Contents

1. Introduction

Mathematics has been studied around the globe for thousands of years. Early mathematics was developed mainly by the Greeks. In 1526, the first mathematical symbols were used; these being + and - [1] and were soon followed by multiplication, division and many other symbols used. One of the main problems these mathematicians faced was due to the lack of computational power available as an abacus is only as fast as its user. This gave great problems when trying to evaluate large problems such as mathematical optimisation. To overcome this, many methods of finding the optimal solution to a problem have been researched which require less computation. Early optimisation would have been carried out on one-dimensional space, for example, find the optimal solution of x^2+5x+1. This type of problem can be solved in many ways but once the variable space is increased into multiple dimensions then these techniques are often inadequate or require too much computation. Engineers are faced with problems such as these frequently and methods to find an ideal solution are still sought after. Such problems may be as simple as evaluating profit of a design process, but can extend into complex evaluation of calculation, for example, as an electronics engineer, it is extremely important to produce hardware which is fast, reliable and available to the public as soon as possible. Mathematical optimisation may be used to find the optimal method to implement an array of logic within a microprocessor. No matter what the problem is, a tool to find the general solution is required.
This is a report of the foundations of mathematical optimisation and how a method was applied in writing a program to evaluate the optimal solution of an unconstrained two-problem. In the following section, the problem is studied to produce a 'formal specification', then in the next section, several methods of evaluating the solution to such problems are discussed and one is chosen for use in the program. The mathematical programming language MATLAB is discussed and reasons explaining the choice of it for the problem are detailed. The design process of the program will be illustrated along with problems encountered and how these were solved. A conclusion of the report follows which will also recommend future work to be studied in this area. Finally a User Manual is included for details on how to use the program written to obtain a solution for a problem in 2D or less.

2. Formal Specification

The task given was vague and before any further design could take place, it was necessary to produce a formal specification to follow. This allowed the design to be valid and verified at each stage of the process. The formal specification is shown below:
"The requirement is to produce a piece of software that will accept a continuous, unconstrained two-dimensional function and suitable ranges. The software should then calculate the optimal solution to the problem within these ranges. The optimal solution will be given by the minimum value within the ranges. One algorithm will be used to obtain a solution and it will be based on the Simplex Algorithm. To ensure ease of use, a graphical user interface will be developed to accept the function to be

optimised and suitable range limits. As a method of proving the solution, the software will also provide a graphical output of the function if the user requests. As an additional feature, an exhaustive search algorithm should be included to demonstrate the efficiency and accuracy of the Simplex Algorithm used."
This specification differs from the task given in that a graphical user interface has been included to ensure the program is easy to use.

3. Optimisation Methods

As mentioned previously, several methods for optimisation have been developed. In the example above, the one-dimensional solution can be obtained using a variety of algorithms including:

Exhaustive Search
Golden Section Method
Differentiation (Gradient Method)
Simplex

Many others are used but generally not for such a simple problem. It must be noted that if an optimisation problem is to find maxima rather than minima, then the negative of the function can be taken and the minima calculated: these minima will then correspond to the location of the maxima in the original function.
The Exhaustive Search method is the slowest method available from the selection. However, it always obtains the optimal solution within a given range. The method is as follows:

1. Start at lower range and evaluate value to problem
2. Increase variable to be solved by arbitrary amount, delta and solve for it
3. Repeat previous step until the whole range is covered
4. The optimal solution is the min/max found in step 2

As can be seen, this is a very simple method of obtaining the solution but requires a lot of computation. It must also be noted that if the step delta is too big, the exact solution will not be obtained, ideally delta is infinitesimally small but this would be impossible to evaluate.
The Golden Section Method offers an algorithm which will find the solution in far fewer steps, therefore less time. The Golden Section relates to Fibonacci's studies. The Fibonacci sequence is a very basic sequence where the current value is obtained by summing the previous two values as shown:

$$x_n = x_{n-1} + x_{n-2}$$

$$x_0 = 1, x_1 = 1$$

This results in the following sequence of numbers

1,1,2,3,5,8,13,21,34,55,89…

The golden ratio is defined as the ratio of:

$$\frac{x_n}{x_{n-1}}$$

This value is approximately 1.618033989 but the exact value can be found by

$$F_r = \frac{\sqrt{5}+1}{2}$$

This ratio can be found frequently in nature and in many mathematical problems. The population of rabbits from generation to generation approximates to the Fibonacci sequence as does the spiral on a snails shell (If an axis is drawn from the centre outward, the distance from the centre is approximately the Fibonacci sequence). The Fibonacci ratio (also known as the golden section) has many interesting properties such as:

$$F_r^{\ 2} = F_r + 1 \quad \text{and} \quad \frac{1}{F_r} = F_r - 1$$

The Fibonacci ratio is used in the Golden Section Method algorithm. The algorithm is as follows:

1. Evaluate the values of the problem at the range limits
2. Split the range using $1/F_r$ and $1\text{-}1/F_r$ and evaluate the values at these points
3. Select the three lowest values and split the limits using $1/F_r$
4. Repeat the previous step until the limits are sufficiently close
5. The optimal solution is now found

This sounds like an ideal algorithm as it converges rapidly and produces the solution, but the problem lies in the solution obtained. This method can miss the optimal solution or get caught in ‘local’ minima as shown below in figure 1. It can be seen that the three lowest points do not enclose the true minimum solution, X, and the algorithm will provide a solution for the local minimum, o. The true minimum was ‘skipped’ by the algorithm.

