Purdue University

ECE 580: Optimization Methods for Systems and Control

FunWork #4

Ahmed Mohamed
akaseb@purdue.edu

April 10, 2015

1 Problem 1

$$A_1 = \begin{pmatrix} 1 & 1 \end{pmatrix}$$

$$A_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

$$A_1^\dagger = A_1^\top (A_1 A_1^\top)^{-1} = (2)^{-1} A_1^\top = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$$

$$A_2^\dagger = (A_2^\top A_2)^{-1} A_2^\top = (1)^{-1} A_2^\top = \begin{pmatrix} 1 & 0 \end{pmatrix}$$

$$(A_1 A_2)^\dagger = (1)^\dagger = (1)$$

$$A_2^\dagger A_1^\dagger = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} = (0.5)$$

$$(A_1 A_2)^\dagger \neq A_2^\dagger A_1^\dagger$$

2 Problem 2

The Griewank function is defined as:

$$f(x_1, x_2, ..., x_n) = 1 + \frac{1}{4000}\sum_{i=1}^{n} x_i^2 - \prod_{i=1}^{n} cos(\frac{x_i}{\sqrt{i}})$$

Listing 1 shows the source code of the Griewank function:

```
function [f] = gk(x)

sum = 0;
prod = 1;

for i = 1:length(x)
  sum = sum + x(i)^2/4000;
  prod = prod * cos(x(i)/sqrt(i));
end

f = sum - prod + 1;

end
```

Listing 1: The source code of the Griewank function

Listing 2 shows the source code of the PSO function that implements the Particle Swarm Optimization algorithm. It can be used for both minimization and maximization of a given function. The following configurations are used:

1. The minimum value for each coordinate of the particle positions is -5, and the maximum is 5. The positions are always clipped to be in that range.
2. The minimum value for each coordinate of the particle velocities is -0.5, and the maximum is 0.5. The velocities are always clipped to be in that range.
3. The number of iterations is 100.
4. The size of the population is 100.
5. The inertial constant is 0.8.
6. The cognitive coefficient is 2.
7. The social coefficient is 2.

```
function [] = pso(f, is_minimize, file_name)

min_x = -5;
max_x = 5;

min_v = -0.5;
max_v = 0.5;

max_k = 100;
d = 100;

w = 0.8;
c1 = 2;
c2 = 2;

x = min_x + (max_x - min_x) .* rand(2, d);
v = rand(2, d);
p = x;

```

```
g = x(:, 1);
for i = 2:d
  if (f(x(:, i)) < f(g) && is_minimize) || (f(x(:, i)) > f(g) && ~is_minimize)
    g = x(:, i);
  end
end

overall_best = zeros(1, max_k);
best = zeros(1, max_k);
worst = zeros(1, max_k);
average = zeros(1, max_k);

for k = 1:max_k
  r = rand(2, d);
  s = rand(2, d);

  v = w * v + c1 * r .* (p - x) + c2 * s .* bsxfun(@minus, g, x);
  v = max(min_v, min(max_v, v));

  x = x + v;
  x = max(min_x, min(max_x, x));

  worst(k) = f(x(:, 1));
  best(k) = f(x(:, 1));

  for i = 1:d
    average(k) = average(k) + f(x(:, i));

    if (f(x(:, i)) < f(p(:, i)) && is_minimize) || ...
      (f(x(:, i)) > f(p(:, i)) && ~is_minimize)
      p(:, i) = x(:, i);
      if (f(x(:, i)) < f(g) && is_minimize) || ...
        (f(x(:, i)) > f(g) && ~is_minimize)
        g = x(:, i);
      end
    end

    if (f(x(:, i)) > worst(k) && is_minimize) || ...
      (f(x(:, i)) < worst(k) && ~is_minimize)
      worst(k) = f(x(:, i));
    end
    if (f(x(:, i)) < best(k) && is_minimize) || ...
      (f(x(:, i)) > best(k) && ~is_minimize)
      best(k) = f(x(:, i));
    end
  end

  average(k) = average(k) / d;
  overall_best(k) = f(g);
end

g
f(g)


file_id = fopen(strcat(file_name, '.txt'),'w');
fprintf(file_id, 'Final Solution: x* = [%.10f %.10f] \nf(x*) = %.10f\n', ...
  g(1), g(2), f(g));

k = [1:max_k];
figure
plot(k, overall_best)
print(strcat(file_name, 'overall_best'), '-dpng')
figure
plot(k, best)
print(strcat(file_name, 'best'), '-dpng')
figure
```

```
plot(k, average)
print(strcat(file_name, 'average'), '-dpng')
figure
plot(k, worst)
print(strcat(file_name, 'worst'), '-dpng')

end
```

