Due date: Maple: Monday, April 18, at the beginning of the lab
C++/Java: Monday, April 25, at the beginning of the lab.

1 Objectives of this Assignment

Our plans for this assignment are

- To implement optimization algorithms in 1D;
- To start exploring optimization in dimension 2.

For this assignment, I impose again specifications for the Java/C++ implementation. When grading your labs, the TA will execute a program that we have written.

2 Optimization in 1D

2.1 A naive gradient descent algorithm

We saw in class that a necessary condition for a point \( x^* \) to be a local minimum of a function \( f \) is that the derivative of \( f \) at \( x^* \) must be equal to 0. The geometric interpretation of this is that the derivative at a point \( x \) indicates the slope of the graph of \( f \) at \((x, f(x))\). If the derivative at \( x \) is not 0, then it means that the slope at \( x \) is tilted one way, and a small step in the “down” direction will bring us to a new point where the value of \( f \) is lower.

This leads to a very simple, if crude algorithm for optimization in 1D: Start from an point \( x_0 \). At step \( k \), determine the derivative of \( f \) at \( x_{k-1} \). If the derivative is positive, take a small step to the left. If it is negative, take a small step to the right. Stop when you feel that you have converged.

Maple 1. Implement the above algorithm. Of course, you will need for this to define more precisely some of the fuzzy terms that I have used. Try to justify in your comments the choices that you make.

2.2 Bracketing

When we search for a local minimum \( x^* \) of a function \( f : x \mapsto f(x) \), we need to bracket that minimum, just as we had to bracket the zero of a nonlinear equation. It is not enough to give the two endpoints, \( a \) and \( b \), of an interval, because we have no way to verify that a minimum is encountered on \([a, b]\) if the only things we know are the values taken by \( f \) at \( a \) and \( b \). When we were seeking a zero of the function, it was sufficient to check that \( f(a) f(b) < 0 \). A minimization bracket is a
triplet \((a, c, b)\) such that \(f(a) > f(c)\) and \(f(b) > f(c)\), as shown in Figure 1. If these conditions are verified, then we know that at least one local minimum must exist over \([a, b]\), that is, an \(x^*\) such that

There exists a (possibly small) neighborhood \(V\) of \(x^*\) such that for all \(x\) in \(V\), \(f(x) > f(x^*)\).

Or, in more rigorous mathematical terms,

\[
\exists \varepsilon > 0, \forall x \in [a, b], |x - x^*| < \varepsilon \implies f(x) > f(x^*).
\]

To refine our estimate of the location of the minimum, we need to subdivide our bracket. If we pick a new point \(d \in [c, b]\), then we have two possible cases, based on the value of \(f(d)\):

- If \(f(d) > f(c)\), then a “safe” interpretation of the known values taken by \(f\) over \([a, b]\) gives us a generic graph that looks like the one on the left side of Figure 2, and our new bracket should be \((a, c, d)\). Note that it does not matter whether \(f(d) > f(b)\), as shown in the figure, or not.

- If \(f(d) < f(c)\), then a “safe” interpretation of the known values taken by \(f\) over \([a, b]\) gives us a generic graph that looks like the one on the right side of Figure 2, and our new bracket should be \((c, d, b)\).
2.3 Where to cut

We could simply say “cut the segment in the middle,” like we did for the bisection algorithm. The problem is that the next time we subdivide, we will not be able to keep the $1/2 - 1/2$ proportion, as shown in Figure 3.

![Figure 3: Further subdivision of the bracket: We cannot always keep the 1/2 – 1/2 proportion.](image)

If our initial bracket is $(a, c, b)$, with $c$ the midpoint of $[a, b]$ (on the left in Figure 3), then the next subdivision will take place at $d$, which should be either the midpoint or $[a, c]$ or that of $[c, b]$. Let’s say that we select the latter, as shown in the central part of Figure 3. The problem with this subdivision is, of course, that we may end up with $(a, d, c)$ as our new bracket, in which case the “central” point is not in the center at all.

To select the best proportions for our bracket, let us start from a bracket $(a, c, b)$ such that $[a, c]$ represents a fraction $\alpha < 1/2$ of the entire segment, $[a, b]$. Of course, $[c, b]$ represents a fraction $1 - \alpha$ of $[a, b]$ (left in Figure 4). Let us now select our next point $d$ in segment $[c, d]$ so that $[b, d]$ represents a fraction $\beta$ of $[a, b]$.

![Figure 4: Further subdivision of the bracket using the golden ratio proportion.](image)

The first thing we can deduce from the proportions of the new bracket, as shown on the right side of Figure 4, is that we should take $\alpha = \beta$. Next, if we want the new bracket to have the same proportions as the old one, we need to select $\alpha$ such that

$$\frac{1 - 2\alpha}{1 - \alpha} = \alpha,$$

which, after you develop it, gives you a nice polynomial equation in terms of $\alpha$:

$$\alpha^2 - 3\alpha + 1 = 0.$$

The solution of this equation that lies between 0 and 1 is $\alpha = (3 - \sqrt{5})/2$, so that $1 - \alpha = (\sqrt{5} - 1)/2$, also known as the golden ratio.

**Maple 2.** Implement the golden section search minimization algorithm.
2.4 Initial bracketing

The golden section search algorithm starts with an initial search bracket \((a, c, b)\). How do we get such a bracket in practice? Typically, we are given either an interval \([x_{\text{min}}, x_{\text{max}}]\) and an initial point \(x_0\) from which to start the search, and we try to determine a search bracket close to \(x_0\). The truth is, there is no pretty, exact way to do that. Everybody hacks some ugly little piece of code that performs this initial bracketing.

