

## Q1. Linear Programming

### a) Dual Problem

Minimise d such that:

$$d = 8y_1 + 15y_2$$

Subject to:

$$2y_1 + y_2 \geq 3$$

$$2y_1 + 3y_2 \geq 2$$

$$y_1 + 3y_2 \geq 1$$

$$y_1, y_2 \geq 0$$

### b) Solving Problem P

objective  $f = 3x + 2y + 1z$

so for linprog:

$$f = -(3, 2, 1)^T$$

subject to

$$Ax \leq b$$

shown below:

```
f = -[3 2 1];
A = [2 2 1; 1 3 3];
b = [8; 15];
lb = zeros(length(f),1);
[x, fval] = linprog(f, A, b, [],[],lb,[])
```

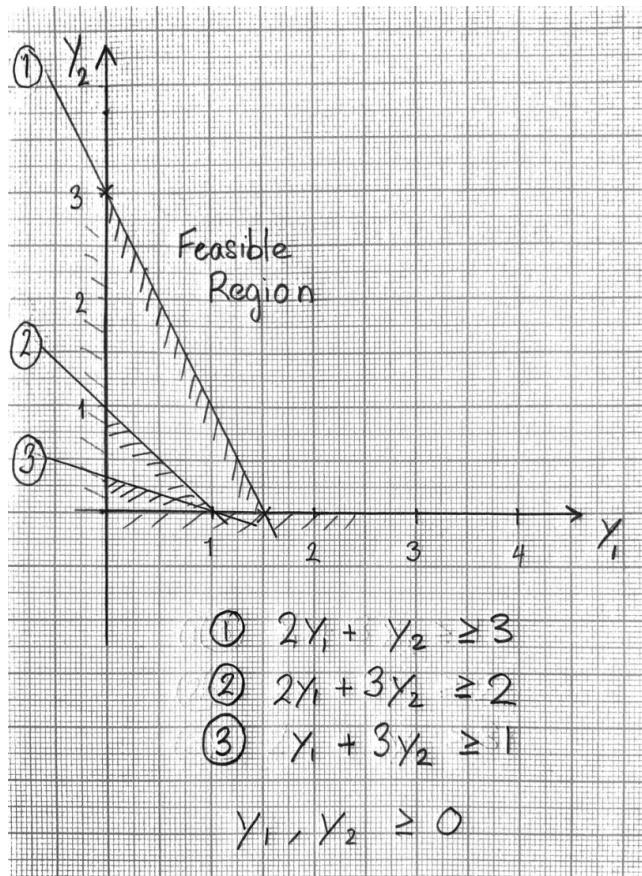
Optimal solution found.

```
x = 3x1
 4
 0
 0
fval = -12
```

```
% As this is a maximisation problem the true solution fval is *-1:
fval = -fval
```

```
fval = 12
```

### c) Solving the Dual Using Graphical Methods



**Figure 1: the constraints of the dual problem plotted. The constraints are numbered corresponding to the problem P variables that they are taken from.**

From the figure there are two feasible solutions (vertices) that may be suitable for the minimisation problem:

$$Y_{sol1} = (0, 3) \text{ or } Y_{sol2} = (1.5, 0)$$

The optimal solution minimises  $d = 8y_1 + 15y_2$  and so it clear that  $Y_{sol2}$  is optimal as it produces the smallest d value:

$$Y^* = (1.5, 0) \text{ with a value of 12.}$$

**This confirms that the optimal objective function values of problems P and D are equal and the strong duality theorem holds.**

#### d) Complementary Slackness Theorem

In the optimal solution of problem P, only the first variable is positive. The complementary slackness theorem states that the corresponding dual constraint (to the positive variable in P) is binding (not slack), whereas the constraints in D corresponding to the zero variables in the solution of P are slack.

From the graph in figure 1 it is clear that constraints 2 and 3 are slack, as they do not effect the feasible region, this corresponds to variables  $x_2, x_3$  in the solution  $X^*$  of P which are zero.

## e) Discussion of Solving the Primal vs Dual

### The primal problem:

2 constraints ( 2 slack variables )

$n$  variables

### The dual problem:

$n$  constraints ( $n$  slack variables)

2 variables

- **Number of dimensions to be considered**

The dual problem will actually require more variables than just the main variables as the simplex style method that is used by linprog for solutions will use slack variables for all constraints to turn them into equality constraints with unknowns. From this aspect, the dual problem will likely become as more complex to solve as the total number of variables will remain the same, if the slack variables are included, and the number of constraints increases.

- **Redundancy in constraints**

As shown previously, the complementary slackness theorem shows that for any zero variables in the optimal solution of the primal problem, the corresponding constraints in the dual are slack. This means that although the number of variables gets transposed to the number of constraints in the dual, many of them are likely to be redundant in the feasible region in the solution space.

- **Method of linprog solving**

Linprog uses a simplex style method for solving LP problems by default. However there is an option for linprog to use a different solver, this is accessed for large scale problems: linprog can use a linear interior point solver to get the optimal solution [1]. This method traverses the feasible region instead of checking the vertices and so is more resistant to the number of constraints:

the big O complexity of the interior point solver is given as:

$$O\left(n^{3.5} \log\left(\frac{1}{\epsilon}\right)\right) \quad [2]$$

Where  $n$  is the number of variables and  $\epsilon$  is the target accuracy. In this description, the number of constraints does not impact the complexity and so it would be fair to conclude that the problem, P or D, with fewer true variables is the least complex from a computing standpoint.

## References

[1] <http://www.ece.northwestern.edu/local-apps/matlabhelp/toolbox/optim/linprog.html>

[2] <https://pubs.siam.org/doi/book/10.1137/1.9781611971453>

## Q2. Integer linear Programming

Introducing the problem cost matrix

```
clc; clear  
c = readmatrix("ProblemRankData.csv");  
[students, groups] = size(c)
```

```
students = 63  
groups = 14
```

```
total = students*groups;  
f = reshape(c', [], 1);
```

### a) Each student can only do one project

Make A matrices:

The A matrices must be constructed in a way that when multiplied with the solutions x it returns less than or equal to 1 (as all the x values will be limited to 1 or 0).

Suppose each of the costs (preferences) in the matrix are identified as  $C_{i,j}$  where i is the student and j is the group. The maximum sum of C over i can only be 1 as this is the sum of how many groups that student is allocated to:

for all students i:  $\sum_j^{groups} C_{i,j} \leq 1$  (1)

This translates to

$$C_{1,1} + C_{1,2} + \dots + C_{1,j} \leq 1$$

$$C_{2,1} + C_{2,2} + \dots + C_{2,j} \leq 1$$

...

