Computing Coursework

Candidate 20626, 10th March 2017

The Minimum of Rosenbrock's Parabolic Valley
Rosenbrock's Parabolic Valley is defined by the function

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

where x_0 and x_1 are 2D plane coordinates, and – by differentiating – it is possible to find the values of x_0 and x_1 of the minimum. Differentiating with respect to x_0 gives

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

and differentiating with respect to x_1 gives

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

Since the minimum of the function must occur when Equations 2 and 3 are equal to zero, Equation 3 can be expanded to find that $x_1 = x_0^2$. Substituting for x_1 into Equation 2,

$$-400x_0(x_0^2 - x_0^2) - 2(1 - x_0) = 0\,, \tag{4}$$

which can be simplified to give $x_0 = 1$ and hence $x_1 = 1$. Thus, the minimum occurs at the point (1, 1), which has the corresponding y value of $y = 0$.

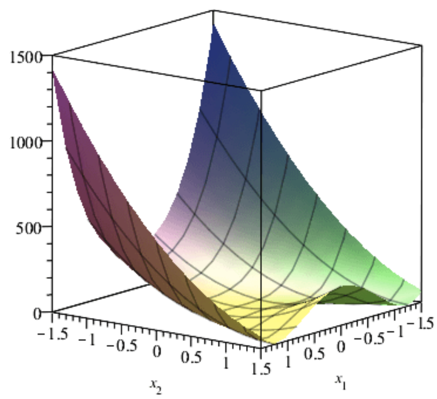

Figure 1. 3D plot of Rosenbrock's Parabolic Valley [1]. y *is represented on the vertical axis.*

Rosenbrock's Parabolic Valley Numerically
By using different functions called from within the 'main' function of a C program, it is possible to numerically determine the minimum point. In the program written below, a function called 'write_file' is called form within 'main', allowing simplification of the code. The 'write_file' sets a loop to cycle through all the possible x_0 values in the defined range $-2 \le x_0 \le 2$, calls to the function 'F(x)' to input different values of x_0 into Equation 1, and writes each x_0 and the corresponding value of $F(x_0, x_1)$ into a file. By writing the data to a text file via the 'write_file' function instead of the

main, the arguments of the file name and the maximum and minimum x_0 values can easily be changed.

```
#include <stdio.h>

double F(double x) {  //Defines the function F(x)

  double x1;  //Defines x1 type

  x1 = 1;
  //Defines value of x1

  return (100*(x1-(x*x))*(x1-(x*x)))+((1-x)*(1-x));
  // Defines output of function
}

void write_file (char *filename, double y0, double y1) {
//Defines the function write_file

  int i;  //Defines i to use as a loop counter
  double x;  //Defines x variable type
  FILE *p_file;  //Points to file location in memory and allows data to be written there

  p_file = fopen(filename, "w");  //Opens file for writing

  for (i = 0; i<=100; i++ ) {  //Defines loop conditions to run through

    x = y0 + i*(y1-y0)/100.;
    //Determines value for x within interval between y0 and y1 values (yet to
    //be specified). Division by 100 allows the loop condition to produce 100 values.

    fprintf(p_file, "%lf, %lf\n", x, F(x));
    // Prints values of x and F(x) to the file (name yet to be defined)
  }
}

int main () {

  write_file("values.txt", -2, 2);
  //Names the file being written to, as well as the values of y0 and y1 (the
  //arguments. This runs the write file function and hence writes the file
  //values.txt with the values from the function F(x).

  printf("Enter 'cat values.txt' to see written data file\n");
  //Makes the user interface more pleasant
}
```

