PH20105, Hand-in-exercise 2019-2020

Candidate number: 22853

1 – Minimum of Rosenbrock's parabolic valley analytically

The minimum of the valley was first found analytically through use of partial derivatives and by recognising that the sum of the squares of two real numbers cannot be negative.

$$y = F(x_0, x_1) = 100(x_1 - x_0^2)^2 + (1 - x_0)^2 \qquad (1)$$

$$\frac{\partial y}{\partial x_0} = 400x_0^3 - 400x_0x_1 + 2x_0 - 2$$

$$\frac{\partial y}{\partial x_1} = 200(x_1 - x_0^2)$$

At a stationary point:

$$\frac{\partial y}{\partial x_0} = \frac{\partial y}{\partial x_1} = 0$$

Reveals two stationary points:

$$x_1 = x_0^2$$

$$400x_0^3 - 400x_0^3 + 2x_0 - 2 = 0$$

$$x_0 = 1$$

$$x_1 = \pm\sqrt{x_0} = \pm 1$$

Stationary points at (1,1) & (1, -1). F(1,1) = 0 & F(1,-1) = 4. But F is the sum of two squares and so must be greater than or equal to zero. Therefore, the global minima must reside at (1,1).

Confirm with the second derivative test with Hessian determinant H:

$$F_{x_0x_0} = \frac{\partial^2 y}{\partial x_0^2} = 1200x_0^2 - 400x_1 + 2$$

$$F_{x_1x_1} = \frac{\partial^2 y}{\partial x_1^2} = 200$$

$$F_{x_0x_1} = \frac{\partial^2 y}{\partial x_0 \partial x_1} = -400x_0$$

$$H = F_{x_0x_0}(1,1)F_{x_1x_1}(1,1) - F_{x_0x_1}(1,1)^2 = 802 \times 200 - (1600) = 4$$

H is positive therefore the point is either a maximum or a minimum (not a saddle point). $F_{x_0x_0}(1,1)$ & $F_{x_1x_1}(1,1)$ are both positive therefore the point is a minimum as expected. Therefore the global minimum is at (1, 1) with F(1, 1) = 0.

2 – Plot of Rosenbrock's parabolic valley

To better understand the shape of the valley, equation (1) was plotted covering $x_0 = -2 \dots 2$ with constant $x_1 = 1$. This was achieved using the C code shown in Appendix A to accurately calculate F values and record them to a text file. The text file was read and plotted in python using matplotlib as shown in Appendix B.

