PH20105 Experimental Physics and Computing 2

Hand-in exercise 2022-2023 Semester 2 C Coursework

A report on PH20105 C coursework containing the methods and procedures undertaken to solve the problems

---

University of Bath

Candidate Number: 24705

March 18, 2023

PH20105: C Computing Coursework

Candidate Number: 24705

Department of Physics,
University of Bath, Bath
BA2 7AY, UK

Date Submitted: March 18, 2023

Contents

Introduction

This file contains the findings and a report for the C coursework of March 2023, Semester 2 for module PH20105. The student candidate number for submission is 24705. Included in the appendices are all pieces of code used throughout.

The report focuses on analysing the Rosenbrock function to find the minimum of the parabolic valley along with discussions about approaches and methods taken to improve cycles and results. This is an important aspect of physics and is commonly used in research as well as modeling techniques. Implementing the methodology into automated software can make processes more reliable as well as a lot quicker than those of standard methods. This report focuses on three main areas which include finding the minimum analytically, Rosenbrock's parabolic valley numerically, and the downhill simplex method. The main aspect, the downhill simplex method, is written in C while the programs used to draw graphs were written in python. The code for all aspects, if not included within the section, is included in the appendix at the end of this report.

The Rosenbrock parabolic valley is a function of two variables and is commonly used for testing optimisation algorithms. The global minimum of the function is determined through different methodologies outlined in this report along with the findings from these. The sections below refer to the relevant parts within the assigned document.

Finding the minimum analytically

The Rosenbrock function is given below in Eq.1 and is used throughout the report and for any calculations of function value. The function contains a single minimum which is situated inside a narrow, parabolic-shaped flat valley. The function can be assumed to be at a minimum when evaluated and has a value equal to 0 at the coordinates used. The values of (x_0, x_1) can be found analytically knowing this. The analytical derivation is shown below.

The Rosenbrock function used throughout is given by,

$$y = F(x_0, x_1) = 100(x_1 - x_0^2)^2 + (1 - x_0)^2. \tag{1}$$

The function contains two variables and to allow for calculation of the minimum partial differentiation must be used with respect to both x_0 and x_1 separately. The two versions in the partial differential form are shown below,

$$\frac{\partial F}{\partial x_0} = 400x_0(x_0^2 - x_1) + 2(x_0 - 1), \tag{2}$$

$$\frac{\partial F}{\partial x_1} = 200(x_1 - x_0^2). \tag{3}$$

An alternative way of displaying this is using grad which is often used for vector fields and can be used in the case of a varying field. This is shown below and ultimately demonstrates the same information however is a nice way to view the equations.

$$\nabla F(x_0, x_1) = \begin{pmatrix} 400x_0(x_0^2 - x_1) + 2(x_0 - 1) \\ 200(x_1 - x_0^2) \end{pmatrix}$$

The next appropriate step was to set both of these equations equal to zero and rearrange them. The minimum of a function is widely known to exist where the function itself has a value equal to 0. The most useful form for this was to find an equation so that x_1 was in terms of x_0 and then substitute this into the equation. This is shown below and was derived from Eq.3,

$$x_1 = x_0^2. \tag{4}$$

Then sub Eq.4 into Eq.2 to arrive at the equation below,

$$2(x_0 - 1) + 400x_0(x_0 - x_0) = 0.$$

This can then be rearranged and it is apparent that the bracket containing $(x_0 - x_0)$ cancels to zero, leaving the equation below,

$$2(x_0 - 1) = 0.$$

Then follows simple rearrangement, and it is known for this to equal zero the bracket must be equal to zero. This then allowed for the values of x_0 to be calculated as below,

$$x_0 - 1 = 0$$

$$x_0 = 1.$$

This result could then be used in conjunction with Eq.4 to calculate the value of x_1. This showed that the value of the minimum point is situated at $(1, 1)$ with the value of the function $F(x_0, x_1)$ being equal to 0 at this point. A test program was then written in order to confirm the values of coordinates and what corresponding function value they outputted. This confirmed that the coordinates calculated did in fact yield a minimum value of the function. The code is shown in Appendix 1, however can be done alternatively by hand.

The final solution calculated analytically is therefore,

$$(x_0, x_1) = (1, 1)$$

$$F(x_0, x_1) = 0.$$

Rosenbrock's parabolic valley numerically

Visualisation of functions that contain multiple variables can be extremely beneficial in understanding the general shape of the curve, allowing for a basic understanding of where the minimum could be situated. Initially, a 3D plot in python was conducted which allowed for the general shape of the function to be displayed. This is shown in Fig,1 and is particularly useful for observing the parabolic valley in which the minimum is located.

