MATH3511/6111: Scientific Computing

02. Rootfinding and Nonlinear Equations

Lindon Roberts

Semester 1, 2022

Office: Hanna Neumann Building #145, Room 4.87
Email: lindon.roberts@anu.edu.au
Based on lecture notes written by S. Roberts, L. Stals, Q. Jin, M. Hegland, K. Duru.

Australian
National
University

Solving Equations

In this section, we look at the general problem of 'rootfinding'.

Rootfinding

Given a function $f : \mathbb{R} \to \mathbb{R}$, find $x^* \in \mathbb{R}$ such that

$$f(x^*) = 0.$$

One way rootfinding arises is in finding solutions to an equation. For example, if we want to solve

$$e^x = \frac{1}{x-1},$$

then we can equivalently try to solve the rootfinding problem

$$e^x - \frac{1}{x-1} = 0.$$

(this problem has one solution, $x^* \approx 1.278$)

Solving Equations

Some rootfinding problems are easy:

- If $f(x) = ax + b$ and $a \neq 0$, then f has one root, $x^* = -b/a$
- If $f(x) = x^2 - c$, then:
  - If $c < 0$ then f has no roots
  - If $c = 0$ then f has one root, $x^* = 0$
  - If $c > 0$ then f has two roots, $x^* = \pm\sqrt{c}$

(actually, computing $\sqrt{c}$ is not necessarily easy, and rootfinding methods give us a way of calculating square roots!)

We don't want a case-by-case approach: this is cumbersome, and not every equation has a closed-form solution.

We want a way of finding a solution that works for any function.

Existence of Solutions

Before writing an algorithm, we should first check if a solution exists.

Theorem (Intermediate Value Theorem)

If $f : \mathbb{R} \to \mathbb{R}$ is continuous on the interval $[a, b]$ and $f(a)$ and $f(b)$ have different signs (one is positive, one is negative), then there exists a point $x^* \in [a, b]$ such that $f(x^*) = 0$.

It is easy to check if a solution exists, but not to find it!

`https://www.reddit.com/r/Jokes/comments/7o3cmk/in_a_hotel_a_engineer_a_physicist_and_a/`

However, it's a good start: no point searching for something that doesn't exist.

Range Reduction

In general, it is useful to not work with numbers that are too large or too close to zero. This is because computers store numbers to a fixed level of accuracy (and in binary).

We will talk about this more later, but if you are working with numbers that have very different orders of magnitude, you can get large errors in your calculations. If we can, we should make sure our problem is set up for this situation.

Example

Suppose we want to calculate $\sqrt{c}$ by solving $x^2 - c = 0$ for some (large) $c > 0$.

We can always write $c = d \times 4^n$ for some integer n and some $d \in [1, 4)$.

Then we can calculate $\sqrt{d}$ by solving $x^2 - d = 0$, and our final solution is $\sqrt{c} = \sqrt{d} \times 2^n$.

So, we only need to build an algorithm that calculates square roots of numbers in $[1, 4)$, and use scaling to calculate other square roots.

Note: multiplying by powers of two is easy when numbers are stored in binary

Iterative Methods

Many numerical algorithms are iterative:

1. Start with an initial guess x_0
2. Use x_0 to calculate a new (better) solution estimate x_1
3. Use x_1 (and maybe x_0) to calculate a better estimate x_2
4. Use x_2 (and maybe x_0 and x_1) to calculate a better estimate x_3
5. ...

We want algorithms that produce a sequence $x_0, x_1, x_2, x_3, \ldots$ which approaches the true solution ("convergence"). Generally,

- The closer the initial guess is to the solution, the faster the sequence will converge
  - How to pick an initial guess? Depends on the situation — use whatever information you have!
- One algorithm is "better" than another if it produces a sequence converging to the solution faster

Example Problem

Let's try to calculate $\sqrt{3}$ by solving $f(x) = 0$ for $f(x) = x^2 - 3$.

Since $f(1) = -2$ and $f(2) = 1$, the IVT guarantees that f has a root in the interval $[1, 2]$.

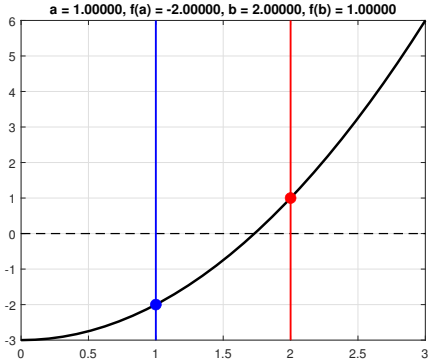


Figure 1. Plot of rootfinding problem $x^2 - 3 = 0$.

Bisection Method

The simplest rootfinding method is called the bisection method. The key idea of the bisection method is the following:

- Suppose we know $f(a)$ and $f(b)$ have different signs
- Consider the midpoint $m = (a + b)/2$
- If $f(m) = 0$, we have found a solution
- Otherwise, $f(m)$ has a different sign to either $f(a)$ or $f(b)$
- This means that either a solution exists in $[a, m]$ or $[m, b]$
  - For example, if $f(a) < 0$ and $f(b) > 0$, then if $f(m) > 0$, we know a solution exists in $[a, m]$
- This gives a new, smaller interval where a solution exists

If we repeat this process, the size of the interval shrinks to zero (so we have a very good idea of where the solution is).

Bisection Method: Example

Using $f(x) = x^2 - 3$, currently $a = 1$ and $b = 2$. Note $f(a) < 0$ and $f(b) > 0$.

Then $m = (a + b)/2 = 1.5$ and so $f(m) = -0.75 < 0$. New interval is $[m, b]$.

