MAST30013 Techniques In Operations Research

Flat Plains Radio: Efficient Location Proposal

Ned Mahony (695584)
Luke Sitzler (696290)
Kurt Stoeckl (694984)
William Troiani (636335)

Department of Mathematics and Statistics
University of Melbourne

Abstract. A classical idea in the field of numerical optimisation is one that involves minimising the total cost of building a resource while also maximising its useful properties. This report will detail an applied example of this, where the optimal location of two radio stations (R1 & R2) in 'Flat Plains' will be proposed. The aim is to minimise the magnitude of a cost function with respect to several different objective functions. In order to achieve this result, a number of different numerical optimisation techniques will be used. An evaluation of the accuracy and efficiency of each method will be discussed and the optimal solution for each individual objective function stated. Finally, there will be an investigation into a potential compromise solution that aims to satisfy all of the objective functions.

1 Introduction

The Flat Plains area contains 5 towns (A, B, C, D and E) that need to be serviced by the two radio stations. There are three parties with conflicting interests which will be examined individually. Firstly, the council's objective is to minimise the total length of all roads. They require that R1 is directly connected to towns A, B and C as well as R2 being connected to towns C, D and E. In addition, there must be a road connecting R1 and R2 (this is also true for the remaining objectives).

Secondly, the managers want the radio stations to be as physically accessible as possible so that they can be accessed easily if there are transmission issues. In particular, they want to minimise the maximum distance from R1 to A, B and C and the maximum distance from R2 to C, D and E.

Finally, the owners wish to minimise power costs. This cost function is proportional to the square of the distance the signal travels from the radio station. They require that R1 transmits to A, B and C while radio station R2 transmits to B, C, D and E.

Therefore, the principle aim of this project is to obtain a solution for each of the individual parties and then eventually reach a compromise solution. Each objective will be analysed individually using 3 different numerical methods. We will then compare the efficiency of each method as well as propose which one provides the most accurate result.

The methods we have selected are: the Method of Steepest Descent, Newton's Method and the Nelder-Mead Method. All of these optimisation techniques are different from one another and require varying levels of input information. To evaluate the performance of each method, a comparison of the computational time as well as the number of iterations that are needed to reach an optimal solution will be considered. Also, we will take into account the level of information and assumptions that are made for each method. For example, Newton's Method requires being able to compute the inverse of the Hessian which assumes that the function that we are dealing with is C^2. In contrast, the Steepest Descent Method does not require any second-order information about the function so thus assumes only that it is C^1, while the Nelder-Mead Method uses nothing more than the objective function.

To conclude the investigation, we will form a compromise solution between the three parties. We will do this by formulating a number of problems each with different objectives. We will then proceed to solve these new problems through an additional method: Particle Swarm.

Throughout the report we will draw on research that we have found online, in

textbooks and through professionals working in industry, as well as what we have covered in class. All of these references will be detailed at the end of the report.

The locations of the towns (in local coordinates, measured in kilometres) are given in the following table. We assume the region is completely flat.

Town	x-coordinate	y-coordinate
A	50.50	207.35
B	146.25	324.60
C	308.50	320.15
D	406.80	190.45
E	520.85	47.20

The rest of the paper is structured as follows. In Section 2 we introduce and discuss each of the methods used in solving the problems. In Section 3 we present our results and analyse the performance of each of the methods in solving the particular problems at hand. In Section 4 we investigate a number of ways in which we can reach a compromise solution.

2 Methods

Method 1: The Method of Steepest Descent

In essence the Method of Steepest Descent is a greedy algorithm which seeks to minimise the value of a function by consistently heading in the direction that results in the largest decrease in the function in the short run. An intuitive grasp of how the method works can be found by visualizing a ball placed on the side of a valley. This ball will roll with gravity towards the lowest point in the valley. However, we add in the condition that once the ball starts rolling in a direction it must continue on that direction until its height in the valley is minimized, then it stops, before rolling in another direction until it reaches the bottom. This produces a right-angled zig-zag pattern in the path of the ball.

The main advantage of the Method of Steepest Descent is its relative simplistic nature, meaning it is easy to implement. However, offsetting this are several key disadvantages.

Firstly, this method requires the gradients of the objective functions. The gradients of sub-problems 1 and 3 were found quite rapidly and they existed for almost all $\boldsymbol{x}$, however in order to obtain the gradient of sub-problem 2 it was necessary to express the objective function in terms of the Heaviside function, a non-trivial task.

(Note: The only point at which this Heaviside generated function differs in value is when the distance of the radio station from all 3 towns is exactly the same. At this point the Heaviside created function will actually be $\frac{3}{2}$ the value of the piecewise function. However, this is a singular point and is not a significant problem in numerical analysis.)

The fact that the gradient of the objective function for sub-problems 1 and 2 may fail to exist at a point posed another potential problem for the Method of Steepest Descent. This problem by itself was largely insignificant due to the imprecise nature of computing and numerical algorithms rendering the chances of the method non-deliberately reaching such a point very small. However, further analysis of sub-problem 2 revealed that the gradient of the problem would never actually approach zero, due to

jumping between different piecewise regions of the function. Hence a different condition for termination given a certain tolerance was needed. This new condition was chosen to be

$$\text{if } |x_k - x_{k+1}| < \epsilon \text{ end}$$

A third limitation is that the theoretical rate of convergence of this method is linear, which in the scheme of numerical methods is quite slow [4].

In addition, we expected the convergence of this method for sub-problem 2 to be quite poor; this was inferred from the ill-conditioned nature of the Hessian. A near singular Hessian suggests that in order for rapid convergence of a descent method, it may be required to step in a direction almost orthogonal to the gradient, however the Steepest Descent Method, by definition, does not do this [10].

Method 2: Newton's Method

The Newton Method can be described as the classical second order method when we are talking about unconstrained multivariable optimisation techniques. It is similar to the Steepest Descent Method and differs because it encapsulates the use of second order information. Therefore, in practice Newton's Method would get to the minimum in one step if the second order approximation held exactly and if the function being minimised was a quadratic function.

Given any arbitrary C^2 objective function f, the idea of Newton's Method is to minimise at each iteration the quadratic approximation of f around the current iterate $\boldsymbol{x}_k$. The algorithm begins by selecting a descent direction. In order to do this we need to find several input variables: the Jacobian and the inverse of the Hessian. It should be noted that this is generally a computationally expensive process. The Newton direction is given by

$$\boldsymbol{d}^k := -\nabla^2 f(\boldsymbol{x}_k)^{-1} \nabla f(\boldsymbol{x}_k).$$

The following diagram depicts how the Newton direction is superior to the descent direction used in the Steepest Descent Algorithm. The red line is the Newton direction and the green line is the Steepest Descent direction.

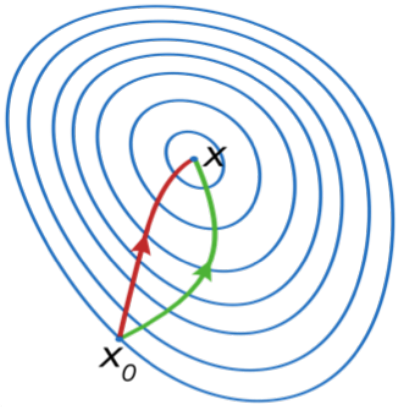

Figure 1: A visual comparison of Newton's Method and the Method of Steepest Descent [2].

It is clear that because the Newton Method utilises second order information it is able to approximate a more accurate direction compared to the Steepest Descent direction in theory.

It is noted that for an arbitrary C^2 function, there is no guarantee that the Hessian is invertible, nor is it always true that the Newton direction is a descent direction. Therefore, at each iteration we must check both of these conditions.

Assuming that the Hessian is positive definite, the next step of the algorithm involves selecting an appropriate step size. If the Hessian is not positive definite we use the Steepest Descent direction. The step size is calculated by solving a single-variable minimisation problem using the Golden Section Search. Once this has been found, the algorithm will select the new point and check if it is a minimum (subject to some tolerance). If it is not, it will repeat this process until a minimum is found.

As discussed above, in practice, the Newton Method should require less time to reach an estimation of the optimal solution compared to the Steepest Descent Method. However it does require some potentially expensive calculations in terms of the time needed to compute them. Thus, overall it does have both positive and negative attributes. These will be evaluated in more detail later in the report when the method is analysed with respect to the particular problems at hand.

Method 3: The Nelder-Mead Method

In order to gain an intuitive understanding of the way this algorithm functions, the description of the algorithm in n dimensions is accompanied by pictures of each step in the algorithm displayed in two dimensions. Please note for the following explanation, in order for the text to line perfectly with the pictures, one must only count to $n+1 = 3$.

In this general description of the Nelder-Mead algorithm [9], we are considering the problem of minimising a function $f(\boldsymbol{x})$ where $\boldsymbol{x} \in \mathbb{R}^n$.

We first work out the value of the function at each of the vertices in our simplex. Since our explanation here is in n dimensions, this corresponds to calculating $f(\boldsymbol{x}_1), f(\boldsymbol{x}_2), \ldots, f(\boldsymbol{x}_{n+1})$. Then we order these vertices in order from smallest value function to largest value function. We can assume here that $f(\boldsymbol{x}_1) \leq f(\boldsymbol{x}_2) \leq \ldots \leq f(\boldsymbol{x}_{n+1})$.

Now, since $\boldsymbol{x}_{n+1}$ yielded the worst function value, we decide not to explore the area beyond $\boldsymbol{x}_{n+1}$, and instead we explore the area on the other side of $\boldsymbol{x}_{n+1}$, that is, $\boldsymbol{x}_r$. Note: in two (three) dimensions, calculating $\boldsymbol{x}_r$ is equivalent to reflecting the triangle about the line (triangular face) $\boldsymbol{x}_1$ to $\boldsymbol{x}_2$ (to $\boldsymbol{x}_3$). To do this, we first calculate the midpoint of $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ (or the midpoint of the triangular face $\boldsymbol{x}_1, \boldsymbol{x}_2, \boldsymbol{x}_3$). In higher dimensions, this step will be replaced by calculating the average of all points $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$.

This is where the first coefficient for the Nelder-Mead method is used [5], the reflection coefficient. The point $\boldsymbol{x}_r$ will be chosen by calculating alpha multiplied by d, the Euclidean distance between $\boldsymbol{x}_{n+1}$ and the midpoint of the other points. In this context, we will restrict ourselves to a reflection coefficient of size 1, which gives a true reflection of the simplex.

Then a calculation of $f(\boldsymbol{x}_r)$ is performed, and we divide the possibilities for this value into four cases, and perform a different action depending on which of these outcomes arises. The four possibilities are:

1. $f(\boldsymbol{x}_r) < f(\boldsymbol{x}_1)$ (i.e. $\boldsymbol{x}_r$ is the best point we have found so far).

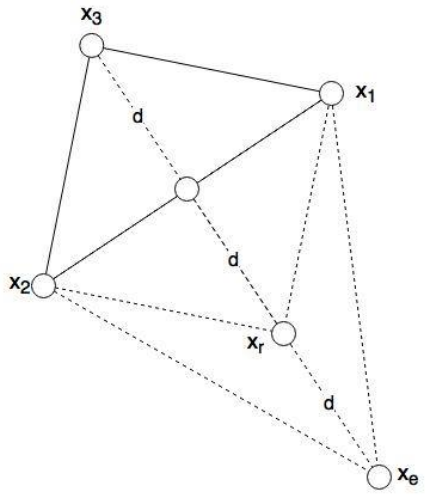

Figure 2: A visualisation of the reflection and expansion steps of the Nelder-Mead Method.

2. $f(\boldsymbol{x}_1) \leq f(\boldsymbol{x}_r) \leq f(\boldsymbol{x}_n)$ (i.e. $\boldsymbol{x}_r$ is a decent point, equal to or worse than the best we have found so far)
3. $f(\boldsymbol{x}_n) < f(\boldsymbol{x}_r) < f(\boldsymbol{x}_{n+1})$ (i.e. $\boldsymbol{x}_r$ is not a very good point, but not the worst we have found)
4. $f(\boldsymbol{x}_{n+1}) \leq f(\boldsymbol{x}_r)$ (i.e. $\boldsymbol{x}_r$ is equal to or worse than the worst point we have found so far)

Figure 3 shows the action of the algorithm based off which of these four outcomes occurs. When reading the flow chart, one always starts at the centre, and moves their way out depending on the function values. Once a leaf of the tree is reached, a calculation is performed to check if $f(\boldsymbol{x}_{n+1}) - f(\boldsymbol{x}_1) < tolerance$. If $f(\boldsymbol{x}_{n+1}) - f(\boldsymbol{x}_1) \geq tolerance$, then the process begins again from the centre, and if not, the algorithm terminates.

We now discuss the justification for each step of the algorithm.

