Contents

6 Root finding

We will now look at solving algebraic equations. In general, this reduces to a root-finding problem, for in general, if we wish to find a solution to $f(x) = g(x)$, this is equivalent to finding a root of the expression $f(x) - g(x)$. We will briefly review how to solve a simple linear equation and a system of linear equations, and then we will look at four techniques for finding a root of a non-linear equation, including:

1. the bisection method,
2. the constrained secant method,
3. Newton's method, and
4. the secant method.

We will end by looking how we can use Newton's method to find an approximation to the root of a system of non-linear equations.

6.1 A linear equation and a system of linear equations

The only algebraic equation that we can solve with any confidence is the simple equation

$$ax = b,$$

which is equivalent to finding a root of $ax - b$. Finding a unique solution is guaranteed so long as $a \neq 0$, and if $a = 0$, then either there are infinitely many solutions if $b = 0$ and no solutions if $a \neq 0$.

We have already discussed the techniques for solving a system of linear equations in a numerically stable manner; namely, to apply partial pivoting. Again, solving the following system of non-linear equations

$$\begin{aligned} ax + by &= c \\ dx + ey &= f \end{aligned}$$

is equivalent to finding a simultaneous root of the two functions

$$\begin{aligned} ax + by - c \\ dx + ey - f \end{aligned}.$$

Thus, if you were to find a simultaneous root of the algebraic expressions,

$$\begin{aligned} f(x, y) &= 3x + 7y + 1 \\ g(x, y) &= x - 5y + 4 \end{aligned}$$

finding a simultaneous solution to $f(x, y) = 0$ and $g(x, y) = 0$ is equivalent to solving

$$\begin{aligned} 3x + 7y &= -1 \\ x - 5y &= -4 \end{aligned}.$$

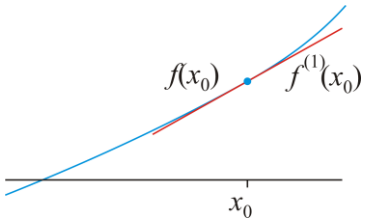
We will therefor proceed to finding solutions to a non-linear equation.

6.2 A non-linear equation

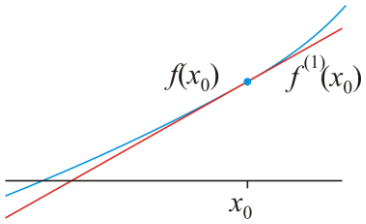
Suppose we have a non-linear equation $f(x) = g(x)$, which is essentially a root-finding problem for the algebraic expression $f(x) - g(x)$. In calculus, you determined that if you had an approximation x in the vicinity of the root, you could determine that the tangent to the function at x is given by the expression

$$f^{(1)}(x_0)(x-x_0)+f(x_0).$$

This tangent line is shown in this image:



As you may suspect, if we are in the vicinity of a root, the root of this tangent line will be close to the root of the actual function:



We can find the root of the tangent line by solving for

$$\begin{aligned} f^{(1)}(x_0)(x-x_0)+f(x_0)&=0 \\ f^{(1)}(x_0)(x-x_0)&=-f(x_0) \\ x-x_0&=-\frac{f(x_0)}{f^{(1)}(x_0)} \\ x&=x_0-\frac{f(x_0)}{f^{(1)}(x_0)} \end{aligned}$$

Thus, the root of this tangent line is $x_0-\frac{f(x_0)}{f^{(1)}(x_0)}$. This can be our next approximation to the root. This was all covered in first-year calculus; however, let us now examine the error. To begin, however, we will look at a different technique: the bisection method.

6.2.1 The bisection method

If we start with a continuous function f defined on an interval $[a, b]$, then if $f(a)$ and $f(b)$ have opposite signs, we know by the intermediate value theorem that there must be a point on (a, b) that is a root.

We could approximate the root by the mid-point; however, it is probably better to just approximate the root by whichever $|f(a)|$ or $|f(b)|$ is smaller in absolute value.

To improve our approximation, we could apply the same rule as in binary search: select the mid-point, only now, not the mid-point of the array, but the mid-point of the interval $[a, b]$.

Thus,

1. let $c \leftarrow \frac{a+b}{2}$,
2. in the unlikely case that $f(c) = 0$, c is a root and we are finished,
3. otherwise,
    a. if $f(a)$ and $f(c)$ the same sign, update a as a root is on the interval $[c, b]$,
    b. otherwise, a root is on the interval $[a, c]$.

6.2.1.1 Implementation issues

Now, when we implement the bisection method in C++, we need to know when to stop.

First, because we are assuming that we are looking for a root of a function f, then if we have an approximation to a root x, then $|f(x)|$ must be small, so we allow the user to specify this constraint: the user can specify ε_{abs} so that $|f(x)| < \varepsilon_{abs}$. Next, given a and b together with $f(a)$ and $f(b)$, which we have already calculated, should we return a or b or perhaps the mid-point $\frac{a+b}{2}$. If we restrict ourselves to returning only a or b, then we should return whichever has a smaller corresponding absolute value: if $|f(a)| < |f(b)|$, return a; otherwise, return b. Because we know nothing about $|f(c)|$, if we were to return c, it may be a worse approximation than either a or b, and if we're calculating $f(c)$ to avoid this issue, we might as well continue one more iteration of the bisection method.

Another problem is we want to be close to the root, but we are simply creating a sequence of approximations: how do we know when we are *close enough* to the root? To solve this, we will say that we are *close enough* to the root if the distance between the end-points is close; that is, $b - a < \varepsilon_{step}$ for an ε_{step} specified by the user. Now, we don't have to keep all approximations, only the current and the most previous approximation.

6.2.1.2 Implementation in C++

We could now write an algorithm that finds such a root:

```
#include <cassert>
#include <stdexcept>
#include <iostream>

// Function declarations
int sign( double x );
double bisection_method( double f( double x ), double a, double b,
                         double eps_step, double eps_abs,
                         unsigned int max_iterations );

// Function definitions
int sign( double x ) {
    return (0.0 < x) - (x < 0.0);
}

double bisection_method( double f( double x ), double a, double b,
                         double eps_step, double eps_abs,
                         unsigned int max_iterations ) {
    assert( a < b );
    double fa{f(a)};
    double fb{f(b)};
    double f_min{std::min(std::abs(fa), std::abs(fb))};
    unsigned int iterations{0};

    if ( fa == 0.0 ) {
        std::clog << "Number of iterations: 0" << std::endl;
        return a;
    } else if ( fb == 0.0 ) {
        std::clog << "Number of iterations: 0" << std::endl;
        return b;
    }

    if ( sign( fa ) == sign( fb ) ) {
        throw std::invalid_argument( "f(a) and f(b) have the same sign." );
    }

    while ( ((b - a) >= eps_step) && (f_min >= eps_abs) ) {
        ++iterations;

        if ( iterations > max_iterations ) {
            throw std::range_error( "Exceeded the maximum number of iterations" );
        }

        double c{(a + b)/2};
        double fc{f(c)};

        f_min = std::min( f_min, std::abs( fc ) );

        if ( fc == 0.0 ) {
            std::clog << "Number of iterations: " << iterations << std::endl;
            return c;
        } else if ( sign( fa ) == sign( fc ) ) {
            // 'f(a)' and 'f(c)' have the same sign
            a = c;
            fa = fc;
        } else {
            // 'f(c)' and 'f(b)' have the same sign
            assert( sign( fc ) == sign( fb ) );
            b = c;
            fb = fc;
        }
    }

    std::clog << "Number of iterations: " << iterations << std::endl;
    return ( std::abs( fa ) < std::abs( fb ) ) ? a : b;
}
```

You can validate this algorithm with a test run:

```
#include <cmath>
#include <iostream>

// Function declarations
// ...
int main();

// Function definitions

double y( double t ) {
        return 2.3*std::exp(-t) – 5.0*t*std::exp(-t);
}

int main() {
        std::cout.precision( 16 );

        std::cout << "Bisection method" << std::endl;
        std::cout << bisection_method( std::cos, 1.0, 2.0, 1e-6, 1e-6, 100 )
                  << std::endl;
        std::cout << bisection_method( y,        0.0, 0.5, 1e-6, 1e-6, 100 )
                  << std::endl;

        return 0;
}
```

The output is:

```
Bisection method
Number of iterations: 18
1.570796966552734
Number of iterations: 19
0.4600000381469727
```

The correct answer for the first to16 decimal digits is **1.570796**326794897, so the relative error 0.00001999 %, and with each iteration, the error reduces by a factor of two. In the second case, the correct answer is 0.46, so you can see the error by inspection.

Now, if we know the interval is [a, b] and the signs of the function evaluated at the end-points differ, then we could state that the error is $h = |b - a|$. With each iteration, the error drops by a factor of two, so after one iteration, the error is $0.5h$. Thus, we can say that the error is $O(h)$.

We will now refine this technique, by using a better guess than just the mid-point.

6.2.2 The bracketed secant method (or *false position* method)

When searching a sorted array of size *n*, it is usual to apply a binary search, which is analogous to the bisection method. However, suppose you had a phone book and you were looking up Prof. Zarnett's phone number. You would not open the book in the middle, determine that 'Z' falls after 'M', and then jump half-way towards 'Z' again. Instead, you would make an educated guess as to how far to jump. Similarly, if you were searching a sorted array of size *n*, where `array[103] == 5.5` and `array[999] == 88.2`, and you were searching for the entry containing 19.7, the logical first choice would be calculate $\frac{19.7-5.5}{88.3-5.5} \approx 0.1714$ and then calculate

```
k = 103 + round(0.1714*(999 - 103));
```

which works out to entry 257, which is approximately 17.14% into the array from the lower bound. Such a search is valid if the entries are approximately uniformly distributed, in which case the convergence time is O(ln(ln(*n*)); which is significantly faster than a binary search. This search is called an *interpolation search*.

