Optimisation Theory and Applications Coursework

Written By:

Jamie Bell-Thomas

-

ws19177@bristol.ac.uk

-

Student Number: 1820499

University of Bristol
Engineering Design
Faculty of Engineering

Submitted in December 2022

1 Linear Programming

1.a

	Primal 'P'	Dual 'D'
Objective Function	$\max\ z_x = 3x_1 + 2x_2 + x_3$	$\min\ z_y = 8y_1 + 15y_2$
Constraint Equations	$2x_1 + 2x_2 + x_3 \leq 8$	$2y_1 + y_2 \geq 3$
	$x_1 + 3x_2 + 3x_3 \leq 15$	$2y_1 + 3y_2 \geq 2$
	$x_1, x_2, x_3 \geq 0$	$y_1 + 3y_2 \geq 1$
		$y_1, y_2 \geq 0$

1.b

MatLab's linprog function is used to solve linear programming problems. This section will look at the input parameters for linprog. It will then look at solving the Primal problem defined in Section 1.a using linprog.

1.b.1 linprog

Syntax:

$$\text{[x,fval,exitflag]} = \text{linprog(f,A,b,Aeq,beq,lb,ub)}$$

This solves:

$$\min\ f^T x \text{ such that } \begin{cases} A \cdot x \leq b, \\ Aeq \cdot x = beq, \\ lb \leq x \leq ub. \end{cases}$$

$x =$ Optimal decision variable values
fval = Objective function value at the optimum point
exitflag = An integer code for the reason the solver halted its iterations [4]

Note how linprog [3] solves minimisation problems by default. This means the Primal problem has to be converted from a maximisation to a minimisation problem.

$$\max z_x = 3x_1 + 2x_2 + x_3$$
$$\min z_x = -3x_1 - 2x_2 - x_3$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad f = \begin{bmatrix} -3 \\ -2 \\ -1 \end{bmatrix} \qquad A = \begin{bmatrix} 2 & 2 & 1 \\ 1 & 3 & 3 \end{bmatrix}$$

$$b = \begin{bmatrix} 8 & 15 \end{bmatrix} \qquad lb = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad Aeq = beq = ub = \begin{bmatrix} \end{bmatrix}$$

$$\text{[x,fval,exitflag]} = \text{linprog(f,A,b,Aeq,beq,lb,ub)}$$
$$x = \begin{bmatrix} 4 & 0 & 0 \end{bmatrix}$$
$$\text{fval} = -12$$
$$\text{exitflag} = 1$$

This means the function has converged to an optimum point, which is: $x_1 = 4$, $x_2 = x_3 = 0$. This produced an optimal value of 12 in the original maximisation problem.

1.c

Dual problem derived in Section 1.a:

$$\begin{aligned} \min z_y &= 8y_1 + 15y_2 \\ \text{Subject to:} & \\ 2y_1 + y_2 &\geq 3 \\ 2y_1 + 3y_2 &\geq 2 \\ y_1 + 3y_2 &\geq 1 \\ y_1, y_2 &\geq 0 \end{aligned}$$

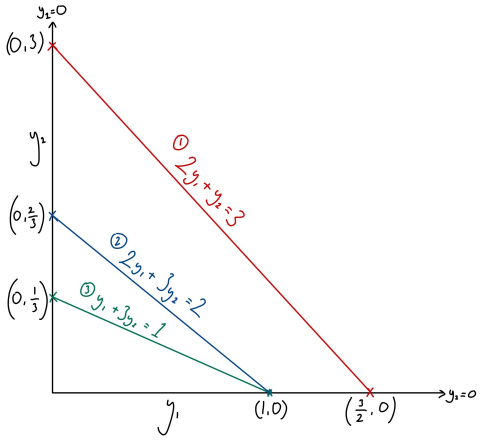


Figure 1: Constraint Equations Plotted

The constraint equations in the dual problem define that the area under the respective constraint lines is deemed infeasible. This information coupled with a visual inspection of Figure 1 shows that constraints 2 and 3 (as defined on the graph) are redundant constraints.

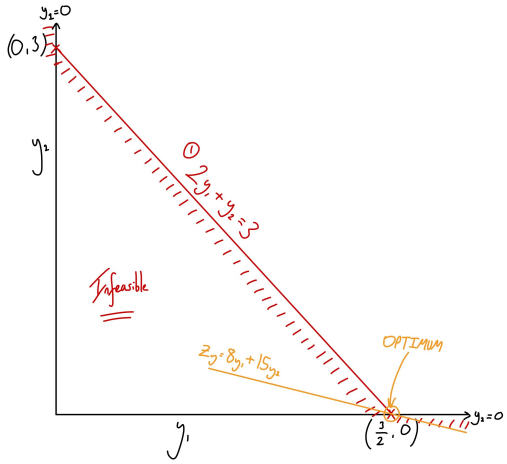


Figure 2: Useful Constraint Equations and Objective Function Plotted in Optimum Position

Figure 2 shows the one remaining feasible constraint plotted along side the objective function in its optimum position. The optimum position can be determined from the objective function's gradient. Since the gradients aren't equal, the optimum point will be at one of the vertices of the feasible region. As, in this instance, the gradient of the objective function is less than the gradient of the constraint, the optimum point to achieve the minimum value for z_y is at the bottom right of the graph where the constraint line meets the y_1 axis. A simple calculation shows this point is: $y_1 = \frac{3}{2}$ and $y_2 = 0$. Substituting these values back into the dual objective function:

$$\begin{aligned} z_y &= 8y_1 + 15y_2 \\ &= 8 \times \frac{3}{2} + 15 \times 0 \\ &= 12 \end{aligned}$$

The dual problem has returned the same optimal function value as the primal, therefore verifying the strong duality theorum.

1.d

Complementary slackness theorem states that if $\mathbf{x}$ and $\mathbf{y}$ are feasible solutions to a primal-dual pair of linear programming problems, then both are optimal if and only if:

$$\mathbf{y}^T(\mathbf{A}\mathbf{x}-\mathbf{b})=0$$
$$\text{and}$$
$$\mathbf{x}^T(\mathbf{A}^T\mathbf{y}-\mathbf{f})=0$$

Testing the first condition:

$$\begin{aligned}\mathbf{y}^T(\mathbf{A}\mathbf{x}-\mathbf{b}) &= \begin{bmatrix}\frac{3}{2} & 0\end{bmatrix}\left(\begin{bmatrix}2 & 2 & 1\\ 1 & 3 & 3\end{bmatrix}\begin{bmatrix}4\\0\\0\end{bmatrix}-\begin{bmatrix}8\\15\end{bmatrix}\right)\\ &= \begin{bmatrix}\frac{3}{2} & 0\end{bmatrix}\left(\begin{bmatrix}8\\4\end{bmatrix}-\begin{bmatrix}8\\15\end{bmatrix}\right)\\ &= \begin{bmatrix}\frac{3}{2} & 0\end{bmatrix}\begin{bmatrix}0\\-11\end{bmatrix}\\ &= 0\end{aligned}$$

Testing the second condition:

$$\begin{aligned}\mathbf{x}^T(\mathbf{A}^T\mathbf{y}-\mathbf{f}) &= \begin{bmatrix}4 & 0 & 0\end{bmatrix}\left(\begin{bmatrix}2 & 1\\ 2 & 3\\ 1 & 3\end{bmatrix}\begin{bmatrix}\frac{3}{2}\\0\end{bmatrix}-\begin{bmatrix}3\\2\\1\end{bmatrix}\right)\\ &= \begin{bmatrix}4 & 0 & 0\end{bmatrix}\left(\begin{bmatrix}3\\3\\\frac{3}{2}\end{bmatrix}-\begin{bmatrix}3\\2\\1\end{bmatrix}\right)\\ &= \begin{bmatrix}4 & 0 & 0\end{bmatrix}\begin{bmatrix}0\\1\\\frac{1}{2}\end{bmatrix}\\ &= 0\end{aligned}$$

1.e

A primal problem composed of n decision variables and two inequality constrains will convert into a dual problem with two decision variables and n inequality constraints. My initial thoughts were that a reduction in dimensions from n to 2 would lead to a much lower processing time. However the back end of linprog will convert all inequalities to standard equations using slack variables. Since there are now n constraints there will be n slack variables meaning the number of dimensions in play hasn't gone down at all, instead it has risen by 2.
The second point I'd like to discuss is that n is large. This means that there will be a large number of inequality constraints and subsequently lead to a high volume of redundancies. linprog has no way of determining if a constraint equation is redundant or not and will consider all constraints evenly. This means solving this problem the way will lead to a large amount of computational power bring allocated to the definition of redundant constraints.

2 Integer Linear Programming

2.a

(2a and 2b were done together)

2.b

Problem Definition This problem looks to optimise group allocations of a pool of students to a selection of problems. If we have a p students and q projects, there will be pq decision variables. A decision variable will be shown as x_{ij}. All decision variables will be binary and they correlate to whether or not student i has been selected for project j (i.e $x_{ij} = 1$ if selected and $x_{ij} = 0$ if not)

Constraints There are two governing constraints in this problem. The first is that each student can only be allocated to a single project and the second is that the number of students allocated to a project must be between the maximum and minimum group size. Let's express these constraints algebraically:

$$\sum_{j=1}^{q} x_{ij} = 1 \tag{1}$$

$$g_{min} \leq \sum_{i=1}^{p} x_{ij} \leq g_{max} \tag{2}$$

Building the Constraint Matrices in MatLab The best way to visualise how the constraint matrices are built is to demonstrate it in a smaller 'dummy' data set. Let's consider a scenario where 4 students are applying to 2 projects. This can be represented in a table as seen in Table 2.b

Visualisation of the layout of decision variables

	Pr_1	Pr_2
Su_1	x_{11}	x_{12}
Su_2	x_{21}	x_{22}
Su_3	x_{31}	x_{32}
Su_4	x_{41}	x_{42}

1D layout of decision variables

$\text{Su}_1 Pr_1$	x_{11}
$\text{Su}_1 Pr_2$	x_{12}
$\text{Su}_2 Pr_1$	x_{21}
$\text{Su}_2 Pr_2$	x_{22}
$\text{Su}_3 Pr_1$	x_{31}
$\text{Su}_3 Pr_2$	x_{32}
$\text{Su}_4 Pr_1$	x_{41}
$\text{Su}_4 Pr_2$	x_{42}

One of the initial steps in the back-end of the intlinprog function is converting the matrix of decision variables into a column vector. This means each row of the constraint matrix will need to be the same length as the number of decision variables (in this instance pq). Each row in the constraint matrix will consist of 0's and 1's depending on which decision variables you want to 'activate' to consider within that constraint. To identify which constraint coefficients are active we need to look at how the geometry of active decision variable matrix relates to the sequence of active decision variables in the 1D array.

Equality Constraint Constraint 1 (Equation 1) states that each student can be allocated to exactly one project. It is an equality constraint. Geometrically this means each row in the decision variable matrix sums to 1. Let's express this as firstly a series of equations followed by its constraint matrix representation.

Equation Representation	Matrix Representation

$$\begin{aligned} x_{11} + x_{12} &= 1 \\ x_{21} + x_{22} &= 1 \\ x_{31} + x_{32} &= 1 \\ x_{41} + x_{42} &= 1 \end{aligned} \qquad \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_{11} \\ x_{12} \\ x_{21} \\ x_{22} \\ x_{31} \\ x_{32} \\ x_{41} \\ x_{42} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

The constraint matrix is built by setting the kth set of q numbers to 1, where k is the row number and q is the number of projects (so in this instance you want the first set of two on the first row, the second set of two on the second row etc..). The constraint matrix will always have the dimensions pxpq, representing a constraint for each of the p students and how each constraint interacts with the pq decision variables. Below looks at how the constraint matrix and constraint limits were coded in MatLab.

```
equality_constraint = zeros(p,pq); %Initialise equality constraint matrix
for i = 1:p
    for k = 1:q
        equality_constraint(i,k+(q*(i-1))) = 1; %Add a sequence of values to 1. The position of thevvalues changed is a functionv
        %of row and column
    end
end
equality_constraint_limits = ones(p,1); %Initialise a column vector for the equality constraint limits.
```

This initialises a pxpq zeros matrix and line by line changes a series of q numbers on each line to 1. The series that is changed shifts to the right by an amount of q from one row of the array to the next. The equality constraint matrix that is created will be the Aeq variable of intlinprog [2] and the 1D array of 1's will be beq.

Inequality Constraint Constraint 2 (Equation 2) states that the number of students in each group must be greater than or equal to the minimum group size and less than or equal to the maximum group size. Since there are two inequalities in this equation, it will need to be split into two constraints. The first subsequent constraint will need to be inverted (multiply both sides by -1) so the inequality signs are consistent between constraints as intlinprog inequality constraints are less than or equal to ($\leq$) the constraint limits by default. [2]

Initial Constraint	Constituent Constraints
$g_{min} \leq \sum_{i=1}^{p} x_{ij} \leq g_{max}$	$\sum_{i=1}^{p} x_{ij} \leq g_{max}$
	$-\sum_{i=1}^{p} x_{ij} \leq -g_{min}$

To build the constraints matrix, the same approach will be used as it was for constraint 1. Geometrically, $\sum_{i=1}^{p} x_{ij}$ means the sum of each column. As before, let's express this as equations and matrices.

