Chapter 1

Root finding algorithms

1.1 Root finding problem

In this chapter, we will discuss several general algorithms that find a solution of the following canonical equation

$$f(x) = 0 \tag{1.1}$$

Here $f(x)$ is any function of one scalar variable x[1], and an x which satisfies the above equation is called a *root* of the function f. Often, we are given a problem which looks slightly different:

$$h(x) = g(x) \tag{1.2}$$

But it is easy to transform to the canonical form with the following relabeling

$$f(x) = h(x) - g(x) = 0 \tag{1.3}$$

Example

$$3x^3 + 2 = \sin x \quad \rightarrow \quad 3x^3 + 2 - \sin x = 0 \tag{1.4}$$

For some problems of this type, there are methods to find analytical or closed-form solutions. Recall, for example, the quadratic equation problem, which we discussed in detail in chapter 4. Whenever it is possible, we should use the closed-form solutions. They are

[1] Methods to solve a more general equation in the form $\vec{f}(\vec{x}) = 0$ are considered in chapter 13, which covers optimization.

usually exact, and their implementations are much faster. However, an analytical solution might not exist for a general equation, i.e. our problem is *transcendental*.

Example

The following equation is transcendental

$$e^x - 10x = 0 \tag{1.5}$$

We will formulate methods which are agnostic to the functional form of eq. (1.1) in the following text[2].

1.2 Trial and error method

Broadly speaking, all methods presented in this chapter are of the *trial and error* type. One can attempt to obtain the solution by just guessing it, and eventually he would succeed. Clearly, the probability of success is quite small in every attempt. However, each guess can provide some clues which would point us in the right direction. The main difference between algorithms is in how the next guess is formed.

A general numerical root finding algorithm is the following

- Make a guess (x_i)
- Make a new **intelligent** guess (x_{i+1}) based on this trial x_i and $f(x_i)$
- Repeat, as long as, the required precision on the function closeness to zero

$$|f(x_{i+1})| < \varepsilon_f \tag{1.6}$$

and solution convergence

$$|x_{i+1} - x_i| < \varepsilon_x \tag{1.7}$$

are not reached. The solution convergence check is optional, but it provides estimates on the solution precision.

[2] MATLAB has built-in functions which can solve the root finding problem. However, programming the algorithms outlined below has great educational value. Also, studying general methods for finding the solution might help you in the future, when you will have to make your own implementation in a programming language which does not have a built-in root finder. Besides, if you know what is under the hood, you can use it more efficiently or avoid misuse.

1.3 Bisection method

To understand the bisection method, let's consider a simple game: someone thinks of any integer number between 1 and 100, and our job is to guess it.

If there are no clues provided, we would have to probe every possible number. It might take as many as **100** attempts to guess correctly. Now, suppose that after each guess, we are getting a clue: our guess is "high" or "low" when compared to the number in question. Then we can split the search region in half with every iteration. We would need at most only **7** attempts to get the answer, if we do it right. Clearly, this is much faster.

Example

Let's say the number to find is 58.

1. For the first guess, we choose the middle point in the interval 1–100, which yields the first guess 50.
2. We hear that our guess is "low", so we will search in the upper half: 50–100. The second guess will be in mid of 50–100, i.e. 75.
3. We hear that this is "high", so we divide 50–75 in half again. For the third guess, we say 63.
4. We hear "high". We split 50–63 in half again, and say 56.
5. We hear "low". So, we split the upper half of the region 56–63, and say 59.
6. We hear "high", and split the low part of 56–59, and say 57.
7. We hear "low", so we made our final matching guess 58.

In total we made 7 guesses, which is the worst case scenario for this strategy.

The shown example outlines the idea of the bisection method: divide the region in two equal halves, and operate on the remaining half. Below is the *pseudo-code*[3] for this algorithm, which works for any continuous function provided that we *bracketed* the root, i.e. we provided two points at which our function has opposite signs.

[3] The pseudo-code is designed for human reading. It omits parts which are essential for a correct computer implementation.

Bisection algorithm's pseudo-code

1. Decide on maximal allowed deviation (ε_f) of the function from zero and the root precision (ε_x).
2. Make an initial root enclosing bracket for the search, i.e. choose a positive end x_+ and negative end x_- such that $f(x_+) > 0$ and $f(x_-) < 0$. Note that $+$ and $-$ refers to the function sign and not to the relative position of the bracket ends.
3. Begin the searching loop.
4. Calculate the new guess value $x_g = (x_+ + x_-)/2$
5. If $|f(x_g)| \leq \varepsilon_f$ and $|x_+ - x_g| \leq \varepsilon_x$ stop: we have found the root with the desired precision[a].
6. Otherwise, reassign one of the bracket ends: if $f(x_g) > 0$ then $x_+ = x_g$ else $x_- = x_g$.
7. Repeat the searching loop.

[a] Think about why we are using the modified solution convergence expression and not the condition of eq. (1.7).

Figure 1.1 shows the several first iterations of the bisection algorithm. It shows with bold stripes the length of the bracketed region. The points marked as $X_{\pm i}$ are positions of the negative ($-$) and positive ($+$) ends of the root enclosing bracket.

