3 Tools

We would like to find solutions to mathematical equations used by engineers to model the real world. Unfortunately, from calculus, you are aware that there are many problems that simply have no solutions. Consequently, we would like to approximate such solutions in the computer. Similarly, we will also want to simulate the real world using computers, and there is always variability in the real world.

Consequently, we required to use floating-point numbers in a computer to approximate these solutions and to run simulations. There is always the problem, however, that the flaws with floating-point representations will produce an approximation or simulation that is horribly incorrect. For example, something as simple as approximating a derivative can be frustrating, as we know from calculus that

$$\frac{\mathrm{d}}{\mathrm{d}x} f(x) = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h} .$$

Thus, if we keep choosing smaller and smaller values of *h*, we should get better and better approximations:

```
>> cos(1)              % The correct answer
   0.540302305868140

>> % A for-loop with 'i' going from 0 to 17
>> for i = 0:17
    h = 10^-i;                          % h = 1.0, 0.1, 0.01, 0.001, etc.
    (sin(1 + h) - sin(1 - h))/(2*h)
end

   0.454648713412841
   0.539402252169760
   0.540293300874733
   0.540302215817690
   0.540302304967710
   0.540302305856999
   0.540302305895857
   0.540302305673812
   0.540302308449370
   0.540302302898255
   0.540302247387103
   0.540301137164079
   0.540290034933832
   0.540123501480139
   0.544009282066327
   0.555111512312578
   0.555111512312578
   0
```

It seems to get a better and better answer as *h* starts to get smaller, but as *h* gets too small, the accuracy begins to drop off—our approximation of the derivative of sine at 1 is getting worse.

Tabulating this data, the issue becomes more clear:

h	h	Approximation
10^{-1}	0.1	0.497363752535389
10^{-2}	0.01	0.536085981011869
10^{-3}	0.001	0.539881480360327
10^{-4}	0.0001	0.540260231418621
10^{-5}	0.00001	0.540298098505865
10^{-6}	0.000001	0.540301885121330
10^{-7}	0.0000001	0.540302264040449
10^{-8}	0.00000001	0.540302302898255
10^{-9}	0.000000001	0.540302358409406
10^{-10}	0.0000000001	0.540302247387103
10^{-11}	0.00000000001	0.540301137164079
10^{-12}	0.000000000001	0.540345546085064
10^{-13}	0.0000000000001	0.539568389967826
10^{-14}	0.00000000000001	0.544009282066327
10^{-15}	0.000000000000001	0.555111512312578
10^{-16}	0.0000000000000001	0.000000000000000

To see what is happening, it is easier to look at the binary, where the correct answer is

0.1000101001010001010000001111101101010000011010001100

h	Approximation
2^{-1}	**0.0**10011111110001001100000110000100011011000001001110100
2^{-1}	**0.01**10111000011000000011011110000100000110100100111100000
2^{-3}	**0.0111**11001000001011101110110001001110100000111000000000
2^{-4}	**0.1000**00110111111101101110100011011101011101001011100000
2^{-5}	**0.10000**11011101101111001001000111011101111101101000000
2^{-6}	**0.100010**00101000001111110010011011010001101000010000000
2^{-7}	**0.1000100**10111100101111001111010101110010110000110000000
2^{-8}	**0.100010011**11001010111010000100110110111000100000000000
2^{-9}	**0.100010100**00110110110000000001001001000110110000000000000
2^{-2}	**0.1000101000**11011001010001101110000110010001100000000000
2^{-11}	**0.1000101001**00001111001001011101110010111100000000000000
2^{-12}	**0.1000101001001**01010000101000100010111011110000000000000
2^{-13}	**0.1000101001001**10111100010110011010101001100000000000000
2^{-14}	**0.1000101001001111**10010001101001101110111000000000000000
2^{-15}	**0.1000101001010000**11010010001001010101000000000000000000
2^{-16}	**0.1000101001010000**1010100110010000100000000000000000000
2^{-17}	**0.1000101001010001**0001010101000110000000000000000000000
2^{-18}	**0.1000101001010001001**0010110010000011000000000000000000
2^{-19}	**0.1000101001010001001**1001100000111000000000000000000000
2^{-20}	**0.1000101001010001001**1001110000101000000000000000000000
2^{-21}	**0.1000101001010001001111**0100100000000000000000000000000
2^{-22}	**0.1000101001010001001111**1011001110000000000000000000000
2^{-23}	**0.1000101001010001001111111**0101000000000000000000000000
2^{-24}	**0.1000101001010001010000000**0010000000000000000000000000
2^{-25}	**0.1000101001010001010000000**1010000000000000000000000000
2^{-26}	**0.1000101001010001010000000**1100000000000000000000000000
2^{-27}	**0.1000101001010001010000000**1000000000000000000000000000
2^{-28}	**0.1000101001010001010000000**1000000000000000000000000000
2^{-29}	**0.1000101001010001010000000**0000000000000000000000000000
2^{-30}	**0.1000101001010001010000000**0000000000000000000000000000
2^{-31}	**0.1000101001010001010000**0000000000000000000000000000000
2^{-32}	**0.1000101001010001010000**0000000000000000000000000000000
2^{-33}	**0.1000101001010001010000**0000000000000000000000000000000
2^{-34}	**0.1000101001010001010**0000000000000000000000000000000000
2^{-35}	**0.1000101001010001010**0000000000000000000000000000000000
2^{-36}	**0.1000101001010001010**0000000000000000000000000000000000
2^{-37}	**0.1000101001010001**0000000000000000000000000000000000000
2^{-38}	**0.1000101001010001**0000000000000000000000000000000000000
2^{-39}	**0.1000101001010001**0000000000000000000000000000000000000
2^{-40}	**0.1000101001010**0000000000000000000000000000000000000000
2^{-41}	**0.1000101001010**0000000000000000000000000000000000000000
2^{-42}	**0.1000101001010**0000000000000000000000000000000000000000
2^{-43}	**0.1000101001**0000000000000000000000000000000000000000000
2^{-44}	**0.1000101001**0000000000000000000000000000000000000000000
2^{-45}	**0.1000101001**0000000000000000000000000000000000000000000
2^{-46}	**0.1000101**0000000000000000000000000000000000000000000000
2^{-47}	**0.1000101**0000000000000000000000000000000000000000000000
2^{-48}	**0.1000101**0000000000000000000000000000000000000000000000
2^{-49}	**0.1000101**0000000000000000000000000000000000000000000000
2^{-50}	**0.1000101**0000000000000000000000000000000000000000000000
2^{-51}	**0.1000101**0000000000000000000000000000000000000000000000
2^{-52}	0.0000000000000000000000000000000000000000000000000000

To avoid such issues, there are six tools that we will use throughout this course to approximate solutions and run simulations:

1. weighted averages,
2. iteration,
3. linear algebra,
4. interpolation,
5. Taylor series, and
6. bracketing.

We will quickly introduce each of these.

1. Weighted averages

Suppose you have *n* values all of which in some way approximate some value:

$$x_1, x_2, \ldots, x_n.$$

The average of these *n* values is defined as $\frac{x_1 + x_2 + x_3 + \cdots + x_n}{n}$.

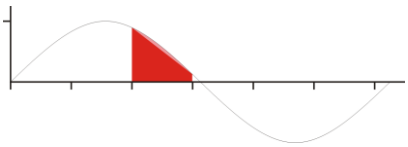
This is essentially the same as calculating $\frac{1}{n}x_1 + \frac{1}{n}x_2 + \frac{1}{n}x_3 + \cdots + \frac{1}{n}x_n$, so we have *n* coefficients, all of which are equal and $\underbrace{\frac{1}{n} + \frac{1}{n} + \frac{1}{n} + \cdots + \frac{1}{n}}_{n \text{ times}} = 1$.