Here is the general idea of the initial bracketing. Starting from \(x_0\), we first determine whether the function is increasing or decreasing there. This decides which side of \(x_0\) we will be doing the search. Let’s say that the function is decreasing at \(x_0\). We decide then that \(x_0\) will be the endpoint of our bracket. Having \(a\), you can pick an \(x_1\) very close to \(a\) such that \(f(x_1) < f(a)\). Knowing \(x_1\), you also have a value for the right endpoint of the bracket, \(b\). If you also have \(f(b) > f(x_1)\), then congratulations, you have a bracket. Otherwise, you must increase \(x_1\) until the corresponding \(b\) verifies the condition. Be careful, though that you don’t increase \(x_1\) too fast: if at any point you get \(f(x_1) \geq f(a)\), you went too far and you must decrease \(x_1\) again.

How do we increase/decrease the value of \(x_1\)? Either add a small step or multiply by a scale factor close to 1 (e.g. 1.1 or 0.99). Clever combinations of both techniques are recommended.

3 Explorations in dimension 2

This time we are dealing with a real function of \(n\) variables:

\[
\begin{align*}
    f &: \mathbb{R}^n \rightarrow \mathbb{R} \\
    \mathbf{x} = (x, y) &\mapsto f(\mathbf{x}) = f(x, y).
\end{align*}
\]

As you know, this type of function is typically represented as an “elevation map,” the elevation at \(\mathbf{x} = (x, y)\) being given by the value taken by \(f\) at \(\mathbf{x}\). Our objective is still to find a local minimum \(\mathbf{x}^*\) for \(f\). Let us consider the current “point” \(\mathbf{x}\) in the vicinity of the minimum

3.1 Naive gradient descent

As we saw in class, the gradient of the function indicates the direction of greatest slope of \(f\) (pointing up). It would therefore seem to be a good idea to simply to compute a new gradient at each step and advance in the direction opposite to the gradient by a small amount (the step of the iteration). Ideally, you would want to reduce the length of the step as you get closer to the minimum.

Report 1. Discuss the advantages and problems of such an algorithm.

3.2 Maple implementation

Maple 3. Implement the above algorithm. This time again, there are a number of fuzzy terms and ideas that you will need to clarify: size of the steps, when do we stop, etc.
4 C++/Java Implementation

4.1 \(n\)-dimensional functions

This is a straightforward adaptation of the 1D class.

- The parent class is an abstract class named `FunctionClass`. If you program in C++, you should provide a header file named `NDFunctionClass.h`.

- `NDFunctionClass`'s derived classes should offer the following methods:
  - `public NDFunctionClass(double xmin[], double xmax[])` a constructor taking as parameters the endpoints of the interval along all axes, for the domain on which the function is defined. In C++ you will need to pass along as well the number of variables, \(n\) (as an `int`).
  - `public FunctionClass(int n)` a constructor for a function defined over \(\mathbb{R}^n\).
  - `public double func[](double x[])` returns the value of the function at \(x\). If \(x\) is not in the valid range for the function, an exception should be thrown.
  - `public double jFunc[][](double x[])` returns the value of the function’s Jacobian matrix at \(x\) when this derivative is defined. If \(x\) is not in the valid range for the function, or if the derivative is not defined, an exception should be thrown.
  - `public boolean isJacobianDefined()` returns true if the function’s Jacobian matrix is defined, false otherwise.

4.2 The optimizer classes

5 Evaluation

5.1 What to hand in

A: You should upload to the EnVision server a folder containing the following:

1. A complete, cleaned-up CodeWarrior project folder.
   - Complete means that everything needed to compile and execute the project is there: source files, project file, header files (.h files) and precompiled header source files (.pch files) if you program in C++, test data files.
   - Cleaned-up means that files resulting from the compilation should be removed: .exe files, .sym files, precompiled header binary files (if in C++), and the entire `<name of project>` data folder (close the project before you throw away this folder).

2. Maple worksheets if you did any prototyping using Maple (either remove the results from the worksheet or export the worksheet as text first, please).


B: You should hand in hard copies of your report and of your Maple worksheet.
5.2 Point distribution

The maximum number of points is 100, but extra points could be awarded for excellent aspects of the project or report. The point distribution for this assignment is as follows:

**Maple Modelling**
- Accomplishes what was demanded: 25 pts
- Comments and analysis: 10 pts

**C++/Java Code**
- Accomplishes what was demanded: 15 pts
- Good class design: 10 pts
- General quality & readability: 10 pts

**Report**
- Discussion and analysis of the results: 20 pts
- General quality of the writing and presentation: 10 pts

5.3 Various point penalties

Hopefully we won’t have to apply many of these:

- Project folder incomplete or not properly cleaned up: -5 pts
- Report file missing from the project folder: -5 pts
- Maple file missing: 0 for that part

**Late penalties**
- Printed copy of the report, 1 day late: -5 pts
- Project folder (uploaded to EnVision server), per day late: -5%

If you submit a project late, then it is your responsibility to notify the TA (with CC. to me) that the project is finally available for download on the EnVision server. If you fail to do so, then the “late penalty clock” will keep ticking until the TA gets around to checking your folder on the EnVision server and notices your project. Unless asked explicitly to do so, do not mail your project folder as an attachment.