Nonlinear Equations: Bisection

Math 131: Numerical Analysis

J.C. Meza

2/6/24

Nonlinear equations

Suppose that we have a scalar, nonlinear equation $f : \mathbb{R}^1 \to \mathbb{R}^1$ on an interval $[a, b]$

If $x^* \in \mathbb{R}$ is such that

$$f(x^*) = 0, \tag{1}$$

then we will call x^* a *zero* or *root* of f.

- Standard assumption is that $f(x)$ is continuous on the closed interval $[a, b]$.

Some Examples

1. $f(x) = 6x^2 - 7x + 2 = (2x - 1)(3x - 2)$
2. $f(x) = 2^{x^2} - 10x + 1$
3. $f(x) = \cosh(\sqrt{x^2 + 1} - e^x) + \log|\sin(x)|$
4. $f(x) = \blacksquare \leftarrow$ some black box

Relation to optimization

Many of the methods that we study in this section are also applicable to the problem of optimization. For example:

$$\min f(x)$$

- A minimum will occur at points where $f'(x) = 0$.
- Special care must be taken that the method approaches a minimum and not a maximum or an inflection point, but this is usually not hard to handle.

Notes on general setting

- Except for some special cases, nonlinear equations will not have a closed form solution, so we are naturally led to developing alternative means for solving Equation 1.
- We could have just as easily defined the problem in higher dimensions, i.e. $f : \mathbb{R}^n \to \mathbb{R}^p$, for $n > 1, p > 1$, and n not necessarily equal to p.
- This would most likely be the case for a real-world problem, but while many of the ideas (and algorithms) can be generalized to higher dimensions, the concepts are easier to understand in one dimension to begin with.
- We note in passing that if $n = p$, then this is an example of solving a ***nonlinear system of equations***. If $n \neq p$, then this is an example of a ***nonlinear least squares*** problem.

General Iterative Framework

Most nonlinear equations cannot be solved analytically and as a result most approaches involve some form of iteration, which can be described at a high-level as:

1. First guess a solution,
2. Check to see how accurate it is, and if not satisfied,
3. Update the guess and try again, i.e. go back to step 1

This is the essence of an ***iterative method.***

Challenges

- Using an iterative technique to find roots of equations can be tricky.
- The initial guess can sometimes be important, and if not chosen properly can lead to slow convergence or even non-convergence.
- How one updates the guess is also important. Without some theory behind the updating scheme, an iterative method is nothing more than trial and error.
- Finally, there is always the question as to when to terminate an iterative method, in other words when is an estimate of the root "good enough"?

Note

For all of these reasons, having an algorithm based on sound theoretical foundations is of fundamental importance.

Overview

In the next few sections, we discuss some of the more popular iterative methods for finding the roots of a nonlinear equation and provide some general guidance for when to use them. In particular, we will study:

1. bisection method,
2. Newton's method,
3. secant method and
4. fixed-point iterations

Bisection (Quick Summary)

- One of the simplest methods for solving a nonlinear equation is known as the bisection method.
- The main advantage is the robustness of the method - if the method is applicable to the problem, it is guaranteed to find a solution.
- On the other hand, the method can often take far more iterations than some of the other methods we will discuss.