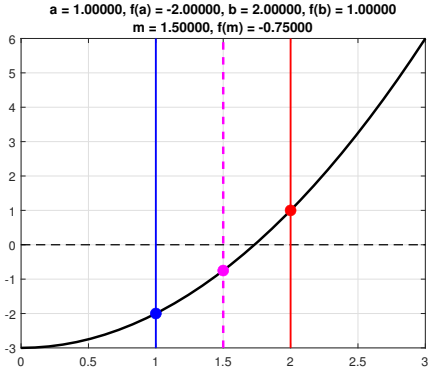


Figure 2. Bisection Method for solving $x^2 - 3 = 0$ after 1 iteration.

Bisection Method: Example

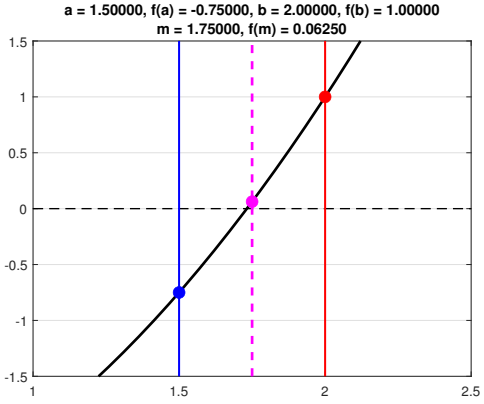


Figure 3. After 2 iterations.

Bisection Method: Example

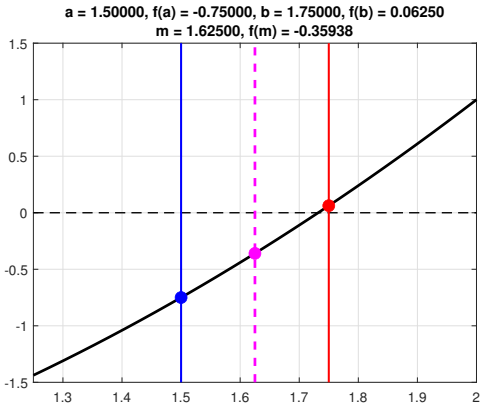


Figure 4. After 3 iterations.

Bisection Method: Example

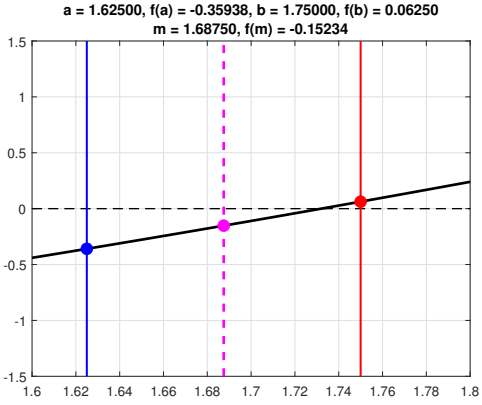


Figure 5. After 4 iterations.

Bisection Method: Example

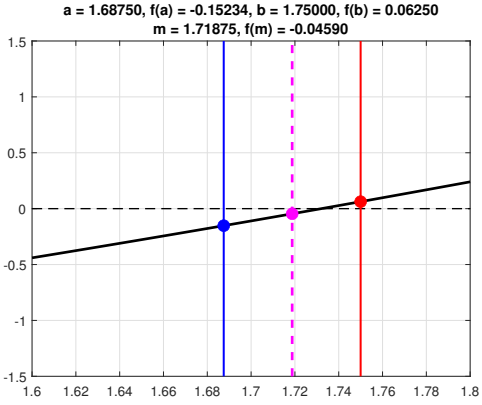


Figure 6. After 5 iterations.

Bisection Method: Example

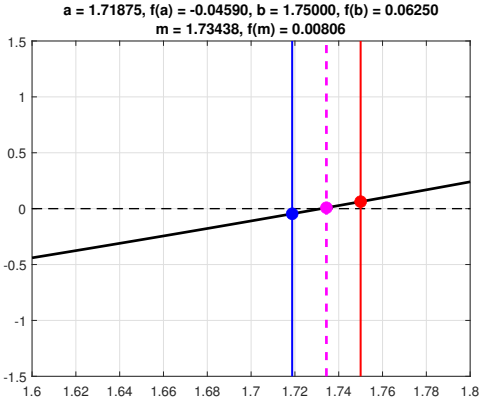


Figure 7. After 6 iterations.

Bisection Method: Example

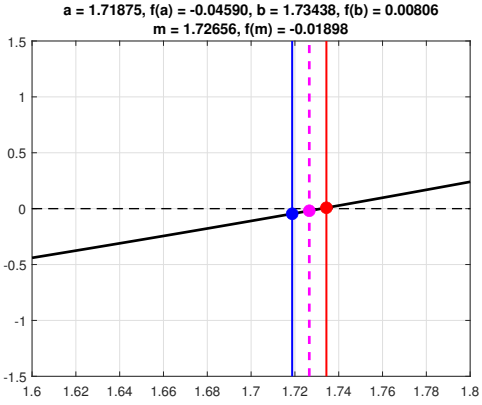


Figure 8. After 7 iterations.

Bisection Method: Example

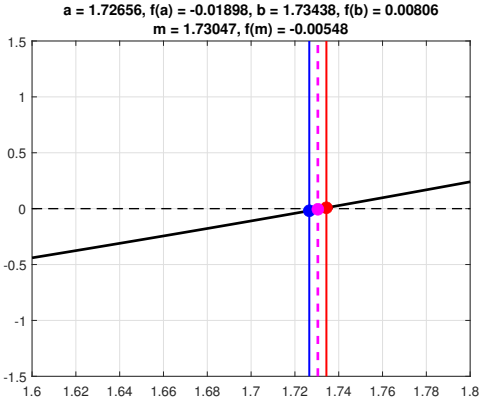


Figure 9. After 8 iterations.

