PH20105, Hand-in-exercise 2019-2020

Introduction: description of the problem

The need to perform function minimization is a very common one in research. A ubiquitous example would be fitting a model to a data set, where the best fit represents the case where the difference between model curve and data points is at a minimum. For the hand-in exercise, you will write a routine that is capable of finding the minimum of an arbitrary function with two parameters (and which is easily generalizable to an n-dimensional parameter space). In other words, to establish the lowest value of y for the function

$$y = F(x_0, x_1). \tag{1}$$

Finding the minimum in one dimension is straightforward, and numerically akin to root finding, which is discussed in lecture 5. In one dimension, supposing we know already the interval within which the minimum lies, what we then need to do is to systematically evaluate smaller sub-intervals (or move these around if they begin to miss the minimum altogether) until we converge on a sufficiently tiny interval that contains the minimum.

In more than one dimension, establishing the interval is more tricky. In two dimensions, we need a surface (rather than a one-dimensional line) that wanders through parameter space while trying to enclose the minimum. It is not immediately obvious how to subdivide the surface during the process; the best route towards the minimum does not necessarily lie neatly along one of the axes.

The approach that you are requested to implement is called the *downhill simplex* method for minimization, courtesy of Nelder & Mead (1965). The paper describes some basic manipulations of a *triangle* shape, shown in Fig. 1, which tries to migrate downhill towards the deepest valley of the landscape set out by $F(x_0, x_1)$. The n-dimensional generalization of the triangle is called a *simplex*, hence the name.

The exercise

Your hand-in will consist of a few parts:

- Source code as a single `.c` file that I can compile and run. This must be submitted through the Moodle submission link.
- Your report with (1) answers to the questions below and (2) an appendix which contains the source code. It is fine to use two pages per side for the source code print. A hardcopy of this report must be delivered to the homework box in the Year 2 Lab.

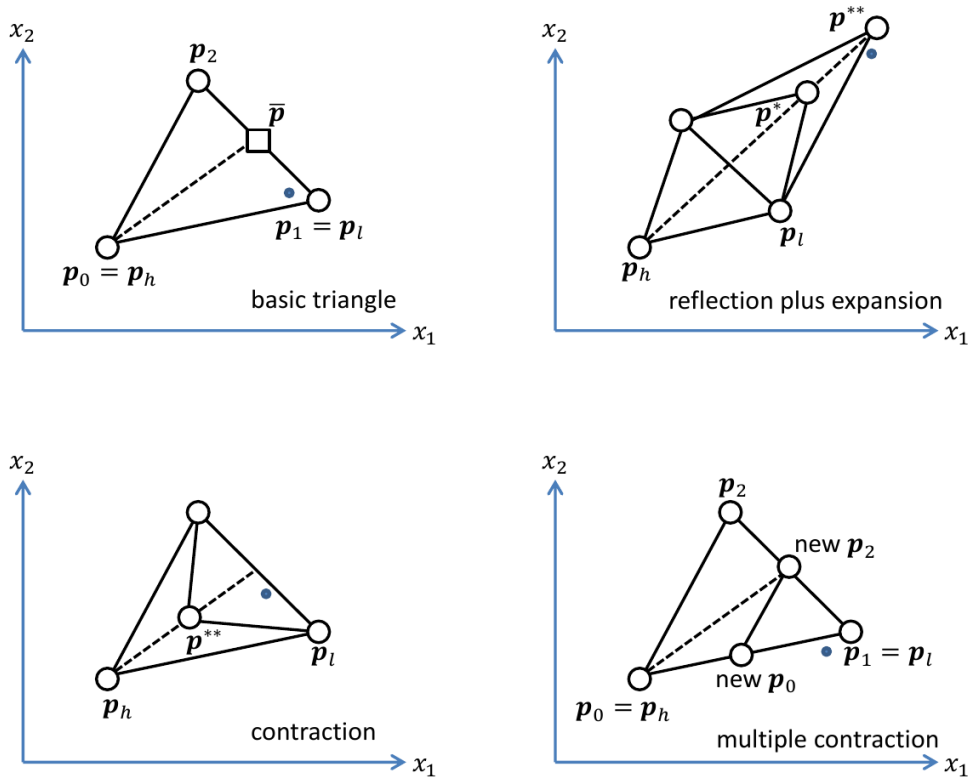

Figure 1: The triangle ("simplex" when generalized to n dimensions). Top left shows the basic triangle, while the other figures show the possible moves. The vertices $\mathbf{p}_i$ are indicated with circles, the midpoint between the $\mathbf{p}_i$, $i \neq h$, with a square. A dot indicates the minimum in parameter space, the idea is that the triangle tries to get all its vertices as close to that point as possible.