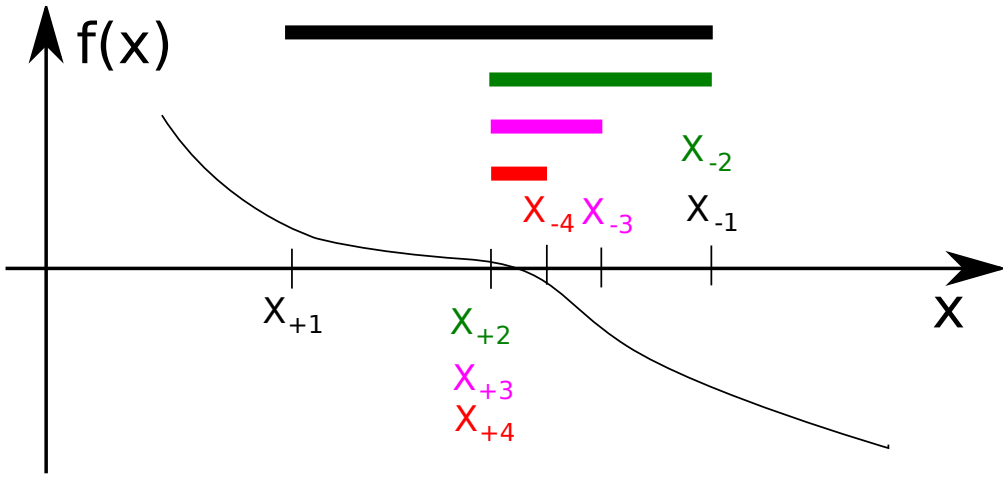

Figure 1.1: The bisection method illustration. $X_{\pm i}$ mark the bracket position on the ith iteration. The root enclosing bracket is indicated by the wide stripe.

The MATLAB implementation of the bisection algorithm is shown below.

Listing 1.1: `bisection.m` (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_root_finding/code/bisection.m)

```
function [xg, fg, N_eval] = bisection(f, xn, xp, eps_f, eps_x)
```

```
% Solves f(x)=0 with bisection method
%
%  Outputs:
%   xg is the root approximation
%   fg is the function evaluated at final guess f(xg)
%   N_eval is the number of function evaluations
%  Inputs:
%   f is the function handle to the desired function,
%   xn and xp are borders of search, i.e. root brackets,
%   eps_f defines maximum deviation of f(x_sol) from 0,
%   eps_x defines maximum deviation from the true solution
%
%  For simplicity reasons, no checks of input validity are done:
%   it is up to user to check that f(xn)<0 and f(xp)>0,
%   and that all required deviations are positive

%% initialization
xg=(xp+xn)/2; % initial root guess
fg=f(xg);     % initial function evaluation
N_eval=1; % We just evaluated the function

%% here we search for root
while ( (abs(xg-xp) > eps_x) || (abs(fg) > eps_f) )
    if (fg>0)
        xp=xg;
    else
        xn=xg;
    end
    xg=(xp+xn)/2;    % update the guessed x value
    fg=f(xg);        % evaluate the function at xg
    N_eval=N_eval+1; % update evaluation counter
end

%% solution is ready
end
```

An interesting exercise for a reader is to see that the while condition is equivalent to the one presented in the step 5 of the bisection's pseudo-code. Also, note the use of the *short-circuiting or* operator represented as `||`. Please have a look at the MATLAB's manual to learn what it does.

1.3.1 Bisection use example and test case

Test the bisection algorithm

For practice let's find the roots of the following equation

$$(x - 10) \times (x - 20) \times (x + 3) = 0 \tag{1.8}$$

Of course, we do not need a fancy computer algorithm to find the solutions: 10, 20, and -3, but knowing the roots in advance allows us to check that we know how to run the code correctly. Also, we will see a typical work flow for the root finding procedure. But most importantly, we can test if the provided bisection code is working correctly: it is always good idea to check new code against known scenarios.

We save MATLAB's implementation of the test function into the file `'function_to_solve.m'`

Listing 1.2: `function_to_solve.m` (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_root_finding/code/function_to_solve.m)

```
function ret=function_to_solve(x)
        ret=(x-10).*(x-20).*(x+3);
end
```

It is a good idea to plot the function to eyeball a potential root enclosing bracket. Have a look at the function in the range from -4 to 2 in fig. 1.2. Any point where the function is negative would be a good choice for a negative end, `xn=-4` satisfies this requirement. Similarly, we have many choices for the positive end of the bracket where the function is above zero. We choose `xp=2` for our test.

One more thing that we need to decide is the required precision of our solution. The higher the precision, the longer the calculation will run. This is probably not a big factor for our test case. However, we really should ensure that we do not ask for the precision beyond the computer's number representation. With 64 bit floats currently used by MATLAB, we definitely should not ask beyond 10^{-16} precision. For this test run, we choose `eps_f=1e-6` and `eps_x=1e-8`.