Equation Representation	Matrix Representation

$$\begin{aligned} x_{11} + x_{21} + x_{31} + x_{41} &\leq g_{max} \\ x_{12} + x_{22} + x_{32} + x_{42} &\leq g_{max} \\ -x_{11} - x_{21} - x_{31} - x_{41} &\leq -g_{min} \\ -x_{12} - x_{22} - x_{32} - x_{42} &\leq -g_{min} \end{aligned} \qquad \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ -1 & 0 & -1 & 0 & -1 & 0 & -1 & 0 \\ 0 & -1 & 0 & -1 & 0 & -1 & 0 & -1 \end{bmatrix} \begin{bmatrix} x_{11} \\ x_{12} \\ x_{21} \\ x_{22} \\ x_{31} \\ x_{32} \\ x_{41} \\ x_{42} \end{bmatrix} \leq \begin{bmatrix} g_{max} \\ g_{max} \\ -g_{min} \\ -g_{min} \end{bmatrix}$$

After a few iterations it was spotted that each constituent constraint matrix can be built by creating an identity matrix of size q and repeating it p times, obviously taking the negative for the lower bound constraint. They are then combined for the global inequality constraint matrix. Let's see how this was developed in MatLab.

```
inequality_constraint = repmat(eye(q),1,p);%Initialise a matrix comprised of a q-sized identity matrix repeated q times
inequality_constraint = [inequality_constraint ; -(inequality_constraint)]; %Add a the negative version of this matrix to the bottom
inequality_constraint_limits=[(ones(q,1)*g_max);(ones(q,1)*g_min*-1)]; %Initialise an inequality constraint limit column vector. This is
%comprised of q, g_max values and q, -g_min values.
```

2.c

The inputs for intlinprog are almost identical to that of linprog which can be found in Section 1.b.1.
Syntax:

$$[\text{x,fval,exitflag}] = \text{linprog(f,intcon,A,b,Aeq,beq,lb,ub)}$$

This solves:

$$\min f^T x \text{ such that } \begin{cases} \text{x(intcon) are integers} \\ A \cdot x \leq b, \\ Aeq \cdot x = beq, \\ lb \leq x \leq ub. \end{cases}$$

intcon intcon is a row vector that defines which of the decision variables have to be positive integers. [2] In this instance all decision variables have to be positive integers. Let's see how this was programmed in MatLab.

```
intcon = 1:pq;
```

Implementing intlinprog The objective function (c_{ij}) is a 63x14 matrix given in the coursework brief. This means $p = 63$ and $q = 14$. Each student gives a project a score from 1-14 indicating a preference order where 1 is the most desirable and 14 is the least desirable. That means this is a standard minimisation problem with the objective function:

$$\min_z = \frac{1}{p} \sum_{i,j} c_{ij} x_{ij}$$

The $\frac{1}{p}$ term is so the operator can see the average student prefence allocation. One thing that needs to be checked is that the objective matrix needs to be in the right format i.e it needs to be at the right orientation so it is a *pxq* matrix as opposed to a *qxp* matrix. All the required information is now present to run intlinprog.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ .. \\ x_{882} \end{bmatrix} \qquad f = c_{ij} \qquad \text{intcon See paragraph 2.c}$$

A & b See paragraph 2.b Aeq & beq See paragraph 2.b $lb = \begin{bmatrix} 0 \end{bmatrix}_{pq\text{x}1}$

$$ub = \begin{bmatrix} \end{bmatrix}$$

```
[x,fval,exitflag] = intlinprog(transpose(data),intcon,inequality_constraint,inequality_constraint_limits,equality_constraint,
                  equality_constraint_limits,zeros(pq,1),[]);
```

Inputs	Outputs
data = ProblemRankData.csv	x = Large column vector of 1's and 0's
$g_{min} = 3$	fval = 83
$g_{max} = 5$	exitflag = 1
	average = $\frac{fval}{p} = 1.3175$

2.d

Part d looks at adding an additional constraint which insures that each student gets their r^{th} preference where r is a small integer. To do this the same dimensions that were used to derive the constraint matrices in Sections 2.a & 2.b.

Firstly, let's explain this equation algebraically and then see how this equation applies to the 4x2 'dummy' data.

Equation Representation	Matrix Representation

$$\begin{aligned} c_{11}x_{11} + c_{12}x_{12} &\leq r \\ c_{21}x_{21} + c_{22}x_{22} &\leq r \\ c_{31}x_{31} + c_{32}x_{32} &\leq r \\ c_{41}x_{41} + c_{42}x_{42} &\leq r \end{aligned} \qquad \begin{bmatrix} c_{11} & c_{12} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & c_{21} & c_{22} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_{31} & c_{32} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & c_{41} & c_{42} \end{bmatrix} \begin{bmatrix} x_{11} \\ x_{12} \\ x_{21} \\ x_{22} \\ x_{31} \\ x_{32} \\ x_{41} \\ x_{42} \end{bmatrix} \leq \begin{bmatrix} r \\ r \\ r \\ r \end{bmatrix}$$

From this it is clear to see that in order to create the constraint matrix for project preference, the objective matrix must be considered and each row must be added to the global inequality constraint matrix in a staggered fashion. Let's assemble the final inequality constraint matrix and inequality limit column vector for the dummy 4x2 data set.

$$\begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ -1 & 0 & -1 & 0 & -1 & 0 & -1 & 0 \\ 0 & -1 & 0 & -1 & 0 & -1 & 0 & -1 \\ c_{11} & c_{12} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & c_{21} & c_{22} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & c_{31} & c_{32} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & c_{41} & c_{42} \end{bmatrix} \begin{bmatrix} x_{11} \\ x_{12} \\ x_{21} \\ x_{22} \\ x_{31} \\ x_{32} \\ x_{41} \\ x_{42} \end{bmatrix} \leq \begin{bmatrix} g_{max} \\ g_{max} \\ -g_{min} \\ -g_{min} \\ r \\ r \\ r \\ r \end{bmatrix}$$

MatLab Code Let's have a look at the code generating this constraint.

```
r = 3 %Initialise r
r_constraint_limit = ones(p,1)*r; %Initialise constraint limit column vector
r_constraint = []; %Initialise constraint matrix
for i = 1:p
    r_constraint_row = [zeros(1,(q*(i-1))) (data(i,:)) zeros(1,q*(p-i))]; %Add a sequence of 0's, preference data and further zeros.
    %Sequence is a function of row number
    r_constraint = [r_constraint ; r_constraint_row]; %Add constraint row to the over all r constraint matrix
end

inequality_constraint = [inequality_constraint;r_constraint]; %Merge r constraint matrix with golbal inequality constraint matrix
inequality_constraint_limits = [inequality_constraint_limits;r_constraint_limit]; %Merge r constraint limits with golbal inequality
%constraint limits
```

2.e

This section looks at how altering the set-up procedure influences the quality of the output.

Variation of g_{max} and g_{min} The majority of combinations of g_{max} and g_{min} can be discounted immediately. There are three key constraints that eliminate most combinations:

$$g_{min} \leq g_{max} \tag{3}$$

$$g_{min} \leq \frac{p}{q} \leq g_{max} \tag{4}$$

$$g_{min} \neq g_{max} \; \forall \; \frac{p}{q} \notin \mathbb{Z} \tag{5}$$

Equation 3 is fairly intuitive, simply saying the maximum group size must be greater than or equal to the minimum group size. Equation 4 is less intuitive. Put simply, if g_{min} exceeds $\frac{p}{q}$, there won't be enough students to fill all the groups and if g_{min} is strictly less that $\frac{p}{q}$, there will be students who can't be allocated projects. Finally, Equation 5 states that if $\frac{p}{q}$ isn't an integer i.e if you can't form a perfect set of uniform sized groups, g_{max} cannot equal g_{min}. Average group allocation scores were calculated for all the remaining combinations.

		g_{max}										
		4	5	6	7	8	9	10	11	12	13	14
g_{min}	1	-	1.3016	1.1587	1.0952	1.0476	1.0317	1.0159	1.0159	1.0159	1.0159	1.0159
	2	-	1.3016	1.1746	1.1270	1.0794	1.0635	1.0635	1.0635	1.0635	1.0635	1.0635
	3	-	1.3175	1.2540	1.2063	1.1905	1.1905	1.1905	1.1905	1.1905	1.1905	1.1905
	4	-	1.4127	1.3651	1.3492	1.3492	1.3492	1.3492	1.3492	1.3492	1.3492	1.3492
	5	-	-	-	-	-	-	-	-	-	-	-

Table 1: Optimal average allocation preference for all feasible combinations of g_{max} and g_{min}

The value calculated in Section 2.c is highlighted in blue and the optimal values for g_{max} and g_{min} are highlighted in green. Unsurprisingly these values are found where the model is the least constrained. However there is another, more interesting, observation here. Looking at each row, the average score converges to a point where increasing g_{max} no longer reduces the average score. The value of g_{max} at which this occurs reduces as you increase the value of g_{min} in a staggered fashion like so:

	Convergence Point			
g_{min}	1	2	3	4
g_{max}	10	9	8	7

Table 2: Optimal allocation convergence point

This trend makes sense. As you increase the minimum group size, the flexibility of student allocation decreases and once each group has reached g_{min} students, the number of students remaining to be allocated decreases by q, explaining the linear nature of this pattern.

Variations in 'r' As explained in Section 2.d, the r constraint is an inequality constraint which ensures each student gets their r^{th} preference. Let's see how variations in r effect the average allocation score when $g_{max} = 5$ and $g_{min} = 3$.

r	1	2	3	4	5	...	14
fval/p	No feasible solution	No feasible solution	No feasible solution	1.3175	1.3175		1.3175

Table 3: How variance in r affects optimal average allocation score

Table 3 shows that r is currently a very ineffective constraint on this data set due to the fact it either over-constrains the model leaving no feasible solutions, or has no effect on the results. This happens here as the optimal allocation exists at the point at which the r value no longer over-constrains the model. If the data set were larger, this value may be more useful. Let's test this hypothesis.

How larger data sets affect variations in 'r' The first step is to create larger data sets. The first group of data sets that'll be created will have the same number of projects but the number of students will be multiplied by 2,3& 4. It is important that all constraints, limits and variables are adjusted accordingly here. All inequality and equality constraints are created as functions of p and q and will therefore require no extra work. However g_{min} and g_{max} are currently hard coded to 3 and 5 respectively. In order to ensure the only variation that is being examined is the number of students, a new method has to be devised which can map the number the number of students to the group size limits. The proposed solution is a simple one. Firstly calculate the average group size with the calculation $\frac{p}{q}$, then add one and round to nearest number to get g_{max} and subtract one and round to get g_{min}. The code for this can be seen bellow.

```
average_group_size = p/q;
g_max = round(average_group_size+1);
g_min = round(average_group_size-1);
```

				r value				
		g_{max}	g_{min}	1	2	3	4	5
Number of students	63	4	6	No feasible solution	1.1429	1.1429	1.1429	1.1429
	126	8	10	No feasible solution	1.0952	1.0952	1.0952	1.0952
	189	13	15	No feasible solution	1.0899	1.0899	1.0899	1.0899
	252	17	19	No feasible solution	1.1032	1.1032	1.1032	1.1032
	630	44	46	No feasible solution	1.0492	1.0492	1.0492	1.0492

Table 4: Effect of 'r' on larger data sets.

Table 4 shows that r still has no effect on the results even when the number of students is varied. Ultimately this is due to the fact that intlingprog and this r constraint are trying to achieve the same thing which is why the minimum r-value at which a feasible solution exists as the optimum solution occur at the same point. It is important to note that as the data sets are generated randomly there is no pattern between the values on each row. The only behaviour that can be evaluated is how the average allocation score changes as r increases.

Varying powers of the cost matrix (c_{ij}) An important distinction that needs to be clarified is what is meant by putting the cost matrix to a certain power. This analysis will look at varying degrees of the Hadamard product,[10] otherwise known as the element-wise product. Now let's explore how variations in the power of the cost matrix affects the results. For this piece of analysis, the r constraint that was discussed previously has been removed. Initial thoughts on this are that the higher powers will penalise the lower picks (3rd and 4th choices) to a much higher degree than the lower powers. This will cause there to be fewer allocations to the lower selections *aswell* as a reduction in the number of first choice allocations. This will happen because, from the perspective of minimising the objective function, shifting a 4th choice allocation to a 3rd choice, even if this leads to a few 1st choice allocations shifting to 2nd choice, will still reduce the optimal function value due to the heavy penalisation of lower choice allocations. Let's run the various powers of the cost matrix through the intlinprog and see how student allocation varies.