- I will also need a copy of your report submitted through Moodle. This must be in `.pdf` format. It is fine not to include the source code in the pdf file, since you will submit the source code as a `.c` file anyway (see above).

I will subtract up to ten points for hand-ins that do not follow these rules.

1 - the minimum of Rosenbrock's parabolic valley (10 points)

The function that we will try to find the minimum of, is called *Rosenbrock's parabolic valley*:

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

This function was designed as a test function for minimization routines. It has a single minimum, but this minimum lies inside a long, narrow, parabolic shaped flat valley. To find the valley is trivial, but convergence to the global optimum is (numerically) difficult. The values of y, x_0, x_1 at the minimum are nevertheless straightforward to compute analytically. As a warm-up, find this minimum y, and its coordinates x_0, x_1 analytically (i.e. without using a computer). *hand-in: In your report, provide a calculation showing minimum value and coordinates.*

2 - Rosenbrock's parabolic valley numerically (20 points)

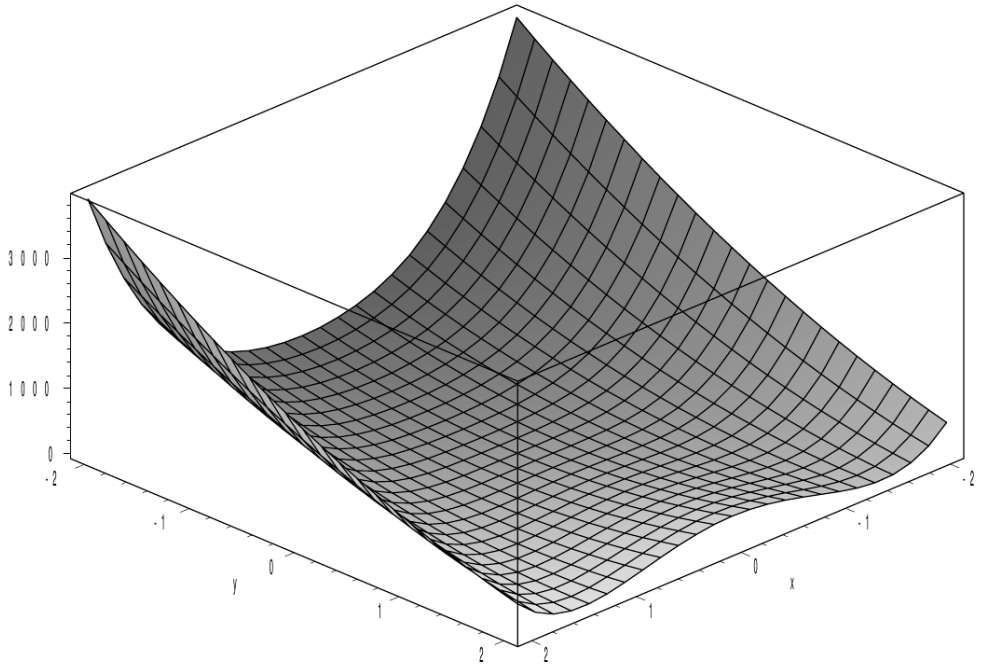

Figure 2: The Rosenbrock valley in a 2D figure. In this figure the x direction is our x_0, the y direction is our x_1. Our function evaluation y is the upwards axis in this figure (not labeled but going up to approximately 4000).

We will need to implement eq. 2 into our program. Once we have done that, we can take a closer look at its the shape to get a feel for its features. A two-dimensional image is provided with Fig. 2, but you will plot a one-dimensional intersection of this. Write a program that writes 100 values of $F(x_0, x_1)$ into a text file, covering $x_0 = -2 \ldots 2$ but keeping x_1 fixed at 1. Don't put the code for writing the date directly in `main`, but use a function instead that is called from within main. This way you can build upon the same code for the minimization program, and just comment out the line in your `main` routine that calls the data-writing function. Show the result in the form of a plot of y against x_0. You will not need C to display the plot, C is merely to compute the function. If you write the data to file in the form of two columns x_0, y separated by a comma, you can use the python code from the first interactive task of lecture 5 verbatim to make your plot. *hand-ins: source code for writing this function content to a text file; image of the plot in your report*.
Your plot will illustrate a common challenge for minimization routines, which is to find the true *global* minimum amid *local* minima that might seem like they represent the lowest value of y, but do not actually do so. Sometimes, it might *appear* that we found the minimum value of the function, only to have missed a far deeper minimum lurking elsewhere in the parameter space. There actually is no definitive answer to that problem, although strong algorithms exist (often using random numbers) that have resolved the issue in a practical sense. This lies beyond the scope of

the exercise, and luckily, in two dimensions, the smaller minimum from your plot is connected to the deeper one by a path through the valley that avoids having to climb uphill again (see Fig. 2).