We find a root of eq. (1.8) with the following code. Notice how we send the handle of the function to solve with `@` operator to `bisection` function.

```
>> eps_x = 1e-8;
>> eps_f = 1e-6;
>> x0 = bisection( @function_to_solve, -4, 2, eps_f, eps_x )
x0 = -3.0000
```

The algorithm seems to yield the exact answer -3. Let's double check that our function is indeed zero at `x0`.

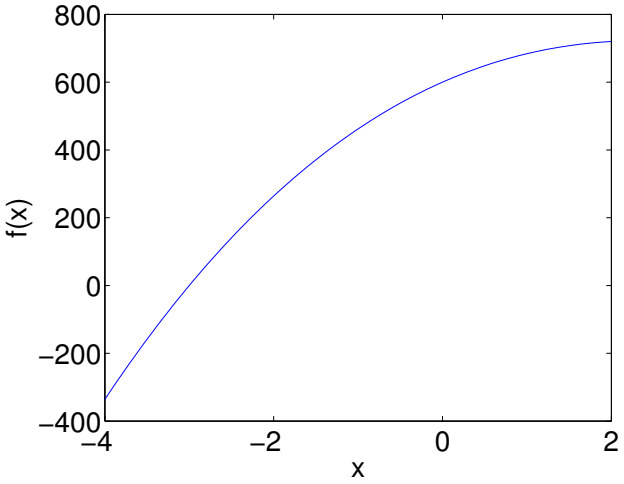

Figure 1.2: Plot of the function to solve $f(x) = (x - 10) \times (x - 20) \times (x + 3)$ in the range from -4 to 2.

```
>> function_to_solve(x0)
ans = 3.0631e-07
```

Wow, the answer is not zero. To explain it, we should recall that we see only 5 significant digits of the solutions, i.e. `-3.0000`, also with `eps_x=1e-6` we requested precision for up to 7 significant digits. So we see the rounded representation of `x0` printed out. Once we plug it back into the function, we see that it satisfies our requested precision for the function zero approximation (`eps_f=1e-6`) with `f(x0) = 3.0631e-07` but not much better. The bottom line: we got what we asked for and everything is as expected.

Notice that we have found only one out of three possible roots. To find the others, we would need to run the `bisection` function with the appropriate root enclosing brackets for each of the roots. The algorithm itself has no capabilities to choose the brackets.

One more example

Now we are ready to find the root of the transcendental eq. (1.5). We will do it without making a separate file for a MATLAB implementation of the function. Instead, we will use anonymous function `f`.

```
>> f = @(x) exp(x) - 10*x;
>> x1 = bisection( f, 2, -2, 1e-6, 1e-6)
x1 =
    0.1118
>> [x2, f2, N] = bisection( f, 2, 10, 1e-6, 1e-6)
```

```
x2 =
    3.5772
f2 =
   2.4292e-07
N =
    27
```

As we can see, eq. (1.5) has 2 roots[4]: `x1=0.1118` and `x2=3.5772`. The output of the second call to `bisection` returns the value of the function at the approximation of the true root, which is `f2=2.4292e-07` and within the required precision `1e-6`. The whole process took only `27` steps or iterations.

1.3.2 Possible improvement of the bisection code

The simplified bisection code is missing validation of input arguments. People make mistakes, typos and all sorts of misuse. Our `bisection` function has no protection against it. For example, if we accidentally swap the positive and negative ends of the bracket in the above example, `bisection` would run forever or at least until we stop the execution. Try to see such misbehavior by executing

```
>> bisection( @function_to_solve, 2, -4, eps_f, eps_x )
```

Once you are tired of waiting, *interrupt the execution* by pressing `Ctrl` and `c` keys together.

Words of wisdom

> "If something can go wrong it will."
>
> Murphy's Law.

We should recall good programming practices from section 4.4 and validate the input arguments[5]. To the very least, we should make sure that `f(xn)<0` and `f(xp)>0`.

1.4 Algorithm convergence

We say that the root finding algorithm has the defined convergence if

$$\lim_{k\to\infty}(x_{k+1}-x_0)=c(x_k-x_0)^m \tag{1.9}$$

[4] It is up to the reader to prove that there are no other roots.
[5] Never expect that a user will put valid inputs.

where x_0 is the true root of the equation, c is some constant, and m is the *order of convergence*. If for an algorithm $m = 1$ then we say that the algorithm converges linearly. The case of $m > 1$ is called superlinear convergence.

It is quite easy to show (by using the size of the bracket for the upper bound estimate of the distance $x_k - x_0$) that the bisection algorithm has linear rate of convergence ($m = 1$) and $c = 1/2$.

Generally, the speed of the algorithm is related to its convergence order: the higher the better. However, other factors may affect the overall speed. For example, there could be too many intermediate calculations or the required memory size could outgrow the available memory of a computer for an otherwise higher convergence algorithm.

If convergence is known, we can estimate how many iterations of the algorithm are required to reach required root-reporting precision. Unfortunately, it is often impossible to define convergence order for a general algorithm.

Example

In the bisection method, the initial bracket size (b_0) decreases by factor 2 on every iteration. Thus, at the step N, the bracket size is $b_N = b_0 \times 2^{-N}$. It should be $< \varepsilon_x$ to reach the required root precision. We need at least the following number of steps to achieve it

$$N \geq \log_2(b_0/\varepsilon_x) \tag{1.10}$$

Conversely, we are getting an extra digit in the root estimate approximately every 3 iterations.