Which is extracted from cost matrix:

$$C = \begin{bmatrix} C_{1,1} & C_{2,1} & C_{3,1} \\ C_{1,2} & C_{2,2} & C_{3,2} \\ C_{1,3} & C_{2,3} & C_{3,3} \end{bmatrix}$$

using matrix multiplication of C and the a coefficient matrix A1 in the form (this is the same as the supply matrix in the transportation problem):

$$A1 = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Where a series of ones rotates by the number of groups between each row. In matlab this can be achieved by using the builtin circshift function:

```
% A for one student can only do one project  
A_circ = zeros(1, total);
```

```

A_circ(1:groups) = 1;

A1 = A_circ;
for i = 1:students-1
    A_new = circshift(A_circ, i*groups);
    A1 = [A1; A_new];
end

% b for one student can only do one project
b1 = ones(1, students);

```

The b1 matrix is just a column vector of ones completing the equation 1

### b) Ensuring group sizes

The max and min group sizes will be defined by:

$$g_{min} \leq \sum_i^{students} C_{i,j} \leq g_{max}$$

In other words, the sum of students in each group must be greater than the minimum group size and less than the maximum.

So for each group (sum over i - number of students):

$$C_{1,1} + C_{2,1} + \dots + C_{i,1} \geq g_{min}$$

$$C_{1,1} + C_{2,1} + \dots + C_{i,1} \leq g_{max}$$

That first quation will need to be rearranged into the normal less than or equal to form by multiplying everything by -1:

$$-C_{1,1} - C_{2,1} - \dots - C_{i,1} \leq -g_{min}$$

Once again, to retrieve these equations from the cost matrix C, matrix multiplication of C and coefficient matrices A2 and A3 will be used:

Both A2 and A3 will take the form of identity matrices stacked horizontally in the form (with the matrix corresponding to the minimum group size being the negative):

$$A1 = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$

Where the number of rows is the number of groups so that when multiplied by the solution it sums the total values for each group. The number of columns is the number of groups times the number of students.

For the b vectors in this they are equal to vectors of the group sizes (with the minimum also being negative):

$$b2_{min} = -\begin{bmatrix} g_{min} \\ g_{min} \\ g_{min} \\ \dots \end{bmatrix}, \quad b3_{max} = \begin{bmatrix} g_{max} \\ g_{max} \\ g_{max} \\ \dots \end{bmatrix}$$

```
% A for max and min group size
A2 = eye(groups);
A2 = repmat(A2, 1, students);
A3 = A2;

% b for max and min group size
minGroup = 3;
maxGroup = 5;

b2_min = minGroup * ones(1,groups);
b3_max = maxGroup * ones(1,groups);
```

### c) Calling intlinprog

intlinprog takes the following inputs: (f, intcon, A, b, A\_eq, b\_eq, lb, ub)

f is the cost matrix

A will contain all the inequality constraint coefficient matrices put together. In this case this is the group size constraints A2 and A3. A2 will correspond to the minimum group size and A3 to the maximum and so A2 will be negative. The corresponding b matrix will also be assembled in the same way:

$$A = \begin{bmatrix} -A2 \\ A3 \end{bmatrix}, \quad b = \begin{bmatrix} -b2_{min} \\ b3_{max} \end{bmatrix}$$

The equality constraints will be the ones that ensure that each student can only be assigned to one group.

So A1 and b1 from part b.

All of these constraints also require that the solution values for student allocations are either 1 or 0: either the student is or isn't in a group. This is enforced by intcon and the bounds. The integer constraint, intcon, is written in a way that every single solution element must be constrained as an integer and so each index of the solution must be passed in 1 to the total number of students times groups.

Finally upper and lower bounds must be added as the lower bound of any of the solution elements is 0 and the upper is 1 and so these are just vectors of the length of the number of students times the number of groups.

```
% intcon

intcon = [1:total];

% bounds
lb = zeros(total, 1);
```

```

ub = ones(total, 1);

% Arranging for intlinprog
A = [A3; -A2];
b = [b3_max'; -b2_min'];

A_eq = A1;
b_eq = b1';

```

Once everything is defined, intlinprog can be called.

```

% Run intlinprog
[x, fval] = intlinprog(f, intcon, A, b, A_eq, b_eq, lb, ub);

```

LP: Optimal objective value is 83.000000.

Optimal solution found.

Intlinprog stopped at the root node because the objective value is within a gap tolerance of the optimal value, options.AbsoluteGapTolerance = 0 (the default value). The intcon variables are integer within tolerance, options.IntegerTolerance = 1e-05 (the default value).

fval

fval = 83

The solution can be visualized very simply by viewing which students got which choice

```

solutions = reshape(x, groups, students)';

% Show results
og_choiceAllocations = solutions.*c;
% worst choice value:
worst_choice_allocated = max(max(og_choiceAllocations))

worst_choice_allocated = 4

```

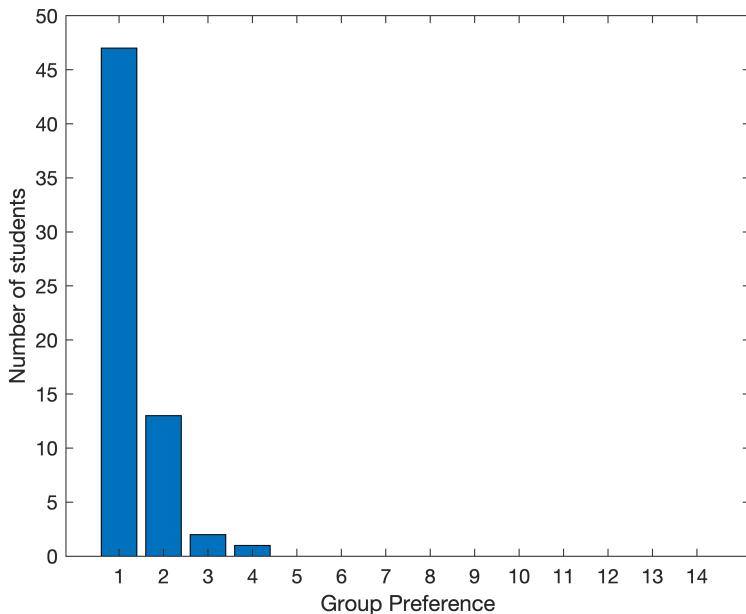
The worst choice is given above.

```

% plot how many of each choice:
choices = [];
for i = 1:14
    choices(i) = sum(og_choiceAllocations(:) == i);
end

bar(choices);
xlabel("Group Preference");
ylabel("Number of students");