3 - Downhill simplex (70 points total)

We will now proceed to implement the downhill simplex method. Numerically speaking, finding the minimum of a function essentially boils down to iterating over a *while* loop until some convergence criterion is met. We will keep checking the function values at the three vertices of the triangle, to see which one lies closest to the minimum, and move the other vertices around accordingly by applying one or more moves from figure 1.

some definitions

Before implementing the method from Nelder & Mead (the full paper is attached to the exercise for background reading, but you do not strictly need it), let us first establish some notation to describe the triangle vertices:

- $\mathbf{p}_i$ are vectors describing the points of the triangle, numbered with index i. Each point $\mathbf{p}_i$ being defined by two coordinates x_0, x_1.
- y_i is the function evaluation at the point $\mathbf{p}_i$.
- h is the index of the vertex with the largest y value, i.e. $\mathbf{p}_h$, with value y_h.
- l is the index of the vertex with the lowest y value.
- $\bar{\mathbf{p}}$ is the centroid of the two $\mathbf{p}_i$ with $i \neq h$. So, if $y_0 > y_1, y_2$, we would have x_0 for $\bar{\mathbf{p}}$ be the midpoint between the x_0's of $\mathbf{p}_1$ and $\mathbf{p}_2$, and x_1 for $\bar{\mathbf{p}}$ be the midpoint between the x_1's of $\mathbf{p}_1$ and $\mathbf{p}_2$.
- Trial vertices plus their evaluations are indicated by $\mathbf{p}^*$, y^* and $\mathbf{p}^{**}$, y^{**}.

What the simplex can do

The simplex will be able to contract, expand and reflect, as illustrated in Fig. 1.
The simplex is *reflected* by exchanging $\mathbf{p}_h$ tentatively by $\mathbf{p}^*$, according to

$$\mathbf{p}^* = 2\bar{\mathbf{p}} - \mathbf{p}_h. \tag{3}$$

(Note how $\bar{\mathbf{p}}$ thus ends up the midpoint between and $\mathbf{p}^*$ and $\mathbf{p}_h$). Remember that the equation above is in vector notation, with $\mathbf{p}_i \equiv (x_{0,i}, x_{1,i})$, so actually represents two equations for our two-dimensional case.
The triangle can be subsequently *expanded*, by shifting $\mathbf{p}^*$ further outwards according to

$$\mathbf{p}^{**} = 2\mathbf{p}^* - \bar{\mathbf{p}}. \tag{4}$$

Alternatively, the triangle is *contracted* according to

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

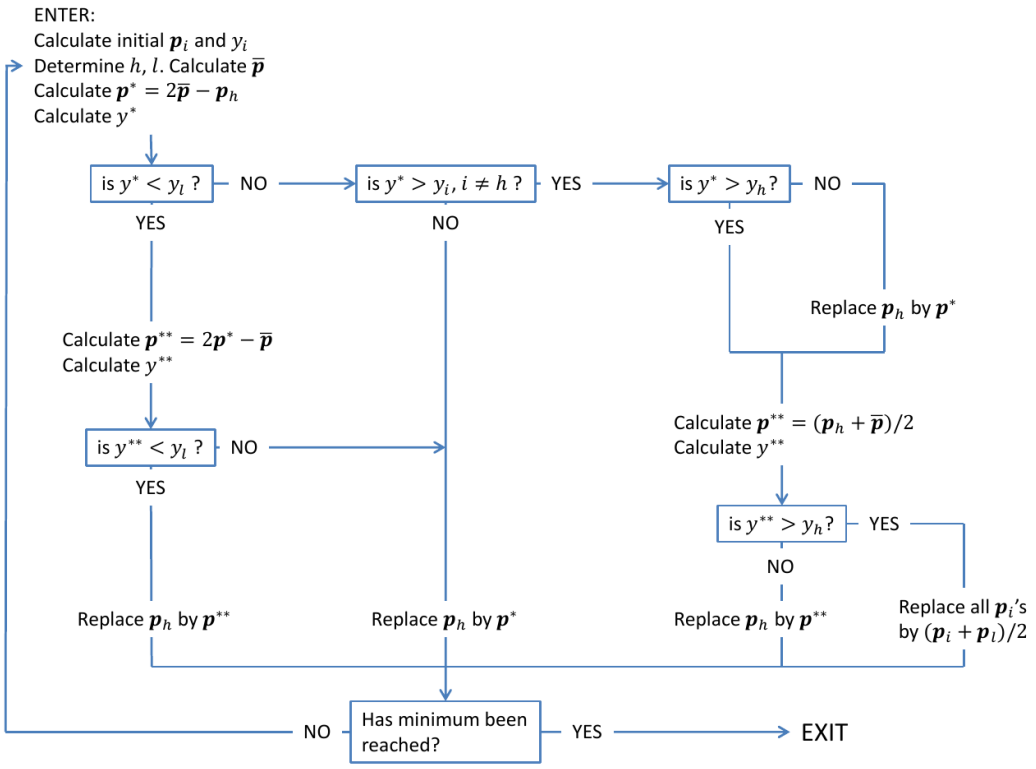

Figure 3: A flow-chart showing the order in which decisions for *reflection*, *expansion* and *contraction* are made.