1. $\boldsymbol{f(x_r) < f(x_1)}$**.** *This step is visualised in Figure 2.* If we draw a line from $\boldsymbol{x}_{n+1}$ to the midpoint of the remaining points, and extend this line, we eventually arrive at $\boldsymbol{x}_r$, which in this case, is the best point we have found so far. So perhaps we should keep searching along this direction, as maybe the function continues decreasing. This is why we now calculate the expansion point $\boldsymbol{x}_e$, and evaluate $f(\boldsymbol{x}_e)$. This is where we use our second coefficient, the expansion coefficient. A larger expansion coefficient means we explore further along this line. Here, we use 2 as our expansion coefficient, in order to double the size of d, as seen in Figure 2. Once we have calculated $\boldsymbol{x}_e$ and $f(\boldsymbol{x}_e)$ there are two possibilities:

   (a) $f(\boldsymbol{x}_e) < f(\boldsymbol{x}_r)$. In this case, our test was fruitful, and we should replace $\boldsymbol{x}_{n+1}$ (our worst point) with $\boldsymbol{x}_e$ (our best point), and then repeat the algorithm if we are still outside our tolerance.

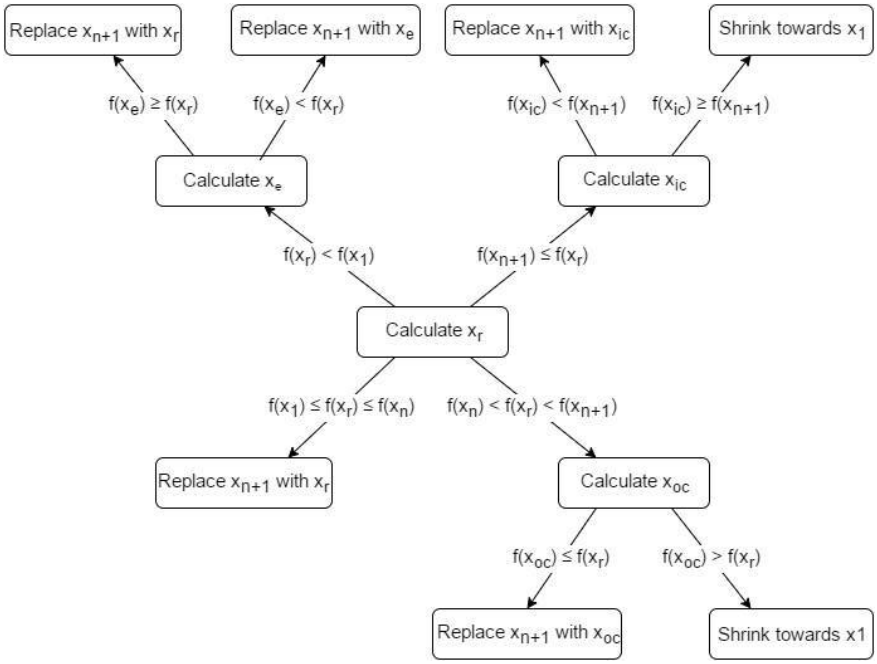

Figure 3: A visualisation of the Nelder-Mead Method.

(b) $f(\boldsymbol{x}_e) \geq f(\boldsymbol{x}_r)$. In this case, our test was not fruitful, and we should replace $\boldsymbol{x}_{n+1}$ (our worst point) with $\boldsymbol{x}_r$ (our best point), and then repeat the algorithm if we are still outside our tolerance.

2. $\boldsymbol{f(x_1) \leq f(x_r) \leq f(x_n)}$. Here, $\boldsymbol{x}_r$ was a good point, but not the best we have found so far. This means there is no point in exploring further out in this direction, so we replace $\boldsymbol{x}_{n+1}$ (our worst point) with the reflected point $\boldsymbol{x}_r$ and check if we are within our tolerance.

3. $\boldsymbol{f(x_n) < f(x_r) < f(x_{n+1})}$. *This step is visualised in Figure 4.* We have found that $\boldsymbol{x}_r$ is the second worst point that we have so far. At this point, we must have headed in the right direction, but we went too far, so we consider an outer contraction point, $\boldsymbol{x}_{oc}$ (as seen in Figure 4). The contraction point is along the same line from $\boldsymbol{x}_{n+1}$ to $\boldsymbol{x}_r$, except it is closer to the midpoint of points $\boldsymbol{x}_1$ to $\boldsymbol{x}_n$ than $\boldsymbol{x}_r$. This is where the third coefficient is used, the contraction coefficient. We use the value $\frac{1}{2}$, so that $\boldsymbol{x}_{oc}$ is exactly halfway in between $\boldsymbol{x}_r$ and the midpoint of points $\boldsymbol{x}_1$ to $\boldsymbol{x}_n$. Once we have calculated $\boldsymbol{x}_{oc}$ and $f(\boldsymbol{x}_{oc})$ there are two possibilities:

   (a) $f(\boldsymbol{x}_{oc}) \leq f(\boldsymbol{x}_r)$. This shows that the outer contraction point is equal to, or better than, the best point we have found so far. So in this case, we should replace $\boldsymbol{x}_n$ with $\boldsymbol{x}_{oc}$ and check whether we terminate the algorithm or not.

   (b) $f(\boldsymbol{x}_{oc}) > f(\boldsymbol{x}_r)$. This implies that the minimum of the function must be somewhere towards $\boldsymbol{x}_1$, so we make use of our last coefficient, the shrink coefficient. We move all points except for $\boldsymbol{x}_1$ towards $\boldsymbol{x}_1$ (by halving the distance they were from $\boldsymbol{x}_1$) and repeating the algorithm if we are still outside our tolerance. This step is visualised in Figure 5.

4. $\boldsymbol{f(x_n) \leq f(x_r)}$. *This step is visualised in Figure 4.* Here, checking $\boldsymbol{x}_r$ showed us that the function actually increases in this direction, so we should change tactics and check a point that is inside the simplex we started with. We do this by calculating the inner contraction point $\boldsymbol{x}_{ic}$, and evaluating the function at this point. Once we have calculated $\boldsymbol{x}_{ic}$ and $f(\boldsymbol{x}_{ic})$ there are two possibilities:

   (a) $f(\boldsymbol{x}_{ic}) < f(\boldsymbol{x}_n)$. The inner contraction point is more optimal than our worst point, so we should replace $\boldsymbol{x}_n$ with $\boldsymbol{x}_{ic}$ and check to see if we are within our tolerance.

   (b) $f(\boldsymbol{x}_{ic}) \geq f(\boldsymbol{x}_n)$. This means we are in a similar position as to in $3(b)$, and we see that the minimum of the function must be towards $\boldsymbol{x}_1$. So we shrink the distances of $\boldsymbol{x}_1$ to all other vertices by moving all vertices except $\boldsymbol{x}_1$ towards $\boldsymbol{x}_1$ (as visualised in Figure 5). We then check the tolerance and repeat the algorithm if necessary.

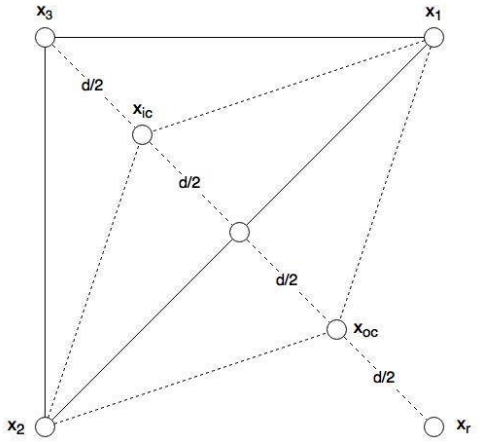

Figure 4: A visualisation of the inside contraction and outside contraction steps of the Nelder-Mead Method.

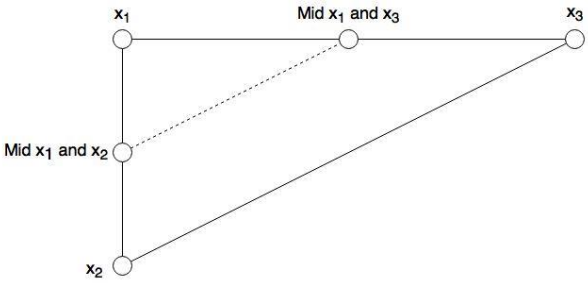

Figure 5: A visualisation of the shrink procedure in the Nelder-Mead Method.

3 Results and Discussion

In this section, we will state our results and analyse the performance of the different methods in obtaining these results. This will be done separately for each sub-problem. In producing these results we conducted a large number of trials varying the tolerance by a factor of 2 each time. We chose to do this as a decrease by an order of 2 corresponds to an additional bit of accuracy. The initial x- and y-coordinates for all calculations were randomly generated in the range $[0, 600]$.

Sub-problem 1: Council

The objective of this sub-problem is to appease the council by minimising the total length of the roads. Thus, the problem here is

$$\begin{aligned}\text{min. } c(x_1, y_1, x_2, y_2) = &\sqrt{(x_1 - 50.50)^2 + (y_1 - 207.35)^2} + \\ &\sqrt{(x_1 - 146.25)^2 + (y_1 - 324.60)^2} + \\ &\sqrt{(x_1 - 308.50)^2 + (y_1 - 320.15)^2} + \\ &\sqrt{(x_2 - 308.50)^2 + (y_2 - 320.15)^2} + \\ &\sqrt{(x_2 - 406.80)^2 + (y_2 - 190.45)^2} + \\ &\sqrt{(x_2 - 520.85)^2 + (y_2 - 47.20)^2} + \\ &\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} +\end{aligned}$$

where (x_1, y_1) are the coordinates of R1 and (x_2, y_2) are the coordinates of R2.

In solving this problem, we used 3 different numerical optimisation methods, all of which will be discussed individually before an overall comparison is provided.

Firstly, it should be noted that all of the methods gave the result (to varying degrees of accuracy):

Radio Station	x-coordinate	y-coordinate
R1	225.431	283.832
R2	406.800	190.450

with corresponding objective function value

$$c(225.431, 283.832, 406.800, 190.450) = 920.487.$$

The Steepest Descent Method was consistent and reliable. As is evident in Figure 6, the average number of iterations required to reach the solution was approximately linear with respect to the change in tolerance. Furthermore, the average computational time required for this tolerance was 0.0083s. Figures 7 depict similar results in terms of consistency with the average time taken varying linearly in terms of $-\log_2(\text{tolerance})$.

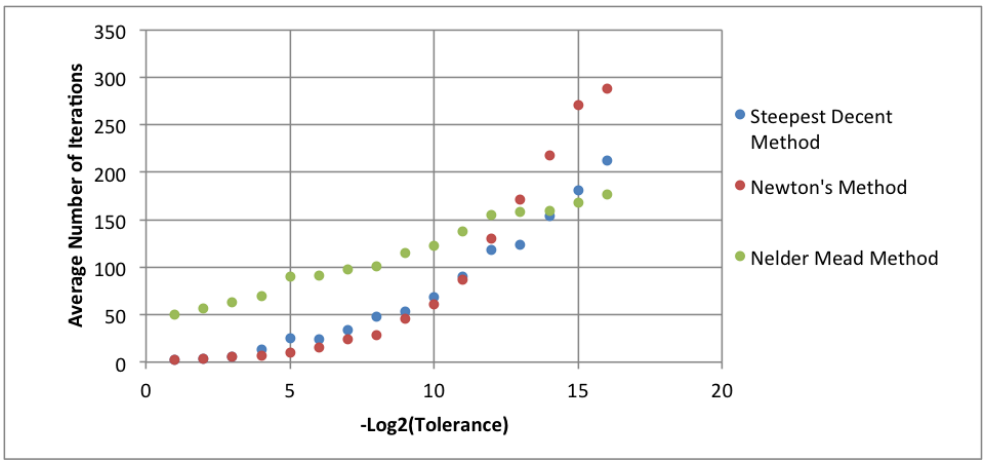

Figure 6: The average number of iterations for each of the 3 methods in solving the problem of minimising $c(x_1, y_1, x_2, y_2)$.

Newton's Method should in practice take less time to compute the optimal solution as it incorporates the use of second order information of the objective function. This allows it to select a more accurate descent direction compared to the Steepest Descent Method. However, this was not clear in the results. For example, as shown in Figure 7, the time taken for Newton's Method to compute the solution was significantly greater than that of the Steepest Descent or the Nelder-Mead Method. In particular, for a value of $x = 14$, Newton's Method required 0.0139s compared to 0.059s for the Steepest Descent and 0.012s for the Nelder Mead Method. The reason this occurred is because MATLAB was unable to calculate the inverse of the Hessian as it was very close to being a singular matrix and would have resulted in inaccurate results. Therefore, as a fall back solution the problem was coded to resort to the Steepest Descent Method to choose a descent direction. Thus, in the cases where this occurred Newton's Method was testing the viability of the Hessian before reverting to the Steepest Descent algorithm to select an appropriate descent direction. This is the reason why the results show Newton's Method taking longer in comparison to the other methods.