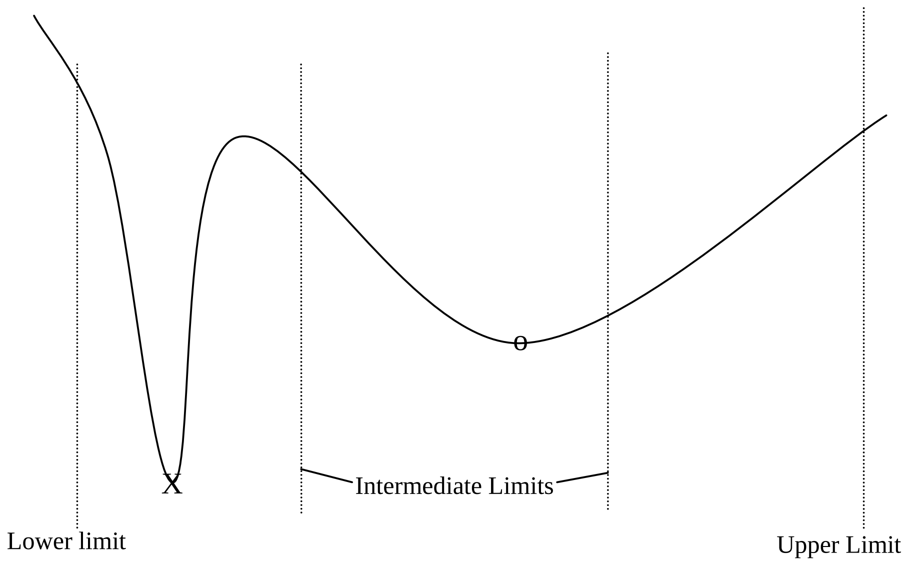


Figure 1: The Golden Section Method ‘skips’ past the true

An improved method of evaluating the minimum solution can be obtained using basic differential calculus. In this basic one-dimensional problem, the solution can be obtained by evaluating where:

$$\frac{df}{dx} = 0$$

This may result in several points. At each point, the gradient is zero, therefore a local minimum/maximum may be obtained. Each point can then be entered into the original function to evaluate which has the minimum solution. For example, take the function:

$$f(x) = x^4 - x^3 - x^2 + x - 5$$

$$\frac{df(x)}{dx} = 4x^3 - 3x^2 - 2x + 1$$

$$\frac{df(x)}{dx} = 0 \text{ when } 4x^3 - 3x^2 - 2x + 1 = 0$$

$$\therefore x \approx 0.64 \text{ or } x = 1$$

$$f(0.64) \approx -5.62 \text{ or } f(1) = -5$$

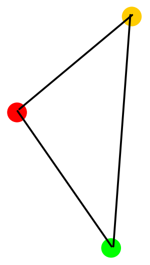
minimum at 0.64 with value of -5.62

This simple process let us obtain the solution quite easily, as long as the roots of the differentiated function are easily obtained. In the example above, there were 2 minima, and the one that resulted in the lowest solution was chosen. This is a very quick method of obtaining the solution, however in multi-dimensional space it becomes very complex to obtain the derivative and once obtained, finding where this equals zero may be impossible. A method of finding a solution rapidly and easily is required as is the extension into multiple dimensions. An algorithm which satisfies these requirements is the Simplex Algorithm.

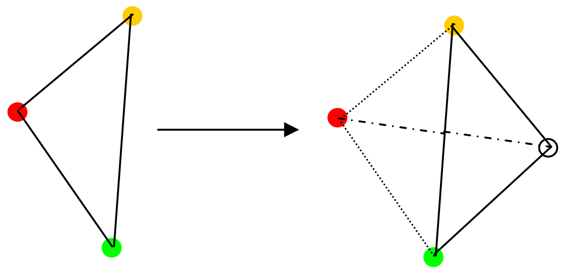
3.1 The Simplex Algorithm

The algorithm used for the program was the downhill simplex or Nelder-Mead Method. This method follows a small set of basic rules which can be easily described. A simplex is a basic shape with one more vertex than dimensions in the problem, for example, the two-dimensional optimisation program written required the use of a simplex with three vertices. It is easy for the brain to think about one, two and three-dimensional space but beyond causes a lot of confusion. Below the simplex algorithm is detailed as applicable to two dimensions but it can be easily extended into further dimensions.

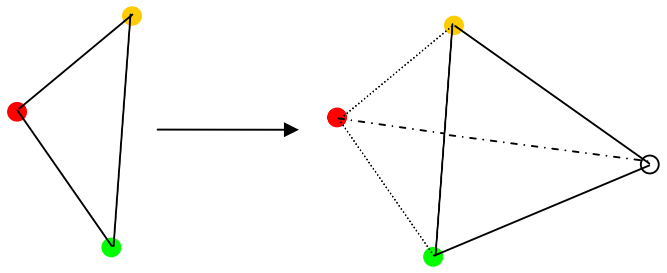
1. A starting point is chosen and a simplex created with each vertex close to the starting point.
2. The value of each vertex is obtained (the sketches that follow will have a red vertex for the highest value, a green for the lowest and amber for the intermediate vertex) as shown:



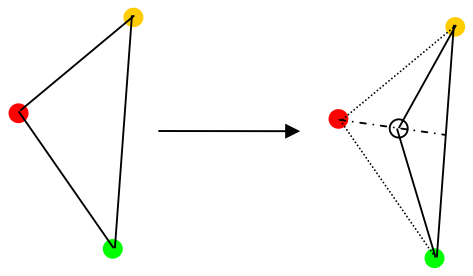
3. To try and obtain a better solution, the highest value vertex is then reflected around the midpoint of the line between the two lowest value vertices and the new value at this point is calculated