Additionally, a contour plot was conducted to demonstrate the gradient of the Rosenbrock function for a narrower range of values. A contour plot is a useful way to visualise a function and can be used to better demonstrate the location of the minimum. The plot shows how the value of the function varies over a range, with the contour lines representing regions of constant function values. The contour plot is widely used for visualising optimisation problems and allows the position of the minimum to be seen more clearly. The contour plot for the Rosenbrock function between the range $-2 < x_0 < 2$ is shown in Fig.2.

The Rosenbrock function was implemented into C code and was used to calculate a series of values based on the boundary conditions assigned. The outcome of this was to graphically display a slice of the function where one of the parameters is kept constant, in this case, x_1. The conditions required are stated as,

- 100 values of the function $F(x_0, x_1)$.
- The range of x_0 is: $-2 < x_0 < 2$. We want 100 values for the functions between this range
- x_1 to be fixed and set equal to 1. This ensures a constant slice through the function.

The values generated from this were saved to a text file and could be extracted using python to plot the values of the function. The final outcome, an intersection plot of the Rosenbrock function, displays the parabolic valley containing the global minimum of interest. The plot is shown in Fig.3 where y, the function value, was plotted as a function of x_0. The code written in python is shown in Appendix 2 along with the writing to file aspect of Appendix 3.

The plot shown in Fig.3 is extremely useful for demonstrating where the minimum of the function could occur and enables a rough estimate for the position of the global minimum. The knowledge and code used in this can then be taken into the next part where the minimum was found using an implementation of the downhill simplex method written in C. It is important to highlight that the figure produced identifies the position of two minima within the range provided and when looking at the function it becomes apparent why this is true.

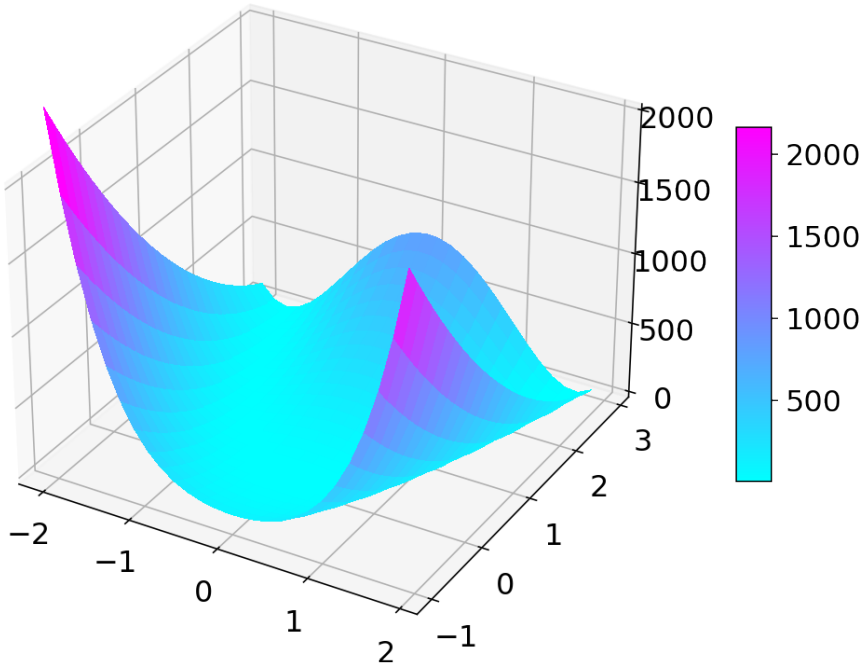

Figure 1: A 3D color map plot of the Rosenbrock function produced in python. This was done by initialising x and y values and then creating a mesh grid of these values. These were then plotted and an appropriate color scale was used to show the gradient of the function at each value. The plot was beneficial in visualising the position of the minimum point to be calculated later, as well as confirming whether the main plot for this part is of the correct shape. In addition, it is also interesting to observe the gradient of the function over a large range of values.