In Figure 7, we can see that there was one case where the time taken by Newton's Method was unusually large. In this case, the algorithm for Newton's Method was able to utilise the information of the Hessian but it is clear that this is expensive in terms of computational time. Further investigation into this showed that if the inner tolerance for the Golden Section Search was decreased then Newton's Method once again outperformed the Method of Steepest Descent. This was interesting as it suggested that an approximate second-order direction is less effective at minimising the objective than an approximate first-order direction.

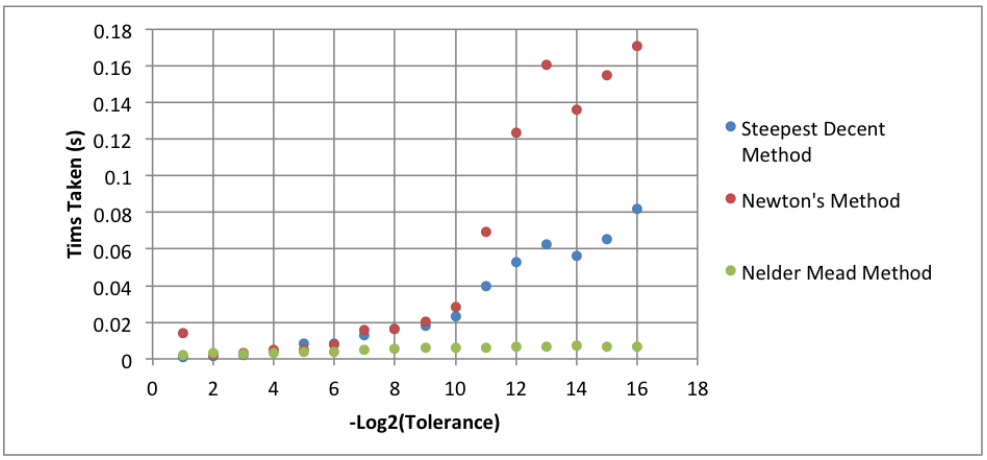

Figure 7: The average computational time for each of the 3 methods in solving the problem of minimising $c(x_1, y_1, x_2, y_2)$.

The Nelder-Mead Method was generally the most efficient in terms of time taken (as is shown in Figure 7). The average number of iterations required for the Nelder-Mead Method was initially greater than that of the Steepest Descent and Newton Methods (Figure 6 between $x = 1$ and $x = 12$). However, it is clear that for higher degrees of accuracy the Nelder-Mead Method used relatively less iterations to reach a solution.

In summary, the council's objective of minimising the total length of all roads was solved using 3 different numerical optimisation techniques. The Nelder-Mead Method was the most efficient in solving this problem as it consistently required less computational time to determine the optimal solution. It also required less iterations than the other methods as the tolerance grew smaller. Newton's Method was not useful as in most cases MATLAB was unable to calculate the inverse of the Hessian. Further, in the case where the Hessian was used to develop a descent direction it is evident that this process is computationally expensive in terms of the time taken. The Method of Steepest Descent was consistent but generally inferior to the Nelder-Mead Method.

Sub-problem 2: Managers

The objective of this sub-problem is to appease the managers by making the radio stations as physically accessible as possible. Thus, the problem here is

$$\begin{aligned} \text{min. } m(x_1, y_1, x_2, y_2) = max\{&\sqrt{(x_1 - 50.50)^2 + (y_1 - 207.35)^2}, \\ &\sqrt{(x_1 - 146.25)^2 + (y_1 - 324.60)^2}, \\ &\sqrt{(x_1 - 308.50)^2 + (y_1 - 320.15)^2}\} + \\ max\{&\sqrt{(x_2 - 308.50)^2 + (y_2 - 320.15)^2}, \\ &\sqrt{(x_2 - 406.80)^2 + (y_2 - 190.45)^2}, \\ &\sqrt{(x_2 - 520.85)^2 + (y_2 - 47.20)^2}\} \end{aligned}$$

where (x_1, y_1) are the coordinates of R1 and (x_2, y_2) are the coordinates of R2.

In solving this problem, we used 3 different numerical optimisation methods, all of which will be discussed individually before an overall comparison is provided.

Firstly, it should be noted that all of the methods gave the result (to varying degrees of accuracy):

Radio Station	x-coordinate	y-coordinate
R1	179.487	263.781
R2	414.656	183.660

with corresponding objective function value

$$m(179.487, 263.781, 414.656, 183.660) = 313.702.$$

For this sub-problem, the gradient and Hessian reliant methods were significantly slower than the Nelder-Mead Method which simply needed to evaluate the function.

The first significant fact that could be gleamed off the data is that the Hessian of this objective function was ill-conditioned in almost the entire region in the local vicinity of the minimum. We can see this from the data (Figure 8), as Newton's Method and the Method of Steepest Descent take an identical number of iterations for almost all trials at all tolerances. This indicates that at each iteration Newton's Method calculated the Hessian, found its condition number and decided it was simply too ill-conditioned to invert (without the introduction of considerable numerical error) then proceeded to use the Method of Steepest Descent.

The convergence of the Method of Steepest Descent was also poorly affected by the Hessian being near singular quite commonly. The ill-conditioned nature of the Hessian, as mentioned in the theoretical discussion, caused the Steepest Descent Method to take a step in a direction that was nearly orthogonal to the direction of the global minima at each iteration, which led to slower convergence.

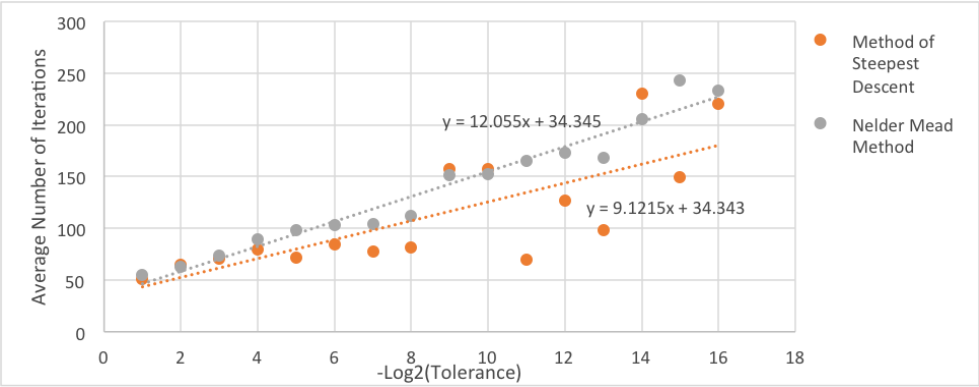

Figure 8: The average number of iterations for the Steepest Descent Method and Nelder-Mead Method in solving the problem of minimising $m(x_1, y_1, x_2, y_2)$.

Comparing the number of iterations of the Nelder-Mead Method with the Method of Steepest Descent we can see that initially at high tolerances in the range $[2^{-1}, 2^{-7}]$ they required a similar number of iterations, after which point the Steepest Descent Method began to require less. The linear trend lines plotted on the graph above suggest that the Method of Steepest Descent required on average 9.12 additional iterations

and the Nelder-Mead Method required on average 12.06 additional iterations for each additional bit of accuracy.

The similar number of iterations at low tolerances and the growing difference at higher tolerances is supported by the nature of the methods. Initially the Nelder-Mead Method starts with information on the value of the function on all the corners of a simplex, whilst the Method of Steepest Descent starts with just the value of the function and its gradient at a point. This initial advantage in information though is outstripped by the fact that at each iteration the Nelder-Mead Method can at most learn the value of the function at two new points, whereas the Steepest Descent Method learns the value of the function and its gradient at a point, which intuitively imparts greater knowledge about the underlying shape of the function. Thus the greater gain in information on a per iteration basis leads to the Method of Steepest Descent outperforming the Nelder-Mead Method on the total number of iterations at lower tolerances.

Figure 9 illustrates just how much the Nelder-Mead Method outperformed the methods that were reliant on the nature of the Hessian for the problem of minimising objective function $m(x_1, y_1, x_2, y_2)$. We can see that Newton's Method took the longest for all tolerances, which was expected as at each iteration it calculated a significant amount of extra information only to find out that it could not use it.

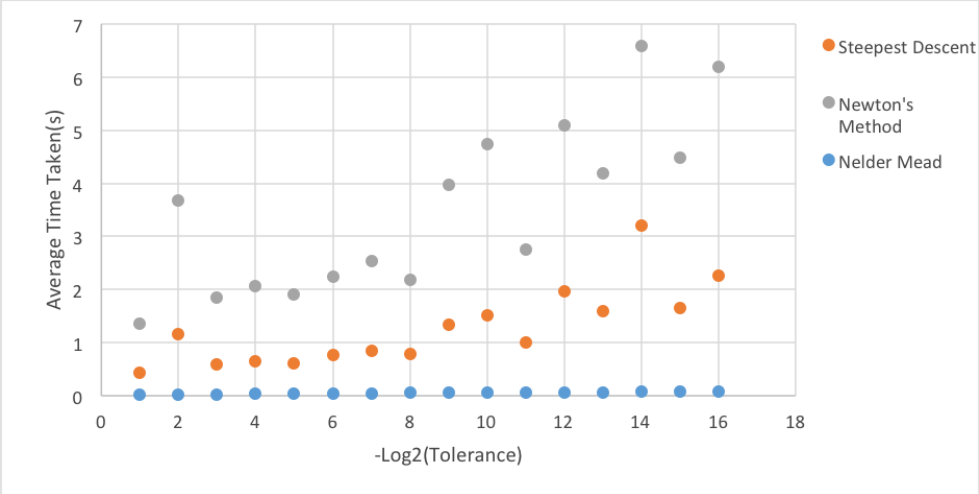

Figure 9: The average computational time for each of the 3 methods in solving the problem of minimising $m(x_1, y_1, x_2, y_2)$.

As the Nelder-Mead Method and the Method of Steepest Descent had relatively similar number of iterations in comparison to their difference in average time taken, we can infer that the Nelder-Mead method had a significantly faster time per iteration.

Thus for objective function $m(x_1, y_1, x_2, y_2)$, the Nelder-Mead Method had the best performance. This can be attributed almost entirely to the ill-conditioned nature of the Hessian, which rendered Newton's Method useless and compromised the performance of the Method of Steepest Descent by forcing it to take steps almost orthogonal to the direction of the global minima.

Sub-problem 3: Owners

The objective of this sub-problem is to appease the owners by minimising power costs, where the power cost required is proportional to the square of the distance the signal travels from the radio station. Station R1 will transmit to A, B and C, while Station R2 will transmit to B, C, D and E. Thus, the problem here is

$$\begin{aligned}\text{min. } o(x_1, y_1, x_2, y_2) =&(x_1 - 50.50)^2 + (y_1 - 207.35)^2 + \\ &(x_1 - 146.25)^2 + (y_1 - 324.60)^2 + \\ &(x_1 - 308.50)^2 + (y_1 - 320.15)^2 + \\ &(x_2 - 146.25)^2 + (y_2 - 324.60)^2 + \\ &(x_2 - 308.50)^2 + (y_2 - 320.15)^2 + \\ &(x_2 - 406.80)^2 + (y_2 - 190.45)^2 + \\ &(x_2 - 520.85)^2 + (y_2 - 47.20)^2\end{aligned}$$

where (x_1, y_1) are the coordinates of R1 and (x_2, y_2) are the coordinates of R2.

In solving this problem, we used 3 different numerical optimisation methods, all of which will be discussed individually before an overall comparison is provided.

Firstly, it should be noted that all of the methods gave the result (to varying degrees of accuracy):

Radio Station	x-coordinate	y-coordinate
R1	168.417	284.033
R2	345.600	220.600

with corresponding objective function value

$$o(168.417, 284.033, 345.600, 220.600) = 170127.$$

It can be concluded that the Steepest Descent Method and Newton's Method outperformed the Nelder-Mead Method in solving this sub-problem, while a case could be made for each of these methods as the best depending on whether number of iterations or computational time is most valued.

Comparing the number of iterations performed by each of the methods, we can see that the Nelder-Mead method took significantly more iterations in order to find a solution than the other two methods. This is due to the fact that first- and second-order information was easily calculable (i.e. the gradient and Hessian of $o(x_1, y_1, x_2, y_2)$) so the number of iterations required by each of the other two methods was significantly less than when solving the other sub-problems. When comparing the number of iterations required by the Nelder-Mead Method in solving this sub-problem to the number of iterations it required in solving Sub-problems 1 & 2, it can be seen that they do not differ significantly. That is, the Nelder-Mead Method performed no worse in solving this problem than it did in solving the others, it is simply the case that the other two methods performed significantly better.