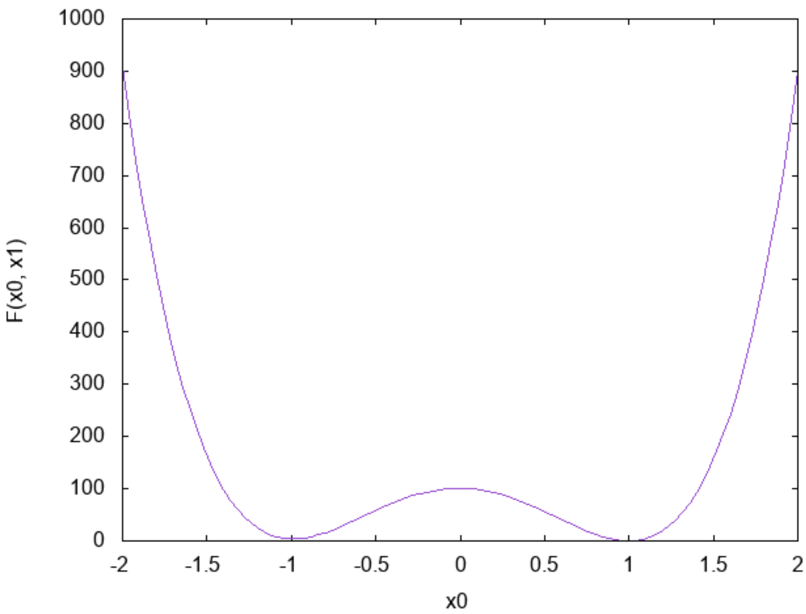

Figure 2. Plot generated from computer program.

Downhill Simplex
Aside from using numerical methods, it is possible to find the minimum of Rosenbrock's Parabolic Valley by applying the Downhill Simplex method via a flowchart [2]. The method takes three different values of x_0 and x_1, allowing a 2D plot of a triangle to be produced, and considers where the vertices lie with respect to the minimum value of the function expressed in Equation 1. The vertices are then repositioned via contraction, expansion and/or reflection to be closer to the minimum. Over many iterations of this method, this allows the vertices to be narrowed down to give the values of x_0 and x_1 at which the function is at a minimum. For the 2D case being considered, the minimum is said to be located once

$$T = \sqrt{\sum_i \frac{(y_i - \bar{y})^2}{2}} < 10^{-8}. \tag{5}$$

By completing the first two iterations by hand, it is possible to see how the vertex positions change and which transformations are applied to the triangle (as illustrated in Figure 3) in the attempt to narrow down on the minimum. To begin with, the vertex positions are set at $P_0 = (0, 0)$, $P_1 = (2, 0)$, $P_2 = (0, 2)$ with corresponding y values of $y_0 = 1$, $y_1 = 1601$ and $y_2 = 401$. By applying one iteration of the method, the new vertices are found to be at $P_0 = (0, 0)$, $P_1 = (-1, 3/2)$, $P_2 = (0, 2)$ with $y_0 = 1$, $y_1 = 29$ and $y_2 = 401$, and $T = 50\sqrt{470}$. This results in the triangle being reflected along the vector $\overrightarrow{P_{0\to 2}}$, and contracted in the positive x_0 direction and in the positive x_1 direction. From the second iteration, it is found that the new vertex positions are $P_0 = (0, 0)$, $P_1 = (-1, 3/2)$, $P_2 = (-3/4, 1/8)$ with $y_0 = 1$, $y_1 = 29$ and $y_2 = 22.83$, and $T = 264.9$. From this, it is seen that the plot is reflected along the vector $\overrightarrow{P_{0\to 1}}$ and contracted in the positive x_1 direction.