As an example of where such a search is valid, consider an array containing these sorted entries:

0.224, 1.07, 1.93, 2.11, 3.30, 3.86, 3.96, 3.96, 4.12, 4.28, 4.55, 6.95, 7.31, 7.37, 7.50, 7.73, 8.00, 8.43, 9.45, 9.96

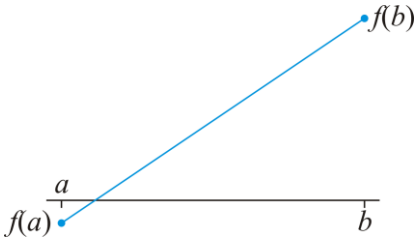
Interpolation search, however, will perform significantly worse than binary search if the entries are:

0.224, 1.07, 1.93, 2.11, 3.30, 3.86, 3.96, 3.96, 4.12, 4.28, 4.55, 6.95, 7.31, 7.37, 7.50, 7.73, 8.00, 8.43, 9.45, **996**

Why?

Similarly, in finding a root on an interval [*a*, *b*], suppose *f*(*a*) = –0.00003 and *f*(*b*) = 0.00024. One may assume that the root is likely closer to *a* than it is to *b*:

Rather than choosing the mid-point, it would be reasonable to choose a point closer to *a*. How much closer? If we find the interpolating linear polynomial, we can find the root of that interpolating polynomial:



The interpolating polynomial is

$$\frac{f(b)-f(a)}{b-a}(x-a)+f(a)\ .$$

Finding the root simply requires us to equate this to zero and solve for *x*:

$$\frac{f(b)-f(a)}{b-a}(x-a)+f(a)=0$$

$$\frac{f(b)-f(a)}{b-a}(x-a)=-f(a)$$

$$x-a=-f(a)\frac{b-a}{f(b)-f(a)}$$

$$x=a-f(a)\frac{b-a}{f(b)-f(a)}$$

The expression is a little awkward, so we can simplify it:

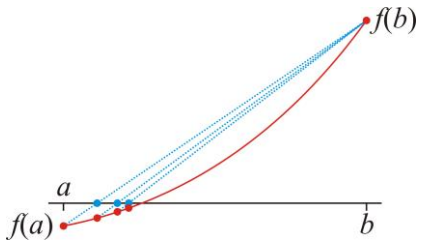
$$\frac{f(b)a-f(a)a-f(a)b+f(a)a}{f(b)-f(a)}=\frac{f(b)a-f(a)b}{f(b)-f(a)}.$$

Thus,

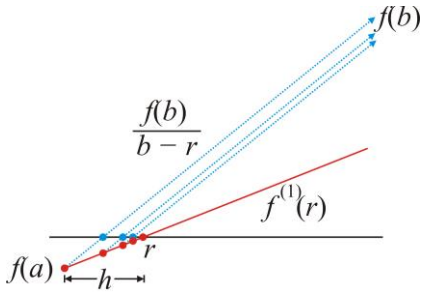
1. let $c \leftarrow \frac{f(b)a-f(a)b}{f(b)-f(a)}$,
2. in the unlikely case that $f(c)=0$, *c* is a root and we are finished,
3. otherwise,
    a. if *f*(*a*) and *f*(*c*) have the same sign, update *a* as the a root is on the interval [*c*, *b*],
    b. otherwise, a root is on the interval [*a*, *c*].

6.2.2.1 Error analysis

In general, most functions will be either concave up or concave down at a root: having an inflection point at a root is unlikely. Thus, in general, only one end-point will be updated:



Now, zooming in on a point very close to the root, we get the following image:



Note that $f^{(1)}(r)=\frac{-f(a)}{h}$ or $h=\frac{-f(a)}{f^{(1)}(r)}$ and that the next approximation moves closer to the root by $\frac{f(b)}{b-r}=\frac{-f(a)}{\Delta}$ so $\Delta=\frac{f(a)(r-b)}{f(b)}$, so if the error was h, it is now

$$\frac{\frac{-f(a)}{f^{(1)}(r)}-\frac{f(a)(r-b)}{f(b)}}{\frac{-f(a)}{f^{(1)}(r)}}h=\left(1-\frac{f^{(1)}(r)}{f(b)/(r-b)}\right)h.$$

You will note that $1-\frac{f^{(1)}(r)}{f(b)/(r-b)}$ is a constant near the root, so the convergence is O(h). You will note that if $\frac{f(b)}{r-b}$ is close to the slope at the derivative, we have faster convergence. Thus, one solution is to alternate between the bisection method and the secant method, the bisection method usually bring b closer to the root.

6.2.2.2 Implementation issues

Now, when we implement the secant method in C++, we need to know when to stop.

Like with the bisection method, we will let the user can specify ε_{abs} so that $|f(x)| < \varepsilon_{abs}$. Additionally, we will return that end-point that is smaller. One problem is, however, that the interval will not shrink to zero; instead, in general, only one end-point will change from iteration to iteration. Consequently, we will stop if $a_{prev} - a < \varepsilon_{step}$ and $|f(a)| < \varepsilon_{abs}$ or $b - b_{prev} < \varepsilon_{step}$ and $|f(b)| < \varepsilon_{abs}$ specified by the user. Now, we don't have to keep all approximations, only the current and the most previous approximation for each end-point.

6.2.2.3 Implementation in C++

We could now write an algorithm that finds such a root:

```
#include <cassert>
#include <stdexcept>
#include <iostream>
#include <limits>

double bracketed_secant_method( double f( double x ),
                      double a, double b,
                      double eps_step, double eps_abs,
                      unsigned int max_iterations ) {
    assert( a < b );
    double fa{f(a)};
    double fb{f(b)};
    double f_min{std::min(std::abs(fa), std::abs(fb))};
    double a_prev{-std::numeric_limits<double>::infinity()};
    double b_prev{ std::numeric_limits<double>::infinity()};
    unsigned int iterations{0};

    if ( fa == 0.0 ) {
        std::clog << "Number of iterations: 0" << std::endl;
        return a;
    } else if ( fb == 0.0 ) {
        std::clog << "Number of iterations: 0" << std::endl;
        return b;
    }

    if ( sign( fa ) == sign( fb ) ) {
        throw std::invalid_argument( "f(a) and f(b) have the same sign." );
    }
```

```
        do {
            ++iterations;

            if ( iterations > max_iterations ) {
                throw std::range_error( "Exceeded the maximum number of iterations" );
            }

            double c{(fb*a - fa*b)/(fb - fa)};
            assert( (a <= c) && (c <= b) );
            double fc{f(c)};
            f_min = std::min( f_min, std::abs( fc ) );

            if ( fc == 0.0 ) {
                std::clog << "Number of iterations: " << iterations << std::endl;
                return c;
            } else if ( sign( fa ) == sign( fc ) ) {
                // 'f(a)' and 'f(c)' have the same sign
                a_prev = a;
                a = c;
                fa = fc;
            } else {
                // 'f(c)' and 'f(b)' have the same sign
                assert( sign( fc ) == sign( fb ) );
                b_prev = b;
                b = c;
                fb = fc;
            }
        } while ( ((a_prev - a >= eps_step) || (std::abs( fa ) >= eps_abs))
               && ((b - b_prev >= eps_step) || (std::abs( fb ) >= eps_abs))  );

        std::clog << "Number of iterations: " << iterations << std::endl;

        return ( std::abs( fa ) < std::abs( fb ) ) ? a : b;
    }
```

We can test this

```
    std::cout << "Secant method" << std::endl;
    std::cout << bracketed_secant_method( std::cos, 1.0, 2.0, 1e-6, 1e-6, 100 )
              << std::endl;
    std::cout << bracketed_secant_method( y, 0.0, 0.5, 1e-6, 1e-6, 100 ) << std::endl;
```

If we run this, we get the output:

```
    Number of iterations: 3
    1.570796325773051
    Number of iterations: 12
    0.4600002580872375
```

6.2.3 Newton's method

You learned Newton's method in first year, and we introduced the topic on root finding by describing how we came up with the technique by using a first-order Taylor series to get the next approximation of the root to be

$$x_n \leftarrow x_{n-1} - \frac{f(x_{n-1})}{f^{(1)}(x_{n-1})}.$$

We will now perform an error analysis on Newton's method. Now, from the Taylor series, let us assume that x is a value near a root r. In this case, we have:

$$f(r) = f(x) + f^{(1)}(x)(r-x) + \frac{1}{2} f^{(2)}(\xi)(r-x)^2$$

where $r < \xi < x$. Under the assumption that r is a root, we have that the left-hand side of this equation is zero:

$$0 = f(x) + f^{(1)}(x)(r-x) + \frac{1}{2} f^{(2)}(\xi)(r-x)^2.$$

Solving for r, we have

$$-f(x) - \frac{1}{2} f^{(2)}(\xi)(r-x)^2 = f^{(1)}(x)(r-x)$$

$$-\frac{f(x)}{f^{(1)}(x)} - \frac{1}{2}\frac{f^{(2)}(\xi)}{f^{(1)}(x)}(r-x)^2 = (r-x)$$

$$r = x - \frac{f(x)}{f^{(1)}(x)} - \frac{1}{2}\frac{f^{(2)}(\xi)}{f^{(1)}(x)}(r-x)^2$$

Therefore, as an approximation to r, the error of x is $r - x$, but this equation indicates that the error of $x - \frac{f(x)}{f^{(1)}(x)}$ as an approximation to r is $-\frac{1}{2}\frac{f^{(2)}(\xi)}{f^{(1)}(x)}(r-x)^2$. In the vicinity of the root, $-\frac{1}{2}\frac{f^{(2)}(\xi)}{f^{(1)}(x)}$ will be reasonably constant and reasonable in size so long as we do not have a double or higher-order root (in which case, the denominator will be close to zero). If $r - x$ is sufficiently small, then $(r-x)^2$ will again be smaller.