The bisection method is great: it always works and it is simple to implement. But its convergence is quite slow. The following several methods attempt to improve the guess by making some assumptions about the function shape.

1.5 False position (*regula falsi*) method

If the function is smooth and its derivative does not change too much, we can naively approximate our function as a line. We need two points to define a line. We will take negative and positive ends of the bracket as such line-defining points, i.e. the line is the chord joining the function bracket points. The point where this chord crosses zero is our new guess point. We will use it to update the appropriate end of the bracket. The regula falsi method is illustrated in fig. 1.3.

Since the root remains bracketed all the time, the method is guaranteed to converge to the root value. Unfortunately, the convergence of this method can be slower than the bisection's convergence in some situations, which we show in section 1.9.

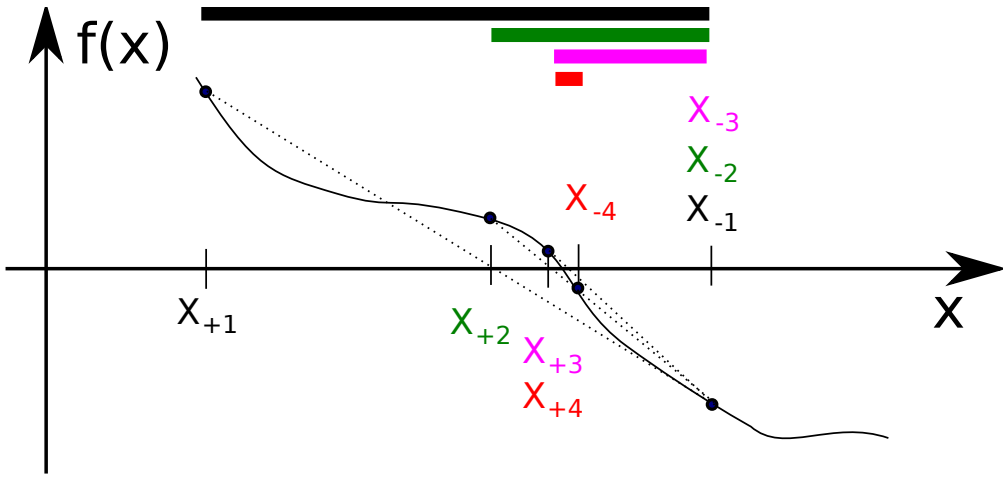

Figure 1.3: The regula falsi method illustration. We can see how the new guess is constructed, and which points are taken as the bracket ends. A wide stripe indicates the bracket size at a given step.

Regula falsi method pseudo-code

1. Choose proper initial bracket for search x_+ and x_- such that $f(x_+) > 0$ and $f(x_-) < 0$.
2. Loop begins.
3. Draw the chord between points $(x_-, f(x_-))$ and $(x_+, f(x_+))$.
4. Make a new guess value at the point of the chord intersection with the 'x' axis

$$x_g = \frac{x_- f(x_+) - x_+ f(x_-)}{f(x_+) - f(x_-)} \tag{1.11}$$

5. If $|f(x_g)| \leq \varepsilon_f$ and the root convergence is reached

$$(|x_g - x_-| \leq \varepsilon_x) \vee (|x_g - x_+| \leq \varepsilon_x) \tag{1.12}$$

   then stop: we found the solution with the desired precision.
6. Otherwise, update the bracket end: if $f(x_g) > 0$ then $x_+ = x_g$ else $x_- = x_g$.
7. Repeat the loop.

Note: the algorithm resembles the bisection pseudo-code except the way of updating x_g and check of the x_g convergence.

1.6 Secant method

The secant method uses the same assumption about the function, i.e. it is smooth and its derivative does not change too widely. Overall, the secant method is very similar to the regula falsi, except we take two arbitrary points to draw a chord. Also we update with the newly guessed value the oldest used point for the chord drawing as illustrated in fig. 1.4. Unlike in the false position method, where one of the ends is sometimes (or even never) updated, the ends are always updated, which makes the convergence of the secant algorithm superlinear: the order of convergence m is equal to the *golden ratio* [1], i.e. $m = (1 + \sqrt{5})/2 \approx 1.618\ldots$. Unfortunately, because the root is not bracketed, **the convergence is not guaranteed**.

