

# Contents

<i>Root-finding</i>	5
<i>Systems</i>	31
<i>Curve fitting</i>	51
<i>Numerical integration</i>	71
<i>Initial value problems</i>	89
<i>Two-point boundary value problems</i>	107
<i>Partial differential equations</i>	127
<i>Index</i>	157



# *Introduction*

BROADLY SPEAKING, Numerical Analysis is the use of various algorithms to find numerical approximations to solutions of mathematical problems, as well as estimating how correct these approximations are. These mathematical problems are either too complex or too time-consuming to be solved by hand. Numerical Analysis finds applications in all fields of engineering and science as real-world problems are in most cases complex.

IN THESE NOTES we will explore a number of different mathematical problems and how to approach them numerically. More precisely, we will learn how to numerically solve equations and systems, differentiate and integrate functions, solve differential equations as well as partial differential equations.

A COMMON FEATURE of numerical methods is that they introduce unwanted errors. Since we never can compute an exact solution, it is of utmost importance to have an idea of the size of the error, otherwise our approximation is pretty much useless. How we evaluate the error will depend on the context but will be a central feature in these notes. In many cases, it is difficult to reduce the error significantly without impacting the running time of algorithms, so we will need to strike a good balance between precision and speed.

OUR APPROACH will be rather practical. For each problem we study, we aim to obtain a satisfying theoretical understanding of the relevant numerical methods, why they work and how to control the errors involved. But our main objective is to be able to implement these methods to solve practical examples. A large chunk of the text is therefore devoted to implementation, that is, programming. Example codes will be of two kinds: small self-contained routines which solve an unspecified problem, and extensive solutions to a precise practical problem.

EACH METHOD will therefore be studied from three points of view: theory, error analysis and implementation.



# Root-finding

While solving equations in one variable is one of the simplest mathematical problems, it is both relevant in its own right and serves as an introduction to the concepts of Numerical Analysis which we will use in the following chapters. In mathematical terms, our goal is to find the root of a given function  $f(x)$  in one real variable  $x$ .

**Definition 1.** Let  $f : I \rightarrow \mathbb{R}$  be a function defined on some interval  $I$ . A root of  $f$  is a number  $r$  such that  $f(r) = 0$ .

Graphically speaking, roots are found where the graph of  $f$  intersects the  $x$ -axis. Now not all interesting equations are of the form  $f(x) = 0$  but we will need to rewrite them in that form in order to use the numerical methods in this chapter.

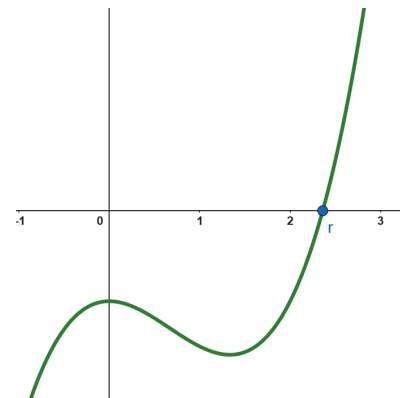
## The bisection method

### Theory

The bisection method is by far the simplest way to numerically find a root of a function  $f(x)$ . Speed is the main issue here as it usually needs a high number of iterations. However its flexibility allows for the bisection method to be usable in almost any example. At the heart of the bisection method is the Intermediate Value Theorem, a version of which we recall here.

**Theorem 1** (Intermediate Value Theorem). Let  $f$  be a continuous function on  $[a, b]$  such that  $f(a)$  and  $f(b)$  have different signs. Then  $f$  has a root between  $a$  and  $b$ .

Looking at Figure 1 we see that  $f(2) < 0$  and  $f(3) > 0$  so that  $f$  has a root on the interval  $[2, 3]$  as stated. Clearly the smaller the interval the better we approximate the root. The bisection method is an iterative method which divides the length of the interval by 2 while ensuring the root stays within the interval. More precisely one step of the bisection method proceeds as follows.



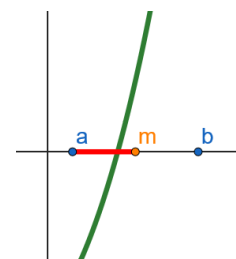
**Figure 1:** This function  $f$  has a root  $r$  between 2 and 3

For instance if our equation is

$$x^3 + x = 100$$

we rewrite it as  $x^3 + x - 100 = 0$  and let  $f(x) = x^3 + x - 100$ .

The intervals  $[1, 3]$  and  $[2, 2.5]$  would work just as well.



**Figure 2:** Root between  $a$  and  $m$  (Step 3)

1. Start with an interval  $I = [a, b]$  on which the root is located.
2. Evaluate  $f$  at the midpoint  $m = \frac{a+b}{2}$ .
3. If  $f(a)$  and  $f(m)$  have opposite signs, set  $I = [a, m]$ ,
4. otherwise if  $f(m)$  and  $f(b)$  have opposite signs, set  $I = [m, b]$ .
5. Reiterate with the new interval  $I$ .

Note that steps 3 and 4 make sure that the root is located on the new interval  $I$ . This process can be iterated as many times as we wish - clearly we get a better and better approximation for the root each step of the way.

### *Implementation*

Following the informal discussion of the algorithm in the previous section, we can now write a program that uses the bisection method to compute an approximation for the root of a function up to a certain precision. The program takes as input three variables:

- Two numbers  $a$  and  $b$  which are the endpoints of the starting interval  $[a, b]$ . As stated before they have to be such that the function has a unique root on the interval. The program performs a small check.
- A number  $\text{tol}$  (short for tolerance). The approximated value returned by the program should be correct within the tolerance. In other words, it is an upper bound for the error.

Once we are satisfied with the precision, what is our best approximation for the root? Without any further information, our best guess will be the midpoint of the last interval. With this choice, the worst-case scenario is for the error to be half the interval length. Therefore we want the algorithm to stop when this half-length is less than the tolerance, the highest acceptable value for the error.

The general algorithm for the bisection method is on the next page.

```

1 function r_approx=bisect(a,b,tol)
2     if sign(f(a))*sign(f(b))>0
3         error('no root on the interval')
4     end
5     fa=f(a);
6     while (b-a)/2 > tol
7         m=(a+b)/2; fm=f(m);
8         if sign(fa)*sign(fm)<0
9             b=m;
10        else
11            a=m; fa=fm;
12        end
13    end
14    r_approx=(a+b)/2;
15 end

```

Lines 2-4 check whether  $f(a)$  and  $f(b)$  have opposite signs, and kills the program if it is not the case.

The loop stops when the half interval length  $(b-a)/2$  is less than the tolerance value.

The program returns the midpoint of the last interval  $[a, b]$

By construction the algorithm returns an approximate root  $r_{\text{approx}}$  such that

$$|r - r_{\text{approx}}| < \text{tol}$$

where  $r$  is the exact value of the root. The program assumes that the function  $f$  is already defined. The simplest way to do so is to define a Matlab function `f.m` which computes  $f(x)$  for any given value of  $x$ . We now look at a simple example. We consider the equation

$$x^2 - 7 = 0$$

which obviously has one positive root  $r = \sqrt{7}$ . We define the function  $f(x) = x^2 - 7$  as a Matlab function:

```

1 function y=f(x)
2     y=x^2-7;
3 end

```

and save this as a function `f.m`. By graphing  $f(x)$  we guess that the root is located on the interval  $[2, 3]$ . The value for the tolerance is for us to choose, for instance we let  $\text{tol} = 10^{-4}$ . We call the program with

```

1 a=2;b=3;tol=10^(-4);
2 r_approx=bisect(a,b,tol)

```

which returns

Be sure that `bisect.m` and `f.m` are located in the same folder. The function doesn't have to be called `f`, but if you choose another name you will need to somewhat modify `bisect.m` so that it calls the right function. Other options: define  $f$  within the bisection program, or have the function be a variable in `bisect`.

Alternatively we see that  $2^2 = 4 < 7$  but  $3^2 = 9 > 7$ .

```
Command Window
r_approx =
fx      2.645690917968750
```

Compared with the actual root

$$r = \sqrt{7} = 2.645751311064591,$$

we indeed see that

$$|r - r_{approx}| = 6.04 \times 10^{-5}$$

which is less than the tolerance. For more precision we simply need to choose a smaller value for tol. For instance if  $\text{tol} = 10^{-10}$  we obtain

```
Command Window
r_approx =
fx      2.645751311036292
```

which is much closer to the true value. However a higher precision requires more iterations which can be expensive would our function  $f(x)$  be more complicated.

$$|r - r_{approx}| = 2.83 \times 10^{-11}$$

### Implementation - More involved example

Typically we will want to study complex functions involving many variables or parameters. While the bisection method does not apply to functions of many variables, we can imagine fixing all parameters except one and let it act as the main variable. As an example consider the Van der Waals equation which describes  $n$  moles of a static gas:

$$\left(P + a \frac{n^2}{V^2}\right) (V - nb) = nRT$$

where

- $P$  is the pressure of the gas in bars,
- $V$  is the volume of the gas in liters,
- $T$  is the temperature in kelvin,

The Van der Waals equation is an extension of the ideal gas law  $PV = nRT$  which takes into account molecular interaction as well as the finite size of the molecules.



- $R$  is the ideal gas constant,  $R = 0.08314 \text{ L} \cdot \text{bar} \cdot \text{K}^{-1} \cdot \text{mol}^{-1}$ ,
- and the constants  $a$  and  $b$  are called the Van der Waals constants and depend on the material. The values for dioxygen  $\text{O}_2$  are  $a = 1.382 \text{ bar} \cdot \text{L}^2 \cdot \text{mol}^{-1}$  and  $b = 0.03186 \text{ L} \cdot \text{mol}^{-1}$ .

It is rather unpleasant to solve the Van der Waals equation for  $V$ . If we assume the temperature and the pressure to be known, we can however use the bisection method to solve for  $V$ . First off we rewrite the equation as

$$\left(P + a \frac{n^2}{V^2}\right)(V - nb) - nRT = 0$$

which we think of as a function of  $V$  while the rest are given parameters. In Matlab we define

```
1 function y=f(V)
2     a=1.382;b=0.03186;
3     T=296;P=1;R=0.08314;n=2;
4     y=(P+a*n^2./V.^2).* (V-n*b)-n*R*T;
5 end
```

The function  $f$  has very little physical meaning. However the correct value of  $V$  for all these parameters is the one such that  $f(V) = 0$ . Therefore we need to find a root of  $f$ . To set up the bisection method we only need to find an interval where  $f$  switches signs. Since we don't have any intuition for the root (except that  $V > 0$ ) we choose a rather random interval  $[0, 100]$ . The graph seems to show that the root lies between 40 and 60 which we use as our first interval  $[a, b]$ . We could certainly do better but this is completely satisfactory.

We run the bisection method for these values of  $a$  and  $b$  and a small enough tolerance:

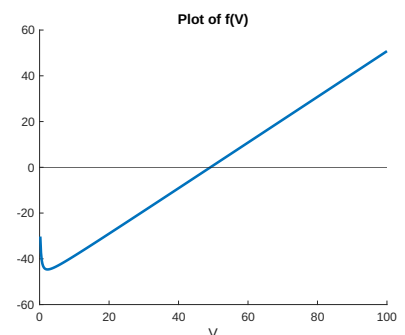
```
1 a=40;b=60;tol=10^(-4);
2 V_approx=bisect(a,b,tol)
```

which returns

```
Command Window
V_approx =
fx      49.1703
```

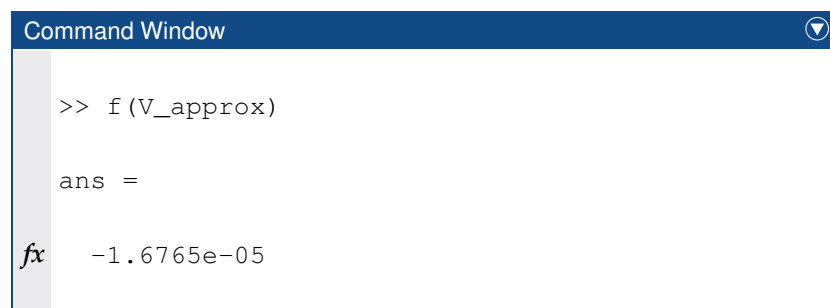
A standard approximation would be to use the ideal gas law to write  $\frac{n^2}{V^2} \simeq \frac{nRT}{P}$  but we will not go that route.

Later we will set up a more modular approach which allows us to change the parameters easily. We do prepare for this already by using  $.*$  multiple times.



**Figure 4:** A plot of  $f(V)$  on the interval  $[0, 100]$ . We see that a root lies on in the interval  $[40, 60]$ . The graph doesn't show well that  $f(V) \rightarrow -\infty$  when  $V \rightarrow 0^+$ .

This certainly matches Figure 4. As a further check we calculate the value of  $f(V_{\text{approx}})$  by plugging into  $f$ . Hopefully we get an answer which is close to  $f(V_{\text{exact}}) = 0$ .



```

Command Window
>> f(V_approx)

ans =

fx    -1.6765e-05

```

The answer is indeed very close to 0. Note that we have two ways to compute the error. The first is the **backward error**

$$|V_{\text{exact}} - V_{\text{approx}}|,$$

which tells us how far from the root our approximation is. By construction, this error will always be lower than the tolerance. The second is the above **forward error**

$$|f(V_{\text{approx}})|,$$

which tells us how close  $V_{\text{approx}}$  is to solving the equation  $f(V) = 0$ . The forward error is easily obtained since it does not involve the unknown exact solution. Unfortunately there is no general formula linking the two types of errors.

The shape of the function  $f$  matters. For instance if the graph  $f$  is very flat near the root we can have  $f(x) \approx 0$  for  $x$  relatively far away from the root.

This method works but if we change the value of one parameter we need to start from scratch. Let us imagine that all parameters except  $P$  are fixed and we wish to find the volume  $V$  for a range of values for  $P$  (say  $P = 1, 2, \dots, 10$ ). To do so requires some modularity in the definition of the function  $f(V)$ . For maximal modularity we define:

```

1 function y=g(V,a,b,T,P,R,n)
2     y=(P+a*n^2./V.^2).* (V-n*b)-n*R*T;
3 end

```

Note that we do not call this function  $f$  as the name is reserved for functions of one variable that the bisection method works on.

as a function of seven variables. Now the bisection method will not understand what to do with such a function and we must specify that  $V$  is the main variable here. The following code snippet does exactly that.

```

1 a=1.382;b=0.03186;T=296;R=0.08314;n=2;
2 for P=1:10
3     f=@(V) g(V,a,b,T,P,R,n);
4 end

```

$f = @(V) \dots$  defines the right-hand side as function of the variable  $V$  in Matlab. The technical term is "function handle". If for instance  $P = 1$  this  $f$  is precisely the same  $f$  as before.

We need to slightly modify `bisect.m` so that it takes the function  $f$  as a variable (since the function is moving with  $P$ ). With this in mind we can calculate the volume  $V$  for  $P = 1, 2, \dots, 10$  and plot it as a function of  $P$ . We use a starting interval of  $[1, 1000]$  which should be large enough and  $\text{tol} = 0.0001$ .

The only change is for the first line to become `function r=bisect(f,a,b,tol)`.

```

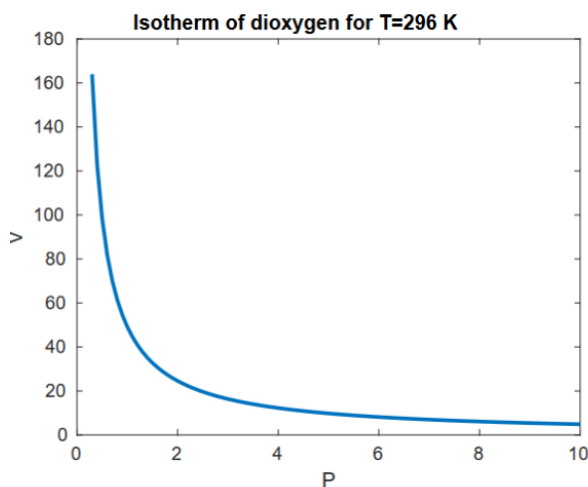
1 a=1.382;b=0.03186;T=296;R=0.08314;n=2;
2 for P=1:10
3     f=@(V) g(V,a,b,T,P,R,n);
4     V_approx(P)=bisect(f,1,1000,0.0001);
5 end

```

which calculates the vector

Command Window							
V_approx =							
	49.1704	24.5609	16.3578	12.2563	9.7953	8.1548	6.9829
<i>fx</i>	6.1041	5.4205	4.8737				

which for  $P = 1, 2, \dots, 10$  displays the corresponding value of  $V$ . Using a finer set of values for  $P$  we obtain the following graph showing  $V$  as a function of  $P$ .



**Figure 5:** Relationship between pressure  $P$  and volume  $V$  at constant temperature  $T = 296$  K. Such a graph is called an isotherm.

### Error Analysis and Potential Problems

The bisection method allows for precise monitoring of the error. That is because the intervals shrink in a systematic way at each iteration. If the starting interval  $[a, b]$  has length  $b - a$ , then after  $n$  steps we are left with an interval of length

$$\frac{b - a}{2^n}$$

Since the maximum error is half the length of the final interval, we can state the following:

**Theorem 2.** *After  $n$  iterations of the bisection method with starting interval  $[a, b]$  the error is at most*

$$\frac{b - a}{2^{n+1}}$$

This is a perfect situation where the error is completely in our control. Unfortunately we will see that this is rather untypical. To better compare the bisection method with other root-finding methods we consider the ratio of the errors in two consecutive steps. If we let  $e_n$  be the maximal error at stage  $n$  it should be clear that

$$\frac{e_{n+1}}{e_n} = \frac{1}{2}$$

Generally speaking, a numerical method such that

$$\lim_{n \rightarrow \infty} \frac{e_{n+1}}{e_n} = L$$

where  $L$  is a constant is said to have **linear rate of convergence**. Such methods are such that the gains in precision in each step are constant or nearly so. We will shortly see that other methods can improve the rate of convergence tremendously.

A good rule of thumb is that one needs three to four iterations to obtain an extra digit.

AS SIMPLE AS the bisection method is there are however three main pitfalls one needs to be careful about. It mostly has to do with the choice of the initial interval  $[a, b]$ .

1. The interval  $[a, b]$  does not contain a root. This can be solved by either trial and error or graphing the function to find  $a$  and  $b$  such that  $f(a)$  and  $f(b)$  have opposite signs.
2. The interval  $[a, b]$  contains more than one root. In that case we lose control over which root the bisection method is trying to approximate. This can be solved by choosing a smaller starting interval containing only the one root we are interested in.

To be more precise,  $e_n$  is the absolute value of the error - we usually do not care whether we overshoot or undershoot.

The word linear refers here to the fact that  $e_{n+1}$  is (approximately) a linear function of  $e_n$ .

After three iterations, the maximal error has been divided by  $2^3 = 8$  while an extra digit requires dividing it by 10.

3. The tolerance is too high. This is especially important if the starting interval is relatively small. For instance if we start working on  $[0.4, 0.5]$  and use a tolerance of 0.2 the algorithm will stop immediately.

### Newton's method

The bisection method has two immediate drawbacks: its slow linear speed, and its inability to scale to functions with many variables. Newton's method is a way to solve both problems, however it comes at a cost.

#### Theory

Newton's method is graphically simple. The setting is the same as before, namely find a root  $r$  of a function of one variable  $f(x)$ . The first step of Newton's method proceeds as follows:

1. Start with some initial guess  $x_0$  for the root.
2. Draw the tangent line to  $f(x)$  at  $(x_0, f(x_0))$ .
3. The tangent intersects the  $x$ -axis at  $x_1$ .
4. Use  $x_1$  as a better approximation for the root and reiterate with  $x_1$  instead of  $x_0$ .

In essence we replace  $f(x)$  by its linear approximation  $L(x)$  at  $x_0$  (that is, the tangent) and find a root for  $L(x)$ . In order to compute  $x_1$  from  $x_0$ , recall the equation of the tangent at  $x_0$  is

$$y = f'(x_0)(x - x_0) + f(x_0).$$

To find the intersection point with the  $x$ -axis we let  $y = 0$ :

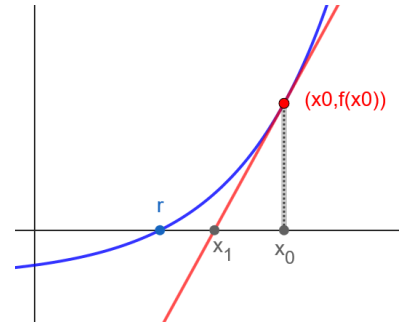
$$0 = f'(x_0)(x_1 - x_0) + f(x_0) \Rightarrow x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$$

which is our formula for  $x_1$ . The next step is to iterate this construction, as seen on Figure 7. The same calculation as above yields

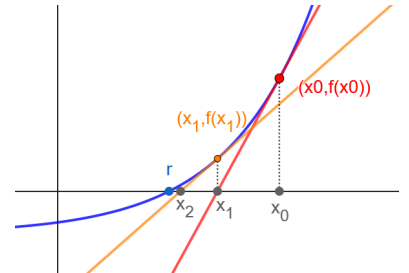
$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}.$$

This leads to the following definition.

As the algorithm enters the while loop it stops whenever  $(b - a)/2 < \text{tol}$ . Here we have  $(b - a)/2 = 0.05$  while  $\text{tol} = 0.2$ .



**Figure 6:** One step of Newton's method. In blue we have the graph of  $f(x)$  and its root  $r$ . The tangent to  $f(x)$  at  $x_0$  intersects the  $x$ -axis at  $x_1$ .



**Figure 7:** Two steps of Newton's method. We draw the tangent at  $(x_1, f(x_1))$  which intersects the  $x$ -axis at  $x_2$ .

**Definition 2.** Let  $f : \mathbb{R} \rightarrow \mathbb{R}$  be a function and  $x_0 \in \mathbb{R}$ . Newton's method is defined inductively by

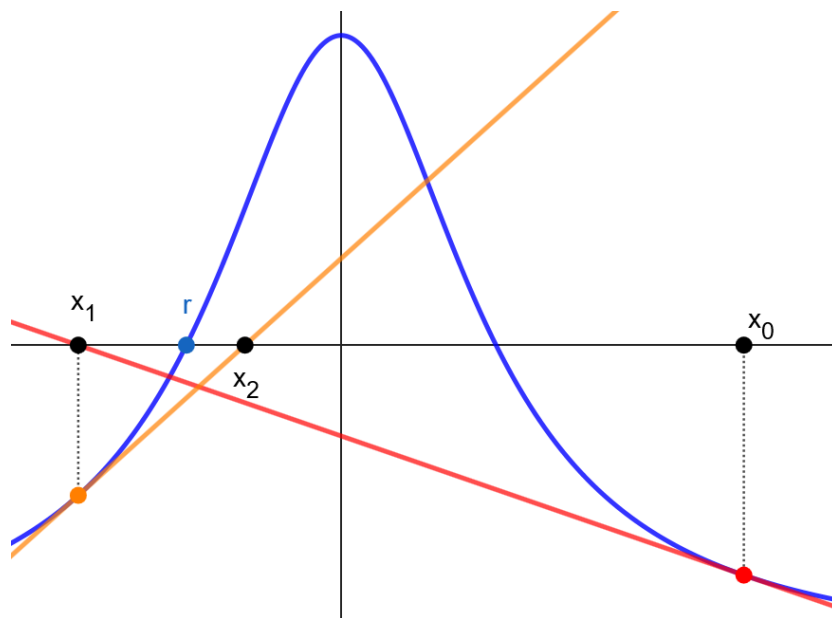
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n \geq 0$$

provided  $f'(x_n) \neq 0$ .

Figure 7 seems to suggest that Newton's method always converges to a root  $r$  of  $f(x)$ . This is unfortunately not true and Newton's method can fail in a number of different ways.

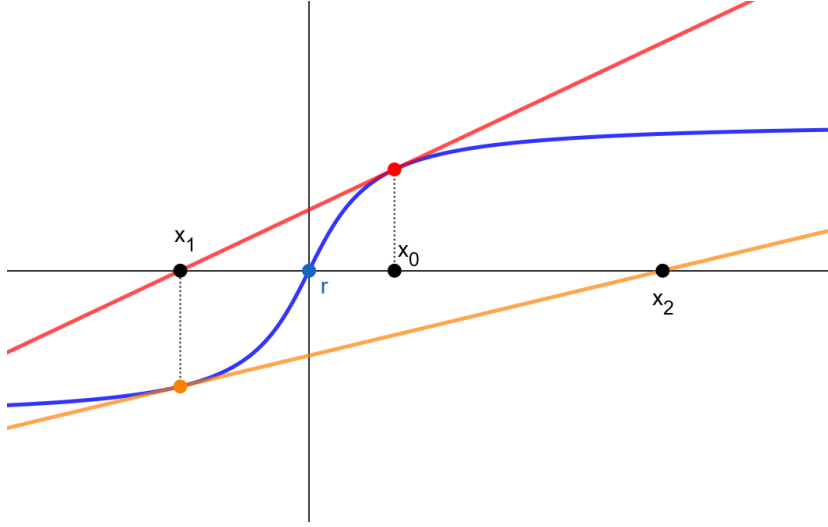
If  $f(x)$  has two or more roots, Newton's method does not necessarily converge to the one which is closest to the initial guess  $x_0$ , as shown on Figure 8.

Compare to the Bisection Method which always converges to a root.



**Figure 8:** Two steps of Newton's method for  $f(x) = \frac{1}{1+x^2} - \frac{1}{2}$  with initial guess  $x_0 = 2.6$ . Newton's method converges to the negative root  $r = -1$ . However if  $x_0 = 1.7$  it converges to the positive root  $r = 1$ .

Newton's method can also diverge to  $\infty$  or  $-\infty$  as shown on Figure 9. Finally, Newton's method can get stuck in a loop, end up outside the domain of the function or suddenly stop if  $f'(x_n) = 0$ . These facts



**Figure 9:** Two steps of Newton's method for  $f(x) = \arctan(x)$  with initial guess  $x_0 = 2$ . One can show that  $|x_n| \rightarrow \infty$ , alternating signs. Note that the tangent at  $x_2$  has a very small (positive) slope so that  $x_3$  will be a large negative number. If  $x_0 = 1$ , Newton's method converges quickly to the unique root  $r = 0$ .

all point towards the importance of the choice of  $x_0$  which should be close enough to the root.

**Theorem 3.** Let  $f : \mathbb{R} \rightarrow \mathbb{R}$  be a function with a root  $r$  such that  $f$ ,  $f'$  and  $f''$  are continuous. If  $x_0$  is chosen near enough  $r$  then Newton's method converges to  $r$ .

The theorem proves that Newton's method is **locally convergent**.

*Proof.* Recall that Newton's method is defined through

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

We define a new function

$$g(x) = x - \frac{f(x)}{f'(x)}$$

so that Newton's method can be rewritten as  $x_{n+1} = g(x_n)$ . Since  $f(r) = 0$  we have

$$g(r) = r.$$

Moreover the quotient differentiation rule gives

$$g'(x) = 1 - \frac{f'(x)f'(x) - f(x)f''(x)}{f'(x)^2}.$$

Again using  $f(r) = 0$  we obtain:

**Objective:** prove that  $x_n \rightarrow r$  if  $x_0$  is chosen near enough  $r$ .

This is an example of a fixed-point iteration and the equation  $g(r) = r$  is precisely saying that  $r$  is a fixed point of  $g$ .

$f(r) = 0$  because  $r$  is a root.

$$g'(r) = 1 - \frac{f'(r)f'(r) - 0 \cdot f''(r)}{f'(r)^2} = 1 - 1 = 0.$$

Since  $g'(r) = 0$  and  $g'$  is continuous, there exists an interval  $I$  near  $r$  where

$$|g'(x)| \leq \frac{1}{2}, \quad x \in I$$

and in the following we assume that  $x_0$  is in  $I$  - this is what the theorem means by **chosen near enough**.

Now consider the first step of Newton's method, rewritten as

$$x_1 = g(x_0).$$

Subtracting  $r$  on both sides gives:

$$x_1 - r = g(x_0) - r = g(x_0) - g(r).$$

The mean value theorem gives a way to involve  $g'$ : there exists  $c$  in between  $x_0$  and  $r$  such that

$$x_1 - r = g'(c)(x_0 - r).$$

We take absolute values:

$$|x_1 - r| = |g'(c)||x_0 - r|$$

and use that  $|g'(c)| \leq \frac{1}{2}$ . Therefore

$$|x_1 - r| \leq \frac{1}{2}|x_0 - r|.$$

In particular  $x_1$  is closer to  $r$  than  $x_0$  is and belongs to the same small interval  $I$ . This allows us to repeat the argument for  $x_2, x_3$  and so forth, obtaining:

$$|x_n - r| \leq \frac{1}{2}|x_{n-1} - r|$$

for any  $n \geq 1$ . But then we obtain

$$|x_n - r| \leq \frac{1}{2}|x_{n-1} - r| < \frac{1}{2} \cdot \frac{1}{2}|x_{n-2} - r| \leq \cdots \leq \frac{1}{2^n}|x_0 - r|.$$

If we let  $n \rightarrow \infty$  we obtain immediately that  $x_n \rightarrow r$  and Newton's method is convergent provided  $x_0$  is near enough  $r$ .  $\square$

### Implementation

So far we have described how Newton's method is iterated but have not discussed a stopping condition. The simplest way is to compare the last two iterations and stop iterating when the difference is small enough, that is smaller than the tolerance. A basic implementation of Newton's method is as follows. The program takes as input two variables: the initial guess  $x_0$  and the tolerance  $\text{tol}$ .

Intuitively: continuous functions can not change abruptly. There is nothing special about the number  $1/2$ , and we could use any number  $< 1$ .

Recall  $g(r) = r$ .

Mean value theorem: If  $f$  is continuous on  $[a, b]$  and differentiable on  $]a, b[$  there exists  $c$  between  $a$  and  $b$  such that  $f(b) - f(a) = f'(c)(b - a)$ . We use  $a = r$  and  $b = x_0$ .

Since  $c$  is trapped between  $x_0$  and  $r$  it must belong to the same interval  $I$  where  $g'$  is smaller than  $1/2$ .

This proves that Newton's method does not escape the small interval  $I$  around the root and therefore does not misbehave as in Figures 8 and 9.

$1/2^n \rightarrow 0$  while  $|x_0 - r|$  is constant.

$x_n$  and  $x_{n+1}$  in the language of the previous section



```

1 function r_approx=newton(x0,tol)
2     x=x0;oldx=x0+2*tol;
3     while abs(x-oldx) > tol
4         oldx=x;
5         x=x-f(x)/Df(x);
6     end
7     r_approx=x;
8 end

```

$x$  takes the role of  $x_{n+1}$  and  $oldx$  that of  $x_n$ . The initial value of  $oldx$  ensures that we enter the loop since  $x-oldx$  is twice the tolerance.

As for the bisection method, the program assumes that  $f$  and  $f'$  (coded as  $Df$ ) are defined as functions. We could decide to define them within `newton.m` or have the functions as inputs. Since Newton's method is not guaranteed to converge, such a single program presents a potential risk of getting stuck in an endless loop. A simple fix is to force exit after a certain number of iterations (say 30) and display a message. As Theorem 3 shows, a failure of Newton's method is an indication of a badly chosen initial value  $x_0$ .

A more advanced method would be to relax the tolerance if no answer is found within a few iterations. If Newton's method still diverges at low precision, the program could print an error message.

```

1 function r_approx=newton(x0,tol)
2     x=x0;oldx=x0+2*tol;
3     counter=1;maxiterations=30;
4     while abs(x-oldx) > tol
5         oldx=x;
6         x=x-f(x)/Df(x);
7         counter=counter+1;
8         if counter>maxiterations
9             disp('Newton method unsuccessful')
10            break
11        end
12    end
13    r_approx=x;
14 end

```

Let us now solve again the Van der Waals equation using Newton's method and compare its performance with that of the bisection method. Recall that we wish to solve

$$\left(P + a \frac{n^2}{V^2}\right)(V - nb) - nRT = 0$$

The equation is  $f(V) = 0$ .

for  $V$ , all other parameters being known. We use the same  $f$  subroutine as before.

```

1 function y=f(V)
2     a=1.382;b=0.03186;
3     T=296;P=1;R=0.08314;n=2;
4     y=(P+a*n^2./V.^2).* (V-n*b)-n*R*T;
5 end

```

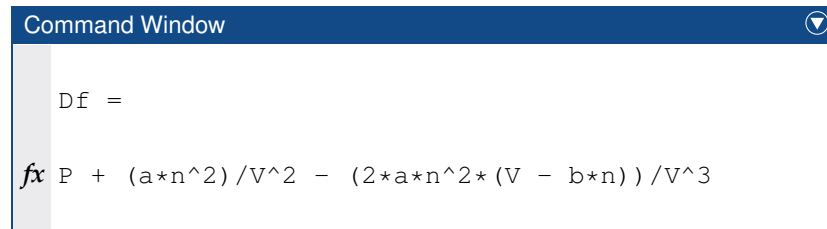
A new feature of Newton's method is that we need to know the derivative  $f'(V) = 0$  as well. Whether we choose to compute  $f'(V)$  by hand or using symbolic calculations in Matlab, it is very important to hard code it is a function as well. Letting Matlab do the menial work, we obtain

```

1 syms a b T n P V R
2 Df=diff( (P+a*n^2/V^2)*(V-n*b)-n*R*T,V)

```

which computes



Command Window

Df =

$$f_x P + (a*n^2)/V^2 - (2*a*n^2*(V - b*n))/V^3$$

which we take as an expression for  $f'(V)$ . We immediately hardcode it:

```

1 function y=Df(V)
2     a=1.382;b=0.03186;
3     T=296;P=1;R=0.08314;n=2;
4     y=P + (a*n^2)/V^2 - (2*a*n^2*(V - b*n))/V^3;
5 end

```

We are finally ready to run Newton's method:

```

1 x0=40;tol=0.0001;
2 V_approx=newton(x0,tol)

```

The answer is thankfully the same as when we used the bisection method.

Symbolic calculations in any language are very time consuming and we can not afford to differentiate the same function repeatedly.

Line 1 defines all parameters as symbolic variables while Line 2 differentiates the expression  $f(V)$  with respect to  $V$ . This remains valid however the parameters are chosen.

Straight copy and paste from the command window.

The initial value  $x_0 = 40$  is chosen by reading the graph of  $f(V)$ , see Figure 4.



```
Command Window
V_approx =
fx      49.1703
```

### Comparison of Newton's method and the Bisection method

Since Newton's method is more complicated to set up, we expect a payoff in terms of speed. We slightly modify both `bisect.m` and `newton.m` so that both routines count the number of iterations needed in order to obtain an error under the same tolerance.

The difference ends up being substantial. Results are summarized in Table 1. The bisection method exhibits clearly its linear rate of convergence, needing three to four iterations to gain a digit. Newton's method is vastly outperforming it. We study in more detail just how quickly Newton's method converges in the following section.

Tolerance	Bisection	Newton
$10^{-1}$	7	2
$10^{-2}$	10	2
$10^{-3}$	14	3
$10^{-4}$	17	3
$10^{-5}$	20	3
$10^{-6}$	24	3
$10^{-7}$	27	3
$10^{-8}$	30	3
$10^{-9}$	34	4

**Table 1:** A speed comparison between the bisection method and Newton's method when solving the Van der Waals equation. Column 1 shows the tolerance, Column 2 the number of iterations needed for the Bisection method while Column 3 shows the number of iterations needed by Newton's method. Starting values are kept constant.