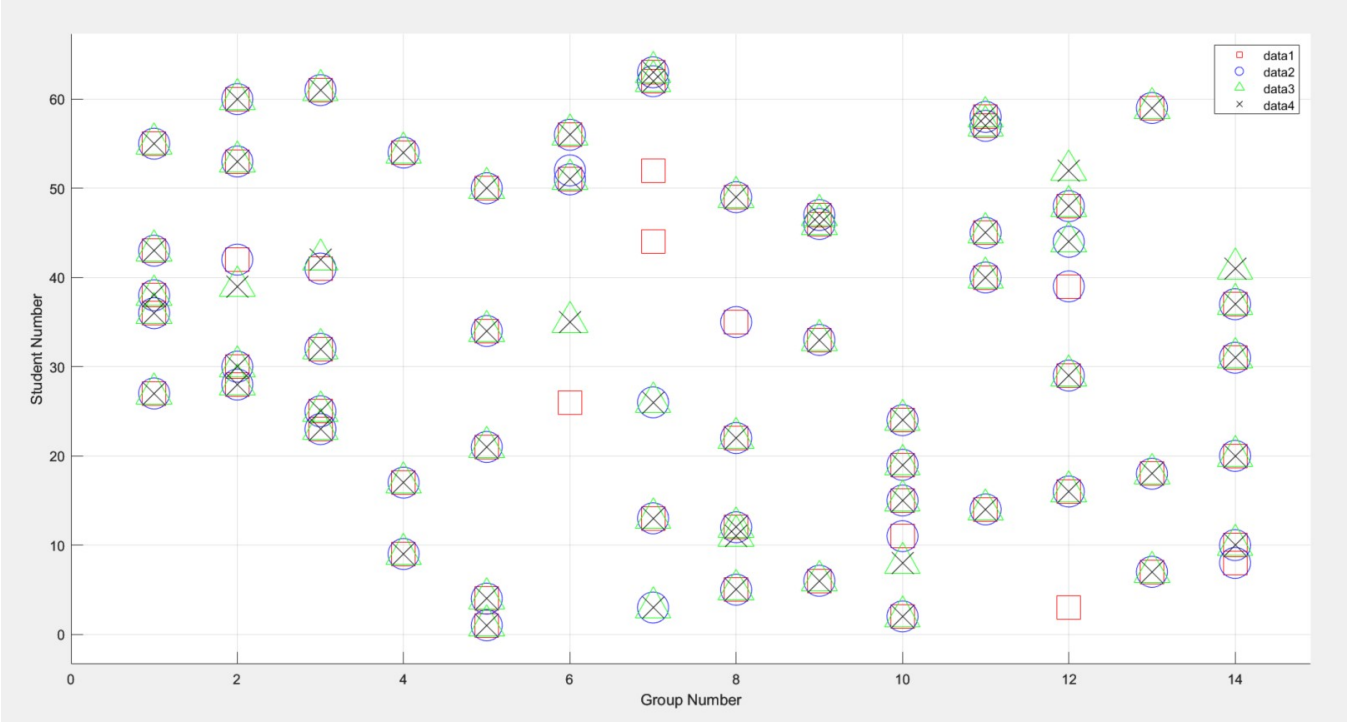


Figure 3: Student allocations of c_{ij}, c_{ij}^2, c_{ij}^3, c_{ij}^4

Figure 3 is quite a busy diagram so let's break it down. Firstly, each symbol represents the allocation of that student to that project. The different symbols represent the allocations of the various powers of c_{ij}. The allocations have been superimposed so that differences can be spotted easily. The first and most obvious observation is that there are differences in the allocations. This is what will be explored now. The second and less obvious observation is that there are no differences between the allocations for c_{ij}^3 and c_{ij}^4 as can be seen in Figure 4. For this reason c_{ij}^4 will be discounted from any further analysis

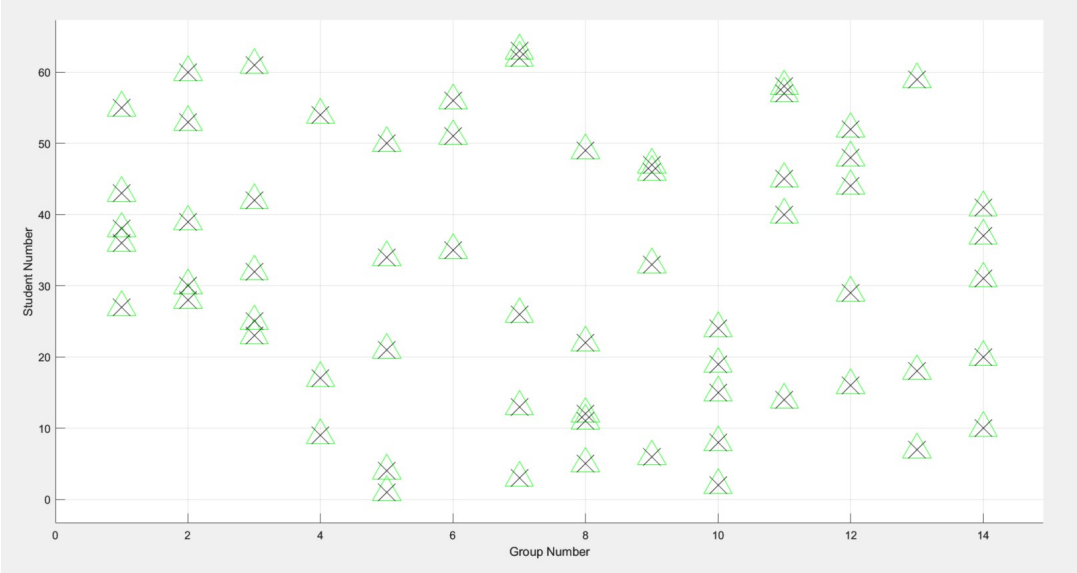


Figure 4: Student allocations of c_{ij}^3 c_{ij}^4

Let's look at the allocation data from the perspective of allocation frequencies.

		Allocation Frequencies			
		1st	2nd	3rd	4th
Cost Matrices	c_{ij}	47	13	2	1
	c_{ij}^2	47	13	2	1
	c_{ij}^3	45	16	1	1

Table 5: Allocation frequencies for varying powers of c_{ij}

Table 5 verifies the hypothesis laid out in paragraph 2.e. The values in green show that the distribution of the allocations have shifted towards centre due to the significantly higher penalisation for lower allocations. The code for how these values were calculated is seen below

```
clc
clear

data = readtable('ProblemRankData.csv');
data = data{:,:};
original_data = data;
size = size(data);
%Initialise parameters
p = size(1,1);
q = size(1,2);
pq = p*q;
g_max = 5;
g_min = 3;

%Compute equality constraints
equality_constraint = zeros(p,pq);
for i = 1:p
    for k = 1:q
        equality_constraint(i,k+(q*(i-1))) = 1;

    end
end
equality_constraint_limits = ones(p,1);

%Compute inequality constraints
inequality_constraint = repmat(eye(q),1,p);
inequality_constraint = [inequality_constraint ; -(inequality_constraint)];
inequality_constraint_limits=[(ones(q,1)*g_max);(ones(q,1)*g_min*-1)];

%Compute intcon
intcon = 1:pq;

%Insert intlinprog into a for loop that goes over each power for the cost matrix
for j = 1:3
    data = original_data.^j;
    [x,fval,exitflag] = intlinprog(transpose(data),intcon,inequality_constraint,inequality_constraint_limits,equality_constraint,
                        equality_constraint_limits,zeros(pq,1),[]);
    %Reshape decision variables into the correct format.
    x = reshape(x,q,p);
    x = x.';
```

```
    average = (fval/p) %Objective value
    clear size

    %These for loops check the allocation of the decision variables and sets all unselected decision variable positions to
    %zero in the cost matrix. This allows for a visual inspection of allocations for a sanity check (all values should be small)
    for i = 1:p
        for k = 1:q
            if x(i,k) == 0
                data(i,k) = 0;
            end
        end
    end
    %This displays the frequency of 1st, 2nd, 3rd and 4th allocations based off the remaining values in the cost matrix
    allocationSplit = [sum(data(:) == 1) sum(data(:) == 2^j) sum(data(:) == 3^j) sum(data(:) == 4^j)];
    disp(allocationSplit)

end
```

3 Nonlinear Optimisation — Quadratic Programming

3.a

Objective Function:

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

x and y unconstrained

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

$$\frac{df}{dy} = 10y - 8$$

Minimum point therefore $\frac{df}{dy} = 0$

$$0 = 10y^* - 8$$

$$y^* = \frac{4}{5}$$

$$f(x,y^*) = 2x^2 + 5 \cdot \left(\frac{4}{5}\right)^2 - 2x \cdot \left(\frac{4}{5}\right) - 2x - 8 \cdot \frac{4}{5}$$

$$f(x,y^*) = 2x^2 - \frac{18}{5}x - \frac{16}{5}$$

$$\frac{df}{dx} = 4x - \frac{18}{5}$$

Minimum point therefore $\frac{df}{dx} = 0$

$$0 = 4x^* - \frac{18}{5}$$

$$x^* = \frac{9}{10}$$

$$(x^*, y^*) = \left(\frac{9}{10}, \frac{4}{5}\right)$$

3.b

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

$$\nabla\left[f(x,y)\right] = \begin{pmatrix} 4x - 2y - 2 \\ 10y - 2x - 8 \end{pmatrix}$$

$$\left(x^{(0)}, y^{(0)}\right) = (0,0)$$

$$\nabla\left[f\left(x^{(0)}, y^{(0)}\right)\right] = \begin{bmatrix} -2 \\ -8 \end{bmatrix}$$

This is the initial step direction. Now we need to calculate the step magnitude (λ)

$$\max f\left(\begin{bmatrix} x^{(0)} \\ y^{(0)} \end{bmatrix} + \lambda \nabla\left[f\left(x^{(0)}, y^{(0)}\right)\right]\right)$$

$$\max f\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix} + \lambda \begin{bmatrix} -2 \\ -8 \end{bmatrix}\right)$$

$$\max f\left(\begin{bmatrix} -2\lambda \\ -8\lambda \end{bmatrix}\right)$$

$$f(-2\lambda, -8\lambda) = 8\lambda^2 + 320\lambda^2 - 32\lambda^2 + 4\lambda + 64\lambda$$

$$= 296\lambda^2 + 68\lambda$$

$$\max\ (296\lambda^2 + 68\lambda)$$

$$\frac{df}{d\lambda} = 592\lambda + 68$$

$$\text{Maximum point therefore } \frac{df}{d\lambda} = 0$$

$$\lambda^* = -\frac{17}{148}$$

$$\begin{bmatrix} x^{(1)} \\ y^{(1)} \end{bmatrix} = \begin{bmatrix} x^{(0)} \\ y^{(0)} \end{bmatrix} + \lambda \nabla [f(x^{(0)}, y^{(0)})]$$

$$\begin{bmatrix} x^{(1)} \\ y^{(1)} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} - \frac{17}{148} \begin{bmatrix} -2 \\ -8 \end{bmatrix}$$

$$\left(x^{(1)}, y^{(1)}\right) = \left(\frac{17}{74}, \frac{34}{37}\right)$$

$$\left(x^{(1)}, y^{(1)}\right) = (0.2297, 0.9189)$$

3.c

The inputs for quadprog [5] are almost identical to that of linprog which can be found in Section 1.b.1.
Syntax:

$$\text{[x,fval,exitflag] = quadprog(H,f,A,b,Aeq,beq,lb,ub)}$$

This solves:

$$\min\ \tfrac{1}{2}x^T H x + f^T x \text{ such that } \begin{cases} A \cdot x \leq b, \\ Aeq \cdot x = beq, \\ lb \leq x \leq ub. \end{cases}$$

Since there are no constraints:

$$A = b = Aeq = beq = lb = ub = [\,]$$

$$\text{For: } f(x, y) = 2x^2 + 5y^2 - 2xy - 2x - 8y$$

$$H = \begin{bmatrix} 4 & -2 \\ -2 & 10 \end{bmatrix}$$

$$f = \begin{bmatrix} -2 \\ -8 \end{bmatrix}$$

$$\text{This returns: } \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Let's see how this was developed in code.

```
H = [4 -2;-2 10];
f = [-2;-8];
[x,fval,exitflag] = quadprog(H,f)
```

3.d

Figure 5 shows a plot of the surface $f(x, y)$.