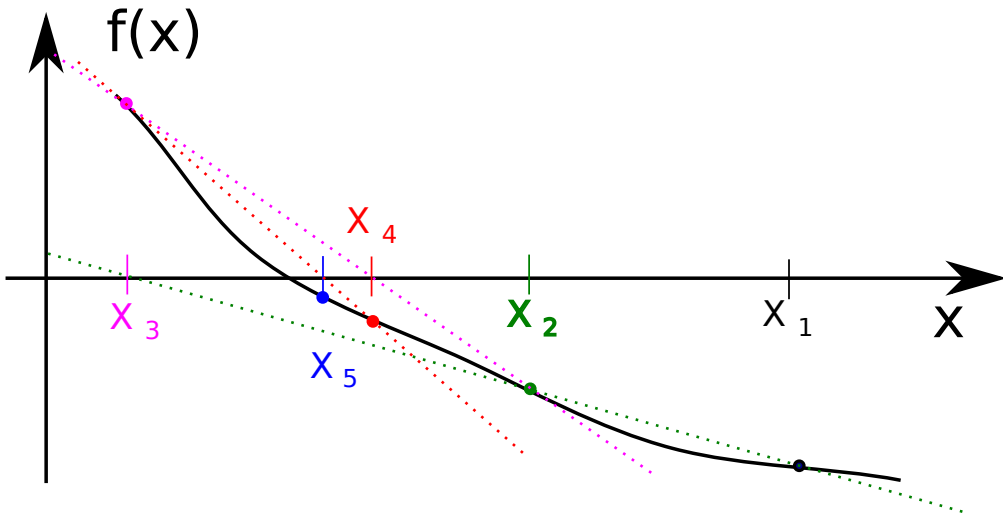

Figure 1.4: The secant method illustration

Outline of the secant algorithm

1. Choose two arbitrary starting points x_1 and x_2.
2. Loop begins.
3. Calculate next guess according to the following iterative formula

$$x_{i+2} = x_{i+1} - f(x_{i+1})\frac{x_{i+1} - x_i}{f(x_{i+1}) - f(x_i)} \tag{1.13}$$

4. Throw away x_i point.
5. Repeat the loop till the required precision is reached.

1.7 Newton-Raphson

The Newton-Raphson method also approximates the function with a line. In this case, we draw a line through a guess point $(x_g, f(x_g))$ and make the line's slope equal to the derivative of the function itself. Then we find where this line crosses 'x' axis and take this point as the next guess. The process is illustrated in fig. 1.5. The process converges quadratically, i.e. $m = 2$, which means that we double the number of significant figures with every iteration [1], although the calculation of the derivative could be as time consuming as calculating the function itself (see for example numerical derivative algorithms in chapter 7), i.e. one iteration with Newton-Raphson is equivalent to 2 iterations with some other algorithm. So, the order of convergence is actually $m = \sqrt{2}$. The downside is that convergence to the root is not guaranteed and is quite sensitive to the starting point choice.

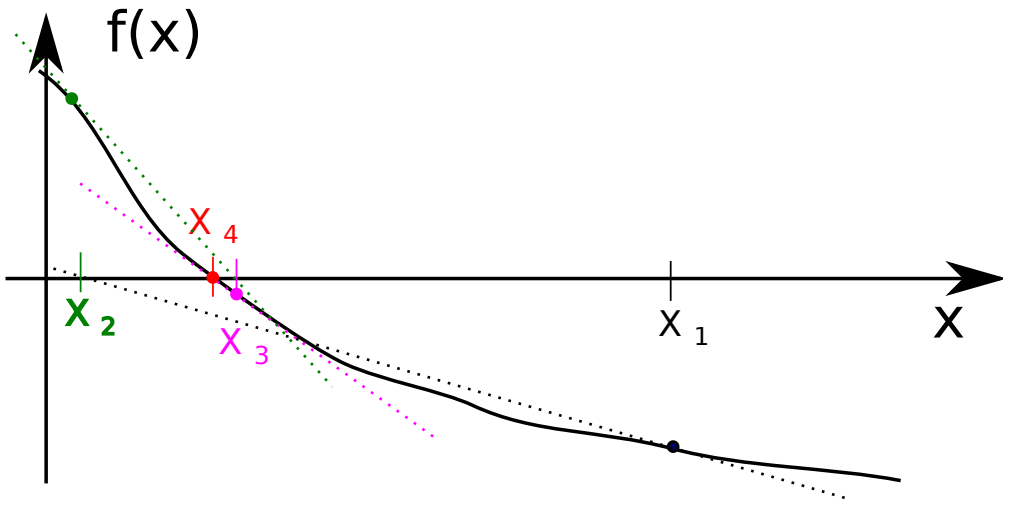

Figure 1.5: The Newton-Raphson method illustration

Outline of the Newton-Raphson algorithm

1. Choose an arbitrary starting point x_1.
2. Loop begins.
3. Calculate next guess according to the following iterative formula

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \tag{1.14}$$

4. Repeat the loop till the required precision is reached.

One has a choice how to calculate f'. It can be done analytically (the preferred method, but it requires additional programming of the separate derivative function) or numerically (for details see chapter 7).

Below you can see the simplified implementation of the Newton-Raphson algorithm without the guess convergence test (this is left as exercise for the reader).