Bisection Method: Example

Step	a	b	m	f(m)
1	1.00000	2.00000	1.50000	-0.75000000
2	1.50000	2.00000	1.75000	0.06250000
3	1.50000	1.75000	1.62500	-0.35937500
4	1.62500	1.75000	1.68750	-0.15234375
5	1.68750	1.75000	1.71875	-0.04589844
6	1.71875	1.75000	1.73438	0.00805664
7	1.71875	1.73438	1.72656	-0.01898193
8	1.72656	1.73438	1.73047	-0.00547791
9	1.73047	1.73438	1.73242	0.00128555
10	1.73047	1.73242	1.73145	-0.00209713
11	1.73145	1.73242	1.73193	-0.00040603
12	1.73193	1.73242	1.73218	0.00043970
13	1.73193	1.73218	1.73206	0.00001682

Table. Calculation of $\sqrt{3} = 1.73205080756888$ using the Bisection Method.

Bisection Method: First Version

Let's formalise this into a precise algorithm which works for any rootfinding problem $f(x) = 0$.

Algorithm Bisection Method for solving $f(x) = 0$ (first version).

Input: a and b that surround a root (i.e. $f(a)$ and $f(b)$ are of opposite sign), tolerance $\epsilon > 0$.

1: Evaluate $f(a)$ and $f(b)$.
2: Compute the midpoint $m = (a + b)/2$ and evaluate $f(m)$.
3: **if** $|f(m)| < \epsilon$ **then** ← *$f(m) \approx 0$, and m is at most $|b - a|/2$ from the root*
4: Stop, returning approximate root m and maximum error in solution $|b - a|/2$.
5: **else if** $f(a)$ and $f(m)$ are of opposite sign **then**
6: Set $b = m$ and repeat from step 1. ← *New interval is $[a, m]$*
7: **else**
8: Set $a = m$ and repeat from step 1. ← *New interval is $[m, b]$*
9: **end if**

Bisection Method: Implementation

```
# first implementation of bisection algorithm
def bisect(f, inita, initb, tol):

    fval = tol+1.0   # initialise the function evaluation
    a = inita        # set the starting value of a
    b = initb        # set the starting value of b

    # while |f(m)| > tol continue to iterate
    while abs(fval)> tol:
        m = (a+b)/2.0       # find the midpoint
        fval = f(m)         # evaluate the function at the midpoint
        if f(a)*fval < 0.0: # find the new range [a, b]
            b = m
        else:
            a = m
        print([a, b, m, fval])  # display the intermediate results

    return m, abs(b-a)/2.0
```

Test Problem

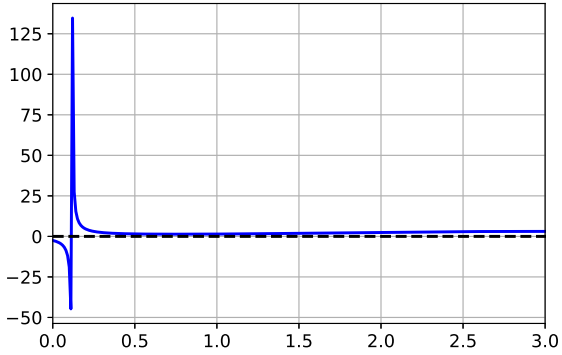


Figure 10. Plot of function $f(x) = \frac{x^3+4x^2+3x+5}{2x^3-9x^2+18x-2}$ over the interval $[0, 3]$.

Test Problem

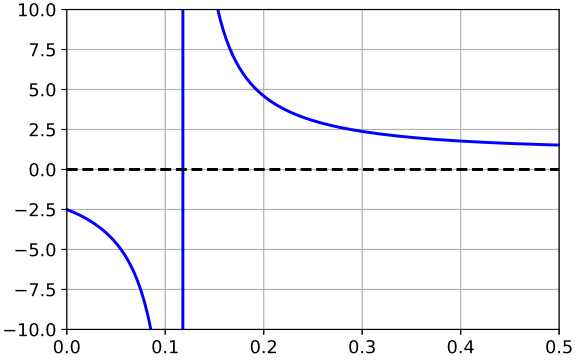


Figure 11. Plot of function $f(x) = \frac{x^3+4x^2+3x+5}{2x^3-9x^2+18x-2}$ over the interval $[0, 0.5]$.

Test Problem

Our algorithm fails to find the root of this function:

```
[0.11787656679530745, 0.11787656679530789, 0.11787656679530789, 1015345378381355.6]
[0.11787656679530745, 0.11787656679530767, 0.11787656679530767, 3046036135144066.5]
[0.11787656679530756, 0.11787656679530767, 0.11787656679530756, -1.2184144540576264e+16]
[0.11787656679530756, 0.11787656679530761, 0.11787656679530761, 6092072270288133.0]
[0.11787656679530756, 0.11787656679530759, 0.11787656679530759, 1.2184144540576266e+16]
Traceback (most recent call last):

File "bisection.py", line 69, in <module>
bisect(f, 0, 0.5, 1e-5)

File "bisection.py", line 60, in bisect
fval = f(m)          # evaluate the function at the midpoint

File "bisection.py", line 16, in f
return (x**3 + 4*x**2 + 3*x + 5.0) / (2*x**3 - 9*x**2 + 18*x - 2.0)

ZeroDivisionError: float division by zero
```

More Test Problems

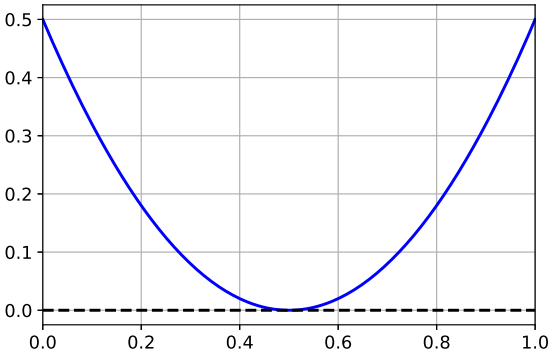


Figure 12. Another function which may cause problems.

