CS319 – Scientific Computing

Lab 2: An optimization problem (and functions in C++)

In this lab, you will develop an algorithm for solving a classic problem in scientific computing: optimisation problem. That is, find the maximum (or minimum) value of a given function on a given interval. This will give us a context in which to study some of the intricacies of functions in C++, including

- recursion;
- global and local variables;
- pass-by-reference;
- functions as parameters/arguments to other functions.

1 Optimization

Typical optimization problems you might be familiar with include

- at what speed does my car have the best fuel efficiency?
- what processes will minimise my manufacturing costs?
- what is the maximum height a ball will reach if thrown with particular initial velocity? Etc.

We'll study this in a somewhat mathematical framework. We'll take a function, f with a given formula, which we call the *objective function*. We find the value of m that maximises f in a given interval, $[a, b]$. That is, find m such that $a \leq m \leq b$, and $f(m) \geq f(x)$ for all $x \in [a, b]$.

As a specific example, we'll try to maximise $f(x) = e^x - 3x^2 + 2x$ in the interval $[-1, 3]$, as shown in Figure 1. The solution is not available as a simple formula but, to 10 decimal digits, it is 0.6538082734.

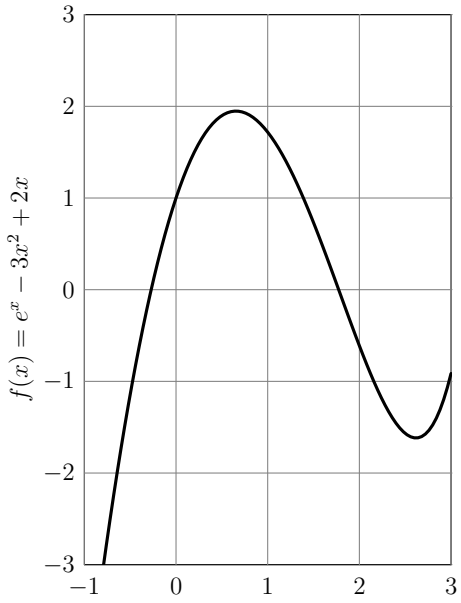

Figure 1: A sample objective function: $f(x) = e^x - 3x^2 + 2x$.

2 Algorithm

We'll solve this problem using a *bisection* algorithm:

1. Choose points a and b (where $a < b$) such that the maximum in between a and b.
2. Take c to be the midpoint of a and b.
3. Take l to be the midpoint of the left interval, a and c, and r to be the midpoint of the right interval, c and b.
4. Compare $f(l)$, $f(c)$ and $f(r)$, and
   - if $f(c)$ is largest set $a = l$, and $b = r$.
   - if $f(l)$ is largest keep a, and set $b = c$.
   - if $f(r)$ is largest keep b, and set $a = c$.
5. Repeat the algorithm until $b - a$ is less than some prescribed tolerance.

The first step is illustrated in Figure 2. In this case, since $f(c) > f(l) > f(r)$, for the second step we will set $a = l$ and $b = r$.

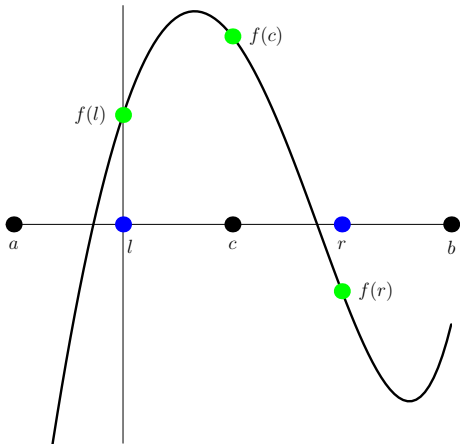

Figure 2: First step of the algorithm

The following code implements the algorithm in a rather basic way

```
double f(double x)
{
  return( exp(x) - 3*x*x + 2*x );
}
```

```
double bisection(double a, double b)
{
    double c = (a+b)/2.0;
    while ( (b-a) > 1e-6)
    {
        c = (a+b)/2.0;
        double l = (a+c)/2.0, r=(c+b)/2.0;

        if ( (f(c) > f(l)) && (f(c) > f(r)) )
        {
            a=l;
            b=r;
        }
        else if ( f(l) > f(r) )
            b=c;
        else
            a=c;
    }
    return(c);
}
```

You can download this program, and its test harness, from the course website. It is called `Bisection.cpp`

This implementation of the bisection algorithm has several shortcoming. Your task is to make as many of the following improvements as possible.

(a) The termination criterion is that the interval containing the maximum is less that 10^{-6}. Different applications may require different bounds. Introduce a *global* variable for this bound, the value of which the user can choose.[1]

(b) To verify the efficiency of an optimisation algorithms, we need to record some basic statistics of its operation. At the very least, we should count the number of iterations (i.e., steps through the loop) that were required.[2]

Modify your code so that a variable which records the number of iterations is passed (by reference) the bisection function. This should be reported by the `main()` function. What data type will you use for this new variable? Why?

(c) It is possible that the `while` loop in the `bisection` function does not terminate. To avoid an infinite loop, add an argument to the `bisection()` function that controls the maximum number of iterations the algorithm will take. A suitable warning message should be generated if the convergence is not achieved.

(d) Write a *recursive* version of the bisection algorithm. Rather than using a `while` loop, we could just check if $(b-a)$ is sufficiently small, and return $(\mathbf{a+b})/\mathbf{2}$ if it is. If it is not small enough, call the `bisection()` function again with new bounds.[3]

(e) The program provided with this lab finds the maximum of the objective function $f(x) = e^x - 3x^2 + 2x$ in the interval $[-1, 3]$. Chosing a different to optimise over is easy: just change the arguments to `bisection`. But what if we want to optimise a different objective function, or more than one objective function?

To make our `bisection()` implementation more general, we would like to pass the name of the objective function to be optimised as one of the parameters. This is reasonably easy in C++. Just add something like the following to the parameter list:
`double ObjFn(double)`
Then :

- you give the name of the objective function you would like optimised as an argument to the bisection function;
- the objective function must take a single `double` as its argument and return a double.

Make this change to your code. Verify it works by finding the maximum of two different objective functions.

.....

Assignment: Write a programme that implements the bisection algorithm, following the suggestions in parts (a)–(e) above. Use you code to solve the following problems:

1. Find two non-negative (real) numbers a and b, such that $a + b = 10$ and the expression $a\sqrt{b}$ is maximized.

2. At particular moment, airplane P_1 is 100km west of airplane P_2. The plane P_1 is travelling north at a constant speed of 500 km/h, while P_2 is travelling west at 800 km/h. What is the smallest distance between the two planes over the next hour? (To solve this problem you may have to convert it from a minimization problem to a maximization problem. Also, your programme needs to report the distance between P_1 and P_2, and not just the point in time when they are closest).

Submit your code though the `labs` section of Blackboard by 9am, Monday 6^{th} Feb.

If you would like to make the task a little more challenging, try writing a version that minimises the number of evaluations of the objective function.

[1] In C++ a variable is local to the function, or code block, where it is defined. However, if the variable is defined outside of the `main()` function, then it is *global*, and shared by all functions. It is very rare that the use of global variables is good practice; it reduces modularity and complicates code re-use. We are using one here just so that we know how they work.

[2] Typically, the most expensive part of the computation is the function call to `f()`. So we should really count the number of times that is evaluated. If you read the description of this exercise in "Solving PDEs in C++", most of the concern is in minimising the number of function evaluations.

[3] In this simple implementation, there are no major advantages to using recursion. However, a variant could be developed to reduce the number of function evaluations.