```



```
choice1 = choices(1)
```

```
choice1 = 47
```

```
choice2 = choices(2)
```

```
choice2 = 13
```

```
choice3 = choices(3)
```

```
choice3 = 2
```

```
choice4 = choices(4)
```

```
choice4 = 1
```

#### d) Set a hard limit on the worst preference allowed:

To implement this the A\_eq matrix should be copied (into A\_hard) to contain the values of c instead of ones. This will mean that the limits imposed by a copy of b\_eq (b\_hard) will be used as hard limits on choice maximum, in the form of A\_hard\*x <= b\_hard. This will be appended to the A inequality matrix: (everything else will stay the same).

```
A_hard = A1;
A_hard_circ = zeros(students,total);
A_hard_circ(1,1:groups) = c(1,:);
for i = 1:students-1
    A_hard_circ(i+1,(groups*i)+1:(groups*(i+1))) = c(i+1,:);
end
```

```
A_hard_circ
```

```
A_hard_circ = 63x882
 10      2      5      9      1     14      6      3     12      4     13      7     11 ...
```

`b_hard` will be just the maximum choice allocation

```
max_choice = 3;  
b_hard = max_choice*ones(students,1);
```

## New intlinprog call

```
[x, fval] = intlinprog(f, intcon, [A;A_hard_circ], [b;b_hard], A_eq, b_eq,lb, ub);
```

No feasible solution found.

Intlinprog stopped because no point satisfies the constraints.

There should be no feasible solutions as now with a lower value of  $r$ . There is no integer solution that will satisfy a smaller maximum preference than 4 as that is already being optimised, and values of  $r$  greater than 4 also do not affect the result as that would be not optimal.

### e) Exploration and Experimentation

## Replacing the cost matrix with higher powers

```

all_choices = [];
for power = 1: 20
    c_power = c.^power;
    f_power = reshape(c_power',[1],1);

    % Run intlinprog
    [x, fval] = intlinprog(f_power, intcon, A, b, A_eq, b_eq,lb, ub);

    solutions = reshape(x, groups,students)';
    choiceAllocations = solutions.*c;

    % how many of each choice:
    choices = [];
    for i = 1:14
        choices(i) = sum(choiceAllocations(:) == i);
    end

    all_choices(power,:) = choices;

```

optimal value, options.AbsoluteGapTolerance = 0 (the default value). The intcon variables are integer within tolerance, options.IntegerTolerance = 1e-05 (the default value).

all\_choices

Some values have changed but not many, the worst allocation is still a 4th choice group. The power stops making a difference once the cost matrix is cubed. Strangely this means that the number of second choice options increases at the cost of fewer first and third choice. This is perhaps because the 4th choice option has to be used otherwise the solution is very different.

## How does varying group size impact the solution?

```

counter = 1;
all_choices = zeros(14*14,14);

for min_group = 1:14
    for max_group = 1:14
        b2_min = min_group * ones(1,groups);
        b3_max = max_group * ones(1,groups);
        b = [ b3_max'; -b2_min' ];

        % Run intlinprog
        [x, fval,exitflag, output] = intlinprog(f, intcon, A, b, A_eq, b_eq,lb, ub);
        if(isempty(x))
        else
            solutions = reshape(x, groups,students)';
            choiceAllocations = solutions.*c;

            % how many of each choice:
            choices = [];
            for i = 1:14
                choices(i) = sum(choiceAllocations(:) == i);
            end
            all_choices(counter,:) = choices;
        end
        if(isempty(fval))

```

```

        fval = 0;
    end
    fval_table(counter,1) = fval;
    fval_table(counter,2) = min_group;
    fval_table(counter,3) = max_group;

    counter= counter + 1;
end
end

```

No feasible solution found.

Intlinprog stopped because no point satisfies  
the constraints.

No feasible solution found.

Intlinprog stopped because no point satisfies  
the constraints.

No feasible solution found.

Intlinprog stopped because no point satisfies  
the constraints.

No feasible solution found.

Intlinprog stopped because no point satisfies  
the constraints.

LP:                   Optimal objective value is 82.000000.

Optimal solution found.

Intlinprog stopped at the root node because the  
objective value is within a gap tolerance of the  
optimal value, options.AbsoluteGapTolerance = 0  
(the default value). The intcon variables are  
integer within tolerance,  
options.IntegerTolerance = 1e-05 (the default  
value).

LP:                   Optimal objective value is 73.000000.

Optimal solution found.

Intlinprog stopped at the root node because the  
objective value is within a gap tolerance of the  
optimal value, options.AbsoluteGapTolerance = 0  
(the default value). The intcon variables are  
integer within tolerance,  
options.IntegerTolerance = 1e-05 (the default  
value).

LP:                   Optimal objective value is 69.000000.

```
No feasible solution found.  
Intlinprog stopped because no point satisfies  
the constraints.
```

```
No feasible solution found.  
Intlinprog stopped because no point satisfies  
the constraints.
```

```
No feasible solution found.  
Intlinprog stopped because no point satisfies  
the constraints.
```

```
No feasible solution found.  
Intlinprog stopped because no point satisfies  
the constraints.
```

```
No feasible solution found.  
Intlinprog stopped because no point satisfies  
the constraints.
```

```
No feasible solution found.  
Intlinprog stopped because no point satisfies  
the constraints.
```

```
No feasible solution found.  
Intlinprog stopped because no point satisfies  
the constraints.
```

```
No feasible solution found.  
Intlinprog stopped because no point satisfies  
the constraints.
```

```
No feasible solution found.  
Intlinprog stopped because no point satisfies  
the constraints.
```