More Test Problems

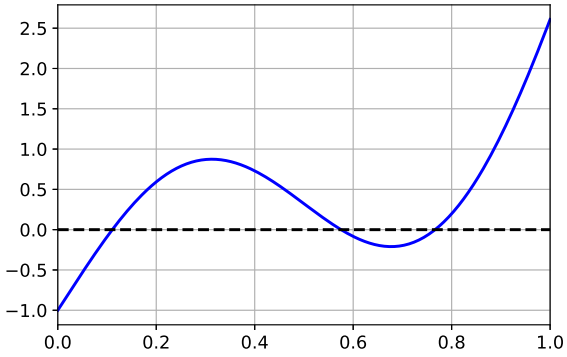


Figure 13. Another function which may cause problems.

More Test Problems

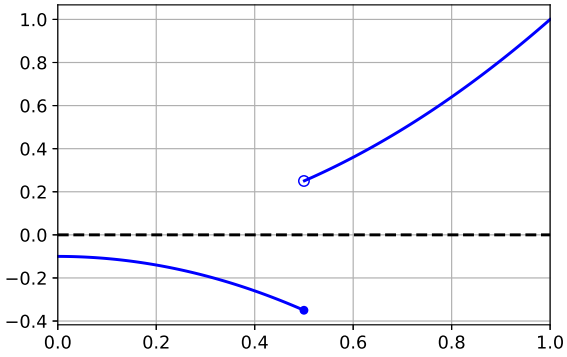


Figure 14. Another function which may cause problems.

Bisection Method: Second Version

Algorithm Bisection Method for solving $f(x) = 0$ (second version).

Input: a and b that surround a root, x-tolerance δ, f-tolerance ϵ, iteration limit N.

1: Evaluate $u = f(a)$, $v = f(b)$ and $e = b - a$, and initialise step counter $n = 0$.
2: **if** $\text{sign}(u) = \text{sign}(v)$ **then** stop with failure (bad initial interval)
3: Set $e = e/2$, $m = a + e$, $w = f(m)$ and $n = n + 1$. ← *calculate next iterate*
4: **if** $e < \delta$ **then** ← *m is at most δ from the root*
5: Stop and return m, e and w (success)
6: **else if** $|w| < \epsilon$ **then** ← *$f(m) \approx 0$*
7: Stop and return m, e and w (probable success)
8: **else if** $n = N$ **then** ← *hit iteration limit*
9: Stop and return m, e and w (failure to converge)
10: **end if**
11: **if** $\text{sign}(u) \neq \text{sign}(w)$ **then**
12: Set $b = m$ and $v = w$, and repeat from step 3. ← *New interval is $[a, m]$*
13: **else**
14: Set $a = m$ and $u = w$, and repeat from step 3. ← *New interval is $[m, b]$*
15: **end if**

Bisection Method: Weaknesses

Our second version is better than our first version, but there are still some potential weaknesses:

- Computers represent numbers in $\mathbb{R}$ with finite precision (usually approx. 15–16 significant figures). This can mean calculation of $f(m)$ can have errors, and may have the wrong sign!
  - Most likely a problem when $f(m) \approx 0$.
  - We will study this effect (roundoff error) later in the course.
- Can only find (at most) one root.
- How to set tolerances δ and ϵ, and iteration limit N?
- May be tricked by a singularity (or other discontinuity/jump in $f(x)$).

However, in most cases the bisection method is a reliable algorithm.

Bisection Method: Analysis

As long as $f(x)$ is continuous on the search interval, the bisection method converges to a root of f (ignoring rounding error).

When it stops, there is a root r in the final interval $[a, b]$, so the midpoint m is at most $d = (b - a)/2$ away from r:

$$|m - r| \leq d.$$

- Our algorithm produces a sequence of intervals $[a_n, b_n]$ for each iteration $n = 0, 1, 2, \ldots$.
  - Use the convention that $[a_0, b_0]$ is the starting interval.
- This gives a sequence of midpoints $m_n = (a_n + b_n)/2$.
- How quickly does m_n approach the true root r?

Bisection Method: Analysis

Since the root r is always in $[a_n, b_n]$, we always have the error

$$|m_n - r| \leq d_n,$$

where $d_n = (b_n - a_n)/2$.

Since our algorithm halves the interval width at every iteration, we have $d_n = d_{n-1}/2$. Therefore (by an induction argument),

$$|m_n - r| \leq d_n = \frac{d_{n-1}}{2} = \frac{d_{n-2}}{4} = \cdots = \frac{d_0}{2^n}.$$

That is,

$$|m_n - r| \leq \frac{d_0}{2^n} \to 0 \quad \text{as } n \to \infty.$$

The value $d_0 = (b_0 - a_0)/2$ is a constant depending on the starting interval (smaller starting interval means smaller errors).

Bisection Method: Analysis

The error $|m_n - r|$ approaches zero, so this means our sequence m_n converges to the true root r as $n \to \infty$ (as we would hope!).

However, we don't have infinite time. Instead, suppose we just want an approximate value for the root, with $|m - r| < \delta$ for some (small) $\delta > 0$.