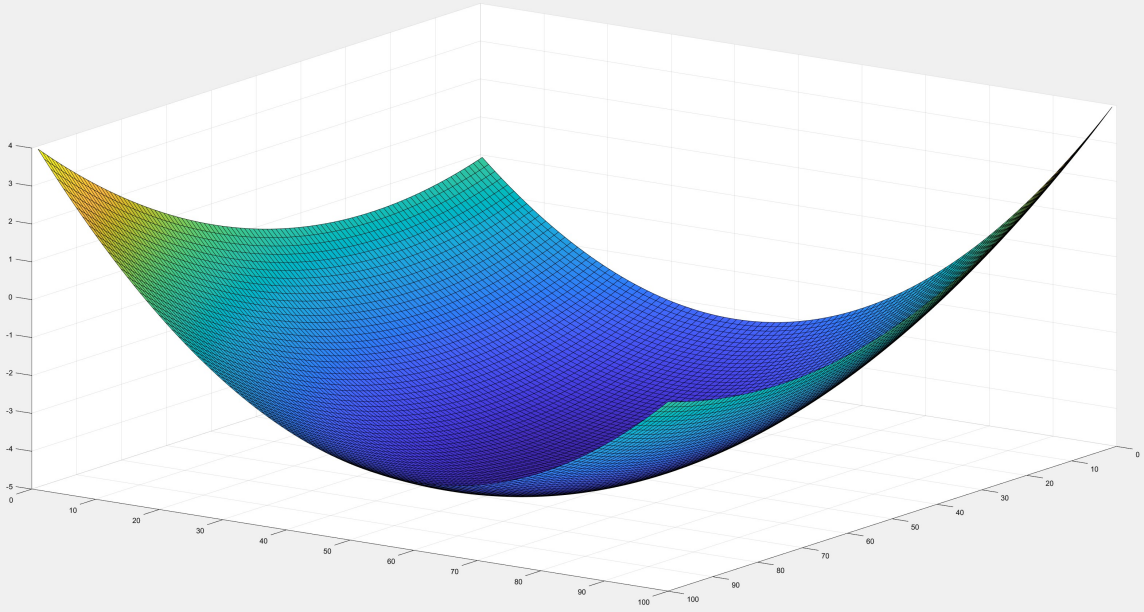


Figure 5: Surface plot of $f(x, y)$

Now the basic geometry has been established, let's look at $f(x, y)$ as a contour plot in the same domain with the correct scaling. On this contour plot the coordinates calculated in parts i, ii and iii will be presented.

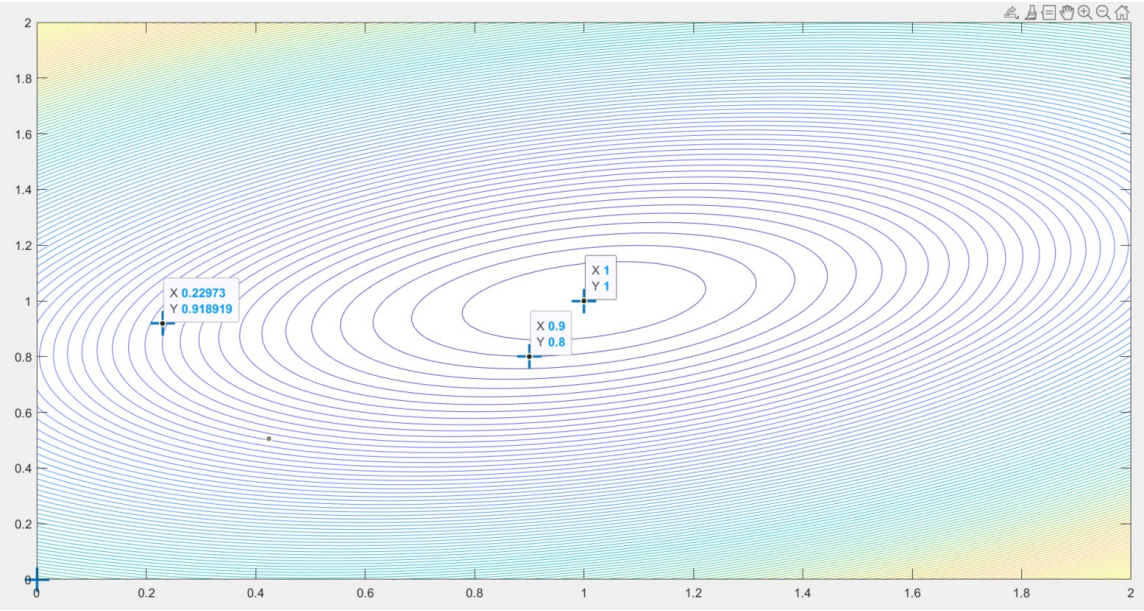


Figure 6: Contour plot of $f(x, y)$

Starting with the set of coordinates calculated in this Section (3.a). These were calculated by freezing variables individually and working out the stationary point one dimension at a time. By overlaying this result onto the plot in Figure 6 it has been shown that this primitive method serves as good estimator for the true minimum point but, due to its inaccuracy, nothing more.

The next observation that can be explained is the geometrical relationship between the origin $\left(x^{(0)}, y^{(0)}\right)$ and the point (0.2297,0.9189) $\left(x^{(1)}, y^{(1)}\right)$. The set of coordinates calculated in Section 3.b were the product of the first iteration of the steepest-descent algorithm, with a starting point at the origin. An iteration of the steepest-descent algorithm can be divided into two steps, the first is determining the direction of travel. As the name suggests, the steepest direction will always be chosen, this is given by the gradient of the function at the current point, which is calculated by taking the dot product of the coordinates with the derivative vector (∇). On a contour plot this would be visualised as travelling in an orthogonal direction to that of the contour at that point. The second step is

determining the magnitude of the direction travel. This is embodied in the scalar value, λ. To determine this value, the travel vector is computed in terms of λ and maximised by differentiating with respect to λ. The magnitude of λ represents the point at which the calculated direction of travel is no longer toward but rather away from the minimum point. On a contour plot this is represented as the point at which the travel vector is tangential to a contour as seen in Figure 7. This means sequential vectors in this algorithm will be orthogonal to one another. Figure 8 shows these ideas in more detail by estimating the outputs of the next three iterations for the steepest descent algorithm for the same scenario laid out in Figure 6.

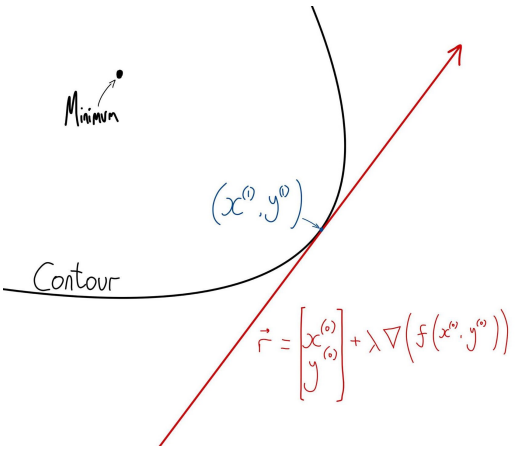


Figure 7: Contour representation of λ

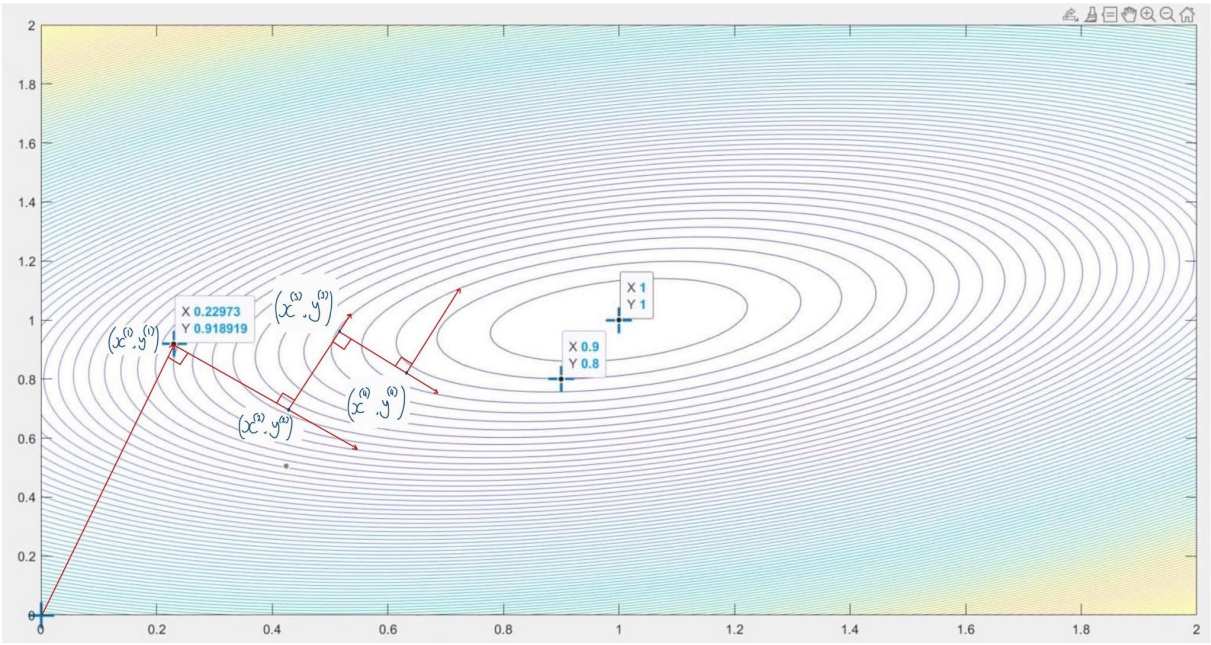


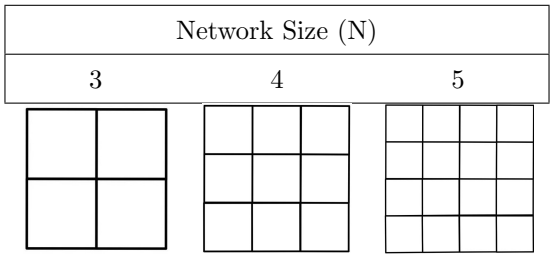
Figure 8: Contour representation of λ

Finally, The set of coordinates calculated in Section 3.c. This is the simplest observation. This graph verifies the work done by quadprog [5]. By carrying out a large number iterations of the steepest descent algorithm it ultimately returned the correct global minimum at the point (1,1).

3.e

Kirchoff Solver. The problem at hand is to monitor the effective resistance of a square lattice 'Manhattan' resistor network as its dimensions change. The first step is to develop a network of nodes and edges using MatLab's build in graph function.

Constructing Node Network. The objective is to assemble node networks in the following pattern for varying positive integer values of N:



The first step was to establish a consistent node labelling system so a pattern can determined a and building method can be devised as a function of N. The bottom left node is node 1 and the top right node is node N. Figure 9 looks at the node labelling system that was chosen.

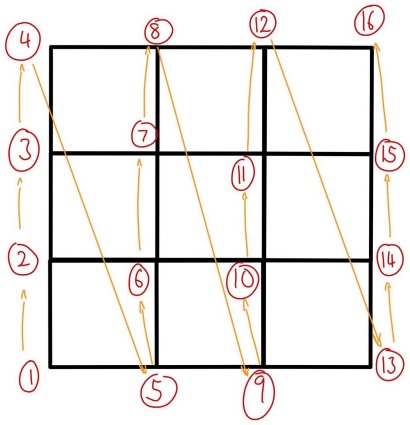


Figure 9: Node labelling system applied to the N=4 system.

From here, the necessary edges were added to each node sequentially, ensuring that no duplicate edges were created. Once this had been done a clear pattern for the number of edges emerging from each node started to form. This is shown in Figure 10.

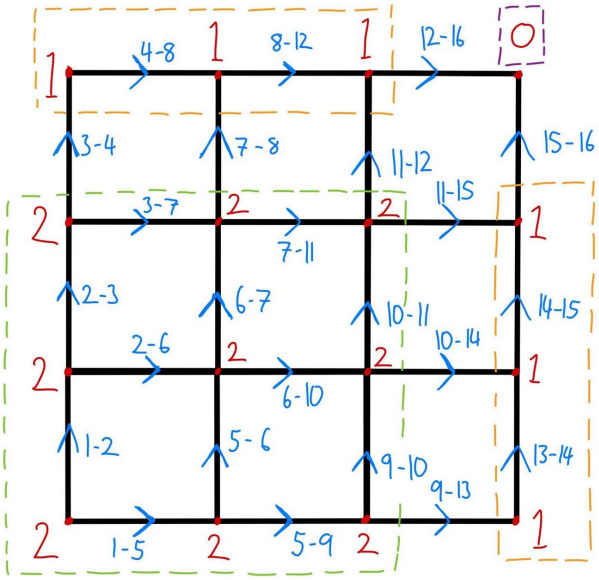


Figure 10: Node labelling system applied to the N=4 system.