For example, consider the function $y(t) = 2.3\ e^{-t} - 5.0t\ e^{-t}$. This is a solution to a 2nd-order initial value. We note that $y(0) = 2.3$, and $y(0.5) < 0$ (why?). Thus, there must be a root and it is likely in the vicinity of $t = 0.5$. Let us try Newton's method. The sequence of approximations is

0.5, 0.4615384615384615, 0.4600023632281697, 0.4600000000055849, 0.4600000000000001, 0.4600000000000000

Approximation	Error	Error squared
0.5	0.04	0.0016
0.4615384615384615	0.0015384615384615	0.000002367
0.4600023632281697	0.0000023632281697	0.000000000005585
0.4600000000055849	0.0000000000055849	

Now, at the root, the ratio $-\frac{1}{2}\frac{f^{(2)}(0.46)}{f^{(1)}(0.46)} = -1$ exactly.

Now, suppose you want to find the first root of the response $y(t) = 1.2\ e^{-t}\cos(t) - 3.7\ e^{-t}\sin(t)$. We know the solution is in the vicinity of 0.3, so we will start with that value:

Approximation	Error	Error squared
0.3	–0.0136206522166983	0.0001855
0.3134384429127443	–0.0001822093039540	0.00000003320
0.3136206190245318	–0.0000000331921665	0.00000000000001102
0.3136206522166972	–0.0000000000000011	

6.2.3.1 Implementation issues

Now, when we implement Newton's method in C++, we need to know when to stop. First, there may be no root, so we should allow the user to specify a maximum number of iterations.

Next, how do we know when to stop? First, because we are assuming that we are looking for a root of a function f, then if we have an approximation to a root x, then $|f(x)|$ must be small, so we allow the user to specify this constraint: the user can specify ε_{abs} so that $|f(x)| < \varepsilon_{abs}$.

Another problem is we want to be close to the root, but we are simply creating a sequence of approximations: how do we know when we are *close enough* to the root? To solve this, we will say that we are *close enough* to the root if the distance between the current approximation and the previous approximation is close; that is, $|x_k - x_{k-1}| < \varepsilon_{step}$ for an ε_{step} specified by the user. Now, we don't have to keep all approximations, only the current and the most previous approximation.

Another issue with Newton's method is that we do not have the derivative. There are three ways of getting the derivative:

1. Having the user provide a second function that returns the derivative of the first function for which we are finding the root.
2. Approximating the derivative using the techniques we saw previously.
3. Automatic differentiation: the algorithm means of taking one function definition and creating a new function definition that returns the derivative of the first.

6.2.3.2 Implementation in C++

Implementing Newton's method in C++.

```
#include <cassert>
#include <stdexcept>
#include <iostream>

double newtons_method( double  f( double x ),
                       double df( double x ),
                       double x,
                       double eps_step, double eps_abs,
                       unsigned int max_iterations ) {
    double fx{f(x)};
    double x_prev;

    if ( fx == 0.0 ) {
        std::clog << "Number of iterations: 0" << std::endl;
        return x;
    }

    unsigned int iterations{0};

    do {
        ++iterations;

        if ( iterations > max_iterations ) {
            throw std::range_error( "Exceeded the maximum number of iterations" );
        }

        x_prev = x;
        x = x - fx/df( x );
        fx = f(x);

        if ( fx == 0.0 ) {
            std::clog << "Number of iterations: " << iterations << std::endl;
            return x;
        }
    } while ( (std::abs(x – x_prev) >= eps_step)
           && (std::abs( fx ) >= eps_abs) );

    std::clog << "Number of iterations: " << iterations << std::endl;

    return x;
}
```

We can include in our test above one for Newton's method, including the lines

```
std::cout << "Newton's method" << std::endl;
std::cout << newtons_method( std::cos, dcos, 1.5, 1e-6, 1e-6, 100 ) << std::endl;
std::cout << newtons_method( y, dy, 0.5, 1e-6, 1e-6, 100 ) << std::endl;
```

The output is now

```
Number of iterations: 3
1.570796326794897
Number of iterations: 4
0.46
```

Note that both of these answers are much more accurate than the bisection method, but both

6.2.4 The secant method

Given a function $f(x)$, if you do not know the derivative, it is not possible to apply Newton's method, but suppose you have two approximations to the root x_0 and x_1. We will assume that x_1 is a better approximation of the root, meaning that $|f(x_1)| < |f(x_0)|$, and if this is not true, just swap the two values.

Finding a line that passes through these two points allows you to find a line that has a root of that interpolating line. That root could be a better approximation of the root:

$$x_2 \leftarrow \frac{x_0 f(x_1) - x_1 f(x_0)}{f(x_1) - f(x_0)}.$$

At this point, we can continue using x_1 and x_2. In general, we set

$$x_k \leftarrow \frac{x_{k-2} f(x_{k-1}) - x_{k-1} f(x_{k-2})}{f(x_{k-1}) - f(x_{k-2})},$$

and we continue iterating. Unlike the bracketed secant method, here we just continue using the last two approximations to find the next approximation. The issues here are the same as those for Newton's method: the sequence may not converge even if a root exists, whereas with the bracketed secant method, the method is guaranteed to converge.

To give two examples, consider the function $f(x) = x^2 - 10$. This has a root close to $x = 3$, so let us start with $x_0 = 3$ and $x_1 = 3.1$. We now iterate:

$$x_2 \leftarrow \frac{3 f(3.1) - 3.1 f(3)}{f(3.1) - f(3)} = 3.163934426229508.$$

Next, we continue with x_1 and x_2, to calculate

$$x_3 \leftarrow \frac{3.1 f(3.163934426229508) - 3.163934426229508 f(3.1)}{f(3.163934426229508) - f(3.1)} = 3.162261188170635$$

If we continue iterating, we get

$$\begin{aligned} x_4 &= 3.162277655854531 \\ x_5 &= 3.162277660168391 \\ x_6 &= 3.162277660168380 \\ x_7 &= 3.162277660168380 \end{aligned}$$

After this, if we were to apply one more step, the denominator would be zero, in which case x_8 would be calculated as a floating-point infinity; however, at this point, we can stop because the last two approximations are equal.

Now, one of the issues with the bracketed secant method, and that which forces it to execute in $O(h)$, is that as one end-point of the interval gets closer to the root, the other end-point remains fixed. In this case, both points are always the best approximations of the root, so the rate of convergence is faster: it is $O(h^\phi)$ where ϕ is the golden ratio $\phi = \frac{\sqrt{5}-1}{2} \approx 1.6180339887498950$. We will not prove this in this course.

6.2.5 Examples

We will look at how each of these methods works at finding a root of the following two functions:

1. $f(x) = x^2 - 2$
2. $y(t) = 6.535\, e^{-3.193t} \cos(1.842t) - 1.038\, e^{-3.193t} \sin(1.842t)$

The first function has a root on the interval [1, 2] and the second has a root on the interval [0, 1]. Plots of these functions are shown in .

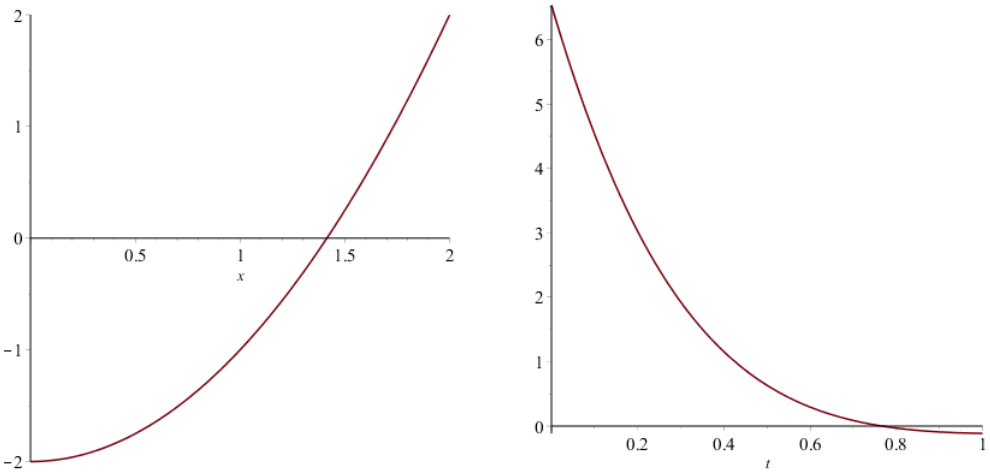


Figure 1. Plots of $f(x)$ on [1, 2] and $y(t)$ on [0, 1].