How many iterations n does it take to guarantee $|m_n - r| < \delta$? Since $|m_n - r| \leq d_0/2^n$, we need

$$n \geq \log_2 \left(\frac{d_0}{\delta} \right) = \log_2 d_0 - \log_2 \delta.$$

For example, to get one extra decimal place of accuracy in m_n, we need to reduce δ by a factor of 10. This requires an extra $\log_2 10 \approx 3.3$ iterations. That is, every extra digit of accuracy requires another 3–4 iterations.

Orders of Convergence

This course will involve lots of analysis like this: a method produces a sequence of values $x_1, x_2, x_3, \ldots$ which approaches a limit x^* (which usually is the solution to our original problem). We need a way to describe "how fast does a sequence approach its limit". We do this by comparing the error $|x_{n+1} - x^*|$ with the previous error $|x_n - x^*|$.

- The rate of convergence is linear if there exists a constant $C < 1$ and integer N such that

$$|x_{n+1} - x^*| \leq C|x_n - x^*|, \qquad \text{for all } n \geq N.$$

- The rate of convergence is superlinear if there exists a sequence ϵ_n converging to zero and integer N such that

$$|x_{n+1} - x^*| \leq \epsilon_n|x_n - x^*|, \qquad \text{for all } n \geq N.$$

"for all $n \geq N$" means we don't care about what happens early on, only the eventual rate of convergence.

Orders of Convergence

- The rate of convergence is quadratic if there exists a constant C (not necessarily $C < 1$) and integer N such that

$$|x_{n+1} - x^*| \leq C|x_n - x^*|^2, \qquad \text{for all } n \geq N.$$

- The rate of convergence is order m if there exists a constant C (not necessarily $C < 1$ except for $m = 1$) and integer N such that

$$|x_{n+1} - x^*| \leq C|x_n - x^*|^m, \qquad \text{for all } n \geq N.$$

  So, $m = 1$ is linear convergence and $m = 2$ is quadratic convergence.

Any convergence order $m > 1$ is superlinear.

Orders of Convergence: Example

For the bisection method, our error at iteration n is at most

$$d_n = \frac{d_0}{2^n}.$$

That is, d_n converges to zero and

$$d_{n+1} = \frac{1}{2} d_n.$$

This is linear convergence (with $C = 1/2$ and $N = 0$).

Orders of Convergence: Example

The "Babylonian method" is an ancient method for computing square roots (known since at least AD60). To compute $\sqrt{c}$, pick a starting guess x_0 and then define

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{c}{x_n}\right), \qquad \text{for } n = 0, 1, 2, \ldots.$$

Calculating $\sqrt{2}$ starting from $x_0 = 1$, we get

n	xn	xn^2
0	1.0000000000000000	1.0000000000000000
1	1.5000000000000000	2.2500000000000000
2	1.4166666666666665	2.0069444444444444
3	1.4142156862745097	2.0000060073048824
4	1.4142135623746899	2.0000000000045106

Orders of Convergence: Example

Let's estimate the order of convergence by plotting $|x_{n+1} - \sqrt{2}|/|x_n - \sqrt{2}|^\alpha$ for different α:

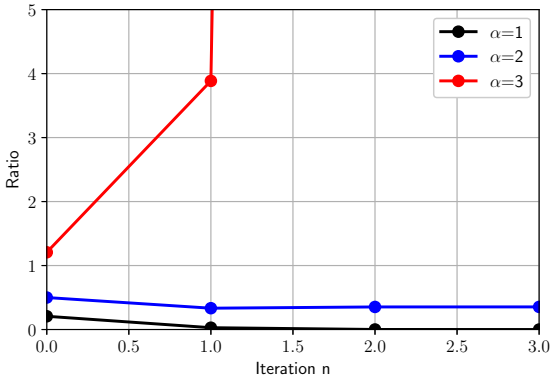


Figure 15. Estimating order of convergence for the Babylonian method.

Orders of Convergence: Example

The errors are:

```
n             xn                |xn-sqrt(2)|
0    1.0000000000000000    0.4142135623730951
1    1.5000000000000000    0.0857864376269049
2    1.4166666666666665    0.0024531042935714
3    1.4142156862745097    0.0000021239014145
4    1.4142135623746899    0.0000000000000002
```

For a method that converges quadratically, the number of correct digits eventually **doubles every iteration** (roughly). Since we only store numbers to 16 digits, we don't need to run many iterations.

Compare with bisection method (linear convergence): one extra digit takes 3–4 iterations.

Second-Order Methods for Rootfinding

The Babylonian method is a second-order method for calculating square roots. Is there a second-order method for general rootfinding problems $f(x) = 0$?

Yes, but we need to use more information about $f(x)$:

- Bisection only uses the sign of $f(x)$, not even the value of $f(x)$ (e.g. positive and close to zero vs. positive and large)
- We will use both the value of $f(x)$ and its first derivative $f'(x)$.

The algorithm is called Newton's Method (sometimes Newton-Raphson method).

History: Isaac Newton (1669) and Joseph Raphson (1690) applied to polynomials, extended to general functions by Thomas Simpson (1740). The Babylonian method is the same as Newton's method applied to $f(x) = x^2 - c$.

Iterative Approximations

The basic framework of many algorithms, including Newton's method, is iterative approximations:

1. Begin with an estimate x_0 of the solution x^*
2. At the n-th step, attempt to estimate the error $e_n = x^* - x_n$. Suppose we get $\widetilde{e}_n \approx e_n$.
3. Define our new estimate $x_{n+1} = x_n + \widetilde{e}_n$.
4. Repeat from step #2 until our solution is "sufficiently accurate" (e.g. $\widetilde{e}_n$ is small).