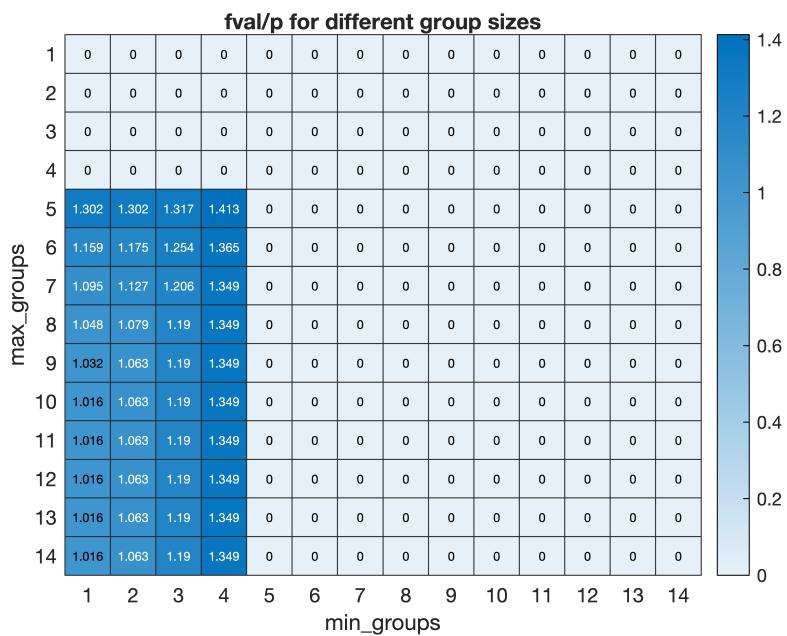
```
No feasible solution found.  
Intlinprog stopped because no point satisfies  
the constraints.
```

```
fval_table;  
% heatmap of all feasible solutions  
fvals = fval_table(:,1)./students;  
min_groups = fval_table(:,2);
```

```

max_groups = fval_table(:,3);
tbl = table(fvals,min_groups,max_groups);
h = heatmap(tbl,'min_groups','max_groups',ColorVariable='fvals',Title="fval/p for different group sizes");

```



The table above shows how the objective value varies with the group size limitation. As expected the lowest fval occurs when the min number is 1 and the max groups is large. Intlinprog return no solution (seen here as 0) when the minimum group size is greater than  $63/14 = 4.5$  or when the maximum group size is smaller than 4.5.

### Randomising row order of cost matrix

```

% run random perm
random_c = c(randperm(size(c, 1)), :);

f = reshape(random_c',[,1];
% A for max and min group size
A2 = eye(groups);
A2 = repmat(A2, 1, students);
A3 = A2;
% b for max and min group size
minGroup = 3;
maxGroup = 5;
b2_min = minGroup * ones(1,groups);
b3_max = maxGroup * ones(1,groups);
% intcon
intcon = [1:total];
% bounds
lb = zeros(total, 1);
ub = ones(total, 1);
% Arranging for intlinprog

```

```

A = [A3;-A2];
b = [ b3_max'; -b2_min'];
A_eq = A1;
b_eq = b1';

% Run intlinprog
[x, fval] = intlinprog(f, intcon, A, b, A_eq, b_eq,lb, ub);

```

LP: Optimal objective value is 83.000000.

Optimal solution found.

Intlinprog stopped at the root node because the objective value is within a gap tolerance of the optimal value, options.AbsoluteGapTolerance = 0 (the default value). The intcon variables are integer within tolerance, options.IntegerTolerance = 1e-05 (the default value).

```

solutions = reshape(x, groups,students)';

% Show results
rand_choiceAllocations = solutions.*random_c;
% worst choice value:
worst_choice_allocated = max(max(rand_choiceAllocations))

```

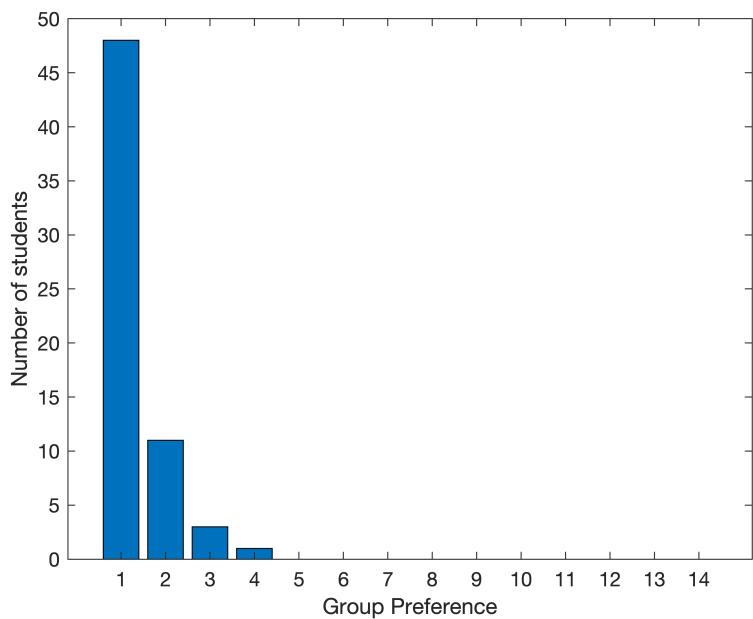
worst\_choice\_allocated = 4

```

% plot how many of each choice:
choices = [];
for i = 1:14
    choices(i) = sum(rand_choiceAllocations(:) == i);
end

bar(choices);
xlabel("Group Preference");
ylabel("Number of students");

```



The random permutation of the rows in the cost matrix does not appear to effect the allocation of groups and the work choice is not impacted.

### Q3. Non Linear Optimisation

Non linear optimisation.

$$f(x, y) = 2x^2 + 5y^2 - 2xy - 2x - 8y$$

a.

i) Freeze  $x=0$  to calculate  $y^*$ , freeze  $y=y^*$  to calculate  $x^*$ :

$$f(0, y) = 5y^2 - 8y$$

$$f'(0, y) = 10y - 8 \text{ differentiate}$$

$$y^* : 10y - 8 = 0, y^* = 0.8 \text{ calculate minimum}$$

$$f(x, 0.8) = 2x^2 - 1.6x - 2x$$

$$f'(x, 0.8) = 4x - 3.6 \text{ differentiate}$$

$$x^* : 4x - 3.6 = 0, x = 0.9 \text{ calculate minimum}$$

ii) Steepest descent algorithm:

Gradient of f:

$$\nabla f = \begin{pmatrix} 4x - 2y - 2 \\ 10y - 2x - 8 \end{pmatrix}$$

With starting point  $\underline{X}^0 = (0, 0)$

The starting direction is:

$d = -\nabla f|_{X=(0,0)}$  it is negative because we are finding the minimum.

$$d = (2, 8)$$

For one iteration of the steepest descent algorithm, the distance to travel along the steepest direction must be found. This is done by minimising the resulting value of f as a function of  $\lambda$ .