If we have *n* coefficients $c_1, c_2, c_3, \ldots, c_n$ with the property $c_1 + c_2 + c_3 + \cdots + c_n = 1$, then we say that

$$c_1x_1 + c_2x_2 + c_3x_3 + \cdots + c_nx_n$$

is a *weighted average* of the values x_1 through x_n.

In some cases, a weighted average gives a better approximation of a value than a simple average. For example, consider calculating the integral $\int_2^3 \sin(x)\,dx$. One way is to calculate the area of the trapezoid shown in red:



That area is $\frac{\sin(2)+\sin(3)}{2} \approx 0.52521$. This is the average of the two end-points. Could we not get a better approximating by taking the average of three points: $\frac{\sin(2)+\sin(2.5)+\sin(3)}{3} \approx 0.54963$. The correct answer, is of course $\cos(2)-\cos(3) \approx 0.57385$, so it is slightly better; however, it happens that a weighted average is actually better in this case:

$$\frac{1}{4}\sin(2)+\frac{1}{2}\sin(2.5)+\frac{1}{4}\sin(3) \approx 0.56184\,.$$

Consider the relative error of each of these:

	Approximation	Percent relative error
two-point average	$\frac{\sin(2)+\sin(3)}{2}$	8.48 %
three-point average	$\frac{\sin(2)+\sin(2.5)+\sin(3)}{3}$	4.22 %
three-point weighted average	$\frac{1}{4}\sin(2)+\frac{1}{2}\sin(2.5)+\frac{1}{4}\sin(3)$	2.09 %

Later in this course, we will understand why the three-point weighted average is so significantly better.

Practice questions

1. The weighted average described above for integration is referred to as the *composite trapezoidal* rule. We will revisit integration later; however, there is an even more suitable choice of weights: approximate the value of the function on the interval [2, 3] by using the function evaluated at the exact same points, but use the weights 1/6, 2/3 and 1/6. Is this a valid weighted average? What is the percent relative error of this approximation of the integral?

If your calculations are correct, you should get a percent relative error of approximately 0.0358 %.

2. Choose a transcendental function for which you know the antiderivative and an interval of width one (1) and attempt to approximate the integral over this interval by taking the average of four equally spaced points. Next, instead, give the internal two points a weight of 3/8 while giving the two boundary points a weight of only 1/8. What is the relative error of each approximation?

3. Advanced question: Find the integral of the quadratic polynomial $\int_4^5 \left(3x^2 - x\right)dx = 56.5$. Use the weighted averages in Question 1 to approximate this integral. What answer did you get? Assume that this formula is correct for all quadratic polynomials. Why does this suggest that the approximation of the integral of sin(*x*) is so accurate?

2. Iteration

The process of iteration is where an operation repeated performed on an initial approximation to a solution. This means that we start with an initial approximation x_0 and we apply a mathematical operation f to that value and we call that result x_1, so $x_1 = f(x_0)$. We then find $x_2 = f(x_1)$, and then $x_3 = f(x_2)$, and so on. There is no guarantee that this sequence will converge; for example, if $f(x) = x^2$, then here are two sequences:

1. 0.1, 0.01, 0.0001, 0.00000001, 0.0000000000000001, …
2. 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, …
3. 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, …

The first converges to (but never reaches) 0, the second remains at the point, and the third diverges to infinity.

Here is an example of an operation that converges to $\sqrt{2}$: if $x < \sqrt{2}$ then $\frac{1}{x} > \frac{1}{\sqrt{2}}$ so $\frac{2}{x} > \frac{2}{\sqrt{2}} = \sqrt{2}$. Thus, the average of these two values should be a better approximation of $\sqrt{2}$: $\frac{1}{2}\left(x + \frac{2}{x}\right) = \frac{x}{2} + \frac{1}{x}$. This formula was known to the ancient Babylonians, so let's try it out, calculating with 50 digits of precision.

Because $1.4^2 = 1.96$ and $1.5^2 = 2.25$, let's start with 1.4:

x_0 = 1.4
x_1 = 1.4142857142857142857142857142857142857142857142857
x_2 = 1.4142135642135642135642135642135642135642135642136
x_3 = 1.4142135623730950499992895789028639393416700696942
x_4 = 1.4142135623730950488016887242096980790767550505500
x_5 = 1.4142135623730950488016887242096980785696718753770
x_6 = 1.4142135623730950488016887242096980785696718753770

The answer doesn't change after x_6, so let's see how good that approximation is: The correct answer to 52 digits is 1.4142135623730950488016887242096980785696718753769 48, so we are ever so slightly off; ending in 69 would have been slightly more correct.

2.1 Fixed-point theorem

Given any function f, if we want to find a solution to the equation

$$x = f(x),$$

one technique that may find such a solution is to start with an initial point x_0 and then to iteratively evaluate that value at that function. The fixed-point theorem says that if this iteration sequence converges, then it must converge to a solution of this equation.

For example, if we want to find a solution to $x = \cos(x)$, we note that $\cos(x) \approx 1 - \frac{x^2}{2!}$, so using algebra, a solution to $\cos(x) = x$ is approximately $1 - \frac{x^2}{2!} = x$ or a root of $2 - 2x - x^2 = 0$. From algebra, we know that the best solution is $\sqrt{3} - 1 \approx 0.7320508076$, so let's start with $x_0 = 0.7320508076$ and we will repeatedly find $x_1 = \cos(x_0)$, and so on:

x_0 = 0.7320508076
x_1 = 0.7438052150

$x_2 = 0.7358974081$
$x_3 = 0.7412286644$
$x_4 = 0.7376395280$
$x_5 = 0.7400581377$
$x_6 = 0.7384293559$
$x_7 = 0.7395267137$
$x_8 = 0.7387876072$
$x_9 = 0.7392855176$
$x_{10} = 0.7389501371$
$x_{11} = 0.7391760615$
$x_{12} = 0.7390238798$
$x_{13} = 0.7391263929$
$x_{14} = 0.7390573396$

It seems to bounce above and below, but converge to a point. That point is approximately 0.7390851332 which is indeed the point where $x = \cos(x)$.

If we try to find a solution to $x = \sin(x)$ and if we were to start with the same initial value, we find it converges, but much more slowly:

$x_0 = 0.7320508076$
$x_1 = 0.6683964409$
$x_2 = 0.6197282848$
$x_3 = 0.5808139959$
$x_4 = 0.5487046324$
$x_5 = 0.5215824580$
$x_6 = 0.4982528026$
$x_7 = 0.4778914976$
$x_8 = 0.4599079195$
$x_9 = 0.4438655962$
$x_{10} = 0.4294336785$
$x_{11} = 0.4163559688$
$x_{12} = 0.4044304285$
$x_{13} = 0.3934952020$
$x_{14} = 0.3834188302$

If you wait long enough (a few million iterations), you will see that this, however, does converge to the solution to $x = \sin(x)$, namely 0.

On the other hand, the equation $x = \tan(x)$ also has a solution at $x = 0$; however, if we try iterating, we find that it never appears to converge.

Consequently, the fixed-point theorem only gives us a possible means of finding a solution to $x = f(x)$, but it doesn't guarantee we will find such a solution.