Listing 1.3: NewtonRaphson.m (available at http://physics.wm.edu/programming_with_MATLAB_book/ch_root_finding/code/NewtonRaphson.m)

```
function [x_sol, f_at_x_sol, N_iterations] = NewtonRaphson(f, xguess, eps_f,
    df_handle)
%   Finds the root of equation f(x)=0 with the Newton-Raphson algorithm
%   f - the function to solve handle
%   xguess - initial guess (starting point)
%   eps_f - desired precision for f(x_sol)
%   df_handle - handle to the derivative of the function f(x)

%  We need to sanitize the inputs but this is skipped for simplicity

        N_iterations=0;  % initialization of the counter
        fg=f(xguess);    % value of function at guess point

        while( (abs(fg)>eps_f) ) % The xguess convergence check is not
           implemented
                xguess=xguess - fg/df_handle(xguess); % evaluate new guess
                fg=f(xguess);
                N_iterations=N_iterations+1;
        end
        x_sol=xguess;
        f_at_x_sol=fg;
end
```

Using Newton-Raphson algorithm with the analytical derivative

Let's see how to call the Newton-Raphson if the analytical derivative is available. We will solve

$$f(x) = (x-2) \times (x-3) \tag{1.15}$$

It is easy to see that the derivative of the above $f(x)$ is

$$f'(x) = 2x - 5 \tag{1.16}$$

To find a root, we should first implement code for f and f'

```
>> f = @(x) (x-2).*(x-3);
>> dfdx = @(x) 2*x - 5;
```

Now we are ready to execute our call to `NewtonRaphson`

```
>> xguess = 5;
>> eps_f=1e-8;
>> [x_1, f_at_x_sol, N_iterations] = NewtonRaphson(f, xguess, eps_f, dfdx)
x1 =
    3.0000
f_at_x_sol =
   5.3721e-12
N_iterations =
     6
```

In only 6 iterations, we find only the root $x_1 = 3$ out of two possible solutions. Finding all the roots is not a trivial task. But if we provide a guess closer to another root, the algorithm will converge to it.

```
>> x2 = NewtonRaphson(f, 1, eps_f, dfdx)
x2 =
    2.0000
```

As expected, we find the second root $x_2 = 2$. Strictly speaking, we find the approximation of the root, since the `x_sol2` is not exactly 2:

```
>> 2-x_sol2
ans =
   2.3283e-10
```

but this is what we expect with numerical algorithms.

Using Newton-Raphson algorithm with the numerical derivative

We will look for a root of the following equation

$$g(x) = (x - 3) \times (x - 4) \times (x + 23) \times (x - 34) \times \cos(x) \tag{1.17}$$

In this case we will resort to the numerical derivative, since the $g'(x)$ is too cumbersome. It is likely to make an error during the analytical derivative derivation.

Firstly, we implement the general forward difference formula (see why this is not the best method in section 7.3).

```
>> dfdx = @(x, f, h) (f(x+h)-f(x))/h;
```

Secondly, we implement $g(x)$

```
>> g = @(x) (x-3).*(x-4).*(x+23).*(x-34).*cos(x);
```

Now, we are ready to make the **specific** numerical derivative implementation of $g'(x)$. We choose step `h=1e-6`

```
>> dgdx = @(x) dfdx(x, g, 1e-6);
```

Finally, we search for a root (pay attention: we use here `g` and `dgdx`)

```
xguess = 1.4; eps_f=1e-6; x_sol = NewtonRaphson(g, xguess, eps_f, dgdx)
x_sol =
    1.5708
```

Note that $\pi/2 \approx 1.5708$. The algorithm converged to the root which makes $\cos(x) = 0$ and, consequently, $g(x) = 0$.

1.8 Ridders' method

As the name hints, this method was proposed by Ridders [2]. In this method, we approximate the function from eq. (1.8) with a nonlinear one to take its curvature into account, thus making a better approximation. The trick is in a special form of the approximation function which is the product of two equations

$$f(x) = g(x)e^{-C(x-x_r)} \tag{1.18}$$

where $g(x) = a + bx$ is a linear function, C is some constant, and x_r is an arbitrary reference point. The advantage of this form is that if $g(x_0) = 0$, then $f(x_0) = 0$, but once we know coefficients a and b, the x where $g(x) = 0$ is trivial to find.

We might expect a faster convergence of this method, since we do a better approximation of the function $f(x)$. But there is a price to pay: the algorithm is a bit more complex and we need an additional calculation of the function beyond the bracketing points, since we have three unknowns a, b, and C to calculate (we have freedom of choice for x_r).

If we choose the additional point location $x_3 = (x_1 + x_2)/2$, then the position of the guess point x_4 is quite straight forward to calculate with a proper bracket points x_1 and x_2.

$$x_4 = x_3 + \operatorname{sign}(f_1 - f_2)\frac{f_3}{\sqrt{f_3^2 - f_1 f_2}}(x_3 - x_1) \tag{1.19}$$

here $f_i = f(x_i)$, and $\operatorname{sign}(x)$ stands for the sign of the function's argument: $\operatorname{sign}(x) = +1$ when $x > 0$, -1 when $x < 0$, and 0 when $x = 0$. The search for the guess point is illustrated in fig. 1.6 where $x_r = x_3$.