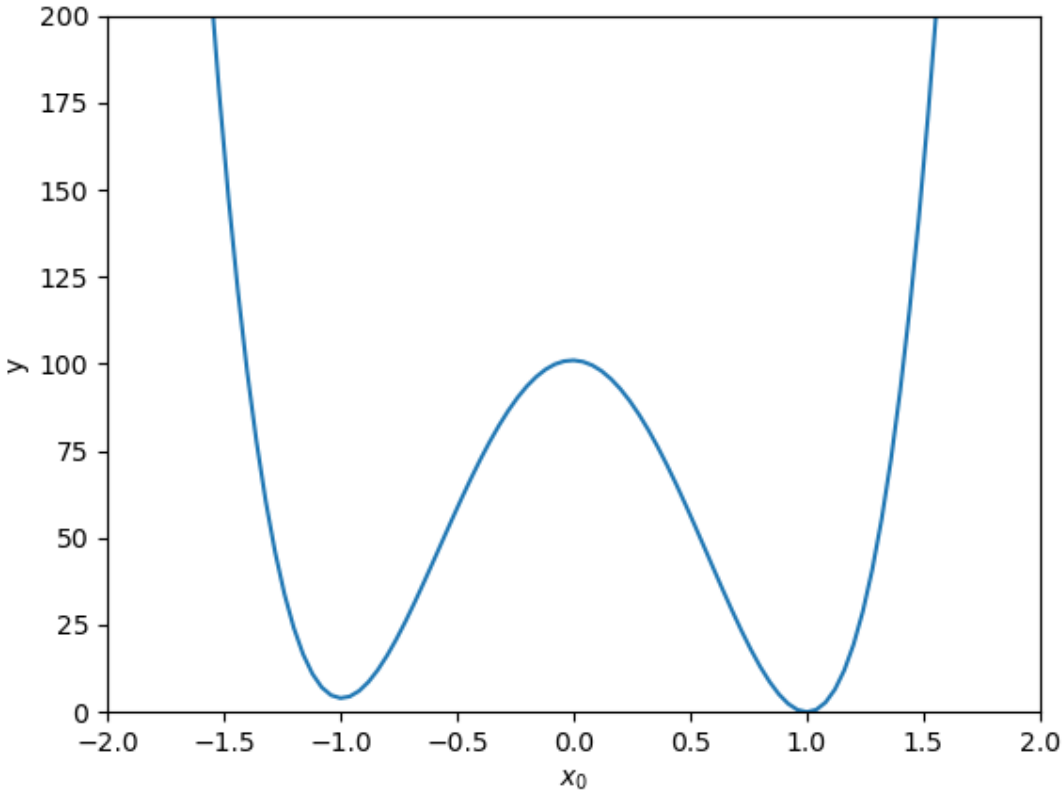

Figure 1. A plot of the Rosenbrock function with the y-axis zoomed to show the slight difference in height between local minima. Plotted in python, see Appendix B.

Figure 1. reveals that what looks like two similar minima from a glance, are in fact slightly different. The plot confirms the calculation from part 1 that the global minimum is at (1,1) at y = 0.

3 – Downhill simplex

The downhill simplex algorithm was followed by hand and the results were as follows:

- ENTER.
- Iteration 1.
- p0 = (0, 0) p1 = (2, 0) p2 = (0, 2)
- y0 = 1 y1 = 1601 y2 = 401
- l = 0 h = 1

- pbar = (0, 1)
- REFLECTION p* = (-2, 2)
- y* = 409
- Is y* < yl? (is 409 < 1) NO
- Is y* > y0 AND y* > y2? (409 > 1 AND 409 > 401) YES
- Is y* > yh? (409 > 1601) NO
- p1 = p* = (-2, 2)
- y1 = 409
- CONTRACTION p** = (-1, 1.5)
- y** = 29
- is y** > y1? (29 > 409) NO
- p1 = p** = (-1, 1.5)
- ybar = (0 + 1.5 + 2)/3 = 7/6 ~= 1.1667
- Standard deviation = $\sqrt{\frac{(0-1.1667)^2+(1.5-1.1667)^2+(2-1.1667)^2}{3}} \sim = 0.85$
- Has minimum been reached? (0.85 >> 10^{-8} & iterations = 1) NO
- Iteration 2.
- p0 = (0, 0) p1 = (-1, 1.5) p2 = (0, 2)
- y0 = 1 y1 = 29 y2 = 401
- l = 0 h = 2
- pbar = (-0.5, 0.75)
- REFLECTION p* = (-1, -0.5)
- y* = 229
- Is y* < yl? (is 229 < 1) NO
- Is y* > y0 AND y* > y1? (229 > 1 AND 229 > 29) YES
- Is y* > yh? (229 > 1601) NO
- p2 = p* = (-1, -0.5)
- y2 = 229
- CONTRACTION p** = (p2 + pbar)/2 = ((-1 + -0.5)/2, (-0.5 + 0.75)/2)
- p** = (-0.75, 0.125)
- y** = 22.2 (3 sig. figs.)
- is y** > y2? (22.3 > 229) NO
- p2 = p** = (-0.75, 0.125)
- ybar = (0 + 1.5 + 0.125)/3 = 13/24 ~= 0.54167
- Standard deviation = $\sqrt{\frac{(0-0.54167)^2+(1.5-0.54167)^2+(0.125-0.54167)^2}{3}} \sim = 0.68$
- Has minimum been reached? (0.68 >> 10^{-8} & iterations = 2) NO
- Result of first 2 iterations:
  - p0 = (0, 0) y0 = 1
  - p1 = (-1, 1.5) y1 = 29
  - p2 = p** = (-0.75, 0.125) y2 = 22.2
  - sd ~= 0.68

The downhill simplex algorithm was implemented in the C code shown in Appendix C. The program successfully found the correct minima with properties as follows.

