Candidate number: 24563

PH20105 Hand-In-Exercise 2022-23

1-The minimum of the Rosenbrock valley

Logic:

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

y is composed of two squares and therefore $y \geq 0$

Assuming that the minimum occurs at $y = 0$ => $100(x_1 - x_0^2)^2 = 0$ and $(1 - x_0)^2 = 0$

Both terms are squares meaning one cannot be the inverse of the other and therefore both must be equal to zero for $y = 0$.

The square root of zero is zero => $x_1 - x_0^2 = 0$ and $1 - x_0 = 0$

By substituting and solving for x_1 and x_0 it is seen that the minimum occurs at $F(1,1)$

$x_0 = 1$ => $x_1 - (1)^2 = 0$ => $x_1 = 1$

Differentiating:

$$\frac{\partial y}{\partial x_0} = -400x_0(x_1 - x_0^2) - 2(1 - x_0)$$

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

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

From the second equation => $x_1 = x_0^2$

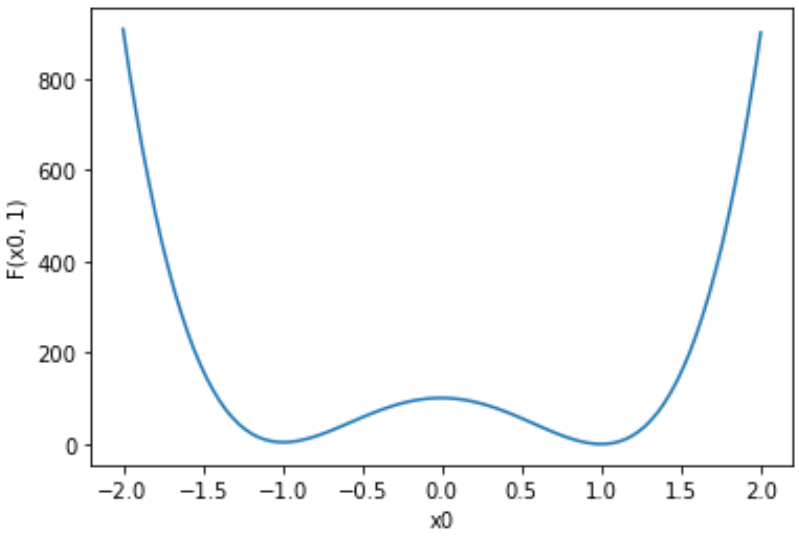
Subbing into the first equation => $x_0 = 1$ and therefore $x_1 = \pm 1$

Subbing $x_1 = -1$ into the original equation, $y = 400$

Subbing $x_1 = 1$ into the original equation, $y = 0$

Hence, the minimum occurs at $F(1,1)$. This is a minimum and not a saddle point since y cannot take any values lower than zero.

2-Rosenbrock's parabolic valley numerically



3-Downhill simplex

The algorithm used in the code is slightly different to the one given in the instructions because it was found that to apply the algorithm in more than two dimensions it must loop over a large section of the algorithm to repeat the step: is $y^* > y_i,\ i \neq h$?

This is quite inefficient, so another version of the algorithm was found that required less steps and didn't employ a loop[1]. The code returned using both methods was the same (the vertices, evaluations and number of iterations) so it was concluded that it was a valid alternative.

1st iteration:

$P_0 = \binom{0}{0},\ P_1 = \binom{2}{0},\ P_2 = \binom{0}{2}$

$y_0 = 1,\ y_1 = 1601,\ y_2 = 401$ => $y_0 = y_l,\ y_1 = y_h,\ y_2 = y_i$ and $P_0 = P_l,\ P_1 = P_h,\ P_2 = P_i$