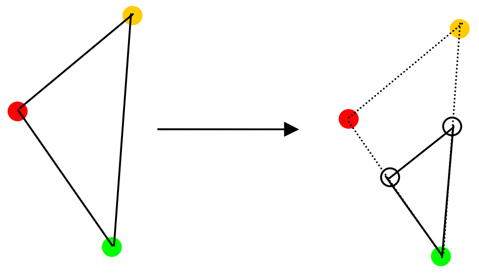
4. If the previous step resulted in a lower value then it is repeated, however the point is not just reflected but also extended to obtain a minimum result more rapidly. The value of this new point is then calculated



5. The previous step is repeated until a better point is not obtained. When this occurs, the highest point is no longer reflected but contracted toward the midpoint as shown. The new point is then calculated



6. This last step is repeated until either unsuccessful, or if the high point gets too close to the midpoint. Once this happens, all points are scaled toward the low point and the new points calculated



7. This whole process is repeated until the three vertices become sufficiently close.

This algorithm works well in obtaining a solution. Due to its expanding simplex, it does not get tricked easily into local minima unless they are quite large. The algorithm also works very quickly as no complex maths are required, just scaling, and it uses much less calculations than the exhaustive search.
The disadvantage of such an algorithm is it can get 'stuck' in local minima and some algorithms can result in a solution in much fewer steps. These problems are overcome in the ease of application of the simplex. To successfully implement the algorithm a sufficient programming language was required.

4. MATLAB

It was decided for this program to be written in MATLAB. MATLAB provides powerful manipulation and ease of use for working with mathematical problems. It was originally intended for rapid matrix manipulation, for example finding the inverse, or eigenvalues. MATLAB has been enhanced to include many different fields of mathematics and since has become a very powerful tool for engineers worldwide. MATLAB also allows relatively simple implementation of a graphical user interface, or GUI, which results in programs written being easy to use.
The syntax of MATLAB is similar to that of the ANSI c programming language but does not require specific libraries to be included nor such a rigorous use of end-of-command semi-colons. The language will not be discussed in detail here but many texts are available for understanding and using MATLAB.

For this application, knowledge of several areas was required including how to declare a function, how to find the value of the function and how to plot in three dimensions. Although the problems are only two-dimensional a third dimension makes the function easier to visualise. The MATLAB Student Edition user's guide was consulted for many of the problems met but some were not covered by the text. Along with MATLAB comes extensive documentation and available online is the helpdesk. These were used to aid some difficulties. Finally, if the problem could not be solved, the MATLAB newsgroup (comp.soft-sys.matlab) was used and response was generally rapid.

To input a function to MATLAB as a string, the input command can be used with the *'s'* switch to make the entered text stored as a string, for example:

```
ftbm=input('Please input function to be minimsed','s');
```

After this command, the string could be converted to a format that MATLAB understands using the *inline* command with syntax as follows:

```
f=inline(ftbm);
```

This creates an inline function for use within the program. To evaluate this function, the *feva*l command can be used but an easier alternative was found:

```
value=f(5,7);
```

Assuming the function given was that of two variables. With these problems solved, a program could begin to take shape.

As with any program, the good use of comments are required whilst in development. This allows for easy debugging and also allows a third party to understand and follow the flow of the program.

5. The Design

The first step in the design process was to use the previously mentioned knowledge of inputting functions to MATLAB and produce a program that would find the roots of a one-dimensional problem. This basic program used the bisection method. Some PDL for this is shown below where a and b were limits specified by the user:

```
Check for opposite signs of f(a) and f(b)
while not converged
    c=(a+b)/2
    if |f(c)| < tolerance then converged
    else
        if f(c) > 0 then b=c
            else a=c
    evaluate f(a) and f(b)
end while
```

This program checks to make sure that there is an obvious root between the specified limits before going ahead. It then takes a new value exactly in-between the limits and checks to see if it is close enough to the root. If it is not then the two points that enclose the root are kept and the program loops until the root is found.

This program could then be modified to converge quicker but it was decided to make a start on the larger problem.

To allow an easy visualisation of the function to be minimised, it was decided to first produce a small program that would also benefit testing to see if the declared minimum was indeed correct.
The program was as follows:

```
% This next section is merely plotting out function to check for minimum, will be removed for final copy
x=linspace(xlower,xupper,25); % Make a matrix between x limits with 50 points (all columns equal)
y=linspace(ylower,yupper,25); % Make a matrix between y limits with 50 points (all rows equal)
[X,Y]=meshgrid(x,y); % Make a grid with the points in x and y
Z=f(X+eps,Y+eps); % Evaluate the function at each of these points
surf(X,Y,Z); % Draw a surface graph
shading interp % Use interpolation to shade graph
% End of plotting
```

The first line is a comment to explain what the section is about. The second line creates 50 points between the lower and upper *x* limits specified by user. The same is done for *y* in the next line. The line that follows creates a matrix with all values of *x* (storing them in columns), and *y* (storing them in rows)and stores them in *X* and *Y* respectively. Line five then evaluates the function at each point and stores these results in Z. The graph is drawn in the next line and finally the graph is shaded using interpolation. The program ends with another comment to show that this is the end of the said section.
This little program displays an approximate graph of the function to be minimised. Although it is simple, it is using a similar method to exhaustive search to obtain all values of the function. This is only used for a visual aid, hence the step size, and is not used by the simplex algorithm in any way.
The design was laid out as shown below in figure 2 to ensure every stage ran smoothly.