2.2 Implementation of the fixed-point theorem and halting conditions

Consider the following C++ function:

```
double fixed_point( double f( double ), double x_current );

double fixed_point( double f( double ), double x_current ) {
    double x_previous; // Uninitialized

    do {
        x_previous = x_current;

        x_current = f( x_current );
    } while ( x_previous != x_current );

    return x_current;
}
```

Now, consider the following program:

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

double fixed_point( double f( double ), double x_current );
int main();

// Definition of 'fixed_point(...)'...

int main() {
    std::cout.precision( 16 );
    std::cout << fixed_point( std::cos, 1.0 ) << std::endl;

    return 0;
}
```

The output is `0.7390851332151607`, which is correct to all but the last digit, as the correct answer to 18 decimal digits of precision is 0.7390851332151642. However, consider the following:

```
double ulgy_function( double x ) {
    return 1.01*cos(x);
}
```

If we execute this function, it appears to be caught in an infinite loop, which is peculiar, as the solution to

$$x = \cos(x)$$

should be very close to a solution to the equation

$$x = 1.01\cos(x),$$

and, of course, it is: to 18 decimal-digits of precision, the solution is 0.743479254976848120.

If you were to print `x_current` each time it is calculated, you would observe this phenomenon:

```
0.5457053289268212
0.8633090370555448
0.6564254333312303
0.8001005948933742
0.7036008888973354
0.7701426521339598
...skip 80 iterations...
0.7434792549768471
0.7434792549768489
0.7434792549768476
0.7434792549768484
0.743479254976848
0.7434792549768482
0.7434792549768481
0.7434792549768482
0.7434792549768481
```

You will see that subsequent calculations of `x_current` alternate forever between two values. In general, if we are calculating a sequence of approximations x_0, x_1, x_2, …, we will say that we are *close enough* if $|x_{k+1} - x_k|$ is less than some tolerance. We will use the symbol $\varepsilon_{\text{step}}$ to indicate this tolerance:

```
double fixed_point( double f( double ), double x_current, double step_tolerance );

double fixed_point( double f( double ), double x_current, double step_tolerance ) {
    double x_previous; // Uninitialized

    do {
        x_previous = x_current;

        x_current = f( x_current );
    } while ( std::abs( x_previous - x_current ) >= step_tolerance );

    return x_current;
}
```

In general, if the distance between successive approximations $|x_{k+1} - x_k| < \varepsilon_{\text{step}}$, then $|x_{k+1} - x| < c\varepsilon_{\text{step}}$ for some constant c, usually assumed to be small. As a general rule, if you want the absolute error to be less than some fixed value ε_{max}, use $\varepsilon_{\text{step}} = 0.1\varepsilon_{\text{max}}$. For the previous example, if we use a tolerance of 10^{-11}, we get the approximation 0.7434792549801468 that, as an approximation to 0.743479254976848120, has an absolute error of 3.30 x 10^{-12}.

Note, however, that this is not always true: if we try to find the fixed point of $x = \sin(x)$, the approximation when using the same tolerance is `0.0003914867511324241`, which has a significantly larger absolute error when compared to the correct solution $x = 0$.

Now, in some cases, there may not be a solution, or this technique may not be able to find a solution. For example, fixed-point iteration will not find a solution to $x = 2.0 + 3.0x$. Indeed, if we execute this function, we get a peculiar return value:

```
inf
```

This is peculiar, because our test is if

```
std::abs( x_previous - x_current ) >= step_tolerance
```

which you may ask, how did this ever return `false`?

Suppose that for whatever reason, `x_current == inf`. With the next calculation, 2.0 + 3.0*x* will again evaluate to `inf`. Thus, the condition is

```
std::abs( inf - inf ) >= step_tolerance
```

However, `inf - inf` evaluates to `nan`, and any comparison with `nan` returns `false`. Thus, if any evaluation of `x_current` results in either `inf` or `nan`, we should stop iterating. The `cmath` macro `std::isfinite(…)` evaluates to `true` if the argument is If this is the case, we will throw the exception `std::range error`:

```
#include <cmath>
#include <stdexcept>

double fixed_point( double f( double ), double x_current, double step_tolerance );

double fixed_point( double f( double ), double x_current, double step_tolerance ) {
    double x_previous; // Uninitialized

    do {
        x_previous = x_current;

        x_current = f( x_current );

        if ( !std::isfinite( x_current ) ) {
            throw std::range_error{ "fixed-point iteration produced 'inf' or 'nan'" };
        }
    } while ( std::abs( x_previous - x_current ) >= step_tolerance );

    return x_current;
}
```

However, this is still not sufficient—it may simply happen that a solution exists, but either fixed-point iteration does not find it, or does not find it in a sufficiently reasonable amount of time. For example, there are countably many solutions to the equation $x = \tan(x)$, yet

```
std::cout << fixed_point( std::tan, 1.0, 1e-14 ) << std::endl;
```

takes a total of 40.959 seconds to return the approximation `1.748172941959977e-05`, which is not with the tolerance of the actual solution $x = 0$. On the other hand, there may be no solution, or if there is, the algorithm does not converge to it. This is demonstrated with the two functions $\frac{2}{x}$, which has two solutions, and $-\frac{2}{x}$. In both cases, fixed-point iteration never converges to either solution, for in both cases, given x_0, we seen that $x_2 = x_0$.

Thus, we are required to add an additional requirement: a maximum number of iterations.

```
#include <cmath>
#include <stdexcept>

double fixed_point( double f( double ), double x_current,
                    double step_tolerance, unsigned int max_iterations );

double fixed_point( double f( double ), double x_current,
                    double step_tolerance, unsigned int max_iterations ) {
    double x_previous; // Uninitialized

    for ( unsigned int iterations{0}; iterations < max_iterations; ++ iterations ) {
        x_previous = x_current;

        x_current = f( x_current );

        if ( !std::isfinite( x_current ) ) {
            throw std::range_error{ "fixed-point iteration produced 'inf' or 'nan'" };
        }

        if ( std::abs( x_previous - x_current ) < step_tolerance ) {
            return x_current;
        }
    }

    throw std::runtime_error{ "fixed-point iteration did not converge" };
}
```

Now, a user would use this function as follows:

```
double result;  // Uninitialized

try {
    result = fixed_point( f, 1.32, 1e-10, 100 );
} catch ( std::range_error &e ) {
    // fixed-point iteration halted as either 'inf' or 'nan' was generated
} catch ( std::runtime_error &e ) {
    // fixed-point iteration did not converge
}
```

In our design above, we are throwing exceptions whenever the fixed-point iteration does not converge. An alternative strategy would be to return not-a-number. This would signal that a solution was not found (although providing less evidence why) but would require the user to check the return value.