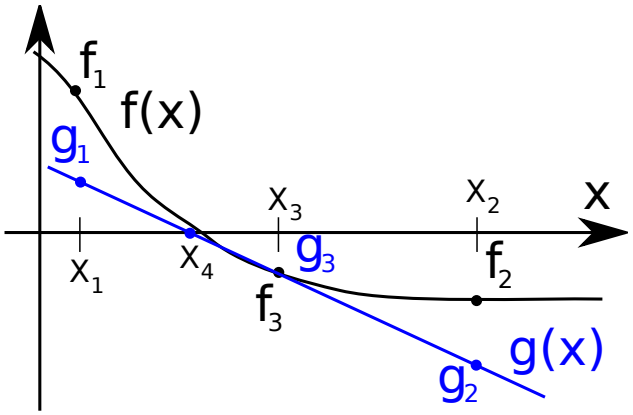

Figure 1.6: The Ridders' method illustration. The reference point position $x_r = x_3$ and $x_3 = (x_1 + x_2)/2$.

Ridders algorithm outline

1. Find **proper** bracketing points for the root x_1 and x_2. It is irrelevant which is positive end and which is negative, but the function must have different signs at these points, i.e. $f(x_1) \times f(x_2) < 0$.
2. Loop begins.
3. Find the midpoint $x_3 = (x_1 + x_2)/2$
4. Find a new approximation for the root

$$x_4 = x_3 + \text{sign}(f_1 - f_2)\frac{f_3}{\sqrt{f_3^2 - f_1 f_2}}(x_3 - x_1) \tag{1.20}$$

   where $f_1 = f(x_1), f_2 = f(x_2), f_3 = f(x_3)$.
5. If x_4 satisfies the required precision and the convergence condition is reached, then stop.
6. Rebracket the root, i.e. assign new x_1 and x_2, using old values
   - one end of the bracket is x_4 and $f_4 = f(x_4)$
   - the other is whichever of (x_1, x_2, x_3) is closer to x_4 **and provides the proper bracket**.
7. Repeat the loop.

In Ridders' algorithm, the root is always properly bracketed; thus the algorithm always converges, and x_4 is always guaranteed to be inside the initial bracket. Overall, the convergence of the algorithm is quadratic per cycle ($m = 2$). However, it requires evaluation of the $f(x)$ twice for f_3 and f_4; thus it is actually $m = \sqrt{2}$ [2].

1.9 Root finding algorithms gotchas

The root bracketing algorithms are bulletproof and always converge, but convergence of the false position algorithm could be slow. Normally, it outperforms the bisection algorithm, but for some functions that is not the case. See for example the situation depicted in fig. 1.7. In this example the bracket shrinks by small amount on every iteration. The convergence would be even worse if the long horizontal tail of the function runs closer to the 'x' axis.

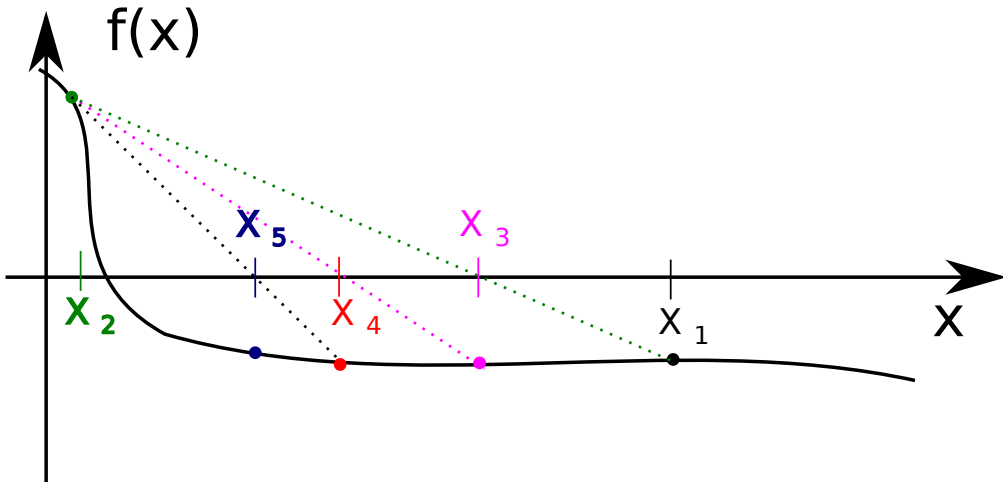

Figure 1.7: The false position slow convergence pitfall. Think what would happen if the long horizontal tail of the function lies even closer to the x-axis.

The non bracketing algorithms, such as Newton-Raphson and secant, usually converge faster than their bracketing counterparts. However, their convergence is not guaranteed! In fact, they may even diverge and run away from the root. Have a look at fig. 1.8 where a pitfall of the Newton-Raphson method is shown. In just three iterations, the guess point moved far away from the root and our initial guess.

Words of wisdom

There is no *silver bullet* algorithm which would work in all possible cases. We should carefully study the function for which root is searched, and see if all relevant requirements of an algorithm are satisfied. When unsure, sacrifice speed and choose a more robust but slower bracketing algorithm.