$\overline{P} = \frac{P_i + P_l}{2} = \binom{0}{1}$

$P^* = 2\overline{P} - P_h = \binom{-2}{2}$ => $y^* = 409$

$y^* > y_i$ => $y^* < y_h$

$P^{**} = \frac{P^* + \overline{P}}{2} = \binom{-1}{1.5}$ => $y^{**} = 29$

$y^{**} < y^*$

$P_h = P^{**} = \binom{-1}{1.5} = P_1$ => $P_0 = \binom{0}{0},\ P_1 = \binom{-1}{1.5},\ P_2 = \binom{0}{2}$

$n = 2,\ \sqrt{\sum_i \frac{(y_i - \overline{y})^2}{n}} \approx 832.7$

$832.7 > 10^{-8}$

2nd iteration:

$P_0 = \binom{0}{0},\ P_1 = \binom{-1}{1.5},\ P_2 = \binom{0}{2}$

$y_0 = 1,\ y_1 = 29,\ y_2 = 401$ => $y_0 = y_l,\ y_1 = y_i,\ y_2 = y_h$ and $P_0 = P_l,\ P_1 = P_i,\ P_2 = P_h$

$\overline{P} = \frac{P_i + P_l}{2} = \binom{-0.5}{0.75}$

$P^* = 2\overline{P} - P_h = \binom{-1}{-0.5}$ => $y^* = 229$

$y^* > y_i$ => $y^* < y_h$

$P^{**} = \frac{P^* + \overline{P}}{2} = \binom{-0.75}{0.125}$ => $y^{**} = \frac{1421}{64} \approx 22.2$

$y^{**} < y^*$

$P_h = P^{**} = \binom{-0.75}{0.125} = P_2$ => $P_0 = \binom{0}{0},\ P_1 = \binom{-1}{1.5},\ P_2 = \binom{-0.75}{0.125}$

$n = 2,\ \sqrt{\sum_i \frac{(y_i - \overline{y})^2}{n}} \approx 223.3$

$223.3 > 10^{-8}$

Coordinates and outputs at the end of the 2nd iteration:

$P_0 = \binom{0}{0}$, $P_1 = \binom{-1}{1.5}$, $P_2 = \binom{-0.75}{0.125}$

$y_0 = 1$, $y_1 = 29$, $y_2 \approx 22.2$

Comparing to the code:

Final vertices:

(0.000000, 0.000000)

(-1.000000, 1.500000)

(-0.750000, 0.125000)

Evaluations:

1.000000

29.000000

22.203125

Iterations: 2

These coordinates match with those calculated previously.

The full code returns:

Final vertices:

(1.000108, 1.000217)

(0.999851, 0.999698)

(1.000017, 1.000041)

Evaluations:

0.000000

0.000000

0.000000

Iterations: 57

Checking the code in higher dimensions:

Implementing the code in the appendix, the Rosenbrock function was replaced for one with three variables:

$$y = F(x_0, x_1, x_2) = (1 - x_0)^2 + 100(x_1 - x_0^2)^2 + 100(x_2 - x_1^2)^2$$

The appropriate alterations were made to the rest of the code by changing the starting vertices and the number of dimensions to 3.

These were the starting positions:

$$P_0 = \begin{pmatrix} -1 \\ 1 \\ -1 \end{pmatrix}, \; P_1 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}, \; P_2 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}, \; P_3 = \begin{pmatrix} 2 \\ 4 \\ 8 \end{pmatrix}$$

The code returns:

Final vertices:

(1.000000, 1.000000, 1.000000)

(1.000010, 1.000030, 1.000061)

(1.000040, 1.000085, 1.000180)

(0.999958, 0.999920, 0.999844)

Evaluations:

0.000000

0.000000

0.000000

0.000000

Iterations: 59

This shows that the code works in multiple dimensions since the minimum of the Rosenbrock function is known to be at $(1, 1, 1)$.

References:

1. Nash, J.C. (1979). Compact Numerical Methods: Linear Algebra and Function Minimisation. Adam Hilger, Bristol. ISBN 978-0-85274-330-0.