### Newton's method - Error analysis

In order to showcase the speed of Newton's method, we will study a simple case where an exact solution is known. Our equation is

$$f(x) = x^3 - x^2 + x - 1 = 0$$

and its only root  $r = 1$ . We can easily set up Newton's method as before, using  $f'(x) = 3x^2 - 2x + 1$ . The next step is to run Newton's method with initial value  $x_0 = 3$  and let the program print out all iterations  $x_n$  (not just the final one). We also compute the error at stage  $n$

$$e_n = |x_n - r|$$

as well as the ratio  $e_{n+1}/e_n$  between successive errors (for  $n \geq 1$ ).

Note that  $f(x) = (x - 1)(x^2 + 1)$  so 1 is the only root.

$n$	$x_n$	$e_n =  x_n - r $	$e_{n+1}/e_n$
0	2.0000000000000000	1.0000000000000000	0.4444444444444444
1	1.4444444444444444	0.4444444444444444	0.293785310734463
2	1.130571249215317	0.130571249215317	0.114726269151389
3	1.014979952280910	0.014979952280910	0.014757257941487
4	1.000221063019761	0.000221063019761	0.000221014156897
5	1.000000048858057	0.000000048858057	0.000000049991563
6	1.0000000000000002	0.0000000000000002	N/A

**Table 2:** First six iterations of Newton's method for  $f(x) = x^3 - x^2 + x - 1$  with initial guess  $x_0 = 2$ .

Recall that linear rate of convergence means  $e_{n+1}/e_n$  is approximately constant, while Table 2 shows that it tends quicker and quicker to 0.

Table 2 displays a much faster rate of convergence than linear. We see that one digit at correct at step 3, three at step 4, seven at step 5, and fourteen at step 6. This suggests that the number of correct digits doubles at each step of the way. The following theorem explains why this is the case.

**Theorem 4.** *Let  $f$  be a function such that  $f$ ,  $f'$  and  $f''$  are continuous. Assume Newton's method*

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

*converges to a root  $r$  such that*

$$f'(r) \neq 0$$

*and let  $e_n = |x_n - r|$  be the error at stage  $n$ . Then there exists a constant  $M$  such that*

$$\lim_{n \rightarrow \infty} \frac{e_{n+1}}{e_n^2} = M$$

Whether or not Newton's method converges is covered in Theorem 3 and we assume here that everything works out.

Note the new condition  $f'(r) \neq 0$ . If  $f'(r) = 0$ , Newton's method still converges (locally) but does so linearly, not quadratically. Graphically it makes sense that if the graph of  $f$  lies very flat around the root we would lose accuracy.

Theorem 4 shows that Newton's method exhibits a quadratic (second order) rate of convergence, which is a tremendous improvement on linear.

Before we prove Theorem 4 we show how it implies that the number of correct digits is doubled at each step of Newton's method. Imagine that the error  $e_n$  at stage  $n$  is well under  $10^{-4}$ . This would suggest that the first four digits are correct. The Theorem then shows that

$$e_{n+1} \simeq M e_n \ll M \cdot 10^{-8}$$

If the constant  $M$  is not too large, this suggests eight correct digits.

Second order because of the  $e_n^2$  term.

This is far from rigorous, for a better proof we would need to define precisely what we mean by  $k$  correct digits.

Now we change Table 2 by having  $e_{n+1}/e_n^2$  in the rightmost column.

$n$	$x_n$	$e_n =  x_n - r $	$e_{n+1}/e_n^2$
0	2.000000000000000	1.000000000000000	0.444444444444444
1	1.444444444444444	0.444444444444444	0.661016949152543
2	1.130571249215317	0.130571249215317	0.878648782491166
3	1.014979952280910	0.014979952280910	0.985133841867682
4	1.000221063019761	0.000221063019761	0.999778964101382
5	1.000000048858057	0.000000048858057	1.023199992115598
6	1.000000000000000	0.000000000000000	N/A

This seems to agree with Theorem 4 with  $M = 1$ . The discrepancy at the very last step can be explained by machine precision issues. We now proceed with the proof of Theorem 4.

*Proof.* As in the proof of Theorem 3 we let

$$g(x) = x - \frac{f(x)}{f'(x)}$$

so that Newton's method is to be thought of as  $x_{n+1} = g(x_n)$ . Earlier we proved  $g(r) = r$  and  $g'(r) = 0$ . In order to study the error better, we will also need the value of  $g''(r)$ . Recall that

$$g' = 1 - \frac{f'^2 - f \cdot f''}{f'^2} = \frac{f \cdot f''}{f'^2}.$$

Differentiating once more gives

$$g'' = \frac{f'^2(f \cdot f''' + f' \cdot f'') - 2f \cdot f' \cdot f''^2}{f'^4}.$$

At  $x = r$  we have  $f(r) = 0$  and therefore

$$g''(r) = \frac{f''(r)}{f'(r)}.$$

Our next step is to write down the second-order Taylor approximation for  $g$  at  $r$ . With  $g(r) = r$ ,  $g'(r) = 0$  and  $g''(r)$  as above, we obtain

$$g(x) = r + \frac{f''(r)}{2f'(r)}(x - r)^2 + O((x - r)^3).$$

With this in hand we are ready to prove the theorem. We use the Taylor approximation at  $x = x_n$ . Since by definition  $g(x_n) = x_{n+1}$  we obtain

**Table 3:** First six iterations of Newton's method for  $f(x) = x^3 - x^2 + x - 1$  with initial guess  $x_0 = 2$  in light of Theorem 4. It seems plausible that  $\lim_{n \rightarrow \infty} \frac{e_{n+1}}{e_n^2} = 1$ .

We write  $f$  instead of  $f(x)$  and so on for readability's sake.

This is a thoroughly uninteresting calculation and the reader is invited to fill in the blanks.

Here we use the assumption  $f'(r) \neq 0$ . All terms in the numerator are 0 except the one involving  $f' \cdot f''$ .

In general we have  
 $g(x) = g(c) + g'(c)(x - c) + \frac{1}{2}g''(c)(x - c)^2 + O((x - c)^3)$   
 around  $c$  (here  $c = r$ ).

That is the  $x_n$  calculated by Newton's method

Notice the sudden appearance of  $x_n - r$  and  $x_{n+1} - r$ , the absolute values of which are  $e_n$  and  $e_{n+1}$ .

$$x_{n+1} - r = \frac{f''(r)}{2f'(r)}(x_n - r)^2 + O((x_n - r)^3).$$

Dividing the equation by  $(x_n - r)^2$  and taking absolute values gives:

$$\frac{|x_{n+1} - r|}{|x_n - r|^2} = \left| \frac{f''(r)}{2f'(r)} + O(x_n - r) \right|.$$

The left-hand side is simply  $e_{n+1}/e_n^2$  as in the theorem. As  $n \rightarrow \infty$  we have  $x_n - r \rightarrow 0$  and therefore

$$\lim_{n \rightarrow \infty} \frac{e_{n+1}}{e_n^2} = \left| \frac{f''(r)}{2f'(r)} \right|$$

and the theorem is proved with

$$M = \left| \frac{f''(r)}{2f'(r)} \right|.$$

□

In our previous example we had  $f(x) = x^3 - x^2 + x - 1$  with root  $r = 1$ . Since  $f'(x) = 3x^2 - 2x + 1$  and  $f''(x) = 6x - 2$  we see that

$$f'(r) = 2, \quad f''(r) = 4$$

and therefore

$$M = \left| \frac{4}{2 \cdot 2} \right| = 1$$

in agreement with Table 3.

### Golden section search - Finding minima

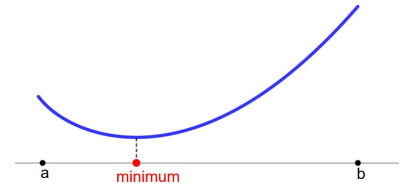
Our next topic is only tangentially related to root-finding, however the idea will prove somewhat similar to the bisection method. Given a function  $f$  defined on an interval  $[a, b]$ , can we easily find where  $f$  takes its minimum value? This is an important problem in optimization theory where we usually want to minimize or maximize some output of our system. We will assume our function has a simple structure with a single minimum value as shown on Figure 10, such functions are called **unimodal**.

#### Theory

If the function  $f$  is simple enough to be differentiated, we can just compute  $f'(x)$  and apply any of this chapter's numerical methods to find where  $f'(x) = 0$ . What if  $f'(x)$  is too complex to be computed? Can we still find the position of the minimum by simply calculating values of  $f$ ? The basic idea is as follows, starting on an interval  $[a, b]$ .

Because the Newton approximations  $x_n$  converge to the root.

It is indeed a constant, although not one that we can easily compute beforehand.



**Figure 10:** This function  $f$  is an example of a unimodal function. We assume  $f$  is decreasing up until the minimum value, and then increases.

There is an obvious way to do this, simply by calculating many values for  $f(x)$  and finding the smallest one. However we need thousands of values to even achieve poor accuracy, and since we assume  $f$  is complex, we would want to call  $f$  as seldom as possible.

1. Choose two points  $x_1$  and  $x_2$  on the interval.
2. Compute  $f(x_1)$  and  $f(x_2)$  and compare the values.
3. If  $f(x_1) < f(x_2)$  then the minimum value cannot be on the right of  $x_2$  since the function has already increased between  $x_1$  and  $x_2$ . So we only consider the interval  $[a, x_2]$ , see Figure 11
4. If  $f(x_1) > f(x_2)$  then the minimum value cannot be on the left of  $x_1$  since the function still decreases between  $x_1$  and  $x_2$ . So we only consider the interval  $[x_1, b]$ , see Figure 12.
5. Repeat with whichever interval comes out of Steps 3 and 4.

Steps 3 and 4 ensure that the new interval is smaller, and if we always choose  $x_1$  and  $x_2$  in the same manner we see that the interval length will be multiplied by a fixed number at each step. This is very similar to how the bisection method works, so we expect a linear rate of convergence.

It remains to decide how to choose  $x_1$  and  $x_2$  in the most efficient manner. We have the two following conditions in mind:

- $x_1$  and  $x_2$  should be symmetric with respect to the midpoint of the interval.
- Values should be used more than once. Looking at Figure 11 we would like to use  $f(x_1)$  at the next iteration. On Figure 12 we would reuse  $f(x_2)$ . This is for efficiency's sake since we assume that it is time-consuming to compute values of  $f$ .

To simplify things assume that the first interval  $[a, b]$  is  $[0, 1]$ . Symmetry is achieved by requiring

$$x_1 = 1 - x_2,$$

so we only need to figure out the position of  $x_2$ . The lower interval on Figure 13 has length  $x_2$  and if the points are chosen using the same ratio twice we must have

$$x_1 = \text{Length of interval} \times \text{Ratio} = x_2 \cdot x_2 = x_2^2.$$

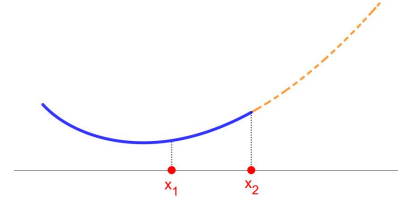
Plugging in  $x_1 = 1 - x_2$  we obtain the equation

$$1 - x_2 = x_2^2 \Rightarrow x_2^2 + x_2 - 1 = 0.$$

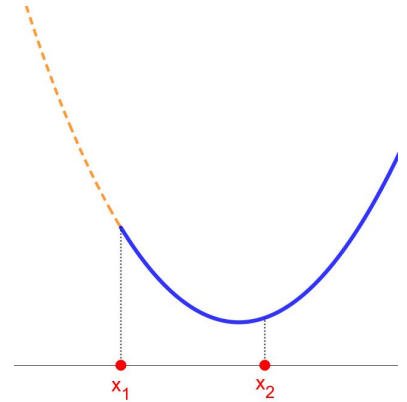
This is a simple quadratic equation with two roots

$$x_2 = \frac{-1 \pm \sqrt{5}}{2}.$$

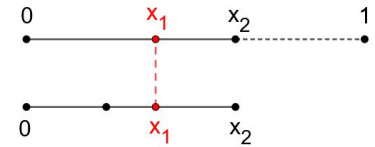
More later on how to choose  $x_1$  and  $x_2$  in an efficient and systematic fashion.



**Figure 11:** Case where  $f(x_1) < f(x_2)$ . The minimum cannot be on the right of  $x_2$  since the function has started increasing already.



**Figure 12:** Case where  $f(x_1) > f(x_2)$ . The minimum cannot be on the left of  $x_1$  since the function is still decreasing at that point.



**Figure 13:** How to choose  $x_1$  and  $x_2$  so that the red point  $x_1$  becomes the second point in the next iteration?

Evidently only the positive root makes sense so we choose

$$x_2 = \frac{-1 + \sqrt{5}}{2} := \varphi \quad x_1 = 1 - \varphi.$$

In the case of a general interval  $[a, b]$ , we use the same ratios  $\varphi$  and  $1 - \varphi$  which leads to

$$x_2 = a + \varphi(b - a) \quad x_1 = a + (1 - \varphi)(b - a).$$

### Implementation

The golden section search is set up very much like the bisection method, needing only a starting interval  $[a, b]$  and an error tolerance  $\text{tol}$ . The output value is the midpoint of the last computed interval, so we want half its length to be less than the tolerance value. We need to be careful in reusing previously computed values as much as possible.

```

1 function min_approx=goldensearch(a,b,tol)
2     phi=(sqrt(5)-1)/2;
3     x1=a+(1-phi)*(b-a);
4     x2=a+phi*(b-a);
5     f1=f(x1);f2=f(x2);
6     while (b-a)/2>tol
7         if f1<f2
8             b=x2; x2=x1; x1=a+(1-phi)*(b-a);
9             f2=f1; f1=f(x1);
10        else
11            a=x1; x1=x2; x2=a+phi*(b-a);
12            f1=f2; f2=f(x2);
13        end
14    end
15    min_approx=(a+b)/2;
16 end

```

$x_2$  is the famous golden ratio  $\varphi$  which gives the algorithm its name.

The program assumes  $f$  to be defined somewhere else, and assumes  $f$  has an unique minimum on  $[a, b]$ .

Lines 7-9: case  $f(x_1) < f(x_2)$ , so the next interval is  $[a, x_2]$ . The new  $x_2$  is the old  $x_1$  and we reuse  $f_1$ . A new  $x_1$  has to be computed from scratch. See Figure 13.

Lines 10-12: case  $f(x_1) > f(x_2)$ , so the next interval is  $[x_1, b]$ . The new  $x_1$  is the old  $x_2$ , a new  $x_2$  has to be computed from scratch.

We test it on the function

$$f(x) = x^2 - \ln(x+1)$$

which appears to be unimodal and have a minimum between 0 and 1. We define the function in Matlab:

```

1 function y=f(x)
2 y=x^2-log(x+1);
3 end

```



and run the golden section search method:

```
1 a=0;b=1;tol=10^(-10);
2 min_approx=goldensearch(a,b,tol)
```

yielding

```
Command Window
min_approx =
fx      0.366025407537693
```

which is confirmed on Figure 14.

### Implementation - More involved example

As mentioned before, we will typically use the golden section search when  $f(x)$  is a much more complicated function. Such functions are usually the outcome of a lengthy piece of code and are not defined in a straightforward way as a function of one variable. We will encounter such functions in the next chapters, for instance as the solution of a system or a differential equation solved numerically.

### Error analysis

The situation is very similar to the bisection method and Theorem 2, the only difference being by how much the interval shrinks at every iteration. Figure 13 shows clearly that the shrinking factor is the golden ratio  $\varphi = \frac{-1 + \sqrt{5}}{2}$ . We can therefore state the following:

**Theorem 5.** After  $n$  iterations of the golden section search with starting interval  $[a, b]$ , the error is at most

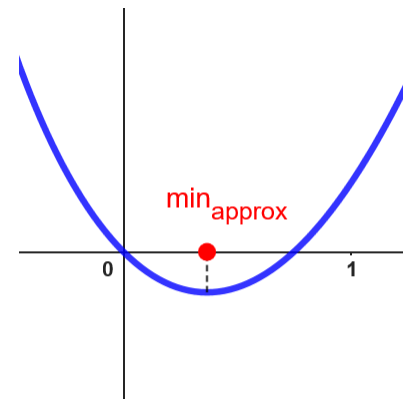
$$\frac{b-a}{2} \varphi^n$$

where  $\varphi = \frac{-1 + \sqrt{5}}{2} \approx 0.61$ .

Similarly it should be clear that

$$\frac{e_{n+1}}{e_n} = \varphi$$

if  $e_n$  is the maximum error at stage  $n$ , showcasing a linear rate of convergence.



**Figure 14:** A graph of  $f(x) = x^2 - \ln(x+1)$  showing the minimum at approximately 0.366.

otherwise we would solve  $f'(x) = 0$  for instance by the bisection method

unlike  $f(x) = x^2 - \ln(x+1)$

The extra division by 2 is because the program returns the midpoint of the last interval, the error being then at most half its length.

### Other root-finding methods (optional material)

Newton's method and the bisection method lie on opposite sides of the spectrum. The bisection method is relatively slow and reliable, while Newton's speed is much quicker but not guaranteed to converge. The key difference is that we do not need to compute  $f'(x)$  in order to use the bisection method. This has not mattered so far, but some functions are just too complicated to be differentiated. Do we have to settle for the sluggish bisection method?

#### The secant method

The secant method takes inspiration from Newton's method. Instead of using the tangent line (obtained through differentiation) we draw a line through two points on the graph of  $f$  and find where it intersects the  $x$ -axis, see Figure 15. The first step of the secant method is therefore as follows:

1. Start with two initial guesses  $x_0$  and  $x_1$  for the root.
2. Draw the line through  $(x_0, f(x_0))$  and  $(x_1, f(x_1))$ .
3. This line intersects the  $x$ -axis at  $x_2$ .
4. Repeat with  $x_1$  and  $x_2$  instead of  $x_0$  and  $x_1$ .

From this we can derive how  $x_2$  is calculated from  $x_0$  and  $x_1$ . The red line through  $(x_0, f(x_0))$  and  $(x_1, f(x_1))$  has equation

$$y - f(x_1) = \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_1).$$

We set  $y = 0$  to find  $x_2$ :

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

which yields

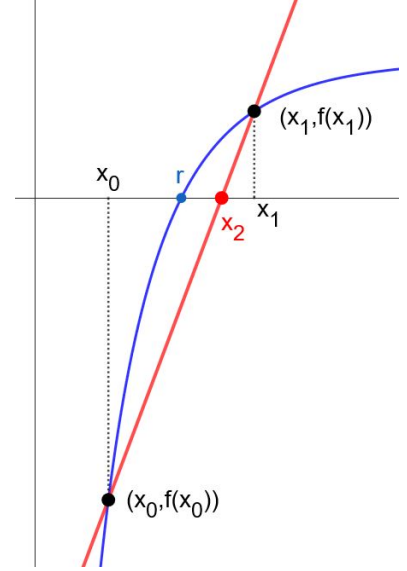
$$x_2 = x_1 - f(x_1) \frac{x_1 - x_0}{f(x_1) - f(x_0)}.$$

It is easy to see how that is iterated.

**Definition 3.** Let  $f : \mathbb{R} \rightarrow \mathbb{R}$  and  $x_0, x_1$  be given numbers. The secant method is defined inductively through

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

for all  $n \geq 1$ .



**Figure 15:** One step of the secant method. In blue is the graph of  $f(x)$  with root  $r$ . Initial guesses are  $x_0$  and  $x_1$ . The next guess is where the red secant line intersects the  $x$ -axis.

The slope is

$$s = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

and the point  $(x_1, f(x_1))$  lies on the line.

Definition 3 shows a clear resemblance to Newton's method

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

By definition of the derivative we have

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

since hopefully  $x_n$  is closed to  $x_{n-1}$ . In graphical terms, the secant line is an approximate tangent. See chapter ??? for more details.

An obvious choice for the starting values  $x_0$  and  $x_1$  is the endpoints of an interval containing the root, as in the bisection method. The implementation is rather straightforward:

```

1 function r_approx=secant(x0,x1,tol)
2     x2=x1+2*tol;
3     while abs(x2-x1) > tol
4         x1=x2;
5         f1=f(x1); f0=f(x0);
6         x2=x1-f1*(x1-x0)/(f1-f0);
7         x0=x1;
8     end
9     r_approx=x2;
10 end

```

As usual we assume  $f$  is defined outside of this program. Line 5 makes sure we do not compute  $f(x_1)$  twice for efficiency's sake.

As in the case of Newton's method we can prove that the secant method is locally convergent. Moreover the secant method is slower than Newton's method but still significantly faster than the bisection method as the following table suggests.

$n$	Bisection error	Newton error	Secant error
1	0.5000000000000000	1.090909090909091	0.6666666666666667
2	0.2500000000000000	0.500983209801845	0.451612903225807
3	0.1250000000000000	0.158390629858317	0.372364579954590
4	0.0625000000000000	0.021456703854125	0.160223618672317
5	0.0312500000000000	0.000450619773757	0.052187670996048
6	0.0156250000000000	0.000000202966699	0.009272881803156
7	0.0078125000000000	0.000000000000041	0.000494179187320
8	0.0039062500000000	0.000000000000000	0.000004562341550

**Table 4:** First eight iterations of all three methods for  $f(x) = x^3 - x^2 + x - 1$  with initial guesses  $a = 0$ ,  $b = 3$  for the bisection method,  $x_0 = 3$  for Newton's method and  $x_0 = 0$ ,  $x_1 = 3$  for the secant method.

Newton's method reaches machine precision after 8 iterations, the secant method after 13, and the bisection method after 51.

However note that each iteration of Newton's method calls the derivative  $f'$  which is usually more costly than calling the simpler  $f$ .

The bisection method shows its usual pattern of dividing the error by 2 at each step. Newton's method has a quadratic rate of convergence as discussed earlier. The secant method seems to need a few iterations to find its footing but then converges quickly to the root, although not as quickly as Newton's method. This is an evidence of **superlinear rate of convergence** which we explore now further.

**Theorem 6.** Assume the secant method  $x_n$  converges to a root  $r$  of a function  $f$ . Let  $e_n = x_n - r$  be the error at stage  $n$ .

Then if  $p = \frac{1 + \sqrt{5}}{2}$  we have

$$\left| \frac{e_{n+1}}{e_n^p} \right| \leq C$$

where  $C$  is a constant independent of  $n$ .

*Proof.* The secant method is defined as

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

Subtracting  $r$  from both sides gives:

$$x_{n+1} - r = (x_n - r) - f(x_n) \frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}.$$

Since

$$e_n - e_{n-1} = (x_n - r) - (x_{n-1} - r) = x_n - x_{n-1}$$

we rewrite this as

$$e_{n+1} = e_n - f(r + e_n) \frac{e_n - e_{n-1}}{f(r + e_n) - f(r + e_{n-1})}.$$

We approximate both the denominator and  $f(r + e_n)$  by Taylor expansion:

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

Dividing both sides of the fraction by  $f'(r)$  and letting  $M = \frac{f''(r)}{2f'(r)}$  leaves us with

$$e_{n+1} \approx e_n - \frac{e_n(1 + Me_n)(e_n - e_{n-1})}{e_n(1 + Me_n) - e_{n-1}(1 + Me_{n-1})}$$

which gives

$$e_{n+1} \approx e_n - \frac{e_n(1 + Me_n)}{1 + M(e_n + e_{n-1})}$$

and finally

$$e_{n+1} \approx \frac{e_{n-1}e_n M}{1 + M(e_n + e_{n-1})}.$$

The bisection method satisfies a similar theorem with  $p = 1$  and Newton's method with  $p = 2$ , see Theorems 2 and 4. Note that  $\frac{1+\sqrt{5}}{2} \simeq 1.61$  lies in between.

Repeated use of  $x_n = r + e_n$  by definition of  $e_n$ .

Recall  $f(x) \simeq f(r) + (x - r)f'(r) + \frac{1}{2}(x - r)^2 f''(r)$ . Moreover  $f(r) = 0$ . Apply for  $x = r + e_n$  to obtain:

$$f(r + e_n) \approx e_n f'(r) + \frac{e_n^2}{2} f''(r)$$

and likewise for  $f(r + e_{n-1})$ :

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

To simplify the proof, we omit higher order terms.

Cancel out  $e_n - e_{n-1}$ .

Combine the two fractions, there is a bit of algebra behind the scenes.

We make the final approximation that  $e_n + e_{n-1} \approx 0$  since the errors tend to 0 and obtain

$$e_{n+1} \approx Me_n e_{n-1}.$$

It remains to prove that

$$\left| \frac{e_{n+1}}{e_n^p} \right| \leq C$$

where  $p = \frac{1+\sqrt{5}}{2}$  and  $C$  is a constant. To do so define the sequence

$$u_n = \frac{e_{n+1}}{e_n^p} \Rightarrow e_{n+1} = u_n e_n^p.$$

Likewise we have

$$e_n = u_{n-1} e_{n-1}^p.$$

Combine both expressions to obtain:

$$e_{n+1} = u_n u_{n-1}^p (e_{n-1})^{p^2}.$$

The first part of the proof showed that

$$e_{n+1} \approx Me_n e_{n-1}$$

so that:

$$Me_n e_{n-1} \approx u_n (u_{n-1})^p (e_{n-1})^{p^2} \Rightarrow Me_n \approx u_n (u_{n-1})^p (e_{n-1})^{p^2-1}.$$

Now  $p = \frac{1+\sqrt{5}}{2}$  is a root of the quadratic equation  $x^2 - x - 1 = 0$ . In particular  $p^2 - 1 = p$  which gives us:

$$Me_n \approx u_n (u_{n-1})^p (e_{n-1})^p.$$

Since  $e_n = u_{n-1} (e_{n-1})^p$  by definition we are left with

$$M \approx u_n (u_{n-1})^{p-1}.$$

If such an equation holds we can not have  $u_n \rightarrow \infty$  hence the objective is attained and the theorem is proved.  $\square$

Very similar to the asymptotic equality

$$e_{n+1} \approx Me_n^2$$

obtained in Newton's method.

**Objective:** show that  $u_n$  is bounded, that is, does not tend to  $\infty$ .

Set  $e_n = u_{n-1} e_{n-1}^p$  in the above equation.

Second step is grouping the  $e_{n-1}$  terms together.

Its roots are  $x = \frac{1 \pm \sqrt{5}}{2}$  by elementary calculations. Another surprise appearance for the golden ratio!

Simplify  $e_n$  and  $u_{n-1} (e_{n-1})^p$  on both sides of the equation.

Because  $M$  is a finite constant.



# Systems

In the previous chapter we learned how to solve one equation in one variable. In complex situations, we need to consider many unknowns and (usually as many) equations, giving rise to an equation system. This is typical of physical systems with a number of variables which affect one another.

## Linear systems

Linear systems are the subject of linear algebra and are a rather simple problem to solve. They will prove however important in solving general (non-linear) systems and discretizing differential equations. It is therefore necessary to understand them properly.

We consider linear systems of  $n$  equations in the  $n$  variables  $x_1, x_2, \dots, x_n$ . As an example

$$\begin{cases} 2x_1 - 3x_2 + 5x_3 &= 4 \\ 5x_1 + 3x_2 - x_3 &= 0 \\ x_1 - 7x_2 + 3x_3 &= 0 \end{cases}$$

is a  $3 \times 3$  linear system. Linear systems are better represented as a matrix equation

$$Ax = \mathbf{b}$$

where  $A$  is the coefficient matrix of the system,  $\mathbf{x} = (x_1 \ x_2 \ \dots \ x_n)^T$  and  $\mathbf{b}$  is the right-hand side of the system. In general  $n \times n$  systems have a unique solution. This is an important result in linear algebra, called the inverse matrix theorem.

**Theorem 7.** Let  $A$  be a  $n \times n$  *invertible* matrix and  $\mathbf{b} \in \mathbb{R}^n$ . Then the linear system  $Ax = \mathbf{b}$  has a unique solution given by

$$\mathbf{x} = A^{-1}\mathbf{b}$$

where  $A^{-1}$  is the inverse matrix of  $A$ .

We will consider systems with more equations than variables in a later chapter.

We use the usual convention that letters in boldface such as  $\mathbf{x}$  represent vectors.

In the above example

$$A = \begin{pmatrix} 2 & -3 & 5 \\ 5 & 3 & -1 \\ 1 & -7 & 3 \end{pmatrix}$$

$$\mathbf{b} = \begin{pmatrix} 4 \\ 0 \\ 0 \end{pmatrix}.$$

Equivalent condition for invertibility:  $\det(A) \neq 0$ . Invertible matrices are also called non-singular.

If  $A$  is not invertible we either have no solution or infinitely many solutions, depending on  $b$ . However a randomly chosen matrix is overwhelmingly likely to be invertible, so this is usually not an issue.

It turns out that computing  $A^{-1}$  is not the most efficient way to solve a linear system. It is approximately twice as quick to use the classical method of Gaussian elimination. Matlab solves linear systems using a variant of Gaussian elimination called the pivot method with the following command:

```
1 x=A\b;
```

In the above example we define

```
1 A=[2 -3 5 ; 5 3 -1 ; 1 -7 3];
2 b=[4 ; 0 ; 0];
3 x=A\b
```

Important here that  $b$  is a column vector.

to obtain



The screenshot shows the MATLAB Command Window with the following output:

```
Command Window
x =
    -0.0580
     0.4638
    1.1014
```

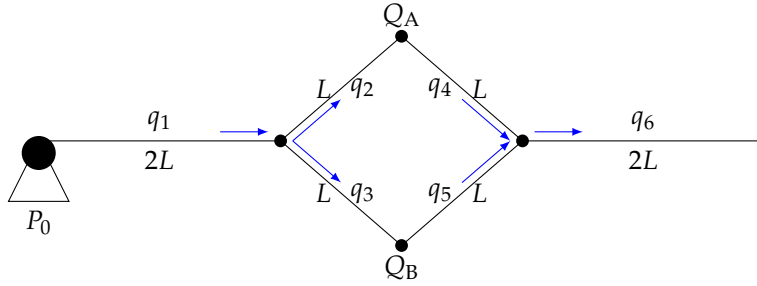
This method differs greatly from other numerical methods in the sense that Matlab computes an exact solution, up to machine precision. No approximation is involved.

### *More involved example - Pipe flow*

In this example, we see how to turn a system of equations derived from physical principles into a matrix equation that Matlab understands and can solve.

The picture below shows a network of pipes through which water at  $20^\circ\text{C}$  is flowing. Water is pumped in the system at outlet pressure  $P_0$  (that is  $P_0$  over external air pressure) and the network is open on the other end. The length of the pipes is indicated on the figure where  $L = 1\text{ m}$  and their diameter is  $d = 0.2\text{ m}$ . At the two points  $A$  and  $B$ , water leaves the network with flow rate  $Q_A = 2 \times 10^{-5}\text{ m}^3\text{ s}^{-1}$  and  $Q_B = 4 \times 10^{-5}\text{ m}^3\text{ s}^{-1}$  respectively.





**Figure 16:** Schematics of the pipe network. The blue arrows show the direction of the water flow.

We would like to compute the flow rates through each of the six pipes. These are labeled  $q_1$  through  $q_6$  and are expressed in  $\text{m}^3/\text{s}$ . We therefore need six equations. Four of them are readily obtained by examining each junction in the network: the rate at which water enters must equal the rate at which it leaves:

$$\begin{aligned} q_1 &= q_2 + q_3 \\ q_2 &= q_4 + Q_A \\ q_3 &= q_5 + Q_B \\ q_4 + q_5 &= q_6 \end{aligned}$$

For example the second equation is obtained by examining the upper point  $A$ . Water enters with flow rate  $q_2$  and leaves either along the pipe with flow rate  $q_4$  or out of the system with flow rate  $Q_A$ .

We now make the assumption that the water pressure is low (so-called laminar flow). In this case pressure  $P$  is linearly related to flow rate  $q$  via the Hagen-Poiseuille equation

$$\Delta P = \frac{128\mu \cdot (\text{length of pipe})}{\pi d^4} q$$

$d = 0.2 \text{ m}$  is the diameter of the pipe while the length is either  $L$  or  $2L$ .

where  $\mu = 1.002 \times 10^{-3} \text{ Pa}\cdot\text{s}$  is the dynamic viscosity of  $20^\circ \text{C}$  water. Along the loop part of the network the net difference in pressure must be 0 and therefore

$$\Delta P = 0 = \frac{128\mu L}{\pi d^4} (q_2 + q_4 - q_5 - q_3)$$

Signs are according to water flow direction shown by the blue arrows. Note that all four pipes have the same length  $L$ .

We clearly can divide by the constant and obtain the simpler equation

$$q_2 - q_3 + q_4 - q_5 = 0$$

Finally, starting on the left, going through  $A$  and exiting in the right yields the final equation:

$$\Delta P = P_0 = \frac{128\mu L}{\pi d^4} (2q_1 + q_2 + q_4 + 2q_6)$$

The right-hand side is at external air pressure and water is pumped with  $P_0$  overpressure. Note that pipes 1 and 6 are twice as long as the other ones which explains the extra factors 2.

We obtain a total of six linear equations for the six unknowns  $q_1, \dots, q_6$ .

In order to solve the system we need to rewrite in matrix form  $Ax = b$ . To do so we rewrite the equations as follows:

here  $x = (q_1 \ q_2 \ \dots \ q_6)^T$ .

$$\left\{ \begin{array}{rcl} q_1 - q_2 - q_3 & = & 0 \\ q_2 - q_4 & = & Q_A \\ q_3 - q_5 & = & Q_B \\ q_4 + q_5 - q_6 & = & 0 \\ q_2 - q_3 + q_4 - q_5 & = & 0 \\ 2q_1 + q_2 + q_4 + 2q_6 & = & \frac{\pi d^4}{128\mu L} P_0 \end{array} \right.$$

All terms including  $q_1, \dots, q_6$  on one side. The other terms form the vector  $b$ .

This suggests defining

$$A = \begin{pmatrix} 1 & -1 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 1 & -1 \\ 0 & 1 & -1 & 1 & -1 & 0 \\ 2 & 1 & 0 & 1 & 0 & 2 \end{pmatrix}, b = \begin{pmatrix} 0 \\ Q_A \\ Q_B \\ 0 \\ 0 \\ \frac{\pi d^4}{128\mu L} P_0 \end{pmatrix}$$

It is very useful to write the equations for  $q_1, \dots, q_6$  in ascending order of index to see the coefficient matrix immediately.

Solving the system with Matlab is now a simple matter of defining these two objects and all necessary parameters:

```

1  L=1;d=0.2;
2  QA=2*10^(-5);QB=4*10^(-5);
3  P0=0.01;
4  mu=1.002*10^(-3);
5
6  A=[1 -1 -1 0 0 0;
7     0 1 0 -1 0 0;
8     0 0 1 0 -1 0;
9     0 0 0 1 1 -1;
10    2 1 0 1 0 2;
11    0 1 -1 0 -1 0]
12
13 b=[0 QA QB 0 (pi*d^4)/(128*mu*L)*P0 0]';
14
15 q=A\b

```

the outcome of which is a vector  $q$  in  $\mathbb{R}^6$  containing all flow rates  $q_1, \dots, q_6$ . We will revisit this example in the case of higher water flow in which case the system becomes non-linear.

### Number of operations

Although Gaussian elimination theoretically always succeeds, there are limits to performance as the size  $n$  of the system grows. Knowing

these limits is especially important in discretization processes where  $n$  reflects how finely we discretize space. Since Gaussian elimination is rather straightforward we can actually compute how many basic operations are necessary to solve the system.

