Programmation 1 - HAX816X - 2021/2022 - Homework - Iterative Methods for Nonlinear Equations

The mandatory deliveries are specified in red color*. Please, respect the naming convention.*

In the following, the bisection and Newton's methods are recalled to iteratively find a solution to a nonlinear equation $f(x) = 0$ in a given interval $[a, b]$. If f is a continuous function on $[a, b]$ and $f(a)$ and $f(b)$ have different signs (one is positive and the other is negative), then the Intermediate Value Theorem guarantees that there is at least one root r in (a, b) such that $f(r) = 0$. The goal of an iterative method is to find a sequence of numbers r_n such that r_n converges to a root r when $n \to \infty$.
On a computer, an iterative process has to be stopped after finite many iterations. The stopping criterion can be $|r_m - r_{m-1}| < b$ or $|f(r_m)| < \epsilon$, for some given small numbers b and ϵ. When either criterion is satisfied, r_m may be taken as an approximation to an exact root r. Due to the approximate nature of an iterative method and computer roundoff errors , r_m may never satisfy $f(r_m) = 0$ exactly, no matter how many iterations are used. The value $f(r_m)$ is called a residual.

1. the Bisection method

The bisection method assumes that f is continuous on $[a, b]$ and $f(a)f(b) < 0$. It first computes the middle point $c = (a + b)/2$ and then tests if $f(a)f(c) < 0$. If it is true, then f must have a root in the smaller interval $[a, c]$. If it is false, then f must have a root in $[c, b]$. In either case, rename the smaller interval as $[a, b]$, which contains a root but whose size is reduced by half. Repeat this process until $|b - a| < \delta$ or $|f(c)| < \epsilon$ with δ and ϵ small, and then the middle point c is taken as an approximate root of f. This algorithm can be written easily in a recursive program.

$\mapsto$ code a function implementing this method in a bisection.cpp file. As a guideline, here is the corresponding declaration:

```
double bisection(double a, double b, double (*f) (double) , double delta, double epsilon)
```

You may write a code with a close resemblance to the algorithm. It may recursively calls the function itself to shrink the interval size by half in each recursion and stops when the interval size or residual is small.

$\mapsto$ this function can be tested in a main_bisection.cpp file to find solutions to :

$$f(x) = x^{-1} - 2^x, \quad [a, b] = [0, 1],$$
$$g(x) = 2^{-x} + e^x + 2cox(x) - 6 \quad [a, b] = [1, 3].$$

Common declarations and definitions can be put in a header file called bisection.hpp. The source code containing function definitions can be put in another file called bisection.cpp. Note that The functions f may involve division by zero. When it occurs, terminate the program by calling the function exit(), declared in header <stdlib.h>.

2. Newton's method

Let $f(x)$ be a function of independent variable x , $f'(x)$ its first-order derivative, and an initial approximation x_p to a root r of f. Newton's algorithm tries to find a better approximation x_n to this root r, and repeat this process until convergence. The idea is to first compute the root of its linear approximation at x_p and then let x_n be this root. The linear approximation of $f(x)$ at x_p is:

$$L_f(x_p) = f(x_p) + f'(x_p)(x - x_p).$$

This is exactly the first two terms of its Taylor's expansion at x_p. Geometrically, the curve $f(x)$ is approximated near the point $(x_p, f(x_p))$ by its tangent line at this point. To find the root of this linear approximation, set $L_f(x_p) = 0$, that is

$$f(x_p) + f'(x_p)(x - x_p) = 0,$$

and solve for x to get $x = x_p - f'(x_p)^{-1} f(x_p)$. Then, let this x be the next approximate root x_n. That is:

$$x_n = x_p - f'(x_p)^{-1} f(x_p).$$

This is the iterative formula of Newton's method. If $|x_n - x_p| < \delta$ or $|f(x_n)| < \epsilon$, where δ and ϵ are two given small positive numers, then stop and take x_n to be the approximate root of $f(x)$ near r. Otherwise, let $x_p = x_n$ and repeat this process. In order to prevent entering an infinite loop, this process should also be stopped when the total number of iterations has exceeded a prescribed number n_max.

↦ code a function implementing the Newton's method in a newton.cpp file. As a guideline, here is the declaration of the required function, to e put in a separate newton.hpp header file:

```
typedef double (*pfn) (double); \\ define a function type
double newton(double xp, pfn f, pfn fd, double delta, double epsilon, int n_max)
```

where fd refers to the derivative of f.

↦ this program can be tested in a main_newton.cpp file to find a root of $f(x) = x^3 - 5x^2 + 3x + 7$ near 5.

You may observe that Newton's method finds an approximate root with very small residual to this polynomial function. In general, the initial approximation x_p (also called initial guess) may not have to be very close to a root r. But a good initial guess can speed up the convergence and even affect the convergence or divergence of the method. That is, Newton's method may converge with one initial guess and diverge with another. Once a good initial guess is obtained, Newton's method converges more rapidly than the bisection method.

3. Organization in a class

To conclude, the root-finding functions can be put together in a class dedicated to ... roots finding. The common declarations are put in a header file. A user can just include the header file and call the method of the class for root-finding applications, with parameters to choose the method.

↦ create a simple class in which the operator() is overloaded for either the bisection or the Newton method, depending on a user provided parameter during the constructor call. This will be put in a separate class_root_finding.hpp header file and a class_root_finding.cpp source file.

↦ test your class in a main_class_root_finding.cpp file.

Notice that the bisection and Newton's methods are programmed here in traditional C style. They require passing a function pointer as argument, which is hard to be inlined or optimized, and may impose function calling overhead for each function evaluation f(x) inside them. Later on in the Lectures, some efficient techniques will be detailed to overcome the function calling overhead in passing a function pointer to another function call.