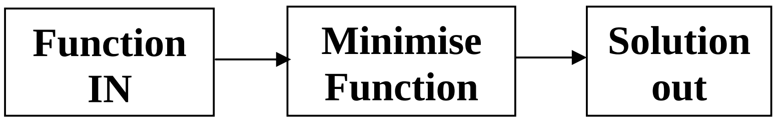


Figure 2: The Simple Overall Design

This is a very simple block diagram but every design needs to start somewhere. A good software design process will split the design into several stages, keeping each quite simple. In the above diagram, the first block had already been completed. A small piece of code had been produced to get a function from the user and store it in a form usable by MATLAB. The ‘Minimise Function’ block is where most of the design is based, instead of being a black-box it was expanded as shown below in figure 3:

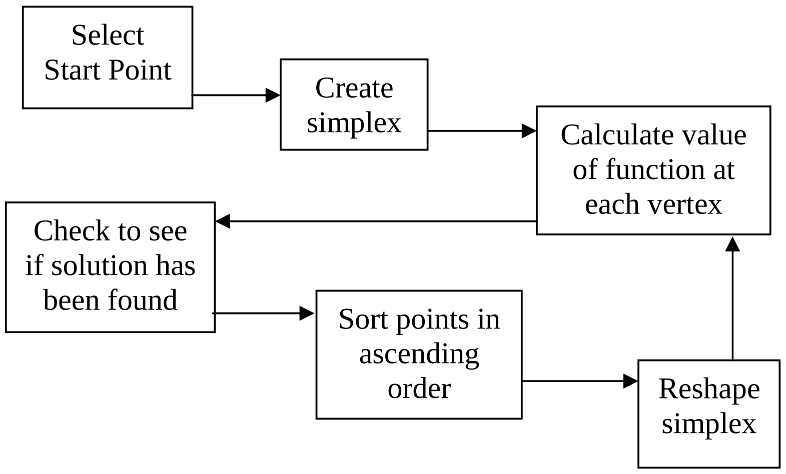

Select Start Point
Create simplex
Calculate value of function at each vertex
Check to see if solution has been found
Sort points in ascending order
Reshape simplex

Figure 3: Black Box of Design Process

The sort was included for the algorithm to work; *point1* was always given the worst value and *point3* was always given the best value. With this in mind, the algorithm could then operate without knowledge of each value as the problem had become generalised. Each box within this process was then to be coded and then verified. A typical software engineering problem uses validation and verification at each stage of a design. This allows for the formal specification to be adhered to.

<u>5.1 The first draft</u>

With the design specified, and a suitable block diagram of how the design was to flow, coding begun for each individual step.

5.1.1 Start Point

The start point was chosen to be at the middle of the ranges, so code was as follows:

```
point1=[(xupper+xlower)/2 (yupper+ylower)/2]; % First point is initialised in middle of limits
        of x and y given
```

xlower, *xupper*, *ylower* and *yupper* were the range values specified by the user. This was thought to be an ample starting point and easily calculated.

5.1.2 Create Simplex

To create the simplex, the start point was used and values for point2 and point3 were obtained relative to the range given by the user. This would allow the simplex to be appropriately sized for a general problem, for example if a user specified ranges from -50 to 50, then the simplex would be given a ‘length’ of 1, or if a user specified -1 to 1 for the range then the simplex would be given a ‘length’ of 0.02. In other words, the ‘length’ of the simplex was taken as 1% of the range specified. Code for this was as shown below:

```
% Create a simplex with 3 points to minimise 2-D unconstrained problem
point1=[(xupper+xlower)/2 (yupper+ylower)/2]; % First point is initialised in middle of limits
        of x and y given
point2=[point1(1)+(xupper-xlower)/100 point1(2)]; % Second point is initialised in middle of
        limits of x and y given + delta(x) evaluated by splitting difference of xlower and
        xupper by 100
```

```
point3=[point1(1) point1(2)+(yupper-ylower)/100]; % Third point is initialised in middle of
            limits of x and y given + delta(y) evaluated by splitting difference of ylower and
            yupper by 100
% End creation of simplex
```

This shows the code for selecting start point as well as the creation of points based on this start point.

5.1.3 Calculation of Each Point

The next stage was easy, all that had to be done was to evaluate the value of the function at each vertex. This was achieved using the following piece of code:

```
% Evaluate simplex values at points
value1=f(point1(1),point1(2)); % Evaluate function at point1
value2=f(point2(1),point2(2)); % Evaluate function at point2
value3=f(point3(1),point3(2)); % Evaluate function at point3
% End evaluate simplex values
```

5.1.4 Check for Solution

This stage was a little more complex. How does one know if a solution has been reached? The gradient may be found, numerically and if the function was 'flat' enough then the solution would be met. This would work but had a few problems. If a function is generally flat, then the optimal solution may not be obtained, for example consider the function

$$f(x,y) = \frac{x+y}{1\times 10^{6}}$$

This function would have a solution at the same place if it were not divided by one million but if the gradient method was used to check for 'flatness' then it may cease prematurely and provide the wrong solution. Another disadvantage is that the simplex may still be relatively large and an accurate solution would not be reached. Instead, to check for a solution it was decided to check the size of the simplex. To do this, the following code fragment was produced:

```
% Are we at minimum?
temp1=abs(point1-point2)+abs(point1-point3)+abs(point2-point3); % temp variable to see
            how close each point is
temp2=abs(temp1(1))+abs(temp1(2)); % another temp variable to ensure close in x-direction
            and y-direction
if temp2<((xupper-xlower)/1e4+(yupper-ylower)/1e4)/2 %If we are at minimum
   disp('Minimum found'); ;  % then display message
   break % Quit from loop
end % otherwise, do nothing
% End minimum check
```

This checked to see if the simplex was a certain size no matter what the range was. This is due to the accuracy of solutions, all variables used were only accurate to 4 decimal places and as such, if the simplex was smaller, then a solution was obtained.