**Theorem 8.** *Solving an  $n \times n$  linear system by Gaussian elimination takes approximately  $\frac{2}{3}n^3$  basic operations (additions and multiplications).*

*Proof.* We want to reduce the matrix to a form as shown in the margin. To eliminate an entry in the first column, we need to perform two operations on every other entry ( $n - 1$  in total). Reducing the whole first column therefore requires

$$\text{Nb of entries} \times \text{Operations needed} = (n - 1)(2(n - 1)) = 2(n - 1)^2$$

operations. In a similar fashion the second column requires

$$(n - 2)(2(n - 2)) = 2(n - 2)^2$$

The grand total is therefore:

$$2(n - 1)^2 + 2(n - 2)^2 + \cdots + 2 = 2 \sum_{i=1}^{n-1} i^2$$

Using induction one can prove that this is equal to

$$2 \frac{(n - 1)n(2n - 1)}{6} \approx 2 \frac{n \cdot n \cdot 2n}{6} = \frac{2n^3}{3}$$

ignoring lower order terms. □

The main takeaway of Theorem 8 is that the number of operations needed scales quickly with  $n$ . Modern computers can perform around  $10^{12}$  basic operations a second. Solving a  $10000 \times 10000$  system ( $n = 10^4$ ) requires about

$$\frac{2}{3}n^3 \approx 6.7 \times 10^{11}$$

operations according to Theorem 8 so we would expect Matlab to need at around one second to solve it. This is still manageable but if  $n = 10^5$  we would need twenty minutes.

Another issue is that a  $10^5 \times 10^5$  matrix can not be easily stored since it has  $10^{10}$  elements, taking many gigabytes of hard disk space. Fortunately there are other ways to store large matrices and solve large linear systems provided the matrix has specific properties.

A more precise count would be  $\frac{2}{3}n^3 + \frac{3}{2}n^2 - \frac{7}{6}n$  but for large  $n$  the leading term is the important one.

Final form of a reduced matrix

$$\begin{pmatrix} X & X & X & \cdots & X & X \\ 0 & X & X & \cdots & X & X \\ 0 & 0 & X & \cdots & X & X \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & X \end{pmatrix}$$

A typical row operation would be to replace the second row  $R_2$  by  $R_2 - 2R_1$ , that is, one addition and one multiplication along the rest of the row.

The proof is not quite perfect as it ignores the effect on row operations on the vector  $b$ , as well as the final step of solving the system in diagonal form. Both steps are of order  $n^2$  so that the result holds in any case.

as of 2023...

one teraflop in computer jargon

By Theorem 8 a tenfold increase in  $n$  means a thousand-fold increase in operation count.

### Large linear systems - Sparse matrices

Many large systems have the properties that each equation contains only a small number of variables. This makes physical sense - it is somewhat likely that variables are only affected by the variables near them. This means that the matrix  $A$  contains mostly zeroes, a so-called **sparse matrix**.

**Definition 4.** A  $n \times n$  matrix is called sparse if only a small number of elements on each row and column are non-zero.

A common example of a sparse matrix are so-called tridiagonal matrices which occur naturally when solving boundary values problems. All omitted entries are 0.

$$T = \begin{pmatrix} 3 & 1 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 1 & 3 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & 1 & 3 & 1 & 0 & \cdots & 0 & 0 & 0 \\ \vdots & & \ddots & \ddots & \ddots & & & & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & & & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & & & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & 1 & 3 & 1 \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 3 \end{pmatrix}$$

Such a matrix  $T$  can be defined a sparse matrix in Matlab by only specifying the non-zero entries, saving a lot of hard disk space. The command

```
1 T=sparse(i,j,values);
```

defines a sparse matrix  $T$  from the vector triplet  $i, j, \text{values}$  such that the element on row  $i(k)$  and column  $j(k)$  is equal to  $\text{values}(k)$ . For example the code

```
1 n=10;
2 i=[2:n , 1:n , 1:n-1]';
3 j=[1:n-1 , 1:n , 2:n]';
4 values=[ones(n-1,1); 3*ones(n,1); ones(n-1,1)]';
5 T=sparse(i,j,values)
```

defines the  $10 \times 10$  version of  $T$  above. Printing  $T$  shows the location of non-zero elements, and their values:

There is no fixed definition of what "small" means here. For instance we could demand the the total number of non-zero elements is less than a small multiple of  $n$  such as  $3n$ .

The entries do not need to be three constant numbers although it is the case when solving time-independent differential equations.

The vectors  $i$  and  $j$  viewed together contain the locations of non-zero elements. It starts with  $(2,1), (3,2), \dots, (n, n-1)$ , followed by  $(1,1), (2,2), \dots, (n,n)$  and finally  $(1,2), (2,3), \dots, (n-1,n)$ .

The vector  $\text{values}$  contains the corresponding entries in the matrix.

The command `ones(n,1)` defines the

vector  $\begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$  in  $\mathbb{R}^n$ .

This method uses the strengths of Matlab in dealing with vectors. It is not recommended to use large loops to construct  $T$ .

Command Window		
T =		
(1, 1)	3	
(2, 1)	1	
(1, 2)	1	
(2, 2)	3	
(3, 2)	1	
(2, 3)	1	
(3, 3)	3	
<i>fx</i>	....	

This only shows the three left-most columns.

The command

```
1 A=full(T);
```

converts  $T$  to a full matrix if needed. Linear systems involving sparse matrices are solved using the normal command for instance:

```
1 b=[1; zeros(n-2,1); 1];
2 T\b;
```

which runs in 0.01 second although the size of the matrix is  $n = 10^5$ . Matlab takes advantage of the fact that most elements of  $T$  are zero and only perform Gaussian elimination where it is needed.

Note that this may fail if the size of  $T$  is too large.

$T$  is the tridiagonal matrix defined earlier for  $n = 10^5$ .

Compare it with the twenty minutes needed to solve the system with the direct method. The number of operations is now of order  $n$  instead of  $n^3$ .

### Iterative methods - Jacobi method

Another strategy to deal with large systems is to use an iterating scheme to approximate the solution, instead of solving it exactly by Gaussian elimination. This again relies on the matrix having a specific property.

**Definition 5.** A matrix  $A$  is called **diagonally dominant** if the diagonal elements are larger than the sum of all other elements on the same row (in absolute value). That is for every  $1 \leq i \leq n$  we have

$$|a_{i,i}| > \sum_{j \neq i} |a_{i,j}|.$$

When solving the linear system  $Ax = b$  we rewrite

$$A = D + R$$

The matrix  $A = \begin{pmatrix} 4 & 1 & 0 \\ -2 & 5 & 1 \\ 1 & 4 & 6 \end{pmatrix}$  is diagonally dominant since

$$4 > 1 + 0$$

$$5 > 2 + 1$$

$$6 > 1 + 4.$$

where  $D$  is the diagonal part of  $A$  and  $R$  is the rest. The system becomes

$$Dx + Rx = b \Rightarrow x = D^{-1}(b - Rx).$$

Note that since  $D$  is diagonal,  $D^{-1}$  is simply found by taking the inverse of the diagonal elements. This suggests using the iteration

$$x_{n+1} = D^{-1}(b - Rx_n)$$

with given initial value  $x_0$ .

**Definition 6.** Assume  $A$  is a diagonally dominant  $n \times n$  matrix, and let  $x_0 \in \mathbb{R}^n$ . Write  $A = D + R$  where  $D$  is the diagonal of  $A$ . The Jacobi method is defined through

$$x_{n+1} = D^{-1}(b - Rx_n)$$

It turns out that the Jacobi method always converges as long as the matrix is diagonally dominant. The choice of  $x_0$  has no effect, unlike for Newton's method.

**Theorem 9.** Assume  $A$  is an invertible diagonally dominant  $n \times n$  matrix. The Jacobi method converges to the solution of  $Ax = b$  however  $x_0$  is chosen.

To prove the theorem we will need to define a **matrix norm**, a measure of the size of a matrix.

**Definition 7.** Let  $B$  be a  $n \times n$  matrix. For each row of  $B$  one can compute the sum of absolute values of the elements of the row. The matrix norm of  $B$  is the largest such sum. In other words

$$\|B\| = \max_{1 \leq i \leq n} \sum_{j=1}^n |a_{i,j}|$$

The matrix norm is for a matrix what length is for a vector and absolute value is for a number, a measure of how big or small the matrix is.

As an example, if  $\|B\| < 1$  and  $x$  is a fixed vector then

$$B^n x \rightarrow 0$$

as  $n \rightarrow \infty$ . Similarly if  $\|B\| < 1$  and  $x$  is a fixed vector, the geometric series

$$x + Bx + B^2x + \dots$$

converges. With this in hand we can prove Theorem 9.

For  $A$  as above,  $D = \begin{pmatrix} 4 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 6 \end{pmatrix}$  and

$$R = \begin{pmatrix} 0 & 1 & 0 \\ -2 & 0 & 1 \\ 1 & 4 & 0 \end{pmatrix}.$$

In our example

$$D^{-1} = \begin{pmatrix} 1/4 & 0 & 0 \\ 0 & 1/5 & 0 \\ 0 & 0 & 1/6 \end{pmatrix}.$$

The Jacobi method only performs vector additions and multiplication of a  $n$ -vector with a  $n \times n$  matrix, both of which needing  $O(n^2)$  basic operations rather than  $O(n^3)$ . Therefore we expect the Jacobi method to better scale for large values of  $n$ .

If  $B = \begin{pmatrix} 4 & -3 & 2 \\ -2 & 1 & 1 \\ 1 & 1 & 9 \end{pmatrix}$ , the first row gives  $4 + 3 + 2 = 9$ , the second  $2 + 1 + 1 = 4$  and the third  $1 + 1 + 9 = 11$ . Therefore  $\|B\| = 11$ , the max of the three sums.

Analogous to  $r^n \rightarrow 0$  if  $|r| < 1$ .

Analogous to the regular geometric series  $1 + k + k^2 + \dots$  which converges to  $\frac{1}{1-k}$  if  $|k| < 1$ .

*Proof.* For simplicity we rewrite the Jacobi method as

$$\mathbf{x}_{n+1} = D^{-1}\mathbf{b} - D^{-1}R\mathbf{x}_n = \mathbf{y} + B\mathbf{x}_n$$

where  $\mathbf{y} = D^{-1}\mathbf{b}$  is a fixed vector and  $B = -D^{-1}R$  is a fixed matrix.

Because  $A$  is diagonally dominant, we can show that  $B = -D^{-1}R$  is such that on every row, the sum of the absolute values is always less than 1 (see an example in the margin). In other words  $\|B\| < 1$ .

Now the Jacobi method starts with  $\mathbf{x}_0$  followed by

$$\mathbf{x}_1 = \mathbf{y} + B\mathbf{x}_0$$

then

$$\mathbf{x}_2 = \mathbf{y} + B\mathbf{x}_1 = \mathbf{y} + B(\mathbf{y} + B\mathbf{x}_0) = \mathbf{y} + B\mathbf{y} + B^2\mathbf{x}_0$$

and similarly if  $n \in \mathbb{N}$ :

$$\mathbf{x}_n = (\mathbf{y} + B\mathbf{y} + \cdots + B^{n-1}\mathbf{y}) + B^n\mathbf{x}_0.$$

Because  $\|B\| < 1$  we see that the geometric series in the brackets converges while  $B^n\mathbf{x}_0$  tends to 0. This shows that the Jacobi method converges to some vector  $\mathbf{x}$ .

It remains to show that this vector  $\mathbf{x}$  is precisely the solution of  $A\mathbf{x} = \mathbf{b}$ . The original Jacobi method is

$$\mathbf{x}_{n+1} = D^{-1}(\mathbf{b} - R\mathbf{x}_n)$$

Letting  $n \rightarrow \infty$  gives

$$\mathbf{x} = D^{-1}(\mathbf{b} - R\mathbf{x})$$

which rearranged gives us

$$D\mathbf{x} = \mathbf{b} - R\mathbf{x} \Rightarrow (D + R)\mathbf{x} = \mathbf{b}$$

which is  $A\mathbf{x} = \mathbf{b}$  since  $A = D + R$ . □

### Jacobi method - Implementation

To solve the linear system  $A\mathbf{x} = \mathbf{b}$  we set up a program taking in four variables: the matrix  $A$ , the vector  $\mathbf{b}$ , an initial guess  $\mathbf{x}_0$  and a tolerance. The idea is to first compute  $D$  and  $R$ , and then iterate Jacobi method. We stop the loop whenever the difference between  $\mathbf{x}_n$  and  $\mathbf{x}_{n+1}$  is less than the tolerance, exactly like for Newton's method. There is however a small technical issue. Since we are working with vectors and not numbers we cannot simply use the absolute value and use instead vector norm.

Example:  $R = \begin{pmatrix} 0 & 1 & 0 \\ -2 & 0 & 1 \\ 1 & 4 & 0 \end{pmatrix},$

$$D^{-1} = \begin{pmatrix} 1/4 & 0 & 0 \\ 0 & 1/5 & 0 \\ 0 & 0 & 1/6 \end{pmatrix},$$

$$\text{so } -D^{-1}R = -\begin{pmatrix} 0 & 1/4 & 0 \\ -2/5 & 0 & 1/5 \\ 1/6 & 4/6 & 0 \end{pmatrix}.$$

Since  $A$  is diag. dominant we know for instance that the third diagonal element 6 is more than the sum of all other elements on the third row ( $1 + 4$ ). This is the same as saying  $1/6 + 4/6 < 1$ . This analysis can be done on any row.

Next in line would be

$$\mathbf{x}_3 = \mathbf{y} + B\mathbf{x}_2 =$$

$$\mathbf{y} + B(\mathbf{y} + B\mathbf{y} + B^2\mathbf{x}_0)$$

which is

$$\mathbf{y} + B\mathbf{y} + B^2\mathbf{y} + B^3\mathbf{x}_0$$

Because  $\mathbf{x}_n \rightarrow \mathbf{x}$  we also have  $\mathbf{x}_{n+1} \rightarrow \mathbf{x}$ .

Notice how  $\mathbf{x}_0$  could have been anything throughout the course of the proof. No need to be near enough the solution like with Newton's method.

The norm of a vector in  $\mathbb{R}^n$  is simply its length  $\|\mathbf{x}\| = \sqrt{x_1^2 + \cdots + x_n^2}$ , computed through `norm(x)` in Matlab. More on vector norms later.

```

1 function x=jacobi(A,b,x0,tol)
2     D=diag(A);
3     R=A-diag(D);
4     x=x0; oldx=x+2*tol;
5     while norm(x-oldx)>tol
6         oldx=x;
7         x=(b-R*oldx)./D;
8     end
9 end

```

$\text{diag}(A)$  is a vector containing the diagonal elements, and  $\text{diag}(D)$  makes a matrix out of it that we can subtract from  $A$ . The element-wise division on line 7 is much quicker than calculating the inverse of  $D$  - recall that  $D$  is a diagonal matrix so its inverse is obvious.

The program assume  $A$  to be diagonally dominant - we might want to perform a check beforehand.

It is interesting to compare the performance of the Jacobi method and accurate system solving through Gaussian elimination. The difference is very noticeable on large systems. We define a random  $10000 \times 10000$  matrix  $A$  and a random vector  $b \in \mathbb{R}^{10000}$  through the command:

```

1 n=10000;
2 A=rand(n,n); b=rand(n,1);

```

The elements of  $A$  and  $b$  are uniformly distributed on the interval  $[0,1]$ .

and make it diagonally dominant by increasing sufficiently the diagonal elements. We time how long Matlab takes to solve the system in an exact fashion:

```

1 tic
2 A\b;
3 toc

```

This is so that we may use the Jacobi method.

which takes a bit more than 3 seconds. The Jacobi method with low accuracy takes 0.5 seconds:

```

1 tic
2 x0=zeros(n,1);
3 x=jacobi(A,b,x0,10^(-3));
4 toc

```

By Theorem 9 we may use any initial vector  $x_0$ , the zero vector is the simplest choice.

and with higher tolerance  $10^{-10}$  the running time hardly increases. If we increase the size of the matrix to 20000, Gaussian elimination takes about 22 seconds (eight times longer than  $n = 10000$ ) while the Jacobi method needs 2 seconds (four times longer than before). This is consistent with Gaussian elimination being a  $O(n^3)$  method while the Jacobi method is a  $O(n^2)$  method. We see that the Jacobi method is clearly outperforming its competitor for large  $n$ , but recall that it only works on diagonally dominant matrices.

If we double  $n$ , then  $n^3$  is eight times larger because  $2^3 = 8$ . Likewise for  $n^2$ .

There exists other types of iterative methods working on (other) specific kinds of matrices. The conjugate gradient method is such a method for positive symmetric matrices.



## Non-linear systems

We now consider general non-linear systems. We still assume that there are as many variables as equations. This is an example of such a system for three variables  $x_1$ ,  $x_2$  and  $x_3$ .

$$\begin{cases} x_1^2 + 3x_1x_2 + x_3^3 = 0 \\ 1 - x_1^3x_2 + \ln(x_2^2 + 1) = 0 \\ 3x_1^3 - 3x_2^3 + \cos(x_3) = 0 \end{cases}$$

In general there is no way to find an exact solution to such a system. Even worse, there are no general theorems about the existence or the total number of solutions. It is also difficult (or impossible) to obtain rough estimates graphically, the way we graphed roots in the first chapter.

We therefore need to rely on iterative methods to find an approximate solution. The method of choice is the multivariate Newton's method, which extends its one-dimensional counterpart to systems. Newton's method was defined to solve equations of the form  $f(x) = 0$ , see Definition 2. We therefore need to rewrite systems as one single equation of similar type. In the above example we let

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, \quad \mathbf{F}(\mathbf{x}) = \begin{pmatrix} x_1^2 + 3x_1x_2 + x_3^3 \\ 1 - x_1^3x_2 + \ln(x_2^2 + 1) \\ 3x_1^3 - 3x_2^3 + \cos(x_3) \end{pmatrix}$$

turning our system into a single vector equation

$$\mathbf{F}(\mathbf{x}) = \mathbf{0}.$$

In general,  $\mathbf{F}$  is defined on an subset of  $\mathbb{R}^n$  with values in  $\mathbb{R}^n$ .

## Multivariate Newton's method

Recall that in one dimension Newton's method is defined inductively by

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

Now if our function  $\mathbf{F}$  is a function of  $n$  variables with values in  $\mathbb{R}^n$ , its derivative is not a number but can be represented by the Jacobi matrix  $J$ . If we write the coordinates of  $\mathbf{F}$  as  $f_1, \dots, f_n$  then:

$$J = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1} & \frac{\partial f_n}{\partial x_2} & \cdots & \frac{\partial f_n}{\partial x_n} \end{pmatrix}$$

Compare to Theorem 7. There is no overarching theory for non-linear systems.

sometimes also called Newton-Raphson method

The components of the vector-valued function  $\mathbf{F}$  are usually denoted by  $f_1(\mathbf{x}), f_2(\mathbf{x}) \dots$

As in the one-dimensional case, we may need to move terms around to obtain this precise form.

same  $n$  if there are as many variables as there are equations.

Note that  $J$  depends on  $\mathbf{x}$  since all partial derivatives are functions of  $x_1, \dots, x_n$ . It is however very cumbersome to specify this dependency everywhere, so we skip it when no misunderstanding is possible.

Note that  $J$  is a square  $n \times n$  matrix because we have as many variables as equations. We cannot divide by a matrix (the way we divide by  $f'(x_n)$  in one dimension) but we can multiply by the inverse Jacobi matrix  $J^{-1}$ . This leads to the following definition.

**Definition 8.** Let  $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$ , and let  $x_0 \in \mathbb{R}^n$ . The multivariate Newton's method is defined inductively by

$$x_{n+1} = x_n - J(x_n)^{-1}F(x_n)$$

where  $J(x_n)$  is the Jacobi matrix of  $F$ , evaluated at  $x_n$ .

Multivariate Newton's method is locally convergent: if  $x_0$  is near enough a solution to  $F(x) = 0$ , then  $x_n$  will converge to that root. As in one dimension the method has a quadratic rate of convergence. Note however that each iteration is much more expensive than in one dimension as we need to calculate with vectors and matrices instead of simple operations on numbers.

We mentioned earlier that it is expensive to compute an inverse matrix in comparison with solving a linear system. This motivates us to rewrite Newton's method in terms of linear systems. To do so let

$$s = J(x_n)^{-1}F(x_n).$$

Multiplying both sides of the equation by the Jacobi matrix gives

$$J(x_n)s = F(x_n)$$

which is a linear system of the form  $As = b$ . Therefore we may change Newton's method into a two step calculation:

- Solve the linear system  $J(x_n)s = F(x_n)$  for  $s$ .
- Let  $x_{n+1} = x_n - s$ .

This serves as the basis for a general implementation of the method.

```

1 function x=newtonmult(x0,tol)
2     x=x0; oldx=x+2*tol;
3     while norm(x-oldx)>tol
4         oldx=x;
5         s=J(x)\F(x);
6         x=x-s;
7     end
8 end

```

As usual this requires both  $F$  and the Jacobi matrix  $J$  to be defined as functions. We show how to do so in two examples.

The proof of both these facts is analogous to the one-dimensional case, see Theorems 3 and 4, but is rather technical.

where  $A = J(x_n)$  and  $b = F(x_n)$

Some authors move the minus sign into the first step, solving  $J(x_n)s = -F(x_n)$  and letting  $x_{n+1} = x_n + s$

Note the use of vector norm to stop the loop as  $x$  and  $oldx$  are vectors. Line 5 solves the system  $J(x_n)s = F(x_n)$ .

*Implementation - simple example.*

Let us solve the following system of three equations for the variables  $x_1$ ,  $x_2$  and  $x_3$ :

$$\begin{cases} x_1^3 - 2x_2 - 2 = 0 \\ x_1^3 - 5x_3^2 + 7 = 0 \\ x_2x_3^2 - 1 = 0 \end{cases}$$

so we define as before

$$F(x) = \begin{pmatrix} x_1^3 - 2x_2 - 2 \\ x_1^3 - 5x_3^2 + 7 \\ x_2x_3^2 - 1 \end{pmatrix}, \quad \text{with } x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

To run Newton's method we also need to compute the Jacobi matrix, we can do this by hand:

$$J(x) = \begin{pmatrix} 3x_1^2 & -2 & 0 \\ 3x_1^2 & 0 & -10x_3 \\ 0 & x_3^2 & 2x_2x_3 \end{pmatrix}$$

It is of utmost importance to hard code this matrix. Having Matlab compute it from scratch symbolically is very time consuming. Hence we define the two sub functions:

```
1 function y=F(x)
2 x1=x(1);x2=x(2);x3=x(3);
3 y=[ x1^3-2*x2-2;
4     x1^3-5*x3^2+7;
5     x2*x3^2-1];
6 end
```

and

```
1 function jac=J(x)
2 x1=x(1);x2=x(2);x3=x(3);
3 jac=[ 3*x1^2 , -2 , 0 ;
4       3*x1^2 , 0 , -10*x3 ;
5       0 , x3^2 , 2*x2*x3 ];
6 end
```

Contrary to the one-dimensional case, it is difficult to obtain a good estimate of a solution by graphical means, and we may need to try several different values of  $x_0$  before obtaining convergence. The system seems somehow to suggest small values for  $x_1$ ,  $x_2$  and  $x_3$ , so we try the following:

As before  $f_1$ ,  $f_2$  and  $f_3$  are the coordinates of  $F$ , that is, one equation of the system. General formula is

$$J(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \frac{\partial f_1}{\partial x_3} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \frac{\partial f_2}{\partial x_3} \\ \frac{\partial f_3}{\partial x_1} & \frac{\partial f_3}{\partial x_2} & \frac{\partial f_3}{\partial x_3} \end{pmatrix}$$

Line 2 is unnecessary but makes the code much more readable. Matlab is best suited to have  $F$  and  $J$  taking vectors as inputs rather than three numbers.

```

1      x0=[1;1;1];tol=10^(-5);
2      x=newtonmult(x0,tol)

```

yielding

Command Window

```

x =
    1.442249570307408
    0.500000000000000
fx  1.414213562373095

```

The system is simple enough to be solved analytically and has four solutions:

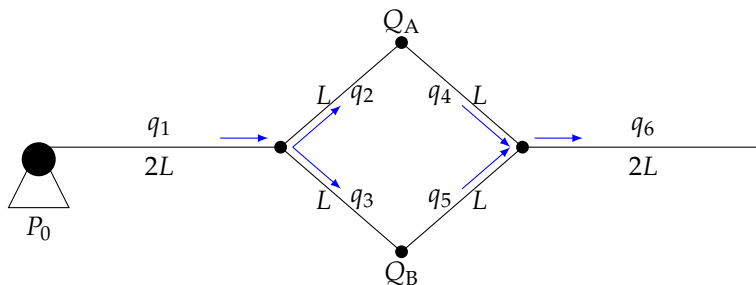
$$x = \begin{pmatrix} \pm \sqrt[3]{3} \\ 1/2 \\ \pm \sqrt{2} \end{pmatrix}$$

and we seem to converge to the one solution with two plus signs, which is indeed the closest to our first guess

$$x_0 = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

### Implementation - Pipe network revisited

We study again the pipe network



subtract row 1 from row 2 to obtain

$$-2x_2 + 5x_3^2 - 9 = 0$$

Row 3 gives

$$x_3^2 = \frac{1}{x_2}$$

so we obtain

$$-2x_2 + \frac{5}{x_2} - 9 = 0$$

which is eventually a quadratic equation for  $x_2$ . The negative solution  $-5$  is impossible due to  $x_3^2 = 1/x_2$  so we have  $x_2 = 1/2$ . We can then solve for one variable at a time using Row 3 and Row 1.

**Figure 17:** Schematics of the pipe network. The blue arrows show the direction of the water flow. Water exits the network at A and B with known flow  $Q_A = 2 \times 10^{-5} \text{ m}^3/\text{s}$  and  $Q_B = 4 \times 10^{-5} \text{ m}^3/\text{s}$ . Water enters the system from the left at overpressure  $P_0$ .

The first four equations

$$\begin{aligned}
 q_1 &= q_2 + q_3 \\
 q_2 &= q_4 + Q_A \\
 q_3 &= q_5 + Q_B \\
 q_4 + q_5 &= q_6
 \end{aligned}$$

still hold. If the water pressure is high enough, the relationship between pressure drop  $\Delta P$  and flow rate  $q$  becomes non-linear and is given by the Darcy-Weisbach equation:

$$\Delta P = \frac{8f\rho L}{\pi^2 d^5} q^2$$

where  $f = 0.02$  is the Darcy friction factor,  $\rho = 998 \text{ kg/m}^3$  is the density of water at  $20^\circ \text{C}$ , while  $L = 1 \text{ m}$  and  $d = 0.2 \text{ m}$  are the dimensions of the pipes (length and diameter). Going around the square loop yields the equation

$$\Delta P = 0 = \frac{8f\rho L}{\pi^2 d^5} (q_2^2 + q_4^2 - q_5^2 - q_3^2)$$

while going from left to right along the upper path yields

$$\Delta P = P_0 = \frac{8f\rho L}{\pi^2 d^5} (2q_1^2 + q_2^2 + q_4^2 + 2q_6^2)$$

These equations form a non-linear system for  $q_1, \dots, q_6$ . To write it in the form  $F(q) = 0$  we rewrite it as

$$\begin{cases} q_1 - q_2 - q_3 = 0 \\ q_2 - q_4 - Q_A = 0 \\ q_3 - q_5 - Q_B = 0 \\ q_4 + q_5 - q_6 = 0 \\ q_2^2 - q_3^2 + q_4^2 - q_5^2 = 0 \\ 2q_1^2 + q_2^2 + q_4^2 + 2q_6^2 - \frac{\pi^2 d^5}{8f\rho L} P_0 = 0 \end{cases}$$

so we naturally define

$$F(q) = \begin{pmatrix} q_1 - q_2 - q_3 \\ q_2 - q_4 - Q_A \\ q_3 - q_5 - Q_B \\ q_4 + q_5 - q_6 \\ q_2^2 - q_3^2 + q_4^2 - q_5^2 \\ 2q_1^2 + q_2^2 + q_4^2 + 2q_6^2 - \frac{\pi^2 d^5}{8f\rho L} P_0 \end{pmatrix}, \quad \text{where } q = \begin{pmatrix} q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \\ q_6 \end{pmatrix}$$

In order to execute Newton's method we only need the Jacobi matrix of  $F$ . It is easily calculated by hand, although we also may use symbolic differentiation in Matlab to compute it for us.

$$J = \begin{pmatrix} 1 & -1 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 1 & -1 \\ 0 & 2q_2 & -2q_3 & 2q_4 & -2q_5 & 0 \\ 4q_1 & 2q_2 & 0 & 2q_4 & 0 & 4q_6 \end{pmatrix}$$

Again the length of pipes 1 and 6 is  $2L$ , explaining the extra factors two.

although four of these equations are linear, it is still non-linear as a whole

We divide equations 5 and 6 by the large constant  $\frac{8f\rho L}{\pi^2 d^5}$ .

Remember to hard code the answer as symbolic differentiation is costly.

Recall  $Q_A, Q_B$  are constants, as is  $\frac{\pi^2 d^5}{8f\rho L} P_0$  so they vanish when differentiating.

We define these two objects in Matlab as functions. The water enters the system at overpressure  $P_0 = 0.5$  Pa.

```

1 function y=F(q)
2 QA=2*10^(-5);QB=4*10^(-5);P0=0.5;
3 L=1;d=0.2;
4 f=0.02;rho=998;
5 q1=q(1);q2=q(2);q3=q(3);q4=q(4);q5=q(5);q6=q(6);
6 y=[q1-q2-q3;
7     q2-q4-QA;
8     q3-q5-QB;
9     q4+q5-q6;
10    q2^2-q3^2+q4^2-q5^2;
11    2*q1^2+q2^2+q4^2+2*q6^2-(pi^2*d^5)/(8*f*rho*
12    L)*P0];
end

```

Again much simpler to define F as a function of a vector q and later defining the coordinates as on Line 5. The code gets very messy if everything is a function of six variables.

and similarly

```

1 function jac=J(q)
2 QA=2*10^(-5);QB=4*10^(-5);P0=0.5;
3 L=1;d=0.2;
4 f=0.02;rho=998;
5 q1=q(1);q2=q(2);q3=q(3);q4=q(4);q5=q(5);q6=q(6);
6 jac=[1 -1 -1 0 0 0;
7     0 1 0 -1 0 0;
8     0 0 1 0 -1 0;
9     0 0 0 1 1 -1;
10    0 2*q2 -2*q3 2*q4 -2*q5 0 ;
11    4*q1 2*q2 0 2*q4 0 4*q6];
12 end

```

In order to use Newton's method, we only need an initial guess for  $q_1, \dots, q_6$ . One simple idea is to use the same initial value  $q_0$  for all six variables. The equation

$$P_0 = \frac{8f\rho L}{\pi^2 d^5} (2q_1^2 + q_2^2 + q_4^2 + 2q_6^2)$$

then becomes

$$P_0 = \frac{8f\rho L}{\pi^2 d^5} (2q_0^2 + q_0^2 + q_0^2 + 2q_0^2) = \frac{54f\rho L}{\pi^2 d^5} q_0^2$$

which we solve for  $q_0$ :

$$q_0 = \sqrt{\frac{P_0 \pi^2 d^5}{54f\rho L}} \approx 0.001$$

Set  $q_1 = q_2 = q_4 = q_6 = q_0$ .

No need for accuracy here, one significant digit is enough.

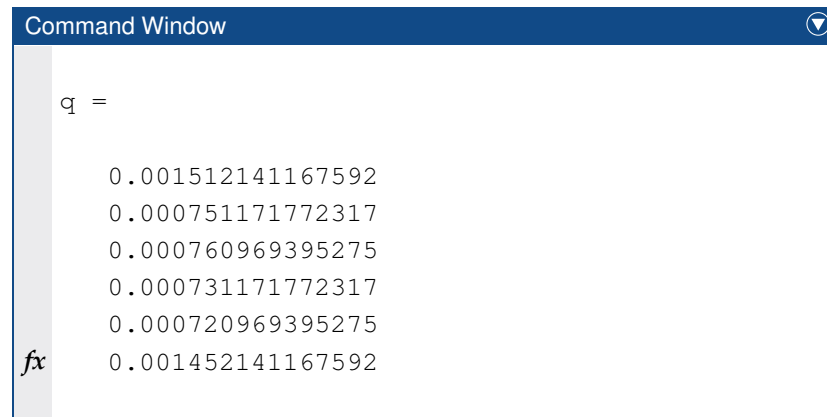
It therefore makes sense to use the initial guess

$$q_0 = (0.001 \ 0.001 \ 0.001 \ 0.001 \ 0.001 \ 0.001)^T$$

We are now ready to execute Newton's method:

```
1 x0=0.001*ones(6,1);
2 tol=10^(-7);
3 x=newtonmult(x0,tol)
```

which yields the approximate solution



Command Window

```
q =
    0.001512141167592
    0.000751171772317
    0.000760969395275
    0.000731171772317
    0.000720969395275
fx    0.001452141167592
```

The flow rates  $q_1, \dots, q_6$  are indeed somewhat near 0.001, although the water flow is divided into lower and upper path for  $q_2, \dots, q_5$ . That  $q_6$  is smaller than  $q_1$  can be explained by the fact that water exits the system at two points along the way. We can be convinced that this answer makes sense!

### Further experimentation using the bisection method

A natural follow-up would be to study the effects of a variable outlet pressure  $P_0$ . For more modularity, we change the functions for  $F$  and its Jacobi matrix so that the take  $P_0$  as an extra variable.

Easily coded using **ones** in Matlab.

Here we must use a rather low error tolerance. The values of  $q$  are  $\approx 10^{-3}$  so a tolerance of  $10^{-5}$  would only guarantee two significant digits.

Similarly we expect  $q_4 < q_2$  and  $q_5 < q_3$ .

No value for  $P_0$  is given, it is instead a variable.

```

1 function y=F(q,P0)
2 QA=2*10^(-5);QB=4*10^(-5);
3 L=1;d=0.2;
4 f=0.02;rho=998;
5 q1=q(1);q2=q(2);q3=q(3);q4=q(4);q5=q(5);q6=q(6);
6 y=[q1-q2-q3;
7    q2-q4-QA;
8    q3-q5-QB;
9    q4+q5-q6;
10   q2^2-q3^2+q4^2-q5^2;
11   2*q1^2+q2^2+q4^2+2*q6^2-(pi^2*d^5)/(8*f*rho*
12   L)*P0];

```

and similarly for the Jacobi matrix. We also need to slightly modify Newton's method to accommodate this change.

```

1 function x=newtonmult(x0,tol,P0)
2 x=x0;oldx=x+2*tol;
3 while norm(x-oldx)>tol
4     oldx=x;
5     s=J(x,P0)\F(x,P0);
6     x=x-s;
7 end
8 end

```

We can now find flow rates for a variety of values of  $P_0$ , say from 0.5 Pa to 5 Pa. We shall concentrate on  $q_6$ , the flow rate on the right-hand side of the network.

```

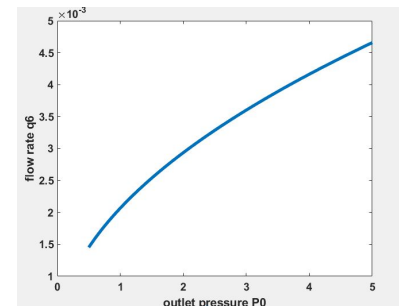
1 x0=0.001*ones(6,1);
2 tol=10^(-7);
3 counter=1;
4 for P0=0.5:0.1:5
5     q=newtonmult(x0,tol,P0);
6     q6(counter)=q(6);
7     counter=counter+1;
8 end
9 Pvector=0.5:0.1:5;
10 plot(Pvector,q6)

```

Now let us imagine that we would like the flow rate out of the pipe network  $q_6$  to be at most  $3 \times 10^{-3} \text{ m}^3/\text{s}$ . What value of outlet pres-

Two changes:  $P_0$  is a variable, and both J and F need  $P_0$  as input.

Same initial value as before. Line 4 specifies the values of  $P_0$  used, from 0.5 to 5 with an interval of 0.1. Line 5 solves the system via Newton's method for the current value of  $P_0$ . Line 6 extracts  $q_6$  from the solution vector.