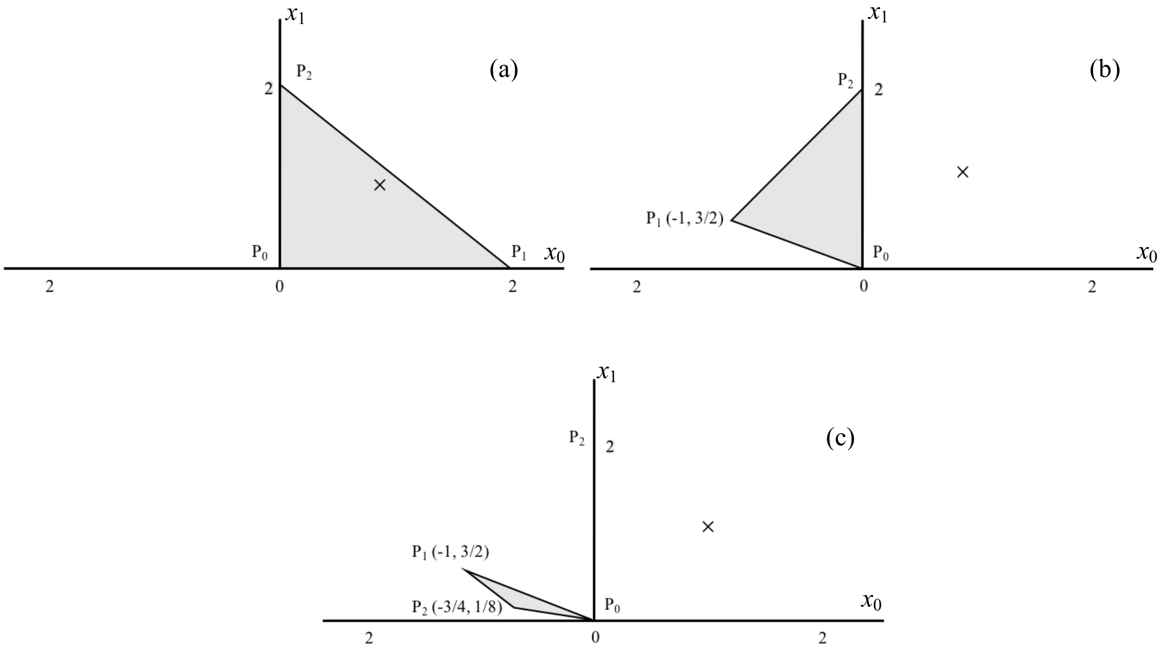

Figure 3. Contractions for the triangular plot described by vertices for each iteration: (a), starting plot; (b), result of 1st iteration; (c), result of 2nd iteration. × represents numerically determined minimum point (1, 1).

Within a C program, it is possible to find the tiny triangular plot formed by a set of vertices in which the minimum point lies. Initially, the functions 'F(x0, x1)' and 'iteration' are declared, and these are called from within the 'main' function later in the program to determine the values of y_0, y_1 and y_2 and T. This allows simplification of the code. In the 'main' function, the initial vertices of $P_0 = (0, 0)$, $P_1 = (2, 0)$, $P_2 = (0, 2)$ are declared first. A FOR loop is then declared – allowing up to 1000 iterations to occur (which may be changed to allow more or fewer iterations) – which first contains an IF tree that evaluates which values of y are largest and smallest, labels each y value accordingly and then labels the vertices P_h, P_m and P_l based upon which y is associated with them. The values of $\bar{P}$, P^* and y* are then calculated. Together, this determines the initial values ready to be inputted into method's flowchart. Next, the FOR loop uses and IF tree which evaluates the flowchart conditions to decide which branches the program should pass down, allowing new vertex positions and their corresponding values of y to be determined. The initial vertex positions are then overwritten so the program can consider the new triangular plot formed in the next loop. Finally, the FOR loop uses three IF and ELSE statements to consider whether the minimum point or maximum number of iterations has been reached, and then outputs to the terminal accordingly.