While the number of iterations required by the Steepest Descent Method and Newton's Method were very similar, Newton's Method slightly outperformed the Steepest Descent in this area. This can be attributed to the fact that at some iterations Newton's Method defaults to the Method of Steepest Descent (when the Hessian matrix is ill-conditioned and unable to be inverted without introducing significant error). In

effect, Newton's Method is identical to the Steepest Descent Method for a particular iteration only when it cannot perform a more effective iteration without having to use Steepest Descent. Thus, the number of iterations taken by Newton's Method can be expected to be at most the number of iterations taken by the Steepest Descent Method.

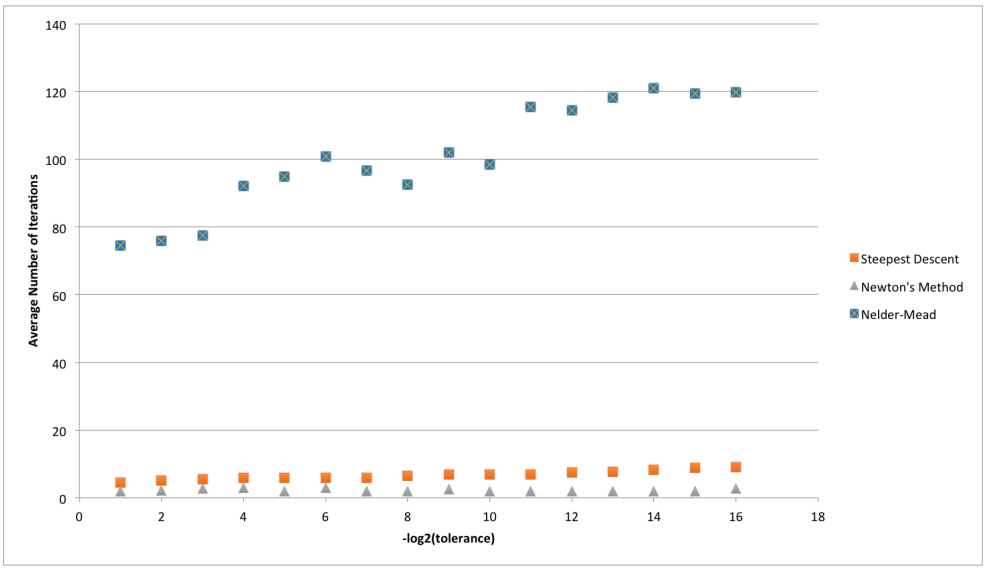

Figure 10: The average number of iterations for each of the 3 methods in solving the problem of minimising $o(x_1, y_1, x_2, y_2)$.

Now, when comparing the methods based on computational time, the Steepest Descent Method can be seen to be the best performing. It consistently solved the problem faster than the other two methods, as can be easily seen in Figure 11. Based on this measure, the only indicator that Steepest Descent is not the best performing method is the fact that the line of best fit for its data points has a slightly steeper gradient than the other lines of best fit. This indicates that should you continue to increase precision (i.e. lower the tolerance value) the Steepest Descent Method will eventually perform slower than the other methods. However, since the methods have been tested to quite high precision and the slope is only slightly steeper while the value of the data points is still noticeably lower, it is reasonable to conclude that the Steepest Descent Method performs the best.

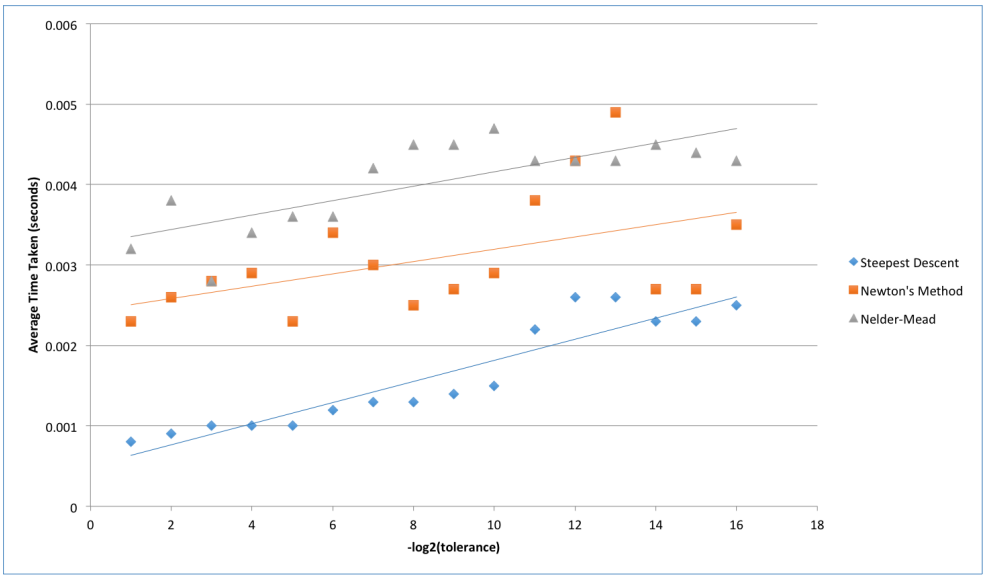

Figure 11: The average computational time for each of the 3 methods in solving the problem of minimising $o(x_1, y_1, x_2, y_2)$.

In conclusion, the Steepest Descent Method can be seen to be the most effective for solving this sub-problem due to the fact that it took significantly less time than the other methods while only required slightly more iterations than Newton's Method.

4 Compromise Solution

So far, we have proposed efficient locations of the two radio stations with respect to each objective function separately. In this section of the report, we intend to analyse the problem with respect to all three conflicting interests simultaneously to form a compromise solution.

To combine the objective functions, we will use five different methods. Firstly, we will model the problem as three different constrained non-linear programs and use the l_2-penalty method in order to solve them. Then we will develop a cost function that incorporates the respective real-world costs associated with obtaining each objective. Finally, we will use economic welfare functions to strike a compromise between the three parties. The welfare functions will be: the sum of the normalised objective functions, the max of the normalised objective functions and a variant of Sen's function.

In order to ascertain the optimal locations of Radio Station 1 and 2 for these compromise functions, we need to use a numerical method that doesn't require the gradient of the underlying function and can determine the global minima of a potentially non-convex region. The numerical method we have chosen to use is the Particle Swarm Method.

Particle Swarm Method

Particle Swarm Optimisation is an optimisation method for non-linear functions that does not require the gradient. It was inspired by basic social models of bird-flocking that were observed to be self-optimising. In essence the algorithm works by having a

large number of 'birds' fly around the solution space. Initially their velocity is random however if a bird finds a new deposit of 'food', more than it has previously found, it remembers this location and tells nearby birds. At each subsequent iteration a bird's velocity is modified by the location of the largest deposit of food it has found and the largest deposit of food its neighbours have found. In this manner all birds tend to eventually move to the largest deposit of food.

Particle Swarm Optimisation is a relatively simple algorithm that as the name would suggest employs swarm intelligence. It is of common consensus that swarm intelligence displays five basic properties [6]:

1. **Proximity Principle:** Each unit of the swarm is capable of making basic calculations with regard to its immediate environment.

2. **Quality Principle:** The swarm in its entirety should be able to respond to quality factors.

3. **Principle of Diverse Response:** The swarm (at least initially) should not commit its entire population to a small region.

4. **Principle of Stability:** The swarm should not change its overarching behaviour for minor environment changes.

5. **Principle of Adaptability:** The swarm should be able to change its overarching behaviour due to a change in environment, if the cost of the change is computationally worth it.

Note: In regards to actual implementation, the options of the inbuilt particleswarm function of MATLAB were altered such that the algorithm closely resembled that which was discussed in the paper published by John Kennedy and Russell Eberhart in 1995 [7]. However, the adaptive inertia of the MATLAB implementation was kept as it was found to drastically improve the performance of the algorithm. Adaptive inertia allows the swarm to more finely search an area once it nears an optimum.

Objective Functions

1: l_2-penalty function

The first method we will use to combine the three objective functions from the sub-problems outlined in Section 3 is modelling the problem as a constrained optimisation problem. We will do so in three different ways:

1. We will minimise the objective function of sub-problem 1 $c(\boldsymbol{x})$ subject to the condition that the objective functions of the other two sub-problems ($m(\boldsymbol{x})$ and $o(\boldsymbol{x})$) take values no more than 7% more than the optimum values calculated previously. Note that we have allowed the objective functions to be no more than 5% in the other problems. The reason why we have chosen 7% here is because the problem has no feasible solution if 5% is used.

$$\begin{aligned} \min \quad & c(x_1, y_1, x_2, y_2) \\ \text{s.t.} \quad & m(x_1, y_1, x_2, y_2) \leq 1.07 \times 313.702 \\ & o(x_1, y_1, x_2, y_2) \leq 1.07 \times 170127 \end{aligned}$$

2. We will minimise the objective function of sub-problem 2 $m(\boldsymbol{x})$ subject to the condition that the objective functions of the other two sub-problems ($c(\boldsymbol{x})$ and $o(\boldsymbol{x})$) take values no more than 5% more than the optimum values calculated previously.

$$\begin{aligned} \min \quad & m(x_1, y_1, x_2, y_2) \\ \text{s.t.} \quad & c(x_1, y_1, x_2, y_2) \leq 1.05 \times 920.487 \\ & o(x_1, y_1, x_2, y_2) \leq 1.05 \times 170127 \end{aligned}$$

3. We will minimise the objective function of sub-problem 3 $o(\boldsymbol{x})$ subject to the condition that the objective functions of the other two sub-problems ($c(\boldsymbol{x})$ and $m(\boldsymbol{x})$) take values no more than 5% more than the optimum values calculated previously.

$$\begin{aligned} \min \quad & o(x_1, y_1, x_2, y_2) \\ \text{s.t.} \quad & c(x_1, y_1, x_2, y_2) \leq 1.05 \times 920.487 \\ & m(x_1, y_1, x_2, y_2) \leq 1.05 \times 313.702 \end{aligned}$$

The l_2-penalty functions for the above problems are as follows:

1.

$$P_\alpha^1(x) = c(\boldsymbol{x}) + \frac{\alpha}{2}((m(\boldsymbol{x}) - 1.07 \times 313.702)_+)^2 + ((o(\boldsymbol{x}) - 1.07 \times 170127)_+)^2$$

2.

$$P_\alpha^2(x) = m(\boldsymbol{x}) + \frac{\alpha}{2}((c(\boldsymbol{x}) - 1.05 \times 920.487)_+)^2 + ((o(\boldsymbol{x}) - 1.05 \times 170127)_+)^2$$

3.

$$P_\alpha^3(x) = o(\boldsymbol{x}) + \frac{\alpha}{2}((c(\boldsymbol{x}) - 1.05 \times 920.487)_+)^2 + ((m(\boldsymbol{x}) - 1.05 \times 313.702)_+)^2$$

2: Cost Function

We will now develop a cost function that encapsulates the real-world expenses of building and maintaining the required roads between the radio stations and towns. Intuitively, the cost function should be dependent on two variables: time and total length of the road. For simplicity, we will fix the time variable to be 10 years in order to remove the dependence on time of the cost function. We will assume that the road being built is two lanes in width with no median barrier. We have reached out to professionals working in industry [8] and have received a quote that indicated that the cost of building such a road is approximately AUD$1 million per kilometre. In addition, the maintenance costs were quoted at being 10% of the capital investment per annum.

Objective 3 involves minimizing the total power costs associated with operating the radio stations. We assume that the radio stations are not operated using internal power generators and that they need to be connected to the grid using overhead power lines. The cost to construct a 22kV overhead power line is approximately $120,000 per

kilometre [3]. These costs can be expressed in terms of objective 1. The operational and maintenance costs were quoted to be minimal so these will be excluded from the report. Thus, as we found the transmission costs to be minimal relative to the other costs, we will exclude objective 3 from the cost function.

Furthermore, we will assume that each radio station will employ 10 people. These employees are compensated for their travel time at a rate of approximately \$0.50 per kilometre of travel. The number of working days in a year (in Australia) is 261, which is another factor that is incorporated into the equation. By compensating the employees equally based off the maximum distance they could potentially have to travel, we can express these costs in terms of objective 2. These assumptions provide us with a basis to construct the following cost function:

$$\begin{aligned} cost(\boldsymbol{x}) &= (1,000,000 + 100,000 \times 10 + 120,000)c(\boldsymbol{x}) + (10 \times 0.50 \times 261)m(\boldsymbol{x}) \\ &= 2,120,000c(\boldsymbol{x}) + 1305m(\boldsymbol{x}) \end{aligned}$$

where $c(\boldsymbol{x})$ is the objective function of sub-problem 1 and $m(\boldsymbol{x})$ is the objective function of sub-problem 2.

3: Sum of the Normalised Objective Functions

In order to use welfare functions to reach a compromise it is necessary to express the objective functions in equivalent units. The natural way to do so is to normalise each objective function relative to their minimums as found previously (when each objective was being minimised on its own).