The pattern that emerged was that the top right hand node (node N^2) has zero out going nodes. The nodes on the top and right set of edges had one outgoing node each and the remaining block of $(N-1)^2$ nodes have two out going edges. The out going edge for nodes on right hand side are vertical meaning they're moving from node i to $i+1$, for the nodes on the top face, emerging edges are moving horizontally meaning they're moving from node i to $i+N$ and the central nodes have a vertical and horizontal edge meaning there are two edges going from node i to $i+1$ and $i+N$ respectively. The final piece of the puzzle is the edge weightings. These are given by the resistances which are all equal to 1Ω. Now the patterns have been established, let's see how they were implemented in MatLab:

```
N = ;
%A network of size N has (N+1)^2 nodes
%Define the number of nodes and their IDs
NodeCount = N.^2;
NodeID = linspace(1,NodeCount,NodeCount);

%Aim is to get a node between each point in a manhatten network format
%Nodes can be split into groups, starting at
%the bottom right and progressing up through each column sequentially, each
%node sends either one or two edges (except the last which sends zero)
% This splits them into groupings in terms of N
topNodeID = linspace(N,NodeCount,N);
rightNodeID = linspace((NodeCount-N)+1,NodeCount,N);
centralNodeID = setdiff(NodeID,[rightNodeID topNodeID]);

%This uses the groupings with the geometrical relationship to their
%recipiants and makes the groupings of the recieving nodes
topConnectionNodeID = topNodeID(1,1:N-1) + N;
rightConnectionNodeID = rightNodeID(1,1:N-1) + 1;
horizontal_vertical_steps = [N 1];
centralConnectionNodeID = [];
for i=1:(N-1)^2
    centralConnectionNodeID = [centralConnectionNodeID [horizontal_vertical_steps+centralNodeID(1,i)]];
end
%These are then married up using graph(). All resistances = 1 aswell
sending = [sort([centralNodeID centralNodeID]) topNodeID rightNodeID];
sending(sending==NodeCount) = []; &Remove N^2 node as sending node
recieving = [centralConnectionNodeID topConnectionNodeID rightConnectionNodeID];
%Be careful that sending and recieving nodes aare aligned in their respective lists
resistances = ones(1,length(sending));
G = graph(sending,recieving,resistances);
```

Now the graph has been constructed, the node and edge data can be fed into the function *Kirchoffsolve.m*. This will return a solved network, with the current passing through each edge and the voltage at each Node. Let's see how this was implemented and plotted aswell as the results that were given.

```
edge_position = table2array(G.Edges(:,1));
edge_weight = table2array(G.Edges(:,2));
[X,V]=KirchoffSolve(edge_position,edge_weight,1,NodeCount);
%X = edge currents
%V = Node voltages
%Graph new data where weightings = current
G = graph(sending,recieving,X);
G.Edges.LWidths = 4*G.Edges.Weight/max(G.Edges.Weight);
G.Nodes.Voltages = V;
%Build network coordinates to it's plotted in a square shape.
x = []
y = []
for k = 1:N
    x = [x (zeros(1,N)+k)]
    y = [y 1:N]
end
p = plot(G,'XData',x,'YData',y,'LineStyle','-')
p.LineWidth = G.Edges.LWidths;
p.Marker = 'o';
p.NodeColor = 'r';
p.MarkerSize = 6;
p.NodeLabel = [];
```

Results returned using this code do not have node or edge values plotted as these become difficult to examine when the network grows in size. Instead the edge thicknesses correspond to their weighting which is, in this case, the current passing through that connection. Figure 11 shows a 7x7 network plotted with this script.

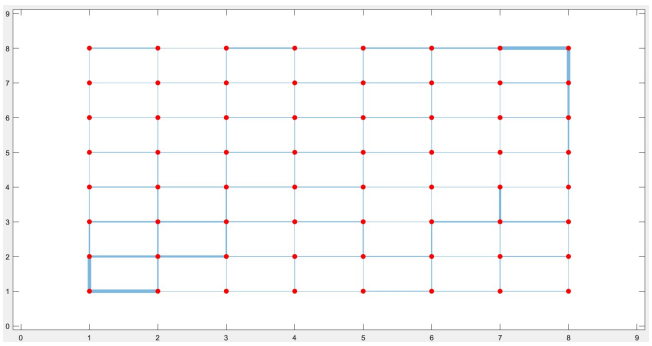


Figure 11: Calculated - N=8 System

Node and edge data can easily be added to the plot by slightly extending the plot function. However as mentioned previously, these clutter the graph and will become illegible when the network size grows.

```
...
p = plot(G,'XData',x,'YData',y,'EdgeLabel',G.Edges.Weight,'NodeLabel',G.Nodes.Voltages,'LineStyle','-')
```

This is what that looks like:

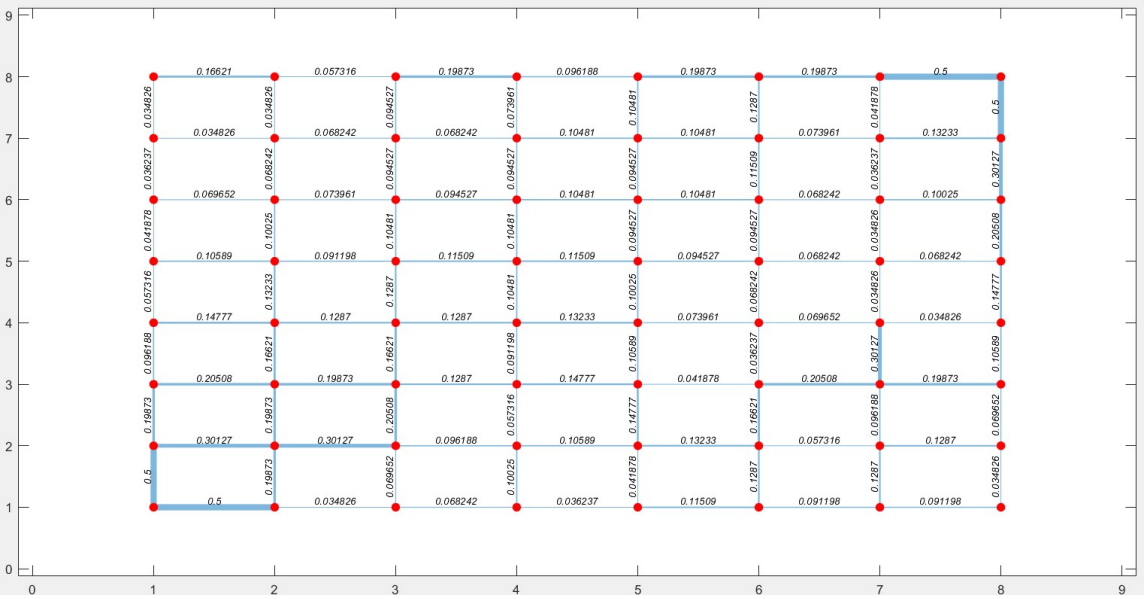


Figure 12: Calculated - 8x8 System with annotated nodes and edges

Investigating effective resistance As this network is comprised of ohmic resistors, the network resistance is simply given by:

$$R_{network} = \frac{V_{Cathode} - V_{Anode}}{I}$$
$$I = 1$$
$$V_{Anode} = 0$$
$$R_{network} = V_{Cathode}$$

This makes the network resistance very easy to determine from the code.

```
...
[X,V]=KirchoffSolve(edge_position,edge_weight,1,NodeCount);
...
network_resistance = max(V)
```

Let's investigate how network resistance varies for different values of N by inserting the whole script into a for loop that iterates through ascending values of N and recording the network resistance for each.

Network Size	2	3	4	5	6	7	8	9	10	11	12	13
Network Resistance (Ω)	1.500	1.857	2.136	2.366	2.560	2.729	2.878	3.012	3.133	3.243	3.345	3.439

Table 6: Effective resistance from 2x2 to 13x13 network

While looking at the effective resistances of each network, let's also start looking at the computation time. To ensure this is accurate, the *tic ... toc* function will be used, all graphing and plotting functions will be removed and the timer will stop the moment the network resistance is calculated. The code for this (minus previously seen notation) is below.

```
clc
clear

tic
N = ;

NodeCount = N.^2;
NodeID = linspace(1,NodeCount,NodeCount);

topNodeID = linspace(N,NodeCount,N);
rightNodeID = linspace(NodeCount+1-N,NodeCount,N);
centralNodeID = setdiff(NodeID,[rightNodeID topNodeID]);

topConnectionNodeID = topNodeID(1,1:N-1) + N;
rightConnectionNodeID = rightNodeID(1,1:N-1) + 1;
```

```
horizontal_vertical = [N 1];
centralConnectionNodeID = [];
for i=1:(N-1)^2
    centralConnectionNodeID = [centralConnectionNodeID [horizontal_vertical+centralNodeID(1,i)]];
end

sending = [sort([centralNodeID centralNodeID]) topNodeID rightNodeID];
sending(sending==NodeCount) = [];
recieving = [centralConnectionNodeID topConnectionNodeID rightConnectionNodeID];
resistances = ones(length(sending),1);
edge_data = [sending(:) recieving(:)];

[X,V]=KirchoffSolve(edge_data,resistances,1,NodeCount);
network_resistance = max(V);
toc
```

Network Size	20	30	40	50	60	70	80	90	100
Network Resistance (Ω)	3.954	4.450	4.806	5.084	5.312	5.505	5.673	5.821	5.956
Processing Time (s)	0.017	0.021	0.031	0.041	0.059	0.084	0.118	0.176	0.229

Network Size	110	120	130	140	150	200	250	300	400	500	600
Network Resistance	6.074	6.183	6.284	6.378	6.466	6.830	7.1125	7.344	7.709	7.993	8.224
Processing Time (s)	0.319	0.410	0.546	0.735	0.867	2.518	5.585	18.626	73.030	185.62	410.978

Let's look at the graphs for both network resistance and processing time against network size.

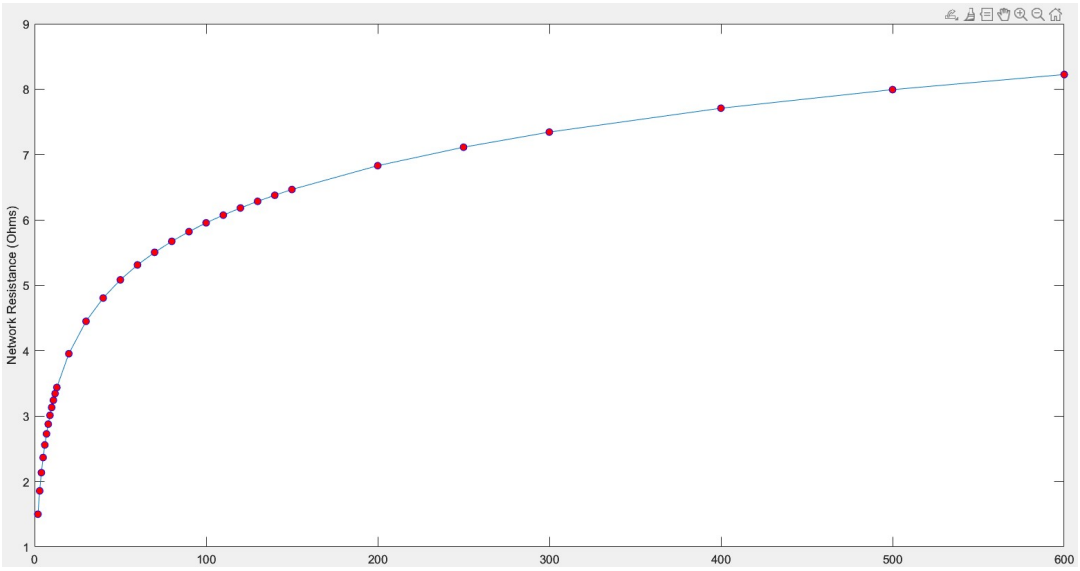


Figure 13: Network resistance against network size

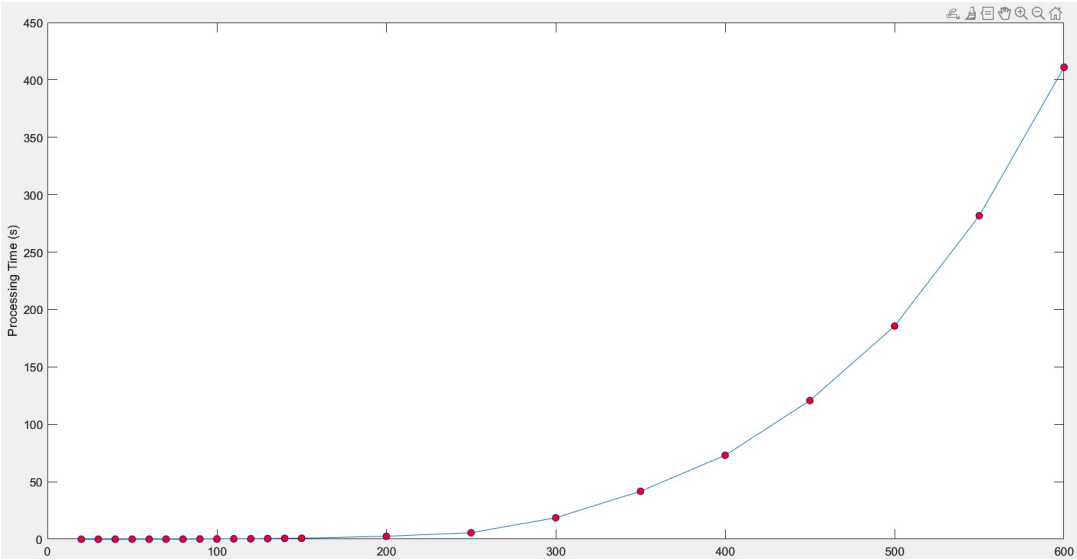


Figure 14: Processing time against network size

Figures 13 and 14 both show very interesting results. They seem to be the inverse of one another with network resistance showing a logarithmic trend and processing time displaying exponential characteristics. *Microsoft Excel's* line fit was used test this hypothesis by determining estimate equations for these lines as well as the respective coefficient of determination values [9]. These were the results:

Network resistance:

$$y = 1.208\ln(x) + 0.4064$$

$$R^2 = 0.9988$$

Processing time:

$$y = 0.0321e^{0.018x}$$

$$R^2 = 0.8767$$

However if processing time is modelled as a polynomial of order 3:

$$y = 4.29 \times 10^{-6}x^3 - 0.0018x^2 + 0.2257x - 7.2818$$

$$R^2 = 0.9993$$

The hypothesis laid out previously was only partly correct. Network resistance as a function of network size is certainly logarithmic, however processing time as a function of network size is certainly more accurately modelled as a polynomial. In the making of the Figures 13 and 14, a 1000x1000 network was attempted. This iteration was cut short due to how long it was taking. By substituting $x = 1000$ into the equation above, it can be estimated that it would take about 45 minutes to complete. Further estimations are expressed below.