The use of IF trees in this program allows effective evaluation for every possible state of the system, and additionally makes it easier to follow which part of the code corresponds to which part of the flowchart. Furthermore, the declaration of the 'iteration' function outside 'main' is a versatile and simplified method of checking if the minimum has been reached in each loop.

```
#include <stdio.h>
#include <math.h>  //Required for use of 'pow' function

double F(double x0, double x1) {  //Defines the function F(x0, x1)

  return (100.*(x1-(x0*x0))*(x1-(x0*x0)))+((1.-x0)*(1.-x0));
  //Defines output of this function
}

double iteration(double e, double f, double g, double h) {
//Defines Equation 8 in task script to give iteration condition

  return sqrt((((e - h)*(e - h))/2.) + (((f - h)*(f - h))/2.) + (((g - h)*(g - h))/2.));
  //Defines output of this function
}


int main() {

  double a[2] = {0., 0.};  //Defines initial minimum vertex
  double b[2] = {2., 0.};  //Defines initial maximum vertex
  double c[2] = {0., 2.};  //Defines initial middle-valued vertex

  int N, i; //Defines loop counters for main program and p_bar loop
  double y0, y1, y2, y_star, y_2star, y_bar, yl, ym, yh;
  double p_bar[2], p_star[2], p_2star[2], p0[2], p1[2], p2[2];
  double pl[2], pm[2], ph[2];
  //Declaration of variable types. Note that the largest y value is represented by
  //yh, the middle value by ym and the smallest value by yl. The vertices which
  //correspond to these y values are represented by arrays labeled ph, pm and pl.
  //p0, p1 and p2 are the names of the vertices before they have been ordered from
  //largest to smallest.

  for (N = 0; N <= 1000; N++) {  //Definition of FOR loop conditions for main program

    y0 = F(a[0], a[1]);
    y1 = F(b[0], b[1]);
    y2 = F(c[0], c[1]);
    //Passes arguments to function to find values. These change for each loop and are
    //ordered from largest to smallest by the following IF/ELSE IF conditions.

    p0[0] = a[0]; p0[1] = a[1];
    p1[0] = b[0]; p1[1] = b[1];
    p2[0] = c[0]; p2[1] = c[1];
    //Sets values for vertices using all values in a, b and c arrays. This is reset for
    //each loop using output from flowchart.


    /* Following conditions determine yl, ym, yh and pl, pm, ph */
```

```
if (y0 > y2 && y2 > y1) {
//IF condition where y0 is largest and y1 is smallest
  yh = y0;
  yl = y1;
  ym = y2;
  //Defines values for yl, ym and yh

  ph[0] = p0[0]; ph[1] = p0[1];
  pm[0] = p2[0]; pm[1] = p2[1];
  pl[0] = p1[0]; pl[1] = p1[1];
  //Defines values for pl, pm and ph arrays
}

else if (y0 > y1 && y1 > y2) {
//ELSE IF condition where y0 is largest and y2 is smallest
  yh = y0;
  yl = y2;
  ym = y1;
  //Defines values for yl, ym and yh

  ph[0] = p0[0]; ph[1] = p0[1];
  pm[0] = p1[0]; pm[1] = p1[1];
  pl[0] = p2[0]; pl[1] = p2[1];
  //Defines values for pl, pm and ph arrays
}

else if (y0 < y1 && y1 < y2) {
//ELSE IF condition where y2 is largest and y0 is smallest
  yh = y2;
  yl = y0;
  ym = y1;
  //Defines values for yl, ym and yh

  ph[0] = p2[0]; ph[1] = p2[1];
  pm[0] = p1[0]; pm[1] = p1[1];
  pl[0] = p0[0]; pl[1] = p0[1];
  //Defines values for pl, pm and ph arrays
}

else if (y0 < y2 && y2 < y1) {
//ELSE IF condition where y1 is largest and y0 is smallest
  yh = y1;
  yl = y0;
  ym = y2;
  //Defines values for yl, ym and yh

  ph[0] = p1[0]; ph[1] = p1[1];
  pm[0] = p2[0]; pm[1] = p2[1];