Listing 2: The source code of the PSO function

```
clc;
clear;

f = @gk;
pso(f, 1, 'problem2');
pso(f, 0, 'problem3');
```

Listing 3: The main source code of problems 2 and 3. It invokes the PSO for both minimization and maximization.

```
Final Solution: x* = [-0.0001206289 0.0002274571]
f(x*) = 0.0000000202
```

Listing 4: The output of Problem 2

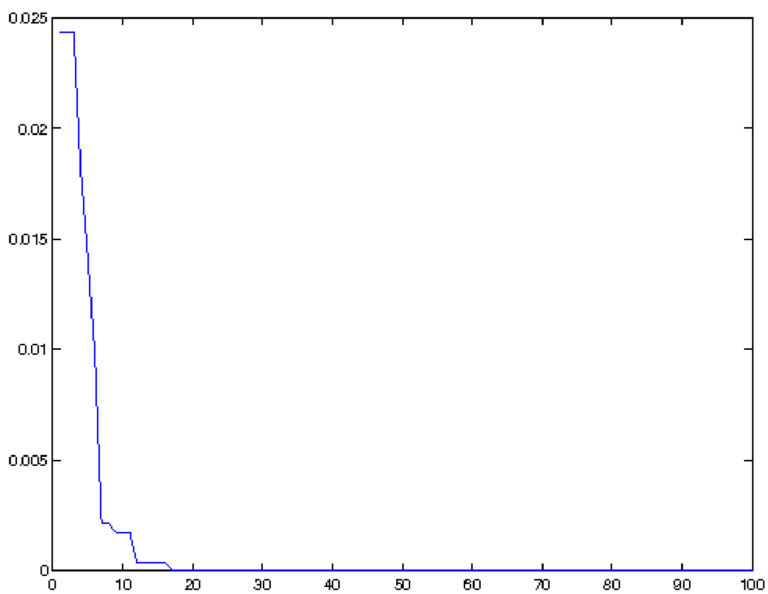

Figure 1: The objective function of the global best over the generations. This function is always decreasing because the global best is maintained.

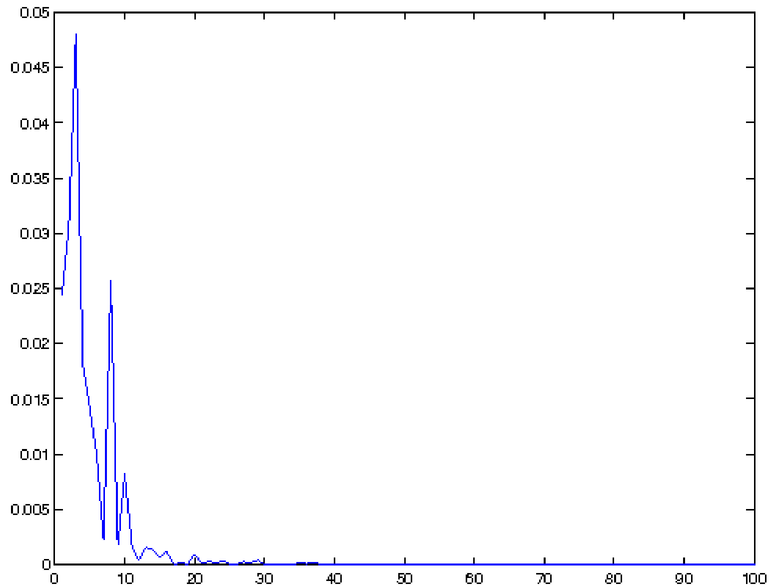

Figure 2: The objective function of the best particle in each generation.