Let's first describe the general method visually

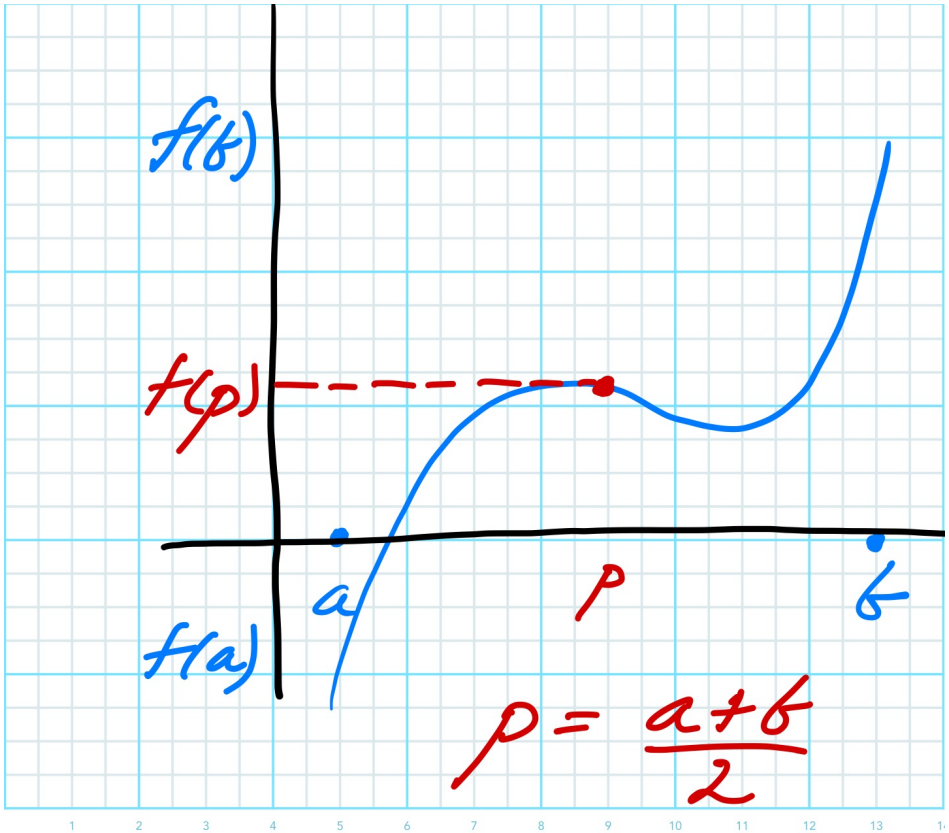

Figure 1: Bisection Method

Idea

- Suppose that we happen to know two points where the function is of opposite sign as in Figure 1.
- Using the Intermediate Value Theorem (IVT) we also know that if the function is continuous, then the function must take every value in between the two values:

Important

In particular, ***it must be equal to 0 somewhere in the interval***.

Mathematically

If $f(x) \in C[a,b]$ and $f(a) \cdot f(b) < 0$, then there exists an $x^* \in [a,b]$ such that $f(x^*) = 0$. This leads us to the following general procedure:

1. Start with an interval $[a,b]$ such that $f(a) \cdot f(b) < 0$.
2. Cut the interval in half (bisect).
3. Evaluate the function f at the midpoint of the interval.
4. Choose whichever sub-interval contains a sign change.
5. Repeat as necessary.

Start by writing this in pseudocode:

Given a_0, b_0 s.t. $f(a_0)f(b_0) < 0$
for k=0, . . . , maxiter **do**
    $mid \leftarrow (a_k + b_k)/2$
    if $f(a_k)f(mid) < 0$ **then**
        $a_{k+1} \leftarrow a_k; b_{k+1} \leftarrow mid$
    else
        $a_{k+1} \leftarrow mid; b_{k+1} \leftarrow b_k$
    end if
end for

Figure 2: Bisection Algorithm

Example

Consider $f(x) = x^3 + 4x^2 - 10 = 0$ on $[1, 2]$ where $x^* = 1.365$. We should first check that the function has opposite signs on the given interval:

$$f(1) = 1 + 4 - 10 = -5$$
$$f(2) = 8 + 16 - 10 = 14$$
$$f(1) \cdot f(2) < 0 \qquad\qquad YES!$$

1. Compute $p = \frac{a+b}{2} = \frac{1+2}{2} = 1.5$
2. Evaluate $f(1.5) = 3.375 + 9 - 10 = 2.375$
3. $f(a) \cdot f(p) = f(1) \cdot f(1.5) < 0$ so we choose left interval and set $b = p$
4. Repeat

After 13 iterations $p = 1.365112305$, $f(p) = -0.00194$, and $|b_{14} - a_{14}| = 0.000122070$.

Let's take a look at how this pseudocode could be implemented in python.

```
import numpy as np
def bisect(a0, b0, ftol, maxiter=30):
    ak = a0
    bk = b0
    iters = 0
    for k in range(1, maxiter):
        mid = (ak + bk)/2
        fmid = fx(mid)
        if (fx(ak)*fmid < 0):
          bk = mid
        else:
          ak = mid
        if (np.abs(fmid) < ftol):
            iters = k
            break
```

Let's define a function and call the bisection algorithm. For this example, let's use the function:

$$f(x) = (x - 1.5)^3 - x + 2$$

on the interval $I = [a, b] = [0, 2.5]$.

```
# Example function

def fx(x):
  fvalue = (x-1.5)**3 - x + 2
  return fvalue
```

Call the bisection algorithm.

For initial values we'll use an initial starting guess of $a_0 = 0.0, b_0 = 2.5$ and a function tolerance of $ftol = 10^{-6}$

```
#Initialize
a = 0.0
b = 2.5
ftol = 1.e-6

# Check assumptions
fa  = fx(a)
fb  = fx(b)
print ("fa = %f, fb = %f" %(fa, fb))

# Call Bisection function
xstar = bisect(a, b, ftol)
```

When should we stop?