**Figure 18:** Plot of  $q_6$  as a function of  $P_0$  on the interval  $[0.5, 5]$ .



sure  $P_0$  should be used? From the graph, it seems that a value of  $P_0 \approx 2$  Pa is the answer. How to find this value more precisely? In other words we wish to solve the equation

$$q_6(P_0) = 3 \times 10^{-3}.$$

The bisection method is a good choice. Indeed this is an equation of one variable  $P_0$  for which the bisection method applies. One-dimensional Newton's method seems difficult to use, as we don't really have a way to calculate the derivative of the  $q_6$  function. This function is only computable one data point at a time: for each given  $P_0$ , we use multivariate Newton's method to find  $q_6(P_0)$ .

To set up the bisection method we first rewrite the equation in the form  $f(x) = 0$ :

$$q_6(P_0) - 3 \times 10^{-3} = 0.$$

or  $f(P_0) = 0$ . The next step is to define  $f$  as a function of one variable in Matlab. For instance:

```

1 function y=f(P0)
2     QA=2*10^(-5);QB=4*10^(-5);
3     L=1;d=0.2;
4     f=0.02;rho=998;
5
6     x0=0.001*ones(6,1);
7     tol=10^(-7);
8     q=newtonmult(x0,tol,P0);
9
10    y=q(6)-3*10^(-3);
11 end

```

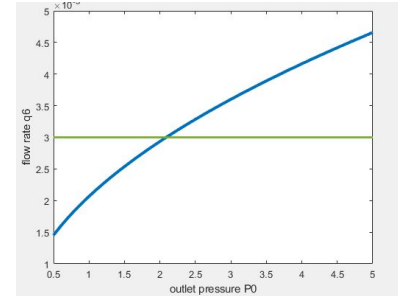
To use the bisection method, we also need a starting interval. A quick look at Figure 19 suggests the interval  $[1, 3]$  for  $P_0$  as clearly the intersection point lies somewhere in between. We finally execute the bisection method:

```

1 a=1;b=3;tol_bisect=10^(-5);
2 P0_target=bisect(a,b,tol_bisect)

```

This computes the following value for target  $P_0$ :



**Figure 19:** Same plot with target value  $q_6 = 3 \times 10^{-3} \text{ m}^3/\text{s}$  indicated in green.

Lines 2-4 set up the constants, except  $P_0$ . Lines 6-8 solve Newton's method for the variable value of  $P_0$ . Line 10 returns  $q_6$  minus the target value.

No need to reduce the size of the interval further. The bisection method is guaranteed to converge if the interval contains the root.

Note the appearance of a second tolerance - this one for the bisection method. It is a measure of error for  $P_0$ . The tolerance in Newton's method is a measure of error for  $q$ . They don't have to be equal. In our case, the values of  $q_6$  are much smaller than the values of  $P_0$  so  $\text{tol}_{\text{bisect}}$  does not have to be as small (accurate).

```

Command Window
P0_target =
fx      2.089012145996094

```

in full agreement with Figure 19. As a further check we run Newton's method for this particular value of  $P_0$  and obtain:

```

1 x0=0.001*ones(6,1);
2 tol=10^(-7);
3 P0=P0_target;
4 newtonmult(x0,tol,P0)

```

The answer agrees with  $q_6 \approx 3 \times 10^{-3}$ .

```

Command Window
ans =
0.003059997546978
0.001525048278480
0.001534949268499
0.001505048278480
0.001494949268499
fx      0.002999997546978

```

It is always a good idea to check such answers, either graphically, or by solving Newton's method again for this value of  $P_0$ . The value of  $q_6$  should be very close to the target value  $3 \times 10^{-3}$ .

# Curve fitting

Curve fitting is the art of finding a function which fits to a finite number of data points. There are two attitudes towards solving that problem:

- Given data points  $(x_1, y_1), \dots, (x_n, y_n)$  find a function  $f$  such that  $f(x_i) = y_i$  for all  $i$ . The function exactly fits the data and may be turn out to be complicated for large  $n$ . This process is called interpolation and is typically used for a small number of data points.
- Given data points  $(x_1, y_1), \dots, (x_n, y_n)$  find a **simple** function  $f$  such that  $f(x_i) - y_i$  is minimized in some sense. The function does not quite fit the data but is easily used to underscore the structure of a large data set. This process is called least squares fitting, an example of which is linear regression (if  $f$  is a linear function).

These two ideas lie at opposite sides of the spectrum and are largely independent.

## Interpolation

Let us begin by defining the meaning of interpolation. We think of data points being given as measurements of a quantity  $y$  for different values of  $x$ .

**Definition 9.** Let  $(x_1, y_1), \dots, (x_n, y_n)$  be points in  $\mathbb{R}^2$  with distinct  $x$  coordinates. A function  $f : \mathbb{R} \rightarrow \mathbb{R}$  is said to interpolate the  $n$  points if

$$f(x_i) = y_i \quad \text{for all } 1 \leq i \leq n$$

We begin by examining the simplest interpolation problem. Given two points  $(x_1, y_1)$  and  $(x_2, y_2)$  with  $x_1 \neq x_2$ , we find a function  $f$  such that

$$f(x_1) = y_1, f(x_2) = y_2.$$

The solution is graphically obvious, take  $f$  to be the linear function going through both points, see Figure 20. It is simple enough to write

We want the function to be smooth, so using line segments to connect the points does not count as interpolation.

For instance if  $n = 3$  we cannot expect a linear function to go through three data points. But we can find the best line in a sense which has to be defined properly.

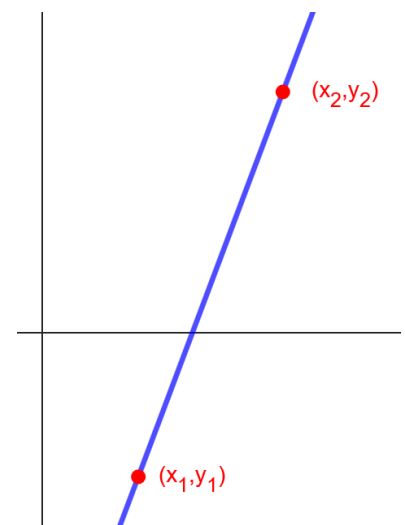


Figure 20: Two-point interpolation.

down the line's equation, letting  $y = f(x)$  be the linear function:

$$f(x) - y_1 = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1).$$

Although this formula for  $f(x)$  is correct, it would be more natural to derive a symmetrical formula where  $x_1$  and  $x_2$  play the same role. Splitting  $y_1$  and  $y_2$  apart yields:

$$\begin{aligned} f(x) &= y_1 + y_2 \frac{x - x_1}{x_2 - x_1} - y_1 \frac{x - x_1}{x_2 - x_1} \\ &= y_1 \left(1 - \frac{x - x_1}{x_2 - x_1}\right) + y_2 \frac{x - x_1}{x_2 - x_1} \end{aligned}$$

and finally (reversing the order):

$$f(x) = y_1 \frac{x - x_2}{x_1 - x_2} + y_2 \frac{x - x_1}{x_2 - x_1}.$$

This is a very satisfying formula for  $f(x)$ . It is helpful to rewrite

$$f(x) = y_1 L_1(x) + y_2 L_2(x)$$

with

$$L_1(x) = \frac{x - x_2}{x_1 - x_2} \quad L_2(x) = \frac{x - x_1}{x_2 - x_1}.$$

In this form we immediately notice

$$f(x_1) = y_1 L_1(x_1) + y_2 L_2(x_1) = y_1 \cdot 1 + y_2 \cdot 0 = y_1$$

and likewise  $f(x_2) = y_2$  so this function  $f$  does interpolate the two data points.

### *Lagrange interpolation*

We move on to  $n$ -point interpolation. In the same fashion that a line interpolates two points, we can construct a polynomial of degree  $n - 1$  to interpolate  $n$  points. It is called the Lagrange interpolation polynomial.

Since the case  $n = 3$  displays all the relevant features, we begin with three-point interpolation. Parabolas are used instead of straight lines (see Figure ), but the general formula is very similar.

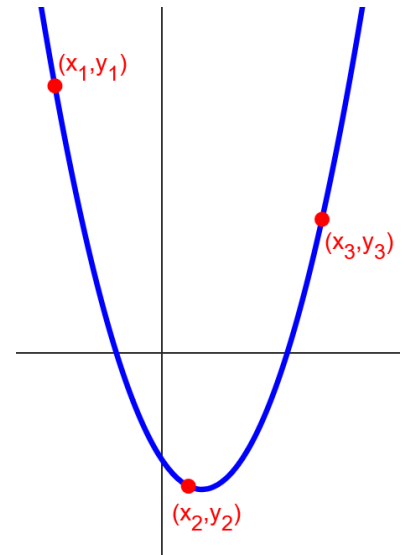
This form is symmetrical with respect to the two data points, unlike the first one.

Note that  $L_1$  and  $L_2$  satisfy:

$$L_1(x_1) = 1, L_1(x_2) = 0,$$

$$L_2(x_1) = 0, L_2(x_2) = 1.$$

Linear functions are polynomials of degree 1.



**Theorem 10.** Let  $(x_1, y_1)$ ,  $(x_2, y_2)$  and  $(x_3, y_3)$  be given points with distinct  $x$ -coordinates. Let

$$L_1(x) = \frac{(x - x_2)(x - x_3)}{(x_1 - x_2)(x_1 - x_3)}, \quad L_2(x) = \frac{(x - x_1)(x - x_3)}{(x_2 - x_1)(x_2 - x_3)},$$

$$L_3(x) = \frac{(x - x_1)(x - x_2)}{(x_3 - x_1)(x_3 - x_2)}.$$

Then the polynomial

$$P(x) = y_1 L_1(x) + y_2 L_2(x) + y_3 L_3(x)$$

interpolates the data points.

*Proof.* From the definition of  $L_1$  we obtain:

$$L_1(x_1) = \frac{(x_1 - x_2)(x_1 - x_3)}{(x_1 - x_2)(x_1 - x_3)} = 1, \quad L_1(x_2) = L_1(x_3) = 0.$$

Similarly we have

$$L_2(x_2) = 1, \quad L_2(x_1) = L_2(x_3) = 0$$

and

$$L_3(x_3) = 1, \quad L_3(x_1) = L_3(x_2) = 0.$$

By definition of  $P(x)$  we then obtain

$$f(x_1) = y_1 L_1(x_1) + y_2 L_2(x_1) + y_3 L_3(x_1) = 1 \cdot y_1 + 0 + 0 = y_1$$

and similarly

$$f(x_2) = y_2, \quad f(x_3) = y_3.$$

□

A similar proof yields a general theorem.

**Theorem 11.** Let  $(x_1, y_1), \dots, (x_n, y_n)$  be given points with distinct  $x$ -coordinates. For  $1 \leq i \leq n$ , let

$$L_i(x) = \frac{\prod_{j \neq i} (x - x_j)}{\prod_{j \neq i} (x_i - x_j)}.$$

Then the polynomial

$$P(x) = \sum_{i=1}^n y_i L_i(x)$$

interpolates the data points. The polynomial  $P$  is called the Lagrange interpolating polynomial.

**Objective:** Show that  $P(x_i) = y_i$  for  $i \in \{1, 2, 3\}$ .

Completely analogous with two-point interpolation.

$\prod$  is the symbol for product, it is akin to  $\Sigma$  for sums. For instance

$$\prod_{j \neq i} (x - x_j)$$

is

$$(x - x_1)(x - x_2) \cdots (x - x_{j-1}) \cdot (x - x_{j+1}) \cdots (x - x_n)$$

Theorem 11 is a rather cumbersome way of calculating an interpolating function, however it works well enough for small values of  $n$  and three-point interpolation will prove useful for numerical integration, see Definition ???.

### Small example

We find a third degree polynomial interpolating the four points  $(-1, 3)$ ,  $(0, 1)$ ,  $(1, 5)$  and  $(2, 2)$ . We begin by computing the polynomials  $L_1(x)$  to  $L_4(x)$ . By Theorem 11 we have

$$L_1(x) = \frac{(x - x_2)(x - x_3)(x - x_4)}{(x_1 - x_2)(x_1 - x_3)(x_1 - x_4)} = -\frac{x(x - 1)(x - 2)}{6}$$

Similarly

$$L_2(x) = \frac{(x - x_1)(x - x_3)(x - x_4)}{(x_2 - x_1)(x_2 - x_3)(x_2 - x_4)} = \frac{(x + 1)(x - 1)(x - 2)}{2}$$

$$L_3(x) = \frac{(x - x_1)(x - x_2)(x - x_4)}{(x_3 - x_1)(x_3 - x_2)(x_3 - x_4)} = -\frac{x(x + 1)(x - 2)}{2}$$

$$L_4(x) = \frac{(x - x_1)(x - x_2)(x - x_3)}{(x_4 - x_1)(x_4 - x_2)(x_4 - x_3)} = \frac{x(x - 1)(x + 1)}{6}$$

so that the Lagrange interpolation polynomial is

$$\begin{aligned} P(x) &= 3L_1(x) + L_2(x) + 5L_3(x) + 2L_4(x) \\ &= -\frac{13}{6}x^3 + 3x^2 + \frac{19}{6}x + 1 \end{aligned}$$

### Newton's divided differences

These calculations clearly become overwhelmingly complex for large values of  $n$ . Another concern is the fact that if we cannot reuse previous calculations if a new data point is added as the  $L_i(x)$  need to be recomputed from scratch.

A solution to both problems is using Newton's divided differences to compute the Lagrange interpolation polynomial in a simpler fashion. We look for an interpolating polynomial  $P(x)$  of the form

$$c_1 + c_2(x - x_1) + c_3(x - x_1)(x - x_2) + \cdots + c_n(x - x_1) \cdots (x - x_{n-1})$$

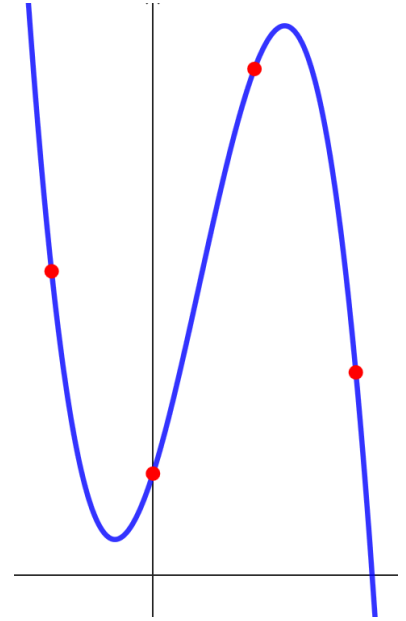
where the coefficients  $c_1, \dots, c_n$  are the unknowns. We want  $P(x)$  to satisfy the conditions

$$P(x_i) = y_i, \quad 1 \leq i \leq n.$$

In other words:

$x_1 = -1, x_2 = 0, x_3 = 1, x_4 = 2$  while  $y_1 = 3, y_2 = 1, y_3 = 5$  and  $y_4 = 2$ .

Easy to check that the interpolating conditions are satisfied, that is  $P(-1) = 3, P(0) = 1, P(1) = 5, P(2) = 2$ .



**Figure 21:** Third-degree polynomial  $P(x)$  interpolating  $(-1, 3)$ ,  $(0, 1)$ ,  $(1, 5)$  and  $(2, 2)$ .

That is the definition of interpolation.

- Setting  $x = x_1$  immediately gives

$$P(x_1) = y_1 = c_1.$$

- Setting  $x = x_2$  then gives

$$P(x_2) = y_2 = c_1 + c_2(x_2 - x_1) \Rightarrow c_2 = \frac{y_2 - y_1}{x_2 - x_1}.$$

since  $c_1 = y_1$

The expression for  $c_2$  is a so-called divided difference.

- Setting  $x = x_3$  then gives us

$$P(x_3) = y_3 = c_1 + c_2(x_3 - x_1) + c_3(x_3 - x_1)(x_3 - x_2)$$

that is

$$y_3 = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x_3 - x_1) + c_3(x_3 - x_1)(x_3 - x_2).$$

Using  $c_1 = y_1$  and  $c_2 = \frac{y_2 - y_1}{x_2 - x_1}$

Solving for  $c_3$  yields

$$c_3 = \frac{\frac{y_3 - y_1}{x_3 - x_1} - \frac{y_2 - y_1}{x_2 - x_1}}{x_3 - x_2}.$$

With a bit of algebra one can rewrite this expression in a more suitable form:

$$c_3 = \frac{\frac{y_3 - y_2}{x_3 - x_2} - \frac{y_2 - y_1}{x_2 - x_1}}{x_3 - x_1}.$$

We see a wonderful recursion at work here: the expression of  $c_3$  is a nestled divided difference, as both terms in the numerator are divided differences of the same kind as  $c_2$ . Defining the level one divided differences as

$$f[x_1 \ x_2] := \frac{y_2 - y_1}{x_2 - x_1} \quad f[x_2 \ x_3] := \frac{y_3 - y_2}{x_3 - x_2}$$

so that

$$c_3 = \frac{f[x_2 \ x_3] - f[x_1 \ x_2]}{x_3 - x_1}$$

it makes sense to call  $c_3$  a level two divided difference and use the notation

$$f[x_1 \ x_2 \ x_3].$$

In general we define the Newton divided differences as follows.

The top expression is

$$\frac{(y_3 - y_1)(x_2 - x_1) - (y_2 - y_1)(x_3 - x_1)}{(x_3 - x_2)(x_3 - x_1)(x_2 - x_1)}$$

and the bottom expression is

$$\frac{(y_3 - y_2)(x_2 - x_1) - (y_2 - y_1)(x_3 - x_2)}{(x_3 - x_2)(x_3 - x_1)(x_2 - x_1)}$$

The denominators are the same. Expanding both numerators shows that they are also equal.

**Definition 10.** Let  $(x_1, y_1), \dots, (x_n, y_n)$  be points with distinct  $x$ -coordinates. The Newton divided differences are defined recursively through

- $f[x_i, x_{i+1}] = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$
- $f[x_i, x_{i+1}, x_{i+2}] = \frac{f[x_{i+1}, x_{i+2}] - f[x_i, x_{i+1}]}{x_{i+2} - x_i}$
- $f[x_i, x_{i+1}, x_{i+2}, x_{i+3}] = \frac{f[x_{i+1}, x_{i+2}, x_{i+3}] - f[x_i, x_{i+1}, x_{i+2}]}{x_{i+3} - x_i}$

and so on and so forth.

We could define the level zero divided differences  $f[x_i] = y_i$  to set up the recursion one step lower.

Divided differences are defined for a range of  $i$  which should be obvious from the definition. For instance third-level  $f[x_i, x_{i+1}, x_{i+2}]$  is defined for  $1 \leq i \leq n-2$ .

Inspired by our derivation of  $c_3$  we state without proof the following theorem.

**Theorem 12.** Let  $(x_1, y_1), \dots, (x_n, y_n)$  be points with distinct  $x$ -coordinates. Define the coefficients  $c_i$ ,  $1 \leq i \leq n$  by

$$c_1 = y_1, c_2 = f[x_1, x_2], c_3 = f[x_1, x_2, x_3] \dots$$

$$c_n = f[x_1, x_2, x_3, \dots, x_n]$$

Then the polynomial

$$P(x) = c_1 + c_2(x - x_1) + c_3(x - x_1)(x - x_2) + \dots \\ \dots + c_n(x - x_1) \dots (x - x_{n-1})$$

interpolates the points  $(x_1, y_1), \dots, (x_n, y_n)$ .

Theorem 12 allows us to compute the coefficients of an interpolating polynomial quickly in a recursive fashion. Moreover were we to add one data point  $(x_{n+1}, y_{n+1})$  we only need to compute one extra term

$$c_{n+1}(x - x_1) \dots (x - x_n).$$

The previous  $n$  ones are still correct.

### Implementation

In order to use Theorem 12 we need to set up an algorithm that computes all possible divided differences and stores the results in a matrix  $D$ .

- The first column of  $D$  is initialized by  $D(i, 1) = y_i$  for  $1 \leq i \leq n$ .
- The second column contains the level one divided differences that is

For instance  $c_3$  needs both  $f[x_1, x_2]$  and  $f[x_2, x_3]$ . Then  $c_4$  needs  $f[x_1, x_2, x_3]$ ,  $f[x_2, x_3, x_4]$ , the latter requiring  $f[x_3, x_4]$ . So we see we have to compute every possible combination of divided differences.

$D(i+1, 1) = y_{i+1}$  and  $D(i, 1) = y_i$  by step 1.



$$D(i,2) = f[x_i \ x_{i+1}] = \frac{D(i+1,1) - D(i,1)}{x_{i+1} - x_i}$$

Clearly we only allow  $1 \leq i \leq n-1$ .

- The third column contains the level two divided differences that is

$$D(i,3) = f[x_i \ x_{i+1} \ x_{i+2}] = \frac{D(i+1,2) - D(i,2)}{x_{i+2} - x_i}$$

We only allow  $1 \leq i \leq n-2$ .

- And so on until we reach column  $n$  which has only one element.

In the end we do not need the whole matrix  $D$ , only the topmost coefficients corresponding to

$$y_1, f[x_1 \ x_2], f[x_1 \ x_2 \ x_3] \dots$$

They are assigned to the output vector  $c$ . The inputs of the program are two vectors  $x$  and  $y$  of equal length, representing the data points  $(x_i, y_i)$ . This leads to the following code.

```

1  function c=divideddiff(x,y)
2      n=length(x);
3      for i=1:n %set column 1, j=1
4          D(i,1)=y(i);
5      end
6
7      for j=2:n %set column j
8          for i=1:(n-j+1)
9              D(i,j) = ( D(i+1,j-1)-D(i,j-1) ) /
10                 ( x(i+j-1)-x(i) );
11          end
12      end
13
14      for j=1:n
15          c(j)=D(1,j);
16      end
end

```

This algorithm calculates the coefficients  $c_i$  of the interpolating polynomial in form

$$c_1 + c_2(x - x_1) + c_3(x - x_1)(x - x_2) + c_n(x - x_1) \cdots (x - x_{n-1})$$

We therefore need a companion program which can evaluate or plot this polynomial, knowing the coefficients. For efficiency's sake it is better to write the polynomial in so-called nested form

$$D(i+1,2) = f[x_{i+1} \ x_{i+2}]$$

$$D(i,2) = f[x_i \ x_{i+1}] \text{ by step 2.}$$

that is  $f[x_1 \ x_2 \ \cdots \ x_n]$

$n$  is the number of data points. You might want to check that  $x$  and  $y$  have the same length.

Lines 3-5 initialize the first column as the  $y$ -values.

Lines 7-11 go recursively through the divided differences, one level ( $j$ ) at a time. The bounds for  $i$  shrink as  $j$  grows as shown earlier.

Lines 13-15 recover the coefficients  $c_i$  as the first divided differences at each level i.e.  $y_1, f[x_1 \ x_2], f[x_1 \ x_2 \ x_3] \dots$

For  $n = 4$  the nested form is  
 $y_1 + (x - x_1)(c_2 + (x - x_2)(c_3 + (x - x_3)c_4))$ .

Nested form drastically reduces the number of multiplications needed to evaluate a polynomial.

$$c_1 + (x - x_1) \left( c_2 + (x - x_2) \left( c_3 + (x - x_3) (\cdots + c_n (x - x_{n-1})) \right) \right).$$

The following program takes as inputs the coefficients  $c_i$ , the interpolation points  $x_i$  (both in vector form) as well as the real variable  $t$  to compute the value of the polynomial at  $t$ .

```

1 function y=nested(c,x,t)
2     n=length(c);
3     y=c(n);
4     for i=n-1:-1:1
5         y=c(i)+y.*(t-x(i));
6     end
7 end

```

Easier to call it  $t$  rather than  $x$  if we use  $x_i$  for the interpolation points.

Note the dot on line 5 as we will want  $t$  to be a vector, e.g. for plotting purposes.

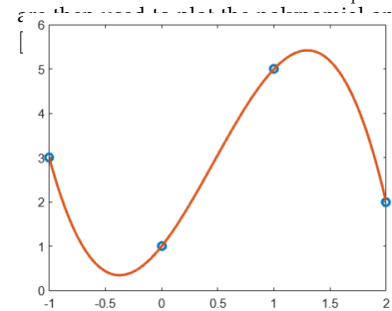
It is now simple to use both programs to compute an interpolating polynomial and plot it. The following code draws Figure 22.

```

1 x=[-1 0 1 2];
2 y=[3 1 5 2];
3 c=divideddiff(x,y);
4 t=-1:0.01:2;
5 y_plot=nested(c,x,t);
6 plot(x,y,'o',t,y_plot)

```

$x$  and  $y$  are the coordinates of the interpolation points as on Figure 22. The vector  $c$  contains the coefficients in nested form. The vectors  $t$  and  $y_{\text{plot}}$



**Figure 22:** Interpolation polynomial through  $(-1, 3)$ ,  $(0, 1)$ ,  $(1, 5)$  and  $(2, 2)$ .

### Runge phenomenon

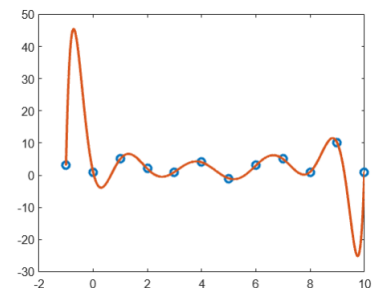
Newton's divided differences compute the Lagrange polynomial with maximum accuracy, up to machine precision. Unfortunately Lagrange interpolation becomes unstable for larger values of  $n$ . Using  $n = 12$  data points and the following code, we draw Figure 23.

```

1 x=[-1 0 1 2 3 4 5 6 7 8 9 10];
2 y=[3 1 5 2 1 4 -1 3 5 1 10 1];
3 c=divideddiff(x,y);
4 t=-1:0.01:10;
5 y_plot=nested(c,x,t);
6 plot(x,y,'o',t,y_plot)

```

We notice unreasonably large oscillations on  $[-1, 0]$  and  $[9, 10]$ . This interpolation polynomial, although it goes through the twelve data points, cannot be used for predicting purposes near the beginning and end of the interval. This problem is called the Runge phenomenon and is an unavoidable artifact of Lagrange interpolation:



**Figure 23:** Interpolation polynomial through twelve equally spaced points on the interval  $[-1, 10]$ . Notice large oscillations at the end parts of the interval although all  $y$ -values are relatively small.

polynomials of high degree such as the eleventh degree polynomial of Figure 23 very often oscillate wildly. A solution is to use several formulas (usually third-degree polynomials) and glue them together, a method known as cubic spline interpolation.

### Linear least squares

A different approach to curve fitting is to use a single simple function (such as a line) to best approximate a large data set. We first study the linear case before using more complex functions. We are given  $n$  data points  $(x_1, y_1), \dots, (x_n, y_n)$  and want to find a linear function

$$y = hx + k$$

which best fits the data.

Ideally the four points would be on the line, yielding the system

$$\begin{cases} hx_1 + k = y_1 \\ hx_2 + k = y_2 \\ \vdots \\ hx_n + k = y_n \end{cases}$$

This linear system has  $n$  equations and just two unknowns  $h$  and  $k$ , so in general we have no solution. The objective is to find the values for  $h$  and  $k$  such that the equations are as true as can be. We see curve fitting is essentially the same problem as trying to solve an overdetermined linear system, that is, one with more equations than variables. Our system is  $Ax = b$  with

$$A = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix}, x = \begin{pmatrix} h \\ k \end{pmatrix}, b = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

It has no exact solution however our aim is to minimize the so-called residual error

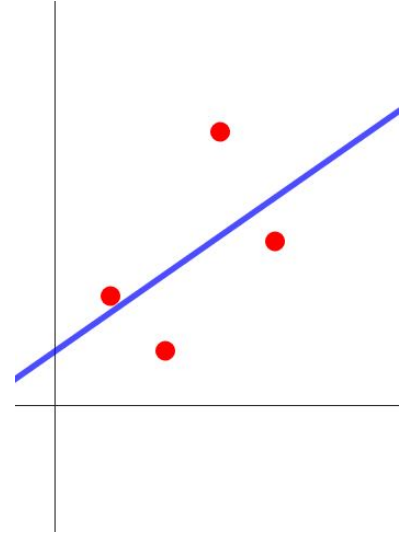
$$|Ax - b|$$

where  $|\cdot|$  is the usual vector length in  $\mathbb{R}^n$ .

### The normal equations

It turns out that we can find simple equations which this error-minimizing vector  $x$  must satisfy: the so-called normal equations.

Consider the squared residual error  $|Ax - b|^2$ :



**Figure 24:** Best line for  $(1, 2)$ ,  $(2, 1)$ ,  $(3, 5)$  and  $(4, 3)$ .

unless the points happen to be all on the same line

Intuition: we find the vector  $x$  which gives the result closest to the target  $b$  when plugged into the system.

It is much more practical to square the length  $|Ax - b|$  to get rid of unnecessary square roots.

$$(hx_1 + k - y_1)^2 + (hx_2 + k - y_2)^2 + \cdots + (hx_n + k - y_n)^2.$$

Here we consider  $x_i$  and  $y_i$  to be given, while  $h$  and  $k$  are the unknowns. Therefore the problem is to find a minimum of the function of two variables

$$E(h, k) = \sum_{i=1}^n (hx_i + k - y_i)^2.$$

According to calculus such a minimum is found where the gradient of  $E$  is equal to 0. We compute both derivatives using the chain rule:

$$\frac{\partial E}{\partial h} = \sum_{i=1}^n 2x_i(hx_i + k - y_i) = 0$$

$$\frac{\partial E}{\partial k} = \sum_{i=1}^n 2(hx_i + k - y_i) = 0$$

which we rewrite as

$$\begin{cases} \left( \sum_{i=1}^n x_i^2 \right) h + \left( \sum_{i=1}^n x_i \right) k = \sum_{i=1}^n x_i y_i \\ \left( \sum_{i=1}^n x_i \right) h + nk = \sum_{i=1}^n y_i \end{cases}$$

This is a new linear system for  $h$  and  $k$ , but this time it has only two equations, so we expect a unique solution to be found. The solution of that system minimizes the error and is to be thought of the best approximation to a solution. Moreover this system is deeply tied to our original unsolvable system  $Ax = b$ . Indeed consider the transpose matrix  $A^T$ :

$$A^T = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ 1 & 1 & \cdots & 1 \end{pmatrix}$$

Direct calculations yield

$$A^T A = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ 1 & 1 & \cdots & 1 \end{pmatrix} \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^n x_i^2 & \sum_{i=1}^n x_i \\ \sum_{i=1}^n x_i & n \end{pmatrix}$$

and similarly

$$A^T b = \begin{pmatrix} x_1 & x_2 & \cdots & x_n \\ 1 & 1 & \cdots & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^n x_i y_i \\ \sum_{i=1}^n y_i \end{pmatrix}$$

$E$  for error

$\nabla E = \left( \frac{\partial E}{\partial h} \frac{\partial E}{\partial k} \right)$  for a function of two variables  $h$  and  $k$ .

dividing by 2 along the way.

Notice  $\sum_{i=1}^n k = nk$  as the summand doesn't depend on  $i$ .

Remember that the  $x_i$  and  $y_i$  are given numbers, and that  $h, k$  are the unknowns.

$$\text{Recall } A = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix}, b = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

Since  $A^T$  is  $2 \times n$  and  $A$  is  $n \times 2$ , then  $A^T A$  is a square  $2 \times 2$  matrix.

We notice that the new  $2 \times 2$  linear system is none other than

$$A^T A x = A^T b$$

which is called the normal equations. In general we can prove the following:

**Theorem 13.** Let  $Ax = b$  be a linear system with more equations than variables. Define the normal equations as the square linear system:

$$A^T A x = A^T b.$$

Then the solution to the normal equations minimizes the residual error  $|Ax - b|$ .

### Implementation

Solving the normal equations is simple - it could even be done by hand for a small system. Most of the work is setting up the system  $Ax = b$  which fits to our model. We study an example where we fit a sinusoidal curve to data by solving the appropriate normal equations.

Our data set consists of monthly average temperatures in Reykjavík over the 2010-2019 time period. This yields a total of 120 equally spaced measurements which are plotted on Figure 25.

We are not surprised to notice that the data is roughly periodic with period 12 months. This suggests the following model for temperature as a function of time  $t$  (in months):

$$T = c_1 + c_2 \cos\left(\frac{2\pi}{12}t\right) + c_3 \left(\frac{2\pi}{12}t\right)$$

where  $c_1$ ,  $c_2$  and  $c_3$  are coefficients to be determined. This is obviously a simplification as the data is not quite periodic but replaces noisy discrete data by a single function.

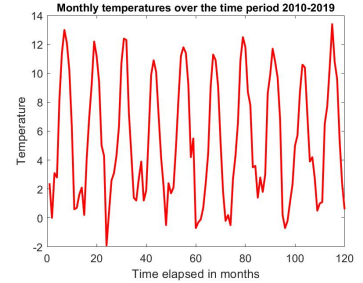
Temperature measurements are partly displayed in Table 5. Plugging this data into our temperature for  $T$  yields a total of 120 equations:

$$\begin{cases} c_1 + c_2 \cos\left(\frac{2\pi}{12}\right) + c_3 \left(\frac{2\pi}{12}\right) & = 2.4 \\ c_1 + c_2 \cos\left(\frac{2\pi}{12} \cdot 2\right) + c_3 \left(\frac{2\pi}{12} \cdot 2\right) & = 0.0 \\ c_1 + c_2 \cos\left(\frac{2\pi}{12} \cdot 3\right) + c_3 \left(\frac{2\pi}{12} \cdot 3\right) & = 3.1 \\ \vdots & \vdots \\ c_1 + c_2 \cos\left(\frac{2\pi}{12} \cdot 120\right) + c_3 \left(\frac{2\pi}{12} \cdot 120\right) & = 0.6 \end{cases}$$

for instance the coefficient of  $h$  in the upper equation is  $\sum x_i^2$ .

The proof is not much more complicated than the above derivation.

Although sin and cos are not linear functions, we shall see that the model consists of linear equations in the coefficients of these functions.



**Figure 25:** Plotting the monthly temperature data.

Note that both  $\cos(2\pi/12t)$  and  $\sin(2\pi/12t)$  are periodic with period 12. The coefficient  $c_1$  is necessary otherwise the average temperature would also be 0.

Month	Temperature
1	2.4
2	0.0
3	3.1
4	2.8
5	8.2
$\vdots$	$\vdots$
120	0.6

**Table 5:** Monthly average temperatures in Reykjavík. Months are counted from January 2010 to December 2019. Source: Icelandic Meteorological Office.

which is a  $120 \times 3$  linear system for  $c_1$ ,  $c_2$  and  $c_3$ . As usual we think of it in the form  $Ax = b$  where

$$A = \begin{pmatrix} 1 & \cos\left(\frac{2\pi}{12}\right) & \sin\left(\frac{2\pi}{12}\right) \\ 1 & \cos\left(\frac{2\pi}{12} \cdot 2\right) & \sin\left(\frac{2\pi}{12} \cdot 2\right) \\ 1 & \cos\left(\frac{2\pi}{12} \cdot 3\right) & \sin\left(\frac{2\pi}{12} \cdot 3\right) \\ \vdots & \vdots & \vdots \\ 1 & \cos\left(\frac{2\pi}{12} \cdot 120\right) & \sin\left(\frac{2\pi}{12} \cdot 120\right) \end{pmatrix}, \quad x = \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix}$$

and  $b$  is the  $120 \times 1$  temperature data vector. Finding the best coefficients is then a matter of coding these objects and solving the normal equations.

```

1 nbreadings = length(temperatures);
2 t=linspace(1,nbreadings,nbreadings)';
3 A(:,1)=ones(nbreadings,1);
4 A(:,2)=cos(pi/6*t);
5 A(:,3)=sin(pi/6*t);
6
7 c=(A'*A) \ (A'*temperatures)
```

The  $120 \times 1$  vector **temperatures** contains the data. As usual it is much more efficient to avoid loops when constructing large objects in Matlab, instead we use the time array  $t = [1 \ 2 \ 3 \ \dots \ 120]$ .  $A'$  is the transpose matrix  $A^T$ .