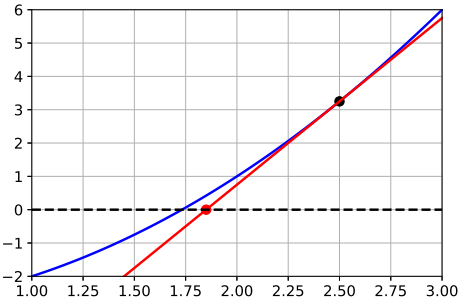
One advantage of this method is that if we are lucky and our approximation $\widetilde{e}_n$ is exact (i.e. $\widetilde{e}_n = e_n$), then we get to the solution in one step:

$$x_{n+1} = x_n + \widetilde{e}_n = x_n + e_n = x_n + (x^* - x_n) = x^*.$$

This is a general method. How does it apply to rootfinding?

Newton's Method

If we want $f(x) = 0$ and we are at x_n, how can we estimate how far away we are from a root?



Distance to true root

$\approx$

Distance to root of tangent line

i.e. x_{n+1} = root of the tangent at x_n.

Figure 16. Motivation for Newton's Method.

Newton's Method: Derivation

The tangent line to f based at x_n is

$$t_n(x) = f(x_n) + f'(x_n)(x - x_n).$$

So, our new iterate is the root of this line (i.e. solve the linear equation $t_n(x) = 0$):

$$\boxed{x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}}$$

This corresponds to the error estimate

$$x^* - x_n \approx \widetilde{e}_n = -\frac{f(x_n)}{f'(x_n)}.$$

Reminder: Taylor Series

Next, we will look at a derivation based on Taylor series (Brook Taylor, 1715). Remember, the Taylor series for $f(x)$ based at $x = a$ is the power series

$$f(a) + f'(a)(x-a) + \frac{f''(a)}{2}(x-a)^2 + \cdots + \frac{f^{(k)}(a)}{k!}(x-a)^k + \cdots$$

If f is infinitely differentiable and x is close to a, then the Taylor series converges to $f(x)$.

In scientific computing, we often try to approximate a function by simpler functions. One very common way to do this is to use a Taylor series, truncated after some number of terms:

$$\boxed{f(x) \approx f(a) + f'(a)(x-a) + \cdots + \frac{f^{(k)}(a)}{k!}(x-a)^k}$$

This approximation is not perfect: the error is

$$e_k(x) = f(x) - \left(f(a) + f'(a)(x-a) + \cdots + \frac{f^{(k)}(a)}{k!}(x-a)^k \right).$$

Reminder: Taylor Series

Taylor's Theorem is an important result that tells us how big the Taylor error is. Usefully, we don't need f to be infinitely differentiable any more, so we can cope with more functions.

Theorem (Taylor's Theorem)

If f is $(k+1)$-times continuously differentiable on the interval between a and x, the Taylor error is

$$e_k(x) = \frac{f^{(k+1)}(\xi)}{(k+1)!}(x-a)^{k+1},$$

where ξ is some (unknown) point between a and x (possibly different for every x, a and k).

If x is close to a and the derivative $f^{(k+1)}$ is not too large, then $e_k(x)$ is probably small (i.e. we have a good approximation).

Note: versions of Taylor's theorem also exist for multivariate functions.

Newton's Method: Derivation

We can also derive Newton's method in a different way. Write the Taylor series for $f(x)$ at x_n:

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + \frac{f''(x_n)}{2!}(x - x_n)^2 + \cdots$$

Then approximate f by taking just the first two terms of the Taylor series:

$$f(x) \approx f(x_n) + f'(x_n)(x - x_n).$$

We expect this to be a good approximation when x is close to x_n.

Replace the hard problem $f(x) = 0$ with the easy problem $f(x_n) + f'(x_n)(x - x_n) = 0$ to get Newton's method.

Newton's Method comes from approximating f near x_n by a simpler function. Here, it is the first two terms of the Taylor series.

Newton's Method: Implementation

Below is a Python implementation of Newton's method (terminating when we take small steps).

```
def newton(f, df, x0, delta):
    x = x0            # initial guess
    err = delta+1.0   # set an upper bound on the error
    print('The inital guess is', x, 'and f(x) =', f(x))

    # while the error estimate is large
    while err > delta :
        y = f(x)        # evaluate the function
        dydx = df(x)    # evaluate the derivative
        dx =  - y/dydx  # find the step size
        err = abs(dx)   # estimate the error
        x = x + dx      # take a step
        print('The new approximate root is', x)
        print('f(x) =',y)
        print('The error estimate is', err)
    return x
```

Newton's Method: Analysis

What is the convergence rate of Newton's Method? If $e_n = r - x_n$ for true root r, we get

$$e_{n+1} = r - x_{n+1} = r - \left(x_n - \frac{f(x_n)}{f'(x_n)} \right) = e_n + \frac{f(x_n)}{f'(x_n)}.$$

We can estimate the size of the RHS using Taylor's theorem with a remainder term:

$$0 = f(r) = f(x_n + e_n) = f(x_n) + f'(x_n)e_n + \frac{f''(\xi_n)}{2} e_n^2,$$

for some ξ_n between x_n and r. Therefore

$$e_{n+1} = e_n + \frac{f(x_n)}{f'(x_n)} = -\frac{f''(\xi_n)}{2f'(x_n)} e_n^2.$$

If x_n is close to r, then ξ_n is also close to r. Then $\frac{f''(\xi_n)}{2f'(x_n)} \approx \frac{f''(r)}{2f'(r)}$, and so

$$e_{n+1} \approx -\frac{f''(r)}{2f'(r)} e_n^2 = Ce_n^2.$$

So Newton's method has quadratic convergence whenever x_n is close to r and $f'(r) \neq 0$.