```
#include <cmath>
#include <stdexcept>

double fixed_point( double f( double ), double x_current,
                    double step_tolerance, unsigned int max_iterations );

double fixed_point( double f( double ), double x_current,
                    double step_tolerance, unsigned int max_iterations ) {
    double x_previous; // Uninitialized

    for ( unsigned int iterations{0}; iterations < max_iterations; ++ iterations ) {
        x_previous = x_current;

        x_current = f( x_current );

        if ( !std::isfinite( x_current ) ) {
            return NAN;
        }

        if ( std::abs( x_previous - x_current ) < step_tolerance ) {
            return x_current;
        }
    }

    return NAN;
}
```

Now, a user would use this function as follows:

```
double result{ fixed_point( f, 1.32, 1e-10, 100 ) };

if ( std::isnan( result ) ) {
    // The fixed-point iteration did not converge--take additional steps
}
```

Practice questions

1. Consider the function defined by $f(x) \overset{\text{def}}{=} x^2$. This has two solutions to the equation $x^2 = x$, namely $x = 0$ and $x = 1$. Answer the following questions: Assuming that ε is a small positive number:

   a. Will $f(0 + \varepsilon)$ be closer or further from 0 than was $0 + \varepsilon$?
   b. Will $f(0 - \varepsilon)$ be closer or further from 0 than was $0 - \varepsilon$?
   c. Will $f(1 + \varepsilon)$ be closer or further from 1 than was $1 + \varepsilon$?
   d. Will $f(1 - \varepsilon)$ be closer or further from 1 than was $1 - \varepsilon$?

Consequently, which solution to $f(x) = x$ will iteration allow us to find (assuming we have an appropriate initial value)? Why?

2. Suppose we are trying to find the root of a polynomial $x^3 - 2x^2 + x - 1 = 0$. This polynomial has one root close to $x = 1.75$. If we rewrite the equation, we get $-x^3 + 2x^2 + 1 = x$. What happens if we start iterating the function defined as $f(x) \overset{\text{def}}{=} -x^3 + 2x^2 + 1$ starting with $x_0 = 1.75$? Does it converge to the root near 1.75? Alternatively, we can some more with the equation to get it in the form $x = f(x)$ as follows:

$$\begin{aligned} x^3 - 2x^2 + x - 1 &= 0 \\ \therefore x^3 &= 2x^2 - x + 1 \\ \therefore x &= \frac{2x^2 - x + 1}{x^2} \end{aligned}$$

What happens if you iterating the function defined as $g(x) \overset{\text{def}}{=} \frac{2x^2 - x + 1}{x^2}$ starting with $x_0 = 1.75$?

3. Show that $x = \frac{x}{2} + \frac{1}{x}$ has $\sqrt{2}$ as a fixed point. In general, pick a positive real number r, and show that $\sqrt{r}$ is a fixed point of $x = \frac{1}{2}\left(x + \frac{r}{x}\right)$. Use this to approximate a value of $\sqrt{7}$.

3. Linear algebra

We will repeatedly find solutions to a system of linear equations. Now, in first year, if you had an n x n matrix A and a target n-dimensional vector $\mathbf{b}$, you could find a solution to $A\mathbf{x} = \mathbf{b}$ by creating the augmented matrix $(A|\mathbf{b})$ and applying the Gaussian elimination algorithm followed by backward substitution:

1. Iterate through the columns starting at column $j = 1$ up to $n - 1$,
   a. for each row starting at $i = j + 1$ up to n,
      i. add $-\frac{a_{i,j}}{a_{j,j}}$ times Row j (the row containing the diagonal of Column j) onto Row i.

This will create a matrix that is in row-echelon form, after which we can apply backward substitution.

Now, if there was a zero on the diagonal in Row j, this algorithm does not work, as the denominator of the ratio would then be zero. Thus, in linear algebra, you saw that you could swap Row j with any Row k with $k > j$ such that $a_{k,j} \neq 0$. If you could not find such a row, then the possibility for a unique solution no longer exists: either there are infinitely many solutions or no solutions.

Thus, our algorithm now looks like:

1. Iterate through the columns starting at column $j = 1$ up to $n - 1$,
   a. if $a_{j,j} = 0$, find a Row k with $k > j$ such that $a_{k,j} \neq 0$ and swap Rows j and k,
   b. for each row starting at $i = j + 1$ up to n,
      i. add $-\frac{a_{i,j}}{a_{j,j}}$ times Row j (the row containing the diagonal of Column j) onto Row i.

This process is called *pivoting*. This is guaranteed to find a unique solution if it exists; however, there are problems when we begin to use floating-point arithmetic. Let us consider the following problem:

$$\begin{aligned} x + y &= 2 \\ 0.00001x + y &= 1 \end{aligned}$$

Intuitively, you can guess that the solution is approximately $x = y = 1$, and indeed, the exact solution is

$$\mathbf{x} = \begin{pmatrix} \frac{10000}{9999} \\ \frac{9998}{9999} \end{pmatrix} = \begin{pmatrix} 1.\overline{0001} \\ 0.\overline{9998} \end{pmatrix}.$$

Suppose, however, we are restricted at any step to storing only four significant digits. We will not formally use the representation `EEMNNN`, but we will simulate this by only ever storing four significant digits.

$$\left(\begin{array}{cc:c} 1.000 & 1.000 & 2.000 \\ 0.00001 & 1.000 & 1.000 \end{array}\right)$$

Adding –0.00001 times Row 1 onto Row 2, we get 1.000 – 0.00001 = 0.99999, but rounded to four decimal places gives the result 1.000, with a similar result for the third column:

$$\left(\begin{array}{cc:c} 1 & 1 & 2 \\ 0.00001 & 1 & 1 \end{array}\right) \sim \left(\begin{array}{cc:c} 1 & 1 & 2 \\ 0 & 1.000 & 1.000 \end{array}\right).$$

Thus, using backward substitution, we get that the solution is $\mathbf{x} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$, which is actually pretty close to our actual solution: the percent relative error is 0.01 %.

If, on the other hand, we swapped the two rows and applied Gaussian elimination, we would get a very different result. From

$$\left(\begin{array}{cc:c} 0.00001 & 1.000 & 1.000 \\ 1.000 & 1.000 & 2.000 \end{array}\right),$$

we would have to add –100000 times Row 1 onto Row 2, and –100000 + 1 = –99999 which equals –100000 if we round to four significant digits. Similarly, –100000 + 2 = –99998, so again, when reduced to four significant digits, results in –100000. Thus, we have