Appendix:

2-Rosenbrock's parabolic valley numerically

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

#define N 2 // Number of dimensions

// Structure containing the coordinates of a point
typedef struct {
    double x[N];
} Point;

// Rosenbrock function
double func(Point P) {
    return 100 * pow(P.x[1] - pow(P.x[0], 2), 2) + pow(1 - P.x[0], 2);
}

//Save function values for a range of input values to a file
void file_data(char* textfile, double xlow, double xhigh, int values) {
    double x0 = xlow;
    double step = (xhigh - xlow) / (values - 1);

    FILE* fp = fopen(textfile, "w");

    for (int i = 0; i < 100; i++, x0 += step) {
        Point P = { x0, 1 };
        fprintf(fp, "%.3lf, %.3lf\n", x0, func(P));
    }
    fclose(fp);
}

int main() {
    // Generate and display values in text file
    file_data("data.txt", -2., 2., 100);

    return 0;
}
```

Python plot:

```python
import matplotlib.pyplot as plt
import numpy as np
data = np.loadtxt("data.txt", delimiter = ', ')
x = np.array(data[:,0])
F = np.array(data[:,1])
plt.plot(x, F)
plt.xlabel("x0")
plt.ylabel("F(x0, 1)")
plt.draw()
plt.show()
```

3-Downhill simplex

```c
/*****************************************************
 *                                                   *
 * Candidate number: 24563                           *
 *                                                   *
 * Downhill Simplex algorithm for finding the        *
 * minimum of the Rosenbrock function                *
 *                                                   *
 * Starting vertices:                                *
 * P_0 = (0,0), P_1 = (2,0), P_2 = (0,2)             *
 *                                                   *
 * This code can by amended for higher dimensions by *
 * changing N to the number of variables in the new  *
 * function and changing the initial vertices        *
 *                                                   *
 *****************************************************/

#include <stdio.h>
#include <math.h>
#include <stdbool.h>

#define N 2 // Number of dimensions

// Constants for reflection, contraction, expansion, and shrinking
#define ALPHA 1.0
#define BETA 0.5
#define GAMMA 2.0
#define RHO 0.5

#define TOL 1e-8 // Tolerance for convergence
#define MAX_ITER 1000 // Maximum number of iterations

// Structure containing the coordinates of a point
typedef struct {
    double x[N];
} Point;

// Rosenbrock function
double func(Point P) {
    return 100 * pow(P.x[1] - pow(P.x[0], 2), 2) + pow(1 - P.x[0], 2);
}

// Sort points in ascending order based on function values
void sortPoints(double Y[], Point P[]) {
    double temp;
    Point pTemp;
    for (int i = 0; i < N; i++) {
        for (int j = i + 1; j < N + 1; j++) {
            if (Y[i] > Y[j]) {
                temp = Y[i];
                Y[i] = Y[j];
                Y[j] = temp;

                pTemp = P[i];
                P[i] = P[j];
                P[j] = pTemp;
            }
        }
    }
}

// Calculate centroid (Pbar) for the first N points
void centroid(Point P[], Point* Pbar) {
    for (int i = 0; i < N; i++) {
        Pbar->x[i] = 0;
        for (int j = 0; j < N; j++) {
```

```
            Pbar->x[i] += P[j].x[i] / N;
        }
    }
}


// Reflect the worst (highest value) point (Ph) with respect to the centroid (Pbar)
void reflect(Point* Ps, Point* Pbar, Point P[]) {
    for (int i = 0; i < N; i++) {
        Ps->x[i] = Pbar->x[i] + ALPHA * (Pbar->x[i] - P[N].x[i]);
    }
}

// Contract the worst point (Ph) towards or reflected point (P*) away from the centroid (Pbar)
void contract(Point* Pss, Point P[], Point* Pbar, Point* Ps, bool inside) {
    for (int i = 0; i < N; i++) {
        if (inside) {
            Pss->x[i] = Pbar->x[i] + BETA * (P[N].x[i] - Pbar->x[i]);
        }
        else {
            Pss->x[i] = Pbar->x[i] + BETA * (Ps->x[i] - Pbar->x[i]);
        }
    }
}