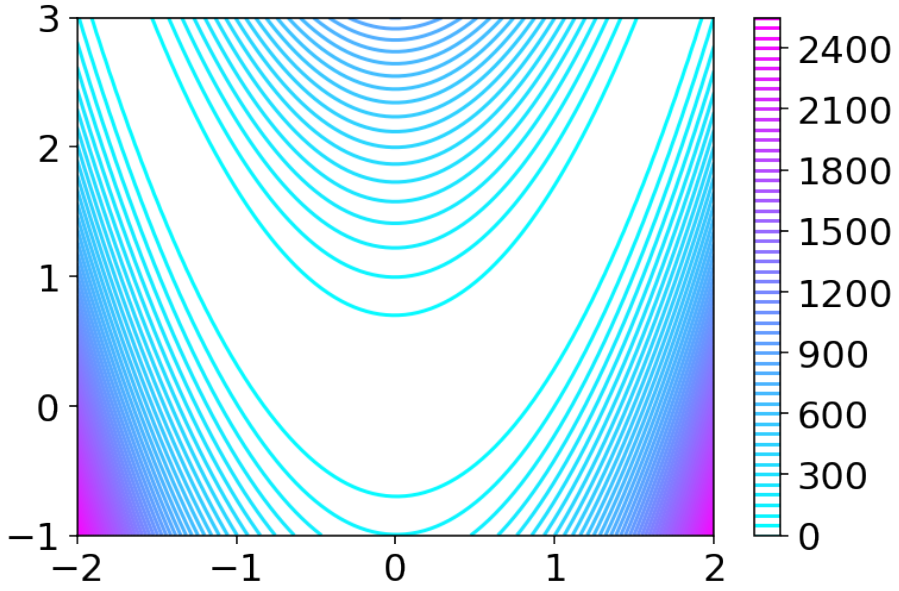

Figure 2: A colored contour plot produced in python of the Rosenbrock function between the values specified in the assignment. The plot is useful in identifying the region of particular interest which contains the minimum point. This can be used to improve efficiency by initialising the simplex in part 3 within a region of greater accuracy. The color scale was chosen to match that of Fig.1 to allow for easier comparison. The scale demonstrates the value of the function at a particular value, which can be useful for checking the value of the function is correct for initial simplex coordinates.

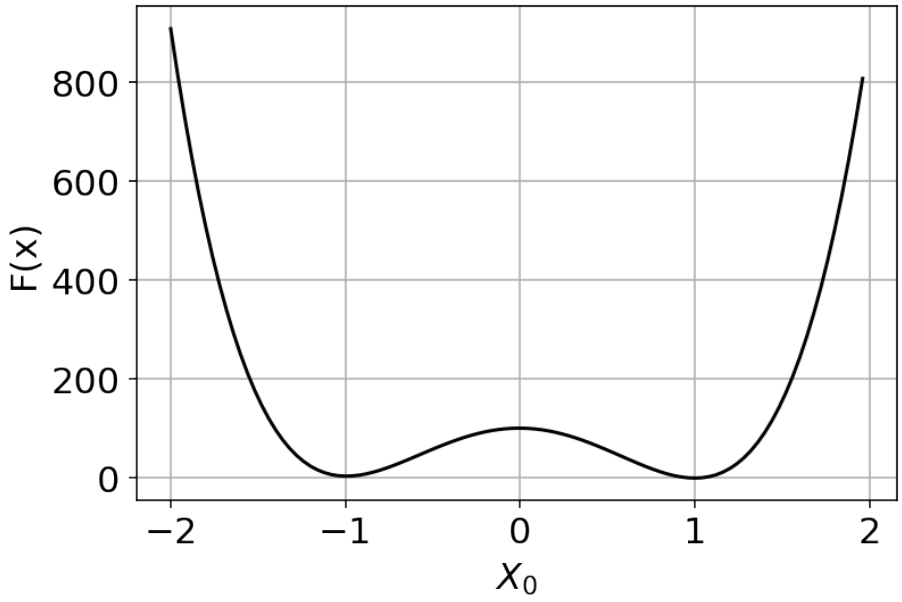

Figure 3: The Rosenbrock function sliced where x_1 was fixed and set equal to 1. The plot was produced in python and allows for visualisation of the positions of the minimum point to be calculated. This can be compared to the other plots and allows for confirmation that it holds the correct shape as expected. The plot successfully displays the parabolic valley and allows for a rough confirmation that the values calculated analytically are correct.