  pl[0] = p0[0]; pl[1] = p0[1];
  //Defines values for pl, pm and ph arrays
}

else if (y1 > y0 && y0 > y2) {
//ELSE IF condition where y1 is largest and y2 is smallest
  yh = y1;
  yl = y2;
  ym = y0;
  //Defines values for yl, ym and yh

  ph[0] = p1[0]; ph[1] = p1[1];
  pm[0] = p0[0]; pm[1] = p0[1];
  pl[0] = p2[0]; pl[1] = p2[1];
  //Defines values for pl, pm and ph arrays
}

else {
//ELSE condition where y2 is largest and y1 is smallest
  yh = y2;
  yl = y1;
  ym = y0;
  //Defines values for yl, ym and yh

  ph[0] = p2[0]; ph[1] = p2[1];
  pm[0] = p0[0]; pm[1] = p0[1];
  pl[0] = p1[0]; pl[1] = p1[1];
  //Defines values for pl, pm and ph arrays
```

```c
}

/* yl, ym, yh and pl, pm, ph have now been determined. The following code
preceding the IF ELSE tree - which considers the different flowchart conditions
- sets all the values for the p_bar, p_star, y_bar and y_star quantities. */

/* This following FOR loop considers all the different cases for pm and pl array
values to find the correct values for the p_bar array values */

for (i = 0; i < 2; i++) {  //Defines a loop to run through array values

  if (pm[i] == 0.) {
    p_bar[i] = pl[i]/2.;  //Defines p_bar array values when pm[i]= 0
  }

  else if (pl[i] == 0.) {
    p_bar[i] = pm[i]/2.;  //Defines p_bar array for when pl[i] = 0
  }

  else if (pm[i] != pl[i]) {
    p_bar[i] = (pl[i] + pm[i])/2.;
    //Defines p_bar array values for when pl and pm are not equal
  }

  else {
    p_bar[i] = pm[i];  //Defines p_bar arrays for when pm[i] = pl[i]
  }
}

p_star[0] = (2. * p_bar[0]) - ph[0];
p_star[1] = (2. * p_bar[1]) - ph[1];
//Defines p_star array values

y_bar = F(p_bar[0], p_bar[1]);
y_star = F(p_star[0], p_star[1]);
//Defines values for y_star and y_bar by passing p_bar and p_star array values
//to F(x0, x1) function.


/* Following IF ELSE tree represents flowchart. The 'NO ->' or 'YES ->'
comments represent movement along flowchart decision tree branches to indicate
which flowchart conditions are being considered in each IF ELSE statement. */

if (y_star < yl) {  //YES
  p_2star[0] = (2. * p_star[0]) - p_bar[0];
  p_2star[1] = (2. * p_star[1]) - p_bar[1];
  //Sets p_2star array values

  y_2star = F(p_2star[0], p_2star[1]);
  //Finds value of y_2star by passing p_2star array values to F(x0, x1) function

  if (y_2star < yl) {  //YES -> YES
    ph[0] = p_2star[0];
    ph[1] = p_2star[1];
    //Changes ph array values to p_2star array values
  }

  else {  //YES -> NO
    ph[0] = p_star[0];
    ph[1] = p_star[1];
    //Changes ph array values to p_star array values
  }
}

else {  //NO

  if (y_star > ym) {  //NO -> YES
  //Implies that y_star is also greater than yl

    if (y_star > yh) {  //NO -> YES -> YES
      p_2star[0] = (ph[0] + p_bar[0])/2.;
      p_2star[1] = (ph[1] + p_bar[1])/2.;
      //Sets p_2star array values

      y_2star = F(p_2star[0], p_2star[1]);
      //Finds value of y_2star by passing p_2star array values to F(x0, x1) function
```

```
        if (y_2star > yh ) {  //NO -> YES -> YES -> YES
          pm[0] = (pm[0] + pl[0])/2.;
          pm[1] = (pm[1] + pl[1])/2.;
          //Changes pm array values

          ph[0] = (ph[0] + pl[0])/2.;
          ph[1] = (ph[1] + pl[1])/2.;
          //Changes ph array values

          /* Note that although flowchart demands that all pi's are changed using
          the above formula, (pl[0 or 1] + pl[0 or 1])/2. = initial pl[0 or 1]
