Programming Concepts in Scientific Computing
Project 5: Non-linear systems

Group 07: Vallez Cyril, Zemour Eliott

January 22, 2021

---

1 Introduction

In this report, we present the implementation of numerical methods for the solution of nonlinear equations (NLE) in C++. An equation is said to be nonlinear when it involves terms of degree higher than 1 in the unknown quantity [1]. These terms may be polynomial or capable of being broken down into Taylor series of degrees higher than 1. The numerical methods considered are the Bisection and other fixed point methods such as Newton and Chord. Finally, we present an extension to systems of nonlinear equations (NLS) solved by the Newton method in $\mathbb{R}^n$.

2 Implementation

2.1 Solver hierarchy structure

We built the following architecture of classes to solve the problem :

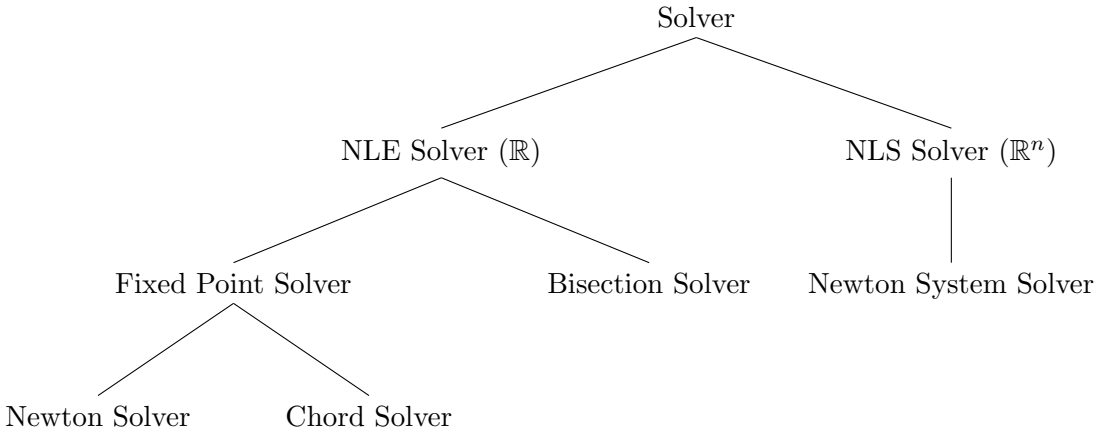

FIGURE 1: Inheritance Tree for NLE Solver class

The classes designed to be instantiated to concretely solve the problem are Newton Solver, Chord Solver and Bisection Solver in the case of scalar non-linear equations, and Newton System Solver in the case of systems of non-linear equations.
The other classes are designed to increase the structure, maintenance and robustness of the whole library.
NLE Solver has a virtual Solve () method, returning a double (the solution of the non-linear equation), while NLS Solver has a virtual Solve () method, returning a std::vector<double> (the solution of the system of non-linear equations). It is this particular Solve method which is defined differently depending on the chosen Solver.

The user then only has to provide some parameters (for example a maximum number of iterations after which the Solver stops, a tolerance, or precision for the solution, etc...), obviously also provide the function to solve for (in the form of a pointer on the function), and then instantiate a Solver with these parameters and call the Solve method.

We also provide external functions which instantiate the class, check for possible exceptions, and then automatically delete the instance and return the solution. This is only a little shortcut, but we found it to be very practical and useful for the user.

2.2 Exceptions hierarchy structure

In order to manage some errors, we built upon an exception system. The structure is very simple and depicted in figure (2).
Even though we could have built only one class for the Div By 0 Exception (error when dividing by 0), we choose to keep this structure in two classes, as in the future the library could be extended and some other forms of exceptions could arise. Those new exceptions would just have to be inherited from the base class Exception, which would make the code clearer and easier to understand.

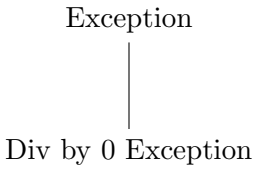

FIGURE 2: Inheritance Tree for the Exceptions system

The Div By 0 Exception is only thrown in the Solve method for Newton Solver and Chord Solver. We used it to try to Solve the problem a second time (and only a second time) changing slightly the initial guess (in fact adding 1 to it). This is particularly useful for Newton Solver. Indeed, the user may quite frequently set his initial guess on a zero of the derivative of the function to Solve for, resulting immediately in a division by 0. Our way allows the program not to crash instantly, but instead to change the initial guess in the hope of finding a point which is non-zero for the derivative, and then Solve as usual.
This is why we try only a second time : it would make no sense to change 20 times the initial guess and trying to Solve each time if the guess is really far away from the root, and the problem is therefore numerically unsolvable.

2.3 Handling other forms of errors

For some other source of errors, we decided not to use the Exception system. For example, when instantiating a Solver we require the user to precise (if he wants to, otherwise we provide a default value) some parameters, like a maximum number of iterations, or a tolerance. Of course, if the number of iterations is not an integer, or is not strictly positive, the Solver will not work. For this reason, if this case arises, we ask the user to re-enter a correct value at the keyboard, until he prompts a correct value or explicitly stop the program.
For the pointers on the function to Solve for (and possibly its derivative if using Newton Solver), we require that the pointers are not null. If this is the case, we just stop the execution of the program, since it is not possible for the user to re-enter the value of the pointer at the keyboard.