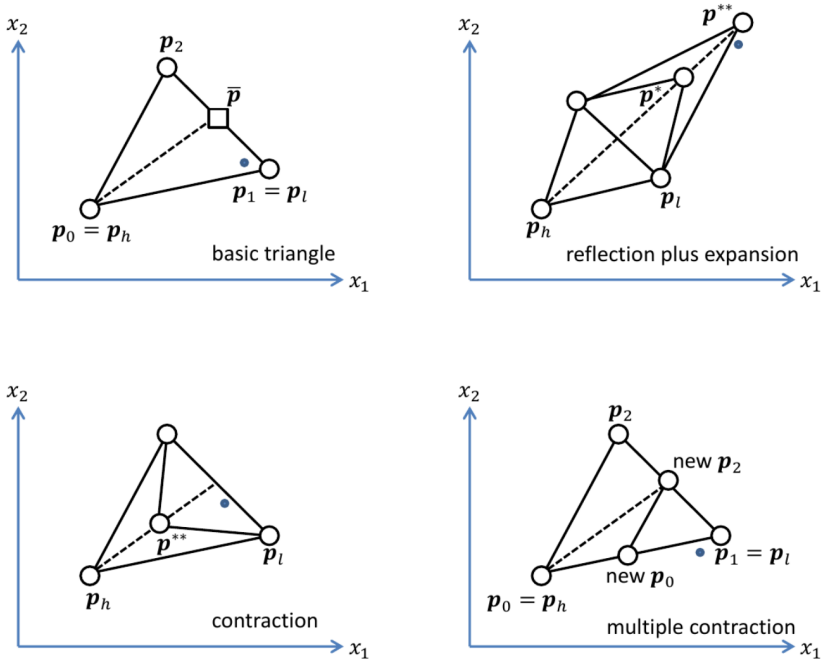

Figure 4: Illustrative diagram of the possible transformation moves on the simplex. These moves were implemented into C, having functions for each using their formulas stated in the assigned document. This was particularly useful for visualisation of how the simplex transforms over the first two iterations which were conducted by hand initially.

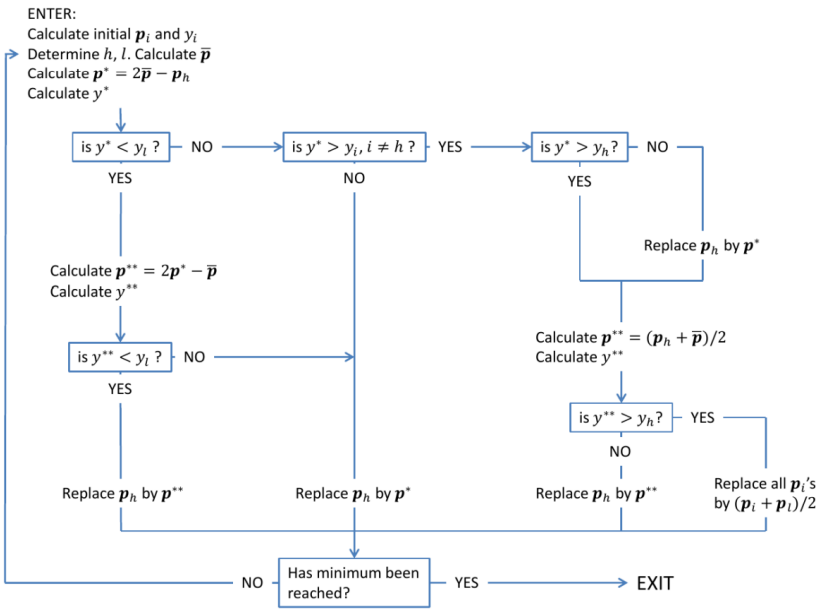

Figure 5: The flow diagram is used to decide the order of moves when conducting the downhill simplex method. This was implemented into C using a variety of techniques and statements and allowed for the calculation of the minimum via computational methods.

Downhill simplex

The downhill simplex method is a technique widely used to find the minimum point of functions and works by implementing an initial simplex which can be transformed accordingly to where the vertices lie with respect to one another. The three vertices are checked continuously to see which one lies closest to the minimum and the other vertices are moved by applying one or more moves from Fig.4. The order of moves is decided according to the flow chart shown in Fig.5. This shows the comparisons that must be made and then the following transformations to follow according to the outcome. The allowed transformations, as shown in Fig.4, are reflection, contraction, and expansion. These can occur at any time according to the flow chart and multiple transformations can be used for one iteration. The formulae for the movements are stated below.