5.1.5 Sort Points

As mentioned earlier, the points were sorted to make coding easier. Reshaping the simplex did not need to check the value of each point if this was done first. Due to there only being three points involved, a basic bubble sort was used with code as shown below:

```
% Order points by value (value1 highest down to value3) - using bubble sort
if value1 < value2
   temp=point1; point1=point2; point2=temp;
   temp=value1; value1=value2; value2=temp;
elseif value2 < value3
   temp=point2; point2=point3; point3=temp;
   temp=value2; value2=value3; value3=temp;
elseif value1 < value2
   temp=point1; point1=point2; point2=temp;
   temp=value1; value1=value2; value2=temp;
end
% End bubble sort
```

A bubble sort works by checking to see if adjacent values are ordered, if they are then it continues, if not it swaps them around before continuing. This is carried out until they are ordered, in this case only three times was necessary.

5.1.6 Reshape Simplex

This block contains the 'brains' to the program. At this stage, all of the work is done in obtaining a better solution, given a set of points. As mentioned earlier, the simplex algorithm works by reflecting or resizing, or both. An enhanced algorithm was developed to hopefully yield results more rapidly. The changes were:

- If the highest point is to be reflected, then it will also be extended no matter if the previous step was successful in obtaining a better solution
- When resizing the simplex, all points were shrunk toward the midpoint of all three points. This resulted in quicker convergence

With these modifications in mind the following code was used:

```
% Reshape simplex
midpoint=(point2+point3)/2; % Find the midpoint between point2 and point3
intermediate=midpoint-point1; newpoint1=point1+2.1*intermediate; % Reflect point1 to
        become closer to minimum and extend by 10%
newvalue1=f(newpoint1(1),newpoint1(2)); % Evaluate value of function at this new point
if newvalue1 < value1 % If reflection gives better solution
   value1=newvalue1; % Then use the new point and value
   point1=newpoint1;
else
   % Otherwise reduce size of Simplex
   % Try a method of finding centre of simplex and then scaling all points toword it.
   centre=(point1+point2+point3)/3; % Evaluate co-ords of centre point
   intermediate1=centre-point1; intermediate2=centre-point2; intermediate3=centre-point3; %
        Find out the vectors from pointx to centre and name it intermediatex
   point1=point1+(1-lambda)*intermediate1; % scale all points toward centre
   point1=point2+(1-lambda)*intermediate2; % Using scaling factor lambda
   point1=point3+(1-lambda)*intermediate3; % This is initialised at the start of the program
end
% End reshape simplex
```

The use of lambda here provided quick variation of scaling factor for experimentation. High values for lambda would result in a slower convergence and low values of lambda would result in more rapid convergence.

6. Problems

This design above was repeated until a minimum was reached, thee range limits breached or until 500 iterations had completed. Each block was tested to ensure that it worked as required. A problem was encountered with the bubble sort algorithm. It was not always sorting the points as expected. This was due to the *elseif* commands used. Each should be replaced by an *end* then *if* command resulting in code:

```
% Order points by value (value1 highest down to value3) - using bubble sort
if value1 < value2
   temp=point1; point1=point2; point2=temp;
   temp=value1; value1=value2; value2=temp;
end
if value2 < value3
   temp=point2; point2=point3; point3=temp;
   temp=value2; value2=value3; value3=temp;
end
if value1 < value2
   temp=point1; point1=point2; point2=temp;
   temp=value1; value1=value2; value2=temp;
end
% End bubble sort
```

This solved the problem so that the bubble sort now worked as intended. The design had another basic flaw in that, the simplex always started at the centre. If there was a local minimum at the centre, then it would fall straight into it without the simplex increasing the size. The starting point was then given a random offset to allow the algorithm more chance of reaching the real solution. This was done by adding the following code:

```
randomstart(1)=rand(1)*(xupper-xlower)/7-rand(1)*(xupper-xlower)/5;%Evaluating a
           randomstart selection
randomstart(2)=rand(1)*(yupper-ylower)/7-rand(1)*(yupper-ylower)/5;
```

randomstart was then added to *point1*. Since *point2* and *point3* were located relative to *point1* it was not necessary to add *randomstart* to them as well.

Another problem with the code was that simplex could easily extend past the limits. The design was then repeated until the simplex went 10% past each limit or until the other conditions were met. This was done by checking if simplex was within the extreme limits: *xlower*1.1, xupper*1.1, ylower*1.1* and *yupper*1.1*.

If the range was set so that one was high and the other very low then this failed, for example if *xupper* was set to 100 and *xlower* to -1, then the simplex could extend 10 units past the upper limit but only 0.1 past the lower limit. With the 1% length of the simplex being just over 1 unit; if the simplex approached the lower limit it may go past it. The extreme limits were then 10% outwith the range given, for example the lower limit in the x-direction was set as *xlower-(xupper-xlower)*0.1*. This solved the problem.

One major problem of the original design was when the optimal solution lay at a one of the boundary limits specified by the user. If this was the case then the algorithm would result in an inaccurate solution. This was particularly noticeable in the function

$$f(x,y) = x + y$$

The optimal solution for this functions lies at the corner of *xlower* and *ylower*. The algorithm would reach one limit and stop resulting in wrong results, for example running the program a few times gave solution at:

(-1,-0.8976)   (-0.5647,-1)   (-1,-0.9778) and (-1, 0.7399)

with the lower limits both set to -1. It can be seen that on no occasion in this small sample size did the algorithm give the true result, infact the algorithm was often out by more than 20% which was unacceptable. To avoid this, a further check was introduced into the ‘Reshape Simplex’ design. If one of the vertices of the simplex went beyond the user-specified limits, then its value would become very large set by value1=1/(0+eps). The value of *eps* is defined as, “The smallest number such that, when added to one, creates a number greater than one on the computer”[2]. It was also found that if point1 was not scaled then the algorithm worked even if the solution lay on a boundary. This provided two minor problems

1. The solution was less accurate for functions with minima within limits
2. The simplex did not converge nearly as rapidly

To overcome these, a check was made to only stop scaling point1 if the simplex was at the boundary. This results in an algorithm which calculates minima accurately at any place in the function. If the minimum is on the boundary then the algorithm takes slightly longer to converge but is still much quicker than an exhaustive search.
A slight modification was made to the algorithm to increase speed. The check to see how small the simplex was done using the intermediate1 vector calculated when resizing the simplex. This size-check would then only need to be done if the simplex was reduced.