Network Size	1000	1100	1200	1300	1400	1500
Predicted Processing Time	45 minutes	1.05 hours	1.41 hour	1.85 hours	2.38 hours	2.99 hours

Network Size	2000	3000	4000	5000	10000	20000	50000
Predicted Processing Time	7.66 hours	1.16 days	2.85 days	5.70 days	6.799 weeks	1.066 years	16.86 years

Table 7: Estimations for processing times for very large networks

It is important to note that the values in Table 7 are only estimates. They do however show how infeasible this method is for much larger networks.

4 Global Optimisation

4.a

This section will contain a brief overview of particle swarm optimisation, with brief introduction followed by methodology and finally some of the pros and cons:

4.a.1 Introduction

Particle swarm optimisation (PSO) is a bio-inspired algorithm proposed by Kennedy and Eberhart in 1995 [11]. It is built on the idea that a school of fish, a flock of birds or an equivalent collection that move in groups can profit from experience of all other members. This method provides a heuristic solution because we can never prove that the real global optimal solution has been found and in complex problems with a high number of dimensions – it usually isn't. However PSO very often produces a solution that is very close to the global optimum.

PSO is a dependency of swarm intelligence which is in turn a dependency of emergent complexity [7]. Emergent complexity is a phenomenon whereby larger entities arise through interactions among smaller or simpler entities and has two necessary components:

1. Fundamental units with simple behaviour
2. An external force for cooperation

This conditions will allow for the emergence of complex behaviour from a collection of relatively simple and fundamental units. This phenomenon isn't inherently natural meaning artificial systems can be developed that exhibit equivalent behaviours.

Swarm intelligence is the useful subset of systems that exhibit emergent complexity. Natural systems are fantastic candidates for this as for complex natural behaviours to exist, they must solve problems effectively or evolution would've rendered them extinct. For this reason, natural systems are perfect examples from which to draw inspiration when creating an optimisation model.

Let's consider a swarm intelligence model – a minimum flocking model of bird-like objects (boids). Each boid is defined by a position and velocity vector and has a total awareness of events within in a certain local vicinity and no awareness of events outside that. PSO is an optimisation algorithm that uses a swarm intelligence model. The basic idea is you put a flock of boids into a search space and give them behaviours which depend on the function being evaluated.

There are 4 steering behaviours in PSO:

1. Convergence – particles steer towards the best value in the vicinity
2. Separation – delegate the search space among the boids
3. Alignment – Create a local agreement about the best movement
4. Cohesion – Prevent the flock from scattering.

4.a.2 Methodology

Lets start with a function $f(x, y)$. The first step is to initialise a random number of points (called particles) and let them look for a minimum point in random directions. Each particle will move on an iterative basis where a particles velocity for a given iteration is determined by the not only the minimum point that particle has ever been to but also the minimal point found by the entire swarm. After a certain number of iterations, the global minimum point is deemed to be minimum point explored by any particle in the swarm. Let's look at the equations that drive this movement.

The position of particle i at iteration t can be denoted by $X_i(t)$, which in a 2D example can be given as a set of coordinates: $X_i(t) = (x_i(t), y_i(t))$. As mentioned earlier, each particle is also defined by a velocity vector $V_i(t) = (v_{xi}(t), v_{yi}(t))$. This means at the start of the next iteration the position of each particle would be:

$$X_i(t+1) = X_i(t) + V_i(t+1)$$

or

$$x_i(t+1) = x_i(t) + v_{xi}(t+1)$$
$$y_i(t+1) = y_i(t) + v_{yi}(t+1)$$

At the end of each iteration, three things must occur:

1. The position of each point must be compared to minimum value reached by that point and updated if necessary.
2. The minimum reached by any point must be reevaluated and updated if necessary.
3. The velocity vector of each particle must be recalculated.

Each particles' velocity for the next iteration is calcualted as follows:

$$V_i(t+1) = wV_i(t) + c_1r_1(b_{local} - X_i(t)) + c_2r_2(b_{vicinity} - X_i(t))$$

r_1 and r_2 are are random variables with a uniform distribution between 0 and 1 [6], c_1 and c_2 are the hyper parameters of the PSO algorithm, they are uniformly distributed between 0 and 1 ($c_1 + c_2 = 1$) and define the ratio between exploration and exploitation, if c_1 is favoured, particles will be prompted to explore their local minimum points where as if if c_2 is favoured, particles will be prompted to explore the current global minimum point. w is the inertia weight which shows the effect of the previous velocity vector, it is between 0 and 1 determining how much of the previous iterations velocity should be maintained. b_{local} is the best $f(X)$ value ever reached by that particle and $b_{vicinity}$ is the best $f(X)$ value reached by any particle. An upper bound is placed on the velocity in all dimensions.

This approach will run continually until convergence. The termination point of this algorithm can be determined in different way, be it a fixed number of iterations or a dynamic approach based off variations (or lack of) in $b_{vicinity}$.

4.a.3

Pros PSO is a gradient-free optimisation method. In Section 3.b, an iteration of the steepest descent was carried out. In that method, the direction from one iteration to the next is given by $-\nabla f(X)$. That means this method is much more appropriate for multi-dimensional optimisation where differentiation isn't possible. It is also a relatively simple concept that isn't too difficult to implement.

Cons PSO can provide low quality solutions if the algorithm hasn't converged and the program is terminated prematurely. As mentioned earlier, this method provides a heuristic solution meaning it cannot be known if the calculated solution is the true global minimum. A high quantity of calculations and checks need to be done every iteration for each particle in the swarm. This can lead to high memory usage as well as requiring a lot of processing, especially when the swarm size becomes large. Finally, for certain problems and parameter inputs, the stochastic variability can be very high. [1]. If your hyper parameters are not calibrated correctly, it becomes very easy for local minima to be identified and returned as the global minima due to a lack of exploration by the particles (this last point will be looked at in more detail in the next section).

4.b

Prenote: I am aware that there is a robust Particle Swarm Optimisation toolbox in MatLab [8]. However for educational purposes I am going to attempt coding my own PSO algorithm from scratch to find the global minimum of the EggHolder function.

Initialising inputs Let's define the initial inputs:

```
clc
clear
%Number of search particles
n_particles = 500;
%Upper and lower bounds of search domain. This would need
%slight modification if area was non-square
upper = 512;
lower = -512;
%Initialise a random set of displacements and velocities for each particle
X = (upper-lower)*rand(2,n_particles)+lower;
V = rand(2,n_particles);
%f will be the function values for each particle
%Assign temporary outputs for this for now.
f = X(1,:);
%b_vicinity is the current identified global minimum
b_vicinity = X(:,find(f==min(f)));
b_vicinity_value = min(f);
%b_local is the minimum point found by each particle
b_local = X;
b_local_value = f;
%Hyper parameters:
%w is intertia weight (how much of the previous velocity is retained)
w = 0.5;
%c1 & c2 define the ratio between exploration and exploitation.
% For now they will be both set to 0.5 (this will be reassessed later)
c1 = 0.5;
c2 = 0.5;
```

This is an example of what the initial state of the swarm would look like on a contour plot of the function:

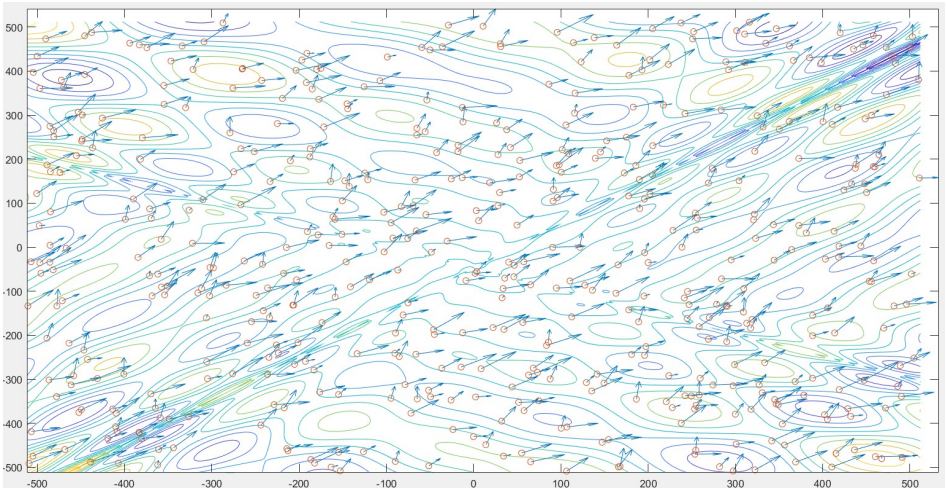


Figure 15: Example of Initial Deployment of particles

The next step is to set up an iteration that carries out the procedure described in Section 4.a.2. Let's see how this was developed in MatLab

```
iteration_count = 1000;
it = 0;

while it < iteration_count
    it = it+1;
    %randomly generate r1 & r2 = 1-r1 as seen on the next line
    r = rand(1,1);
    %Calculate the velocity for the coming iteration
    V = w*V+c1.*r(1,1)*(b_local-X)+c2.*(1-r(1,1))*(b_vicinity-X);
    %New particle locations
    X = X+V;
    %This enforces the cohesion rule, if at any point a particle finds itself
    %outside the domain, it is simply returned to its location in the previous iteration.
    maxX = max(X);
    for i = 1:n_particles
        if abs(maxX(1,i))>512
            X(:,i)=X(:,i)-V(:,i);
        end
    end
    %Evaluate new function values at each point
    x = X(1,:);
    y = X(2,:);
    f = -1.*(y+47).*sin((abs(0.5.*x+y+47)).^0.5)-x.*sin(((abs(x-(y+47))).^0.5));
    %If a new local minimum is identified, substitute it in
    for i = 1:n_particles
        if f(1,i)<b_local_value(1:i)
            b_local(:,i)=X(:,i);
        end
    end
    %If a new global minimum is identified, substitute it in
    if min(f)<b_vicinity_value
        b_vicinity_value = min(f);
        coords = X(:,find(f==min(f)));
        b_vicinity(:,1) = coords(:,1);
    end
end
```

Two significant issues kept occurring here. Firstly particles kept leaving the domain despite the cohesion clause that had been added in. This was due to the inertia weighting being too high leading to ever increasing velocities which lead to particle movement being far, far too high. After a few rounds of trial and error, the value of $w = 0.3$ was found to be the largest value where cohesion was still observed. There was also an issue where if w was too low, particles wouldn't be able to explore outside of the region surrounding their local minimum. If the issue continues to occur, a hard limit would need to be set on the velocity magnitude. The second issue that was encountered was misidentifying of local minima as the global minimum. The three points where this consistently occurred were:

$$f(-454.8615, 381.3727) = -894.57$$
$$f(-457.0550, -383.3147) = -786.52$$
$$f(439.4810, 453.9774) = -935.338$$

Here is are the final results for these incorrect solutions:

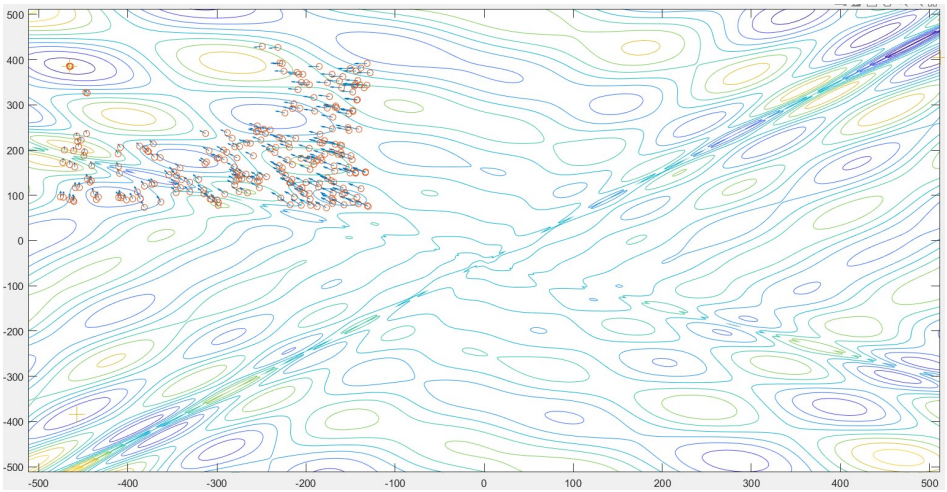


Figure 16: Convergence for $f = -894.57$

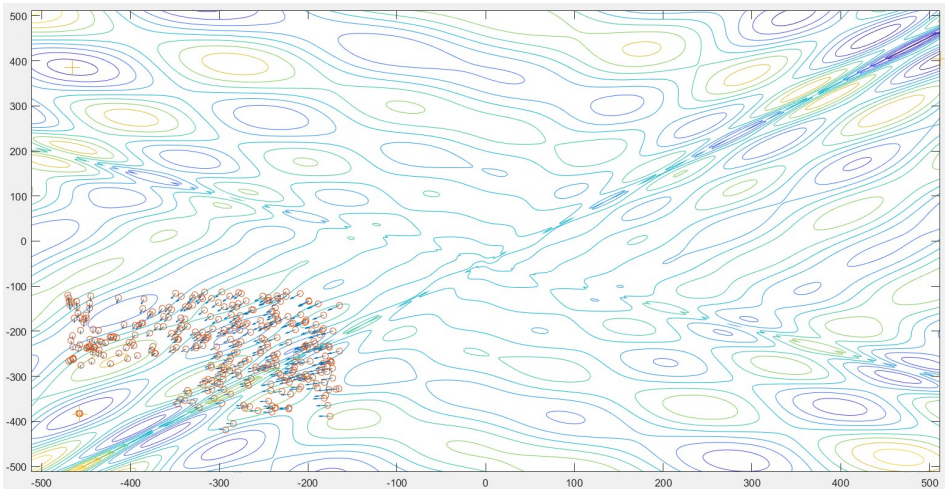


Figure 17: Convergence for $f = -786.52$

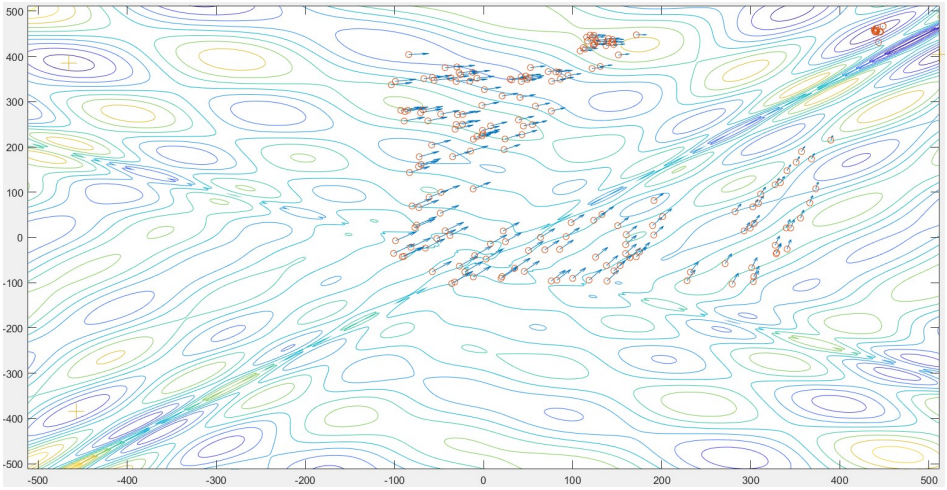


Figure 18: Convergence for $f = -935.338$

After some thought it was determined that this was caused by the model having an excessively high exploitation weighting (c_2 was too big) so when a relatively strong and central local minimum was identified, all particles were quickly redirected towards it, meaning the true global minimum, which resides right on the edge of the domain could never be discovered. c_1 and c_2 were adjusted to 0.87 and 0.13 respectively to place a much heavier emphasis on exploration of the feasible region. This returned much stronger results, so much so that the convergences seen in Figures 16 and 17 rarely occur. However the true global minimum was rarely returned by this method with the algorithm often returning the results found in 18.

To solve this, a brute force strategy was adopted. The plan was to redeploy the entire algorithm on an iterative loop, keeping track of the iteration that discovered the lowest point and storing the particle positions, vectors, local minimums and its vicinity minimum, as it is likely that this point would be situated near the global minimum. Once the correct vicinity had been determined, the parameters of that iteration are entered into a modified algorithm, weighted toward exploitation ($c_1 = 0.4$ & $c_2 = 0.6$), so this area can be investigated in more detail, hopefully converging towards the true global minimum. Let's see the results produced by this model in three separate instances.

	Instance			
	1	2	3	Global Minimum
fval	-959.4598	-959.5036	-959.6397	-959.6407
fval coordinates	(512.0,404.8324)	(512.0,404.8842)	(512.0,404.2706)	(512.0,404.2319)

Table 8: Three individual test cases of upgraded PSO algorithm

While there is a brute force element to this method, with a standard run time between 90 and 120 seconds, Table 8 shows excellent results achieving near perfection in the vast majority of cases, incorrect outputs can still emerge however they are far less likely with this set up with the correct global optimum being identified in vast majority of cases. This is the final script that achieved these results minus previously seen notation.

```
clc
clear
tic
dummyx = linspace(-512,512,1024);
dummyy = linspace(-512,512,1024);
[xx,yy]=meshgrid(dummyx,dummyy);
zz = -1.*(yy+47).*sin((abs(0.5.*xx+yy+47)).^0.5)-xx.*sin(((abs(xx-(yy+47))).^0.5));
p = contour(xx,yy,zz);
hold on
%External iteration value
bigIt = 0;
%Initialise best iteration variables
bestf = 0;
bestf_coords = [];
best_X = [];
best_V = [];
best_b_local = [];
best_b_vicinity = [];
%This is here for error catching purposes
optimum = -959.6407;
n_particles = 2000;
upper = 512;
lower = -512;
iteration_count = 200;
while bigIt < 100
    bigIt = bigIt + 1
    bestf
    X = (upper-lower)*rand(2,n_particles)+lower;
    f = X(1,:);
    b_vicinity = X(:,find(f==min(f)));
    b_vicinity_value = min(f);
    b_local = X;
    b_local_value = f;
    w = 0.3;
    c1 = 0.87;
    c2 = 0.13;
    V = rand(2,n_particles);
    it = 0;

    while it < iteration_count
        it = it+1;
        r = rand(1,1);
        %This didn't work
        %c2 = it/iteration_count;
        %c1 = 1-c2;
        V = w*V+c1.*r(1,1)*(b_local-X)+c2.*(1-r(1,1))*(b_vicinity-X);
        X = X+V;
        maxX = max(X);
        for i = 1:n_particles
            if abs(maxX(1,i))>512
                X(:,i)=X(:,i)-V(:,i);
            end
        end
        x = X(1,:);
        y = X(2,:);
        f = -1.*(y+47).*sin((abs(0.5.*x+y+47)).^0.5)-x.*sin(((abs(x-(y+47))).^0.5));
        for i = 1:n_particles
            if f(1,i)<b_local_value(1:i)
                b_local(:,i)=X(:,i);
            end
        end
        if min(f)<b_vicinity_value
            b_vicinity_value = min(f);
            coords = X(:,find(f==min(f)));
            b_vicinity(:,1) = coords(:,1);
```

```
        end
    end
    x = b_vicinity(1,1);
    y = b_vicinity(2,1);
    f = -1.*(y+47).*sin((abs(0.5.*x+y+47)).^0.5)-x.*sin(((abs(x-(y+47))).^0.5));
    if f < bestf
        disp("new best iteration")
        bestf = f
        bestf_coords = [b_vicinity(1,1) b_vicinity(2,1)]
        best_X = X;
        best_V = V;
        best_b_local = b_local;
        best_b_vicinity = b_vicinity;
    end
    if bestf<optimum
        %This is just here for error catching purposes
        disp("optimum either reached or exceeded")
        break
    end
end
X = best_X
V = best_V
b_local = best_b_local;
b_vicinity = best_b_vicinity;
w = 0.25;
c1 = 0.4;
c2 = 0.6;
for m = 1:100
    r = rand(2,1);
    V = w*V+c1.*r(1,1)*(b_local-X)+c2.*r(2,1)*(b_vicinity-X);
    X = X+V;
    maxX = max(X);
    for i = 1:n_particles
        if abs(maxX(1,i))>512
            X(:,i)=X(:,i)-V(:,i);
        end
    end
    x = X(1,:);
    y = X(2,:);
    f = -1.*(y+47).*sin((abs(0.5.*x+y+47)).^0.5)-x.*sin(((abs(x-(y+47))).^0.5));
    for i = 1:n_particles
        if f(1,i)<b_local_value(1:i)
            b_local(:,i)=X(:,i);
        end
    end
    if min(f)<b_vicinity_value
        b_vicinity_value = min(f);
        coords = X(:,find(f==min(f)));
        b_vicinity(:,1) = coords(:,1);
    end
end
    quiver(X(1,:),X(2,:),V(1,:),V(2,:));
    scatter(X(1,:),X(2,:));
    plot(bestf_coords(1,1), bestf_coords(1,2),'*','MarkerSize', 25)

b_vicinity_value
b_vicinity
toc
```

Finally, let's look at the final convergence. (Figure 19)

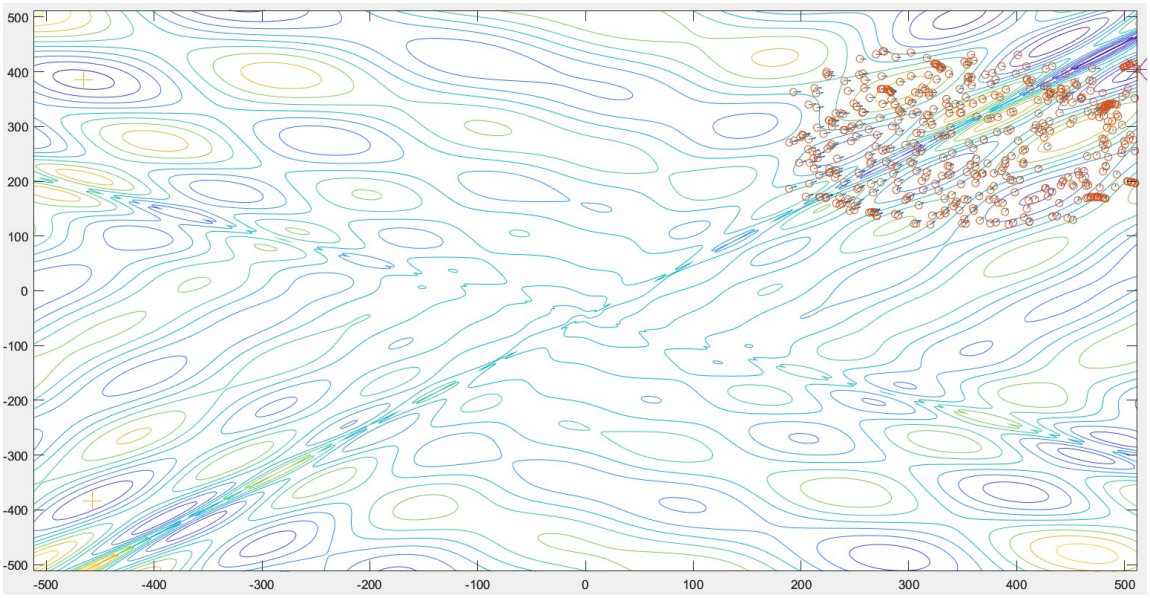


Figure 19: Convergence for $f = -959.6407$

4.c

eddieSecretFunction.p is a relatively simple function. It is in the form:

$$f\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = Y$$

However when the same input was run through it multiple times, different values would be returned. This implies there is a random component within the secret function. The first step was to loop identical inputs and track all the different outputs. Plotting a sorted version of these results should hopefully indicate the type of relationship between this random variable and the output. This was done using the following code:

```
clear
clc
range = 10000;
N = linspace(1,range,range);
outputs = [];
for i = 1:range
outputs = [outputs (eddieSecretFunction([0;0]))];
end
p = plot(N,sort(outputs(1,:)),'DisplayName',"x = 0")
xlabel('N')
ylabel('output')
legend
```

These are the results:

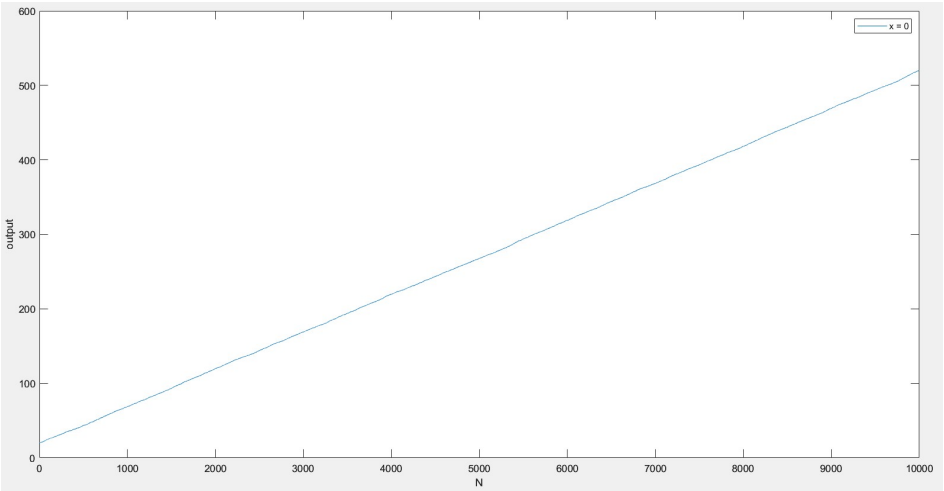


Figure 20: Distribution of random variable at $x = 0$.