3 Tests

3.1 Google testing framework: GoogleTest

The numerical methods implemented are tested using Google testing framework GoogleTest[1]. Indeed, for several non linear functions, we verified that $|f(x^*)|$ was small for x^* being the point to which the method has converged.

[1] https://github.com/google/googletest

3.1.1 NLE Solver

We tested our implementation on different non-linear functions for all NLE Solvers (Bisection, Chord and Newton), namely :

$$\begin{cases} f(x) = x^3 - 2 = 0 \\ f(x) = \sin(x) - 1/2 = 0 \\ f(x) = \exp(-x^2) + \cos(x) - 3/2 = 0 \\ f(x) = x^6 + x^5 = 0 \end{cases}$$

In all tests, we checked that the solution returned by the program was close enough to the solution, to a precision decided by the argument tolerance.
We also tested that the algorithm does throw a Div by 0 Exception in Newton Solver when setting the initial guess on a zero of the derivative.

3.1.2 Aitken acceleration

Furthermore, the Aitken acceleration feature is tested using the `std::chrono` library[2]. Indeed, as one accelerated step of a fixed point method is clearly more costly than a non-accelerated one, it does not make much sense to compare the number of iterations for reaching convergence. Instead, we aimed to compare the computational time of such methods. However, for the numerical methods considered, solving the NLE is very fast in time and the computational time of such methods is mainly caused by the output of the algorithms (via `std::cout`). Therefore, in order to test the acceleration feature, we manually suppressed the outputs (by commenting the `std::cout` lines) in order to evaluate the true computational time with and without the acceleration feature. Each algorithm is run 100 000 times so that we measure a non-zero time. As a result, we found that for both Chord and Newton algorithms, the Aitken acceleration feature does not improve the computational time (even if the methods converge in a smaller number of iterations).

3.1.3 NLS Solver

Newton System Solver was tested on a simple non-linear system in 2D given by :

$$\mathbf{f}(\mathbf{x_1}, \mathbf{x_2}) = \begin{pmatrix} \exp(x_1) + x_2 - 4 \\ x_2^2 - 4 \end{pmatrix} = \mathbf{0}$$

Since this part was only an extension of the principal program (NLE Solvers), we did not run more tests other than this one, which was working correctly.

4 Problems and limitations

4.1 Theoretical limitations

Let $f(x) = 0$ be the non-linear equation that we want to solve.
First, note that numerical methods for solving non-linear equations are not always stable, independently of the implementation of the solver. The first requirement (other than that $f(x)$ indeed have a real root) is that the function $f(x)$ is continuous. If this is the case, then Bisection Solver is assured to converge towards a root. For this reason, Bisection Solver is the most robust of all solver, but also the slower. Chord and Newton Solver are faster, but requires more conditions for the convergence to be assured. Newton for example requires that the derivative exists and does not cancel. For the exact mathematical formulation and exact convergence conditions, see [1].
So for example, roots of piece-wise continuous functions may be found by the Solvers, but not necessarily (for example for functions containing $\tan(g(x))$, or $\arccos(g(x))$, or even $1/x$ etc...) .
Moreover, if the functions $f(x)$ has more than 1 root, the root to which the Solvers will converge depends on the initial guess (or left and right edge for Bisection Solver) and even on the Solver itself (Chord and Newton may not converge to the same root from the same initial guess).

[2] https://www.cplusplus.com/reference/chrono/

4.2 Problems and limitations of implementation

One major limitation of our implementation is the incapacity to check for NaN which could be arising from the function given by the user. For example, if the user provide the function $1/x$ and the initial guess $x = 0$, the evaluation $f(x)$ will result in $f(x) = \text{NaN}$, which will not stop the program but rather continue the iteration process until the max iteration has been reached. The program will then return NaN. So there is a waste of computational time, and a lack of feedback to the user to give him insight of what went wrong. The same goes if $f(x_n)$ (or $f'(x_n)$ for Newton's method) goes to infinity for any iteration $n <$ max iter. This is also true if any of the components of the inverse jacobian matrix diverges at one step of the iteration process for Newton System Solver.

The implementation of NLS Solver also contains small limitation for the user. Indeed, in order to check that the size of the function is the same as the size of the initial guess, the class has an attribute *dimension*. But when setting a new function or new initial guess, we check that the actual dimension is coherent with the new attribute. This means that if a user instantiate an object inheriting from NLS Solver (the only possibility in this version of the code is Newton System Solver), and then wants to solve a system of different dimension with the same instance, he must set the new dimension BEFORE setting the new function and new initial guess (and new inverse jacobian). This is a bit restrictive, since the user must be well aware of this fact for the program to run normally.

5 Conclusion

We presented in this report our implementation of a library for solving scalar non-linear equations using Newton, Chord and Bisection methods, as well as an extension for solving non-linear systems of equations, using Newton's method. The implementation was made in C++. It was tested using GoogleTest, and could be further refined in the future, for example by improving the limitations pointed out in section 4.2.

References

[1] Louis Esch, Robert Kieffer and Thierry Lopez, *Asset and Risk Management: Risk Oriented Finance*, pp. 375-381. Appendix 8: *Numerical Methods for Solving Nonlinear Equations*. Available at: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118673515.app8.