- Before proceeding further, we should discuss when and how to terminate an iterative algorithm. This decision is one of great importance in real-world applications because many of the problems are expensive or time consuming.
- Both the expense and time are usually a result of the complexity of the function being evaluated.
- In some cases it could take hours if not days of computer time to yield one function evaluation. In these cases, it is not unusual that a scientist or an engineer will decide to terminate an algorithm based simply on how much computer time they are willing to use.

Some options

- There are numerous possibilities for convergence criteria, each with pros and cons.
- The most obvious would be to check to see how close we are to our desired solution, but in general this would be impossible since we don't know what the solution is ahead of time (except for academic exercises).
- On the other hand, it is not unreasonable to assume that we might have some sort of bound, for example in some cases, the solution might be known to be positive or have a minimum/maximum value.

First option

- Suppose that a general iterative algorithm has produced a sequence of iterates starting with an initial guess:

$$x_0,\ x_1,\ x_2,\ \ldots,\ x_k.$$

- One logical approach is to take a look at the magnitude of $f(x_k)$ and check to see how close we are to zero:

$$|f(x_k)| < atol$$

Second option

- Another frequent approach is to stop an iteration when "sufficient" progress has been made. In other words, an engineer is simply interested in reducing the initial function value by some fraction, i.e.

$$\frac{|f(x_k)|}{|f(x_0)|} < ftol$$

Other approaches

- In general, these are good approaches, but there are cases for which progress towards the solution may be slow and little is to be gained from each new iteration.
- In this case, we may decide to stop if we believe we are not making sufficient progress towards a solution.
- This could be construed, for example, if the difference between successive iterates (step size) becomes small:

$$|s_k| = |x_{k+1} - x_k| < stol$$

Relative error

- Here, we should note that it might also make sense to check that the relative step size is small:

$$\frac{|x_{k+1} - x_k|}{|x_{k+1}|} < rtol$$

- In practice, many algorithms will employ some combination of these (and sometimes others).
- Experience and knowledge of the specific problem are usually needed to ensure that we don't stop too early or waste time iterating for too long.

Convergence of Bisection method

- One of the first questions one should ask about any iterative method is when and under what conditions do we expect that it might converge to a solution.
- In the case of the bisection method, it turns out that the only condition we need to have is that the function $f(x)$ is continuous, given that the two initial points yield functions values with opposite signs, i.e. IVT.
- Moreover, if these conditions hold, we can prove that the error bound between the solution and the iterates x_k can be given by:

$$|x^* - x_k| \leq \frac{b-a}{2^k}, \quad k = 0, 1, \ldots. \tag{2}$$

Proof

Notice that for $k = 0$ all we're really saying is that the root must lie within the given interval $[a_0, b_0]$. At the next step, the interval and hence the error is cut in half so that

$$|x^* - x_1| \leq \frac{b_0 - a_0}{2}, \quad (k = 1)$$

At each iteration, the interval is cut in half by construction, so that at the kth iteration we get our desired result.

As constructed then, ***the bisection method cannot fail***. This type of method is known as a ***robust*** algorithm.

Error Bounds

Additionally, we can use the error bound to estimate the number of iterations required to achieve a certain accuracy, ϵ.

Using Equation 2 we want

$$|x^* - x_k| \leq \frac{b-a}{2^k} \leq \epsilon.$$

Taking logs of both side we have

$$\log(b-a) - \log 2^k \leq \log \epsilon,$$

or rearranging

$$\log(b-a) - \log \epsilon \leq k \log 2.$$

If we would like to have the error be less than ϵ then solving for the number of iterations gives us

$$k \geq \frac{\log[(b-a)/\epsilon]}{\log 2}. \tag{3}$$

Example

Let's set $\epsilon = 0.001$ and $a = 1, b = 2$.

- Then solving for k, we have

$$k \geq \frac{\log[1/0.001]}{\log 2} = \frac{3}{0.301} \approx 9.97,$$

- so $k = 10$ iterations should suffice to achieve an error tolerance of $\epsilon = 0.001$.

Summary

- Bisection is a robust algorithm for finding the zero of a nonlinear function.
- Need to have 2 initial points where function value is of opposite sign.
- It is guaranteed to converge.

Table 1: **Bisection Method Summary**

Advantages	Disadvantages
Always converges, i.e. ***robust*** algorithm	Need to provide a specific interval, with 2 points where function is of opposite sign
Error bound easily derived and can be used to estimate number of iterations need to achieve a desired tolerance	Slow convergence - error only decreases by 1/2 at each iteration
Can be used to start other methods	Not clear how to generalize to higher dimensions