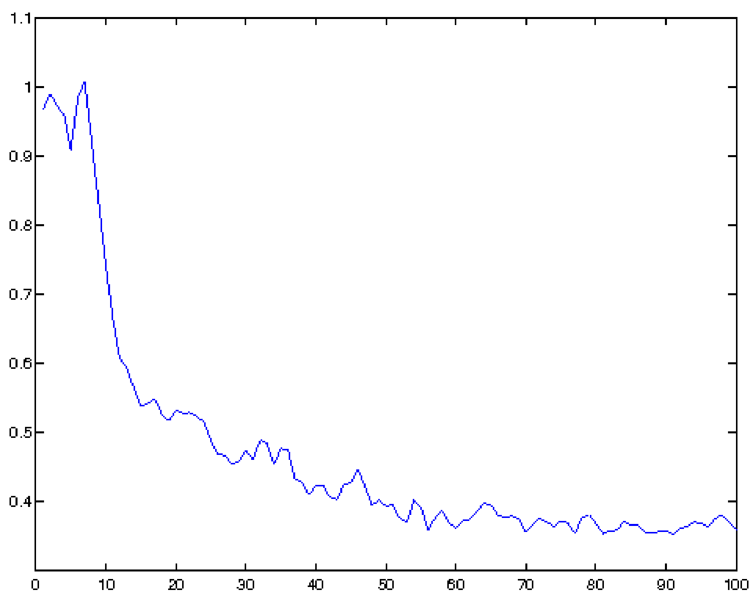

Figure 3: The average objective function of all the particles over the generations.

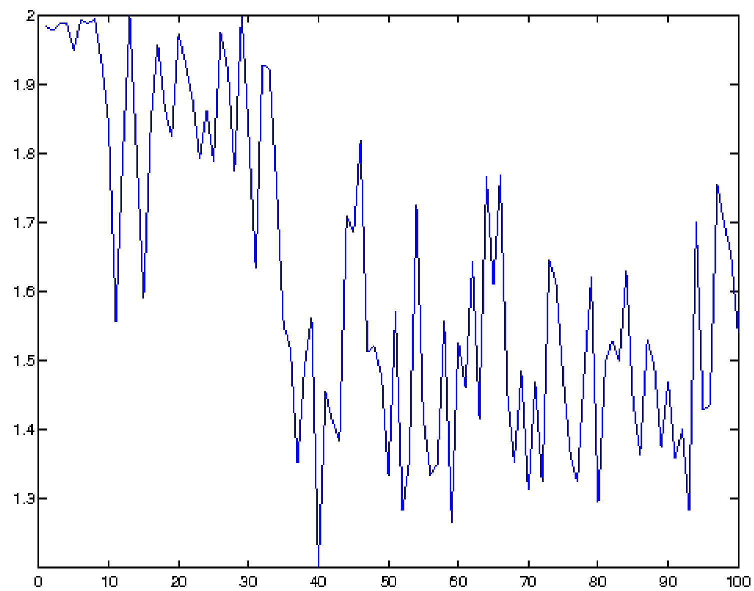

Figure 4: The worst objective function over the generations.

3 Problem 3

The same source code and the configurations of Problem 2 is used in Problem 3 as well. Listing 5 shows the output of Problem 3.

```
Final Solution: x* = [-3.1431671864 0.0003048406]
f(x*) = 2.0024686122
```