The outcome of this code is the best values for the coefficients:

Command Window
▼

```

c =

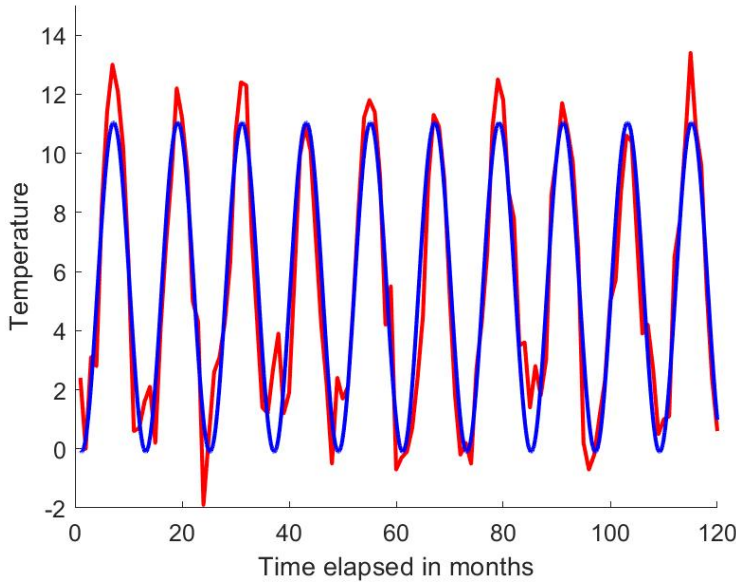
    5.468333333333333
   -4.484273975197159
fx -3.309200415013808
```

The vector  $c$  contains  $c_1$ ,  $c_2$ ,  $c_3$ .

The next step is to plot

$$f(t) = c_1 + c_2 \cos\left(\frac{2\pi}{12}t\right) + c_3 \sin\left(\frac{2\pi}{12}t\right)$$

for these values of  $c_1$ ,  $c_2$  and  $c_3$  alongside the real-world data.



Real data is in red, while the periodic model  $f(t)$  is in blue.

Our simple periodic model performs somewhat accordingly, although it seems to undershoot the high temperatures and struggle with shorter oscillations. A way to improve the model is to incorporate more sine and cosine functions with higher frequency. This is consistent with Fourier analysis theory: a smooth function of period  $T$  can be represented as an infinite sum

in our case  $T = 12$  months.

$$\frac{a_0}{2} + \sum_{n=1}^{\infty} \left( a_n \cos \left( n \frac{2\pi}{T} t \right) + b_n \sin \left( n \frac{2\pi}{T} t \right) \right)$$

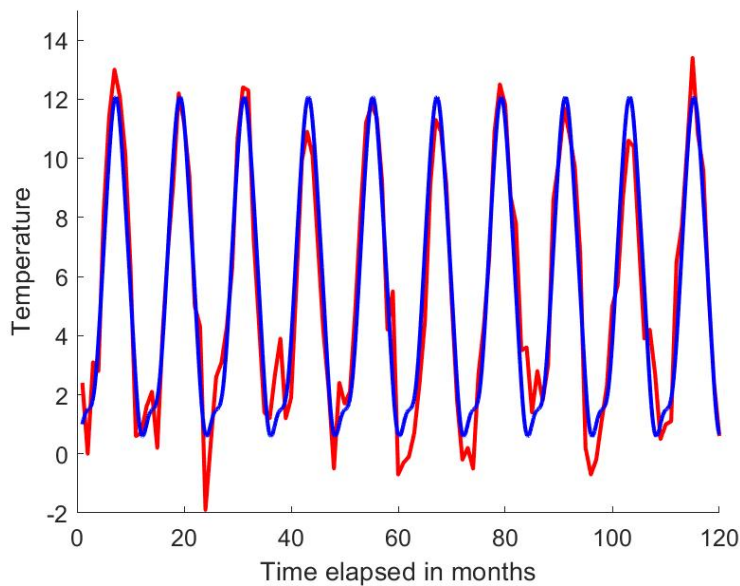
Using a total of nine coefficients, we experiment with the following model:

$$f(t) = c_1 + c_2 \cos \left( \frac{2\pi}{12} t \right) + c_3 \left( \frac{2\pi}{12} t \right) + c_4 \cos \left( 2 \frac{2\pi}{12} t \right) + c_5 \left( 2 \frac{2\pi}{12} t \right)$$

$$c_6 \cos \left( 3 \frac{2\pi}{12} t \right) + c_7 \left( 3 \frac{2\pi}{12} t \right) + c_8 \cos \left( 4 \frac{2\pi}{12} t \right) + c_9 \left( 4 \frac{2\pi}{12} t \right)$$

where the coefficients  $c_1, \dots, c_9$  are to be computed. We can set up a  $120 \times 9$  linear system exactly as before. The normal equations are now a  $9 \times 9$  system for all involved coefficients, which is solved as earlier. Plotting the new model  $f(t)$  alongside the data yields the following chart.

The only difference is how the matrix  $A$  is defined, we need six extra code lines to define columns 4 to 9.



The second model seems to perform better - can we measure by how much? One way is to compute the residual error in both cases. Recall that is defined by

$$\text{residual error} = \|Ac - b\|$$

where  $c$  is the solution to the normal equations and  $Ax = b$  is the original non-square system. In both cases the residual error is the length of a vector in  $\mathbb{R}^{120}$ . In Matlab we add the following line after the normal equations are solved:

```
residual_error=norm(A*c-b)
```

The first model has a residual error of 15.47 while the second has a residual error of 13.13 which convinces us that it is the better model: a smaller residual error is a sign of a better fit to the whole data.

It is often customary to instead compute the so-called Root Mean Square Error (RMSE) which is the residual error divided by the number of measurements, here 120. Since we hold that number constant in these experiments, it does not matter which formula we use.

### *Non-linear least squares*

So far we have only studied linear models, so that curve fitting was a matter of solving a linear system with more equations than variables. More advanced models are likely to be nonlinear, so we are faced with the task of solving a non-linear system with more equations than variables. We adopt the same formalism as with Newton's



method. Such a system in  $n$  variables  $x_1, \dots, x_n$  will be of the form

$$\begin{cases} f_1(x_1, \dots, x_n) = 0 \\ f_2(x_1, \dots, x_n) = 0 \\ \vdots \\ f_m(x_1, \dots, x_n) = 0 \end{cases}$$

with the  $f_i$  being functions of  $n$  variables and  $m > n$ . It will be practical to summarize this system in one vector-valued equation

$$F(\mathbf{x}) = \mathbf{0}$$

where  $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$  and  $\mathbf{x} = (x_1 \ x_2 \ \dots \ x_n)^T$ .

### Gauss-Newton method

Recall that for a function  $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$ , Newton's method was implemented as a two-step method:

- Solve the linear system  $J(\mathbf{x}_n)\mathbf{s} = F(\mathbf{x}_n)$  for  $\mathbf{s}$ .
- Let  $\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{s}$ .

where  $J(\mathbf{x}_n)$  is the Jacobi matrix of  $F$ , evaluated at  $\mathbf{x}_n$ . In the case of  $m$  equations with  $m > n$ , the Jacobi matrix becomes  $m \times n$  and has no inverse. We fix this issue in the spirit of the normal equations, multiplying by the transpose Jacobi matrix to square the system.

**Definition 11.** Let  $F : \mathbb{R}^n \rightarrow \mathbb{R}^m$  with  $m > n$  and  $\mathbf{x}_0 \in \mathbb{R}^n$ . The Gauss-Newton method is defined inductively by the two-step method:

- Solve the linear system  $J(\mathbf{x}_n)^T J(\mathbf{x}_n)\mathbf{s} = J(\mathbf{x}_n)^T F(\mathbf{x}_n)$  for  $\mathbf{s}$ .
- Let  $\mathbf{x}_{n+1} = \mathbf{x}_n - \mathbf{s}$ .

Intuitively Gauss-Newton functions as Newton's method, except only an approximate solution through least squares is computed at each step. It is therefore even less stable than regular Newton's method and one should carefully choose the initial guess  $\mathbf{x}_0$ .

We did the same when implementing multivariate Newton's method, except that  $m = n$  in that case.

The one-step definition was slightly different:  $\mathbf{x}_{n+1} = \mathbf{x}_n - J(\mathbf{x}_n)^{-1}F(\mathbf{x}_n)$  but we should avoid computing an inverse matrix if possible.

Only square matrices can be invertible.

$J$  is a  $m \times n$  matrix so  $J^T J$  is  $n \times n$ . Similarly  $F(\mathbf{x}_n) \in \mathbb{R}^m$  so  $J^T F \in \mathbb{R}^n$  and Step 1 solves a  $n \times n$  linear system.

The abstract code is as follows:

```

1 function x=gaussnewton(x0,tol)
2     x=x0; oldx=x+2*tol;
3     while norm(x-oldx)>tol
4         oldx=x;
5         J=jac(x);
6         s=(J'*J) \ (J'*F(x));
7         x=x-s;
8     end
9 end

```

### Implementation

The field of pharmacokinetics is concerned with the speed of absorption and elimination of medication under various circumstances. When a drug is given orally the concentration of the drug in the blood stream is given by the Bateman equation

$$Q(t) = Q_0 \frac{k_a}{k_a - k_e} \left( e^{-k_e t} - e^{-k_a t} \right)$$

where  $k_a$  is the rate of absorption of the drug, and  $k_e$  is the rate of elimination out of the body. The constant  $Q_0$  is a normalizing factor. We can assume  $k_a > k_e$  (otherwise the drug is eliminated faster than it concentrates) so the fraction is well-defined. A solution to the Bateman equation for some realistic constants is displayed in Figure 26.

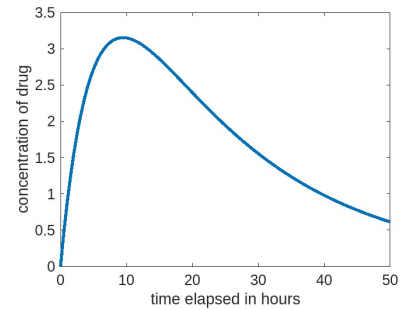
The three parameters are unknown beforehand and it is very important to estimate them in order to administrate the right amount of medication. We do so by using Gauss-Newton method to fit the concentration function  $Q(t)$  to measurement data. The Bateman equation is highly non-linear with respect to  $k_a$  and  $k_e$  and we cannot use linear least squares. Experimental data is displayed in Table 6.

Using this data we obtain a (non-linear) system of eight equations in the three unknowns  $Q_0$ ,  $k_a$  and  $k_e$ :

$$\begin{cases} Q_0 \frac{k_a}{k_a - k_e} \left( e^{-0 \cdot k_e} - e^{-0 \cdot k_a} \right) - 0 = 0 \\ Q_0 \frac{k_a}{k_a - k_e} \left( e^{-5k_e} - e^{-5k_a} \right) - 2.8 = 0 \\ Q_0 \frac{k_a}{k_a - k_e} \left( e^{-10k_e} - e^{-10k_a} \right) - 3.1 = 0 \\ \vdots \\ Q_0 \frac{k_a}{k_a - k_e} \left( e^{-50k_e} - e^{-50k_a} \right) - 0.6 = 0 \end{cases}$$

As usual we assume  $F(x)$  and its Jacobi matrix to be defined as functions of a vector  $x \in \mathbb{R}^n$  in Matlab.

Line 5 is only for efficiency's sake, calling the Jacobi subroutine once instead of three times.



**Figure 26:** Solution to the Bateman equation for constants corresponding to the anti-epileptic drug Fenitoin. The concentration in µg/mL spikes quickly after ingestion and slowly decays thereafter.

Time (hours)	Concentration
0	0
5	2.8
10	3.1
15	2.9
20	2.5
30	1.6
40	1.0
50	0.6

**Table 6:** Experimental data showing concentration of a drug in µg/mL at eight times.

We need to move the concentration values to the left in order to obtain a system of the form  $F(x) = 0$ .

To set the system up in vector form we define:

$$x = \begin{pmatrix} Q_0 \\ k_a \\ k_e \end{pmatrix}, F(x) = \begin{pmatrix} F_1(x) \\ F_2(x) \\ \vdots \\ F_8(x) \end{pmatrix} := \begin{pmatrix} Q_0 \frac{k_a}{k_a - k_e} (e^{-0 \cdot k_e} - e^{-0 \cdot k_a}) - 0 \\ Q_0 \frac{k_a}{k_a - k_e} (e^{-5k_e} - e^{-5k_a}) - 2.8 \\ \vdots \\ Q_0 \frac{k_a}{k_a - k_e} (e^{-50k_e} - e^{-50k_a}) - 0.6 \end{pmatrix}$$

Here  $F : \mathbb{R}^3 \rightarrow \mathbb{R}^8$  so  $m = 3, n = 8$ .

We can define  $F$  as a function in Matlab:

```
1 function y=F(x)
2     Q0=x(1); ka=x(2); ke=x(3);
3     t=[0 5 10 15 20 30 40 50]';
4     c=[0 2.8 3.1 3.0 2.5 1.5 1 0.65]';
5     y=Q0*ka/(ka-ke)*(exp(-t*ke)-exp(-t*ka))-c;
6 end
```

$x$  is a vector in  $\mathbb{R}^3$ , it is practical to define its components as  $Q_0, k_a$  and  $k_e$  for easier typesetting of the various formulas.

Our final calculation is to find the Jacobi matrix of  $F$ . To do so we need to differentiate  $F_1, \dots, F_8$  with respect to  $Q_0, k_a$  and  $k_e$ . This is not the simplest calculation by hand, so we use symbolic differentiation in Matlab to compute these derivatives and hard code the results.

```
1 syms Q0 ka ke t c;
2 y=Q0*ka/(ka-ke)*(exp(-t*ke)-exp(-t*ka))-c;
3 diff(y,Q0)
4 diff(y,ka)
5 diff(y,ke)
```

We also define a symbolic variable  $t$  which then takes the values 0, 5, 10, ..., 50. The symbolic variable  $c$  denotes the experimental concentration data which vanishes when differentiated.

yielding

Command Window

```
diff(y,Q0) =
-(ka*(exp(-ka*t) - exp(-ke*t)))/(ka - ke)

diff(y,ka) =
(Q0*ka*(exp(-ka*t) - exp(-ke*t)))/(ka - ke)^2
- (Q0*(exp(-ka*t) - exp(-ke*t)))/(ka - ke) + (
Q0*ka*t*exp(-ka*t))/(ka - ke)

diff(y,ke) =
- (Q0*ka*(exp(-ka*t) - exp(-ke*t)))/(ka - ke)
fx ^2 - (Q0*ka*t*exp(-ke*t))/(ka - ke)
```

We use these answers to define the Jacobi matrix as a function.

```

1 function J=jac(x)
2     Q0=x(1);ka=x(2);ke=x(3);
3     t=[0 5 10 15 20 30 40 50]';
4     J=[-(ka*(exp(-ka*t) - exp(-ke*t)))/(ka - ke)
5         ,
6         (Q0*ka*(exp(-ka*t) - exp(-ke*t)))/(ka -
7             ke)^2 - (Q0*(exp(-ka*t) - exp(-ke*t)))/
            (ka - ke) + (Q0*ka.*t.*exp(-ka*t))/(
            ka - ke),
            - (Q0*ka*(exp(-ka*t) - exp(-ke*t)))/(ka -
            ke)^2 - (Q0*ka.*t.*exp(-ke*t))/(ka -
            ke)];
8 end

```

With these subroutines in hand we are ready to run Gauss-Newton method. Input variables are an initial vector  $x_0 \in \mathbb{R}^3$  and the error tolerance.

Typical absorption and elimination rates are around

$$k_a \approx 0.1 \quad k_e \approx 0.05$$

which we can use as initial guesses. To determine an initial value for  $Q_0$  we use the above two values together with one of the equations for instance

$$Q_0 \frac{k_a}{k_a - k_e} (e^{-5k_e} - e^{-5k_a}) - 2.8 = 0.$$

Letting  $k_a = 0.1, k_e = 0.05$  gives a very rough approximation:

$$Q_0 \cdot 0.3 - 2.8 = 0 \Rightarrow Q_0 \approx 8.$$

An initial vector hopefully likely to succeed is therefore

$$x_0 = \begin{pmatrix} 8 \\ 0.1 \\ 0.05 \end{pmatrix}.$$

Using an error tolerance of  $10^{-6}$ , we run

```

1 x0=[8 0.1 0.05]'; tol=10^(-6);
2 gaussnewton(x0,tol)

```

yielding the least squares solution

It is **critical** to refrain from using symbolic differentiation within loops such as Gauss-Newton method as these calculations are time-consuming. Better copy paste the results into a hard coded function such as jac.

Dots have to be added wherever vectors are multiplied together for instance  $te^{-k_a t}$ .

Recall  $k_e < k_a$  otherwise the drug is eliminated faster than it enters the body.

No need for any accuracy here.

There is a large range of good initial vectors, however Gauss-Newton diverges when starting at  $Q_0 = 3$ .

$x_0$  needs to be a column vector the way our equations are set up.

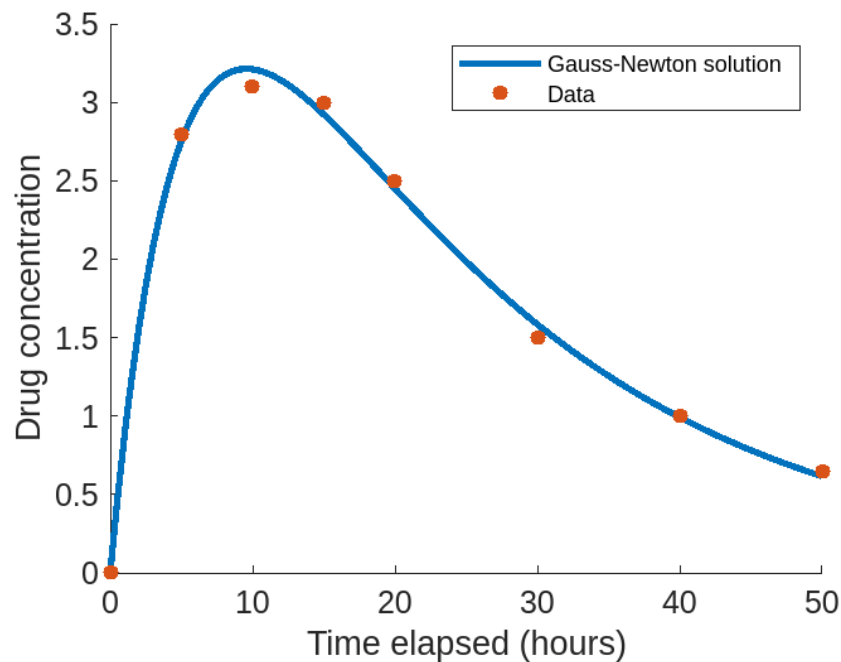


showing the best-fitting values for  $Q_0$ ,  $k_a$  and  $k_e$  in that order. The last step is to plot the Bateman function

$$Q(t) = Q_0 \frac{k_a}{k_a - k_e} \left( e^{-k_e t} - e^{-k_a t} \right)$$

for these value of the parameters, alongside the experimental data. We obtain a very satisfactory result.

Notice how far off our initial guess was!



The function  $Q(t)$  can now be used to read important information such as maximum drug concentration, or when does the concentration goes under a certain threshold.



# Numerical integration

Estimating a definite integral

$$\int_a^b f(x) dx$$

is an important theoretical and practical problem. In many cases integrals are not computable exactly so the only way forward is numerical estimations. We will not attempt to find primitive functions, instead our approach will be geometrical: estimating such an integral is the same as finding the (signed) area of the domain which lies under the graph of  $f(x)$ , see Figure 27.

## Newton-Cotes formulas

The idea behind simple integrating formulas is to approximate  $f(x)$  by a much simpler function on the interval  $[a, b]$ , and compute the integral of the approximation. Newton-Cotes formulas use polynomials, and we study in more detail the three simplest examples:

- Approximation by a constant: **the endpoint rule**.
- Approximation by a linear function: **the trapezoid rule**.
- Approximation by a quadratic function: **Simpson's rule**.

All three methods are derived in the same fashion: replace  $f(x)$  by the simpler function, integrate the simpler function on  $[a, b]$  and use the formula as an approximation for  $\int_a^b f(x) dx$ . Error analysis is then performed for each formula.

## Endpoint rule

The endpoint rule is performed by approximating  $f(x)$  by the constant  $f(a)$ , see Figure 28.

A most famous example of a non-computable integral being the Gaussian integral  $\int_0^a e^{-x^2} dx$  used to compute probabilities connected to the normal / (Gaussian) distribution.

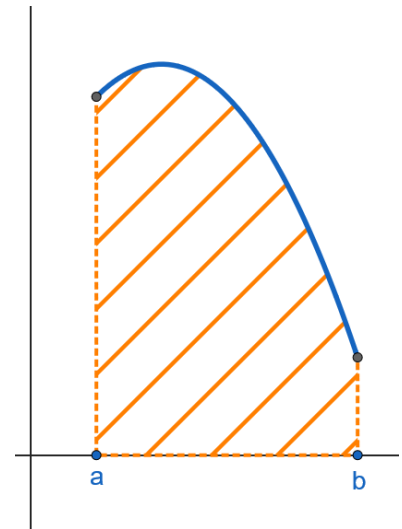


Figure 27: The integral  $\int_a^b f(x) dx$  displayed as area.

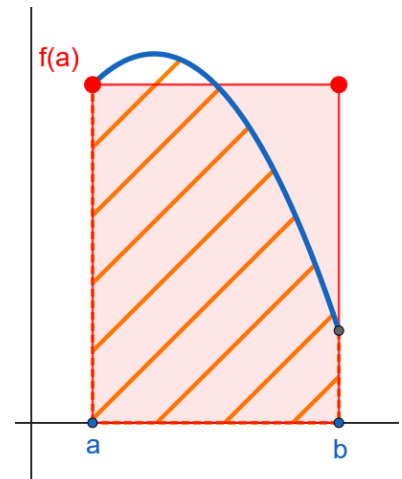


Figure 28: Endpoint rule on the interval  $[a, b]$ . The area of the red rectangle is  $(b - a) \cdot f(a) = hf(a)$ .

**Definition 12.** The (left) endpoint rule is the approximation

$$\int_a^b f(x) dx \approx hf(a)$$

where  $h = b - a$  is the length of the interval.

We could just as well define the right endpoint rule:

$$\int_a^b f(x) dx \approx hf(b).$$

Obviously this approximation is rather poor, especially if the function changes quickly or the interval is large. Somewhat surprisingly we can estimate the error precisely, even without calculating the exact value for the integral.

**Theorem 14.** Assume  $f$  is differentiable on  $[a, b]$  and let  $M$  be the maximum value for  $|f'(x)|$  there. The error in the endpoint rule

$$\int_a^b f(x) dx \approx hf(a)$$

is at most  $M \frac{h^2}{2}$  where  $h = b - a$ .

This theorem tells us the theoretically largest error that can come up while using the endpoint rule. In general the actual error will be smaller, although we cannot predict by how much.

The error is to be understood as

$$\text{Error} = \left| \int_a^b f(x) dx - hf(a) \right|$$

Error = |Real value –  
Approximated value|.

*Proof.* Using  $h = b - a$ , that is  $b = a + h$ , define:

$$E(h) = \int_a^{a+h} f(x) dx - hf(a).$$

so that Error =  $|E(h)|$

We clearly have  $E(0) = 0$ , since both terms are 0 if  $h = 0$ . The fundamental theorem of analysis then shows

$$E'(h) = f(a + h) - f(a).$$

The derivative of  $\int_a^t f(x) dx$  is  $f(t)$ .  
The derivative of  $hf(a)$  w.r.t. to  $h$  is simply  $f(a)$ .

Again we obviously have  $E'(0) = 0$ . Finally:

$$E''(h) = f'(a + h).$$

since  $f(a)$  is a constant

Now by assumption we have  $-M \leq f'(x) \leq M$  on the interval so that

$$-M \leq E''(h) \leq M.$$

in particular it holds for  $x = a + h$



Integrating once on the interval  $[0, h]$  gives

$$-Mh \leq E'(h) - E'(0) \leq Mh \Rightarrow -Mh \leq E'(h) \leq Mh$$

since  $E'(0) = 0$ . Integrating again gives

$$-M \frac{h^2}{2} \leq E(h) - E(0) \leq M \frac{h^2}{2}$$

Because  $E(0) = 0$  we end up with

$$\text{Error} = |E(h)| \leq M \frac{h^2}{2}$$

as claimed in the theorem.  $\square$

In practice, we have no control over  $M$  which depends on the function  $f(x)$ . When we develop composite integration formulas in the next section we will see how to affect  $h$  to obtain better answers.

### The trapezoid rule

In many ways the endpoint rule is as bad as it gets. A very natural extension is to use a linear function going through both endpoints  $(a, f(a))$  and  $(b, f(b))$ , in other words the Lagrange interpolation polynomial through the points. Instead of a rectangle, we obtain a trapezoid which leads to the following definition.

**Definition 13.** The trapezoid rule is the approximation

$$\int_a^b f(x) dx \approx h \frac{f(a) + f(b)}{2}$$

where  $h = b - a$ .

It seems obvious that the trapezoid rule is better than either endpoint rule, and the following theorem cements the idea.

**Theorem 15.** Assume  $f$  is twice differentiable on  $[a, b]$  and let  $M$  be the maximum value for  $|f''(x)|$  there. The error in the trapezoid rule

$$\int_a^b f(x) dx \approx h \frac{f(a) + f(b)}{2}$$

is at most  $M \frac{h^3}{12}$  where  $h = b - a$ .

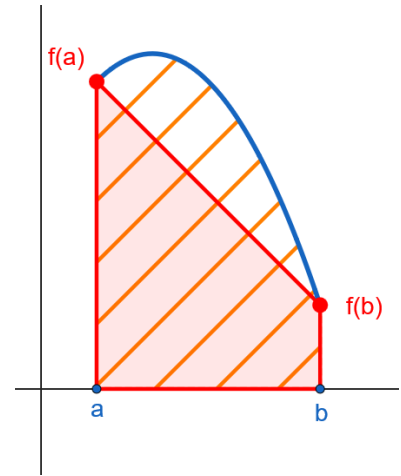
*Proof.* We follow the same idea as in Theorem 14. Again define

$$E(h) = \int_a^{a+h} f(x) dx - h \frac{f(a) + f(a+h)}{2}.$$

$\int_0^h E'' = E'(h) - E'(0)$  again by the fundamental theorem. The integral of  $M$  is clearly  $Mh$ .

As before. The integral of  $h$  is  $h^2/2$ .

We will use this double integration trick again.



**Figure 29:** Trapezoid rule on the interval  $[a, b]$ . The trapezoid has the same area as a rectangle with sides  $b - a$  and  $(f(a) + f(b))/2$ .

In practice  $h$  will be small so an error of order  $O(h^3)$  is a better outcome than order  $O(h^2)$ .

Clearly  $E(0) = 0$ . By the fundamental theorem of analysis, the derivative of the integral is  $f(a+h)$ . We use the product rule to differentiate the rest and obtain:

$$E'(h) = f(a+h) - \frac{f(a) + f(a+h)}{2} - h \frac{f'(a+h)}{2}.$$

Again we have  $E'(0) = 0$ . Using the product rule once again, we obtain

$$E''(h) = -\frac{h}{2} f''(a+h).$$

Now by assumption  $f''$  is between  $-M$  and  $M$  so that

$$-M \frac{h}{2} \leq E''(h) \leq M \frac{h}{2}.$$

We use the same double integration trick as in Theorem 14, using  $E(0) = E'(0) = 0$  along the way, first:

$$-M \frac{h^2}{4} \leq E'(h) \leq M \frac{h^2}{4}$$

and finally

$$-M \frac{h^3}{12} \leq E(h) \leq M \frac{h^3}{12}$$

and since the error is the absolute value of  $E(h)$ , we have proved Theorem 15.  $\square$

The trapezoid rule is often enough, however  $O(h^3)$  is not a tremendous improvement over the endpoint rule. It is easy to find functions for which both the endpoint rule and the trapezoid rule perform very badly, see Figure 30.

### Simpson's rule

The next logical step is to use three points instead of two to approximate the function. It is natural to use the midpoint of the interval

$$m = \frac{a+b}{2}$$

to do so. This is a typical example of three-point Lagrange interpolation: we construct a second-degree polynomial through  $(a, f(a))$ ,  $(m, f(m))$  and  $(b, f(b))$  and integrate it on  $[a, b]$ .

Now there is no simple way to figure out the approximate integral by geometrical means, so we must integrate the quadratic polynomial algebraically. According to Theorem 10 the formula for this interpolation polynomial is

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

$(uv)' = u'v + uv'$  with  $u = h$ ,  
 $v = \frac{f(a)+f(a+h)}{2}$ . Both understood  
as functions of  $h$ .

$$E'(0) = f(a) - \frac{f(a) + f(a)}{2} - 0 = 0.$$

First term:  $f'(a+h)$ .

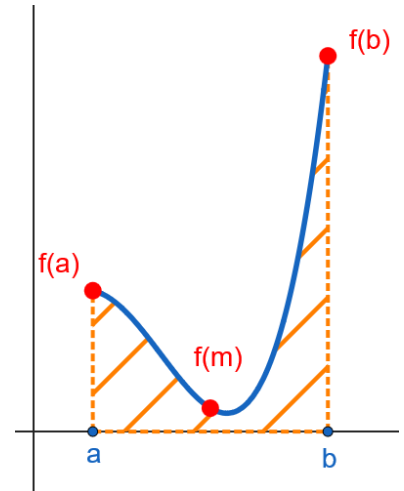
Second term:  $-\frac{f'(a+h)}{2}$ .

Third term:  $-\frac{f'(a+h)}{2} - \frac{h}{2} f''(a+h)$ .

The  $f'(a+h)$  terms cancel out.

Integral of  $h/2$  is  $h^2/4$ .

Integral of  $h^2/4$  is  $h^3/12$ .



**Figure 30:** A more complicated function  $f(x)$ . Note how endpoint and trapezoid rules would perform very badly here.

This can be simplified by using  $h = b - a$  and

$$\frac{h}{2} = m - a = b - m$$

to obtain

$$P(x) = \frac{2}{h^2} \left( f(a)(x-m)(x-b) - 2f(m)(x-a)(x-b) \cdots \right. \\ \left. \cdots + f(b)(x-a)(x-m) \right).$$

Now we integrate termwise and begin with:

$$\int_a^b (x-m)(x-b) dx.$$

The integral is simple enough to be calculated directly, however to obtain a simple form we perform the substitution  $u = x - m$  which changes the integral into:

$$\int_{-h/2}^{h/2} u(u-h/2) du.$$

This simplified form involves only  $h$  instead of three numbers  $a, b$  and  $m$ . Finally we integrate term by term:

$$\left[ \frac{u^3}{3} - \frac{h}{2} \frac{u^2}{2} \right]_{-h/2}^{h/2} = \frac{h^3}{24} - \frac{h}{2} \frac{h^2}{8} + \frac{h^3}{24} + \frac{h}{2} \frac{h^2}{8} = \frac{h^3}{12}.$$

Similarly we calculate

$$\int_a^b (x-a)(x-m) dx = \frac{h^3}{12}$$

and

$$\int_a^b (x-a)(x-b) dx = -\frac{h^3}{6}.$$

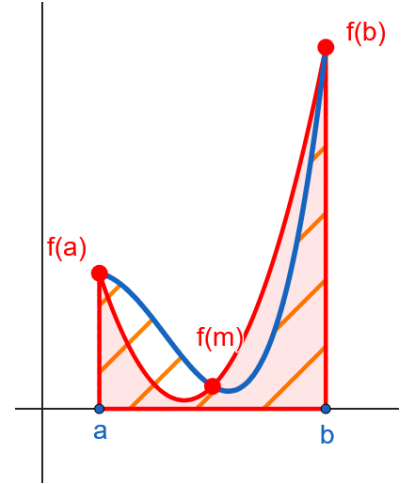
This finally shows that the integral of the interpolation polynomial  $P(x)$  is

$$\int_a^b P(x) dx = \frac{2}{h^2} \left( f(a) \frac{h^3}{12} - 2f(m) \frac{-h^3}{6} + f(b) \frac{h^3}{12} \right)$$

which simplifies to

$$\int_a^b P(x) dx = \frac{h}{6} (f(a) + 4f(m) + f(b))$$

This is Simpson's rule!



**Figure 31:** In red, the three-point interpolation polynomial  $P(x)$  through the three points  $(a, f(a))$ ,  $(m, f(m))$  and  $(b, f(b))$ . Simpson's rule replaces the integral of  $f(x)$  by the integral of  $P(x)$ .

The substitution uses three facts:

$$x - b = x - m + m - b = u - h/2$$

$$x = a \Rightarrow u = a - m = -h/2$$

$$x = b \Rightarrow u = b - m = h/2.$$

We may use the same substitution  $u = x - m$  in both cases.

**Definition 14.** *Simpson's rule is the approximation*

$$\int_a^b f(x) dx \approx \frac{h}{6} (f(a) + 4f(m) + f(b))$$

where  $h = b - a$  and  $m = \frac{a+b}{2}$  is the midpoint of  $[a, b]$ .

Some authors use  $2h = b - a$  instead which has a small effect on both Definition 14 and Theorem 16.

The error analysis of Simpson's rule is much more involved than Theorems 14 and 15. We admit the following result.

**Theorem 16.** *Assume  $f$  is four times differentiable on  $[a, b]$  and let  $M$  be the maximum value for  $|f''''(x)|$  there. The error in Simpson's rule*

$$\int_a^b f(x) dx \approx \frac{h}{6} (f(a) + 4f(m) + f(b))$$

is at most  $M \frac{h^5}{2880}$  where  $h = b - a$ .

mostly because the error is  $O(h^5)$  and therefore decays extremely fast if the interval is small enough. The 2880 factor helps as well.

This is a huge improvement on the trapezoid rule where the error was of order  $O(h^3)$ .

### Comparison of the three methods

We compare how these three numerical methods perform when estimating the integral

$$\int_1^2 (e^{-x} + 2x) dx.$$

Letting  $f(x) = e^{-x} + 2x$ , the exact value of the integral is easily computed:

$$\int_1^2 f(x) dx = \left[ -e^{-x} + x^2 \right]_{-1}^2 = 3 - e^{-2} + e^{-1} \approx 3.23254$$

1. The (left) endpoint rule is

$$\int_1^2 f(x) dx \approx hf(a) = f(1) \approx 2.36788$$

which is a very poor approximation.

2. The trapezoid rule is

$$\int_1^2 f(x) dx \approx h \frac{f(a) + f(b)}{2} = \frac{f(1) + f(2)}{2} \approx 3.25160$$

the error being approximately  $2 \times 10^{-2}$ .

In all cases  $a = 1$ ,  $b = 2$  and  $h = b - a = 1$ .

3. Simpson's rule is

$$\int_1^2 f(x) dx \approx \frac{1}{6} (f(1) + 4f(1.5) + f(2)) \approx 3.23262$$

the error being approximately  $8 \times 10^{-5}$ .

Let us check that the error in Simpson's rule obeys Theorem 16. The fourth derivative is

$$f^{(4)}(x) = e^{-x}$$

Its maximum value on the interval  $[1, 2]$  is then  $e^{-1}$ . Theorem 16 therefore guarantees

$$\text{Error} \leq e^{-1} \frac{h^5}{2880} = \frac{e^{-1}}{2880} \approx 1.3 \times 10^{-4}$$

so that the actual error is indeed (slightly) smaller than the theoretical worst case scenario. The benefits of the error analysis is that we do not need the exact value of the integral to perform it.