Figure 20 shows the relationship between the random variable and the output can accurately be modelled as linear. It was also determined that the maximum and minimum output values were independent of the number of iterations.

$$\max(x_1 = x_2 = 0) \approx 520$$
$$\min(x_1 = x_2 = 0) \approx 20$$

The next step is to run a similar procedure for alternative x_1 values and plot the results on the same graph. It is important when doing this that only x_1 is varied so the variation of x_2 isn't incorporated into the analysis.

```
clear
clc
range = 3;
range2 = 2000;
N = linspace(1,range2,range2);
for k = -range:range
    outputs = [];
    for i = 1:range2
        outputs = [outputs (eddieSecretFunction([k;0]).')];
    end
    p = plot(N,sort(outputs(1,:)),'DisplayName',int2str(k));
    hold on
end
xlabel('N')
ylabel('output')
legend
```

These are the results:

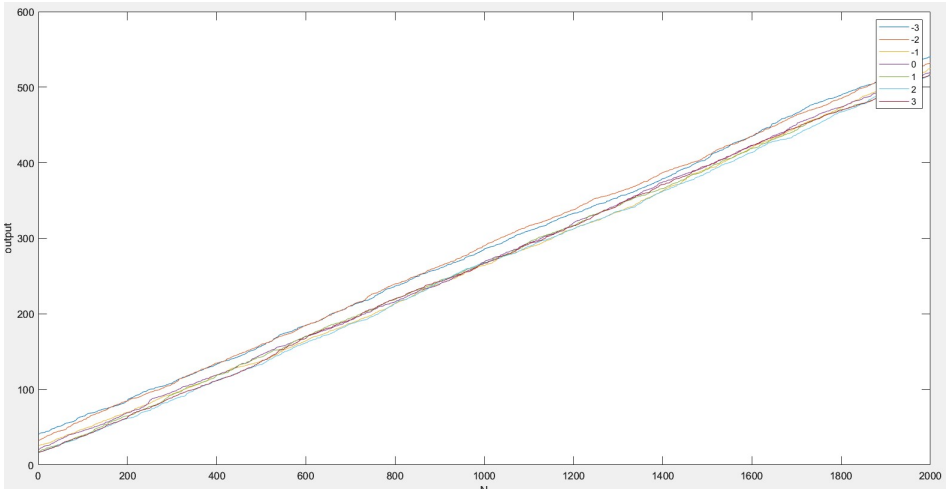


Figure 21: Distribution of random variable for $x = -3 : 3$.

Figure 21 shows that each value of x_1 not only exhibits linear behaviour but also has the same range of outputs as well as the same gradients. As seen in the code, $x_2 = 0$ for the whole script.

$$\max(x_2 = 0) - \min(x_2 = 0) \approx 500$$

Effect of x_2 To investigate the effect of x_2, the same method will be adopted, except x_2 would be varied and changes would be analysed. The results were that variations in x_2 simply caused a further shifts in the distribution of outputs similar to that when x_1 is varied. This shift is independent of the value of x_1. For this reason, for now, x_2 will be set to zero, so the relationship between x_1 and Y can be determined. The effect of x_2 will be determined later.

$$\max - \min \approx 500$$

From here, simple algebra can be used to find the equation for these lines, starting with the equation for a straight line. Each line's outputs will be plotted as a function of the the random variable and y-intercept. The y-intercept will be further mapped as a function of x_1.

In a general case:

$$\begin{aligned} y &= mx + c \\ Y &= mr + \min \\ m &= \frac{\max - \min}{r_{\max} - r_{\min}} \\ Y(x_2 = 0) &= \frac{\max - \min}{r_{\max} - r_{\min}} \cdot r + \min \end{aligned}$$

There are two points we know the values of on each of these lines:

$$\begin{aligned} r = r_{\min} &\mapsto Y = \min \\ r = r_{\max} &\mapsto Y = \max \end{aligned}$$

These can be substituted into the straight line equation to obtain values for r_{max} and r_{min}

$$\begin{aligned} \min &= \frac{\max - \min}{r_{\max} - r_{\min}} \cdot r_{\min} + \min \\ 0 &= \frac{\max - \min}{r_{\max} - r_{\min}} \cdot r_{\min} \\ &\text{Either:} \\ r_{\min} &= 0 \\ &\text{Or:} \\ (\max - \min) &= 0 \\ &\text{This isn't feasible } \therefore \\ r_{\min} &= 0 \end{aligned}$$

$$\begin{aligned} \max &= \frac{\max - \min}{r_{\max} - r_{\min}} \cdot r_{\max} + \min \\ \max \cdot r_{\max} - \max \cdot r_{\min} &= \max \cdot r_{\max} - \min \cdot r_{\max} + \min \\ \min \cdot r_{\max} - \max \cdot r_{\min} &= \min \\ r_{\min} &= 0 \\ \min \cdot r_{\max} &= \min \\ r_{\max} &= 1 \end{aligned}$$

$$r_{\min} = 0$$
$$r_{\max} = 1$$

$$Y_N = (\max - \min) \cdot r + \min \tag{6}$$

The next step is to map min as a function of the input x_1. To do this, the code used to generate Figure 21 will be modified. The range of the input x_1 values will be increased, and instead of plotting each result, the minimum value returned from each x_1 value will be stored. $\min_{x_1}$ will be then plotted against x_1. Here is the code that was used to obtain this data.

```
clear
clc
min_matrix = [];
range = 50;
%Value of range2 has been made much larger. This is because a
%much higher resolution of results is needed to get the most accurate min values
range2 = 5000;
N = linspace(1,range2,range2);
for k = -range:range
    outputs = [];
    for i = 1:range2
        outputs = [outputs (eddieSecretFunction([k;0])];
    end
    min_matrix = [min_matrix min(output)];
end
```

The results were then extracted and transferred to *Microsoft Excel* for analysis. A polynomial line of best fit was calculated for the data. These are the results:

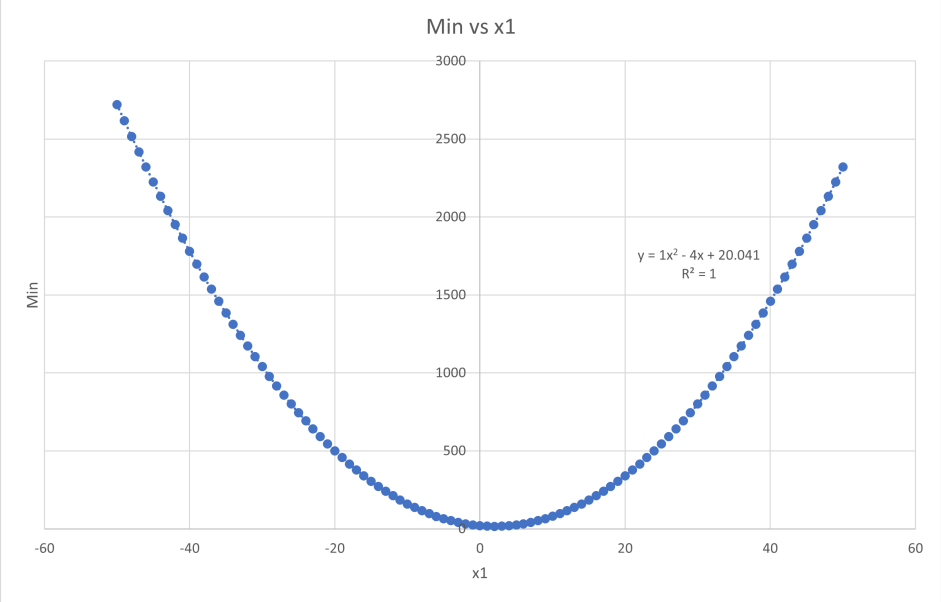


Figure 22: Relationship of $\min(x_2 = 0)$ to x_1

The graph shows that:

$$\min(x_2 = 0) \approx x_1^2 - 4x_1 + 20 \tag{7}$$

Therefore:

$$Y(x_2 = 0) \approx (\max - \min) \cdot r + x_1^2 - 4x_1 + 20 \tag{8}$$

Where:

$$(\max - \min) \approx 500$$
$$0 \leq r \leq 1$$

Working x_2 into this model. As said earlier, variations in x_2 simply cause further shifts in the output distribution, and these shifts are independent of x_1. Sample data for how variations in x_2 translate the output distribution relative to the distribution of $x_2 = 0$ were collected.

x_2	-3	-2	-1	0	1	2	3	4	5
Translation	33	20	9	0	-7	-12	-15	-16	-15

This was plotted and an equation calculated.

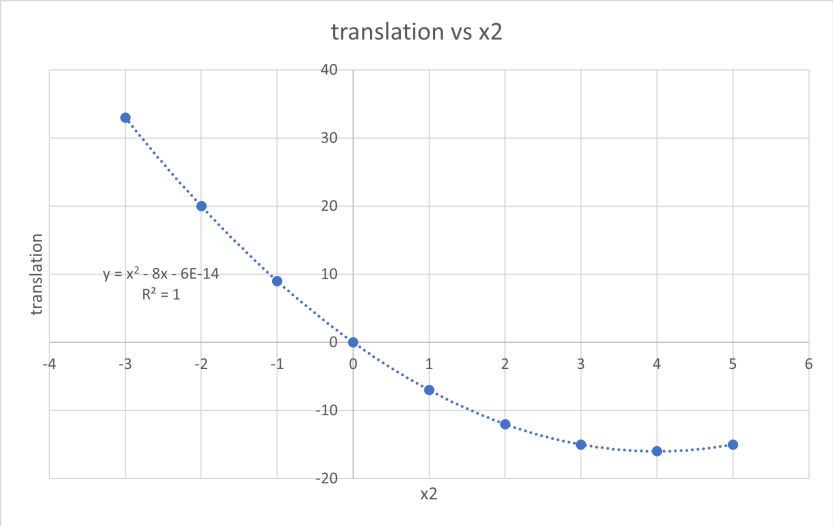


Figure 23

The equation for Y can now be extended:

$$
\begin{aligned}
Y &\approx (\max - \min) \cdot r + x_1^2 - 4x_1 + x_2^2 - 8x_2 + 20 \\
Y &\approx (\max - \min) \cdot r + ((x_1 - 2)^2 - 4) + ((x_2 - 4)^2 - 16) + 20 \\
Y &\approx (\max - \min) \cdot r + (x_1 - 2)^2 + (x_2 - 4)^2
\end{aligned}
$$

Minimising Y Since Y is formed by the addition of independent terms, each term can be minimised individually.

$$\min(Y) \approx \min((\max - \min) \cdot r) + \min((x_1 - 2)^2) + \min((x_2 - 4)^2)$$

$$\min((\max - \min) \cdot r) = 0$$

$$
\begin{aligned}
\min((x_1 - 2)^2) &= 0 \\
x_1^* &= 2
\end{aligned}
$$

$$
\begin{aligned}
\min((x_2 - 4)^2) &= 0 \\
x_2^* &= 4
\end{aligned}
$$

$$\min(Y) \approx 500r$$

Absolute minimum possible point will occur when $r = 0$

$$Y_{min} \approx 0$$

This can only occur at:

$$\mathbf{x} = \begin{bmatrix} 2 \\ 4 \end{bmatrix}$$

References

[1] *A very brief introduction to particle swarm optimization.* URL: http://www.iam.fmph.uniba.sk/ospm/Harman/PSO.pdf.

[2] *MathWorks Help Center - intlinprog.* 2014. URL: https://www.mathworks.com/help/optim/ug/intlinprog.html.

[3] *MathWorks Help Center - linprog.* 2006. URL: https://www.mathworks.com/help/optim/ug/linprog.html.

[4] *MathWorks Help Center - linprog/exitflags.* 2006. URL: https://www.mathworks.com/help/optim/ug/linprog.html#buusznx-exitflag.

[5] *MathWorks Help Center - quadprog.* 2006. URL: https://www.mathworks.com/help/optim/ug/quadprog.html.

[6] *Particle swarm algorithm for solving systems of nonlinear equations.* 2011. URL: https://www.sciencedirect.com/science/article/pii/S0898122111004299#.

[7] *Particle Swarm Optimisation - Churchill CompSci Talks.* 2018. URL: https://www.youtube.com/watch?v=DzcZ6bP4FGw&ab_channel=ChurchillCompSciTalks.

[8] *Particle Swarm Optimization Toolbox.* 2006. URL: https://www.mathworks.com/matlabcentral/fileexchange/7506-particle-swarm-optimization-toolbox.

[9] *Wikipedia - Coefficient of Determination.* 2022. URL: https://en.wikipedia.org/wiki/Coefficient_of_determination.

[10] *Wikipedia - Hadamard Product.* 2022. URL: https://en.wikipedia.org/wiki/Hadamard_product_(matrices).

[11] *Wikipedia - Particle swarm optimization.* 2022. URL: https://en.wikipedia.org/wiki/Particle_swarm_optimization.