$\underline{X}^1 = \underline{X}^0 + \lambda \nabla f(\underline{X}^0)$  Here to find the minimum  $X^1$  a value of  $\lambda$  must be found that minimises  $\underline{X}^1$  as a function of  $\lambda$ .

$$\min f(\underline{X}^0 + \lambda \nabla f(\underline{X}^0))$$

$$\min f((0, 0) + \lambda(2, 8))$$

$$\min f(2\lambda, 8\lambda)$$

Substituting these values into the original function f:

$$\min (8\lambda^2 + 320\lambda^2 - 32\lambda^2 - 4\lambda - 64\lambda)$$

The minimum occurs when the derivative w.r.t  $\lambda$  is 0:

$$\frac{d}{d\lambda}(296\lambda^2 - 68\lambda) = 0$$

$$\lambda^1 = \frac{68}{592}$$

Now the first iterative step can be found

$$\underline{X}^1 = \underline{X}^0 + \lambda^1 d$$

$$\underline{X}^1 = (0, 0) + \frac{68}{592} (2, 8)$$

$$\underline{X}^1 = (0.2297..., 0.9189...)$$

```
nonlin_fun = @(x,y)(2.*x.^2 + 5*y.^2 - 2.*x.*y - 2.*x - 8.*y);
```

```
nonlin_fun(0,0)
```

```
ans = 0
```

```
nonlin_fun(136/592,544/592)
```

```
ans = -3.9054
```

The above verifies that we do move towards the minimum.

### iii) Quadprog

Set up the Hessian matrix H

```
% the hessian can be calculated in matlab with the symbolic toolbox and the
% following code.
syms x y
f = symfun(2*x.^2 + 5*y.^2 - 2*x.*y - 2*x - 8*y,[x y]);
H = double(hessian(f,[x,y]))
```

```
H = 2x2
```

```
 4   -2
 -2   10
```

Quadprog attempts to minimise:  $0.5(x' H x + f' x)$  subject to:  $A^*x \leq b$

In this example there are no constraints, so it is simply H and f\_obj that are used

```
% the f_obj values are just the linear coefficients of the function f
f_obj = [-2 -8]
```

```
f_obj = 1x2
 -2   -8
```

```
[solution, fval] = quadprog(H, f_obj);
```

Minimum found that satisfies the constraints.

Optimization completed because the objective function is non-decreasing in feasible directions, to within the value of the optimality tolerance, and constraints are satisfied to within the value of the constraint tolerance.

```
<stopping criteria details>
```

```
nonlin_fun(solution(1),solution(2))
```

```
ans = -5
```

#### iv) Plot f and above computations

```
x=linspace(0,2,100);
y=linspace(0,2,100);
[X,Y]=meshgrid(x,y);

Z = nonlin_fun(X,Y);

contour(X,Y,Z,100); axis equal; colorbar; title('Function f(x,y)')
hold on

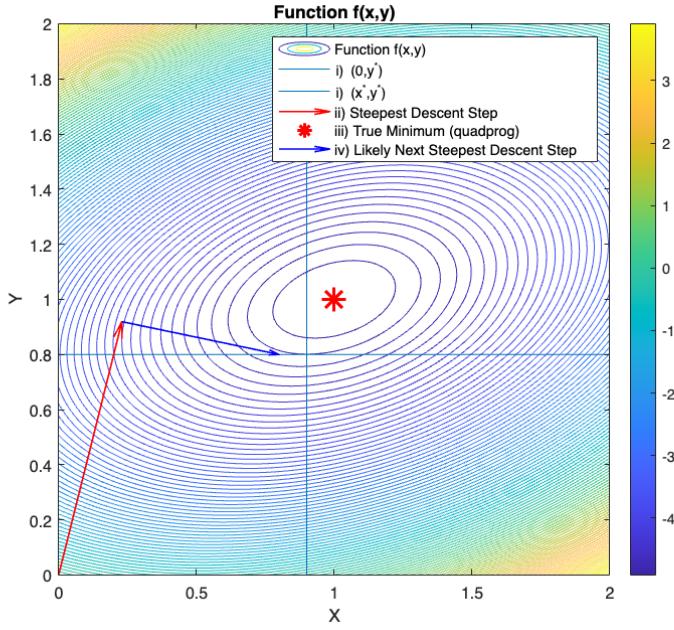
% overlay direction of gradient from starting point
p1 = [0 0];
p2 = [136/592 544/592];
dp = p2-p1;

% likely next steepest descent step (guess from graph)
p3 = p2;
p4 = [0.8 0.8];
dp2 = p4-p3;

% overlay parts i,ii,iii
line([0 2],[0.8 0.8])
line([0.9 0.9],[0 2])
quiver(p1(1),p1(2),dp(1),dp(2),0,LineWidth=1,Color='red')
plot([solution(1)],'o','Marker','*', 'Color','red','MarkerSize',15,'LineWidth',2)
quiver(p3(1),p3(2),dp2(1),dp2(2),0,LineWidth=1,Color='blue')

legend('Function f(x,y)','i') (0,y^*)','i) (x^*,y^*)','ii) Steepest Descent Step','iii)
xlabel('X')
ylabel('Y')

hold off
```



## Discussion

Parts i) demonstrate the two lines where at first  $x$  is frozen at  $x=0$  to find optimal  $y$ , then  $y$  is frozen at that value to find optimal  $x$ . You can see from the contours that both lines appear at a 1D minimum along the other variable. This is not a very accurate method of obtaining a true local minimum but does provide an estimate that is not too far off.

ii) shows the first step of the gradient descent algorithm where the steepest downhill at  $(0,0)$  is followed to its minimum (when it starts going up hill) that is the end of the first step. As shown in iv) I have estimated the next step of the algorithm as a movement along the steepest downhill from the previous point to a minimum in 1D. This will continue until the true minimum is reached.

iii) shows the true solution as given by quadprog.

## b) Kirchoff Solver and Network Preparation

Clear the workspace, set input parameters

```
clear; close all; more off

% define the size of the grid
n = 10;
```

numgrid numbers the grid points in a square with  $n+2$  being the number of nodes of each side to make a side length  $n$ .

delsq creates a grid from the numbered grid points.

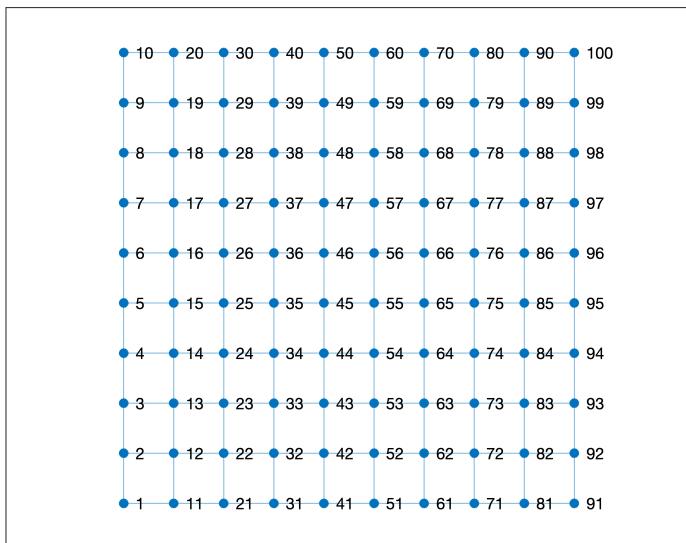
```
grid = delsq(numgrid('S',n+2));

% the graph function then creates a graph object (with nodes and edges)
% from the grid
ngraph = graph(grid,'omitselfloops');

% plot the graph
h = plot(ngraph);

% the data points need to be rounded to nearest integer so that the grid is
% straight
h.XData = floor((0:n^2-1)/n);
h.YData = rem((0:n^2-1), n);

axis equal
```