Listing 5: The output of Problem 3

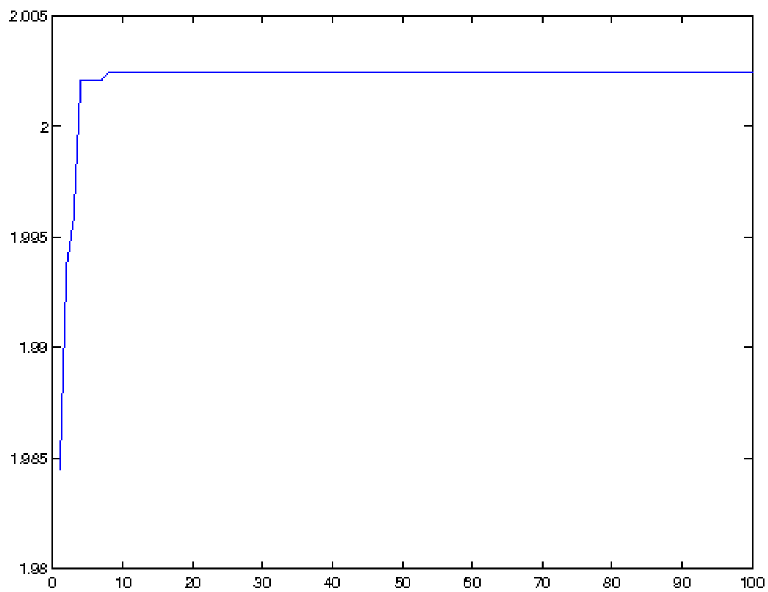

Figure 5: The objective function of the global best over the generations. This function is always decreasing because the global best is maintained.

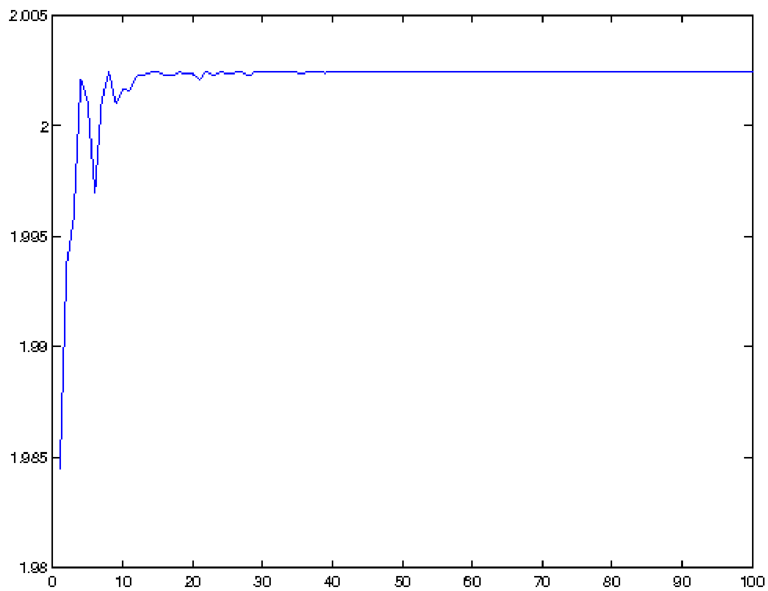

Figure 6: The objective function of the best particle in each generation.

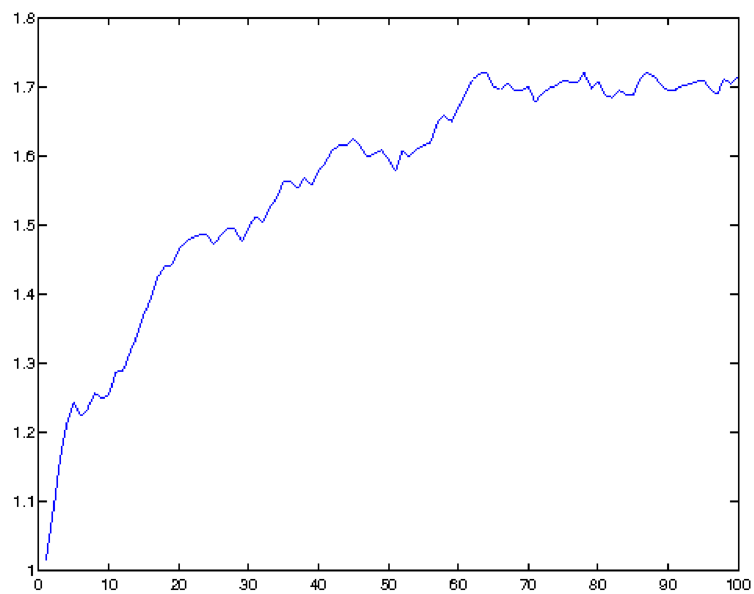

Figure 7: The average objective function of all the particles over the generations.