6.2.5.1 The bisection method

For the quadratic polynomial *f*, if we start with the interval [1, 2], with the bisection method, we get this sequence of approximations:

n	a_n	$f(a_n)$	b_n	$f(b_n)$	c_n	$f(c_n)$
	1	–1	2	2	1.5	0.25
1	1	–1	1.5	0.25	1.25	–0.4375
2	1.25	–0.4375	1.5	0.25	1.375	–0.1094
3	1.375	–0.1094	1.5	0.25	1.4375	0.06641
4	1.375	–0.1094	1.4375	0.06641	1.40625	–0.02246
5	1.40625	–0.02246	1.4375	0.06641	1.421875	0.02173
6	1.40625	–0.02246	1.421875	0.02173	1.4140625	–0.0004272
7	1.4140625	–0.0004272	1.421875	0.02173	1.41796875	0.01064
8	1.4140625	–0.0004272	1.41796875	0.01064	1.416015625	0.005100
9	1.4140625	–0.0004272	1.416015625	0.005100	1.4150390625	0.002336
10	1.4140625	–0.0004272	1.4150390625	0.002336	1.41455078125	0.0009539
11	1.4140625	–0.0004272	1.41455078125	0.0009539	1.414306640625	0.0002633
12	1.4140625	–0.0004272	1.414306640625	0.0002633	1.4141845703125	–0.00008200
13	1.4141845703125	–0.00008200	1.414306640625	0.0002633	1.41424560546875	0.00009063
14	1.4141845703125	–0.00008200	1.41424560546875	0.00009063	1.414215087890625	0.000004315
15	1.4141845703125	–0.00008200	1.414215087890625	0.000004315	1.414199829101562	–0.00003884
16	1.414199829101562	–0.00003884	1.414215087890625	0.000004315	1.414207458496094	–0.00001726
17	1.414207458496094	–0.00001726	1.414215087890625	0.000004315	1.414211273193360	–0.000006475
18	1.414211273193360	–0.000006475	1.414215087890625	0.000004315	1.414213180541992	–0.000001080
19	1.414213180541992	–0.000001080	1.414215087890625	0.000004315	1.414214134216308	0.000001617
20	1.414213180541992	–0.000001080	1.414214134216308	0.000001617		

Because the initial width of the interval was 1, after 20 iterations, the width of the interval is 2^{-20}, so the maximum error is 9.537 x 10^{-7}. As $|f(a)| < |f(b)|$ with our last step, we will approximate the root with x = **1.414213**180541992. The correct solution to 20 digits is **1.414213**5623730950488.

For the function *y*, if we start with the interval [0, 1], with the bisection method, we get this sequence of approximations:

n	a_n	$y(a_n)$	b_n	$y(b_n)$	c_n	$y(c_n)$
	0	6.535	1	–0.1129	0.5	0.6336
1	0.5	0.6336	1	–0.1129	0.75	0.01917
2	0.75	0.01917	1	–0.1129	0.875	–0.07983
3	0.75	0.01917	0.875	–0.07983	0.8125	–0.04115
4	0.75	0.01917	0.8125	–0.04115	0.78125	–0.01408
5	0.75	0.01917	0.78125	–-0.01408	0.765625	0.001719
6	0.765625	0.001719	0.78125	–0.01408	0.7734375	–0.006381
7	0.765625	0.001719	0.7734375	–0.006381	0.76953125	–0.002382
8	0.765625	0.001719	0.76953125	–0.002382	0.767578125	–0.0003444
9	0.765625	0.001719	0.767578125	–0.0003444	0.7666015625	0.0006839
10	0.7666015625	0.0006839	0.767578125	–0.0003444	0.76708984375	0.0001690
11	0.76708984375	0.0001690	0.767578125	–0.0003444	0.767333984375	–0.00008792
12	0.76708984375	0.0001690	0.767333984375	–0.00008792	0.7672119140625	0.00004048
13	0.7672119140625	0.00004048	0.767333984375	–0.00008792	0.76727294921875	–0.00002373
14	0.7672119140625	0.00004048	0.76727294921875	–0.00002373	0.767242431640625	0.000008367
15	0.767242431640625	0.000008367	0.76727294921875	–0.00002373	0.7672576904296875	–0.000007685
16	0.767242431640625	0.000008367	0.7672576904296875	–0.000007685	0.7672500610351563	0.0000003411
17	0.7672500610351563	0.0000003411	0.7672576904296875	–0.000007685	0.767253875732422	–0.000003672
18	0.7672500610351563	0.0000003411	0.767253875732422	–0.000003672	0.7672519683837892	–0.000001665
19	0.7672500610351563	0.0000003411	0.7672519683837892	–0.000001665	0.7672510147094728	–0.0000006621
20	0.7672500610351563	0.0000003411	0.7672510147094728	–0.0000006621		

Because the initial width of the interval was 1, after 20 iterations, the width of the interval is 2^{-20}, so the maximum error is 9.537 x 10^{-7}. As $|y(a)| < |y(b)|$ with our last step, we will approximate the root with t = **0.767250**0610351563. The correct solution to 20 digits is **0.767250**38526760903865

6.2.5.2 Bracketed secant method

For the quadratic polynomial *f*, if we start with the interval [1, 2], with the bracketed secant method, we get this sequence of approximations:

n	a_n	$f(a_n)$	b_n	$f(b_n)$	c_n	$f(c_n)$
	1	–1	2	2	1.333333333333333	-2.222×10^{-1}
1	1.333333333333333	-2.222×10^{-1}	2	2	1.4	-4.000×10^{-2}
2	1.4	-4.000×10^{-2}	2	2	1.411764705882353	-6.920×10^{-3}
3	1.411764705882353	-6.920×10^{-3}	2	2	1.413793103448276	-1.189×10^{-3}
4	1.413793103448276	-1.189×10^{-3}	2	2	1.414141414141414	-2.041×10^{-4}
5	1.414141414141414	-2.041×10^{-4}	2	2	1.414201183431953	-3.501×10^{-5}
6	1.414201183431953	-3.501×10^{-5}	2	2	1.414211438474870	-6.007×10^{-6}
7	1.414211438474870	-6.007×10^{-6}	2	2	1.414213197969543	-1.031×10^{-6}
8	1.414213197969543	-1.031×10^{-6}	2	2	1.414213499851323	-1.768×10^{-7}
9	1.414213499851323	-1.768×10^{-7}	2	2	1.414213551646055	-3.034×10^{-8}
10	1.414213551646055	-3.034×10^{-8}	2	2	1.414213560532626	-5.206×10^{-9}
11	1.414213560532626	-5.206×10^{-9}	2	2	1.414213562057320	-8.931×10^{-10}
12	1.414213562057320	-8.931×10^{-10}	2	2	1.414213562318917	-1.532×10^{-10}
13	1.414213562318917	-1.532×10^{-10}	2	2	1.414213562363800	-2.629×10^{-11}
14	1.414213562363800	-2.629×10^{-11}	2	2	1.414213562371500	-4.511×10^{-12}
15	1.414213562371500	-4.511×10^{-12}	2	2	1.414213562372821	-7.740×10^{-13}
16	1.414213562372821	-7.740×10^{-13}	2	2	1.414213562373048	-1.328×10^{-13}
17	1.414213562373048	-1.328×10^{-13}	2	2	1.414213562373087	-2.278×10^{-14}
18	1.414213562373087	-2.278×10^{-14}	2	2	1.414213562373094	-3.909×10^{-15}
19	1.414213562373094	-3.909×10^{-15}	2	2	1.414213562373095	-6.707×10^{-16}
20	1.414213562373095	-6.707×10^{-16}	2	2		

Notice that because the function is concave up on the interval [1, 2] and $f(b) > 0$, the value b does not change. After 20 iterations, we will approximate the root with $x = 1.414213562373095$, which is correct to all significant digits. The correct solution to 20 digits is 1.4142135623730950488.

For the function y, if we start with the interval [0, 1], with the bracketed secant method, we get this sequence of approximations:

n	a_n	$y(a_n)$	b_n	$y(b_n)$	c_n	$y(c_n)$
	0	6.535	1	-1.129×10^{-1}	0.9830152048905029	-1.110×10^{-1}
1	0	6.535	0.9830152048905029	-1.110×10^{-1}	0.9665986010737283	-1.085×10^{-1}
2	0	6.535	0.9665986010737283	-1.085×10^{-1}	0.9508145012780233	-1.054×10^{-1}
3	0	6.535	0.9508145012780233	-1.054×10^{-1}	0.9357196149741421	-1.018×10^{-1}
4	0	6.535	0.9357196149741421	-1.018×10^{-1}	0.9213616313420384	-9.779×10^{-2}
5	0	6.535	0.9213616313420384	-9.779×10^{-2}	0.9077781438292655	-9.333×10^{-2}
6	0	6.535	0.9077781438292655	-9.333×10^{-2}	0.8949959773563069	-8.855×10^{-2}
7	0	6.535	0.8949959773563069	-8.855×10^{-2}	0.8830309474295765	-8.352×10^{-2}
8	0	6.535	0.8830309474295765	-8.352×10^{-2}	0.8718880464174520	-7.832×10^{-2}
9	0	6.535	0.8718880464174520	-7.832×10^{-2}	0.8615620204076564	-7.305×10^{-2}
10	0	6.535	0.8615620204076564	-7.305×10^{-2}	0.8520382741857717	-6.776×10^{-2}
11	0	6.535	0.8520382741857717	-6.776×10^{-2}	0.8432940245117312	-6.254×10^{-2}
12	0	6.535	0.8432940245117312	-6.254×10^{-2}	0.8352996140314371	-5.745×10^{-2}
13	0	6.535	0.8352996140314371	-5.745×10^{-2}	0.8280198993490837	-5.254×10^{-2}
14	0	6.535	0.8280198993490837	-5.254×10^{-2}	0.8214156353445713	-4.785×10^{-2}
15	0	6.535	0.8214156353445713	-4.785×10^{-2}	0.8154447914279704	-4.341×10^{-2}
16	0	6.535	0.8154447914279704	-4.341×10^{-2}	0.8100637516059210	-3.924×10^{-2}
17	0	6.535	0.8100637516059210	-3.924×10^{-2}	0.8052283667804445	-3.536×10^{-2}
18	0	6.535	0.8052283667804445	-3.536×10^{-2}	0.8008948429125466	-3.177×10^{-2}
19	0	6.535	0.8008948429125466	-3.177×10^{-2}	0.7970204614756886	-2.846×10^{-2}
20	0	6.535	0.7970204614756886	-2.846×10^{-2}		

Notice that because the function is concave up on the interval [0, 1] and $y(a) > 0$, the value a does not change. After 20 iterations, we will approximate the root with $t = 0.7970204614756886$, which unfortunately only correct to the first digit, as the correct solution to 20 digits is 0.76725038526760903865.