Calculate the positions of all nodes and the edges

```
% position of the nodes
x = h.XData;
y = h.YData;

% define edges st (graph object Edges outputs a table of edges and weights
% - we don't need the weights
st = table2array(ngraph.Edges);
st(:,3) = [];
```

Set Edge weights/resistances all as 1 since they are the same length

```
r = ones(length(st),1);
```

Create listAnode and listCnode (as starting node and ending node)

```
listAnode = [1];
listCnode = [n^2];
```

Call the solver, record solve time

```
tic;
[X,V]=KirchoffSolve(st,r,listAnode,listCnode);
```

---

#### Diagnostic Information

Number of variables: 182

Number of linear inequality constraints: 0  
 Number of linear equality constraints: 102  
 Number of lower bound constraints: 0  
 Number of upper bound constraints: 0

Algorithm selected  
 interior-point-convex

---

#### End diagnostic information

Iter	Fval	Primal Infeas	Dual Infeas	Complementarity
0	9.000000e+01	1.000000e+00	9.778500e-01	0.000000e+00
1	1.505835e+00	4.440892e-16	1.665335e-16	0.000000e+00

Optimization completed: The relative dual feasibility, 1.665335e-16, is less than options.OptimalityTolerance = 1.000000e-08, the complementarity measure, 0.000000e+00, is less than options.OptimalityTolerance, and the relative maximum constraint violation, 4.440892e-16, is less than options.ConstraintTolerance = 1.000000e-08.

```
t = toc
```

```
t = 0.2015
```

The voltage drop is the effective resistance of the network as the input current is 1. (V = IR)

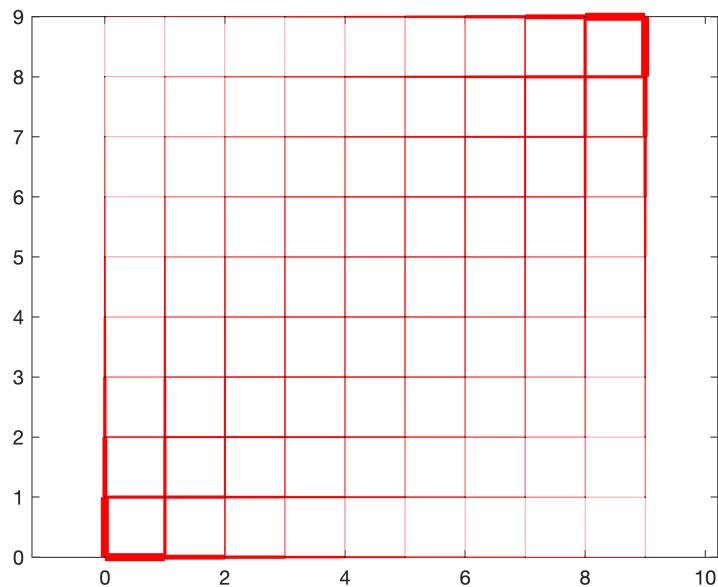
```
Vdrop = V(listCnode(1))-V(listAnode(1))
```

```
Vdrop = 3.0117
```

Plot what it looks like

```
figure;
hp=plot(x(st)',y(st)','r-);
axis equal

for i=1:length(hp)
    hp(i).LineWidth = max(1e-3,4*abs(X(i))/(max(abs(X)))); 
end
hold on
for i=1:max(max(st)) % ie over nodes
    hr(i)=rectangle('Position',[x(i)-0.005,y(i)-0.005,0.01,0.01],'Curvature',[1 1],'FaceColor','red')
end
```



## Exploring the size of N before my computer crashes

Package all the above into a callable function. Function defined with outputs of X,V as above, Vdrop for comparison, and t = time taken to run KirchoffSolve

The following while loop is run calling Kirch\_Run() which is the packaged function and the result is plotted with an exponential line of best fit.

```
% N_plot = [];
% t_plot = [];
% counter = 1;
% N = 4;
% t = 0;
% while t < 10
```

```
% [X,V,t,Vdrop] = Kirch_Run(N);
% N_plot(counter) = N;
% t_plot(counter) = t;
% counter=counter+1;
% N = N*2;
% end
```

The t values are plotted against the N values and an exponential fit is calculated:

```
% g = fitype('a-b*exp(-c*x)');
% f0 = fit(N_plot',t_plot',g,'StartPoint',[0 0 0]);
% xx = linspace(1,max(N_plot),50);
% plot(N_plot,t_plot,'o','DisplayName','Calculated Points');
% hold on
% plot(xx,f0(xx),'r-','DisplayName','Exponential line of Best Fit');
% xlabel("Dimension N")
% ylabel("Time to Compute (s)")
% title("Time Complexity")
% legend()
```

### **The resulting graph of time complexity for a maximum run time of 120s per iteration**

NB: I plotted this in a separate file using all the code above just so I don't run it everytime this script is run.

This is very clearly  $O(\exp(N))$  time complexity and so it will very quickly ruin my computer.