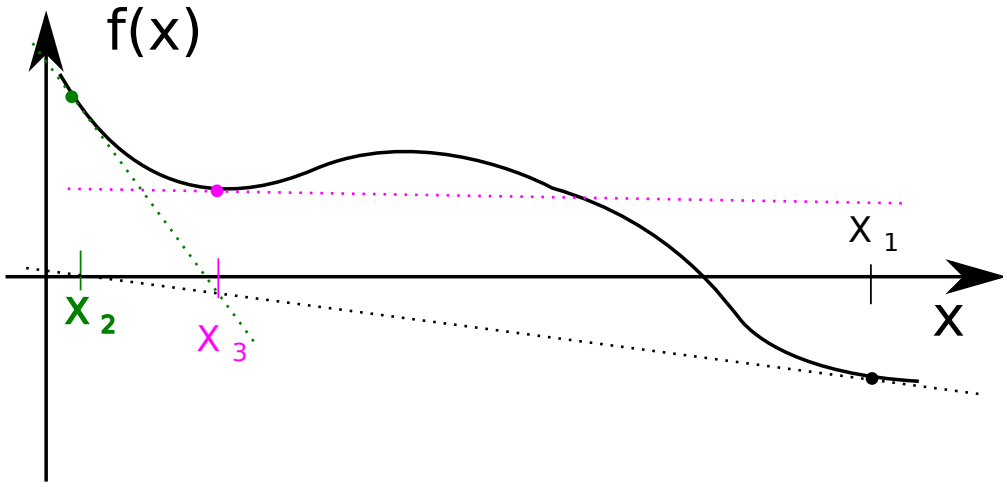

Figure 1.8: The Newton-Raphson method pitfall: the x_4 guess is located far to the right and way farther than original guess x_1

1.10 Root finding algorithms summary

By no means we have considered all root finding algorithms. We just covered a small subset. If you are interested in seeing more information, you may read [1].

Below we present a short summary of root bracketing and non bracketing algorithms.

Root bracketing algorithms
- bisection
- false position
- Ridders'

Pro
- robust i.e. always converge.

Contra
- usually slower convergence
- require initial bracketing

Non bracketing algorithms
- Newton-Raphson
- secant

Pro
- faster
- no need to bracket (just give a **reasonable** starting point)

Contra
- **may not converge**

1.11 MATLAB's root finding built-in command

MATLAB uses `fzero` to find the root of an equation. Deep inside, `fzero` uses combination of bisection, secant, and inverse quadratic interpolation methods. The built-in `fzero` has quite a few options, but in the simplest form we can call it as shown below to solve eq. (1.5).

To search for the root by providing only the starting point

```
>> f = @(x) exp(x) - 10*x;
```

```
>> fzero(f, 10)
ans =
    3.5772
```

We have no control which of possible roots will be found this way. We can provide the proper bracket within which we want to find a root

```
>> f = @(x) exp(x) - 10*x;
>> fzero(f, [-2,2])
ans =
    0.1118
```

In the above case, the bracket spans from -2 to 2. As you can see, we find the same roots as with our `bisection` implementation discussed in section 1.3.1.

1.12 Self-Study

General requirements:

1. Test your implementation with at least $f(x) = \exp(x) - 5$ and the initial bracket [0,3], but do not limit yourself to only this case.
2. If the initial bracket is not applicable (for example, in the Newton-Raphson algorithm) use the right end of the test bracket as the starting point of the algorithm.
3. All methods should be tested for the following parameters `eps_f=1e-8` and `eps_x=1e-10`.

Problem 1: Write proper implementation of the false position algorithm. Define your function as

```
function [x_sol, f_at_x_sol, N_iterations] = regula_falsi(f, xn, xp, eps_f, eps_x)
```

Problem 2: Write a proper implementation of the secant algorithm. Define your function as

```
function [x_sol, f_at_x_sol, N_iterations] = secant(f, x1, x2, eps_f, eps_x)
```

Problem 3: Write a proper implementation of the Newton-Raphson algorithm. Define your function as

```
function [x_sol, f_at_x_sol, N_iterations] = NewtonRaphson(f, xstart, eps_f, eps_x, df_handle)
```

. Note that `df_handle` is a function handle to calculate derivative of the function `f`; it could be either analytical representation of $f'(x)$ or its numerical estimate via the central difference formula.

Problem 4: Write a proper implementation of Ridders' algorithm. Define your function as

```
function [x_sol, f_at_x_sol, N_iterations] = Ridders(f, x1, x2, eps_f, eps_x)
```

Problem 5: For each of the root finding implementation of yours find roots of the following two functions

(a) $f1(x) = \cos(x) - x$ with the 'x' initial bracket [0,1]

(b) $f2(x) = \tanh(x - \pi)$ with the 'x' initial bracket [-10,10]

Make a comparison table for the above algorithms with following rows

(a) Method name

(b) root of $f1(x)$

(c) initial bracket or starting value used for $f1$

(d) Number of iterations to solve $f1$

(e) root of $f2(x)$

(f) initial bracket or starting value used for $f2$

(g) Number of iterations to solve $f2$

If an algorithm diverges with the suggested initial bracket: indicate so, appropriately modify the bracket, and show the modified bracket in the above table as well. Make your conclusions about speed and robustness of the methods.

Bibliography

[1] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007.

[2] C. Ridders. A new algorithm for computing a single root of a real continuous function. *Circuits and Systems, IEEE Transactions on*, 26(11):979–980, Nov 1979.