Simpson's rule performs very well, but ceases to do so on a larger interval. For instance on the interval  $[-1, 2]$  we have

$$\int_{-1}^2 f(x) dx \approx 5.58$$

while Simpson's rule is

$$\frac{3}{6} (f(-1) + 4f(0.5) + f(2)) \approx 5.64.$$

The error has grown to about one percent. This is unavoidable as now the theoretical maximum is

$$M \frac{h^5}{2880} = e \frac{3^5}{2880} \approx 0.23.$$

This suggests that even Simpson's rule is not good enough when the interval is too large. The solution is to use so-called composite methods that perform a simple approximation over and over on small subintervals.

## Composite methods

### Composite trapezoid method

Although Simpson's rule is the better method, we begin to expose the composite trapezoid method as the analysis is much simpler. When estimating the integral

$$\int_a^b f(x) dx$$

Midpoint of  $[1, 2]$  is  $m = 1.5$ .

Max error has to be  $Mh^5/2880$  where  $M$  is the maximum value of the fourth derivative of  $f$ .

$2x$  vanishes and the derivatives of  $e^{-x}$  alternate between  $-e^{-x}$  and  $e^{-x}$ .

$h = 1$  here

Actual error was  $8 \times 10^{-5}$ .

Same calculation as earlier:  
 $[-e^{-x} + x^2]_{-1}^2 = 3 - e^{-2} + e$ .

Midpoint of  $[-1, 2]$  is  $m = 0.5$  while  $h = 3$ .

We have  $h = 3$  and the maximum of the fourth derivative is now  $e^1$  at  $x = -1$ .

we define equally spaced points  $x_0, \dots, x_n$  (see Figure 32) such that

$$a = x_0 < x_1 < x_2 < \dots < x_{n-1} < x_n = b$$

and cut  $[a, b]$  into  $n$  intervals of length  $h = \frac{b-a}{n}$ . Under these conditions we can easily find formulas for the  $x_i$ :

$$x_i = a + ih, \quad 0 \leq i \leq n.$$

On each of the subintervals  $[x_i, x_{i+1}]$ , we then perform the simple trapezoid rule:

$$\int_{x_i}^{x_{i+1}} f(x) dx \approx h \frac{f(x_i) + f(x_{i+1})}{2}$$

to estimate the integral from  $x_i$  to  $x_{i+1}$ . The total integral is then clearly the sum of these local approximations, see Figure 33.

$$\int_a^b f(x) dx \approx \sum_{i=0}^{n-1} h \frac{f(x_i) + f(x_{i+1})}{2}.$$

We can simplify this sum by noting that every point appears twice, except  $x_0 = a$  and  $x_n = b$  which appear only once. This leads to the following definition.

**Definition 15.** Let  $f(x)$  be defined on  $[a, b]$ . We fix an integer  $n$  and divide  $[a, b]$  into  $n$  equally long subintervals as described above. Finally let  $h = \frac{b-a}{n}$ . The composite trapezoid rule with  $n$  subintervals and step size  $h$  is

$$\int_a^b f(x) dx \approx \frac{h}{2} \left( f(x_0) + 2 \sum_{i=1}^{n-1} f(x_i) + f(x_n) \right).$$

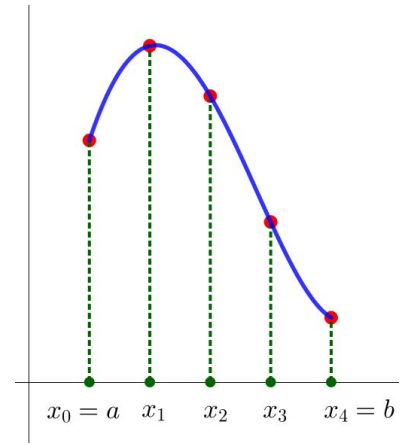
Now that  $h$  (or alternatively  $n$ ) is a parameter which we can change freely, it is important to perform an error analysis for the composite trapezoid method. The heavy lifting was done in Theorem 15.

**Theorem 17.** Assume  $f$  is twice differentiable on  $[a, b]$  and let  $M$  be the maximum value for  $|f''(x)|$  there. The error in the composite trapezoid rule on  $[a, b]$  with step size  $h$  is at most  $M(b-a)\frac{h^2}{12}$ .

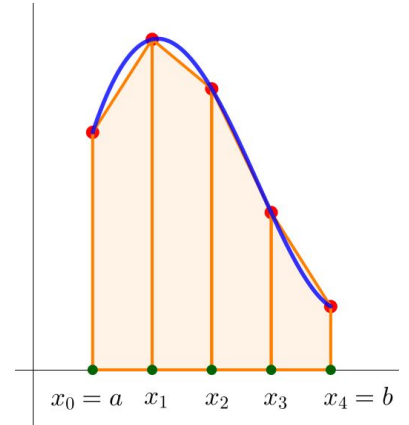
*Proof.* According to Theorem 15 the error in the simple trapezoid rule on  $[x_i, x_{i+1}]$  satisfies

$$\text{Error on } [x_i, x_{i+1}] \leq M \frac{h^3}{12}.$$

The total error is at most the sum of all local errors. Since there is a



**Figure 32:** A function  $f(x)$  on the interval  $[a, b]$ . The interval is cut into  $n = 4$  equally long subintervals.



**Figure 33:** Composite trapezoid rule performed on  $[a, b]$  with  $n = 4$  subintervals.

The local error for the single trapezoid rule on an interval of length  $h$  is  $O(h^3)$  but the global error for the composite method with step size  $h$  is  $O(h^2)$ .

This is not an optimal result and we can get rid of the extra  $b-a$  factor with a bit more care.

That's if all local errors have the same sign and pull in the same direction. There could be some cancelling effects.

total of  $n$  intervals we end up with

$$\text{Global error} \leq Mn \frac{h^3}{12} \leq M(b-a) \frac{h^2}{12}$$

since  $n = \frac{b-a}{h}$ .

□

This follows straight from  $h = (b-a)/n$ .

This is easily coded with a for loop. The program uses three variables:

- The interval  $[a, b]$  where the function is integrated.
- An integer  $n$ , the number of subintervals.

and returns the composite trapezoid approximation  $I$  to the integral.

```

1 function I=trapezoid(a,b,n)
2     h=(b-a)/n;
3     x=a;y=f(a);
4     for i=1:n-1
5         x=x+h;
6         y=y+2*f(x);
7     end
8     y=y+f(b);
9     I=h/2*y;
10 end

```

### Implementation of the composite trapezoid method

The duration of human pregnancies obeys a normal distribution with a mean of  $\mu = 270$  days and a standard deviation of  $\sigma = 17$  days. This means that the probability what the duration of a pregnancy lies in an interval  $[a, b]$  is calculated by the integral

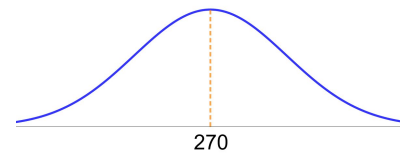
$$\frac{1}{\sigma\sqrt{\pi}} \int_a^b e^{-\left(\frac{x-\mu}{\sigma}\right)^2} dx.$$

The exact value of this integral is not computable. Let us use the composite trapezoid method the interval  $[270, 280]$  to calculate the probability that a pregnancy lasts between 270 and 280 days. We define first the function  $f$  that needs to be integrated:

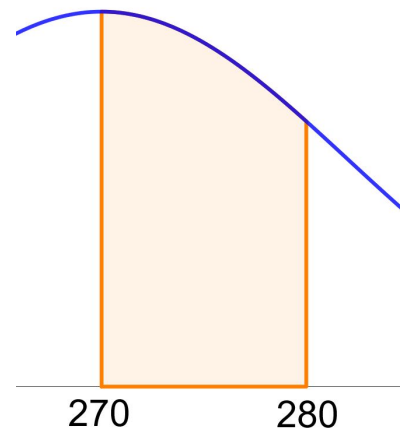
```

1 function y=f(x)
2     mu=270;sigma=17;
3     y=1/(sigma*sqrt(pi))*exp(-(x-mu)/sigma)^2);
4 end

```



**Figure 34:** Gaussian curve  $f(x) = \frac{1}{\sigma\sqrt{\pi}} e^{-\left(\frac{x-\mu}{\sigma}\right)^2}$  showing the probability distribution around the mean 270.



**Figure 35:** Probability interval  $[270, 280]$  added to Figure 34.

and call the trapezoid method, first for a small value of  $n$ .

```
1 a=270;b=280;n=10;
2 trapezoid(a,b,n)
```

This estimates the integral:

Command Window

```
ans =
fx    0.297129128436642
```

It turns out that this value is a rather good approximation. In order to display that the composite trapezoid method is a second-order method we repeat the calculations with various values of  $n$  (and therefore  $h$ ).

```
1 a=270;b=280;n=8;
2 for i=1:10
3     A(i)=trapezoid(a,b,n);
4     n=2*n;
5 end
```

Results are displayed in Table 7. It is clear that they converge to the correct value of the integral as  $n$  grows and  $h$  tends to 0. Is the speed of convergence compatible with Theorem 17, that is, is it second-order in  $h$ ? It is difficult to answer this question properly since Table 7 does not display any errors - we do not know the exact value beforehand! The following result is a little trick which allows us to check the order of a method only from approximate estimates.

**Theorem 18.** Assume a numerical method with step size  $h$  is of order  $k$  that is

$$\text{Error} = O(h^k)$$

as  $h \rightarrow 0$ . If  $y_h$  is the outcome for step size  $h$  then

$$\frac{y_{2h} - y_h}{y_h - y_{h/2}} \rightarrow 2^k$$

as  $h \rightarrow 0$ .

Using the data from Table 7 we compute the ratio

$$\frac{y_{2h} - y_h}{y_h - y_{h/2}}$$

Interpretation: 29.7% of pregnancies last between 270 and 280 days.

mostly due to the gentle nature of  $f(x)$  on the interval

meaning that the error is  $O(h^2)$

We run the trapezoid method on the same interval for  $n = 10, 20, 40, \dots, 1280$  and gather all results in a vector  $A$ .

$n$	$h$	Estimate $y_h$
10	1	0.29712913
20	1/2	0.29723072
40	1/4	0.29725611
80	1/8	0.29726246
160	1/16	0.29726405
320	1/32	0.29726444
640	1/64	0.29726454
1280	1/128	0.29726457

**Table 7:** Outcome of the composite trapezoid method for a range of values of  $n$ . Corresponding  $h = (b - a)/n$  is shown as well for  $a = 270$ ,  $b = 280$ .

Here  $y_h$  is the outcome of the trapezoid composite method for a particular value of  $h$ , see Table 7. In our case,  $k = 2$  so the error ratio should tend to  $4 = 2^2$ .



for  $h = 1/2, 1/4, \dots, 1/64$ . We use the following code:

```

1 for i=2:7
2     errorratio(i-1)=(A(i-1)-A(i))/(A(i)-A(i+1));
3 end

```

The error ratios are:

Command Window	
errorratio =	
Columns 1 through 6	
	4.000998970103989    4.000249607885332
	4.000062393357448    4.000015598111585
<i>fx</i>	4.000003898971983    4.000000953753926

which very clearly tends to  $4 = 2^2$  in agreement with Theorem 18 for  $k = 2$ , the order of the composite trapezoid method.

We finish this section by quickly proving Theorem 18.

*Proof.* Let  $y_{\text{real}}$  be the exact solution towards which the numerical method converges. The (signed) error for step size  $h$  is then

$$e_h = y_h - y_{\text{real}}.$$

Therefore

$$\frac{y_{2h} - y_h}{y_h - y_{h/2}} = \frac{e_{2h} + y_{\text{real}} - e_h - y_{\text{real}}}{e_h + y_{\text{real}} - e_{h/2} - y_{\text{real}}}.$$

The real values cancel out and we obtain

$$\text{Error ratio} = \frac{e_{2h} - e_h}{e_h - e_{h/2}}.$$

Now if the method is of order  $k$  we may expect

$$e_{2h} \approx 2^k e_h \quad e_{h/2} \approx 2^{-k} e_h.$$

Plugging these three formulas into the error ratio, we obtain

$$\text{Error ratio} \approx \frac{2^k e_h - e_h}{e_h - 2^{-k} e_h}.$$

We may cancel out  $e_h$  and are left with:

$$\text{Error ratio} \approx \frac{2^k - 1}{1 - 2^{-k}} = 2^k \frac{1 - 2^{-k}}{1 - 2^{-k}} = 2^k.$$

First and last value have to be excluded since either  $y_{2h}$  or  $y_{h/2}$  is missing.

$A$  contains the data from Table 7. If  $A(i)$  corresponds to  $h$  then  $A(i-1)$  corresponds to  $2h$  and  $A(i+1)$  to  $h/2$ .

This justifies calling this quantity the error ratio. The ratio of the estimates is the same as the ratio of the errors.

Definition is  $e_h = O(h^k)$  so  
 $e_{2h} = O((2h)^k) = 2^k O(h^k) = 2^k e_h$ .  
 Similar for  $e_{h/2}$ .

The latest fraction is 1. We use  $2^k - 1 = 2^k(1 - 1/2^k)$ .

□

### Composite Simpson's method

For fast changing functions, the trapezoid method may need large values of  $n$  to be accurate. This results in an increase in running time. Therefore we implement a composite Simpson's method based on the simple Simpson's rule (see Definition 14). Again, we divide the interval  $[a, b]$  into  $n$  subintervals of length  $h$ , see Figure 36. We now perform Simpson's rule on each subinterval  $[x_i, x_{i+1}]$ :

$$\int_{x_i}^{x_{i+1}} f(x) dx \simeq \frac{h}{6} (f(x_i) + 4f(m_i) + f(x_{i+1}))$$

with  $m_i = \frac{x_i + x_{i+1}}{2}$  being the midpoint of  $[x_i, x_{i+1}]$ . Adding the integrals on each of the  $n$  subintervals we obtain

$$\int_a^b f(x) dx \simeq \frac{h}{6} \sum_{i=1}^n (f(x_i) + 4f(m_i) + f(x_{i+1}))$$

The weights for each point are then as follows:

- $x_0 = a$  and  $x_n = b$  have weight 1.
- $x_1, \dots, x_{n-1}$  have weight 2.
- Midpoints  $m_i$  have weight 4, a total of  $n$  points.

This leads to the definition of the composite Simpson's method.

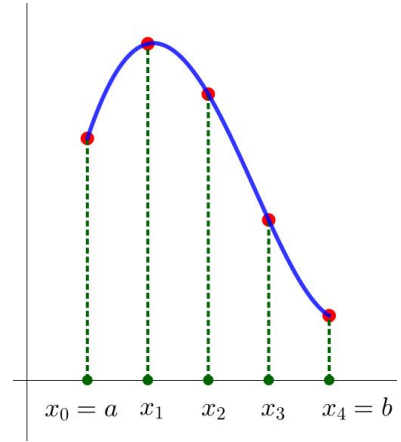
**Definition 16.** Let  $f(x)$  be defined on  $[a, b]$ . We fix an integer  $n$  and divide  $[a, b]$  into  $n$  equally long subintervals

$$[x_i, x_{i+1}], \quad 0 \leq i \leq n-1.$$

Finally let  $h = \frac{b-a}{n}$ . The composite Simpson's method with  $n$  subintervals and step size  $h$  is

$$\int_a^b f(x) dx \approx \frac{h}{6} \left( f(a) + 2 \sum_{i=1}^{n-1} f(x_i) + 4 \sum_{i=0}^{n-1} f(m_i) + f(b) \right)$$

where  $m_i = \frac{x_i + x_{i+1}}{2}$ .



**Figure 36:** A function  $f(x)$  on the interval  $[a, b]$ . The interval is cut into  $n = 4$  equally long subintervals.

All  $x_i$  appear twice except the first and the last. Midpoints appear just once.

Since the local error in Simpson's rule is  $O(h^5)$  (see Theorem 16) we may deduce that the global error is  $O(h^4)$ , meaning that the composite Simpson's method is a fourth-order numerical method. The composite Simpson's rule is coded exactly like the composite trapezoid rule, except we have to account for the midpoints as well.

See the proof of Theorem 17.

```

1 function I=simpsons(a,b,n)
2     h=(b-a)/n;
3     x=a;y=f(a);
4     for i=1:n-1
5         x=x+h;
6         y=y+2*f(x);
7     end
8     x=a-h/2;
9     for i=1:n
10        x=x+h;
11        y=y+4*f(x);
12    end
13    y=y+f(b);
14    I=h/6.*y;
15 end

```

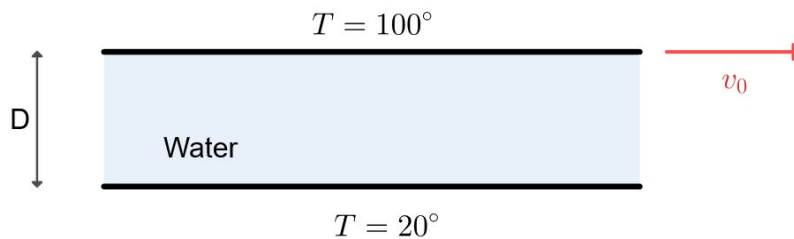
Lines 4-7 go through  $x_1, \dots, x_{n-1}$  and  
Lines 8-12 go through the midpoints  
 $m_0, \dots, m_{n-1}$ .

The dot on Line 14 is in case the inputs  
are vectors.

### Implementation

We consider two parallel plates, the lower one fixed and the upper moving with constant velocity  $v_0 = 0.1$  m/s. The distance between the plates is  $D = 0.1$  m. The lower plate is maintained a temperature of  $20^\circ$  and the top plate at  $100^\circ$ , see Figure 37.

The plates are much longer than the  
gap in between, and may be thought to  
be infinitely long.



**Figure 37:** A schematics of the two plates. The upper plate moves to the right with velocity  $v_0$ . Temperatures of both plates are kept constant as indicated.

The gap between the plates is filled by water, and the movement of the plates induces water flow. Our objective is to compute the velocity of water as a function of  $y \in [0, D]$ . The velocity is given by

$$v(y) = v_0 \frac{\int_0^y \frac{1}{\mu(s)} ds}{\int_0^D \frac{1}{\mu(s)} ds}$$

where  $\mu(s)$  is the viscosity of water at  $y = s$ . Because of the temperature gradient, the viscosity is not constant. Empirical data suggests that viscosity as a function of temperature is expressed by

$y = 0$  at the bottom plate.

The constants  $a$ ,  $b$  and  $c$  are actually obtained through a least squares analysis that fits viscosity measurements.

$$\mu(T) = e^{a+b/T+c/T^2}$$

with  $a = -8.944$ ,  $b = -839.456$ ,  $c = 421194.298$  and  $T$  in Kelvin. We may assume a simple temperature gradient:

$$T(y) = \left(293.16 + 80 \frac{y}{D}\right)$$

From the two formulas for viscosity and temperature we deduce the viscosity as a function of  $y$ . The formula for  $\mu$  can then be plugged into the integrals defining  $v(y)$ .

We compute the integrals using the composite Simpson's method. The first step is defining the function being integrated.

```
1 function func=f(y)
2 a=-8.944;b=-839.456;c=421194.298;D=0.1;
3 T=293.16+80*y/D;
4 mu=exp(a+b./T+c./T.^2);
5 func=1./mu;
6 end
```

We can then compute  $v(y)$  by integrating  $f$  on the intervals  $[0, y]$  and  $[0, D]$ . We let  $y$  be a range of values from 0 to  $D$ .

```
1 D=0.1;v0=0.1;
2 y=linspace(0,D,100);
3 n=100;
4 v=v0*simpsons(0,y,n)/simpsons(0,D,n);
```

Figure 37 suggests that  $y$  should be on the vertical axis, so we plot  $v$  on the  $x$ -axis, see Figure .

### Adaptive quadrature

So far we have only considered methods with a constant step size  $h$ . In all cases, the error can only be estimated (see for instance Theorem 17), we only know how it scales with  $h$ . In practice there is no knowing beforehand what value of  $h$  (or  $n$ ) is good enough for our purposes.

A second issue is that a constant step size makes no attempt at adapting to the function which is being integrated. For a function such as the one on Figure 39, we clearly can get away with using few intervals on the left-hand side of the interval.

Adaptive quadrature solves both problems at once. The objective is to calculate an approximation of the integral within a given error tolerance, while using as few subintervals as possible.

since the viscosity formula assumes Kelvin. Recall  $D$  is the distance between the plates. In Celsius it is  $T = (20 + 80y/D)$  and we see  $T(0) = 20$  and  $T(D) = 100$  as on Figure 37.

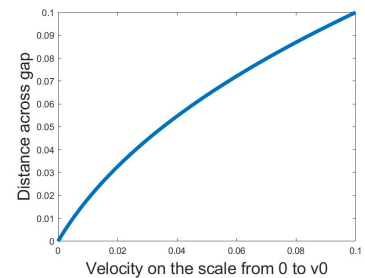
$$\mu(y) = e^{a+b/T+c/T^2} \text{ where } T = (293.16 + 80y/D).$$

It is the same function for both integrals, only the interval of integration varies.

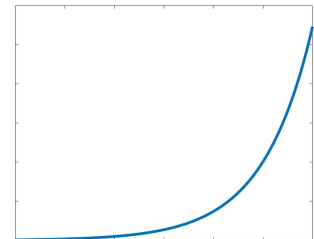
Highly recommended to define the function in steps, first going through  $T(y)$  and  $\mu(y)$ .

Note the dots since  $y$  and  $T$  will be vectors when plotting.

Line 3 defines the accuracy of Simpson's method, we use 100 subintervals. Line 4 computes first the integral from 0 to  $y$  via `simpsons(0,y,n)` and divides by the integral from 0 to  $D$ , which is also calculated through `simpsons`.



**Figure 38:** Velocity as a function of  $y$ . We display  $y$  on the same geometrical axis where it lies on Figure 37.



**Figure 39:** The function  $f(x) = e^x$  on the interval  $[-2, 4]$ .

A rough outline for adaptive quadrature is as follows. We are given a function  $f$  on  $[a, b]$  and an error tolerance  $\text{tol}$ .

1. Compute an approximation  $I_0 \approx \int_a^b f(x) dx$ .
2. Split the interval in two:  $[a, m]$  and  $[m, b]$  where  $m$  is the midpoint. Compute similar approximations to both  $I_1 \approx \int_a^m f(x) dx$  and  $I_2 \approx \int_m^b f(x) dx$ .
3. Compare  $I_0$  with  $I_1 + I_2$ .
4. If the error is acceptable within tolerance, return  $I_1 + I_2$  as a good enough approximation,
5. otherwise restart the procedure recursively on both  $[a, m]$  and  $[m, b]$ . The error tolerance is  $\text{tol}/2$  in each half so that the whole error is less than  $\text{tol}$ .

Step 4 seems impossible: how can we know from three approximations if the error is small enough? We have no clue what is the exact value of the integral is. The answer is through our error analysis of the numerical method used in Steps 1 and 2. If we settle on using the simple Simpson's rule then the error is of order  $Ch^5$  for a fixed constant  $C$  depending only the function  $f$ , see Theorem 16. Therefore Step 1 computes an approximation  $I_0$  such that

$$\int_a^b f(x) dx \approx I_0 + Ch^5.$$

Similarly Step 2 performs similar approximations  $I_1$  and  $I_2$ , however on intervals of length  $h/2$ . Therefore

$$\begin{aligned} \int_a^m f(x) dx &\approx I_1 + C \left(\frac{h}{2}\right)^5 \\ \int_m^b f(x) dx &\approx I_2 + C \left(\frac{h}{2}\right)^5 \end{aligned}$$

which added together give

$$\int_a^b f(x) dx \approx (I_1 + I_2) + C \frac{h^5}{16}.$$

We have found two approximations for  $\int_a^b f(x) dx$  with different error terms. The better approximation has error term  $Ch^5/16$ . Considering both equations at once we obtain

$$I_0 + Ch^5 \approx (I_1 + I_2) + C \frac{h^5}{16},$$

that is

for instance using simple Simpson's rule

using the same rule as in Step 1

$$\int_a^b f(x) dx \approx \frac{h}{6} (f(a) + 4f(m) + f(b))$$

with  $h = b - a$

$[a, m]$  and  $[m, b]$  are half as long as  $[a, b]$ .

since  $\int_a^m f(x) dx + \int_m^b f(x) dx = \int_a^b f(x) dx$ .  
We gather the terms in  $h$ .

Unsurprisingly the approximation  $I_1 + I_2$  is much better (smaller error term) as it uses a finer division into subintervals.

The error term in the better approximation is  $h^5/16$ .

$$I_0 - (I_1 + I_2) \approx \frac{15h^5}{16} \approx 15 \cdot \text{Error term}$$

From these calculations we obtain the very surprising fact that the error can be estimated straight from three approximations:

$$\text{Error term} \approx \frac{1}{15} (I_0 - (I_1 + I_2)).$$

In particular, if

$$|I_0 - (I_1 + I_2)| < 15 \cdot \text{tolerance}$$

then the error is small enough and we can be satisfied with our approximation. In practice, the number 15 might be too optimistic and the tradition is to use 10 rather than 15 to be conservative.

Using our five-step plan together with this estimate for the error, we are now ready to code adaptive quadrature. Inputs are the integration interval and an error tolerance. We need a subroutine to perform simple Simpson's rule.

```

1 function I=adapquad(a,b,tol)
2     I0=simple_simpson(a,b);
3     m=(a+b)/2;
4     I1=simple_simpson(a,m);
5     I2=simple_simpson(m,b);
6     if abs((I0-(I1+I2)))<10*tol
7         I=I1+I2;
8     else
9         tol=tol/2;
10        I=adapquad(a,m,tol)+adapquad(m,b,tol);
11    end
12 end

13
14 function I=simple_simpson(a,b)
15     h=b-a;m=(a+b)/2;
16     I=h/6*(f(a)+4*f(m)+f(b));
17 end

```

A slight modification of this code allows us to keep track of which intervals are being used. For instance we consider

$$\int_0^2 e^x \sin((x^2 \cos(e^x))) dx$$

(see Figure 40) with error tolerance  $10^{-4}$ .

We plot subintervals in alternating colors to visualize them properly. There is a total of nineteen subintervals for tolerance  $10^{-4}$  as

for instance because we completely get rid of higher terms in the approximations above

The number 15 (or 10) is entirely dependent on Simpson's rule. If we were to use the trapezoid method with error term given by Theorem 15, it would come out as  $|I_0 - (I_1 + I_2)| < 3 \cdot \text{tol}$ .

Lines 2-5 compute the three approximations.

If  $|I_0 - (I_1 + I_2)| < 10 \cdot \text{tol}$  then the error on  $[a, b]$  is less than the tolerance, and the answer is good enough. Line 7 returns the best approximation  $I_1 + I_2$ .

Otherwise recursively execute the method on  $[a, m]$  and  $[m, b]$  with half tolerance, so that the error on the whole is less than  $\text{tol}/2 + \text{tol}/2 = \text{tol}$ .

Simple Simpson's rule on the interval  $[a, b]$ .

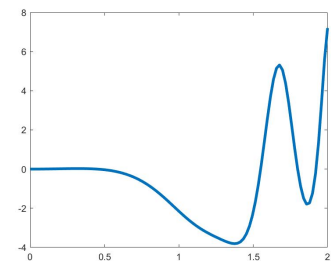
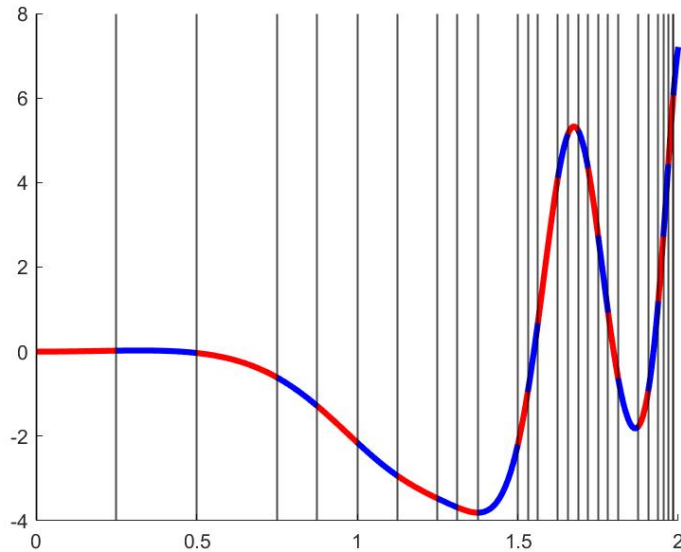


Figure 40: Graph of  $f(x) = e^x \sin(x^2 \cos(e^x))$  on  $[0, 2]$ .

displayed on Figure 41. As expected the program chooses very small intervals at places where the function changes quickly - while  $[0, 1]$  is covered by only five of the nineteen intervals. As a comparison,



**Figure 41:** Integration intervals when adaptive quadrature is performed on  $f(x) = e^x \sin(x^2 \cos(e^x))$  on  $[0, 2]$ . Error tolerance is  $10^{-4}$ . More intervals are used on  $[1.5, 2]$  where the function displays fast changes. Comparatively few intervals are used on  $[0, 1]$ .

would we be using composite Simpson's rule, we would need 53 equally large subintervals to obtain an answer within the same tolerance  $10^{-4}$ .

A very good approximation is computed by running adaptive quadrature for  $\text{tol} = 10^{-8}$ . We then run composite Simpson's rule for increasing values of  $n$  until the outcome is within  $10^{-4}$  of the good approximation.





# Initial value problems

The final three chapters are concerned with solving differential equations subject to some extra conditions. The simplest conditions are so called initial conditions, specifying the value of the function and possibly some of its derivatives at  $t = 0$ . Iterative methods can then be devised to solve the equation for  $t \in [0, T]$ . The same methods can be applied to differential systems as well.

## First degree initial value problems

We consider first degree differential equations together with one initial condition:

$$\begin{cases} y' = f(t, y) \\ y(0) = y_0 \end{cases}$$

where  $f$  is a function of two variables  $t$  and  $y$ , and  $y_0$  is the initial value at  $t = 0$ . We list a few examples of such equations:

- $y' = ky$ , with  $k \in \mathbb{R}$ . This is a simple model of growth ( $k > 0$ ) or decay ( $k < 0$ ) and serves as a good model for radioactive decay.
- $y' = ky(M - y)$  with  $k, M \in \mathbb{R}_+$ . This is called the logistic equation and describes population growth in a restrictive environment.

First degree equations are relatively simple and can often be solved exactly. Many real-world systems are however modelled by higher-order equations, or by differential systems. It turns out that the numerical methods we devise for first degree problems can easily be adapted to more complex situations.

The main theoretical insight is that initial value problems can be turned into an integral. Say we want to solve the differential equation for  $t \in [0, T]$ . We begin by introducing time values

$$0 = t_0 < t_1 < t_2 < \dots < t_n = T$$

Actually  $y = y(t)$  is a function of time. We omit  $t$  unless necessary.

$y_0$  is a given constant.

$$f(t, y) = ky$$

$f(t, y) = ky(M - y)$ . The solutions tend to the so-called carrying capacity  $M$  as  $t \rightarrow \infty$ .

The integral is then calculated using the numerical methods studied in the previous chapter.

with a step size of  $h = T/n$ . Solving the equation numerically means finding an approximate value for the solution at  $t = t_i$ ,  $1 \leq i \leq n$  which we denote by  $y_i$ , see the setup on Figure 42.

Starting with the differential equation

$$y'(t) = f(t, y(t)),$$

we integrate on the interval  $[t_i, t_{i+1}]$  to obtain

$$\int_{t_i}^{t_{i+1}} y'(t) dt = \int_{t_i}^{t_{i+1}} f(t, y(t)) dt.$$

By the fundamental theorem of analysis, we obtain

$$y(t_{i+1}) - y(t_i) = \int_{t_i}^{t_{i+1}} f(t, y(t)) dt$$

that is

$$y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} f(t, y(t)) dt.$$

The integral can be approximated in a variety of ways which we studied in the previous chapter. Each of them gives rise to an iterative scheme

$$y_{i+1} = y_i + \text{Approximation of the integral on } [t_i, t_{i+1}]$$

### Euler's method

Euler's method proceeds by approximation the integral  $\int_{t_i}^{t_{i+1}} f(t, y(t)) dt$  using the endpoint rule

$$\int_{t_i}^{t_{i+1}} f(t, y(t)) dt \approx hf(t_i, y(t_i)) \approx hf(t_i, y_i)$$

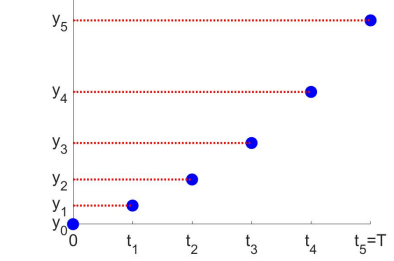
**Definition 17.** Let  $y_0 \in \mathbb{R}$  be given and consider the initial-value problem

$$\begin{cases} y' = f(t, y) \\ y(0) = y_0 \end{cases}$$

Euler's method with  $n$  time steps  $t_i$  and step size  $h = T/n$  numerically solves the initial value problem on  $[0, T]$  by iterating

$$y_{i+1} = y_i + hf(t_i, y_i), \quad 0 \leq i \leq n-1$$

In other words  $y(t_i) \approx y_i$ . Note that the



**Figure 42:** Interval  $[0, T]$  cut in  $n = 5$  intervals. Solving the equation numerically amounts to finding  $y_1 \approx y(t_1), \dots, y_5 \approx y(t_5)$ . Recall that  $y_0 = y(0)$  is given.

Now that we use approximations we use the approximated value  $y_i$  instead of the actual value  $y(t_i)$ .

In general  $\int_a^b f(x) dx \approx hf(a)$  where  $h = b - a$ . The length of the interval  $[t_i, t_{i+1}]$  is indeed the step size  $h$ . Second step is using  $y(t_i) \approx y_i$ .

The abstract implementation of Euler's method is a straightforward iteration. The function  $f(t, y)$  is defined as a subroutine. The Euler solver has three variables:

- the initial value  $y_0 = y(0)$ ,
- the final time value  $T$ ,
- the number of time steps  $n$ .

The outcome is a vector  $\mathbf{y} \in \mathbb{R}^{n+1}$ , containing  $y_0, y_1, \dots, y_n$ .

```

1 function y=eulersolver(y0,T,n)
2     y(1)=y0;h=T/n;t=0;
3     for i=1:n
4         t=t+h;
5         y(i+1)=y(i)+h*f(t,y(i));
6     end
7 end
8
9 function z=f(t,y)
10     z=...
11 end

```

The function  $f(t, y)$  is the variable part of the algorithm - it changes along with the equation we are solving.

Note that the first index of  $y$  is 1 rather than 0 since Matlab unfortunately cannot use the zero index. The last index is therefore  $n + 1$ , defined when  $i = n$ .

Line 4 updates  $t$  at each step.

Lines 9-11 define the function  $f(t, y)$  which depends on the particular example used.

### *Euler's method - simple implementation*

We study the logistic equation

$$y' = ky(M - y)$$

which describes growth in a limited environment. For instance we might consider a population of rabbits growing in an environment that can only sustain a population of  $M = 10000$ . The constant  $k$  describes the rate of growth while the population is small, we use  $k = 0.00005$ . Initially there are 20 rabbits. We update our Euler solver by specifying the function  $f$ . Since  $f$  is actually only a function of  $y$ , and not both  $t$  and  $y$  we could simplify this a bit. It is easiest to specify the values of  $k$  and  $M$  within the function  $f$ .