7. Additional Features

This resulted in a complete working solution to the optimisation problem but the design had to continue to include a GUI and an exhaustive search.

7.1 Exhaustive Search

This was done using the following code:

```
minimum=1/(0+eps); % set the minimum so far to almost infinity
x=linspace (xlower,xupper,250); % set the number of points in x to search
y=linspace (ylower,yupper,250); % and in y
for i = 1:length(x) % for loop, check every value of x
  for j = 1:length(y) % for every value inx, check every value in y
    xcoord=x(i); % set x co-ord
    ycoord=y(j); % set y co-ord
    value=f(xcoord,ycoord); % Evaluate function at this point
    if value<minimum % If this is the lowest solution so far
      xcoordmin=xcoord; % then set min co-ords
      ycoordmin=ycoord;
      minimum=value; % and the value at this point
    end
  end
end
```

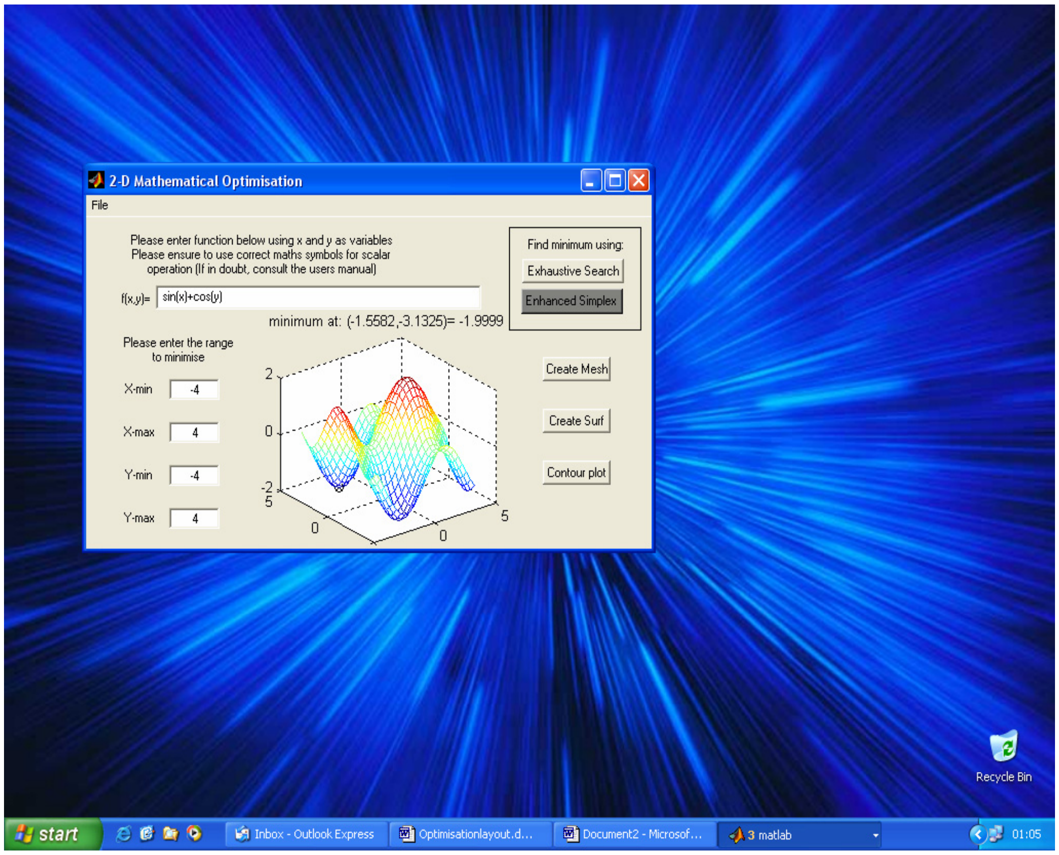
This very simple code took a very long time to execute as it evaluated the function at every point, 62500 of them in the case above. This method did not give an accurate solution either, for example for the function:

$$f(x) = \sin(x) + \cos(y)$$

the exhaustive search would report minimum solution of -1.9999 but the simplex would report a minimum of -2. Other functions made the exhaustive search even worse. The enhanced simplex used in the main design yielded a much more accurate solution and in much less time. Typically, the exhaustive search took over 30 seconds and the enhanced simplex took less than a quarter of a second. These values were recorded using the commands *tic* and *toc,* which display the elapsed time between the two commands.

7.2 Graphical User Interface

To enhance this program and make it very easy to use a graphical user interface was included. This interface was decided to have an area to enter the function and range limits, a set of axes for displaying the approximate graphical output and buttons to choose which method of plotting, and two main buttons for optimising with both algorithms. MATLAB contains a suite of tools for designing the interface, which were easy to use after reading through the example GUI. The picture below in figure 4 shows the final GUI in action.