This welfare function aims to reach an optimal solution where the sum of the normalised objective functions is minimised. This essentially means that the total residual of the individual objective functions from their own respective optimal solutions will be minimised. In practice, this is an efficient method. However, this method of summation places no particular emphasis on equality, for instance if the function can decrease the value of the lowest objective function at the expense of increasing the value of the largest objective function, it will do so if the formers decrease outweighs the latters increase. Naturally this can lead to one party being better off than another. Therefore, it is likely that the solution provided by this method will not be envy-free for the competing parties. This objective function is:

$$sumOfNorm(\boldsymbol{x}) = \frac{c(\boldsymbol{x})}{c^*(\boldsymbol{x})} + \frac{m(\boldsymbol{x})}{m^*(\boldsymbol{x})} + \frac{o(\boldsymbol{x})}{o^*(\boldsymbol{x})}$$

where $c(\boldsymbol{x})$, $m(\boldsymbol{x})$ and $o(\boldsymbol{x})$ are the objective functions of sub-problems 1, 2 and 3 respectively, and $c^*(\boldsymbol{x})$, $m^*(\boldsymbol{x})$ and $o^*(\boldsymbol{x})$ are the optimal values of these functions calculated in Section 3.

4: Max of the Normalised Objective Functions

The general idea behind using this method is that the optimal solution is only as good as the weakest link. By minimising the max function, the algorithm will focus solely on the objective function that is least well off. It should be noted that this focus usually comes at the expense of the other objectives. Therefore, this method may not minimise the sum of the objective functions but it will decrease the spread of the normalised

functions. Thus, the solution provided using this method places a strong emphasis on the equality of the parties. This objective function is:

$$maxOfNorm(\boldsymbol{x}) = max\{\frac{c(\boldsymbol{x})}{c^*(\boldsymbol{x})}, \frac{m(\boldsymbol{x})}{m^*(\boldsymbol{x})}, \frac{o(\boldsymbol{x})}{o^*(\boldsymbol{x})}\}$$

5: Variant of the Sen Function

Amartya Sen proposed a welfare function in 1973 [11] that addressed both the issue of equality and that of maximising the total welfare of society. It makes use of the Gini index [1], which is a relative inequality measure:

$$G = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n}|x_i - x_j|}{2n^2\mu}$$

and

$$sen = meanOfNorm(\boldsymbol{x})(1 - G).$$

A Gini-Coefficient of 1 corresponds with perfect inequality and a Gini-Coefficient of 0 corresponds with perfect equality. Sen's function was concerned with maximising the welfare of society, whilst our objective is to minimise the sum of the normalised objective functions whilst maintaining equality. Thus Sen's function was altered as follows to suit our needs:

$$senVariant(\boldsymbol{x}) = \frac{-1}{meanOfNorm(\boldsymbol{x})(1 - G)}$$

Results/Analysis

The minimums of the above objective functions, found using particle swarm optimisation, were

	R1 x-coord	R1 y-coord	R2 x-coord	R2 y-coord
P_α^1	172.4126	279.9606	391.5746	191.6324
P_α^2	173.2491	278.0488	377.7820	188.2555
P_α^3	173.9758	276.3853	386.1954	178.3030
$cost$	225.4200	283.8757	400.0000	194.2980
$sumOfNorm$	173.4541	277.5783	394.7131	187.2983
$maxOfNorm$	174.3311	275.5720	383.6625	183.7902
$senVariant$	174.3347	275.5642	400.0000	185.6883

In order to analyse the nature of the different compromises proposed by the four functions we produced Figures 12 and 13. These figures show the value of the normalised objective functions at the minimum found by each of the compromise methods.

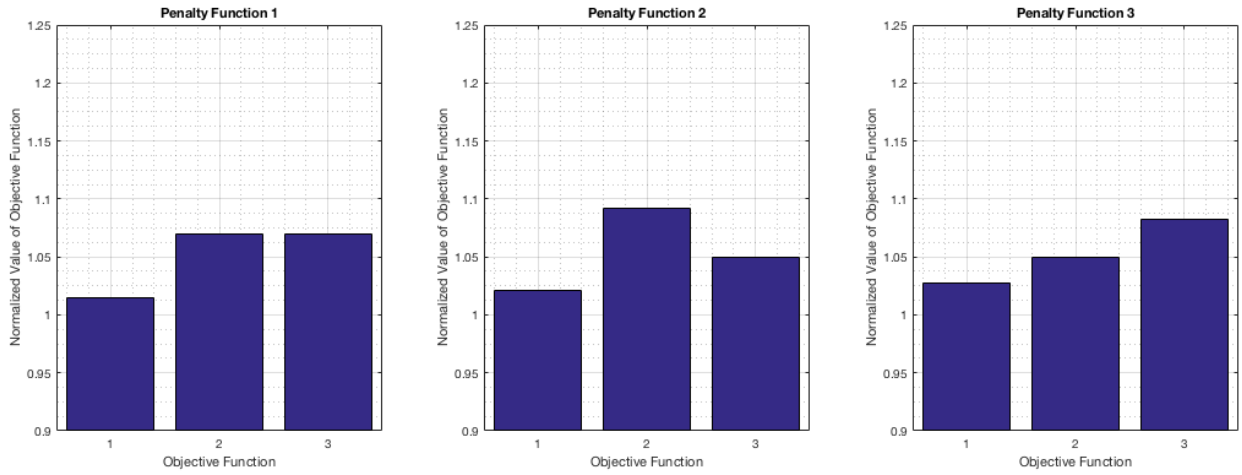

Figure 12: The value of the normalised objective functions at the minimum found by each of the penalty methods.

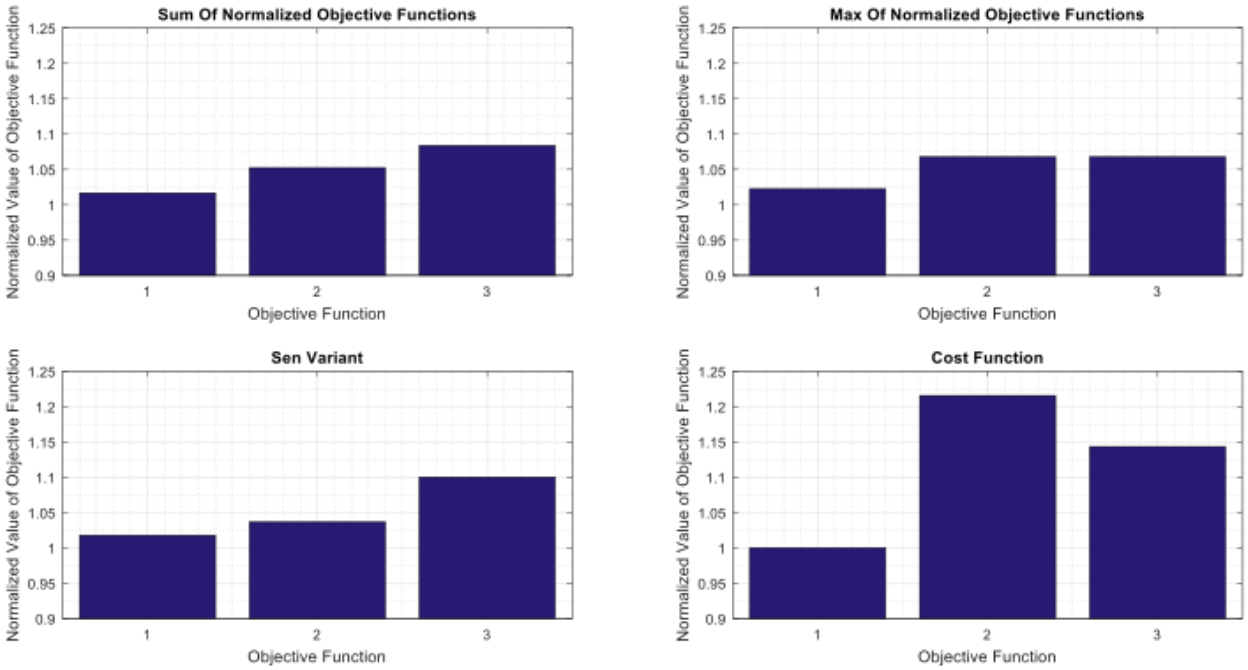

Figure 13: The value of the normalised objective functions at the minimum found by each of the other four compromise methods.

Obviously there was not a solution that was able to attain the optimal solution for all of the objective functions individually (this would be represented by a normal of 1 for each of the objective functions).

None of the penalty methods clearly outperformed any of the others, however it can be argued that Penalty Function 1 returned the most optimal point out of them all. This could be due to the fact that we extended the feasible space by relaxing the

constraints in the objective function; a larger feasible region would lead one to expect a more optimal point.

It is easy to think that the function we are minimising in the constrained problem will always end up close to its optimal value when considered as an unconstrained problem. However, Figure 12 shows that this is not the case. In the graph for Penalty Functions 2 and 3, we can see that the function being minimised in the constrained problem in each case ($m(\boldsymbol{x})$ and $o(\boldsymbol{x})$ respectively) is in fact the function which has the greatest function value relative to its optimum value. The fact that $m(\boldsymbol{x})$ and $o(\boldsymbol{x})$ saw such great relative increases lends weight to the argument that Penalty Function 1 provides the optimal solution.

When considered from an equitable point of view, Penalty Function 1 provides the most desirable solution. This is the case since it results in the smallest maximum relative increase for the three objective functions $c(\boldsymbol{x})$, $m(\boldsymbol{x})$ and $o(\boldsymbol{x})$ out of the three penalty functions.

Comparing the value of the objective functions at the minimising locations of the economic welfare functions, we can see some interesting results. Firstly, the Sum of the Normalised Objective Functions appears to have the smallest total area in the bar graph, which makes sense given that this was what was being minimised for this function. However, this smaller total area came at slight expense to the objective function of sub-problem 2 $m(\boldsymbol{x})$ and a larger expense to the objective function of sub-problem 3 $o(\boldsymbol{x})$.

The Max of the Normalised Objectives partially addressed this, lowering the objective function of sub-problem 3 $o(\boldsymbol{x})$ and raising the objective function of sub-problem 2 $m(\boldsymbol{x})$ until they were at par, with a relative increase in the objective function of sub-problem 1 $o(\boldsymbol{x})$ as well.

Interestingly the Sen Variant welfare function was minimised when the objective function of sub-problem 2 $m(\boldsymbol{x})$ was lowered at the expense of the objective function of sub-problem 3 $o(\boldsymbol{x})$. It appears that the change in the Gini-Coefficient brought on by the greater equality of the objective functions of sub-problems 1 & 2 offset the increase in the objective function of sub-problem 3.

The cost function was quite different to the welfare function. As was discussed earlier it only included the objective functions of sub-problems 1 & 2 and aimed to optimise these with respect to the costs associated with achieving them. Of these costs, the coefficient of the objective function of sub-problem 1 dominated that of the objective function of sub-problem 2, so we can see that the minimum of the cost function was essentially the minimum of the objective function of sub-problem 1 itself.

Therefore, in conclusion, each method resulted in a different compromise, as expected. The particle swarm numerical method performed well and the results we obtained seemed reasonable given the imprecise nature of a compromise solution.

5 Conclusion

Although upon first glance, one may assume that finding an efficient solution to this problem would be an easy task, we have shown throughout this report that this would be naive. There are many factors that need to be considered and a variety of numerical methods that can be implemented, each one having both useful and disadvantageous properties.

Firstly, we implemented three different numerical methods to obtain a solution for each individual party. We found that all of the methods that were applied gave a consistent optimal solution for all the cases. However, the efficiency and practicality of each method differed on numerous occasions. We went on to further analyse the reasons for this.

Lastly, we analysed the problem with respect to all three conflicting interests simultaneously to form a compromise solution. We took a number of different approaches in doing this. We first converted it to a series of constrained problems with different objectives and solved these, before considering the problem from a welfare standpoint and looking to solve it in this way. Finally, we considered the monetary costs involved and modelled the problem so as to minimise these.

We have presented a number of possible solutions to the problem. All of these solutions are viable, however the best solution to choose will depend entirely on the needs of the parties.

References

[1] Gini coefficient. `http://mathworld.wolfram.com/GiniCoefficient.html`. Accessed: 2016-05-20.

[2] Oleg Alexandrov. `https://upload.wikimedia.org/wikipedia/commons/d/da/Newton_optimization_vs_grad_descent.svg`. Accessed: 2016-05-20.

[3] Bertram Birk. Power and Water Northern Territory. Personal communication.

[4] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[5] Fuchang Gao and Lixing Hand. Implementing the Nelder-Mead simplex algorithm with adaptive parameters. *Computational Optimization & Applications*, 51:259–277, 2012.

[6] Yichen Hu. Swarm intelligence.

[7] James Kennedy and Russell Eberhart. Particle swarm optimization. *Proc. IEEE International Conf. on Neural Networks*, 4:1942 – 1948, 1995.