$$\left(\begin{array}{cc:c} 0.00001 & 1.000 & 1.000 \\ 1.000 & 1.000 & 2.000 \end{array}\right) \sim \left(\begin{array}{cc:c} 0.00001 & 1.000 & 1.000 \\ 0 & 100000 & 100000 \end{array}\right).$$

Now, applying backward substitution, we get that the solution is exactly $\mathbf{x} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. This approximation to the solution has a 100 % relative error!

The phenomenon that is being observed here is that you must recall that each line of the system of linear equations represents different information, but when adding a very large multiple of one equation onto another, through the loss of precision, the information in the equation being added onto may be completely lost.

Instead, we must modify our pivoting algorithm to be more general:

1. Iterate through the columns starting at column $j = 0$ up to $n - 2$,
    a. find that Row k with $k \geq j$ such that $|a_{k,j}|$ is the largest of these entries and then swap Rows j and k,
    b. for each row starting at $i = j + 1$ up to n,
        i. add $-\frac{a_{i,j}}{a_{j,j}}$ times Row j (the row containing the diagonal of Column j) onto Row i.

This process of finding the largest entry on or below the diagonal and moving that entry to the diagonal is called *partial pivoting*.

The following is an example of the application of this algorithm:

$$\left(\begin{array}{rrrrr} -0.4 & -2.7 & 4.0 & 2.9 & 2.7 \\ 3.2 & 4.6 & -3.3 & 3.7 & -6.3 \\ 1.2 & -4.4 & 2.0 & 4.3 & -4.6 \\ 4.0 & 2.0 & 0.0 & 1.0 & 8.0 \end{array}\right).$$

Starting with Column 1, we find the largest entry in absolute value on or below the diagonal, so examining –0.4, 3.2, 1.2 and 4.0, we observe that this is in Row 4, so we swap Rows 1 and 4:

$$\left(\begin{array}{rrrrr} 4.0 & 2.0 & 0.0 & 1.0 & 8.0 \\ 3.2 & 4.6 & -3.3 & 3.7 & -6.3 \\ 1.2 & -4.4 & 2.0 & 4.3 & -4.6 \\ -0.4 & -2.7 & 4.0 & 2.9 & 2.7 \end{array}\right)$$

We now add –0.8 times Row 1 onto Row 2, –0.3 times Row 1 onto Row 3, and 0.1 times Row 1 onto Row 4:

$$\left(\begin{array}{rrrrr} 4.0 & 2.0 & 0.0 & 1.0 & 8.0 \\ 0 & 3.0 & -3.3 & 2.9 & -12.7 \\ 0 & -5.0 & 2.0 & 4.0 & -7.0 \\ 0 & -2.5 & 4.0 & 3.0 & 3.5 \end{array}\right)$$

Continuing with Column 2, we find the largest entry in absolute value on or below the diagonal, so from 3.0, –5.0 and –2.5, we note that Row 3 contains the largest, so we swap Rows 2 and 3:

$$\left(\begin{array}{rrrr:r} 4.0 & 2.0 & 0.0 & 1.0 & 8.0 \\ 0 & -5.0 & 2.0 & 4.0 & -7.0 \\ 0 & 3.0 & -3.3 & 2.9 & -12.7 \\ 0 & -2.5 & 4.0 & 3.0 & 3.5 \end{array}\right)$$

Next, we add 0.6 times Row 2 onto Row 3 and x –0.5 time Row 2 onto Row 4:

$$\left(\begin{array}{rrrr:r} 4.0 & 2.0 & 0.0 & 1.0 & 8.0 \\ 0 & -5.0 & 2.0 & 4.0 & -7.0 \\ 0 & 0 & -2.1 & 5.3 & -16.9 \\ 0 & 0 & 3.0 & 1.0 & 7.0 \end{array}\right)$$

Continuing to Column 3, we observe that between –2.1 and 3.0, the largest in absolute value is the second, so we swap Rows 3 and 4:

$$\left(\begin{array}{rrrr:r} 4.0 & 2.0 & 0.0 & 1.0 & 8.0 \\ 0 & -5.0 & 2.0 & 4.0 & -7.0 \\ 0 & 0 & 3.0 & 1.0 & 7.0 \\ 0 & 0 & -2.1 & 5.3 & -16.9 \end{array}\right)$$

We now add 0.7 times Row 3 onto Row 4:

$$\left(\begin{array}{rrrr:r} 4.0 & 2.0 & 0.0 & 1.0 & 8.0 \\ 0 & -5.0 & 2.0 & 4.0 & -7.0 \\ 0 & 0 & 3.0 & 1.0 & 7.0 \\ 0 & 0 & 0 & 6.0 & -12.0 \end{array}\right)$$

Now the matrix is in row-echelon form. Incidentally, we can now find the determinant of the original matrix

$$\begin{pmatrix} -0.4 & -2.7 & 4.0 & 2.9 \\ 3.2 & 4.6 & -3.3 & 3.7 \\ 1.2 & -4.4 & 2.0 & 4.3 \\ 4.0 & 2.0 & 0.0 & 1.0 \end{pmatrix}$$

by multiplying the diagonals and multiplying this by –1 raised to the power of the number of row swaps we did (in this case 3), so the determinant is 360.

However, applying backward substitution, we have that the solution to this system of linear equations is

$$\begin{pmatrix} 2.0 \\ 1.0 \\ 3.0 \\ -2.0 \end{pmatrix}.$$

Practice questions

1. Apply this algorithm to find a solution to the system of linear equations represented by the augmented matrix

$$\left(\begin{array}{cc|c} -2.1 & 2.6 & 6.2 \\ 3.0 & 2.0 & 14.0 \end{array}\right)$$

$$\left(\begin{array}{ccc|c} 2.8 & -0.6 & 7.5 & 18.4 \\ 4.0 & 2.0 & -3.0 & -4.0 \\ -2.4 & 3.8 & 2.8 & -0.6 \end{array}\right)$$

$$\left(\begin{array}{cccc|c} 1.5 & -2.5 & -4.0 & 2.7 & 18.9 \\ -3.0 & 3.4 & 2.8 & 1.4 & -8.6 \\ 5.0 & 1.0 & -3.0 & 1.0 & 16.0 \\ 0.5 & 3.7 & 6.6 & 4.9 & -9.5 \end{array}\right)$$

2. The absolute, relative and percent relative errors of a vector **a** that approximates a solution vector **x** are defined as

$$\|\mathbf{x}-\mathbf{a}\|_2, \quad \frac{\|\mathbf{x}-\mathbf{a}\|_2}{\|\mathbf{x}\|_2} \text{ and } \frac{\|\mathbf{x}-\mathbf{a}\|_2}{\|\mathbf{x}\|_2}\cdot 100\,\%,$$

respectively. Using four digits of accuracy, find solutions to this system of two linear equations first not using partial pivoting, and then using partial pivoting.

$$\left(\begin{array}{cc|c} 0.003572 & 52.35 & 23.91 \\ 9.273 & 0.05982 & 20.92 \end{array}\right)$$

Remember, after any arithmetic operation, be it addition or multiplication, you must round the result to four decimal places before you proceed.

In the first case, you should find that the solution vectors are $\begin{pmatrix} 2.800 \\ 0.4566 \end{pmatrix}$ and $\begin{pmatrix} 2.253 \\ 0.4565 \end{pmatrix}$. The correct answer to ten decimal places is $\begin{pmatrix} 2.2530667379 \\ 0.4565720438 \end{pmatrix}$. What is the relative error of each of the approximations to the solution above?

3.1 Iteration and linear algebra

In a previous question, we saw that if we manipulated the equation $x^3-2x^2+x-1=0$ into the equation $x=\frac{2x^2-x+1}{x^2}$, then we could define a function $g(x)\overset{\text{def}}{=}\frac{2x^2-x+1}{x^2}$ such that if we iterated this function starting with $x_0 = 1.75$, then it would converge to a solution of the equation $x=\frac{2x^2-x+1}{x^2}$, which in turn is a solution to the original equation $x^3-2x^2+x-1=0$.

When you are solving a system of linear equations $A\mathbf{x} = \mathbf{b}$ with a given matrix A and a given target vector **b**, this is not in the form $\mathbf{x} = f(\mathbf{x})$; however, let us proceed as follows:

Every matrix A can be written as a matrix consisting of its diagonal entries plus a matrix consisting of its off-diagonal entries; for example, if

$$A = \begin{pmatrix} 4 & 1 & 0 & -1 \\ 1 & 5 & 2 & 0 \\ 0 & 2 & 20 & 3 \\ -1 & 0 & 3 & -10 \end{pmatrix},$$

then we can write

$$A = \begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 20 & 0 \\ 0 & 0 & 0 & -10 \end{pmatrix} + \begin{pmatrix} 0 & 1 & 0 & -1 \\ 1 & 0 & 2 & 0 \\ 0 & 2 & 0 & 3 \\ -1 & 0 & 3 & 0 \end{pmatrix}.$$

Let us call these two matrices A_{diag} and A_{off}. Thus, we may now write our system of linear equations as

$$(A_{\text{diag}} + A_{\text{off}})\mathbf{x} = \mathbf{b}$$

and using the properties of matrix-vector arithmetic, this equals

$$A_{\text{diag}}\,\mathbf{x} + A_{\text{off}}\,\mathbf{x} = \mathbf{b}.$$

Now, this isn't in the form $\mathbf{x} = f(\mathbf{x})$, but let us take the one term to the right-hand side:

$$A_{\text{diag}}\,\mathbf{x} = \mathbf{b} - A_{\text{off}}\,\mathbf{x}.$$

One fact you likely remember is that it's easy to invert a diagonal matrix:

$$\text{If } A_{\text{diag}} = \begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 5 & 0 & 0 \\ 0 & 0 & 20 & 0 \\ 0 & 0 & 0 & -10 \end{pmatrix}, \text{ then } A_{\text{diag}}^{-1} = \begin{pmatrix} 0.25 & 0 & 0 & 0 \\ 0 & 0.2 & 0 & 0 \\ 0 & 0 & 0.05 & 0 \\ 0 & 0 & 0 & -0.1 \end{pmatrix}.$$

Multiplying both sides of the equation by the inverse of A_{diag}, we get

$$\begin{aligned} \mathbf{x} &= A_{\text{diag}}^{-1}\left(\mathbf{b} - A_{\text{off}}\,\mathbf{x}\right) \\ &= A_{\text{diag}}^{-1}\mathbf{b} - \left(A_{\text{diag}}^{-1}A_{\text{off}}\right)\mathbf{x} \end{aligned}.$$

Suddenly, we have an equation of the form $\mathbf{x} = f(\mathbf{x})$ where

$$f\left(\mathbf{x}\right) \overset{\text{def}}{=} A_{\text{diag}}^{-1}\mathbf{b} - \left(A_{\text{diag}}^{-1}A_{\text{off}}\right)\mathbf{x}\,.$$

In our example, if we were trying to solve $A\mathbf{x} = \mathbf{b}$ where $\mathbf{b} = \begin{pmatrix} 0.5 \\ 0.7 \\ -7.6 \\ -7.7 \end{pmatrix}$, then $A_{\text{diag}}^{-1} = \begin{pmatrix} 0.25 & 0 & 0 & 0 \\ 0 & 0.2 & 0 & 0 \\ 0 & 0 & 0.05 & 0 \\ 0 & 0 & 0 & -0.1 \end{pmatrix}$ and

consequently,

$$A_{\text{diag}}^{-1}\mathbf{b} = \begin{pmatrix} 0.125 \\ 0.140 \\ -0.380 \\ 0.770 \end{pmatrix} \text{ and } A_{\text{diag}}^{-1}A_{\text{off}} = \begin{pmatrix} 0 & 0.25 & 0 & -0.25 \\ 0.2 & 0 & 0.4 & 0 \\ 0 & 0.1 & 0 & 0.15 \\ 0.1 & 0 & -0.3 & 0 \end{pmatrix}.$$

Consequently, we can apply the fixed-point theorem by starting with the initial vector

$$\mathbf{x}_0 = A_{\text{diag}}^{-1}\mathbf{b} = \begin{pmatrix} 0.125 \\ 0.140 \\ -0.380 \\ 0.770 \end{pmatrix},$$

as this is, after all, the solution to $A_{\text{diag}}\,\mathbf{x} = \mathbf{b}$ so this may be a reasonable approximation to a solution to $A\mathbf{x} = \mathbf{b}$.

We now iterate:

$$\mathbf{x}_1 = \begin{pmatrix} 0.2825 \\ 0.267 \\ -0.5095 \\ 0.6435 \end{pmatrix},\ \mathbf{x}_2 = \begin{pmatrix} 0.219125 \\ 0.2873 \\ -0.503225 \\ 0.5889 \end{pmatrix},\ \mathbf{x}_3 = \begin{pmatrix} 0.2004 \\ 0.297465 \\ -0.497065 \\ 0.59712 \end{pmatrix},\ \mathbf{x}_4 = \begin{pmatrix} 0.19991375 \\ 0.298746 \\ -0.4993145 \\ 0.6008405 \end{pmatrix},\ \mathbf{x}_5 = \begin{pmatrix} 0.200523625 \\ 0.29974305 \\ -0.500000675 \\ 0.600214275 \end{pmatrix}, \ldots$$

and it appears that this sequence of vectors is getting closer and closer to the vector

$$\mathbf{x}_{\text{soln}} = \begin{pmatrix} 0.2 \\ 0.3 \\ -0.5 \\ 0.6 \end{pmatrix},$$

which is indeed the exact solution to the system of linear equations given by

$$\begin{pmatrix} 4 & 1 & 0 & -1 \\ 1 & 5 & 2 & 0 \\ 0 & 2 & 20 & 3 \\ -1 & 0 & 3 & -10 \end{pmatrix}\mathbf{x} = \begin{pmatrix} 0.5 \\ 0.7 \\ -7.6 \\ -7.7 \end{pmatrix}.$$

The percent relative error of $\mathbf{x}_4$ as an approximation to the solution vector is already small:

$$\frac{\|\mathbf{x}_{\text{soln}} - \mathbf{x}_4\|_2}{\|\mathbf{x}_{\text{soln}}\|_2} \cdot 100\,\% = 0.0722\,\%;$$

that is, less than one part in one thousand.

A question of run time

How slow are matrix operations? Given an $n \times n$ matrix A and an n-dimensional vector $\mathbf{v}$, calculating $A\mathbf{v}$ requires one to multiply each entry of the matrix by one of the entries in the vector. Consequently, there are n^2 multiplications required, and each of these must be added to an initial value. Thus, the run time is $O(n^2)$.

If you are multiplying a diagonal $n \times n$ matrix and an n-dimensional vector, the run-time drops to $O(n)$. Why?

Multiplying two $n \times n$ matrices requires each row of the first matrix to be element-wise multiplied and added by each column of the second matrix. This element-wise multiplication requires $O(n)$ multiplications and additions, and as this must be done for each row in the first and each column in the second, this requires an overall time of $O(n^3)$.

If you are multiplying a diagonal matrix by a full matrix, the run time drops to $O(n^2)$ and multiplying two diagonal matrices requires on $O(n)$ time.

Gaussian elimination requires you to create a zero at each entry of a matrix below the diagonal. Assuming we do not have to perform any partial pivoting, there are $\frac{n(n-1)}{2}$ such entries that must be zeroed out. As a rough estimate, you may believe therefore that each of these zeroings requires you to add one multiple of a row onto another, and therefore each of these requires n multiplications and additions, resulting in $O(n^3)$. You could be a little more precise and calculate

$$\sum_{j=1}^{n-1}\sum_{i=j+1}^{n}(n-j)=\frac{n(2n-1)(n-1)}{6}\approx\frac{n^3}{3}$$

only to realize that it is still $O(n^3)$ operations.

4 Interpolation

Question: Given n points $(x_1, y_1), \ldots, (x_n, y_n)$, is there a polynomial of degree $n - 1$ of the form

$$y(x) = a_{n-1}x^{n-1} + \ldots + a_2x^2 + a_1x + a_0$$

that passes through all n pointers? From a constraints point-of-view, the answer should be yes, for there are n different constraints, and there are n coefficients that can be manipulated to go through the points.