Newton's Method: Examples

We have already seen Newton's method applied to $f(x) = x^2 - 2$ starting from $x_0 = 1$ (Babylonian method):

n	xn	f(xn)
0	1.0000000000000000	-1.0000000000000000
1	1.5000000000000000	0.2500000000000000
2	1.4166666666666667	0.0069444444444446
3	1.4142156862745099	0.0000060073048829
4	1.4142135623746899	0.0000000000045106
5	1.4142135623730951	0.0000000000000004

Now let's try to find the root $r = 0$ for the function $f(x) = x/(1 + x^2)$, starting from $x_0 = 0.5$:

n	xn	f(xn)
0	0.5000000000000000	0.4000000000000000
1	-0.3333333333333334	-0.3000000000000000
2	0.0833333333333334	0.0827586206896553
3	-0.0011655011655012	-0.0011654995822947
4	0.0000000031664215	0.0000000031664215
5	0.0000000000000000	0.0000000000000000

Newton's Method: Comments

- Newton's method requires that you can compute $f'(x)$. This is easy if you know the equation for $f(x)$, otherwise you need to approximate derivatives (covered later).
- Just like the bisection method, we are not thinking about rounding errors, and we can only find one root
  - No guarantee we find a root close to our starting point, and nearby starting points can converge to different roots. (search "Newton fractal")
  - Limited techniques for finding multiple roots (e.g. deflation).
- What happens if $f'(r) = 0$?
  - Get linear convergence rather than quadratic!
- What happens if x_0 is not close to r?
  - Approximation $\frac{f''(\xi_n)}{2f'(x_n)} \approx \frac{f''(r)}{2f'(r)}$ may not be accurate
  - Also, if e_n is too large then $e_{n+1} \approx Ce_n^2$ might be *bigger* than e_n and we diverge
  - This is a serious problem: we don't know what r is, so we can't guarantee we start close to it!

Newton's Method: Bad starting points

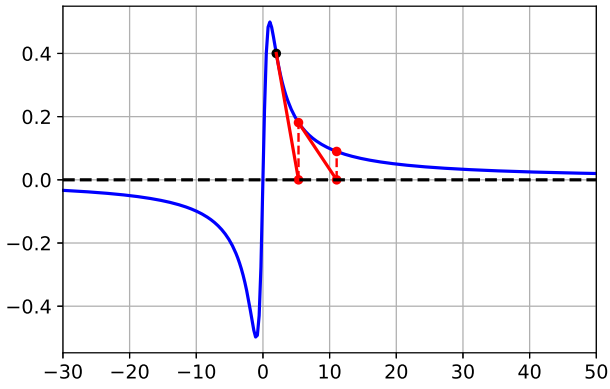


Figure 17. Newton's Method failure: $f(x) = x/(1+x^2)$ starting at $x_0 = 2$.

Newton's Method: Bad starting points

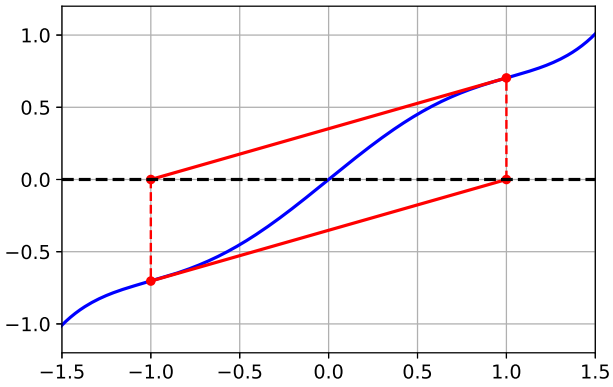


Figure 18. Newton's Method failure: $f(x) = \frac{11}{91}x^5 - \frac{38}{91}x^3 + x$ starting at $x_0 = 1$.

Newton's Method: Bad starting points

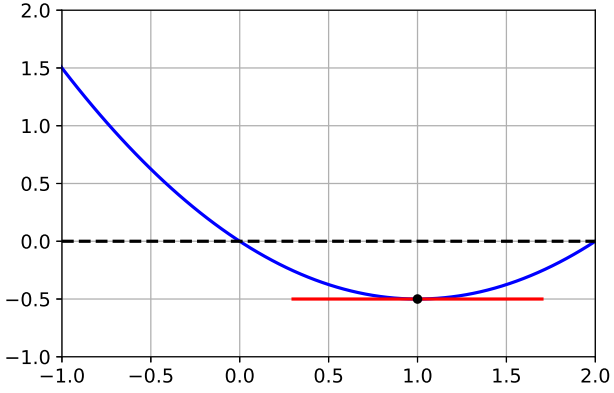


Figure 19. Newton's Method failure: $f(x) = \frac{1}{2}(x-1)^2 - \frac{1}{2}$ starting at $x_0 = 1$.

Global vs. Local Convergence

Two important types of convergence:

- Local convergence: if x_0 is sufficiently close to r, then $x_n \to r$
  - Usually "sufficiently close" is problem-dependent, and can't be easily checked.
- Global convergence: $x_n \to r$ for any choice of starting point x_0
- Newton's method only has local convergence (but is fast), bisection method has global convergence provided $f(a_0)$ and $f(b_0)$ have different signs (but is slow)

In practice, it is common to start by running a globally convergent method (like bisection) then switch to a fast locally convergent method (like Newton) for 4–5 iterations at the end.

Secant Method

We saw that by using the values of $f(x)$ and $f'(x)$ we can build a second-order method.

Question

What happens if we can only compute $f(x)$, and not $f'(x)$?

Idea: use Newton's method, but approximate the derivative $f'(x_n)$ using values of $f(x)$:

$$f'(x_n) \approx \frac{f(x_n + h) - f(x_n)}{h},$$

for some small value of h.

- Requires two evaluations of $f(x)$ at every iteration, so convergence is slower (e.g. if evaluating $f(x)$ is slow).
- Approximation only accurate if h is small, but this also leads to large roundoff errors (so estimate becomes less accurate).