[8] Terry Layman. Northern Territory Government Department of Transport. Personal communication.

[9] J. A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7:308 – 313, 1965.

[10] Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, 2006.

[11] Amartya Sen. *On economic inequality*. Oxford University Press, 1973.

Appendices

Algorithm 1 Steepest Descent

1: **select** $\boldsymbol{x}^0 \in \mathbb{R}^n$.
2: **set** $k = 0$.
3: **calculate** $\boldsymbol{d}^k = -\nabla f(\boldsymbol{x}^k)$.
4: **if** $||\boldsymbol{d}^k|| < \epsilon$ **then** stop.
5: **select** step length t_k by solving the single-variable minimisation problem: min $q(t) = f(\boldsymbol{x}^k + t_k \boldsymbol{d}^k)$.
6: **set** $k = k + 1$.
7: **set** $\boldsymbol{x}^{k+1} = \boldsymbol{x}^k + t_k \boldsymbol{d}^k$.
8: Return to Step 3.

Algorithm 2 Newton's Method

1: **select** $\boldsymbol{x}^0 \in \mathbb{R}^n$.
2: **set** $k = 0$.
3: **if** $||\nabla f(\boldsymbol{x}^k)|| < \epsilon$ **then** stop.
4: **if** $\nabla^2 f(\boldsymbol{x}^k)$ is positive define **then**
5: **set** $\boldsymbol{d}^k = -\nabla^2 f(\boldsymbol{x}^k)^{-1} \nabla f(\boldsymbol{x}^k)$
6: **else set** $\boldsymbol{d}^k = -\nabla f(\boldsymbol{x}^k)$
7: **select** step length t_k by solving the single-variable minimisation problem: min $q(t) = f(\boldsymbol{x}^k + t\boldsymbol{d}^k)$
8: **set** $k = k + 1$.
9: **set** $\boldsymbol{x}^{k+1} = \boldsymbol{x}^k + t_k \boldsymbol{d}^k$.
10: Return to Step 3.

Algorithm 3 Nelder-Mead

1: **order** according to the values at the vertices:
$f(\boldsymbol{x}_1) \leq f(\boldsymbol{x}_2) \leq \ldots \leq f(\boldsymbol{x}_{n+1})$.
2: **if** $f(\boldsymbol{x}_{n+1}) - f(\boldsymbol{x}_1) <$ tolerance **then** stop.
3: **calculate** $\boldsymbol{x}_0$, the centroid of all points except $\boldsymbol{x}_{n+1}$.
4: **compute** reflected point $\boldsymbol{x}_r = \boldsymbol{x}_0 + \alpha(\boldsymbol{x}_0 - \boldsymbol{x}_{n+1})$.
5: **if** $f(\boldsymbol{x}_1) \leq f(\boldsymbol{x}_r) \leq f(\boldsymbol{x}_n)$ **then** let $\boldsymbol{x}_{n+1} = \boldsymbol{x}_r$ and go to Step 1.
6: **if** $f(\boldsymbol{x}_r) < f(\boldsymbol{x}_1)$ **then compute** expanded point $\boldsymbol{x}_e = \boldsymbol{x}_0 + \gamma(\boldsymbol{x}_r - \boldsymbol{x}_0)$
7: **if** $f(\boldsymbol{x}_e) < f(\boldsymbol{x}_r$ **then** let $\boldsymbol{x}_{n+1} = \boldsymbol{x}_e$ and go to Step 1.
8: **else** let $\boldsymbol{x}_{n+1} = \boldsymbol{x}_r$ and go to Step 1.
9: **else** Go to Step 10.
10: **compute** contracted point $\boldsymbol{x}_c = \boldsymbol{x}_0 + \rho(\boldsymbol{x}_{n+1} - \boldsymbol{x}_0)$.
11: **if** $f(\boldsymbol{x}_c < f(\boldsymbol{x}_{n+1})$ **then** let $\boldsymbol{x}_{n+1} = \boldsymbol{x}_c$ and go to Step 1.
12: **else** Go to Step 13.
13: **for** all $i \in \{2, \ldots, n+1\}$ **do** $\boldsymbol{x}_i = \boldsymbol{x}_1 + \sigma(\boldsymbol{x}_i - \boldsymbol{x}_1)$ and go to Step 1.

Algorithm 4 Particle Swarm

set $k = 0$
place particles of swarm across search space with random initial velocities v.
for particle p in swarm **do**
    set $x_{pb}(p) = x_k(p)$ (the personal best location of the particle is set to be its current location).
    set x_{gb} to the location of the best performing particle from initialisation.
while change in global minima $>$ tolerance and a new minimum has been found recently **do**
    for particle p in swarm **do**
        set $a, b = rand(0, 1)$.
        set the velocity of particle: $v = inertia \times v + 2a(x_{pb}(p) - x_k(p)) + 2b(x_{gb} - x_k(p))$.
        update position of particle.
        check if it is a new personal minimum.
        if a new global minimum **then**
            record position of global minimum.
            increase inertia if below max inertia.
        else decrease initial inertia if above min inertia.
        set $k = k + 1$.

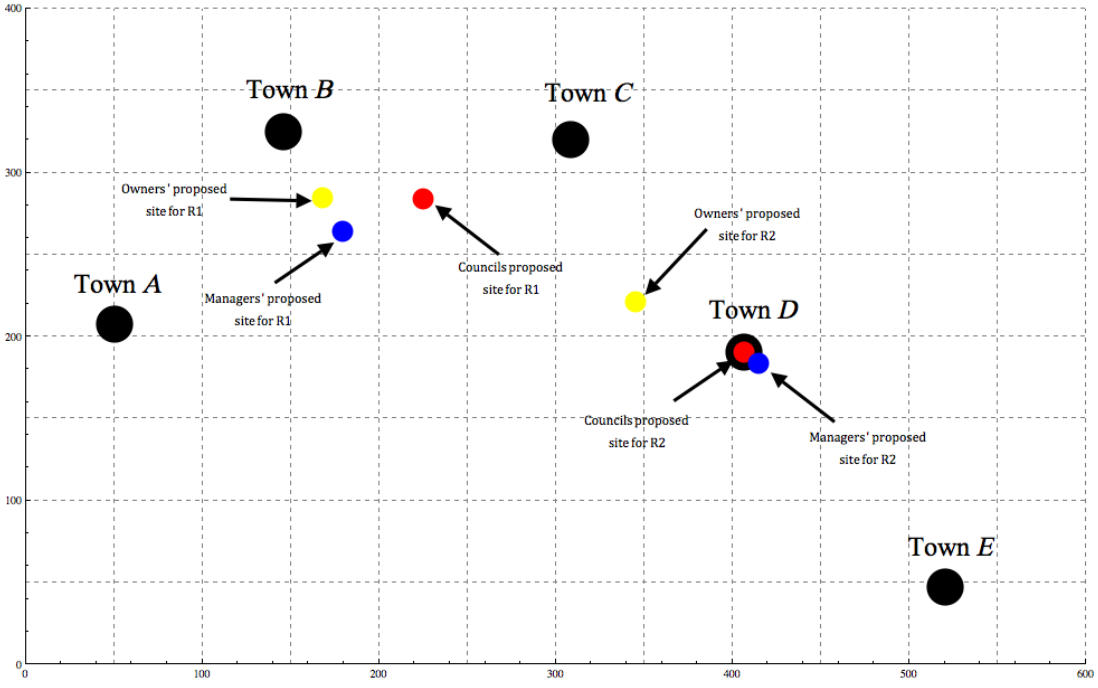

Figure 14: A visual depiction of the proposed radio station sites from sub-problems 1, 2 and 3.

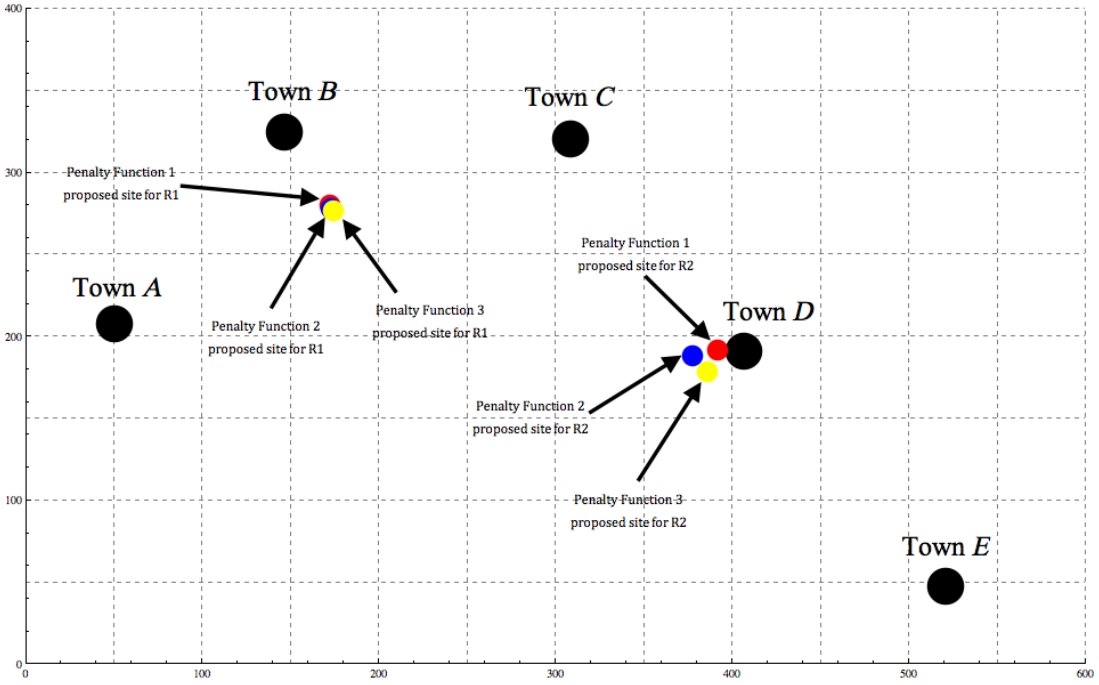

Figure 15: A visual depiction of the proposed radio station sites from the three penalty functions.

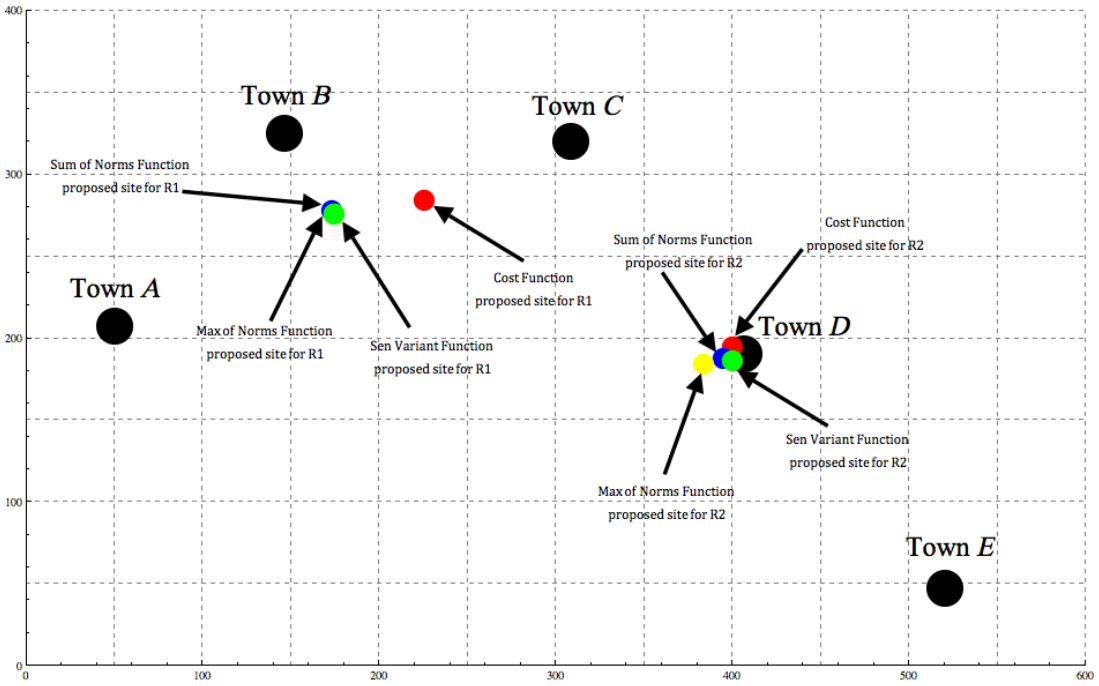

Figure 16: A visual depiction of the proposed radio station sites from the other four objective functions for the compromise solution.