The answer is "yes", so long as all the x-values are unique. To see this, let find these interpolating polynomials:

Given a single point (x_1, y_1), the constant polynomial (i.e., a polynomial of degree 0) $y(x) = y_1$ passes through the point.

Given two points, (x_1, y_1) and (x_2, y_2), a constant polynomial will only pass through both points if $y_1 = y_2$; however, this is usually not the case. Instead, we know from secondary school that you can find a straight line passing through these two points; that is, there is a line $y(x) = a_1x + a_0$ that passes through both points. You would have found such a function by finding values of both a_1 and a_0 algebraically through fist finding the slope and then finding the y-intercept. We will use a different approach.

First, if we the polynomial to pass through both points, then both of these equations must hold:

$$y_1 = a_1x_1 + a_0$$
$$y_2 = a_1x_2 + a_0$$

This is a system of two equations and two unknowns (the unknowns being a_0 and a_1). Note that this can be written as

$$\begin{pmatrix} x_1 & 1 \\ x_2 & 1 \end{pmatrix}\begin{pmatrix} a_1 \\ a_0 \end{pmatrix}=\begin{pmatrix} y_1 \\ y_2 \end{pmatrix}.$$

If you don't see this, please review your first-year linear algebra notes on matrix-vector multiplication. Note that we see that the determinant $x_1 - x_2$, which is non-zero so long as x_1 differs from x_2.

Thus, if we wanted to find a line that passes through the points (2, 5) and (10, 4), we would create the system of equations

$$\begin{pmatrix} 2 & 1 \\ 10 & 1 \end{pmatrix}\begin{pmatrix} a_1 \\ a_0 \end{pmatrix} = \begin{pmatrix} 5 \\ 4 \end{pmatrix},$$

and solving this, using Gaussian elimination with partial pivoting, we have

$$\left(\begin{array}{cc|c} 2 & 1 & 5 \\ 10 & 1 & 4 \end{array}\right) \sim \left(\begin{array}{cc|c} 10 & 1 & 4 \\ 2 & 1 & 5 \end{array}\right) \sim \left(\begin{array}{cc|c} 10 & 1 & 4 \\ 0 & 0.8 & 4.2 \end{array}\right)$$

and using backward substitution, we have $\begin{pmatrix} a_1 \\ a_0 \end{pmatrix} = \begin{pmatrix} -0.125 \\ 5.25 \end{pmatrix}$. Thus, the interpolating straight line must be $-0.125x + 5.25$, and if we substitute in the values of x_1 and x_2, we get the values 5 and 4, respectively.

This is an excellent point to show you where numerical error can result in a poor approximation: let us go back to using exactly 4 decimal digits of precision. Now, let us find the interpolating polynomial that passes through the points (0.001, 11) and (2, 7). The correct answer should be a polynomial very similar to $y(x) = 11 - 2x$. If we did not use partial pivoting, however, our result would be very different:

$$\left(\begin{array}{cc|c} 0.001 & 1 & 11 \\ 2 & 1 & 7 \end{array}\right) \sim \left(\begin{array}{cc|c} 0.001 & 1 & 11 \\ 0 & -1999.0 & -21990. \end{array}\right)$$

and using backward substitution, we get that the solution is $\begin{pmatrix} a_1 \\ a_0 \end{pmatrix} = \begin{pmatrix} 0. \\ 11.00 \end{pmatrix}$, which suggest the interpolating polynomial is the polynomial $y(x) = 11.00$. This is likely not a good approximation to the best answer to four significant digits, which is

$$y(x) = -2.001x + 11.00.$$

If partial pivoting was employed, the solution we get is $\begin{pmatrix} a_1 \\ a_0 \end{pmatrix} = \begin{pmatrix} -2.000 \\ 11.00 \end{pmatrix}$, which suggests the interpolating polynomial is the more reasonable $y(x) = -2.000x + 11.00$. Thus, it is essential to mitigate the effects of numerical error even for a problem as simple as finding an interpolating straight line!

The slope is off by approximately a factor of two.

Moving onto three points: (x_1, y_1), (x_2, y_2), and (x_3, y_3). Again, in general, a straight line will not pass through three points unless they are collinear, a rare event indeed. A quadratic polynomial, however, has three coefficients, and thus it seems reasonable that three constraints should allow us to prescribe the values of those three coefficients:

$$\begin{aligned} y_1 &= a_2 x_1^2 + a_1 x_1 + a_0 \\ y_2 &= a_2 x_2^2 + a_1 x_2 + a_0 \\ y_3 &= a_2 x_3^2 + a_1 x_3 + a_0 \end{aligned}$$

which again defines the system of linear equations

$$\begin{pmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{pmatrix}\begin{pmatrix} a_2 \\ a_1 \\ a_0 \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}.$$

The determinant is $(x_1 - x_2)(x_1 - x_3)(x_2 - x_3)$, so again, in general, the determinant is non-zero so long as the three x-values are different.

As an example, we can find an interpolating polynomial through the three points (–3, 6), (5, 12) and (–2.5, 4.5). We note the points need not be ordered with respect to x. This gives us the system

$$\begin{pmatrix} 9 & -3 & 1 \\ 25 & 5 & 1 \\ 6.25 & -2.5 & 1 \end{pmatrix}\begin{pmatrix} a_2 \\ a_1 \\ a_0 \end{pmatrix} = \begin{pmatrix} 6 \\ 12 \\ 4.5 \end{pmatrix}$$

Solving this, using partial pivoting, we have

$$\left(\begin{array}{ccc|c} 9 & -3 & 1 & 6 \\ 25 & 5 & 1 & 12 \\ 6.25 & -2.5 & 1 & 4.5 \end{array}\right) \sim \left(\begin{array}{ccc|c} 25 & 5 & 1 & 12 \\ 9 & -3 & 1 & 6 \\ 6.25 & -2.5 & 1 & 4.5 \end{array}\right)$$

$$\sim \left(\begin{array}{ccc|c} 25 & 5 & 1 & 12 \\ 0 & -4.8 & 0.64 & 1.68 \\ 0 & -3.75 & 0.75 & 1.5 \end{array}\right) \sim \left(\begin{array}{ccc|c} 25 & 5 & 1 & 12 \\ 0 & -4.8 & 0.64 & 1.68 \\ 0 & 0 & 0.25 & 0.1875 \end{array}\right)$$

Using backward substitution, we have the solution is the vector $\begin{pmatrix} a_2 \\ a_1 \\ a_0 \end{pmatrix} = \begin{pmatrix} 0.5 \\ -0.25 \\ 0.75 \end{pmatrix}$. Thus, the interpolating polynomial is $y(x) = 0.5x^2 - 0.25x + 0.75$.

In general, to find the interpolating polynomial of degree $n - 1$ passing through n points results in a system of n linear equations in n unknowns. Unfortunately, the resulting polynomials are highly sensitive to noise in both the x- and the y-values. Consequently, small changes can result in large changes to the interpolating polynomial; especially if the x-values are not equally, or at least approximately equally spaced. Consequently, we will in general only be finding the interpolating polynomials of a reasonably small number of points.