This particular function  $f(t, y)$  actually does not depend on  $t$ , only on  $y$ . This makes matters easier in some way but we can also just ignore it.

```

1 function y=eulersolver(y0,T,n)
2     y(1)=y0;h=T/n;t=0;
3     for i=1:n
4         t=t+h;
5         y(i+1)=y(i)+h*f(t,y(i));
6     end
7 end
8
9 function z=f(t,y)
10    M=10000;k=0.00005;
11    z=k*y*(M-y);
12 end

```

We solve the equation on a time frame  $t \in [0, 30]$ . The only free variable is the number of time steps  $n$ . Since Euler's method relies on a crude approximation (the endpoint rule) we should use a large value for  $n$ :

```

1 y0=20;T=30;n=1000;
2 y=eulersolver(y0,T,n);

```

which we plot against time values, see Figure 43:

```

1 t=linspace(0,T,n+1);
2 plot(t,y)

```

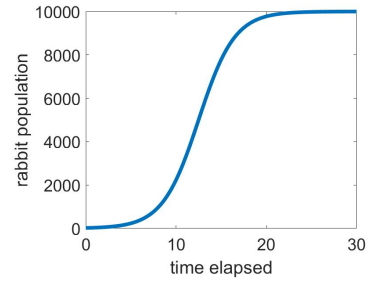
As intended the population grows first exponentially, before settling towards the carrying capacity  $M = 10000$ .

### *Euler's method - error analysis*

We now study how well Euler's method approximates the real solution to the initial value problem, and how the error scales with  $h$  (or  $n$ ). To do so we study one of the simplest initial value problems:

$$\begin{cases} y' = y \\ y(0) = 1 \end{cases}$$

whose solution is clearly  $y_{\text{real}} = e^t$ . We use Euler's method to numerically solve the equation for various values of  $n$  on the interval  $[0, 2]$  and compare the approximation at  $t = 2$  with the real value  $e^2$ . Results are gathered in Table 8. The two main insights from the table are as follows:



**Figure 43:** Logistic equation solved on  $[0, 30]$  by Euler's method with initial value  $y_0 = 20$  and 1000 time intervals.

Note that there are  $n + 1$  time values from  $t_0$  to  $t_n$ . The vector  $y$  also contains  $n + 1$  values.

$n$	$h$	Error at $t = 2$
10	1/5	1.19732
20	1/10	0.66156
40	1/20	0.34907
80	1/40	0.17949
160	1/80	0.09104
320	1/160	0.04585
640	1/320	0.02301
1280	1/640	0.01152
2560	1/1280	0.00577
5120	1/2560	0.00289

**Table 8:** For every value of  $n$ , Euler's method solves the initial value problem and the value at  $t = 2$  is collected. Third column shows deviation with exact value  $e^2$ , that is  $|y_n - e^2|$ . Second column is  $h = 2/n$ .

- Euler's method is very inaccurate. At low values of  $n$  we end up very far from the real value. Large values of  $n$  are necessary even to obtain accuracy within 0.1%, at the cost of higher running time. Euler's method performs even worse in more sensitive systems where errors tend to multiply.
- The error seems to scale linearly with  $h$  and  $n$ , in the sense that if  $h$  is halved, then so is the error.

The second point is made even clearer by plotting the error as a function of  $h$ : the data points from Table 8 seem to fall on a line, which suggests that the error scales linearly with  $h$ , see Figure 44.

However since most data points end up being near  $h = 0$ , it is difficult to draw clear conclusions. Figure 45 plots the logarithm of the error against the logarithm of  $h$  to better spread the points.

The result is a line whose slope is very close to 1. This suggests that

$$\log(\text{error at } t = T) \approx \log(h) + C$$

and taking the exponential of both sides yields

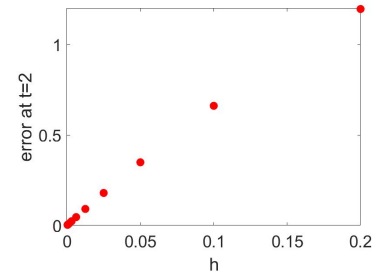
$$\text{error at } t = T \approx e^{\log(h)+C} = C'h$$

with  $C' = e^C$  another constant. This suggests that Euler's method is a numerical method of order 1, that is the error is  $O(h)$ . Order 1 is as bad as it gets. The following theorem is a more precise statement of that fact. We define the error at step  $i$  to be

$$e_i = |y(t_i) - y_i|$$

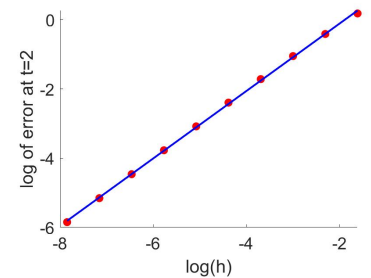
where  $y_i$  is the Euler approximation, and  $y(t_i)$  is the actual value of the real solution. The final error when  $t = T$  is therefore  $e_n$ , computed at step  $n$ .

Equivalently if  $n$  is doubled.



**Figure 44:** Error at  $t = 2$  as a function of  $h$ . Seemingly the error is a linear function of  $h$ .

The constant  $C$  is the intersect of the line on Figure 45.



**Figure 45:** Logarithm of error at  $t = 2$  as a function of  $\log(h)$ . The line has approximate slope 1 as can be determined by the polyfit command.

**Theorem 19.** Assume Euler's method is used to solve the initial value problem

$$\begin{cases} y' = f(t, y) \\ y(0) = y_0 \end{cases}$$

on the interval  $[0, T]$ , using  $n$  subintervals of step size  $h$ . Furthermore assume on  $f$  that:

$$\left| \frac{\partial f}{\partial y}(t, y) \right| \leq L$$

Then the error at step  $n$  satisfies

$$e_n \leq \frac{Mh}{L} (e^{LT} - 1)$$

for some constant  $M$ . In particular  $e_n = O(h)$  and Euler's method is a first order numerical method.

The formula shows that  $e_n$  is at most a constant times  $h$ . Moreover the error grows exponentially with the length  $T$  of the time interval.

Before proving the theorem it is worth recalling that  $y(t)$  is the exact solution to the boundary value problem and  $y_i$  is the Euler approximation at  $t = t_i$ .

*Proof.* We fix a step  $i$  in Euler's method. By definition 17, we have

$$y_{i+1} = y_i + hf(t_i, y_i).$$

The next step is to use Taylor approximation on the function  $y$  between  $t_i$  and  $t_{i+1}$ :

$$y(t_{i+1}) = y(t_i) + hy'(t_i) + Mh^2$$

for some constant  $M$ . Since  $y$  satisfies the differential equation we may replace  $y'$  by  $f(t, y)$  and obtain

$$y(t_{i+1}) = y(t_i) + hf(t_i, y(t_i)) + Mh^2.$$

Taking the difference between the Euler equation and the Taylor equation gives:

$$y(t_{i+1}) - y_{i+1} = y(t_i) - y_i + h(f(t_i, y(t_i)) - f(t_i, y_i)) + Mh^2.$$

Taking absolute values:

$$e_{i+1} \leq e_i + h|f(t_i, y(t_i)) - f(t_i, y_i)| + Mh^2.$$

The last step is to rewrite the quantity involving  $f$  using the mean value theorem:

$$f(t_i, y(t_i)) - f(t_i, y_i) = \frac{\partial f}{\partial y}(t_i, c_i) (y(t_i) - y_i)$$

In general:

$$y(t) = y(c) + y'(c)(x - c) + M(x - c)^2.$$

We use  $c = t_i$ ,  $x = t_{i+1}$  so that

$$x - c = t_{i+1} - t_i = h.$$

Recall that the error at step  $i$  is  $e_i = |y(t_i) - y_i|$  so the error at step  $i + 1$  is  $e_{i+1} = |y(t_{i+1}) - y_{i+1}|$ .

In general:

$g(b) - g(a) = g'(c)(b - a)$ . We use  $b = y(t_i)$ ,  $a = y_i$ , and  $g(y) = f(t_i, y)$  as a function of  $y$ . The derivative  $g'$  is therefore a partial derivative of  $f$ .

Most of the quantities here can be estimated and we obtain:

$$\left| f(t_i, y(t_i)) - f(t_i, y_i) \right| \leq L e_i$$

Altogether we obtain:

$$e_{i+1} \leq (1 + hL)e_i + Mh^2$$

Can we use the above stepwise estimate on the error to obtain a final bound on  $e_n$ ? The following technical lemma provides the answer.

**Technical lemma.** Let  $(e_i)_{0 \leq i \leq n}$  be a sequence such that  $e_0 = 0$  and  $e_{i+1} \leq (1 + a)e_i + b$  for some constants  $a$  and  $b$ . Then

$$e_n \leq \frac{b}{a} \left( (1 + a)^n - 1 \right)$$

We use the technical lemma on the error sequence  $e_i$ . First off, we have

$$e_0 = y(t_0) - y_0 = 0$$

since the approximation and the real solution agree at  $t = 0$ . We have  $a = hL$  and  $b = Mh^2$ . Therefore the lemma yields

$$e_n \leq \frac{Mh^2}{hL} \left( (1 + hL)^n - 1 \right)$$

Here we use the fact that  $(1 + x)^n \leq e^{nx}$  to obtain

$$e_n \leq \frac{Mh}{L} (e^{hnL} - 1)$$

but since  $h = T/n$  we end up with

$$e_n \leq \frac{Mh}{L} (e^{LT} - 1)$$

as claimed in the theorem.  $\square$

By assumption  $\left| \frac{\partial f}{\partial y} \right| \leq L$ . Moreover  $|y(t_i) - y_i| = e_i$  by definition of the error.

This formula shows us how the error at stage  $i + 1$  is related to the error at stage  $i$ .

This technical lemma is proved by induction on  $n$ .

This would be a good enough result however the function  $(1 + hL)^n$  is difficult to estimate for large  $n$ .

Enough to prove  $1 + x \leq e^x$  which can be done by studying the function  $f(x) = e^x - (1 + x)$  which attains its minimum 0 when  $x = 0$ .

### Heun's method

Euler's method is in most cases far too inaccurate to use. The inaccuracy stems from the fact that we solved the initial value problem in integral form

$$y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} f(t, y(t)) dt$$

using the worst possible approximation rule for an integral (the end-point rule). A logical step would be to use our best tool so far, Simpson's rule. A number of technical issues arise, and it proves useful to learn to tackle some of them by using the simpler trapezoid rule:

$$\int_{t_i}^{t_{i+1}} f(t, y(t)) dt \approx \frac{h}{2} \left( f(t_i, y_i) + f(t_{i+1}, y_{i+1}) \right)$$

It looks like we should iterate

$$y_{i+1} = y_i + \frac{h}{2} \left( f(t_i, y_i) + f(t_{i+1}, y_{i+1}) \right).$$

However this formula behaves completely differently than Euler's iteration

$$y_{i+1} = y_i + hf(t_i, y_i)$$

because now  $y_{i+1}$  is featured on both sides of the equation. We can't simply compute  $y_{i+1}$  knowing  $y_i$ . This problem can be solved in two ways:

- Considering

$$y_{i+1} = y_i + \frac{h}{2} \left( f(t_i, y_i) + f(t_{i+1}, y_{i+1}) \right)$$

as an equation for  $y_{i+1}$  which is solved by a numerical method from the first chapter, for instance the bisection method. We will not use this line of thinking, although it is absolutely valid.

- Replacing  $y_{i+1}$  by a simpler approximation, for instance the outcome of one step of Euler's method.

The implicit method using Euler's method as a stepping stone is called Heun's method.

**Definition 18.** Let  $y_0 \in \mathbb{R}$  be given and consider the initial-value problem

$$\begin{cases} y' = f(t, y) \\ y(0) = y_0 \end{cases}$$

Heun's method with  $n$  time steps  $t_i = ih$  and step size  $h = T/n$  numerically solves the initial value problem on  $[0, T]$  by iterating

1.  $w_{i+1} = y_i + hf(t_i, y_i)$
2.  $y_{i+1} = y_i + \frac{h}{2} \left( f(t_i, y_i) + f(t_{i+1}, w_{i+1}) \right)$

The implementation of Heun's method is a straightforward modification to the Euler solver.

In general:

$\int_a^b f(x) dx \approx \frac{h}{2} (f(a) + f(b))$  where  $h = b - a$ . The length of the interval  $[t_i, t_{i+1}]$  is the step size  $h$ .

This is an example of an implicit iteration, whereby  $y_{i+1}$  is not directly calculated from  $y_i$ .

The idea is that Euler's method delivers a crude approximation to  $y_{i+1}$ . We obtain an explicit iteration which only uses  $y_i$  to compute  $y_{i+1}$ .

$w_{i+1}$  is the Euler approximation starting with  $y_i$ . The Euler approximation then replaces  $y_{i+1}$  in the trapezoid rule.

We could easily combine steps 1 and 2 into a single step but readability is improved by using two steps.



```

1 function y=heunsolver(y0,T,n)
2     y(1)=y0;h=T/n;t=0;
3     for i=1:n
4         t=t+h;
5         w=y(i)+h*f(t,y(i));
6         y(i)=y(i)+h/2*(f(t,y(i))+f(t+h,w));
7     end
8 end
9
10 function z=f(t,y)
11     z=... %the function in our diff. eq.
12 end

```

We can now repeat the analysis of the error, using the initial value problem

$$\begin{cases} y' = y \\ y(0) = 1 \end{cases}$$

on  $[0, 2]$  as a benchmark. The real value at  $t = 2$  is  $e^2$ . Heun's method is ran for a range of values of  $h$ , and the final approximated value at  $t = 2$  is compared to  $e^2$ . Figures 46 and 47 show the deviation from the real value as a function of  $h$ , first in a normal plot, then in a log-log plot.

The data points on Figure 46 seem to lie on a parabola. However it is difficult to be convinced since they are concentrated near  $h = 0$ . Moreover, this curve could easily be of the form  $Ch^{2.1}$  rather than  $Ch^2$ . So we turn to Figure 47 which shows the logarithm of the error against the logarithm of  $h$ . We estimate the slope of the line for instance using the polyfit command in Matlab - it turns out to be almost equal to 2. Hence

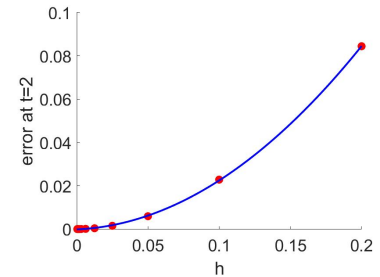
$$\log(\text{error at } t = T) \approx 2\log(h) + C$$

for some constant  $C$ . Taking exponents gives us

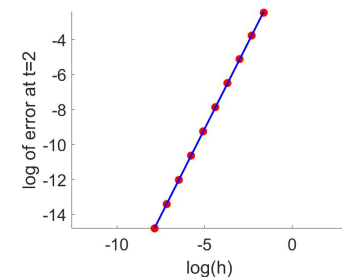
$$\text{error at } t = T \approx e^{2\log(h)+C} = C'h^2$$

which suggests strongly that Heun's method is of order 2.

There is no need to save the Euler approximations  $w_i$  from one step to the next. Pay attention to the changes in  $t$  in the two terms defining  $y(i)$ .



**Figure 46:** Error at  $t = 2$  as a function of  $h$ . The curve looks quadratic which suggests that the error is  $Ch^2$  for some constant  $C$ .



**Figure 47:** Logarithm of error at  $t = 2$  as a function of  $\log(h)$ . The line has approximate slope 2.

**Theorem 20.** Assume Heun's method is used to solve an initial value problem on an interval  $[0, T]$ , using  $n$  subintervals of step size  $h$ . Then the error at  $t = T$  satisfies

$$e_n \leq Ch^2$$

for some constant  $C$ . This means that Heun's method is a second order numerical method.

The proof is much more involved than that of Theorem 19.

### Runge-Kutta method (RK4)

In order to obtain a numerical method of higher order than 2 we use Simpson's rule in order to approximate the integral in

$$y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} f(t, y(t)) dt$$

In general, Simpson's rule is defined by

$$\int_a^b g(x) dx \approx \frac{h}{6} (g(a) + 4g(m) + g(b))$$

where  $m = \frac{a+b}{2}$  is the midpoint and  $h = b - a$ . On the interval  $[t_i, t_{i+1}]$  we obtain

$$\int_{t_i}^{t_{i+1}} f(t, y(t)) dt \approx \frac{h}{6} (f(t_i, y_i) + 4f(t_i + h/2, y_{i+1/2}) + f(t_{i+1}, y_{i+1}))$$

which leads us to want to iterate

$$y_{i+1} = y_i + \frac{h}{6} (f(t_i, y_i) + 4f(t_i + h/2, y_{i+1/2}) + f(t_{i+1}, y_{i+1}))$$

The situation is similar to Heun's method except we now need to estimate both  $y_{i+1/2}$  and  $y_{i+1}$ . The idea behind the so-called Runge-Kutta method is as follows. Assume we have computed  $y_i$  in the previous iteration.

- The first term  $k_1 = f(t_i, y_i)$  is immediately computable. Then  $y_i + h/2 \cdot k_1$  is a rough estimate for  $y_{i+1/2}$ .
- We therefore have

$$f(t_i + h/2, y_{i+1/2}) \approx f(t_i + h/2, y_i + h/2 \cdot k_1)$$

which we define as  $k_2$ .

- This estimate is corrected through a second pass:

$$k_3 = f(t_i + h/2, y_i + h/2 \cdot k_2)$$

Midpoint is  $t_i + h/2$  since the interval has length  $h$ . We temporarily use  $y_{i+1/2}$  to denote an approximate value for the solution there.

Euler approximation on a half interval of length  $h/2$  to obtain estimate at the midpoint.

$k_2$  and  $k_3$  are both used to evaluate the middle term, with equal weight. One overshoots and the other undershoots.

- A final Euler approximation on the whole interval

$$y_{i+1} \approx y_i + hk_3$$

which is used to estimate the final term

$$f(t_{i+1}, y_{i+1}) \approx f(t_{i+1}, y_i + hk_3)$$

which we denote by  $k_4$ .

We gather the necessary formulas in the following definition.

**Definition 19.** Let  $y_0 \in \mathbb{R}$  be given and consider the initial-value problem

$$\begin{cases} y' = f(t, y) \\ y(0) = y_0 \end{cases}$$

The classical Runge-Kutta method with  $n$  time steps  $t_i$  and step size  $h = T/n$  numerically solves the initial value problem on  $[0, T]$  by iterating

1.  $k_1 = f(t_i, y_i)$ .
2.  $k_2 = f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_1\right)$
3.  $k_3 = f\left(t_i + \frac{h}{2}, y_i + \frac{h}{2}k_2\right)$
4.  $k_4 = f(t_{i+1}, y_i + hk_3)$
5.  $y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$

This is one of many numerical methods called Runge-Kutta methods. However this the most used, sometimes called the classical fourth order Runge-Kutta method, or RK4 for short.

This five-step implementation is described here:

```

1 function y=RKsolver(y0,T,n)
2     y(1)=y0;h=T/n;t=0;
3     for i=1:n
4         t=t+h;
5         k1=f(t,y(i));
6         k2=f(t+h/2,y(i)+h/2*k1);
7         k3=f(t+h/2,y(i)+h/2*k2);
8         k4=f(t+h,y(i)+h*k3);
9         y(i+1)=y(i)+h/6*(k1+2*k2+2*k3+k4);
10    end
11 end
12
13 function z=f(t,y)
14     z=... %the function in our diff. eq.
15 end

```

Again we experiment with the initial value problem

$$\begin{cases} y' = y \\ y(0) = 1 \end{cases}$$

on  $[0, 2]$  to judge the accuracy of the method. Our experience with Heun's method leads us to directly consider a log-log plot of the error against  $h$ , see Figure 48.

Again we obtain a line whose slope is slightly less than 4. This suggests

$$\log(\text{error at } t = T) \approx 4 \log(h) + C$$

for some constant  $C$ . Taking exponents gives us

$$\text{error at } t = T \approx e^{4 \log(h) + C} = C' h^4$$

Indeed, Runge-Kutta method is a fourth order method, which is a tremendous improvement on the previous two attempts.

**Theorem 21.** Assume Runge-Kutta's method is used to solve an initial value problem on an interval  $[0, T]$ , using  $n$  subintervals of step size  $h$ . Then the error at  $t = T$  satisfies

$$e_n \leq Ch^4$$

for some constant  $C$ . This means that Runge-Kutta method is a fourth order numerical method.

We admit this result.

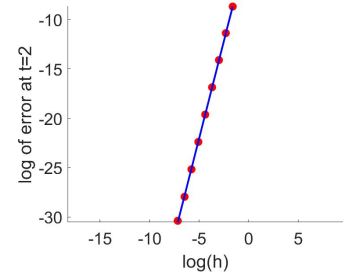
## Differential systems

So far, we have only considered first degree differential equations, and very few real-life examples are of that type. Our first expansion is into the realm of first degree differential systems. Our guiding thread will be the Lotka-Volterra equations used to study the dynamics of a biological system of two species, predators and preys. If we let  $y_1(t)$  be the population of preys as a function of time, and  $y_2(t)$  be the population of predators, the equations are as follows:

$$\begin{cases} y_1' = \alpha y_1 - \beta y_1 y_2 \\ y_2' = \gamma y_1 y_2 - \delta y_2 \end{cases}$$

where  $\alpha, \beta, \gamma$  and  $\delta$  are positive constants. Solving the system requires knowledge of the initial populations

$$y_1(0) = a \quad y_2(0) = b.$$



**Figure 48:** Logarithm of error at  $t = 2$  using Runge-Kutta as a function of  $\log(h)$ . The line has approximate slope 4.

Also known as the predator-prey equations.

Interpretation of the constants:

- $\alpha$ : prey (exponential) growth rate in the absence of predators.
- $\beta$ : predation rate.
- $\gamma$ : predator growth rate in the presence of preys.
- $\delta$ : predator dying rate.

Whether the change is positive or negative is determined by the signs in the equations.

In order to implement any of the previous section's methods it is necessary to convert the system in vector form. To do so define

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

so that the system might be written as

$$\mathbf{y}' = \begin{pmatrix} \alpha \mathbf{y}(1) - \beta \mathbf{y}(1) \mathbf{y}(2) \\ \gamma \mathbf{y}(1) \mathbf{y}(2) - \delta \mathbf{y}(2) \end{pmatrix}$$

together with initial conditions

$$\mathbf{y}_0 = \begin{pmatrix} a \\ b \end{pmatrix}.$$

The Runge-Kutta method requires very little changes in order to proceed. We only need to accommodate the fact that  $\mathbf{y}$  is a vector and not a number:

```

1  function y=RKsolverLotkaVolterra(y0,T,n)
2      y(1,:)=y0;h=T/n;
3      for i=1:n
4          k1=f(y(i,:));
5          k2=f(y(i,:)+h/2*k1);
6          k3=f(y(i,:)+h/2*k2);
7          k4=f(y(i,:)+h*k3);
8          y(i+1,:)=y(i,:)+h/6*(k1+2*k2+2*k3+k4);
9      end
10 end
11
12 function z=f(y)
13     alpha=1.1;beta=0.4;gamma=0.1;delta=0.4;
14     z(1)=alpha*y(1)-beta*y(1)*y(2);
15     z(2)=gamma*y(1)*y(2)-delta*y(2);
16 end

```

since  $y_1 = \mathbf{y}(1)$  and  $y_2 = \mathbf{y}(2)$ . Note that the equations are not linear so we cannot write the system as  $\mathbf{x}' = A\mathbf{x}$  where  $A$  is a fixed matrix.

The program solves the Lotka-Volterra equations on the interval  $[0, T]$  with  $n$  steps. Initial condition is specified by the vector  $\mathbf{y}_0$ . The values of the constants need to be specified in the subroutine  $f$  - they could also be input variables - especially useful if their values is not fixed.

Since the equations are time-independent we may omit  $t$  everywhere and simplify the equations.

The outcome of the program is a vector  $\mathbf{x}$  with two columns (one for each population) and  $n + 1$  rows (since we have  $n$  steps).

We can now plot the data as follows:

```

1 y0=[10;10];T=60;n=2000;
2 y=RKsolverLotkaVolterra(y0,T,n);
3
4 figure
5 t=linspace(0,T,n+1);
6 hold on
7 plot(t,y(:,1))
8 plot(t,y(:,2))

```

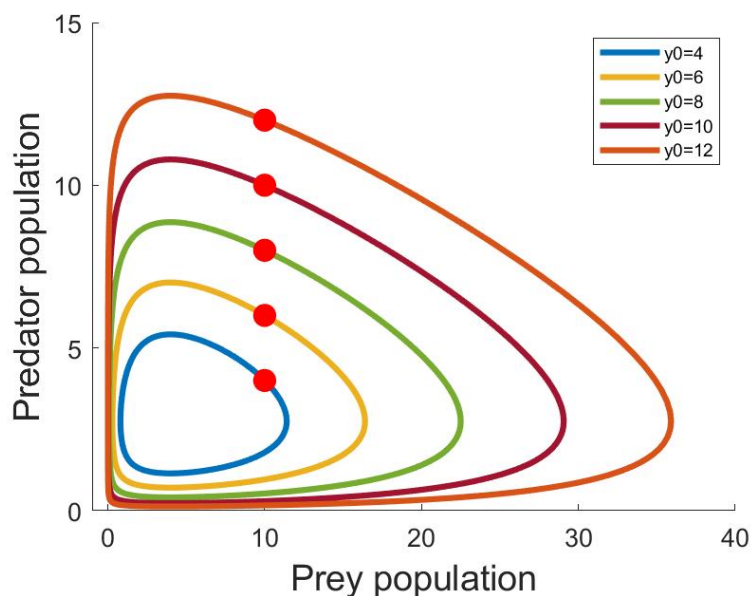
Figure 49 suggests strongly that the solutions are periodic: a cycle of growth and decline. This can be checked further by plotting a phase-space plot: plotting the prey population on the x-axis and the predator population on the y-axis to better see the effect they have on one another.

```

1 y0=[10;10];T=60;n=2000;
2 y=RKsolverLotkaVolterra(x0,T,n);
3 plot(y(:,1),y(:,2))

```

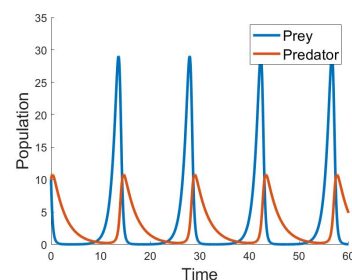
The result is shown on Figure 50. Finally we show a range of initial values for the predator population on Figure 51 while keeping the initial prey population at 10.



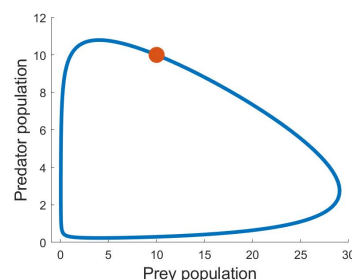
**Figure 51:** Phase-space plot for initial values  $a = 10$  (preys),  $b = 4, 6, 8, 10, 12$  (predators). Initial values are marked in red.

Initial conditions  $a = b = 10$  are specified. We solve on the interval  $[0, 60]$  with 2000 time steps. Note that computations and plotting are completely separated for better efficiency.

The first column of  $y$  contains prey population, that is  $y(:,1)$ . The other contains predator population. Note that we have  $n + 1$  data points, which the time vector  $t$  should reflect.



**Figure 49:** Outcome of the Lotka-Volterra solver on the interval  $[0, 60]$ .



**Figure 50:** Outcome of the Lotka-Volterra solver with prey population on the x-axis and predator population on the y-axis. Initial value  $(10, 10)$  is marked in red. Movement along the curve is counterclockwise - Figure 49 shows that the prey population drops at the start.

### Higher order equations

Second-degree equations and systems are ubiquitous in dynamics. However our numerical methods only handle first degree equations. The trick is to turn one second degree equation into an artificial first degree system. Our guiding thread will be the Van der Pol oscillator equation

$$x'' - \mu(1 - x^2)x' + x = 0$$

which is used to model oscillations with nonlinear damping. The constant  $\mu$  indicates the strength of the damping. We define the vector

$$\mathbf{y} = \begin{pmatrix} x \\ x' \end{pmatrix}.$$

From the Van der Pol equation we obtain

$$\mathbf{y}' = \begin{pmatrix} x' \\ x'' \end{pmatrix} = \begin{pmatrix} x' \\ \mu(1 - x^2)x' - x \end{pmatrix}$$

which is none other than a first degree system for the vector  $\mathbf{y}$ :

$$\mathbf{y}' = \begin{pmatrix} y(2) \\ \mu(1 - y(1)^2)y(2) - y(1) \end{pmatrix}.$$

This in turn can be solved by the Runge-Kutta method as any other differential system.

```

1 function y=RKsolverVanderPol(y0,T,n)
2   y(1,:)=y0;h=T/n;
3   for i=1:n
4     k1=f(y(i,:));
5     k2=f(y(i,:)+h/2*k1);
6     k3=f(y(i,:)+h/2*k2);
7     k4=f(y(i,:)+h*k3);
8     y(i+1,:)=y(i,:)+h/6*(k1+2*k2+2*k3+k4);
9   end
10 end
11
12 function z=f(y)
13   mu=2;
14   z(1)=y(2);
15   z(2)=mu*(1-y(1)^2)*y(2)-y(1);
16 end

```

We simulate the oscillations with initial conditions

$$x(0) = 1 \quad x'(0) = 0$$

Partly because Newton's laws contain the second derivative (acceleration).

It is a model for a mass-spring-damper system, see Figure 52, in situations where energy enters the system at small oscillations and exits the system when the oscillations grow larger. It appears in many fields such as electrical circuits, seismology and the study of neurons. Note that if  $|x|$  grows larger than 1 then the term  $\mu(1 - x^2)$  is negative and works against the movement.

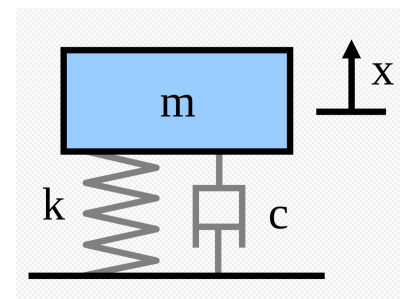
Note that  $x'' = \mu(1 - x^2)x' - x$ .

By definition  $y(1) = x$ ,  $y(2) = x'$ .

Only the function  $f$  differs from earlier.

Line 14 is essentially saying that  $x' = x'$  but is necessary in order to change the whole vector. Line 15 contains the differential equation.

The value of  $\mu$  determines the damping strength.



**Figure 52:** A mass-spring-damper system.

corresponding to an initial positive (upwards on Figure 52) deviation and no initial velocity of the mass. In vector terms we have

$$\mathbf{y}(0) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

We therefore run the code

```
1 y0=[1;0];T=30;n=5000;
2 y=RKsolverVanderPol(y0,T,n);
```

The vector  $\mathbf{x}$  contains both the position and velocity of the mass. It is certainly enough to plot just the position as a function of time:

```
1 plot(linspace(0,T,n+1),y(:,1))
```

The value of  $T = 30$  is enough to show a few oscillations, see Figure 53.

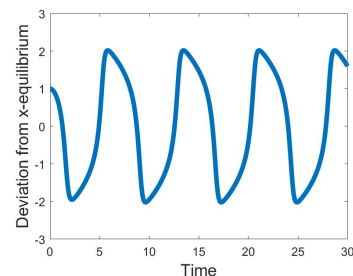
The value for  $n$  which determines the accuracy was chosen rather arbitrarily. It is important to choose a somewhat optimal value which delivers acceptable answers while not taking too long to compute. Since the actual exact solution is unknown, we use a large value for  $n$  as a benchmark and save the last value of the vector  $\mathbf{x}$ . This last value corresponds to the time  $t = 30$ .

```
1 y0=[1;0];T=30;n=50000;
2 yaccurate=RKsolverVanderPol(y0,T,n);
3 lastyaccurate=xaccurate(end,:)
```

We then run the same code for different values of  $n$  and compare the last value of  $\mathbf{y}$  to the most accurate estimate. Since  $\mathbf{y}$  is a vector, we must use a vector norm to compare.

```
1 n=100;
2 for i=1:9
3     nvector(i)=n;
4     y=RKsolverVanderPol(y0,T,n);
5     lasty=y(end,:);
6     error(i)=norm(lasty-lastyaccurate);
7     n=2*n;
8 end
```

The code's output is a error vector showing deviation from our best computed value as a function of  $n$ , see Table 9. Such a table is a useful guide to choose a value of  $n$  which is good enough for our



**Figure 53:** Oscillations of the mass-spring-damper system with damping  $\mu = 2$  and initial conditions  $x(0) = 1$ ,  $x'(0) = 0$ .

This takes about a second to run, which is rather too much.

The code solves the system numerically for  $n = 100, 200, 400, \dots, 25600$  and compares the value at  $t = 30$  to the most accurate estimate.

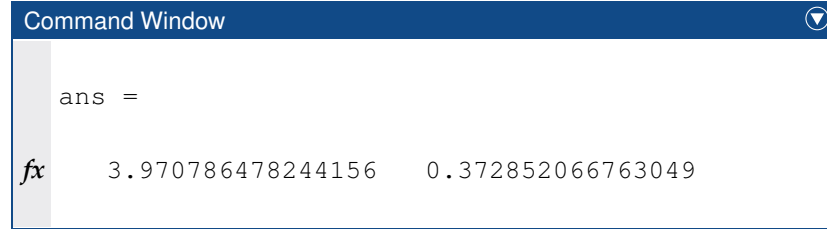


purposes. Note that when  $n$  is doubled, the error is approximately divided by  $16 = 2^4$ , which is consistent with Runge-Kutta being a fourth-order method.

This is confirmed by a log-log-plot of  $h = T/n$  against the error, showing the expected slope of 4, see Figure 54.

```
1 hvector=T/nvector;
2 plot(log(hvector),log(error))
3 polyfit(log(hvector),log(error),1)
```

The result of polyfit is as follows:



very close to the theoretical value of 4.

### Higher order systems

A similar trick can be performed to turn for instance a second-degree system for three variables into a first-degree system for six variables, upon which the Runge-Kutta method can be applied. For instance consider the system

$$\begin{cases} x_1'' &= x_1'x_2 - x_2'x_3 \\ x_2'' &= x_2^2 - x_3^2 \\ x_3'' &= x_1' - x_2 \end{cases}$$

for the three variables  $x_1$ ,  $x_2$  and  $x_3$ . Defining the vector

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_1' \\ x_2 \\ x_2' \\ x_3 \\ x_3' \end{pmatrix} \in \mathbb{R}^6$$

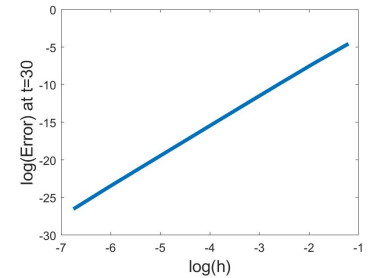
we have

$$\mathbf{x}' = \begin{pmatrix} x_1' \\ x_1'' \\ x_2' \\ x_2'' \\ x_3' \\ x_3'' \end{pmatrix} = \begin{pmatrix} x_1' \\ x_1'x_2 - x_2'x_3 \\ x_2' \\ x_2^2 - x_3^2 \\ x_3' \\ x_1' - x_2 \end{pmatrix}$$

$n$	Error
100	$1.1 \times 10^{-2}$
200	$7.9 \times 10^{-4}$
400	$5.3 \times 10^{-5}$
800	$3.4 \times 10^{-6}$
1600	$2.1 \times 10^{-7}$
3200	$1.3 \times 10^{-8}$
6400	$8.3 \times 10^{-10}$
12800	$5.2 \times 10^{-11}$
25600	$3.0 \times 10^{-12}$

**Table 9:** Error at  $t = 30$  as a function of  $n$  when the Van der Pol equation is solved numerically with  $n$  steps.

Recall that Theorem 21 states that  $\text{Error} = Ch^4$  which in log terms translates to  $\log(\text{Error}) = 4\log(h) + C'$  for some constant  $C'$ .