MATLAB file *Compromises.m*:

```
close all
objectiveFunctions
%minimums found using ParticleSwarm function for each objective.
norm_sum_min_x = [173.4541  277.5783  394.7131  187.2983]
norm_max_min_x = [174.3311  275.5720  383.6625  183.7902]
sen_min_x = [174.3347  275.5642  400.0000  185.6883]
CostMethod_x = [225.4200  283.8757  400.0000  194.2980]

x_mins = [norm_sum_min_x;norm_max_min_x;sen_min_x;CostMethod_x]

ax1 = subplot(1,2,1);
ax2 = subplot(1,2,2);
hold(ax1,'on')
hold(ax2,'on')

handles = {};

%Plots the location of each radio station for each compromise.
for i=1:length(x_mins)
    ax1 = subplot(1,2,1);
    handles{i} = plot(x_mins(i,1),x_mins(i,2),'x','linewidth',2);
    if i==1
        xlabel('RadioStation1 x')
        ylabel('RadioStation1 y')
    elseif i==4
        %handles is a work around for labelling an overlapping data point
        legend([handles{1},handles{3},handles{4}],...
            'Sum Of Normalized Objective Functions',...
            'Sen Variant And Max Of Normalized Objective Functions',...
            'Cost Function','Location','northwest')
    end

    ax2 = subplot(1,2,2);
    plot(x_mins(i,3),x_mins(i,4),'x','linewidth',2)
    if i==4
        xlabel('RadioStation2 x')
        ylabel('RadioStation2 y')
        legend('Sum Of Normalized Objective Functions',...
            'Max Of Normalized Objective Functions','Sen Variant',...
            'Cost Function','Location','northwest')
    end
end
titles = {'Sum Of Normalized Objective Functions',...
    'Max Of Normalized Objective Functions','Sen Variant','Cost Function'};
figure
%displays the value of the normalized objective functions at each found
%compromise.
for i=1:length(x_mins)
    subplot(2,2,i)
    bar([o1(x_mins(i,:))/min1,o2(x_mins(i,:))/min2,o3(x_mins(i,:))/min3])
    ylim([0.9 1.25])
    title(titles(:,i))
    grid on
    grid minor
    xlabel('Objective Function')
    ylabel('Normalized Value of Objective Function')
```

```
end
```

MATLAB file *convert_to_vector_format.m*:

```
function f = convert_to_vector_format(in_f)

    % basically converts the function handle into a string then changes all
    % of the values into an appropriate vector format.
    tempf = func2str(in_f);

    tempf = regexprep(tempf,'x1,x2,y1,y2','x');
    tempf = regexprep(tempf,'x1,y1,x2,y2','x');
    tempf = regexprep(tempf,'x1','x(1)');
    tempf = regexprep(tempf,'x2','x(3)');
    tempf = regexprep(tempf,'y1','x(2)');
    tempf = regexprep(tempf,'y2','x(4)');
    f = str2func(tempf);
```

MATLAB file *NelderMead.m*:

```
function [x_opt, f_opt, fevals] = NelderMead(x0, function_handle, tolerance)

% Function that minimises 'function_handle' using the Nelder-Mead method.
%
% Input:    x0 - a 5 x 4 vector containing vertices of an initial simplex
%           function_handle - the function to be minimised
%           tolerance - the desired precision
%
% Output:   xopt - an 1 x 4 matrix containing the optimal values of x1, y1,
%           x2 and y2
%           fevals - the value of the function we are minimising at each
%                    iteration, a 1 x n+1 dimensional vector where n is the
%                    number of function iterations


% Define constants - we are using the standard values of each of the
% constants in the Nelder-Mead method
alpha = 1;      % reflection constant
beta = 2;       % expansion constant
gamma = 0.5;    % contraction constant
delta = 0.5;    % shrink constant

% terminates the program if function is not converging after a large number
% of iterations
max_feval = 100000;

% Copy the initial vertices into a new matrix
x = x0;
```

```
% check that supplied x0 has the right dimension - prints error message if
% not
[x_rows, x_cols] = size(x);
if (x_rows ~= x_cols + 1)
    fprintf(['ERROR: incorrect input (dimensions of x0 must be m x ', ...
    '(m+1) where m is the number of variables in the problem)\n']);
    return;
end

% check that supplied x0 is in fact a simplex
if (rank(transpose(x0)) < 4)
    fprintf('ERROR: incorrect input (supplied input not a simplex)\');
    return;
end

% Evaluate function at each vertex of the simplex and order vertices in
% ascending order according to objective function value
f = zeros(1, x_cols + 1);

for i = 1:(x_cols + 1)
    f(i) = feval(function_handle, x(i,:));
end

index = 1:(x_cols + 1);
[f, index] = sort(f);
x = x(index,:);

n_feval = x_cols + 1;

% while difference between function value at all vertices of simplex is
% greater than supplied tolerance, continue procedure (provided n_feval is
% less than max_feval)
converged = 0;
diverged = 0;

while (~converged && ~diverged)

    % Compute the midpoint of the simplex opposite the worst point
    x_bar = sum(x(1:x_cols,:))/x_cols;

    % Compute and evaluate the reflection point
    x_r = x_bar + alpha*(x_bar - x(x_rows,:));
    f_r = feval(function_handle, x_r);
    n_feval = n_feval + 1;

    % Accept the reflected point
    if (f(1) <= f_r && f_r <= f(x_cols))
        x(x_rows,:) = x_r;
        f(x_rows) = f_r;

    % Test for possible expansion
    elseif f_r < f(1)
        x_e = x_bar + beta*(x_r - x_bar);
        f_e = feval(function_handle, x_e);
        n_feval = n_feval + 1;

        % Accept the expanded point (or replace x_n+1 with x_r)
        if f_e < f_r
```

```
            x(x_rows,:) = x_e;
            f(x_rows) = f_e;
        else
            x(x_rows,:) = x_r;
            f(x_rows) = f_r;
        end

    % Compute and evaluate the outside contraction point
    elseif (f(x_cols) <= f_r && f_r < f(x_cols + 1))
        x_oc = x_bar + gamma*(x_r - x_bar);
        f_oc = feval(function_handle, x_oc);
        n_feval = n_feval + 1;

        % Accept the outside contraction point
        if f_oc <= f_r
            x(x_rows,:) = x_oc;
            f(x_rows) = f_oc;
        else
            [x, f] = shrink(x, function_handle, delta);
        end

    % Try an inside contraction point
    else
        x_ic = x_bar - gamma*(x_r - x_bar);
        f_ic = feval(function_handle, x_ic);
        n_feval = n_feval + 1;

        % Accept the inside contraction point
        if f_ic < f(x_rows)
            x(x_rows,:) = x_ic;
            f(x_rows) = f_ic;
        else
            [x, f] = shrink(x, function_handle, delta);
        end
    end

    % Re-sort the points
    [f, index] = sort(f);
    x = x(index,:);

    % Test for convergence
    converged = f(x_rows) - f(1) < tolerance;

    % Test for divergence
    diverged = max_feval < n_feval;
end

if (diverged)
    fprintf('\n ERROR: The maximum number of function evaluations was exceeded \n');
end

% Assign output values
x_opt = x(1,:);
f_opt = feval(function_handle, x_opt);
fevals = n_feval;
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
function [x, f] = shrink(x, function_handle, delta)

% Function that shrinks the simplex towards the best point.
%
% Input:    x - a 5 x 4 vector containing vertices of a simplex
%           function_handle - the function to be minimised
%           delta - the shrink constant
%
% Output:   x - a 5 x 4 vector containings vertices of the simplex after
%               shrinking is applied
%           f - the value of the function we are minimising at each
%               iteration, a 1 x n+1 dimensional vector where n is the
%               number of function iterations

[x_rows, x_cols] = size(x);

x1 = x(1,:);
f(1) = feval(function_handle, x1);

for i = 2:(x_cols + 1)
    x(i,:) = delta*x(i,:) + (1-delta)*x(1,:);
    f(i) = feval(function_handle, x(i,:));
end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

MATLAB file *NewtonMethod.m*:

```
% Matlab function m-file:   NewtonMethod.m
% INPUT:
%   f           - the multivariable function to minimise (a separate
%                       user-defined Matlab function m-file)
%  gradf        - function which returns the gradient vector of f evaluated
%                       at x (also a separate user-defined Matlab function
%                       m-file)
% x0                - the starting iterate
% tolerance     - tolerance for stopping criterion of steepest descent method
%                       and interior algs
% T             - parameter used by the "improved algorithm for
%                       finding an upper bound for the minimum" along
%                       each given descent direction
% OUTPUT:
% xminEstimate  - estimate of the minimum at each iteration
% fminEstimate  - the value of f at each iteration
%k              - the number of iterations
%t              - the time taken to find the minima to the given tolerance.
function [xminEstimate, fminEstimate,k,t] = NewtonMethod(f,x0, tolerance, T)
    %needs an initial declarition for symbolic use
    %value doesn't actually effect anything
    x1=0;
    y1=0;
    x2=0;
    y2=0;
    %symbolically finds the jacobian and hessian
```

```
syms x1 x2 y1 y2;
tempf = jacobian(f,[x1,y1,x2,y2]);
temp2f = hessian(f,[x1,y1,x2,y2]);

%converts back to a function
tempf = matlabFunction(tempf);
temp2f = matlabFunction(temp2f);
%converts function handle back to vector format
gradf = convert_to_vector_format(tempf);
hessf = convert_to_vector_format(temp2f);
f = convert_to_vector_format(f);

%rest is just standard alg
t = tic;
k = 0;  % initialize iteration counter

inner_tolerance = min(tolerance,2^-4);
xk = x0;
store = xk;
fk = feval(f,xk);
while ( norm(feval(gradf, xk)) >= tolerance )
   if (cond(Hessian,2)<10^15)
        if (eig(Hessian) > 0)   % Checks to see if the Hessian is positive definite
             % the Newton direction - as a row vector
            dk = transpose(-Hessian^(-1) * transpose(feval(gradf, xk)));
        else
            dk = -feval(gradf, xk);  % the steepest descent direction.
        end
    else
            dk = -feval(gradf, xk);  % the steepest descent direction.
   end
    % minimise f with respect to t in the direction dk, which involves
    % two steps:
    % (1) find upper and lower bound, [a,b]   ,for the stepsize t
    [a, b] = multiVariableHalfOpen(f, xk, dk, T);
     % (2) use golden section algorithm (suitably modified for
     % functions of more than one variable) to estimate the
     % stepsize t in [a,b] which minimises f in the direction dk
     % starting at xk
     [tmin, fmin] = multiVariableGoldenSectionSearch(f, a, b, inner_tolerance, xk, dk);
     % note: we do not actually need fmin, but we do need tmin

     k = k + 1;
     xk = xk + tmin*dk;
     store = [store;xk];

     %not necessary, but keeps track of the convergence of the function
     %toward the mininum.
     fk = [fk;feval(f,xk)];

     if (norm(store(length(store))-store(length(store)-1))<tolerance)
        break
     end
     if toc(t) >100
          break
      end

end
```

```
    % assign output values
    xminEstimate = store;
    fminEstimate = fk;
    t=toc(t);
```

MATLAB file *objectiveFunctions.m*:

```
%These objectives are not expressed as vectors as it makes it easier to
%find their gradients and Hessians, were required using a symbolic module
%of matlab.
objective1 = @(x1,y1,x2,y2) ((x1-50.5)^2+(y1-207.35)^2)^(1/2)+ ...
    ((x1-146.25)^2+(y1-324.6)^2)^(1/2)+((x1-308.5)^2+(y1-320.15)^2)^(1/2)+ ...
    ((x2-308.5)^2+(y2-320.15)^2)^(1/2)+((x2-406.8)^2+(y2-190.45)^2)^(1/2)+ ...
    ((x2-520.85)^2+(y2-47.2)^2)^(1/2)+((x2-x1)^2+(y2-y1)^2)^(1/2)

objective3 = @(x1,y1,x2,y2) ((x1-50.5)^2+(y1-207.35)^2)+ ...
    ((x1-146.25)^2+(y1-324.6)^2)+((x1-308.5)^2+(y1-320.15)^2)+ ...
    ((x2-146.25)^2+(y2-324.6)^2)+((x2-308.5)^2+(y2-320.15)^2)+ ...
    ((x2-406.8)^2+(y2-190.45)^2)+((x2-520.85)^2+(y2-47.2)^2)