- The final three points were:
  - p_0 = (1.000017, 1.000041),
  - p_1 = (1.000108, 1.000217),
  - p_2 = (0.999957, 0.999913).
- The final function evaluations were:
  - $F(p_0) = 4.898155 \times 10^{-9}$,
  - $F(p_1) = 1.194000 \times 10^{-8}$,
  - $F(p_2) = 1.870241 \times 10^{-9}$.
- This indicates a convergence to the global minimum at F(1,1) = 0 as expected.
- The standard deviation was 5.196704×10^{-9}.
- The algorithm took 58 iterations.

CMD Output:

```
C:\Users\George\Documents\CCourse\Coursework\Submission>gcc 22853.c

C:\Users\George\Documents\CCourse\Coursework\Submission>a
Minimum reached.
p0 F(1.000017, 1.000041) = 4.898155e-009
p1 F(1.000108, 1.000217) = 1.194000e-008
p2 F(0.999957, 0.999913) = 1.870241e-009
Standard deviation: 5.196704e-009
Iterations taken: 58
```

The code written to solve this problem takes advantage of some features only available in C99 onwards such as C++ style comments, Boolean data types (using stdbool.h) and mixed declarations (`for (int i; …)`)for more elegant for loops.

Appendix A – C code for part 2

```
// Plots F between (x0 = -2 -> 2) with constant (x1 = 1).
void plot(void)
{
    double p[N] = { 0,1 };
    double (*F_ptr)(double p[N]) = &F;

    FILE *fp;
    fp = fopen("output.txt", "w");

    for (int i = 0; i <= 100; i++)
    {
        // Scales i to plot 100 evenly spaced points between -2 and 2
        p[0] = -2 + (4.0/100) * i;
        fprintf(fp, "%f,%f\n", p[0], (*F_ptr)(p));
    }

    fclose(fp);
```

```
}
```

Appendix B – Python code for part 2

```
import matplotlib.pyplot as plt
import numpy as np
data = np.loadtxt("output.txt", delimiter = ',')
x = np.array(data[:,0])
F = np.array(data[:,1])
plt.plot(x, F)
plt.ylabel('y')
plt.xlabel(r'$x_{0}$')
# Changing the axis bounds zooms the graph.
plt.axis([-2, 2, 0, 200])
plt.draw()
plt.show()
```