          so pl does not need to be changed. */
        }

        else {  //NO -> YES -> YES -> NO
          ph[0] = p_2star[0];
          ph[1] = p_2star[1];
          //Changes ph array values to p_2star array values
        }
      }

      else {  //NO -> YES -> NO
        ph[0] = p_star[0];
        ph[1] = p_star[1];
        //Changes ph array values to p_star array values

        p_2star[0] = (ph[0] + p_bar[0])/2.;
        p_2star[1] = (ph[1] + p_bar[1])/2.;
        //Sets p_2star array values

        y_2star = F(p_2star[0], p_2star[1]);
        //Finds value of y_2star by passing p_2star array values to F(x0, x1) function

        if (y_2star > yh ) {  //NO -> YES -> NO -> YES
          pm[0] = (pm[0] + pl[0])/2.;
          pm[1] = (pm[1] + pl[1])/2.;
          //Changes pm array values

          ph[0] = (ph[0] + pl[0])/2.;
          ph[1] = (ph[1] + pl[1])/2.;
          //Changes ph array values
        }

        else {  //NO -> YES -> NO -> NO
          ph[0] = p_2star[0];
          ph[1] = p_2star[1];
          //Changes ph array values to p_2star array values
        }
      }
    }

    else {  //NO -> NO
      ph[0] = p_star[0];
      ph[1] = p_star[1];
      //Changes ph array values to p_star array values
    }
  }

/* End of flowchart */


yh = F(ph[0], ph[1]);
ym = F(pm[0], pm[1]);
yl = F(pl[0], pl[1]);
//Re-calculates yh, ym and yl ready to be evaluated by iteration function

a[0] = ph[0]; a[1] = ph[1];
b[0] = pm[0]; b[1] = pm[1];
c[0] = pl[0]; c[1] = pl[1];
//Sets the values of the a, b and c arrays to the new ph, pm and pl array values

/*These following IF ELSE statements decide whether a minimum has been found or not,
and what to output if the program is unsuccessful in finding a minimum */

  if (iteration(yl, ym, yh, y_bar) < pow(10.0, -8.0)) {
  //Condition for if program successful in locating minimum. This occurs when iteration
  //function < 10^-8
```

```
      printf("Minimum located within triangular plot formed by following vertices:\n");
      printf("P0 = (%lf, %lf)\n", a[0], a[1]);
      printf("P1 = (%lf, %lf)\n", b[0], b[1]);
      printf("P2 = (%lf, %lf)\n", c[0], c[1]);
      //Prints final vertices
      printf("yl = %lf, ym = %lf, yh = %lf\n", yl, ym, yh); //Prints final yl, ym & yh values
      printf("Number of iterations = %d\n", (N+1));
      N = 1000; //Sets N = 1000 to immediately end the FOR loop
    }


    else if (N == 1000 && iteration(yl, ym, yh, y_bar) >= pow(10.0, -8.0)) {
    //Condition only satisfied if program fails to find minimum. Prints message to user
      printf("*** Minimum not found ***\n");
    }


    else {
      printf("Iteration no.: %d\n", (N+1));
      //Prints iteration no. to make output easier to interpret
      printf("P0 = (%lf, %lf), ", a[0], a[1]);
      printf("P1 = (%lf, %lf), ", b[0], b[1]);
      printf("P2 = (%lf, %lf)\n", c[0], c[1]);
      //Prints vertices for each iteration
      printf("\n");
      //Prints vertex values and tarts new line to make output easier to read
    }
  }
}
```

References

[1] B., 2015. *Plot of Rosenbrock's Parabolic Valley.* Amoeba Method Optimisation in VBA (Simplex Nedler-Mead) [Internet]. [Cited 23 Feb. 17]. Available from: http://docs.chejunkie.com/method-optimization-in-vba-simplex-nelder-mead/.

[2] Nedler JA, Mead R. A Simplex Method for Function Minimisation. *The Computer Journal.* 1965, Jan, 1; 7(4): 309.