Practice questions

1. Find the polynomial interpolating the points (3, 4), (7, 2). You can check your answer by evaluating the polynomial at $x = 3$ and $x = 7$.

2. Find the polynomial interpolating the points (–2, –15), (–1, –7), (2, –7). You can check your answer by evaluating the polynomial at $x = –2$, $x = –1$ and $x = 2$.

3. Without solving the system of linear equations, what is the augmented matrix that would have to be solved in order to find the polynomial interpolating through the points (–4, 7), (–1, 2), (3, 3) and (4, 5).

5 Taylor series

Given a function f, you have seen the Taylor series approximation of an infinitely differentiable function is

$$f(x) = f(x_0) + f^{(1)}(x_0)(x - x_0) + \frac{1}{2}f^{(2)}(x_0)(x - x_0)^2 + \frac{1}{6}f^{(3)}(x_0)(x - x_0)^3 + \cdots$$

Recall that the denominators of the fractions are actually factorials, so more correctly

$$f(x)=\frac{1}{0!}f(x_0)+\frac{1}{1!}f^{(1)}(x_0)(x-x_0)+\frac{1}{2!}f^{(2)}(x_0)(x-x_0)^2+\frac{1}{3!}f^{(3)}(x_0)(x-x_0)^3+\cdots.$$

Written using a sum, we have

$$f(x)=\sum_{k=0}^{\infty}\frac{1}{k!}f^{(k)}(x_0)(x-x_0)^k.$$

Now, as engineers, we are interested in approximating the function in close proximity to a known point, so rather than having two points x and x_0, we will have a point x, and consider another point close to x, namely $x + h$. In this case, the formula can be written as:

$$f(x+h)=\sum_{k=0}^{\infty}\frac{1}{k!}f^{(k)}(x)h^k.$$

If you truncate this approximation, it will only be approximately correct:

$$f(x+h)\approx\sum_{k=0}^{n}\frac{1}{k!}f^{(k)}(x)h^k.$$

For example, using $n = 0$, we have

$$f(x+h)\approx f(x),$$

which is true for very small values of h, as sin(3.295929385) is very close in value to sin(3.295929386). Unfortunately, sin(2.1) is not very similar to sin(2), so it isn't a good approximation when h is large:

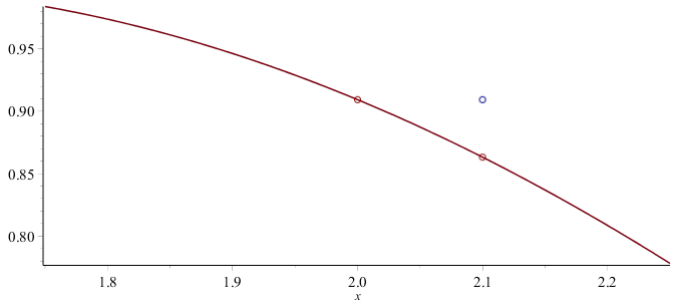


Figure 1. Approximating sin(2.1) by using a value near it: sin(2.0).

Using $n = 1$, we have

$$f(x+h)\approx f(x)+f^{(1)}(x)h,$$

which is a slightly better approximation, for this says to follow the slope of f in the direction of $x + h$. This is shown in the following image:

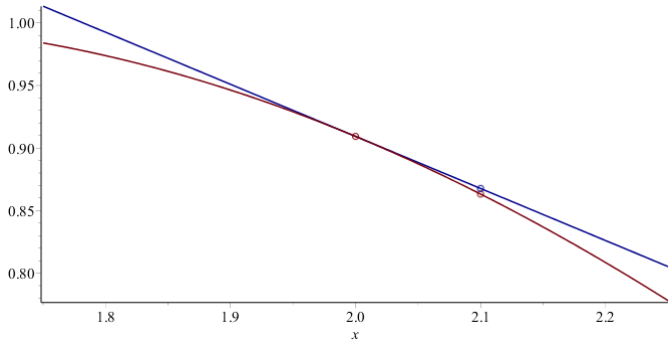


Figure 2. Approximating sin(2.1) by using a 1st-order Taylor series approximation around the point $x = 2$.

Again, the approximation is significantly closer.

We can also add in the concavity:

$$f(x+h) \approx f(x) + f^{(1)}(x)h + \frac{1}{2} f^{(2)}(x)h^2,$$

so now we are following a parabola that *hugs* the curve at x. Now we get an even better approximation of $x + h$, as is shown in this image:

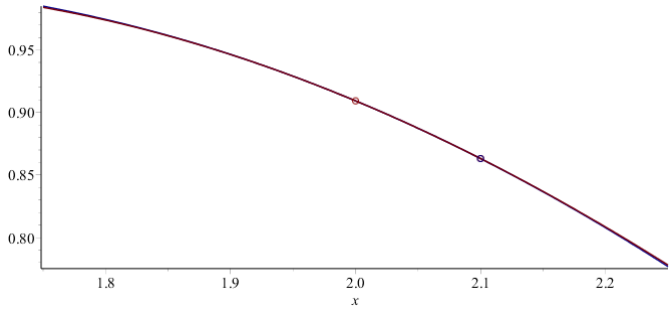


Figure 3. Figure 2. Approximating sin(2.1) by using a 2nd-order Taylor series approximation around the point x = 2.

For your interest, the following are approximations of sin(2.1) using Taylor series approximations. The exact answer is 0.8632093666488738

Order	Approximation	Percent relative error
0	0.9092974268256817	5.34 %
1	0.8676827431709675	0.518 %
2	0.8631362560368391	0.00847 %
3	0.8632056138429302	0.000435 %
4	0.8632094025822090	0.00000416 %
5	0.8632093679033059	0.000000145 %
6	0.8632093666403927	0.000000000983 %

Now, there is a problem: while calculating higher derivatives of the sine function is easy, in other cases, this can be much more problematic, especially if the actual function is not known. Never-the-less, we will attempt to use this tool where possible, as it consistently gives the best approximation.

The important question is, can we in some way determine what the magnitude of the error is? From first year, you learned that the error is

$$f(x+h)=\left(\sum_{k=0}^{n}\frac{1}{k!}f^{(k)}(x)h^{k}\right)+\frac{1}{(n+1)!}f^{(n+1)}(\xi)h^{n+1}.$$

where $x \le \xi \le x+h$ assuming h is positive. What this says is that the error is bounded by the magnitude of the $(n+1)^{st}$ derivative in the neighbourhood of x.

Practice questions

1. Argue that the maximum possible error of an n^{th}-order Taylor series for the sine function is $\frac{h^{n+1}}{(n+1)!}$, regardless as to the values of x and h.

6 Bracketing

In some cases, when analytical techniques are unavailable, the only other possible is to attempt to bracket the solution, and to then attempt to reduce that bracket as much as possible. For example, consider the children's game of hi-lo, where one individual attempts to guess a number from 1 to 100. The person who knows the number responds with either "high" if the number guess is larger than the target number, or "low" if it is lower than the target number.

In this case, the ideal solution is to always guess half-way between the best solutions: first, start with 50, and if that is too low, guess 75, otherwise guess 25, and so on.

In your course in algorithms and data structures, you will find that this is actually a very fast algorithm for finding an entry in an array, being able to do so in $O(\log_2(n))$ operations for an array of capacity n. The issue with numerical analysis is, however, that it is excruciatingly slow as compared to analytical algorithms, and thus is often only the option of last resort.

One example of a bracketing algorithm is finding the root of a function f. Suppose we know that f is continuous and that $f(a) < 0$ and $f(b) > 0$. In that case, there must be a root on the interval (a, b). Guess the midpoint $x_{\text{mid}} = \frac{a+b}{2}$. If $f(x_{\text{mid}}) < 0$, then a root must still be on the interval (x_{mid}, b), while if $f(x_{\text{mid}}) > 0$, then the root must be on (a, x_{mid}). We repeat this algorithm on this new smaller interval.

Each time this algorithm is run, the width of the interval is dropped by half.

Practice questions

1. Using Matlab, approximate a root of $\cos(e^x)$ using the bracketing method starting with the interval (0, 1). In Matlab, you would evaluate this function using `cos(exp(x))`.

Acknowledgements

Michel Georges Najarian
Yousuf Hamdy Ali