2-D Mathematical Optimisation

Please enter function below using x and y as variables
Please ensure to use correct maths symbols for scalar operation (If in doubt, consult the users manual)

f(x,y)= sin(x)+cos(y)

minimum at: (-1.5582,-3.1325)= -1.9999

Please enter the range to minimise

X-min -4
X-max 4
Y-min -4
Y-max 4

Figure 4: Graphical User Interface

Details are not given on the design of the GUI as this step was not required, but some basic knowledge is briefly discussed. To create the GUI, the ‘guide’ command was used. Guide is an acronym for Graphical User Interface Development. This command gave a layout designer and push buttons, axes and editable text was laid out as shown in figure 4. Each of these was given a tag and an m-file was produced. In the m-file, each tag had a separate function and details of this function were entered in the appropriate section. It was noted that variables must be declared as global for every function requiring them unless the ‘handles structure’ was used, in which case, the structure had to be saved after every modification using the *guidata* command. With these requirements in mind the GUI was programmed easily. The graphical output display was also modified to detail the minimum found with a cross and circle.

8. Test Functions

During the design, the algorithm was tested to ensure operation was as expected. Some functions have been detailed in the design process above and how they were calculated correctly or wrongly. Below, figure 5 shows the results from the final program to show its effectiveness of displaying the correct results.

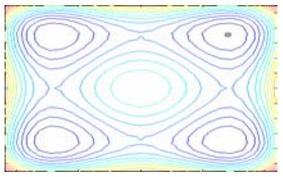
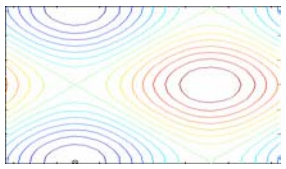
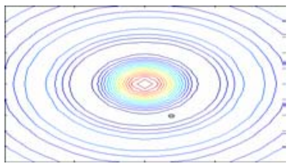
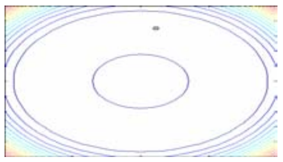
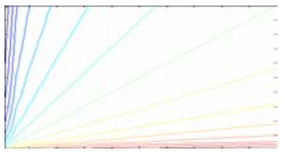
Function to be minimised	**Range set** (xmin, xmax, ymin, ymax)	**Minimum at:**	**Contour plot**
$f(x,y)=(x^4-x^2)+(y^4-y^2)$	(-1.1, 1.1, -1.1, 1.1)	(0.70711,0.70711,-5)	
$f(x,y)=\sin(x)+\cos(y)$	(-3.2, 3.2, -3.2, 3.2)	(-1.5708, -3.1416, -2)	
$f(x,y)=\frac{\sin(R)}{R} \quad R=\sqrt{x^2+y^2}$	(-10, 10, -10, 10)	(1.8964, -4.0736, -0.21723)	
$f(x,y)=R^4-R^2$ $R=\sqrt{x^2+y^2}$	(-1, 1, -1, 1)	(0.11011, 0.69848, -0.25)	
$f(x,y)=\log x-\log y$	(0.1, 10, 0.1, 10)	(0.1, 10, -4.0652)	

Figure 5: Example Results from Final Program

Many other functions have been tested and worked, this is just a few. The algorithm also works with imaginary numbers, so this does not restrict the user to 'real' functions. Overall, the algorithm provides the correct solution for a very broad range of functions and ranges. As mentioned earlier, it also produces these results very quickly indeed, and as such would be useful as a tool for many two-dimensional optimisation problems.

9. Conclusion

At the start of the design, it was thought that it was going to be a difficult process. This was the first piece of major software written by the author. After much thought, it was decided to implement the simplex algorithm. The design became much easier after it had been broken down into several stages. Figures 2 and 3 show this process. Once each stage had been recognised the design of each stage went fluently and problems could easily be recognised. The algorithm had some flaws in that it was not producing the correct result if the optimal solution was at the boundary of limits specified. This was overcome by adapting the algorithm for boundary conditions. The speed of the algorithm was also increased by scaling at all stages and when scaling down, all points were scaled toward a centre point unless at a boundary. Overall, this produced a 20-25% increase in speed.

There is room for improvement in this program. Some speed increase could be obtained by evaluating some variables only once, as several are being calculated in different stages of the design. This may increase the overall speed of the algorithm but it is thought that it will not be by much. The program set out to find the optimal solution to a two-dimensional function. This could easily be extended into three-dimensions and beyond.

References

[1] Jeff Miller, http://members.aol.com/jeff570/operation.html, Dec 2001
[2] "The Student Edition of MATLAB Version 5 User's Guide", D. Hanselman and B. Littlefield, Prentice Hall 1997