Reflection
The simplex is reflected by exchanging $\mathbf{p}_h$ with $\mathbf{p}^*$ as shown below,

$$\mathbf{p}^* = 2\bar{\mathbf{p}} - \mathbf{p}_h. \tag{5}$$

Expansion
The triangle can be expanded, by shifting $\mathbf{p}^*$ further outwards, shown below as,

$$\mathbf{p}^{**} = 2\mathbf{p}^* - \bar{\mathbf{p}}. \tag{6}$$

Contraction
The triangle can finally be contracted shown as,

$$\mathbf{p}^{**} = \frac{\mathbf{p}_h + \bar{\mathbf{p}}}{2}. \tag{7}$$

The initial task was to run through the flowchart and complete the first two iterations by hand which allowed for a better understanding of how the algorithm operates on the function.

The first two iterations of the downhill simplex method on the Rosenbrock function are provided below. It is beneficial to compare the moves to Fig.4 to enable a visualisation and understanding of how the simplex is transforming.

First iteration:

Starting coordinates $= (0,0), (0,2), (2,0)$.

Evaluate the function at these coordinates:

$$y_{(0,0)} = 1,$$

$$y_{(0,2)} = 401,$$

$$y_{(2,0)} = 1601.$$

The initial function values are therefore $= (1, 401, 1601)$

Sort these values into highest(h), middle(m) and lowest(l) points:

$$h = 1601$$

$$m = 401$$

$$l = 1$$

Calculate the centroid

$$\bar{p} = (0,1)$$

Calculate trial vertices and function value for this

$$p* = (0,2) - (2,0) = (-2,2)$$

$$y* = 409$$

Begin flow chart comparisons.
Is the trial vertex lower than the lowest value

$$y* < y_l?$$

No, so right along the flow chart.
Is the trial vertex bigger than the middle function value?

$$y* > y_{mid}?$$

Yes. Move right along the flow chart.
Is the trial vertex bigger than the highest function value?

$$y* > y_h?$$

No. Replace the highest point by the trial vertex.

$$p_h \rightarrow p*$$

Contract

$$p** = \frac{(2,0) + (0,1)}{2} = (1, \frac{1}{2})$$

Reevaluate the function value at this point.

$$y** = 25$$

Compare the value of the new point to the highest point.
Is the value of the function larger than the original highest?

$$y** > y_h?$$

No. Replace the highest point with the new vertex.

$$p_h \rightarrow p**$$

Has the minimum been reached?
No. Has the max number of iterations been reached?

$$(iterationcounter > 1000).$$

No. Has the criteria been met? The criteria is below,

$$\sqrt{\sum_i \frac{(y_i - \bar{y})^2}{n}} < 10^{-8},$$

where n =2.
No. Go back to the beginning with the current points.

Iteration 1 complete.

Second Iteration:

Updated coordinates after first iteration = $(0,0), (1, \frac{1}{2}), (0,2)$.
Evaluate function value for these coordinates.

$$y_{(0,0)} = 1$$

$$y_{(1,\frac{1}{2})} = 25$$

$$y_{(0,2)} = 401$$

Calculate the centroid

$$\bar{p} = (\frac{1}{2}, \frac{1}{4})$$

Calculate the trial vertices and function value for this

$$p* = (1, \frac{1}{2}) - (0,2) = (1, -\frac{3}{2})$$

$$y* = 625$$

Begin comparison.
Is the trial function value smaller than the lowest?

$$y* < y_l?$$

No, move right along the flow chart.
Is the trial vertex bigger than the middle function value?

$$y* > y_{mid}?$$

Yes. Move right along the flow chart.
Is the function value larger than the highest function value?

$$y* > y_h?$$

Yes.
Calculate the new trial vertex points via contraction

$$p** = \frac{(0,2) + (\frac{1}{2}, \frac{1}{4})}{2} = (\frac{1}{4}, \frac{9}{8})$$

$$y** = 113.45$$

Is the new trial larger than the highest function value?

$$y** > y_h?$$

No. Replace the highest function value by the trial function value.

$$p_h \rightarrow p**$$

Has the minimum been reached?
No. Has the max number of iterations been reached?

$$(iterationcounter > 1000).$$

No. Has the criteria been met? The criteria is below,

$$\sqrt{\sum_i \frac{(y_i - \bar{y})^2}{n}} < 10^{-8},$$

where n =2.
No. Go back to the beginning with the current points.
Set the new coordinate points.

$$\text{Updated coordinates} = (0,0), (1, \tfrac{1}{2}), (\tfrac{1}{4}, \tfrac{9}{8})$$

Iteration 2 complete.