To find the minimum, you will need to implement the flow chart provided in Fig. 3 in your program, using equation 2 to find your y-values. The flow chart includes a response to a failed contraction as well, where we replace all $\mathbf{p}_i$ at once according to

$$\mathbf{p}_i = \frac{\mathbf{p}_i + \mathbf{p}_l}{2}. \tag{6}$$

Where to start and where to stop

To start the program, first set up your simplex at the following coordinates:

$$P_0 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad P_1 = \begin{pmatrix} 2 \\ 0 \end{pmatrix}, \quad P_2 = \begin{pmatrix} 0 \\ 2 \end{pmatrix}. \tag{7}$$

then begin updating these coordinates according to the flowchart.
We will consider the procedure complete once the following criterion is met (or 1000 iterations are attempted, whichever comes first):

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

The n denotes the number of dimensions (i.e. two in our case).

The hand-ins

Do the first two iterations through the flow chart by hand, noting down the vertex positions, their y values and the type of actions applied to the triangle (10 points). Here, *one iteration* is defined as going from *ENTER* to *has minimum been reached*. Note also the value for the error check of equation 8. *Hand-in: in your report, the first two iterations, a (hand-drawn) figure for the triangle evolution can be illustrative but is not required for full marks*.
Finally, there is the actual program for finding the minimum of the Rosenbrock function using the downhill simplex method. You will need to write the program that does this, prints the final triangle coordinates and evaluations as well as the number of steps that were taken to get there *Hand-in: in your source code, your implementation of the downhill simplex method. In your report, the answers to the questions as well as the appendix with the source*. You'll probably want to use your program to cross-check your hand-written first two iterations, and vice versa.

Between a passing grade and top marks

In principle the program can be written in a very basic manner, using just an implementation of the Rosenbrock function taking two arguments of type double (or float) and returning a double (or float) and a `main` function, with all variables separately defined in the main function and the process covered by a single *while* loop, containing a decision tree of *if* statements. That will get you points enough for a passing grade in the 70's range, assuming that you answered all the warm-up questions correctly.
However, it is also very important that your program is clear and legible. Poorly coded and/or poorly annotated code can cost you points, and a well-crafted program can get you additional points. In particular:

- Be consistent in your conventions, such as where you define your variables, how you position your brackets and when and by how much you indent between lines.
- Choose clear variable names and function titles. Of course, sometimes x or y is clear enough, given the context.
- Properly annotate your code. No need to add a complete essay next to each line, but make sure that you indicate the main steps and explain the more opaque lines of code in such a way that the code could be understood fully when read from a print-out.

In order to obtaing a *high* grade, make the program more elegant. There are various things you can decide to do, for example:

- Use a pointer to a function instead of hard-wiring the Rosenbrock function in your y evaluation.
- Write your code such that it can be generalized to n dimensions very quickly.
- Along the line of the previous point, group the coordinates x_0, x_1 together in a vector (i.e. a little array with two entries).
- Use a *struct* to group all the information about the triangle (simplex) together.
- ...

These are just examples, and you are under no obligation to follow them. As stated, clearly legible code that successfully performs the requested task and is not wildly inefficient when doing so, will be deemed sufficient for a passing grade.
Finally, for those of you who maybe already have prior experience in C programming or who just enjoy an extra challenge: I will accept a program where the triangle (simplex) is a full-fledged *object* with embedded functions rather than a *struct* -even though object-oriented programming is a feature of C++ rather than C, and the topic is merely mentioned in the lecture notes (you'd need to google a bit and you'd need to use `gpp` to compile, instead of `gcc`). In principle, a perfect grade is possible even without taking the object-oriented route.

Final notes

All resources from the programming lab course are available on Moodle as usual and should contain all information you need to complete the assignment successfully. Note that, in addition to lecture slides and task sheets, there are also a glossary, reference guide and extra annotated worked problems. Further additional resources are available on the Internet (e.g. `http://www.cprogramming.com/`) and you are encouraged to discuss the task with your fellow students.
Plagiarism is still plagiarism though, and I expect the code and in particular the comments to look different for different students (barring obvious standard lines like `for(i=0; i<N; i++)` etc). If two programs are obviously copied from one another rather than merely the result of two students discussing the exercise, I can apply a range of measures from docking points for lack of originality (from *both* copies) to passing them along to the Director of Studies as a formal plagiarism case.

HJvE, March 2020