Assignments

Bachar Bouazza

February 12, 2017

1 Newton-Raphson method

1.1 General setting

The problem of root extraction is an old mathematical problem. Also, the, more general, problem of finding the set of roots of an arbitrary function is well known inn the literature.

The problem is to find $x : f(x) = 0$, where $f \in \mathcal{D}$. In this setting $\mathcal{D}$ is the class of differentiable functions or smooth functions.

$$\begin{aligned} \Psi \colon \mathcal{D} &\to \mathbb{R} \\ f &\mapsto \Psi(f), \end{aligned} \tag{1}$$

$\Psi(f)$ is the set of roots of the function f, in other words:

$$\forall x \in \Psi(f),\ f(x) = 0$$

. The first question to ask is whether $\Psi(f)$ is a function, some thing computable. This definitely is a high-order function or a functional. In the jargon of functional programming, this is an example of operator function. Nevertheless, we will just call it a function.

There are functions without roots for which $\Psi(f) = \emptyset$, the exponential function is one example. It is proven that the class of polynomial functions have finitely many roots, and even, in this case, the extraction of roots is a complex operation. The trigonometric functions ($\sin, \cos, \tan$, for example, have infinitely many roots.

The problem can be generalized to higher dimensional spaces by considering general scalar and vectorial functions. We will see in the implementation section that C++ language, with its template mechanism and generic library, provides ways to perform this kind of generalization.

1.2 Solution and Algorithm

The method of Newton-Raphson is an heuristic solution to the problem of finding roots of an arbitrary differentiable function. The condition of differentiability is a pre-condition in order to use the method. The stability of the method depends on the degree of smoothness of the function. The method computes one, and only one, of the roots of the input function. It takes 3 parameters, (f, x_0, n), where f is the function for which root extraction is sought, x_0 is an initial guess, and n is the maximum number of iterations. The method can be seen as a mathematical function of the form

$$\begin{aligned} \Phi \colon \mathcal{D} \times \mathbb{R} \times \mathbb{N} &\to \mathbb{R} \\ (f, x_0, n) &\mapsto \Phi(f, x_0, n), \end{aligned} \tag{2}$$

where $\Psi(f, x_0, n)$ represents the root of the function f, given the initial guess x_0 and the number of iterations n. This is a pure function in the sense that every point in $(\mathcal{D} \times \mathbb{R} \times \mathbb{N})$ has a unique image in $\mathbb{R}$.

The following listing (Algorithm 1) is the method written in a pseudocode format.

```
Input: $(f, x_0, n) \in \mathcal{C}^1 \times \mathbb{R} \times \mathbb{N}$
Output: $x_1$
$x_1 \leftarrow x_0$
for $i \in \{1.. \cdots n\}$ do
    $x_1 \leftarrow \varphi(f, x_0)$
    if $(!Equal(x_0, x_1))$ then
        if $(x_1 \notin \mathcal{V}_{x_0})$ then
            $x_0 \leftarrow x_1$
        else
            break
        end
    end
    $x_1 \leftarrow x_0$
    break
end
return $x_1$
```

Algorithm 1: Newton-Raphson: $\Phi(f, x_0, n)$

The algorithm is a *composition* of one initialization step, one sequential loop, two decision points, and a function call. The function φ is the core function of the algorithm. This function embody the idea behind the method.

It is where the derivation is taken. This is where the stability condition is needed. Its mathematical definition is given by:

$$\begin{aligned} \varphi\colon \mathcal{D} \times \mathbb{R} \times &\to \mathbb{R} \\ (f, x_0) &\mapsto x_1 = \varphi(f, x_0). \end{aligned} \tag{3}$$

This function takes two parameters, (f, x_0), where f is any differentiable function, and x_0 is a guess, *a priori* guess. Its output is a computed guess, x_1. The relationship between between the input guess and the output can be summerized as:

$$x_0 \preceq x_1.$$

If we consider the sequence as defined by this function:

$$x_1 = \varphi(f, x_0), x_2 = \varphi(f, x_1), \cdots x_n = \varphi(f, x_{n-1}),$$

we will have the following partial order between the sequence of guesses:

$$x_0 \preceq x_1 \preceq x_2, \cdots x_0 \preceq x_n.$$

Notice that the order is not strict and we may end up in stationary points or degenerate sequences. The question is when to stop. The method of Newton-Raphson uses three mechanisms to stop searching: when the maximum number of iteration is reached, if the derivative at the input point is null, and another test whether the new guess is in the neighbourhood of the input guess.