The first two iterations by hand could then be used to check against the code to ensure that it was working in the correct way. The code was written in a structure such that it allowed for ease when changing variables as well as allowing for efficient debugging.

The Rosenbrock equation as well as the possible transformations were written into functions so that if used multiple times it was easy to call upon instead of repeating multiple lines of code numerous times. This was particularly useful as both parts two and three required the rosenbrock function to calculate values. The flow chart, which ensures the correct operations are undertaken, was also implemented into a function. The use of multiple functions to break down the 'main' function allows for the code to be better understood and followed through.

The downhill-simplex function calls upon other functions to incorporate the flow chart and clearly shows which operation is being undertaken. The algorithm used is a derivative-free method that can be used to find the minimum of functions of a similar form. The function takes an array of points, and the initial coordinates for the simplex, and repeatedly evaluate the points based on the criteria of the downhill simplex method until the maximum iterations are reached or the tolerance criteria are met. Once these are met or the minimum is found the code will stop, whichever occurs first.

Throughout the program, pointers are used and they allow for a memory address to be passed instead of values. The benefits of pointers are widespread and examples include improved performance of the code by reducing memory use as well as allowing functions to modify data outside of their purview. There are a variety of pointers used in the code and in the functions the general use of them is to modify variables outside of the function. Examples of this include inside the 'ordering()' function which takes an array,'x1', and three points,' highest,second-highest, lowest', as an argument. The function then updates the variables pointed to by these pointers based on the values in the array 'x1'. This allows for values to be returned more clearly and efficiently. The functions concerning transformations also make use of pointers. The functions update the values, for example, 'reflected-x', based on the input values and the Rosenbrock function. The functions can modify the values directly by using pointers instead of returning multiple values. Pointers are extremely beneficial however it was crucial when writing the code they were used correctly as they can be prone to errors and memory leaks. This was an issue faced while producing the working code and it took multiple attempts in order to correctly implement the pointers.

The program produced the results as expected and showed the global minimum is found at the same coordinates as derived analytically. To reiterate these occur at $(x_0, x_1) = (1, 1)$ with the function, $F(x_0, x_1)$ being equal to 0 for these values. The output of the program is added below with the full code included within the relevant appendix as stated below.

```
//Output of the program
F(x0,x1) = 0.000000
Final Coordinates =(1.000055,1.000125,1.000108,1.000217,1.000108,0.999851)
Minimum Point found at (x_0,x_1) = (1.000055, 1.000125)
Number of iterations = 59
```

Appendix

5.1 Rosenbrock Function Value Calculator

This piece of code is a simple piece used to calculate the value of the Rosenbrock function for any pair of coordinates that can be changed easily within the code. This is an alternative method to using a calculator and is easily adaptable to calculate the value in the final code.

```c
#include <stdio.h>
#include <math.h>

double rosenbrock(double x, double x1) {
    return pow(1 - x, 2) + 100 * pow(x1 - pow(x, 2), 2);
}

int main() {
    double x = 1;
    double x1 = 1;
    double fx = rosenbrock(x, x1);
    //printf("Initial point: x_0 = %f, x_1 = %f, F(x) = %f\n", x, x1, fx);
    printf("Initial point: x_0 = %f\n", x);
    printf("Initial point: x_1 = %f\n", x1);
    printf("Value of the function is: F(x,x1) = %f",fx);
    return 0;
}
```

```
//Output is as follows;
Initial point: x_0 = 1.000000
Initial point: x_1 = 1.000000
Value of the function is: F(x,x1) = 0.000000
```

5.2 Using python to plot from a text file

This section of code written in python is concerned with calculating values of the Rosenbrock function between boundary values and then saving them to a file. This is used to plot the function slice in section 2.

```
import pandas as pd
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
%config InlineBackend.figure_format = 'retina'
%matplotlib inline
plt.rcParams.update({'font.size':16})

data = np.loadtxt("values.txt", delimiter = ", ")
x = np.array(data[:,0])
F = np.array(data[:,1])
plt.plot(x, F,color='Black')
plt.draw()
plt.ylabel('Y')
plt.xlabel('X')
plt.show()
```