%In order to be able to take the gradient of this objective symbolically
%was necessary to express it in terms of the heaviside function.
objective2 = @(x1,y1,x2,y2) ...
     heaviside(((x1-50.5)^2+(y1-207.35)^2)^(1/2)-((x1-146.25)^2+ ...
     (y1-324.6)^2)^(1/2))*heaviside(((x1-50.5)^2+(y1-207.35)^2)^(1/2) ...
     -((x1-308.5)^2+(y1-320.15)^2)^(1/2))*((x1-50.5)^2+(y1-207.35)^2)^(1/2)...
    +heaviside(((x1-146.25)^2+(y1-324.6)^2)^(1/2)-((x1-50.5)^2+ ...
    (y1-207.35)^2)^(1/2))*heaviside(((x1-146.25)^2+(y1-324.6)^2)^(1/2)...
    -((x1-308.5)^2+(y1-320.15)^2)^(1/2))*((x1-146.25)^2+(y1-324.6)^2)^(1/2)...
    +heaviside(((x1-308.5)^2+(y1-320.15)^2)^(1/2)-((x1-50.5)^2+ ...
    (y1-207.35)^2)^(1/2))*heaviside(((x1-308.5)^2+(y1-320.15)^2)^(1/2) ...
    -((x1-146.25)^2+(y1-324.6)^2)^(1/2))*((x1-308.5)^2+(y1-320.15)^2)^(1/2)...
    +heaviside(((x2-308.5)^2+(y2-320.15)^2)^(1/2)-((x2-406.8)^2+ ...
    (y2-190.45)^2)^(1/2))*heaviside(((x2-308.5)^2+(y2-320.15)^2)^(1/2) ...
    -((x2-520.85)^2+(y2-47.2)^2)^(1/2))*((x2-308.5)^2+(y2-320.15)^2)^(1/2)...
    +heaviside(((x2-406.8)^2+(y2-190.45)^2)^(1/2)-((x2-308.5)^2+ ...
    (y2-320.15)^2)^(1/2))*heaviside(((x2-406.8)^2+(y2-190.45)^2)^(1/2) ...
    -((x2-520.85)^2+(y2-47.2)^2)^(1/2))*((x2-406.8)^2+(y2-190.45)^2)^(1/2)...
    +heaviside(((x2-520.85)^2+(y2-47.2)^2)^(1/2)-((x2-308.5)^2+ ...
    (y2-320.15)^2)^(1/2))*heaviside(((x2-520.85)^2+(y2-47.2)^2)^(1/2) ...
    -((x2-406.8)^2+(y2-190.45)^2)^(1/2))*((x2-520.85)^2+(y2-47.2)^2)^(1/2)

%The found mininums of the objectives from the Methods Used in Part A
min1 = 920.487
min2 = 313.702
min3 = 170127

%Converts the Objectives into Vector Form Using a self built function
o1 = convert_to_vector_format(objective1);
o2 = convert_to_vector_format(objective2);
o3 = convert_to_vector_format(objective3);

%Intermediate values used in calcs
meanOfNorm = @(x) sumOfNorm(x)/3
```

```
giniCoefficient = @(x) (abs(o1(x)-o2(x))+abs(o1(x)-o3(x))+abs(o2(x)-o1(x)) ...
    +abs(o2(x)-o3(x))+abs(o3(x)-o1(x))+abs(o3(x)-o2(x)))/(2*3^2*meanOfNorm(x))

%Penalty Functions
P1 = @(x,k) o1(x) + k/2 * max(0, o2(x) - 1.07*min2)^2 + k/2 * max(0, o3(x) - 1.07*min3)^2
P2 = @(x,k) o2(x) + k/2 * max(0, o1(x) - 1.05*min1)^2 + k/2 * max(0, o3(x) - 1.05*min3)^2
P3 = @(x,k) o3(x) + k/2 * max(0, o1(x) - 1.05*min1)^2 + k/2 * max(0, o2(x) - 1.05*min2)^2

%Welfare Functions
sumOfNorm = @(x) o1(x)/min1+o2(x)/min2+o3(x)/min3
maxOfNorm = @(x) max([o1(x)/min1,o2(x)/min2,o3(x)/min3])
sen = @(x) meanOfNorm(x)*(1-giniCoefficient(x))
sen = @(x) -1/(meanOfNorm(x)*(1-giniCoefficient(x)))

%Cost Function uses our coefficient estimates found after consulting
%various sources in industry
costFunction = @(x) (1000000+100000*10+120000)*o1(x)+10*0.5*261*o2(x)
```

MATLAB file *ParticleSwarm.m*:

```
%Uses the inbuilt matlab function particleswarm to find the global minimum
%of objective. Displays graphics and various print options to show
%progress.
function ParticleSwarm(objective)
    %'MinNeighborsFraction',1 means that each particle wil be affected by
    %all other particles regardless of where they are
    %setting Self and Social Ajustment to 2 meanst that the particles'
    %velocity coefficients will be 2. (ie if it was just governed by the
    %one velocity it would overshoot 50% of the time.
    options = optimoptions('particleswarm','MinNeighborsFraction',1,...
    'SwarmSize',100,'FunctionTolerance',10^-6,'SelfAdjustment',2,...
    'SocialAdjustment',2);
    %enabling display graphics
    options = optimoptions(options,'PlotFcn',{@pswplotbestf,@plotSwarm});
    options = optimoptions(options,'Display','iter');
    %setting the search region to be that in a reasonable area surrounding
    %the towns/radio stations.
    lowerBound = [0,0,0,0];
    upperBound = [600,600,400,400];
    [x,fval,exitflag,output] = particleswarm(objective,4,lowerBound,upperBound,options);
    %output results
    x
    fval
    output
```

MATLAB file *Penalties.m*:

```
close all
```

```
objectiveFunctions
%minimums found using ParticleSwarm function for each objective.
P1_min = [172.4126  279.9606  391.5746  191.6324]
P2_min = [173.2491  278.0488  377.7820  188.2555]
P3_min = [173.9758  276.3853  386.1954  178.3030]

x_mins = [P1_min; P2_min; P3_min]


titles = {'Penalty Function 1','Penalty Function 2','Penalty Function 3'};
figure
%displays the value of the normalized objective functions at each found
%compromise.
for i=1:(length(x_mins) - 1)
    subplot(1,3,i)
    bar([o1(x_mins(i,:))/min1,o2(x_mins(i,:))/min2,o3(x_mins(i,:))/min3])
    ylim([0.9 1.25])
    title(titles(:,i))
    grid on
    grid minor
    xlabel('Objective Function')
    ylabel('Normalized Value of Objective Function')
end
```

MATLAB file *penalty1.m*:

```
% MATLAB script to solve peanlty function 1 using Particle Swarm
% alpha value has initial value 2 and is doubled each iteration
% for loop terminates when consecutive iterations have same function value
% (within a tolerance of 0.001).

% initialise an array to store function values for each iteration
fvals = zeros(60,1);

% iteratively increase alpha, notice that alpha = 2^i
for i = 1:60

    % define the penalty function
    P1 = @(x) o1(x) + 2^i/2 * max(0, o2(x) - 1.07*min2)^2 + ...
        2^i/2 * max(0, o3(x) - 1.07*min3)^2;

    % specify bounds for particle swarm
    lowerBound = [0,0,0,0];
    upperBound = [600,600,400,400];

    % run particle swarm
    [x,val,exitflag,output] = particleswarm(P1,4,lowerBound,upperBound,...
        optimoptions('particleswarm','Display','off'));

    % store function value
    fval(i,1) = val;

    % check whether we need to continue
    if (i >= 2 && (fval(i) - fval(i-1)) < 0)
```

```
        x
        fval(i)
        return;
    end
end
```

MATLAB file *penalty2.m*:

```
% MATLAB script to solve peanlty function 2 using Particle Swarm
% alpha value has initial value 2 and is doubled each iteration
% for loop terminates when consecutive iterations have same function value
% (within a tolerance of 0.001).

% initialise an array to store function values for each iteration
fvals = zeros(60,1);

% iteratively increase alpha, notice that alpha = 2^i
for i = 1:60

    % define the penalty function
    P2 = @(x) o2(x) + 2^i/2 * max(0, o1(x) - 1.05*min1)^2 + ...
        2^i/2 * max(0, o3(x) - 1.05*min3)^2;

    % specify bounds for particle swarm
    lowerBound = [0,0,0,0];
    upperBound = [600,600,400,400];

    % run particle swarm
    [x,val,exitflag,output] = particleswarm(P2,4,lowerBound,upperBound,...
        optimoptions('particleswarm','Display','off'));

    % store function value
    fval(i,1) = val;


    % check whether we need to continue
    if (i >= 2 && (fval(i) - fval(i-1)) < 0.0001)
        x
        fval(i)
        return;
    end
end
```

MATLAB file *penalty3.m*:

```
% MATLAB script to solve peanlty function 3 using Particle Swarm
% alpha value has initial value 2 and is doubled each iteration
% for loop terminates when consecutive iterations have same function value
```

```
% (within a tolerance of 0.001).

% initialise an array to store function values for each iteration
fvals = zeros(60,1);

% iteratively increase alpha, notice that alpha = 2^i
for i = 1:60

    % define the penalty function
    P3 = @(x) o3(x) + 2i^2/2 * max(0, o1(x) - 1.05*min1)^2 + ...
        2^i/2 * max(0, o2(x) - 1.05*min2)^2;

    % specify bounds for particle swarm
    lowerBound = [0,0,0,0];
    upperBound = [600,600,400,400];

    % run particle swarm
    [x,val,exitflag,output] = particleswarm(P3,4,lowerBound,upperBound,...
        optimoptions('particleswarm','Display','off'));

    % store function value
    fval(i,1) = val;

    % check whether we need to continue
    if (i >= 2 && (fval(i) - fval(i-1)) < 0.0001)
        x
        fval(i)
        return;
    end
end
```

MATLAB file *plotSwarm.m*:

```
%A graphic for visualising the R4 position of each particle in the swarm.
function stop =  plotSwarm(optimValues,state)
    stop=false;
    %if main algorithm is iterating perform
    if state=='iter'
        ax1 = subplot(2,2,3);
        plot(optimValues.swarm(:,1),optimValues.swarm(:,2),'.')
        axis([0 600 0 400])
        xlabel('RadioStation1 x')
        ylabel('RadioStation1 y')

        ax2 = subplot(2,2,4);
        plot(optimValues.swarm(:,3),optimValues.swarm(:,4),'.')
        axis([0 600 0 400])
        xlabel('RadioStation2 x')
        ylabel('RadioStation2 y')
        hold off
    end
    %once done.
    if state=='done'
        hold on
```

```
    end
end
```

MATLAB file *steepestDescentMethod.m*:

```
% Matlab function m-file:   steepestDescentMethod.m
% INPUT:
%   f           - the multivariable function to minimise (a separate
%                       user-defined Matlab function m-file)
%  gradf        - function which returns the gradient vector of f evaluated
%                       at x (also a separate user-defined Matlab function
%                       m-file)
% x0                - the starting iterate
% tolerance     - tolerance for stopping criterion of steepest descent method
%                       and interior algs
% T             - parameter used by the "improved algorithm for
%                       finding an upper bound for the minimum" along
%                       each given descent direction
% OUTPUT:
% xminEstimate  - estimate of the minimum at each iteration
% fminEstimate  - the value of f at each iteration
%k              - the number of iterations
%t              - the time taken to find the minima to the given tolerance.

function [xminEstimate, fminEstimate,k,t] = steepestDescentMethod(f, x0, tolerance, T)
    %needs an initial declarition for symbolic use
    x1=0;
    y1=0;
    x2=0;
    y2=0;
    %symbolically finds the jacobian
    syms x1 x2 y1 y2;
    tempf = jacobian(f,[x1,y1,x2,y2]);
    %converts back to a function
    tempf = matlabFunction(tempf);
    %converts function handle back to vector format
    gradf = convert_to_vector_format(tempf);
    f = convert_to_vector_format(f);

    %rest is just standard alg
    t = tic;

    k = 0;  % initialize iteration counter

    inner_tolerance = min(tolerance,2^-4);
    xk = x0;
    store = xk;
    fk = feval(f,xk);

    while ( norm(feval(gradf, xk)) >= tolerance )
        % calculate steepest descent direction vector at current iterate xk
        dk = -1*feval(gradf,xk);  % a row vector

        % minimise f with respect to t in the direction dk, which involves
        % two steps:

        % (1) find upper and lower bound,    [a,b]   ,for the stepsize t using the "improved
        % procedure" presented in the lecture notes
        [a, b] = multiVariableHalfOpen(f, xk, dk, T);
```

```
        % (2) use golden section algorithm (suitably modified for
        % functions of more than one variable) to estimate the
        % stepsize t in [a,b] which minimises f in the direction dk
        % starting at xk
        [tmin, fmin] = multiVariableGoldenSectionSearch(f, a, b, inner_tolerance, xk, dk);

        % update the steepest descent iteration counter and the
        % current iterate
        k = k + 1;

        xk = xk + tmin*dk;
        store = [store;xk];
        %feval(f,xk)
        %not necessary, but keeps track of the convergence of the function toward the mininu
        fk = [fk;feval(f,xk)];

        % this is used to terminate for sub-problem 2 due to gradient
        % never actually going to 0
        if (norm(store(length(store))-store(length(store)-1))<tolerance)
            break
        end

    end
    % assign output values

    xminEstimate = store;
    t = toc(t);

    fminEstimate = fk;
```