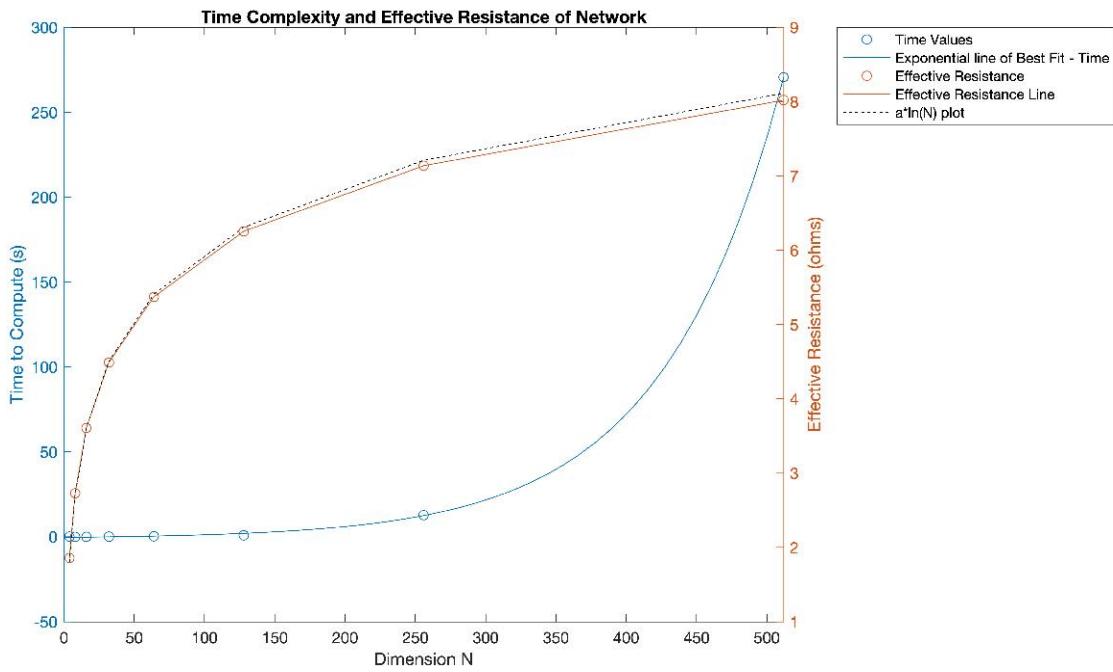
```
val =
General model:
val(x) = a-b*exp(-c*x)
Coefficients (with 95% confidence bounds):
a = -0.05508 (-0.2013, 0.09115)
b = -0.08459 (-0.1367, -0.03245)
c = -0.01911 (-0.02148, -0.01674)
```

The coefficient of the line of best fit of time are shown above.

So the time complexity is more accurately:  $O(0.08 \exp(0.02N))$

For my computer (2.9Ghz i5 with 8gb ram) to take 1 hour to solve a calculation would only require around 540 variables.

The resistance is plotted as joined up scatter.



The resistance curve looks suspiciously like an  $\log(N)$  plot and so I have added a plot of  $a^*\ln(N)$  for comparison with  $a$  being a mean of the resistance values/ $\ln(N)$  for the values plotted.

This suggests that the effective resistance of this sort of network tends towards a value proportional to  $a^*\ln(N)$ . With a value of  $a$  of just under 1.3. This value appears to be a constant related to the shape of the grid.

The  $\ln(N)$  shape also suggests that expanding the grid after a certain point produces limited increases in resistance, while the computation increases significantly. By using a predictive formula calculated from a smaller number of dimensions a fairly accurate estimate can be produced for large number  $N$ . It is likely logarithmic because as the size of the grid increases the current remains the same and small in the majority of the grid. It is only the area of higher currrent in the start and end corners that slightly increases in size.

## Q4. Global Optimisation

### a. Particle Swarm Methods

#### a..1 Summary

The original Particle Swarm Optimisation (PSO) was originally introduced by Dr. Kennedy and Dr. Eberhart [1] in 1995 as a method of finding global optima, at the time for use in neural network optimisation. The algorithm was built on the basis of swarm intelligence, as is found in bird flocks or similar. The main attributes of a swarm that were also found to be the main attributes of PSO algorithms are:

1. The swarm must have the ability to carry out simple space and time calculations. In the case of PSO this means that the particles must be able to change direction and velocity in response to stimuli.
2. The swarm must be able to respond to its surrounding environmental quality. A particle in PSO must be able to assess its objective function value.
3. The swarm has some diversity in its response to stimuli and so there is some variation in movement.
4. The swarm must have some stability and therefore not respond to all stimuli equally/ maintain some sort of its own motion.
5. The opposite to 4, is that the swarm must also adapt its behaviour when the gains made by this change are considered worthwhile.

These concepts are all expressed in the general PSO algorithm.

#### a..2 Method

The general idea of the algorithm is that a population of particles begins at random positions with random velocities in the space. At each iteration the position of each particle is updated by calculating the displacement of the particle due to a calculated velocity. The velocity of the particle is a function of time and is influenced by three parameters:  $\omega$  the inertia weight constant,  $C_1$  the cognitive coefficient, and  $C_2$  the social coefficient. The tuning of these parameters impacts the speed and accuracy of the PSO algorithm [2].

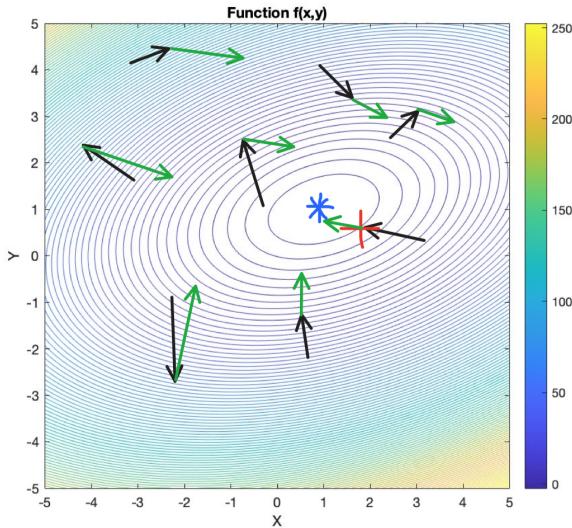


Figure 1: A rough demonstration of how PSO would act on a simple optimisation problem.

At each iteration, each particle updates its best value of the objective function pbest, and the global best gbest is also updated from all the particles. These variables give the swarm a 'memory' and allow locations of the optimal values (so far) to be stored. This allows the swarm intelligence to slowly discover new local optima while centering around a global optima. As shown in Figure 1, the black arrows indicate the start of the algorithm and the green show the actions of the particles on the second iteration. The red cross is the first iteration global best and the blue star is the true optimum. The particles velocities are impacted by their previous velocity ( $\omega$  inertia) and both the location of their own optimal and the swarm optimal from the previous iteration. This relationship is described in the following:

$$X^i(t+1) = X^i(t) + V^i(t+1) \quad (1)$$

$$V^i(t+1) = \omega V^i(t) + c_1 r_1(pbest^i - X^i(t)) + c_2 r_2(gbest - X^i(t)) \quad (2)$$

These two equations govern the actions of all the particles in the algorithm [3]. Equation 1 simply determines new positions at each iteration in the same way as physics particles. Equation 2 determines the direction and magnitude of the velocity through comparison of the current location to the local best and the global best.

$r_1, r_2$  are random numbers given from a distribution giving the algorithm a stochastic quality. The influence of the local versus the global best is determined by the  $C_1$  the cognitive coefficient and  $C_2$  the social coefficient as mentioned above. A greater  $C_1$  increases the exploration of new space by the particles whereas a greater  $C_2$  increases exploitation, i.e. how much each particle is influenced by the discoveries of the rest of the swarm.  $\omega$  is used to vary the inertia of a particle.