5.3 Code for plotting contour plots in Fig.1 and Fig.2.

```
## Code for 3D contour plot
b = 100;
f = lambda x,y: (1-x)**2 + b*(y-x**2)**2;
# Initialize figure
figRos = plt.figure(figsize=(12, 7))
axRos = figRos.gca(projection='3d')

# Evaluate function
X = np.arange(-2, 2, 0.15)
Y = np.arange(-1, 3, 0.15)
```

```
X, Y = np.meshgrid(X, Y)
Z = f(X,Y)

# Plot the surface
surf = axRos.plot_surface(X, Y, Z, cmap='cool',linewidth=0, antialiased=False)
axRos.set_zlim(0, 2000)
figRos.colorbar(surf, shrink=0.5, aspect=10)
ax.set_xlabel('X')
ax.set_ylabel('Y')
plt.show()
plt.savefig('3D.jpg')

##Code for contour line plot
import numpy as np
import matplotlib.pyplot as plt

def rosenbrock(x, y):
    return (1 - x)**2 + 100 * (y - x**2)**2

# Create a grid of points to plot the function
x = np.linspace(-2, 2, 100)
y = np.linspace(-1, 3, 100)
X, Y = np.meshgrid(x, y)

# Evaluate the function at each point in the grid
Z = rosenbrock(X, Y)

# Plot the function as a contour plot
plt.contour(X, Y, Z, levels=50,cmap='cool')
plt.colorbar()
#plt.title('Rosenbrock Function')
#plt.xlabel('x')
#plt.ylabel('y')
plt.show()
```