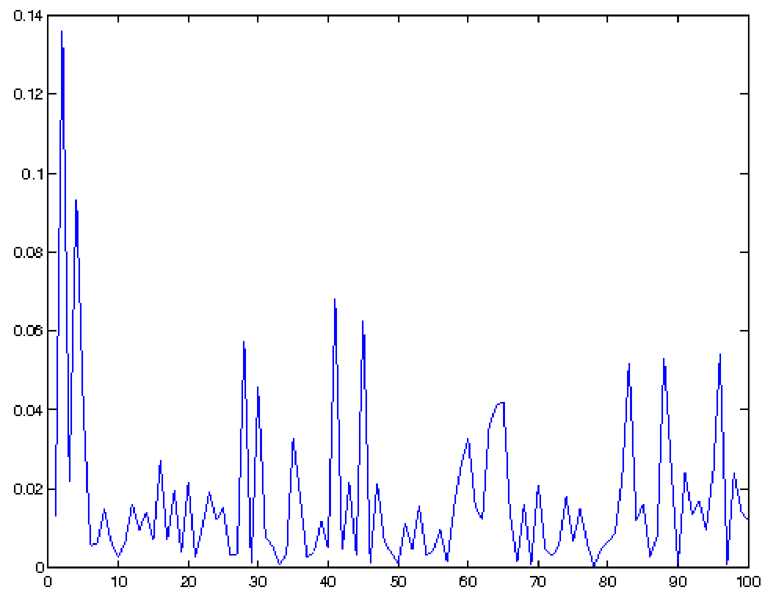

Figure 8: The worst objective function over the generations.

4 Problem 4

There are 10! different possible paths for 10 cities. Listing 6 shows the main source code of Problem 4. It implements the Genetic algorithm with the following configurations:

1. The number of iterations is 200.
2. The size of the population is 200.
3. The probability of crossover is 0.8.
4. The number of elites that are kept at each generation is 5.
5. The fitness function is the inverse of the distance along the route among the cities. The function used to calculate the fitness function is shown in Listing 7.

```
clc;
clear;

max_k = 200;
d = 200;
p_c = 0.8;
elites_num = 5;

cities = [0.4306 3.7094 6.9330 9.3582 4.7758 1.2910 4.83831 9.4560 3.6774 3.2849;
  7.7288 2.9727 1.7785 6.9080 2.6394 4.5774 8.43692 8.8150 7.0002 7.5569];
cities_num = size(cities, 2);

x = zeros(d, cities_num);
m = zeros(d, cities_num);
f = zeros(d, 1);

for i = 1:d
  x(i, :) = randperm(cities_num);
  f(i) = tsp_fitness(cities, x(i, :));
end

for k = 1:max_k

  [sorted_values,sort_index] = sort(f(:),'descend');
  for i = 1:elites_num
    m(i, :) = x(sort_index(i), :);
  end

  for i = elites_num + 1:d
    j = find(cumsum(f) / sum(f) - rand() > 0, 1);
    m(i, :) = x(j, :);

    if(rand() < p_c)
      j = randi([1 cities_num]);
      k = randi([1 cities_num]);

      temp = m(i, j);
      m(i, j) = m(i, k);
      m(i, k) = temp;
    end
  end

  x = m;
  for i = 1:d
    f(i) = tsp_fitness(cities, x(i, :));
  end
end

best = x(find(f == max(f), 1), :)
best_distance = 1 / tsp_fitness(cities, best)
```

```

x = cities(1, best);
x(cities_num + 1) = cities(1, best(1));
y = cities(2, best);
y(cities_num + 1) = cities(2, best(1));
figure
plot(x, y, 'b-o')
print('problem4', '-dpng')
```