Appendix C – Source code for part 3

```
/*****************************************************
 * downhillsimplex.c (C99)                           *
 *                                                   *
 * Candidate number: 22853                           *
 *                                                   *
 * Finds the minimum of a particular Rosenbrock      *
 * function using the Nelder-Mead downhill simpex    *
 * method.                                           *
 *                                                   *
 * Setup at starting points:                         *
 * p0 = (0,0), p1 = (2,0), p2 = (0,2)                *
 *                                                   *
 * Compatible with other single valued functions and *
 * functions of more than 2 variables.               *
 *                                                   *
 *****************************************************/

#include <stdio.h>
// For pow & sqrt
#include <math.h>
// For boolean data types
#include <stdbool.h>
// For memcpy
#include <string.h>
```

```c
// For dimensions other than two, the function and starting
// coordinates must be modified.
#define N 2

// Contains all information relating to the N dimensional simplex.
typedef struct
{
    // Contains the coordinates of each point of the simplex.
    double p[N+1][N];
    // Stores the function evaluation at each point.
    double y[N+1];

    // Indices of highest and lowest points.
    int h,l;

    // Position of the centroid.
    double pbar[N];

    // Position and y value of the first trial point.
    double pstar[N];
    double ystar;

    // Position and y value of the second trial point.
    double pstarstar[N];
    double ystarstar;

} simplex;

// Function declarations
double F(double p[N]);
void init_simplex(simplex *s);
void print_points(simplex *s);
void calc_highest_point(simplex *s);
void calc_lowest_point(simplex *s);
void calc_centroid(simplex *s);
void reflect(simplex *s);
void expand(simplex *s);
void contract(simplex *s);
void replace(simplex *s);
void update_y_values(simplex *s);
double standard_deviation(simplex *s);

int main(void)
{
    simplex s;
    init_simplex(&s);
```

```
// Iterations of the algorithm performed.
int iterations = 0;
// The algorithm will finish when it is done.
bool done = false;
while(!done)
{
    // Updates the highest and lowest point.
    calc_highest_point(&s);
    calc_lowest_point(&s);
    // Calculates the centroid.
    calc_centroid(&s);
    // Performs a reflection.
    reflect(&s);

    if (s.ystar < s.y[s.l])
    {
        // Performs an expansion.
        expand(&s);

        if (s.ystarstar < s.y[s.l])
        {
            // Replaces ph with p**
            memcpy(s.p[s.h], s.pstarstar, sizeof(s.p[s.h]));
            update_y_values(&s);
        }
        else
        {
            // Replaces ph with p*
            memcpy(s.p[s.h], s.pstar, sizeof(s.p[s.h]));
            update_y_values(&s);
        }
    }
    else
    {
        // The result is true only if y* is greater than all y values
        // excluding the highest point.
        bool result = true;
        for (int i = 0; i <= N; i++)
        {
            if ((s.ystar > s.y[i] || s.h == i) && result != false)
            {
                result = true;
            }
            else
            {
                result = false;
            }
        }
```

```
        if (result)
        {
            if (!(s.ystar > s.y[s.h]))
            {
                // Replaces ph with p*
                memcpy(s.p[s.h], s.pstar, sizeof(s.p[s.h]));
                update_y_values(&s);
            }

            // Performs a contraction.
            contract(&s);

            if (s.ystarstar > s.y[s.h])
            {
                // Performs a replacement.
                replace(&s);
            }
            else
            {
                // Replaces ph with p**
                memcpy(s.p[s.h], s.pstarstar, sizeof(s.p[s.h]));
                update_y_values(&s);
            }
        }
        else
        {
            // Replaces ph with p*
            memcpy(s.p[s.h], s.pstar, sizeof(s.p[s.h]));
            update_y_values(&s);
        }
    }

    // Algorithm is complete if standard deviation meets the criterion.
    if (standard_deviation(&s) < pow(10, -8))
    {
        done = true;
        printf("Minimum reached.\n");