This is because in this example, $\frac{y^{(1)}(r)}{y(b)\big/(r-b)} \approx 0.1235$, so with each iteration, the error is only reduced by a factor of 0.8765 times the previous error—making convergence much worse than even the bisection method.

If we alternate between the bisection method and the bracketed false position method, we note that we get significantly faster convergence.

6.2.5.3 Alternating between bisection and the bracketed secant methods

For the quadratic polynomial *f*, if we start with the interval [1, 2] and then alternate applying the bisection method and then the bracketed secant method, we get this sequence of approximations:

n	a_n	$f(a_n)$	b_n	$f(b_n)$	c_n	$f(c_n)$
	1	-1	2	2	1.5	2.500×10^{-1}
1	1	-1	1.5	2.500×10^{-1}	1.4	-4.000×10^{-2}
2	1.4	-4.000×10^{-2}	1.5	2.500×10^{-1}	1.45	1.025×10^{-1}
3	1.4	-4.000×10^{-2}	1.45	1.025×10^{-1}	1.414035087719298	-5.048×10^{-4}
4	1.414035087719298	-5.048×10^{-4}	1.45	1.025×10^{-1}	1.432017543859649	5.067×10^{-2}
5	1.414035087719298	-5.048×10^{-4}	1.432017543859649	5.067×10^{-2}	1.414212445893591	-3.158×10^{-6}
6	1.414212445893591	-3.158×10^{-6}	1.432017543859649	5.067×10^{-2}	1.423114994876620	2.526×10^{-2}
7	1.414212445893591	-3.158×10^{-6}	1.423114994876620	2.526×10^{-2}	1.414213558870409	-9.907×10^{-9}
8	1.414213558870409	-9.907×10^{-9}	1.423114994876620	2.526×10^{-2}	1.418664276873514	1.261×10^{-2}
9	1.414213558870409	-9.907×10^{-9}	1.418664276873514	1.261×10^{-2}	1.414213562367592	-1.556×10^{-11}
10	1.414213562367592	-1.556×10^{-11}	1.418664276873514	1.261×10^{-2}	1.416438919620553	6.299×10^{-3}
11	1.414213562367592	-1.556×10^{-11}	1.416438919620553	6.299×10^{-3}	1.414213562373091	-1.224×10^{-14}
12	1.414213562373091	-1.224×10^{-14}	1.416438919620553	6.299×10^{-3}	1.415326240996822	3.148×10^{-3}
13	1.414213562373091	-1.224×10^{-14}	1.415326240996822	3.148×10^{-3}	1.414213562373095	-4.812×10^{-18}
14	1.414213562373095	-4.812×10^{-18}	1.415326240996822	3.148×10^{-3}		

Notice that because the function is concave up on the interval [1, 2] and $f(b) > 0$, the value *b* does not change when the bracketed secant method is used. After 14 iterations, we will approximate the root with $x = 1.414213562373095$, which is correct to all significant digits. The correct solution to 20 digits is 1.4142135623730950488.

For the function *y*, if we start with the interval [0, 1], with the bracketed secant method, we get this sequence of approximations:

n	a_n	$y(a_n)$	b_n	$y(b_n)$	c_n	$y(c_n)$
	0	6.535	1	-1.129×10^{-1}	0.5	6.336×10^{-1}
1	0.5	6.336×10^{-1}	1	-1.129×10^{-1}	0.9243747240645564	-9.869×10^{-2}
2	0.5	6.336×10^{-1}	0.9243747240645564	-9.869×10^{-2}	0.7121873620322782	6.894×10^{-2}
3	0.7121873620322782	6.894×10^{-2}	0.9243747240645564	-9.869×10^{-2}	0.7994514977621643	-3.055×10^{-2}
4	0.7121873620322782	6.894×10^{-2}	0.7994514977621643	-3.055×10^{-2}	0.7558194298972212	1.247×10^{-2}
5	0.7558194298972212	1.247×10^{-2}	0.7994514977621643	-3.055×10^{-2}	0.7684685780154039	-1.277×10^{-3}
6	0.7558194298972212	1.247×10^{-2}	0.7684685780154039	-1.277×10^{-3}	0.7621440039563125	5.460×10^{-3}
7	0.7621440039563125	5.460×10^{-3}	0.7684685780154039	-1.277×10^{-3}	0.7672701107964198	-2.075×10^{-5}
8	0.7621440039563125	5.460×10^{-3}	0.7672701107964198	-2.075×10^{-5}	0.7647070573763662	2.697×10^{-3}
9	0.7647070573763662	2.697×10^{-3}	0.7672701107964198	-2.075×10^{-5}	0.7672505447408864	-1.678×10^{-7}
10	0.7647070573763662	2.697×10^{-3}	0.7672505447408864	-1.678×10^{-7}	0.7659788010586263	1.343×10^{-3}
11	0.7659788010586263	1.343×10^{-3}	0.7672505447408864	-1.678×10^{-7}	0.7672503859136397	-6.796×10^{-10}
12	0.7659788010586263	1.343×10^{-3}	0.7672503859136397	-6.796×10^{-10}	0.7666145934861330	6.702×10^{-4}
13	0.7666145934861330	6.702×10^{-4}	0.7672503859136397	-6.796×10^{-10}	0.7672503852689191	-1.378×10^{-12}
14	0.7666145934861330	6.702×10^{-4}	0.7672503852689191	-1.378×10^{-12}	0.7669324893775260	3.348×10^{-4}
15	0.7669324893775260	3.348×10^{-4}	0.7672503852689191	-1.378×10^{-12}	0.7672503852676104	-1.398×10^{-15}
16	0.7669324893775260	3.348×10^{-4}	0.7672503852676104	-1.398×10^{-15}	0.7670914373225682	1.673×10^{-4}
17	0.7670914373225682	1.673×10^{-4}	0.7672503852676104	-1.398×10^{-15}	0.7672503852676090	-7.093×10^{-19}
18	0.7670914373225682	1.673×10^{-4}	0.7672503852676090	-7.093×10^{-19}		

Notice that because the function is concave up on the interval [0, 1] and $y(a) > 0$, the value *a* does not change when the bracketed secant method is used, but does change when the bisection method is used. After 18 iterations, we will approximate the root with $t = \mathbf{0.7672503852676090}$, which is correct to all 16 significant digits, as the correct answer to twenty digits is **0.7672503852676090**3865.

6.2.5.4 Newton's method

For the quadratic polynomial *f*, if we start with $x_0 = 1$ with Newton's method, we get the sequence of approximations

n	Approximation
0	1.0
1	1.500000000000000
2	1.416666666666667
3	1.414215686274510
4	1.414213562374690
5	1.414213562373095

Thus, Newton's method converges after five iterations. The correct solution to 20 digits is 1.4142135623730950488.

For the transcendental function *y*, if we start with $t_0 = 0$ with Newon's method, we get the sequence of approximations

n	Approximation
0	0
1	0.2868964785751110
2	0.4996228340749413
3	0.6506384365763257
4	0.6506384365763257
5	0.7365774672714292
6	0.7645413980528708
7	0.7672271761263750
8	0.7672503835477956
9	0.7672503852676093
10	0.7672503852676092
11	0.7672503852676091
12	0.7672503852676090
13	0.7672503852676089
14	0.7672503852676088
15	0.7672503852676093

Thus, Newton's method iterates comes very close to the root after only nine iterations, but then cycles between six different values. The correct solution to 20 digits is 0.76725038526760903865, so t_{12} is the most accurate.

6.2.5.5 The secant method

For the quadratic polynomial f, if we start with the approximations $x_0 = 2$ and $x_1 = 1.5$ and then apply the secant method, we get the sequence of approximations where the last approximation is correct to all digits.

n	Approximation
0	2.0
1	1.5
2	1.428571428571429
3	1.414634146341463
4	1.414215686274510
5	1.414213562688870
6	1.414213562373095
7	1.414213562373095

The last two approximations are equal, so we are finished, and we approximate the root with the point $x = 1.414213562373095$, which is correct to all digits.

For the function y, if we start with the approximations $t_0 = 0$ and $t_1 = 0.5$ and then apply the secant method, we get this sequence of approximations:

n	Approximation
0	0
1	0.5
2	0.5536839616574416
3	0.6694972590725908
4	0.7246973534700393
5	0.7566421605706841
6	0.7659343609517854
7	0.7672067353503918
8	0.7672502022893107
9	0.7672503852421088
10	0.7672503852676093
11	0.7672503852676091
12	∞

The last value occurs as calculating $y(t_{11}) - y(t_{10})$ results in a 0 in the denominator, causing the calculation to return a floating-point infinity. Thus, we would use $t_{11} = 0.7672503852676091$ to approximate the root of the function.

