial cost calculations
131 -    for x = 1:d
132 -            KP = Ks(1,x);
133 -            KI = Ks(2,x);
134 -            Kd = Ks(3,x);
135              % Run Initial Simulation
136 -            simOut = sim('idle_speed_model_project_pid');
137              % store cost in G array
138 -            costs(x) = sum((Speed.Data-800).^2);
139 -    end
140 -    particle_bests = costs;
141 -    [global_best,i] = min(costs);
142 -    G = Ks(:,i);
143 -    COSTS = [global_best];
144 -    t=0;
145 -    cindex = 2;
146 -    while t < 10
147          %randomize r ans s vectors
148 -        r = rand(3,d);
149 -        s = rand(3,d);
150 -        G_ = rand(3,d);
```

Figure 10: PSO Code (2)

```
146 -   while t < 10
147         %randomize r ans s vectors
148 -       r = rand(3,d);
149 -       s = rand(3,d);
150 -       G_ = rand(3,d);
151 -       for x = 1:d
152 -           G_(:,x) = G;
153 -       end
154         % update velocities
155 -       v = (w*v) + ((c1*r).*(P - Ks)) + ((c2*s).*(G_ - Ks));
156         % update gains
157 -       Ks = Ks + v;
158 -       hold on
159 -       scatter(Ks(1,:), Ks(2,:),20,colors(cindex,:))
160 -       cindex = cindex + 1;
161         % initialize new cost array
162 -       costs_new = rand(1,d);
163         % find new costs
164 -       for x = 1:d
165 -           KP = Ks(1,x);
166 -           KI = Ks(2,x);
167 -           Kd = Ks(3,x);
168             % Run Initial Simulation
169 -           simOut = sim('idle_speed_model_project_pid');
170             % store cost in G array
171 -           costs_new(x) = sum((Speed.Data-800).^2);
172 -       end
173
174 -       for x = 1:d
175 -           if costs_new(x) < particle_bests(x)
176 -               particle_bests(x) = costs_new(x);
177 -               P(:,x) = Ks(:,x);
178 -           end
179 -       end
180 -       [m,i] = min(costs_new);
```

Figure 11: PSO Code (3)

```
164 -       for x = 1:d
165 -           KP = Ks(1,x);
166 -           KI = Ks(2,x);
167 -           Kd = Ks(3,x);
168             % Run Initial Simulation
169 -           simOut = sim('idle_speed_model_project_pid');
170             % store cost in G array
171 -           costs_new(x) = sum((Speed.Data-800).^2);
172 -       end
173
174 -       for x = 1:d
175 -           if costs_new(x) < particle_bests(x)
176 -               particle_bests(x) = costs_new(x);
177 -               P(:,x) = Ks(:,x);
178 -           end
179 -       end
180 -       [m,i] = min(costs_new);
181 -       if m < global_best
182 -           global_best = m;
183 -           G_new  = Ks(:,i);
184 -       else
185 -           G_new = G;
186 -       end
187         %diff = abs(m - G);
188 -       G = G_new;
189 -       t = t+1;
190 -       COSTS = [COSTS global_best];
191 -   end
192     %plot(COSTS)
193
194 -   KP = G(1);
195 -   KI = G(2);
196 -   Kd = G(3);
197 -   simOut = sim('idle_speed_model_project_pid');
198     %}
```

Figure 12: PSO Code (4)