The algorithm describing the function φ is given by the pseudo code (Algorithm 2):

```
**Input:** $(f, x_0) \in \mathcal{C}^1 \times \mathbb{R}$
**Output:** $x_1$
$x_1 \leftarrow x_0$
$y'_0 \leftarrow f'(x_0)$
**if** $(!EqualTo0(y'_0))$ **then**
    $y_0 \leftarrow f(x_0)$
    $x_1 \leftarrow x_0 - \frac{y_0}{y'_0}$
**end**
return $x_1$
```

Algorithm 2: Kernel: $\varphi(f, x_0)$

In some sense the method is similar to search problems. The initial choice, x_0, influences the speed by which the method converge and to where it converges if the input function has more than one root. Another important notice is that the method will always give an output. For example, if the function doesn't have a root, the output guess is the same as the input guess.

Finally, the following property

$$\forall f, x, n \;\; \Phi(f, \Phi(f, x_0, n), n) = x_0$$

must be hold and used especially for testing.

1.3 Performance

In the worst case scenarios the algorithm performance is of order $O(n)$. The most important operation is the derivative calculation. There are many methods and algorithms to numerically compute the derivative. The method doesn't use much memory. It is a computing bound type of algorithm. Adding a caching mechanism will boost the performance of the algorithm considerably. If the function, the initial guess, and the number of iteration are fixed, then the root of is fixed.

Meyers in [2] addresses the idea of caching roots of a polynomial function. The caching does not come for free, there is an inherent race condition situation that must be properly addressed. Meyers provides an implementation solution to the problem.

It is well known that for loops are a source of performance issues.

1.4 Implementation

The following code transcript **??** is an implementation of the algorithm functional defined by [3] , in *C++* language. The added parameters textit df, isVerySmall, isCloseEnough are necessary artifacts to allow for different numerical methods, schemes, to implement the derivation concept, the other two are also necessary in order to support different ways to implement relative difference measures. For instance, there are many valid ways to numerically derive a function at a point, the two point methods are among the most known in the literature. The other parameters are very close to each and are used in order to

We are in the situation of the, behavioural design patterns: strategy and template method patterns. I adopted policy-based design (PBD) techniques that are used in many C++ libraries and applications. I first saw this technique in Alexandrescu [1], where it is used in the context of template programming.

```
\label{lst:a_label}
#pragma once
#include "types.hpp"

template <typename DF = Derivative, typename VS, typename CE>
auto newtonRaphson (Function f, Real x0, n =100,
                    DF df, VS isVerySmall, CE isCloseEnough) {

    for (auto i = 0; i < n; i++) {
        auto y0 = f(x0);
        auto dy0 = df(f, x0);

        if (isVerySmall(dy0)) break;  // x0 is a stationary point.

        auto x1=x0-y0/dy0;

        if (isCloseEnough(x0, x1)) {  // No improvement
            x0 = x1;
            break;
        }
        // x1 is a better guess than x0.
        // We restart research using the new guess.
        x0=x1;
    }
    return x0;
}


```

1.5 Generalization

Meyers [2]

1.6 exact environment .....

This is an interesting problem because it allows us to explore high-order functions: functionals and function operator. A functional is simply a function taking at least a function object as a parameter and returns a vector. In the other hand, a function operator returns at least one function object. The Newton-Raphson method is a trivial functional because it takes a first class function, as input, and returns its root. In the same time, the method requires the first derivative of the input function. The derivative operator is a classical function operator: taking a function object as input and returning its derivative object. Other examples are included in STL: algorithm, functional, map reduce .....

1.7 discussion

1.8 Testing

Using C++ testunit ...... I didn't perform numerical stability testing for example.

2 Python

The implementation of this problem is done on python. The following packages were used. I used the following four packages to solve the problem and for testing purposes: pylru, web, datetime, re, unittest, requests, and ast.

2.1 Testability

We need two unit testing and one integration testing. One unit test suit for the *inmemorycache interface, another test suit for the service, and an integration test suit to test the whole system.*

Since the driving idea behind this exercise is performance, we need a way to set up performance testing: comparing a cached store to non cached store. I didn't do it but it can be done. Setting up a service with a cached store and another one without the caching and time requests. Using this ad hock method can allow us to investigate the gained performance.

2.2 to do list

I didn't test $today expansion feature because of client/server time difference. My code need commenting. Design meaningful unit tests for the inmemory cache.

References

[1] *A. Alexandrescu.* Modern C++ Design: Generic Programming and Design Patterns Applied. *Addison-Wesley, Reading, MA., 2nd edition edition, 2001.*

[2] *S. Meyers.* Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. *O'Reilly Media, 2nd edition edition, 2014.*