Listing 6: The source code of Problem 4

```
function [f] = tsp_fitness(cities, order)

cities_num = size(cities, 2);

f = norm(cities(:, order(1)) - cities(:, order(cities_num)));

for i = 2:cities_num
  f = f + norm(cities(:, order(i)) - cities(:, order(i - 1)));
end

f = 1 / f;

end
```

Listing 7: The source code of the function to evaluate the fitness function of a chromosome

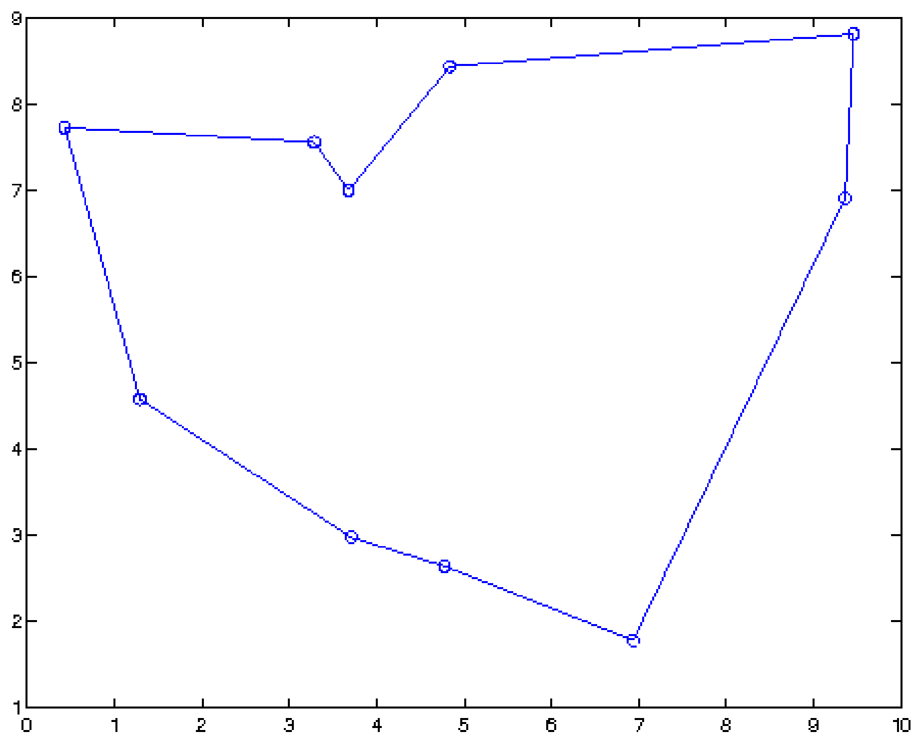

Figure 9: The optimal route among the cities

5 Problem 5

To solve the problem, I use the linprog method in Matlab as shown in Listing 8. Listing 9 shows the output of the problem. It is worth mentioning that the output values are double. However the problem indicates that some of the products should be integers, such as the number of boxes. This shows the need for integer linear programming. So the provided answer is the optimal double solution. If we round the values, this might not be the optimal integer solution.

```
clc;
clear;

f = [-6; -4; -7; -5];

A = [1 2 1 2; 6 5 3 2; 3 4 9 12];
b = [20; 100; 75];
lb = zeros(4,1);

[x, fval, exitflag, output, lambda] = linprog(f, A, b, [], [], lb);
x

file_id = fopen('problem5.txt','w');
fprintf(file_id, 'Solution: x* = [%.10f %.10f %.10f %.10f]\n', ...
  x(1), x(2), x(3), x(4));
fprintf(file_id, 'Revenue: f(x*) = %.10f\n', ...
  6 * x(1) + 4 * x(2) + 7 * x(3) + 5 * x(4));
```

Listing 8: The source code of Problem 5

```
Solution: x* = [14.9999999999 0.0000000001 3.3333333332 0.0000000001]
Revenue: f(x*) = 113.3333333326
```

Listing 9: The output of Problem 5