// Expand the reflected point (P*) further away from the centroid (Pbar)
void expand(Point* Pss, Point* Ps, Point* Pbar) {
    for (int i = 0; i < N; i++) {
        Pss->x[i] = Pbar->x[i] + GAMMA * (Ps->x[i] - Pbar->x[i]);
    }
}

// Shrink all points towards the best (lowest value) point (Pl)
void shrink(Point P[]) {
    for (int i = 0; i < N + 1; i++) {
        for (int j = 0; j < N; j++) {
            P[i].x[j] = P[0].x[j] + RHO * (P[i].x[j] - P[0].x[j]);
        }
    }
}

// Replace an original point (Ph) with a new point (P* or P**)
void replacePoint(Point* new, Point* orig) {
    *new = *orig;
}

// Test if the simplex has reached convergence (standard deviation < 10^-8)
bool minCon(double Y[]) {
    double Ybar = 0, sum = 0;
    for (int i = 0; i < N + 1; i++) {
        Ybar += Y[i] / (N + 1);
    }
    for (int i = 0; i < N + 1; i++) {
        sum += pow(Y[i] - Ybar, 2) / N;
    }
    return (sqrt(sum) < TOL);
}

// Downhill simplex algorithm implementation
void simplex(Point P[]) {
    Point Pbar, Ps, Pss;
    double Ys, Yss, Y[N + 1];

    int a;

    for (a = 0; a < MAX_ITER; a++) {

        // Evaluate function values for each point
```

```
for (int i = 0; i < N + 1; i++) {
    Y[i] = func(P[i]);
}

// Sort points and find centroid
sortPoints(Y, P);
centroid(P, &Pbar);

// Reflect Ph
reflect(&Ps, &Pbar, P);
Ys = func(Ps);

// If Y* is greater than or equal to Yl but less than the second worst value
if (Ys >= Y[0] && Ys < Y[N - 1]) {
    replacePoint(&P[N], &Ps); // Replace Ph with P*
}

// If Y* is less than Yl
else if (Ys < Y[0]) {
    expand(&Pss, &Ps, &Pbar); // Calculate the expanded point P**
    Yss = func(Pss);

    // If Y** is less than Y*
    if (Yss < Ys) {
        replacePoint(&P[N], &Pss); // Replace Ph with P**
    }

    else {
        replacePoint(&P[N], &Ps); // Replace Ph with P*
    }
}

else {

    // If Y* is less than Yh
    if (Ys < Y[N]) {
        contract(&Pss, P, &Pbar, &Ps, 0); // Calculate P**: contract P* away from Pbar
        Yss = func(Pss);

        // If Y** is less than Y*
        if (Yss < Ys) {
            replacePoint(&P[N], &Pss); // Replace Ph with P**
        }

        else {
            shrink(P); // Shrink all points towards Pl
        }
    }

    else {
        contract(&Pss, P, &Pbar, NULL, 1); // Calculate P**: contract Ph towards Pbar
        Yss = func(Pss);

        // If Y** is less than Yh
        if (Yss < Y[N]) {
            replacePoint(&P[N], &Pss); // Replace Ph with P**
        }

        else {
            shrink(P); // Shrink all points towards Pl
        }
    }
}

// Check for convergence
if (minCon(Y)) {
    break;
}
```

```
    }

    // Print results
    printf("Final vertices:\n");
    for (int i = 0; i < N + 1; i++) {
        printf("(");
        for (int j = 0; j < N; j++) {
            printf("%lf", P[i].x[j]);
            if (j < N - 1) {
                printf(", ");
            }
        }
        printf(")\n");
    }

    printf("\nEvaluations:\n");
    for (int i = 0; i < N + 1; i++) {
        printf("%lf\n", func(P[i]));
    }

    printf("\nIterations: %d\n", a + 1);
}

int main() {
    // Define initial points
    Point P[N + 1] = { {0, 0}, {2, 0}, {0, 2} };

    // Run the Downhill Simplex method
    simplex(P);

    return 0;
}
```