6.2.6 Summary of these methods

The following table offers a summary of these four methods:

Method	Requirements	Iteration step	Rate of convergence	Is convergence guaranteed?
Bisection	An interval $[a, b]$ with $f(a)$ having the opposite sign of $f(b)$	Let $c \leftarrow \frac{a+b}{2}$ and update whichever endpoint has the same sign as $f(c)$.	$O(h)$	Yes
Bracketed secant	An interval $[a, b]$ with $f(a)$ having the opposite sign of $f(b)$	Let $c \leftarrow \frac{af(b)-bf(a)}{f(b)-f(a)}$ and update whichever endpoint has the same sign as $f(c)$.	$O(h)$	Yes
Newton's	An initial approximation x_0	Let $x_k \leftarrow x_{k-1} - \frac{f(x_{k-1})}{f^{(1)}(x_{k-1})}$.	$O(h^2)$	No
Secant	Two initial approximations x_0 and x_1 with $\|f(x_0)\| > \|f(x_1)\|$	Let $x_k \leftarrow \frac{x_{k-2}f(x_{k-1})-x_{k-1}f(x_{k-2})}{f(x_{k-1})-f(x_{k-2})}$.	$O(h^{\phi})$	No

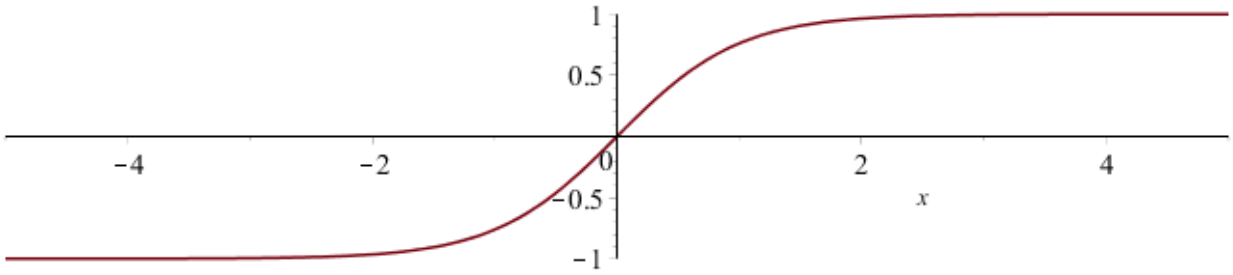
6.2.7 Problems

1. Apply one step of these four methods approximate a root of the function $f(x) = x^3 - 9$ using reasonable initial intervals or points.

2. For the bisection and secant methods, given an interval $[a, b]$ that is small enough, we return a or b depending on whether $|f(a)| < |f(b)|$ or $|f(a)| > |f(b)|$. Why is this choice reasonable? Draw a situation where a continuous function may never-the-less have the point we choose be the sub-optimal choice.

3. For the secant and Newton's method, explain why the technique may not converge (two reasons).

4. Here is a plot of the hyperbolic tangent function (tanh). If you start with $x_0 = 1$, Newton's method converges to the root, but if you start with $x_0 = 1.5$, the next two approximations of the root using Newton's method are –3.508937463704951 and 275.5937484459173, so it is clearly not converging.

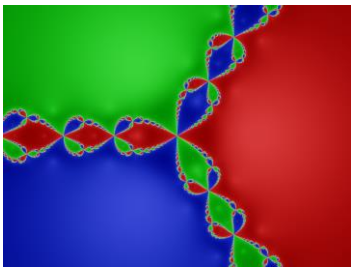


On which interval must the initial point for Newton's method be in order to converge to zero? What expression would you have to solve to find the end points of this interval? What happens if you start with one of the end-points of this interval as x_0? You will note that once you get the equation, find a solution close to $x = 1.1$ by using two steps of Newton's method. Recall that the derivative of $\tanh(x)$ is $1 - \tanh^2(x)$.

5. We described six tools we will use for approximating solutions to various problems. What are the tools used for each of the bisection method, the bracketed secant method, Newton's method and the secant method.

6.2.8 Newton's method and fractals (aside)

Normally, you would expect Newton's method to converge to the root closest to the starting point. This is, however, not the case. If you start with an x_0 that is in the complex plane and try to find a root of the polynomial $z^3 - 1$, you know that the roots are at $1, -\frac{1}{2} \pm j\frac{\sqrt{3}}{2}$. However, the following image shows all points that converge to the root at 1 in red, all points that ultimately converge to the root $-\frac{1}{2} + j\frac{\sqrt{3}}{2}$ in green, and all points that converge to the root $-\frac{1}{2} - j\frac{\sqrt{3}}{2}$ in blue. Nowhere is the boundary between these regions well defined—you can zoom in arbitrarily close, and the boundary continues to be as detailed as you see in this image.



6.3 Systems of non-linear equations

Returning to solving a system of one non-linear equation, observe that the first-order Taylor series around the point x_k is $f(x_k)+(x-x_k)f^{(1)}(x_k)$. This is an equation of the tangent plane at the point x_k. If we let $\Delta x_k = x - x_k$, this becomes $f(x_k)+\Delta x_k f^{(1)}(x_k)$, so we can find a solution to this by equating it to zero $f(x_k)+\Delta x_k f^{(1)}(x_k)=0$, and finding a solution to this is $\Delta x_k = -\dfrac{f(x_k)}{f^{(1)}(x_k)}$. Substituting this back into our definition of $\Delta x_k = x - x_k$, we have $x = x_k + \Delta x_k$; that is, $x = x_k + \left(-\dfrac{f(x_k)}{f^{(1)}(x_k)}\right)$. This is the next approximation to the root, and we call this x_{k+1}.

Now, suppose we have a system of two non-linear equations and two unknowns

$$f(x,y)=0$$
$$g(x,y)=0$$

and we have one approximation to a root (x_k, y_k). We can now write a Taylor series at this point:

$$f(x_k,y_k)+(x-x_k)\frac{\partial}{\partial x}f(x_k,y_k)+(y-y_k)\frac{\partial}{\partial y}f(x_k,y_k)$$
$$g(x_k,y_k)+(x-x_k)\frac{\partial}{\partial x}g(x_k,y_k)+(y-y_k)\frac{\partial}{\partial y}g(x_k,y_k)$$

These are tangent planes at the point (x_k, y_k). Now, let us represent $\Delta x_k = x - x_k$ and $\Delta y_k = y - y_k$, we have

$$f(x_k,y_k)+\Delta x_k\frac{\partial}{\partial x}f(x_k,y_k)+\Delta y_k\frac{\partial}{\partial y}f(x_k,y_k)$$
$$g(x_k,y_k)+\Delta x_k\frac{\partial}{\partial x}g(x_k,y_k)+\Delta y_k\frac{\partial}{\partial y}g(x_k,y_k)$$

We can now equate these to zero, and we have

$$\Delta x_k\frac{\partial}{\partial x}f(x_k,y_k)+\Delta y_k\frac{\partial}{\partial y}f(x_k,y_k)=-f(x_k,y_k)$$
$$\Delta x_k\frac{\partial}{\partial x}g(x_k,y_k)+\Delta y_k\frac{\partial}{\partial y}g(x_k,y_k)=-g(x_k,y_k)$$

This is a system of two linear equations and two unknowns:

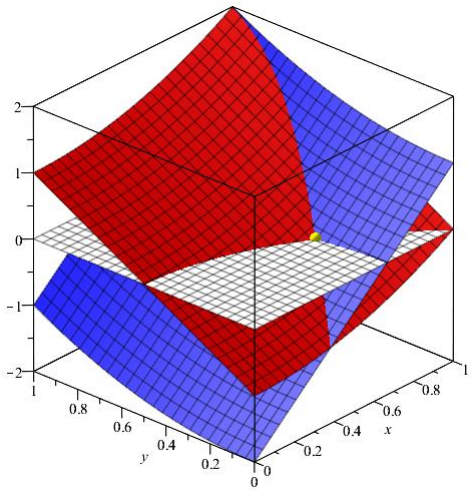
$$\begin{pmatrix}\dfrac{\partial}{\partial x}f(x_k,y_k) & \dfrac{\partial}{\partial y}f(x_k,y_k)\\ \dfrac{\partial}{\partial x}g(x_k,y_k) & \dfrac{\partial}{\partial y}g(x_k,y_k)\end{pmatrix}\begin{pmatrix}\Delta x_k\\ \Delta y_k\end{pmatrix}=\begin{pmatrix}-f(x_k,y_k)\\ -g(x_k,y_k)\end{pmatrix}.$$

Once you solve this for the unknown vector $\begin{pmatrix}\Delta x_k\\ \Delta y_k\end{pmatrix}$, you can then update $x = x_k + \Delta x_k$ and $y = y_k + \Delta y_k$. This will be our next approximation to the root.