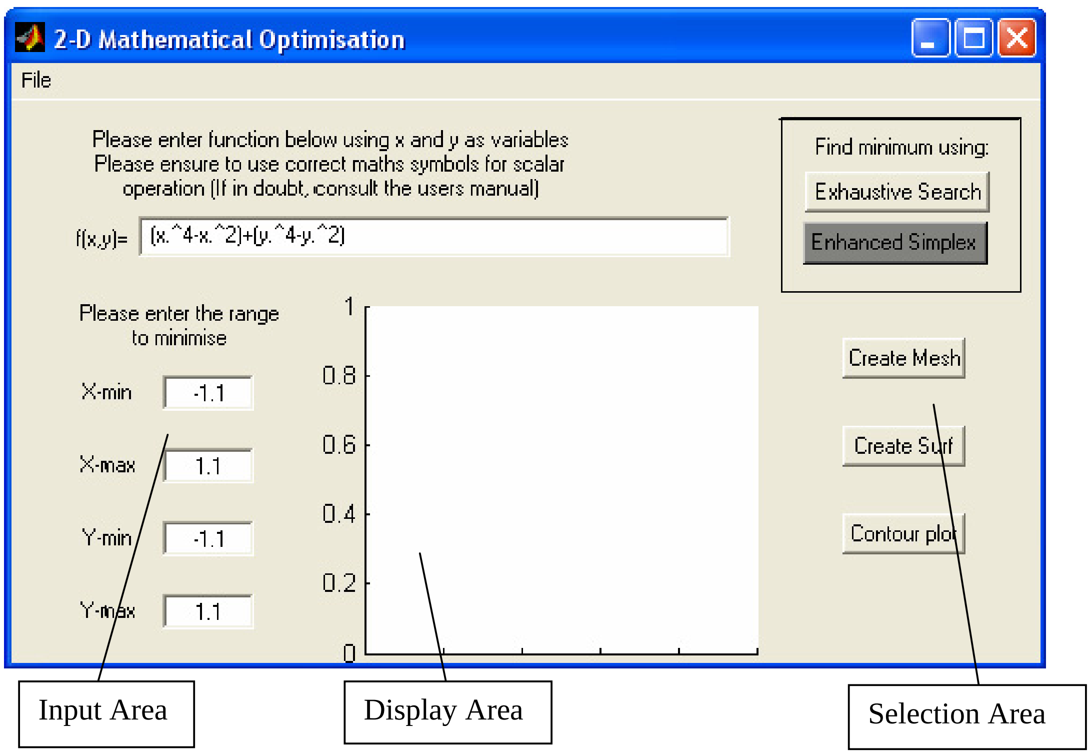
Appendix: User's Manual

This program has been designed to find the minimum solution to a continuous, unconstrained, two-dimensional optimisation problem. It takes input from the user for the function and the range to search over. The user can then chose to find the solution to their problem using the 'Enhanced Simplex' algorithm or an exhaustive search method. The user may also specify an approximate graphical output of their function and can print this out. If the user wishes to find the maximum solution, then the entered function should be the negative of the original. This will result in the optimal solution being located.

There are four files to this program; all should be stored in the current MATLAB path, for example c:\MATLAB\work. This will ensure the program runs correctly. The four files are:

- opt2dui.fig
- opt2dui.m
- simplex2.m
- exhaustive.m

To run the program simply type 'opt2dui' at the MATLAB prompt, this will display the graphical user interface as shown below:



Please enter function below using x and y as variables
Please ensure to use correct maths symbols for scalar operation (If in doubt, consult the users manual)

f(x,y)= (x.^4-x.^2)+(y.^4-y.^2)

Please enter the range to minimise

X-min -1.1
X-max 1.1
Y-min -1.1
Y-max 1.1

The interface is split into 3 main areas:

The input area on the left
The selection area on the right
The display area in the centre.

The input area has 5 boxes for input about the function to be minimised. At the top. The function can be entered at the box starting with ‘f(x,y)=’. To enter the function simply click in the white area (there is a default function ‘(x.^4-x.^2)+(y.^4-y.^2)’ already there. Use <backspace> or <delete> to clear the function box and input your function. Functions must follow a certain format as discussed below. This is the format specified by MATLAB, so if you already know how to do this, you may skip this step.

Inputting a Function

All scalar multiplication/division (multiplying or dividing by a fixed number, e.g. 2) must be entered with * or / for multiplication and division respectively.

e.g. 4*x or y/3

All division and subtraction should be entered using + or – whether it is scalar or variable addition/subtraction

e.g. x+5 or y-2+3

If variables are to be multiplied or divided, then use the operators .* or ./

e.g. x.*y or y./x

If exponents are to be used for variables then the .^ operator should be used, if scalar exponents are required then just use ^

e.g. x.^3 or 5^2

Trigonometric functions need to be entered as normal but multiplication or division of these follow the rules of variables as shown

e.g. sin(x).*cos(y)

If trigonometric functions are to be scaled then it may be done as follows

e.g. 3*cos(y) or (sin(x.*y))/4

To evaluate the square root of a function, this should be entered

e.g. sqrt(x) or sqrt(x.^2+y.^2)

The default function at startup should help clarify some of these matters.

Setting the Range

Again, a default range has been set here. The range values have been labelled as xmin for the minimum value of x to search, xmax for the maximum value of x to search and similarly for y. To change the range, simply click on the corresponding limit you want to change. Using the <backspace> or <delete> keys, clear the existing range and enter a numeric value for your range. Please note that this must be numeric and an error message is displayed if it is not. If pi is required, then it is suggested to use 3.1416.

Finding the minimum

The selection area on the right hand side has been split into main areas. The bottom contains methods of graphically displaying your approximate function and the top contains two buttons to find the solution to your problem.
It is recommended just to use the ‘Enhanced Simplex’ button shaded in dark grey. This will give a very accurate result very quickly. Once you have entered your function and set your range, simply click on this button and the minimum will be displayed at the top of the display area. The minimum displayed gives the location of the minimum and the value at the minimum.

Graphical Output

This feature provides an approximate visual display of the function entered and the minimum is displayed by a ⊗. To view a graphical output, simply click on either of the buttons marked 'Create Mesh', 'Create Surf', or 'Create Contour' once a valid function has been entered. The Mesh display shows a 3-D graph with several points and each adjacent point is connected by a straight line. The Surf display shows a 3-D interpolated graph. This makes the graph appear smooth and is a better approximation to the function. If the Contour is selected, this provides a 2-D top down view of the graph, with the value of the function shown by coloured contours. Red contours show high values for the function, whereas dark blue contours show low values of the function.

Printing Display

The graph may be printed by clicking on the File Menu at the top right and then on display. This requires a printer to be specified in MATLAB which is not covered in this manual. Please consult the MATLAB User's Guide.