A greater inertia causes PSO to take more iterations to find a global optimum (if found) but also increases the likelihood it is not found. Whereas a smaller  $\omega$  means that the PSO algorithm finds a global optimum quicker (if found) but also increases the likelihood that it is not found.

$$W_i = 1.1 - \left( \frac{gbest_i}{pbest_i} \right) \quad (3)$$

The solution to this is varying the inertia with time [2]. It is found that varying it non-linearly by method of Global Local best Inertia Weight (GLbestIW), in equation 3 has the best results. Generally this means the inertia increases the closer each particle is to the global optimum meaning that they are kept in that area.

### a..3 Pros

1. The PSO method, first of all, is a simple algorithm with very basic calculation at each iteration. This means that it is computationally inexpensive and is therefore generally insensitive to the scaling of variables [4]. Similarly this means that the memory and processor requirements for implementing the method are minimal. In terms of computational efficiency the method can also be parallelized meaning each particle can be calculated simultaneously meaning that the algorithms are generally quick to run.
2. Another highly important feature is that the PSO method is derivative free and can therefore be applied to real world non-smooth objective functions [4].
3. By using the PSO method, the algorithm benefits from few parameters which can be tuned for optimal performance [5], as described in section a..2.

### a..4 Cons

1. One of the main drawbacks of the PSO method is that it has a tendency to result in early convergence on sub optimal points. In other words, the method will likely converge in proximity to the true global optimum quickly [4]. For some applications this makes this optimisation method not suitable.
2. The convergence of this method is also not guaranteed for some functions [4].
3. For use in refined search areas, the PSO method is often slower when finding a true optimum in a mid optimal area.

Many of these drawbacks can be limited and controlled through the use of the method parameters, allowing PSO to be used in many different applications successfully.

## References

- [1] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4, 1995.
- [2] Y. Shi and R. Eberhart. A modified particle swarm optimizer. In *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*, pages 69–73, 1998.
- [3] M Senthil Arumugam and M V C Rao. On the performance of the particle swarm optimization algorithm with various inertia weight variants for computing optimal control of a class of hybrid systems. *Discrete Dynamics in Nature and Society*, 2006:79295, 2006.
- [4] Mohd Nadhir Ab Wahab, Samia Nefti-Meziani, and Adham Atyabi. A comprehensive review of swarm optimization algorithms. *PloS one*, 10(5):e0122827, 2015.
- [5] Ahmad Rezaee Jordehi and Jasronita Jasni. Particle swarm optimisation for discrete optimisation problems: a review. *Artificial Intelligence Review*, 43(2):243–258, 2015.

## b. Developing PSO code

```
hold off; clear; close all; clc;
```

Define the eggholder function

```
fun = @(x)(-(x(2)+47)*sin(sqrt(abs((x(1)/2) + (x(2) + 47)))) + ...
-x(1)*sin(sqrt(abs(x(1) - (x(2) +47)))))

fun = function_handle with value:
@(x)(-(x(2)+47)*sin(sqrt(abs((x(1)/2)+(x(2)+47))))+-x(1)*sin(sqrt(abs(x(1)-(x(2)+47)))))

% True value of global optimum (check function works) and for comparison
% later
true = fun([512, 404.2319]);
```

Run `x = particleswarm(fun,nvars)` where nvars is the number of dimensions.

```
nvars = 2;
```

The bounds are the search domain as given in the wikipedia page

```
lb = [-512,-512];
ub = [512,512];
```

The options as with all optimisation can be defined. The swarm size is simply the number of particles, and the number of iterations that are carried out is MaxIterations, after attempting with default parameters, I set the options to those shown below as it provided a correct solution quickly.

Generally a greater number of iterations and a greater swarm size will result in better results at the cost of computational price.

```
options = optimoptions('particleswarm','SwarmSize',200,MaxIterations=20);

[x, fval] = particleswarm(fun,nvars,lb,ub,options)
```

```
Optimization ended: number of iterations exceeded OPTIONS.MaxIterations.
x = 1x2
  512.0000  404.1914
fval = -959.6388
```

```
error = true - fval
```

```
error = -0.0019
```

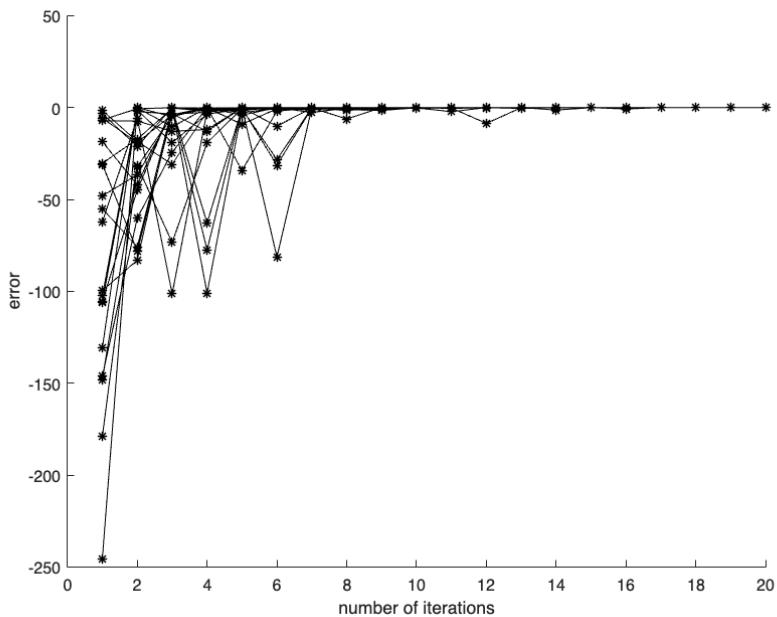
Test for many attempts

```
hold on
for repeat = 1:20
    fvals = [];
    for i = 1: 20
        options = optimoptions('particleswarm','SwarmSize',200,MaxIterations=i);
        [x, fval] = particleswarm(fun,nvars,lb,ub,options);
```

```

    fvals(i) = true - fval;
end
plot(1:i,fvals,'k-',LineWidth=0.1);
plot(1:i,fvals,'k*',LineWidth=1);
ylabel('error');
xlabel('number of iterations');
end

```



After plotting the optimisation for 20 goes at the same problem you can see the stochastic property of the algorithm and also that it is not always guaranteed to converge in the set time steps. A better measure of convergence would be the change on previous fval estimates which would stop the algorithm from diverging after attaining a zero error.