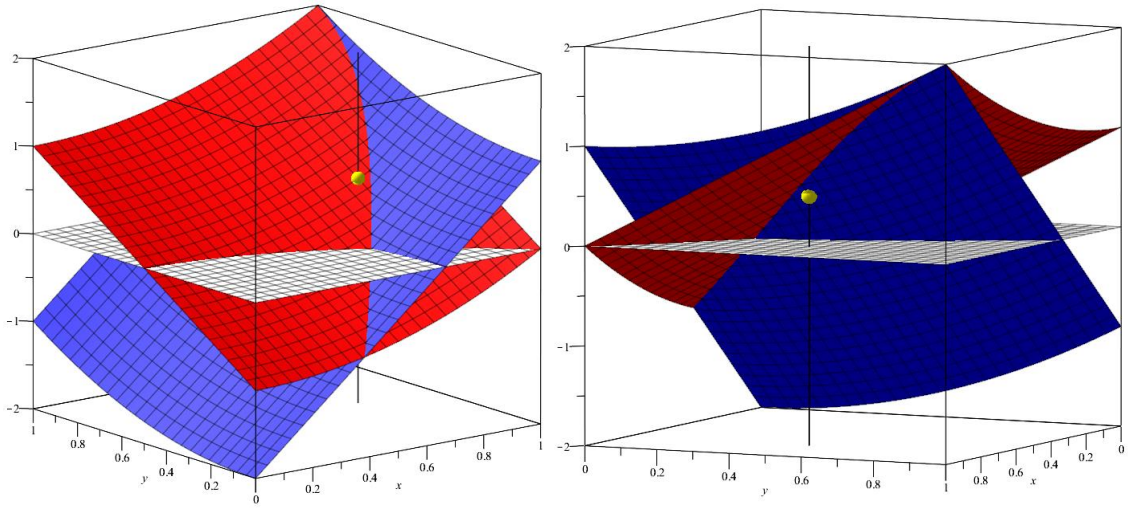
For example, the two functions $x^2 + 2y - 1$ and $3x + y^2 - 2$ have a simultaneous root at the point

$$(x, y) = (0.6372755591552685, 0.2969399308516699)$$

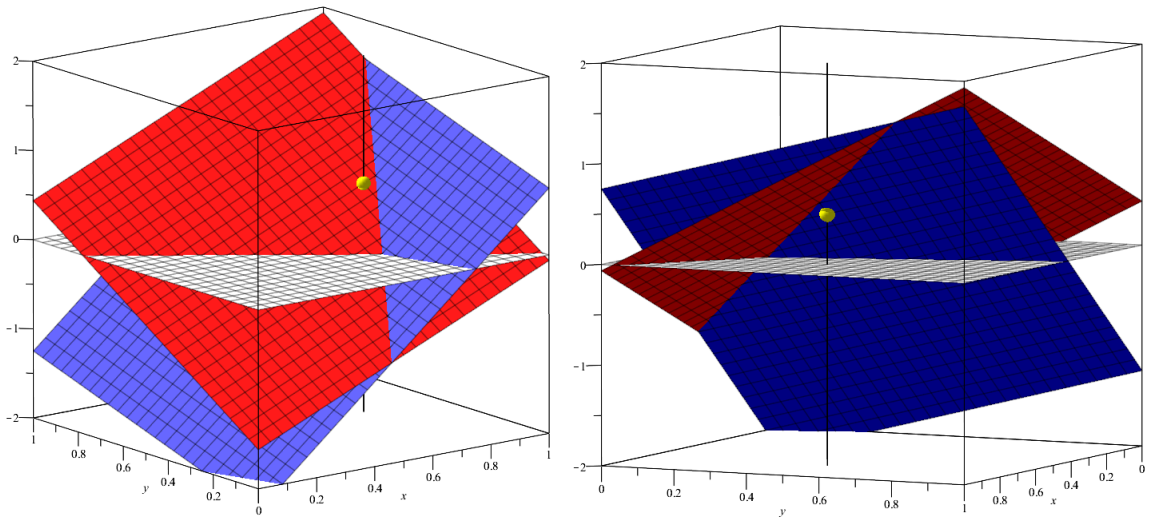
This root is shown as a yellow point in this graph of these two functions, with the first function shown in red and the second in blue.



Suppose we have the approximation $(x, y) = (0.75, 0.5)$ to this root. Neither function is zero at this point, for $0.75^2 + 2 \cdot 0.5 - 1 = 0.5625$ and $3 \cdot 0.75 + 0.5^2 - 2 = 0.5$. Viewing both functions, we see that at this point, neither function is zero:



We can find a tangent plan at these points on both planes:



As you can see on both these planes, there is a simultaneous root of these two planes. These two planes are defined by taking the Taylor series at the point $(x, y) = (0.75, 0.5)$:

$$2\cdot 0.75(x - 0.75) + 2(y - 0.5) + 0.5625$$
$$3(x - 0.75) + 2\cdot 0.5(y - 0.5) + 0.5$$

We can now substitute

$$2\cdot 0.75\Delta x + 2\Delta y + 0.5625 = 0$$
$$3\Delta x + 2\cdot 0.5\Delta y + 0.5 = 0$$

and this defines a system of two equations and two unknowns $\begin{pmatrix} 1.5 & 2 \\ 3 & 1 \end{pmatrix}\begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = \begin{pmatrix} -0.5625 \\ -0.5 \end{pmatrix}$. Solving this, we get

$$\Delta x = -0.09722222222222222,\ \Delta y = -0.2083333333333333$$

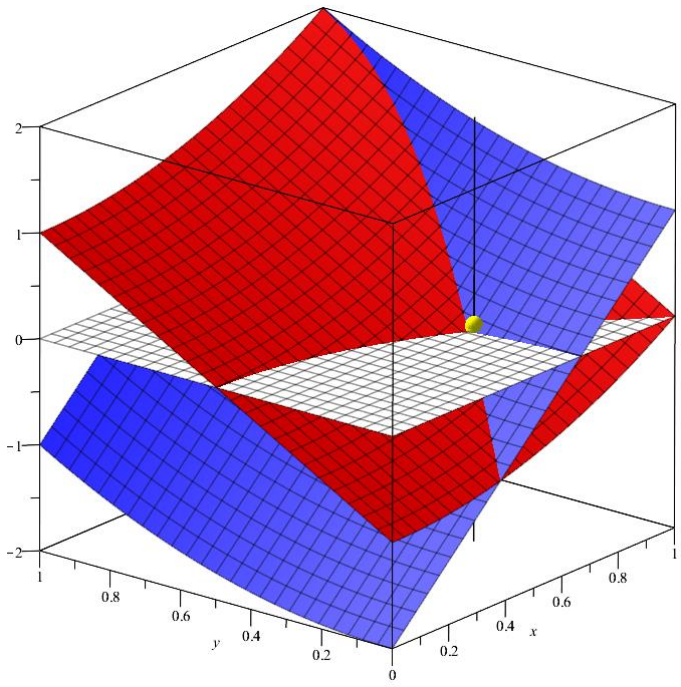
Our next approximation for the root is therefore

$$0.75 + \Delta x = 0.75 - 0.09722222222222222,\ 0.5 + \Delta y = 0.5 - 0.2083333333333333$$

Thus, our next approximation is (0.6527777777777778, 0.2916666666666667). Note that this is significantly closer to the actual root (0.6372755591552685, 0.2969399308516699). The two functions evaluated at this point are much closer to zero:

$$0.009452160493827160,\ 0.04340277777777778$$

Compare and contrast these with 0.5625 and 0.5, and you can also see the point in the next image:



k	$\mathbf{x}_k$	$\mathbf{f}(\mathbf{x}_k)$	$\Delta\mathbf{x}_k$
0	$\begin{pmatrix} 0.75 \\ 0.5 \end{pmatrix}$	$\begin{pmatrix} 0.5625 \\ 0.5 \end{pmatrix}$	$\begin{pmatrix} -0.09722222222222222 \\ -0.2803333333333333 \end{pmatrix}$
1	$\begin{pmatrix} 0.6527777777777778 \\ 0.2916666666666667 \end{pmatrix}$	$\begin{pmatrix} 0.009452 \\ 0.04340 \end{pmatrix}$	$\begin{pmatrix} -0.01551836303824815 \\ -0.005403962291942849 \end{pmatrix}$
2	$\begin{pmatrix} 0.6372594147395296 \\ 0.2970706289586095 \end{pmatrix}$	$\begin{pmatrix} 0.0002408 \\ 0.00002920 \end{pmatrix}$	$\begin{pmatrix} -0.00001615090261970687 \\ -0.0001307021104443827 \end{pmatrix}$
3	$\begin{pmatrix} 0.6372755656421493 \\ 0.2969399268481651 \end{pmatrix}$	$\begin{pmatrix} 0.0000000002609 \\ 0.00000001708 \end{pmatrix}$	

Thus, with each step, you can see that we are approaching a simultaneous root of both of these functions with the actual root being at $\begin{pmatrix} 0.6372755591552685 \\ 0.2969399308516699 \end{pmatrix}$.