We will study this type of derivative approximation later in the course.

Secant Method

An alternative idea is to recycle old function values, and approximate the derivative using

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}},$$

which is accurate when $x_n \approx x_{n-1}$. This only requires one evaluation of f per iteration.

Substituting this approximation into Newton's method gives the secant method

$$\boxed{x_{n+1} = x_n - f(x_n)\left(\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}\right)}$$

(so called because we replace the tangent line with a secant line)

Note that x_{n+1} depends on both x_n and x_{n-1}, so we need two starting points.

Secant Method

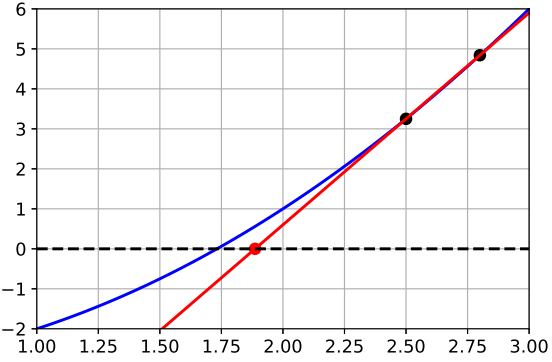


Figure 20. Motivation for Secant Method.

Secant Method: Implementation

Below is a Python implementation of the secant method.

```
def secant(f, x1, x2, delta, epsilon, N):
    f1 = f(x1); f2 = f(x2)      # initial guess
    for step in range(N):       # upper bound on interations
        if f1 == f2:               # avoid devision by zero
            print('error: division by zero')
            return x1
        dx = -f2*(x2-x1)/(f2-f1)  # find step szie
        x3 = x2+dx                # take a step
        err = abs(dx)             # approximate the error
        f3 = f(x3)                # evaluate function at point
        if err < delta or abs(f3) < epsilon:
            return x3             # return if error is small or x3 close to root
        x1 = x2; f1 = f2; x2 = x3; f2 = f3    # next iteration

    # if maximum number of iterations reached print message
    print('error: iteration limit was reached without convergence')
    return x2
```

Secant Method: Analysis

Using similar arguments as Newton's method, if the error is $e_n = r - x_n$, then we can derive

$$e_{n+1} = -\frac{f''(\xi_n)}{2f'(\zeta_n)} e_{n-1} e_n,$$

for some ξ_n and ζ_n between (two of) r, x_{n-1} and x_n. If x_{n-1} and x_n are close to r, then we get

$$e_{n+1} \approx -\frac{f''(r)}{2f'(r)} e_{n-1} e_n = C e_{n-1} e_n,$$

with the same C as Newton's method. Therefore $e_{n+1}/e_n \approx C e_{n-1} \to 0$, so the secant method converges superlinearly.

It does not converge quadratically, since e_{n-1} is (much) larger than e_n, so $C e_{n-1} e_n$ is much larger than $C e_n^2$. Not using $f'(x)$ has meant a slower convergence than Newton.

However, if we assume $|e_{n+1}| \approx A|e_n|^m$ for some order m, it is easy to show that the secant method has order $m = (1 + \sqrt{5})/2 \approx 1.62$ (the golden ratio).

Robust Methods

Both the secant method and Newton's method may not work if we start far away from r.

In practice, most standard rootfinding solvers use a mixture of fast and robust methods:

- Always make sure we have an interval $[a_n, b_n]$ surrounding a root
- First try a secant step (or some other fast-converging sequence)
- If this step is not good enough, use a bisection step

The most common version of this idea is Brent's method (Richard Brent, 1973 — ANU Professor 1978–1998, 2005–2011). It is implemented in Python in the SciPy package (`scipy.optimize.root_scalar`).

Higher dimensions

What if we have d nonlinear equations in d unknowns?

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_d) &= 0, \\ f_2(x_1, x_2, \ldots, x_d) &= 0, \\ \vdots \qquad & \qquad \vdots \\ f_d(x_1, x_2, \ldots, x_d) &= 0. \end{aligned}$$

We can derive a multidimensional version of Newton's method.

Multidimensional Newton's Method

Let's write $\boldsymbol{x} = [x_1, x_2, \ldots, x_d]^T \in \mathbb{R}^d$.

First, we approximate each f_i by its linear (multidimensional) Taylor series at $\boldsymbol{x}_n$:

$$f_i(\boldsymbol{x}_n + \boldsymbol{e}) \approx f_i(\boldsymbol{x}_n) + \frac{\partial f_i(\boldsymbol{x}_n)}{\partial x_1} e_1 + \cdots + \frac{\partial f_i(\boldsymbol{x}_n)}{\partial x_d} e_d,$$

then try to find a step $\boldsymbol{e}$ which makes all the linear Taylor series zero. This reduces to:

$$\boxed{\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - \begin{bmatrix} \frac{\partial f_1(\boldsymbol{x}_n)}{\partial x_1} & \cdots & \frac{\partial f_1(\boldsymbol{x}_n)}{\partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_d(\boldsymbol{x}_n)}{\partial x_1} & \cdots & \frac{\partial f_d(\boldsymbol{x}_n)}{\partial x_d} \end{bmatrix}^{-1} \begin{bmatrix} f_1(\boldsymbol{x}_n) \\ \vdots \\ f_d(\boldsymbol{x}_n) \end{bmatrix}}$$

We will discuss how to solve linear systems later in the course.

This Newton's method also has quadratic convergence (if $\boldsymbol{x}_0$ is close to a non-degenerate solution). For a globally convergent method or methods not requiring derivatives of f_i, we need to use techniques from optimisation (`scipy.optimize.root`).