    }
    // Algorithm terminates if iterations surpass 1000.
    else if (iterations >= 1000)
    {
        done = true;
        printf("Max iterations (1000) reached.\n");
    }
    iterations = iterations + 1;
}
```

```
    print_points(&s);
    printf("Standard deviation: %e\n", standard_deviation(&s));
    printf("Iterations taken: %d\n", iterations);

    return 0;
}

// Function to be minimised, returns function value at point.
double F(double p[N])
{
    // Requires modification to be compatible with higher than 2 dimensions
    return 100*pow(p[1] - pow(p[0], 2), 2) + pow((1 - p[0]), 2);
}

// Plots F between (x0 = -2 -> 2) with constant (x1 = 1).
/*void plot(void)
{
    int i;
    double p[N] = { 0,1 };
    double (*F_ptr)(double p[N]) = &F;

    FILE *fp;
    fp = fopen("output.txt", "w");

    for (i = 0; i <= 100; i++)
    {
        p[0] = -2 + (4.0/100) * (double)i;
        fprintf(fp, "%f,%f\n", p[0], (*F_ptr)(p));
    }

    fclose(fp);
}*/

// Initializes the simplex with starting points.
// Requires modification for larger than 2 dimensions.
void init_simplex(simplex *s) {
    s->p[0][0] = 0;
    s->p[0][1] = 0;
    s->y[0] = F(s->p[0]);

    s->p[1][0] = 2;
    s->p[1][1] = 0;
    s->y[1] = F(s->p[1]);

    s->p[2][0] = 0;
    s->p[2][1] = 2;
    s->y[2] = F(s->p[2]);
}
```

```c
// Prints all of the points and their function evaluation.
void print_points(simplex *s)
{
    for (int i = 0; i <= N; i++)
    {
        printf("p%d F(", i);
        for (int j = 0; j <= N-2; j++)
        {
            // Allows for compatibility with N dimensions requiring
            // N coordinates for N+1 points.

            printf("%f, ", s->p[i][j]);
        }
        printf("%f", s->p[i][N-1]);
        printf(") = %e\n", F(s->p[i]));
    }
}

// Calculates the highest point.
void calc_highest_point(simplex *s)
{
    // Reset highest point.
    s->h = 0;
    for (int i = 0; i <= N; i++)
    {
        if (s->y[i] > s->y[s->h])
        {
            // Swap out highest point with next largest.
            s->h = i;
        }
    }
}

// Calculates the lowest point.
void calc_lowest_point(simplex *s)
{
    // Reset lowest point.
    s->l = 0;
    for (int i = 0; i <= N; i++)
    {
        if (s->y[i] < s->y[s->l])
        {
            // Swap out lowest point with next smallest.
            s->l = i;
        }
    }
}
```

```
// Finds the centroid excluding the highest point.
void calc_centroid(simplex *s)
{
    // Reset pbar.
    for (int i = 0; i <= N-1; i++)
    {
        s->pbar[i] = 0;
    }

    for (int i = 0; i <= N; i++)
    {
        // The highest point is excluded for calculation of the centroid
        if (i != s->h)
        {
            for (int j = 0; j <= N-1; j++)
            {
                s->pbar[j] = s->pbar[j] + (s->p[i][j] * 0.5);
            }
        }
    }
}

// Performs reflection transformation.
void reflect(simplex *s)
{
    for (int i = 0; i <= N-1; i++)
    {
        s->pstar[i] = (2 * s->pbar[i]) - s->p[s->h][i];
    }
    s->ystar = F(s->pstar);
}

// Performs expansion transformation.
void expand(simplex *s)
{
    for (int i = 0; i <= N-1; i++)
    {
        s->pstarstar[i] = (2 * s->pstar[i]) - s->pbar[i];
    }
    s->ystarstar = F(s->pstarstar);
}

// Performs contraction transformation.
void contract(simplex *s)
{
    for (int i = 0; i <= N-1; i++)
    {
```

```
        s->pstarstar[i] = (s->p[s->h][i] + s->pbar[i]) * 0.5;
    }
    s->ystarstar = F(s->pstarstar);
}

// Replaces all points with a point between it and the lowest point.
void replace(simplex *s)
{
    for (int i = 0; i <= N; i++)
    {
        if (i != s-
>h) // the highest point is excluded for calculation of the centroid
        {
            for (int j = 0; j <= N-1; j++)
            {
                s->p[i][j] = (s->p[i][j] + s->p[s->l][j])*0.5;
            }
        }
    }
    update_y_values(s);
}

// Updates y values when points are altered.
void update_y_values(simplex *s)
{
    for (int i = 0; i <= N; i++)
    {
        s->y[i] = F(s->p[i]);
    }
}

// Returns the standard deviation of the y-values.
double standard_deviation(simplex *s)
{
    double sd = 0;
    for (int i = 0; i <= N; i++)
    {
        // Calculate variance.
        sd = sd + pow(s->y[i] - F(s->pbar), 2)*((double)1/N);
    }
    // Return standard deviation as sqrt of variance.
    return sqrt(sd);
}
```