Now, assume we have a system of *n* non-linear equations in *n* unknowns, and we want to find a simultaneous root to all of these. In this case, let us use vector notation:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

and thus we may write the *n* non-linear functions as

$$\mathbf{f}(\mathbf{x}) = \begin{pmatrix} f_1(\mathbf{x}) \\ f_2(\mathbf{x}) \\ \vdots \\ f_n(\mathbf{x}) \end{pmatrix}.$$

We want to find where $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ where $\mathbf{0} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$. We now want to find a Taylor series at an initial point $\mathbf{x}_k$ for each of the *n* functions:

$$\begin{aligned}
&f_1(\mathbf{x}_k) + \frac{\partial}{\partial x_1} f_1(\mathbf{x}_k)(x_k - x_{k,1}) + \frac{\partial}{\partial x_2} f_1(\mathbf{x}_k)(x_k - x_{k,2}) + \frac{\partial}{\partial x_3} f_1(\mathbf{x}_k)(x_k - x_{k,3}) + \cdots + \frac{\partial}{\partial x_n} f_1(\mathbf{x}_k)(x_k - x_{k,n}) \\
&f_2(\mathbf{x}_k) + \frac{\partial}{\partial x_1} f_2(\mathbf{x}_k)(x_k - x_{k,1}) + \frac{\partial}{\partial x_2} f_2(\mathbf{x}_k)(x_k - x_{k,2}) + \frac{\partial}{\partial x_3} f_2(\mathbf{x}_k)(x_k - x_{k,3}) + \cdots + \frac{\partial}{\partial x_n} f_2(\mathbf{x}_k)(x_k - x_{k,n}) \\
&f_3(\mathbf{x}_k) + \frac{\partial}{\partial x_1} f_3(\mathbf{x}_k)(x_k - x_{k,1}) + \frac{\partial}{\partial x_2} f_3(\mathbf{x}_k)(x_k - x_{k,2}) + \frac{\partial}{\partial x_3} f_3(\mathbf{x}_k)(x_k - x_{k,3}) + \cdots + \frac{\partial}{\partial x_n} f_3(\mathbf{x}_k)(x_k - x_{k,n}) \\
&\qquad\qquad \vdots \\
&f_n(\mathbf{x}_k) + \frac{\partial}{\partial x_1} f_n(\mathbf{x}_k)(x_k - x_{k,1}) + \frac{\partial}{\partial x_2} f_n(\mathbf{x}_k)(x_k - x_{k,2}) + \frac{\partial}{\partial x_3} f_n(\mathbf{x}_k)(x_k - x_{k,3}) + \cdots + \frac{\partial}{\partial x_n} f_n(\mathbf{x}_k)(x_k - x_{k,n})
\end{aligned}$$

We can thus

$$\begin{aligned}
&f_1(\mathbf{x}_k) + \frac{\partial}{\partial x_1} f_1(\mathbf{x}_k)\Delta x_{k,1} + \frac{\partial}{\partial x_2} f_1(\mathbf{x}_k)\Delta x_{k,2} + \frac{\partial}{\partial x_3} f_1(\mathbf{x}_k)\Delta x_{k,3} + \cdots + \frac{\partial}{\partial x_n} f_1(\mathbf{x}_k)\Delta x_{k,n} = 0 \\
&f_2(\mathbf{x}_k) + \frac{\partial}{\partial x_1} f_2(\mathbf{x}_k)\Delta x_{k,1} + \frac{\partial}{\partial x_2} f_2(\mathbf{x}_k)\Delta x_{k,2} + \frac{\partial}{\partial x_3} f_2(\mathbf{x}_k)\Delta x_{k,3} + \cdots + \frac{\partial}{\partial x_n} f_2(\mathbf{x}_k)\Delta x_{k,n} = 0 \\
&f_3(\mathbf{x}_k) + \frac{\partial}{\partial x_1} f_3(\mathbf{x}_k)\Delta x_{k,1} + \frac{\partial}{\partial x_2} f_3(\mathbf{x}_k)\Delta x_{k,2} + \frac{\partial}{\partial x_3} f_3(\mathbf{x}_k)\Delta x_{k,3} + \cdots + \frac{\partial}{\partial x_n} f_3(\mathbf{x}_k)\Delta x_{k,n} = 0 \\
&\qquad\qquad \vdots \\
&f_n(\mathbf{x}_k) + \frac{\partial}{\partial x_1} f_n(\mathbf{x}_k)\Delta x_{k,1} + \frac{\partial}{\partial x_2} f_n(\mathbf{x}_k)\Delta x_{k,2} + \frac{\partial}{\partial x_3} f_n(\mathbf{x}_k)\Delta x_{k,3} + \cdots + \frac{\partial}{\partial x_n} f_n(\mathbf{x}_k)\Delta x_{k,n} = 0
\end{aligned}$$

This is a system of linear equations:

$$\begin{pmatrix} \frac{\partial}{\partial x_1} f_1(\mathbf{x}_k) & \frac{\partial}{\partial x_2} f_1(\mathbf{x}_k) & \frac{\partial}{\partial x_3} f_1(\mathbf{x}_k) & \cdots & \frac{\partial}{\partial x_n} f_1(\mathbf{x}_k) \\ \frac{\partial}{\partial x_1} f_2(\mathbf{x}_k) & \frac{\partial}{\partial x_2} f_2(\mathbf{x}_k) & \frac{\partial}{\partial x_3} f_2(\mathbf{x}_k) & \cdots & \frac{\partial}{\partial x_n} f_2(\mathbf{x}_k) \\ \frac{\partial}{\partial x_1} f_3(\mathbf{x}_k) & \frac{\partial}{\partial x_2} f_3(\mathbf{x}_k) & \frac{\partial}{\partial x_3} f_3(\mathbf{x}_k) & \cdots & \frac{\partial}{\partial x_n} f_3(\mathbf{x}_k) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial}{\partial x_1} f_n(\mathbf{x}_k) & \frac{\partial}{\partial x_2} f_n(\mathbf{x}_k) & \frac{\partial}{\partial x_3} f_n(\mathbf{x}_k) & \cdots & \frac{\partial}{\partial x_n} f_n(\mathbf{x}_k) \end{pmatrix} \begin{pmatrix} \Delta x_{k,1} \\ \Delta x_{k,2} \\ \Delta x_{k,3} \\ \vdots \\ \Delta x_{k,n} \end{pmatrix} = \begin{pmatrix} -f_1(\mathbf{x}_k) \\ -f_2(\mathbf{x}_k) \\ -f_3(\mathbf{x}_k) \\ \vdots \\ -f_n(\mathbf{x}_k) \end{pmatrix}.$$

We write this as $\mathbf{J}(\mathbf{f})(\mathbf{x}_k)\Delta\mathbf{x}_k = \frac{\mathrm{d}\mathbf{f}}{\mathrm{d}\mathbf{x}}(\mathbf{x}_k)\Delta\mathbf{x}_k = -\mathbf{f}(\mathbf{x}_k)$. We solve for $\Delta\mathbf{x}_k$ and then set $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \Delta\mathbf{x}_k$.

6.3.1 Problems

1. Find a simultaneous root of the two equations $x^2 + 2y - 1$ and $3x + y^2 - 2$ but now start with the two different points: $x_0 = -2.5$ and $y_0 = -3$ by applying one iteration of Newton's method in two dimensions.

2. Find a simultaneous root of the two equations $x^3 + 2y - 1$ and $3x + y^3 - 2$ starting with each of the following pairs of initial points:

$$\begin{aligned} x_0 &= \ \ 0.7, y_0 = \ \ 0.4 \\ x_0 &= -1.4, y_0 = \ \ 1.8 \\ x_0 &= \ \ 1.6, y_0 = -1.4 \end{aligned}$$

and in each case, apply one iteration of Newton's method. You only need to set up the system of linear equations that must be solved, and you must know how to use that solution to find the points x_1 and y_1.

As an aside: if you would like to solve a system of two equations and two unknowns, you can use Matlab or the GNU equivalent Octave. Go to the web site https://octave-online.net/ and to solve the system of $\begin{pmatrix} 3.2 & -1.5 \\ 1.5 & 4.7 \end{pmatrix}\begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = \begin{pmatrix} 9.6 \\ -8.0 \end{pmatrix}$, enter the statements:

```
octave:1> format long          % print 16 digits of precision
octave:2> [3.2 -1.5; 1.5 4.7] \ [9.6 -8.0]'
```

3. What is the tangent plane to the function sin(x)cos(y) at the point (x, y) = (1, 1)?

4. Starting with (x_0, y_0) = (1.5, 1.5), find a better approximation to the simultaneous root of the two expressions sin(x)cos(y) and cos(x)sin(y).

2019-02-05

x

Acknowledgments

Appendix A: Statistical error

In this course, we discuss numerical error; however, in future courses and throughout your career, you will also discuss errors associated with measurements. Let us look at solving a linear equation $ax = b$. If there are errors in both a and b, how does this affect the solution?

Suppose that $a \pm \sigma_a$ and $b \pm \sigma_b$ represents one standard deviation. In this case, the solution is

$$\begin{aligned}
\frac{b}{a} \pm \sqrt{\left(\sigma_a \frac{\partial}{\partial a}\frac{b}{a}\right)^2 + \left(\sigma_b \frac{\partial}{\partial b}\frac{b}{a}\right)^2} &= \frac{b}{a} \pm \sqrt{\left(\frac{b\sigma_a}{a^2}\right)^2 + \left(\frac{\sigma_b}{a}\right)^2} \\
&= \frac{b}{a} \pm \sqrt{\frac{b^2\sigma_a^2}{a^4} + \frac{\sigma_b^2}{a^2}} \\
&= \frac{b}{a} \pm \sqrt{\frac{b^2}{a^2}\frac{\sigma_a^2}{a^2} + \frac{b^2}{a^2}\frac{\sigma_b^2}{b^2}} \\
&= \frac{b}{a} \pm \left|\frac{b}{a}\right| \sqrt{\left(\frac{\sigma_a}{a}\right)^2 + \left(\frac{\sigma_b}{b}\right)^2}
\end{aligned}$$

Thus, each relative error has a proportional effect on the total error.

For example, suppose that $a = 3.8 \pm 0.4$ and $b = 7.1 \pm 0.9$. In this case, $\frac{7.1}{3.8} \pm \left|\frac{7.1}{3.8}\right| \sqrt{\left(\frac{0.4}{3.8}\right)^2 + \left(\frac{0.9}{7.1}\right)^2} = 1.87 \pm 0.31$.

Thus, the answer is likely on the range [1.56, 2.18].

This appendix is beyond the scope of this course.