5.4 Full C code for implementation of the downhill simplex method on the Rosenbrock function and writing to a text file

This contains the full source code for writing to a file as well as finding the minimum of the Rosenbrock function.

```
#include <stdio.h>
#include <math.h>
//Define the Rosenbrock function as stated:
double rosenbrock(double x0, double x1) {
    return pow(1 - x0, 2) + 100 * pow(x1 - pow(x0, 2), 2);
}
//Define the function used to write to files
void write_to_file(double x1) {
    FILE *datafile;
    datafile = fopen("values.txt", "w");
    //Stating the limits. The divisions of 0.04 ensuring 100 values outputted into
    file.
    for (double x0 = -2; x0 <= 2; x0 += 0.04) {
        double result = rosenbrock(x0, x1);
        fprintf(datafile, "%.4lf, %lf\n", x0, result);//Format of calcualted values.
    }
    fclose(datafile);
    printf("\nValues written to file:\n\n");
}

//Function for finding the order of indicies based on their values
//Comparison checks are done to find the order of the vertex points from highest to
    lowest of all 3 points
void ordering(double x1[3], int *highest, int *second_highest, int *lowest) {
    *highest = 0;
```

```
    *second_highest = 0;
    *lowest = 0;
    for (int i = 0; i < 3; i++) {
        if (x1[i] > x1[*highest]) {
            *second_highest = *highest;
            *highest = i;
        }
        else if (x1[i] > x1[*second_highest]) {
            *second_highest = i;
        }
        if (x1[i] < x1[*lowest]) {
            *lowest = i;
        }
    }
}
//Calculating the centroid from the initial points.
//Follows the method outlined
void calculate_centroid(double points[3][2], int highest_index, double *centroid_x,
    double *centroid_y) {
    double x_sum = 0.0, y_sum = 0.0;
    for (int i = 0; i < 3; i++) {
        if (i != highest_index) {
            x_sum += points[i][0];
            y_sum += points[i][1];
        }
    }
    *centroid_x = x_sum / 2;
    *centroid_y = y_sum / 2;
}
//Function used for reflecting when criteria for reflection met
void reflection(double points[3][2], int highest_index, double centroid_x,
    double centroid_y,
             double *reflected_x, double *reflected_y, double *func_evaluate) {
    *reflected_x = centroid_x + (centroid_x - points[highest_index][0]);
    *reflected_y = centroid_y + (centroid_y - points[highest_index][1]);
    *func_evaluate = rosenbrock(*reflected_x, *reflected_y);
}
//Function used for expanding when criteria met
void expand(double points[3][2], int highest_index, double centroid_x, double
    centroid_y,
            double reflected_x, double reflected_y, double func_evaluate, double *
    expanded_x,
            double *expanded_y, double *func_alternate, double *x1) {
    *expanded_x = centroid_x + 2.0 * (reflected_x - centroid_x);
    *expanded_y = centroid_y + 2.0 * (reflected_y - centroid_y);
    *func_alternate = rosenbrock(*expanded_x, *expanded_y);
    if (*func_alternate < func_evaluate) {
        points[highest_index][0] = *expanded_x;
        points[highest_index][1] = *expanded_y;
        *x1 = *func_alternate;
    } else {
        points[highest_index][0] = reflected_x;
        points[highest_index][1] = reflected_y;
        *x1 = func_evaluate;
    }
}
//Function used for contracting when criteria met
void contract(double points[3][2], int highest_index, double centroid_x, double
    centroid_y,
              double *contracted_x, double *contracted_y, double *func_alternate,
    double *x1) {
    *contracted_x = centroid_x + 0.5 * (points[highest_index][0] - centroid_x);
    *contracted_y = centroid_y + 0.5 * (points[highest_index][1] - centroid_y);
    *func_alternate = rosenbrock(*contracted_x, *contracted_y);
    if (*func_alternate < *x1) {
        points[highest_index][0] = *contracted_x;
        points[highest_index][1] = *contracted_y;
        *x1 = *func_alternate;
    }
  }
//The function where criteria is tested and compared
//All the operations are also undertaken in this
void simplex_method(double points[3][2]){
    double x1[3];
    //These values are used in the movements instead of using fractions, 0.5 is used
    in multiplacation
```

```
    int i, j, l, highest_index, s, m, num_iterations = 0;
    double sum, centroid_x, centroid_y, reflected_x, reflected_y, func_evaluate,
        contracted_x, expanded_y, func_alternate,expanded_x,contracted_y;
    double tolerance = 1e-8; // set the tolerance value here
    // Evaluate the Rosenbrock function at the initial points
    for (i = 0; i < 3; i++) {
        x1[i] = rosenbrock(points[i][0], points[i][1]);
    }
    //Ensuring it runs until max iterations is reached
    while (num_iterations < 1000) {
        // Find the order of indices.
        ordering(x1, &highest_index, &s, &l);
        //Check to see if tolerance criteria has been reached
        if (sqrt((pow((x1[highest_index] - x1[l]),2)/2)) < tolerance) {
            break;
        }
        //Otherwise continue running through iterations
        //Calculate the centroid of the non-highest point
        calculate_centroid(points, highest_index, &centroid_x, &centroid_y);
        // Reflect the highest point through the centroid
        reflection(points, highest_index, centroid_x, centroid_y, &reflected_x, &
    reflected_y, &func_evaluate);
        if (func_evaluate < x1[l]) {
            // Expand the point
            expand(points, highest_index, centroid_x, centroid_y, reflected_x,
    reflected_y, func_evaluate, &expanded_x, &expanded_y, &func_alternate, &x1[
    highest_index]);
        } else if (func_evaluate >= x1[s]) {
        // If the reflected point is worse than the second highest point, contract
    the simplex
            if (func_evaluate < x1[highest_index]) {
            // If the reflected point is still better than the highest point, accept
    it
                points[highest_index][0] = reflected_x;
                points[highest_index][1] = reflected_y;
                x1[highest_index] = func_evaluate;
            }
            // Calculate the point to which the highest point is to be contracted
            contract(points, highest_index, centroid_x, centroid_y, &contracted_x, &
    contracted_y, &func_alternate, &x1[highest_index]);

        } else {
        // If the reflected point is neither better nor worse than the second highest
     point, accept it
        points[highest_index][0] = reflected_x;
        points[highest_index][1] = reflected_y;
        x1[highest_index] = func_evaluate;
    }
    //Increment the iteration counter by 1
    num_iterations++;
}

// Output the information of interest
printf("Final Coordinates:\n(%f,%f)\n(%f,%f)\n(%f,%f)\n",points[0][0],points[0][1],
    points[1][0],points[1][1],points[0][2],points[2][0]);
printf("Minimum Point found at (x0,x1) = (%f, %f)\n", points[0][0], points[0][1]);
printf("Function Value, F(x0,x1), at minimum = %f\n", x1[l]);
printf("Number of iterations completed: %d\n", num_iterations);
}

//Main function for running code and implementing functions
int main(){
    //Calling function and stating x1 is equal to 1
    write_to_file(1);
    //Defining the starting vertices of the simplex
    double points[3][2] = {{0.0, 0.0}, {2.0, 0.0}, {0.0, 2.0}};
    //Run the function for the starting coordinates specified
    simplex_method(points);
    return 0;
}
```

End of Report.