**Figure 54:** log-log plot of the error at  $t = 30$  against  $h = 30/n$ .

which can be coded as

```

1  function y=f(x)
2      y(1)=x(2);
3      y(2)=x(2)*x(3)-x(4)*x(5);
4      y(3)=x(4);
5      y(4)=x(3)^2-x(5)^2;
6      y(5)=x(6);
7      y(6)=x(2)-x(3);
8  end

```

By definition of  $\mathbf{x}$ , we have  $x_1 = \mathbf{x}(1)$ ,  $x'_1 = \mathbf{x}(2)$  and so on and so forth.

We can then use the same general Runge-Kutta solver as in the previous two examples.

## Two-point boundary value problems

The main objective of this chapter is to solve second degree differential equations on an interval  $[0, L]$  together with boundary conditions at the endpoints 0 and  $L$ . The iterative methods of the previous chapter are not suited to this problem, and our strategy will rather be to replace derivatives by a discrete approximation, and turn the problem into a large algebraic system.

Example of boundary conditions are so-called Dirichlet conditions  $y(0) = y(L) = 0$ .

### Numerical differentiation

In this section we develop formulas for evaluating derivatives of a function  $f$  at a given point  $x$ . In particular we are not concerned with symbolic differentiation: our main application will be to substitute derivatives of a unknown function in a differential equation.

#### First derivative

Our simplest problem is the following: given a function  $f$  on an interval  $I$  and  $x \in I$ , how can we approximate  $f'(x)$ , keeping track of the error? A natural starting point is the definition of the derivative as a limit

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}.$$

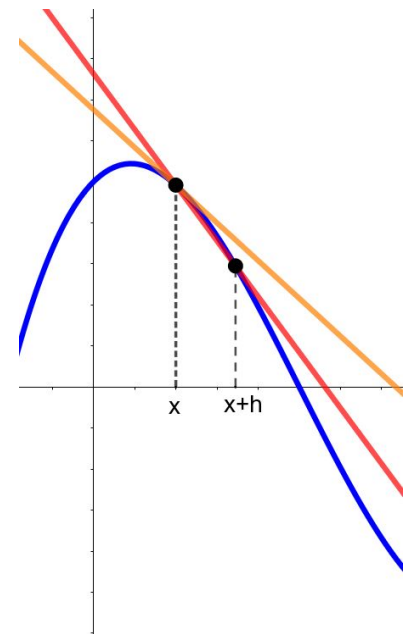
which leads to a straightforward first attempt.

**Definition 20.** The (one-point) forward difference approximation of  $f'$  at  $x$  is

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

where  $h > 0$ .

The graphical interpretation is shown on Figure 55. Our goal  $f'(x)$  is the slope of the tangent, while our approximation is the slope of the red line.



**Figure 55:** In blue the graph of  $f(x)$ , in orange the tangent at  $x$  whose slope is  $f'(x)$ . The red line goes through  $(x, f(x))$  and  $(x+h, f(x+h))$  and its slope is the formula of Definition 20.

The forward difference approximation will only improve as  $h$  decreases, but how quickly? The situation is similar to that of numerical integration whereby we have a free parameter we can adjust to improve the accuracy. It turns out that the forward difference approximation is a first order method which is poor.

**Theorem 22.** *The error in the forward difference approximation*

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

*is of order  $O(h)$ .*

As usual the error is the difference between the approximation and the real value. In essence the theorem says that

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h).$$

*Proof.* We write down Taylor's formula between  $x$  and  $x+h$  and obtain:

$$f(x+h) = f(x) + hf'(x) + O(h^2).$$

Shuffling terms gives us

$$hf'(x) = f(x+h) - f(x) + O(h^2)$$

and dividing by  $h$  we obtain

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h)$$

as claimed by the theorem.  $\square$

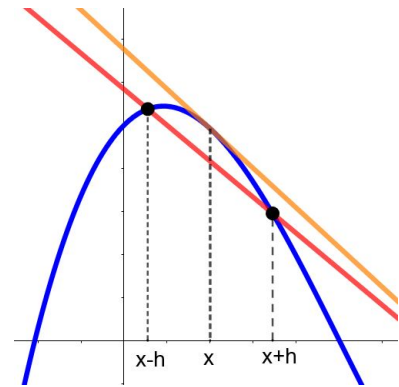
In order to improve we will need to consider the behaviour of  $f$  on both sides of  $x$ , see Figure 56. The red line goes through  $(x-h, f(x-h))$  and  $(x+h, f(x+h))$  so its slope is easily calculated. Graphically we see a tremendous improvement upon Figure 55 as the two lines are almost parallel.

**Definition 21.** *The (two-point) central difference approximation of  $f'$  at  $x$  is*

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

*where  $h > 0$ .*

As expected this approximation performs better and is a second order method.



**Figure 56:** In blue the graph of  $f(x)$ , in orange the tangent at  $x$  whose slope is  $f'(x)$ . The red line goes through  $(x-h, f(x-h))$  and  $(x+h, f(x+h))$  and its slope is the formula of Definition 21.

**Theorem 23.** *The error in the central difference approximation*

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

*is of order  $O(h^2)$ .*

*Proof.* We again begin by writing down Taylor's formula, this time both at  $x+h$  and  $x-h$ . We need to go one step further in the Taylor approximation in order to properly evaluate the error term.

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + O(h^3),$$

$$f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) + O(h^3).$$

Take the difference of the equations to obtain:

$$f(x+h) - f(x-h) = 2hf'(x) + O(h^3)$$

which can be reshuffled to

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

proving that the central difference approximation is a second order method.  $\square$

We test the central difference approximation on the known function  $f(x) = \ln(x)$  at  $x = 1$ . Since  $f'(x) = 1/x$  we clearly have  $f'(1) = 1$  as our exact value. For a range of values for  $h$  we compute the approximation

$$f'(1) \approx \frac{f(1+h) - f(1-h)}{2h}$$

and compute the error

$$\left| \frac{f(1+h) - f(1-h)}{2h} - f'(1) \right|.$$

The results are gathered in Table 10 and shown in a log-log plot on Figure 57. Our two takeaways are as follows:

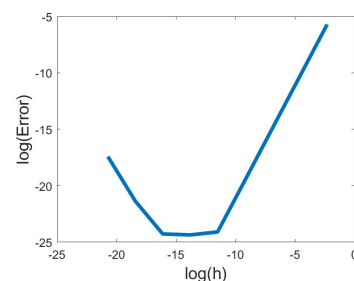
- For large values of  $h$  (right end of the graph) the error behaves according to Theorem 23. If  $h$  is divided by 10 then the error is divided by  $10^2$ . This is the expected behaviour of a second order method. In the log-log plot this is shown by a straight line of slope 2.
- For small values of  $h$  (left end of the graph), the order breaks down completely and the error starts increasing again. Truncations

The formulas are exactly the same except  $-h$  is used instead of  $+h$ . The negative sign cancel out when squared.

If we had stopped at  $O(h^2)$  we might have mistakenly thought that the method was first order. It was necessary to go one step further to take advantage of the fact that the  $h^2$  terms cancel out.

$h$	Error
$10^{-1}$	$3.3 \times 10^{-3}$
$10^{-2}$	$3.3 \times 10^{-5}$
$10^{-3}$	$3.3 \times 10^{-7}$
$10^{-4}$	$3.3 \times 10^{-9}$
$10^{-5}$	$3.4 \times 10^{-11}$
$10^{-6}$	$2.6 \times 10^{-11}$
$10^{-7}$	$2.8 \times 10^{-11}$
$10^{-8}$	$5.2 \times 10^{-10}$
$10^{-9}$	$2.8 \times 10^{-8}$

**Table 10:** Error in the central difference approximation for  $f(x) = \ln(x)$  at  $x = 1$  for various values for  $h$ .



**Figure 57:** Log of error in the central difference approximation for  $f(x) = \ln(x)$  at  $x = 1$  as a function of  $\ln(h)$ .

errors start to dominate and the accuracy deteriorates rapidly. This is due to the fact that both the numerator  $f(1+h) - f(1-h)$  and the denominator  $2h$  are very small numbers and Matlab cannot compute precisely anymore.

The optimal value of  $h$  seems to be around  $10^{-6}$ . The exact value obviously depends on the choice of the function of  $f$  but  $10^{-6}$  is a good rule of thumb in general.

It can be proven that the two-point central difference approximation of Definition 21 is the best approximation using just two points. Higher accuracy can only be attained by using a larger (even) number of points. For instance the formula

$$f'(x) \approx \frac{-f(x+2h) + 8f(x+h) - 8f(x-h) + f(x-2h)}{12h}$$

is a fourth-order central difference approximation. For our purposes of solving differential equations these more precise formulas prove too cumbersome. Table 10 shows that a two point formula is already very accurate.

The central approximations which we have studied so far cannot be used at the endpoint of the domain of the function since either  $f(x+h)$  or  $f(x-h)$  will be undefined. We cannot rely on the simple forward difference approximation

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

which is too inaccurate. A better formula is the following:

**Definition 22.** The three-point forward difference approximation of  $f'$  at  $x$  is

$$f'(x) \approx \frac{-3f(x) + 4f(x+h) - f(x+2h)}{2h}.$$

where  $h > 0$ . Similarly the three-point backward difference approximation is

$$f'(x) \approx \frac{3f(x) - 4f(x-h) + f(x-2h)}{2h}.$$

Both formulas are second-order methods, that is the order is  $O(h^2)$ .

We prove the statement about the order using Taylor's formula as before. We only consider the forward formula.

*Proof.* Consider first Taylor's formula at  $x+h$ :

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + O(h^3)$$

Can be proven using Taylor's formula precisely like Theorem 23.

This is especially relevant when we will consider differential equations with boundary conditions specifying the value of  $y'$  at the endpoints, so-called Neumann conditions.

We proved it is a first-order method.

and at  $x + 2h$ :

$$f(x + 2h) = f(x) + 2hf'(x) + 2h^2f''(x) + O(h^3).$$

We multiply the first one by 4 which gives:

$$\begin{cases} 4f(x + h) &= 4f(x) + 4hf'(x) + 2h^2f''(x) + O(h^3) \\ f(x + 2h) &= f(x) + 2hf'(x) + 2h^2f''(x) + O(h^3) \end{cases}$$

subtracting the equations yields:

$$4f(x + h) - f(x + 2h) = 3f(x) + 2hf'(x) + O(h^3)$$

which rearranged is just

$$f'(x) = \frac{-3f(x) + 4f(x + h) - f(x + 2h)}{2h} + O(h^2).$$

Set  $2h$  instead of  $h$  so that  $(2h)^2/2 = 2h^2$ .

Same trick as in the proof of Theorem 23, we want to get rid of the  $h^2$  terms.

This proves that the error is  $O(h^2)$ .

□

### Second derivative

The next logical step is to derive an approximation for  $f''(x)$ . The approximation should be at least a second order method to match the accuracy of the previous section. Ideally we would like to find an approximation for  $f''(x)$  which uses values of  $f$  but not of the first derivative  $f'$ . As in the previous section we begin by writing down Taylor's formula at  $x + h$  and  $x - h$ , although we will go one term further in the expansion:

$$\begin{cases} f(x + h) &= f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \frac{h^3}{6}f'''(x) + O(h^4) \\ f(x - h) &= f(x) - hf'(x) + \frac{h^2}{2}f''(x) - \frac{h^3}{6}f'''(x) + O(h^4) \end{cases}$$

Instead of subtracting the equations from one another, we add them to obtain

$$f(x + h) + f(x - h) = 2f(x) + h^2f''(x) + O(h^4).$$

A bit of rewriting leads to the following definition/theorem.

Since  $f'$  is already computed numerically the errors would only magnify.

We now want to keep the terms involving  $f''(x)$  and get rid of  $f'(x)$ .

The error term becomes  $O(h^2)$  when the equation is divided by  $h^2$ .

**Definition 23.** The central difference approximation of  $f''$  at  $x$  is

$$f''(x) \approx \frac{f(x + h) - 2f(x) + f(x - h)}{h^2}.$$

It is a second order approximation.

### Partial derivatives

In the next chapter we will consider boundary values in many dimensions - and partial differential equations instead of ordinary ones. We will therefore need to approximate partial derivatives. The formulas of Definitions 21 and 23 translate in a straightforward manner. For instance if  $u(x, y)$  is a function of two variables we may approximate its first partial derivatives by

$$\frac{\partial u}{\partial x}(x, y) \approx \frac{u(x + h, y) - u(x - h, y)}{2h}$$

and

$$\frac{\partial u}{\partial y}(x, y) \approx \frac{u(x, y + k) - u(x, y - k)}{2k}.$$

In general we may want to use different approximation rates  $h$  and  $k$ .

The formula for  $\partial u / \partial x$  keeps  $y$  constant.

### Linear boundary value problems

#### Theory

We consider a second-order differential equation on an interval  $[a, b]$

$$y'' = f(t, y, y')$$

together with two boundary conditions at  $t = a$  and  $t = b$ . In general there are three types of boundary conditions which will be modelled in different ways:

- Dirichlet boundary conditions specify the value of  $y(a)$  and  $y(b)$ , for instance

$$y(a) = 0 \quad y(b) = 1.$$

- Neumann boundary conditions specify the value of  $y'(a)$  and  $y'(b)$ , for instance

$$y'(a) = -1 \quad y'(b) = 0.$$

- Robin boundary conditions specify any linear combination of  $y$  and  $y'$  at the endpoints, for instance

$$2y(a) - y'(a) = 0 \quad y(b) + y'(b) = 0.$$

The general idea is to discretise the interval  $[a, b]$

$$a = t_0 < t_1 < t_2 < \cdots < t_n = b$$

The Van der Pol oscillator

$$y'' - \mu(1 - y^2)y' + y = 0$$

is a typical example of such an equation.

Typical of a controlled system at the endpoints.

Used when the flow in and out of the system is known, e.g. the system is insulated.

Used for more complicated boundary interactions, such as convection.

We clearly have  $t_i = a + hi, 0 \leq i \leq n$ .



with step size  $h = (b - a)/n$ . We want to compute an approximate value at  $t = t_i$ , denoted by  $y_i$ .

We cannot rely on an iterative scheme as in the previous chapter as our solution needs both to satisfy conditions at  $t_0$  and  $t_n$ . Instead we use numerical differentiation to turn the equation inside the interval into a system for  $y_1, y_2, \dots, y_{n-1}$ . The boundary conditions are then used to obtain informations on  $y_0$  and  $y_n$ . If the differential equation is linear we obtain a linear system which is easily solved. The discretization is called the finite difference method.

### Finite difference method - Linear equation - Dirichlet conditions

As a first example we consider a beam supported at both ends, with axial tension load  $P$  and an equally distributed vertical  $q(x)$  load of constant intensity  $w$ . The vertical deflection  $y(x)$  of the beam satisfies the equation

$$y'' - \frac{P}{Es^4}y = -\frac{w}{2Es^4}x(L-x)$$

where  $L$  is the length of the beam,  $s$  its cross section and  $E$  the modulus of elasticity of the material. Since the beam is fixed at both ends we have boundary conditions

$$y(0) = y(L) = 0$$

which are of Dirichlet type. We discretise the interval  $[0, L]$  as before

$$0 = x_0 < x_1 < x_2 < \dots < x_n = L$$

with step size  $h = L/n$ . At an inner point  $x_i$  ( $1 \leq i \leq n-1$ ) we use the approximation

$$y''(x_i) \approx \frac{y(x_i+h) - 2y(x_i) + y(x_i-h)}{h^2}$$

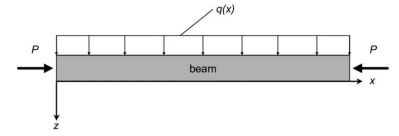
of Definition 23. Now we have  $x_i + h = x_{i+1}$  and  $x_i - h = x_{i-1}$ . Using our notation  $y(x_i) \approx y_i$  gives us the approximate equation

$$y''(x_i) \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2}, \quad 1 \leq i \leq n-1.$$

We now insert this approximation into the differential equation, letting  $x = x_i$  and therefore  $y(x_i) \approx y_i$  along the way. We obtain for  $1 \leq i \leq n-1$ :

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} - \frac{P}{Es^4}y_i = -\frac{w}{2Es^4}x_i(L-x_i)$$

An iterative method such as Runge-Kutta starts with a correct value at  $t_0$  but ends up with an approximate value at  $t_n$ .



**Figure 58:** Schematic drawing of beam subject to both axial and transverse (vertical) load.

Values used later:  $s = 4$  cm,  $L = 100$  cm.  
 $E = 1.3 \times 10^6$  Pa,  $P = 100$  kg,  
 $w = 20$  kg/cm.

Since the points  $x_i$  are separated by the step size  $h$ .

It is considered good practice to multiply the equation by the common denominator  $2h^2Es^4$ , group similar terms and obtain:

$$2Es^4y_{i-1} - (4Es^4 + 2h^2P)y_i + 2Es^4y_{i+1} = -wh^2x_i(L - x_i).$$

We have successfully turned the differential equation at  $x = x_i$  into an algebraic equation for some of the  $y$  variables. We have a total of  $n - 1$  such equations for the  $n + 1$  unknowns  $y_0, \dots, y_n$ . We seem to have too few equations but the boundary conditions deliver two extra equations at the endpoints. In the case of Dirichlet boundary conditions, these equations are trivial:

$$y_0 = y_n = 0$$

It now remains to rewrite this linear system in matrix form and solve it numerically. The value of  $n$  (and therefore  $h$ ) can be chosen freely. The full system of  $n + 1$  equations is:

$$\left\{ \begin{array}{rcl} & y_0 & = 0 \\ 2Es^4y_0 - (4Es^4 + 2h^2P)y_1 + 2Es^4y_2 & = & -wh^2x_1(L - x_1) \\ 2Es^4y_1 - (4Es^4 + 2h^2P)y_2 + 2Es^4y_3 & = & -wh^2x_2(L - x_2) \\ & \dots & \\ 2Es^4y_{n-2} - (4Es^4 + 2h^2P)y_{n-1} + 2Es^4y_n & = & -wh^2x_{n-1}(L - x_{n-1}) \\ & y_n & = 0 \end{array} \right.$$

In matrix form  $Ay = b$  this corresponds to

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ \alpha & \beta & \alpha & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & \alpha & \beta & \alpha & 0 & \dots & 0 & 0 & 0 \\ \vdots & & \ddots & \ddots & \ddots & & & & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & & & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & & & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \dots & \alpha & \beta & \alpha \\ 0 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 1 \end{pmatrix}, b = \begin{pmatrix} 0 \\ \delta x_1(L - x_1) \\ \delta x_2(L - x_2) \\ \vdots \\ \delta x_{n-1}(L - x_{n-1}) \\ 0 \end{pmatrix}.$$

The matrix  $A$  is almost tridiagonal and could therefore be coded as a sparse matrix if necessary. Solving the boundary value problem is then simply a matter of solving this linear system. We begin by

The reason is twofold: primarily avoid dividing by the small number  $h^2$  which can introduce inaccuracy. Secondly the equations are much more elegant without fractions and easier to type in.

The values of  $x_i$  are known from the start:  $x_i = hi$ .

Recall  $y(0) = y(L) = 0$  is our boundary condition.

A large value of  $n$  increases accuracy and running time.

We include the trivial equations  $y_0 = y_n = 0$  as part of the system - with other boundary conditions we will not have the luxury of knowing the values beforehand.

We define the constants  $\alpha = 2Es^4$ ,  $\beta = -(4Es^4 + 2h^2P)$  and  $\delta_i = -wh^2$  for simplicity.

The coordinates of  $b$  are known beforehand since  $x_i = hi$  is given.

Except for the first and last row.

defining  $A$  and  $b$  as a function of the number of steps  $n$ . It is worth recalling that  $A$  is a  $(n+1) \times (n+1)$  matrix since there is a total of  $n+1$  points.

```

1 function [A,b]=finitediff(n)
2 L=100;s=4;E=1.3*10^6;w=20;P=100;h=L/n;
3 alpha=E*s^4;
4 beta=-(2*E*s^4+h^2*P);
5 delta=-w*h^2;
6
7 for i=2:n
8     A(i,i-1)=alpha;
9     A(i,i)=beta;
10    A(i,i+1)=alpha;
11 end
12
13 A(1,1)=1;A(n+1,n+1)=1;
14
15 x=linspace(0,L,n+1);
16 b=(delta*x.*(L-x))';
17 end

```

The main command is simply

```

1 n=100;
2 [A,b]=finitediff(n);
3 y=A\b;

```

The resulting values  $y_0, \dots, y_n$  are plotted on Figure 59. The boundary conditions  $y(0) = y(L) = 0$  are automatically satisfied by construction of  $A$ . The value  $n = 100$  is chosen semi-randomly - our discussion in the chapter on Linear systems tells us that systems of  $10^4$  equations are solved in about a second, so we cannot consider fine discretization unless we use a sparse matrix setup.

### General boundary conditions and sparse setup

We consider another linear boundary value problem, this time with mixed boundary conditions. We will as well construct the coefficient matrix  $A$  as a sparse matrix to allow for larger values of  $n$  to be used.

We consider a copper wire of length  $L$  and radius  $r$ . Its left end (at  $x = 0$ ) is at a constant temperature  $T_{\text{left}}$ . Its right end (at  $x = L$ ) loses heat through convection, and the heatloss also happens along the

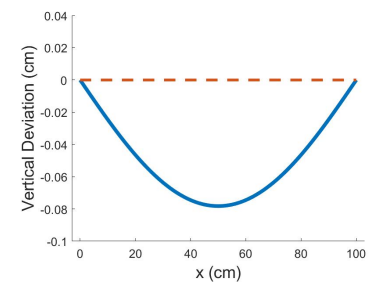
The program only has one variable, the number of steps  $n$ . Necessary constants and the step size are defined on Line 2.

The elements of  $A$  are defined one by one through a loop. We need to address the first and row manually.

Line 15 creates all  $x_i$  values. Note that  $x_0(L - x_0) = x_n(L - x_n) = 0$  so the formula on Line 11 also holds for the first and last coordinate. In general we may need to manually modify them. Note that  $b$  needs to be a column vector.

This is a very inefficient way to define  $A$  - Matlab performs poorly on large loops. We do better in our second attempt.

More on the value of  $n$  later. The vector  $y$  contains the approximate values at  $x_0, x_1, \dots, x_n$ .



**Figure 59:** Vertical deviation of the beam under uniform load. For plotting purposes we need to reverse the sign of the vector  $y$ . The maximum deviation is about 0.8 mm.

length of the rod. The temperature distribution of the rod satisfies the linear differential equation

$$-Kr^2T''(x) + 2HrT(x) = 2HrT_{\text{ext}}$$

where  $T_{\text{ext}}$  is the constant external temperature,  $k$  is the thermal conductivity of copper and  $C$  the convective heat transfer coefficient of the ambient air. The boundary condition at  $x = 0$  is simply [-5mm]

$$T(0) = T_0$$

At  $x = L$  the flow is directly proportional to the difference with the external temperature

$$T'(L) = \frac{C}{k}(T_{\text{ext}} - T(L))$$

The general strategy is as in the previous example. We discretise

$$0 = x_0 < x_1 < x_2 < \cdots < x_n = L$$

with step size  $h = L/n$  and let  $T_i = T(x_i)$ . At an inner point  $x_i$ , where  $1 \leq i \leq n-1$  we use the central difference approximation

$$T''(x_i) \approx \frac{T_{i+1} - 2T_i + T_{i-1}}{h^2}.$$

This turns the differential equation into

$$-kr^2 \frac{T_{i+1} - 2T_i + T_{i-1}}{h^2} + 2HrT_i = 2CrT_{\text{ext}}.$$

Again we multiply both sides by  $h^2$  and rearrange into

$$-kr^2T_{i-1} + \left(2kr^2 + 2Crh^2\right)T_i - kr^2T_{i+1} = 2CrT_{\text{ext}}h^2.$$

The left boundary condition simply states

$$T_0 = T_{\text{left}}.$$

The right boundary condition requires us to use the three point backward approximation of the first derivative, see Definition 22:

$$T'(L) \approx \frac{3T_n - 4T_{n-1} + T_{n-2}}{2h}.$$

The Robin condition then translates to:

$$\frac{3T_n - 4T_{n-1} + T_{n-2}}{2h} = \frac{C}{k}(T_{\text{ext}} - T_n).$$

Again we multiply both sides by  $2kh$  and reorganize the equation into

$$kT_{n-2} - 4kT_{n-1} + (3k + 2hC)T_n = 2hCT_{\text{ext}}.$$

We use the values  $L = 1 \text{ m}$ ,  $r = 0.01 \text{ m}$ ,  $T_{\text{left}} = 150^\circ \text{ C}$ ,  $T_{\text{ext}} = 20^\circ \text{ C}$ ,  $k = 384 \text{ W} \cdot \text{m}^{-1} \cdot \text{K}^{-1}$  and  $C = 5 \text{ W} \cdot \text{m}^{-2} \cdot \text{K}^{-1}$ .

The left end is at constant temperature. This is a Dirichlet boundary condition.

This is a Robin boundary condition which involves both  $T$  and  $T'$  at the endpoint.

This is preparation for the matrix form of the system. We obtain a total of  $n-1$  equations where  $1 \leq i \leq n-1$ .

One extra equation.

There is no point to the right of  $L$  - the rod is finite. In general the formula is

$$f'(x) \approx \frac{3f(x) - 4f(x-h) + f(x-2h)}{2h}$$

Our condition is

$$T'(L) = \frac{C}{k}(T_{\text{ext}} - T(L))$$

The discretized version of  $T(L)$  is simply the last value  $T_n$ .

The final equation of our  $(n+1) \times (n+1)$  system.

For clarity let us rewrite the equations in order from left to right:

$$\left\{ \begin{array}{rcl} T_0 & = & T_{\text{left}} \\ -kr^2 T_0 + (2kr^2 + 2Crh^2) T_1 - kr^2 T_2 & = & 2CrT_{\text{ext}}h^2 \\ -kr^2 T_1 + (2kr^2 + 2Crh^2) T_2 - kr^2 T_3 & = & 2CrT_{\text{ext}}h^2 \\ & \dots & \\ -kr^2 T_{n-2} + (2kr^2 + 2Crh^2) T_{n-1} - kr^2 T_n & = & 2CrT_{\text{ext}}h^2 \\ kT_{n-2} - 4kT_{n-1} + (3k + 2hC) T_n & = & 2hCT_{\text{ext}} \end{array} \right.$$

In matrix form  $AT = \mathbf{b}$  this corresponds to

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ \alpha & \beta & \alpha & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & \alpha & \beta & \alpha & 0 & \dots & 0 & 0 & 0 \\ \vdots & & \ddots & \ddots & \ddots & & & & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & & & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & & & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \dots & \alpha & \beta & \alpha \\ 0 & 0 & 0 & 0 & 0 & \dots & k & -4k & 3k + 2hC \end{pmatrix}, \mathbf{b} = \begin{pmatrix} T_{\text{left}} \\ \delta \\ \delta \\ \vdots \\ \delta \\ 2hCT_{\text{ext}} \end{pmatrix}.$$

We use the shorthands:

$$\alpha = -Kr^2$$

$$\beta = 2kr^2 + 2Crh^2$$

$$\delta = 2CrT_{\text{ext}}h^2$$

The asymmetry of the matrix stems from the different boundary conditions on the left and right end of the rod. It is not quite tridiagonal.

Solving the boundary value problem is again a simple matter of coding these two objects and solving the resulting linear system. This time we will define  $A$  as a sparse matrix to save time and space.

```

1 function [A,b]=finitediff2(n)
2 L=1;r=0.01;Tleft=150;Text=20;k=384;C=5;h=L/n;
3 alpha=-k*r^2;
4 beta=2*k*r^2+2*C*r*h^2;
5 gamma=2*C*r*Text*h^2;
6 %define A as sparse matrix
7 i=[2:n , 2:n , 2:n]';
8 j=[1:n-1 , 2:n , 3:n+1]';
9 values=[alpha*ones(n-1,1);beta*ones(n-1,1);alpha
        *ones(n-1,1)];
10 A=sparse(i,j,values);
11 A(1,1)=1;
12 A(n+1,n-1)=k;A(n+1,n)=-4*k;
13 A(n+1,n+1)=3*k+2*h*C;
14 %define b
15 b=gamma*ones(n+1,1);
16 b(1)=Tleft;
17 b(n+1)=2*h*C*Text;
18 end

```

We then solve the linear system. The sparse setup allows for fairly high values of  $n$ .

```

1 n=10^6;
2 [A,b]=finitediff2(n);
3 T=A\b;

```

The temperature distribution of the vector  $T$  is shown on Figure 60.

### A more general temperature model

We revisit the previous model under a more general assumption on the shape of the rod. Instead of being cylindrical (that is, its radius  $r$  is constant) we assume a variable radius

$$r = r(x) = 0.01 \left( 1 + \frac{1}{1 + (x - L)^2} \right).$$

The length is maintained at  $L = 1$  m. Figure 61 shows the function  $r(x)$ . The differential equation becomes

$$-k \frac{d}{dx} \left( r(x)^2 T'(x) \right) + 2Cr(x)T(x) = 2Cr(x)T_{\text{ext}}$$

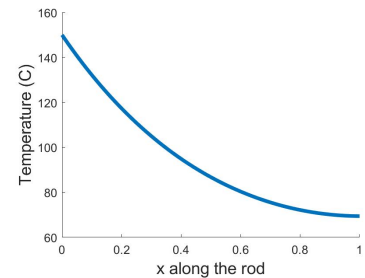
Lines 2-5 define the constants and shorthands.

Lines 7 and 8 declare where non-zero elements of  $A$  are placed. For instance the  $\beta$ s are placed on the diagonal with  $2 \leq i \leq n$  (remove first and last row). The  $\alpha$ s are placed in the same way with one offset to the left and right.

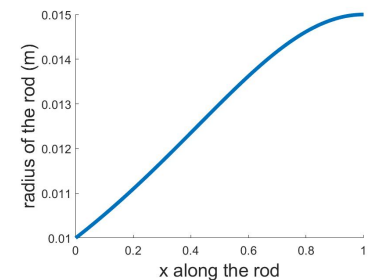
Line 9 then assigns to each pair (row,column) its value - vectorwise. We have to place  $n - 1$  times each constant.

Line 10 creates a sparse matrix with these elements. Top and bottom row and then assigned on Lines 11-13.

This code takes well under a second to compute - a non sparse setup would take years if  $n = 10^6$ .



**Figure 60:** Temperature distribution along the rod with left end maintained at  $150^\circ$  and convection through the rod's length and the right end. Minimum temperature is about  $69.4^\circ$ .



**Figure 61:** Radius of the rod as a function of  $x$ . The rod is widest at its right end with radius  $r = 1.5$  cm and narrowest at its left end where  $r = 1$  cm.

and we assume the same boundary conditions as earlier. We simplify the first term using the power rule:

$$\frac{d}{dx} \left( r(x)^2 T'(x) \right) = r(x)^2 T''(x) + 2r(x)r'(x)T'(x)$$

and our full differential equation becomes

$$-kr^2(x)T''(x) - 2kr(x)r'(x)T'(x) + 2Cr(x)T(x) = 2Cr(x)T_{\text{ext}}.$$

In order to discretise this equation we need not only an approximation for  $T''$  but also for  $T'$ . Recall that the central difference approximation for the first derivative (see Definition 21) is

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

which in our context means

$$T'(x_i) \approx \frac{T_{i+1} - T_{i-1}}{2h}$$

for  $1 \leq i \leq n-1$ . The  $T''$  term is handled as earlier. Because  $r$  and  $r'$  are not constants, we use the shorthand  $r_i = r(x_i)$  and  $r'_i = r'(x_i)$  to rewrite the differential equation as

$$-kr_i^2 \frac{T_{i+1} - 2T_i + T_{i-1}}{h^2} - 2kr_i r'_i \frac{T_{i+1} - T_{i-1}}{2h} + 2Cr_i T_i = 2Cr_i T_{\text{ext}}.$$

We multiply both sides by  $h^2$  and regroup the terms of the same kind to obtain a total of  $n-1$  equations

$$\begin{aligned} & \left( -kr_i^2 + khr_i r'_i \right) T_{i-1} + \left( 2kr_i^2 + 2Cr_i h^2 \right) T_i \dots \\ & + \dots \left( -kr_i^2 - khr_i r'_i \right) T_{i+1} = 2Cr_i h^2 T_{\text{ext}}. \end{aligned}$$

The boundary conditions are precisely the same as before so can write the system in matrix form  $AT = \mathbf{b}$ . Again note that  $r_i$  and  $r'_i$  are variable now.

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\ \alpha_1 & \beta_1 & \gamma_1 & 0 & 0 & \dots & 0 & 0 & 0 \\ 0 & \alpha_2 & \beta_2 & \gamma_2 & 0 & \dots & 0 & 0 & 0 \\ \vdots & & \ddots & \ddots & \ddots & & & & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & & & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & & & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \dots & \alpha_{n-1} & \beta_{n-1} & \gamma_{n-1} \\ 0 & 0 & 0 & 0 & 0 & \dots & k & -4k & 3k + 2hC \end{pmatrix}, \mathbf{b} = \begin{pmatrix} T_{\text{left}} \\ \delta_1 \\ \delta_2 \\ \vdots \\ \delta_{n-1} \\ 2hCT_{\text{ext}} \end{pmatrix}.$$

$(fg)' = f'g + fg'$ . We differentiate the composite function  $r(x)^2$  using the chain rule:

$$\frac{d}{dx} r(x)^2 = 2r(x)r'(x)$$

We will compute  $r'(x)$  later when implementing. It would be an unnecessary complication at this stage.

At  $x = x_i$  we have  $x+h = x_{i+1}$  and  $x-h = x_{i-1}$ . The temperature values there are  $T_{i+1}$  and  $T_{i-1}$ .

For  $1 \leq i \leq n-1$ .

where we use

$$\alpha_i = -kr_i^2 + khr_i r_i' \quad \beta_i = 2kr_i^2 + 2Cr_i h^2$$

$$\gamma_i = -kr_i^2 - khr_i r_i' \quad \delta_i = 2Cr_i h^2 T_{\text{ext}}$$

Implementation begins by calculating the function  $r'$  using symbolic differentiation:

```
1 syms x,L
2 y=0.5+1./(1+(x-L).^2);
3 diff(f,x)
```

the result of which can be then hardcoded into two functions for  $r$  and  $r'$

```
1 function y=rfunc(x)
2 L=1;
3 y=0.01*(0.5+1./(1+(x-L).^2));
4 end
5
6 function y=drfunc(x)
7 L=1;
8 y=-0.01*(2*(x-L)./(1+(x-L).^2));
9 end
```

The main coding of  $A$  and  $b$  is shown on the next page.

Recall  $r_i = r(x_i)$ ,  $r_i' = r'(x_i)$ . The values of  $x_i$  are known beforehand, and so are the functions  $r$  and  $r'$ .

Of course this calculation could easily be performed by hand in this case.

The functions need to be ready to take vectors as inputs.



```

1 function [A,b]=finitediff3(n)
2 L=1;Tleft=150;Text=20;k=384;C=5;h=L/n;
3 x=linspace(0,L,n+1);
4 r=rfunc(x);
5 dr=drfunc(x);
6 r=r(2:end-1);dr=dr(2:end-1);
7
8 alpha=-k*r.^2+k*h*r.*dr;
9 beta=2*k*r.^2+2*C*r*h^2;
10 gamma=-k*r.^2-k*h*r.*dr;
11 delta=2*C*h*r*Text;
12
13 i=[2:n , 2:n , 2:n]';
14 j=[1:n-1 , 2:n , 3:n+1]';
15 values=[alpha';beta';gamma'];
16 A=sparse(i,j,values);
17 A(1,1)=1;
18 A(n+1,n-1)=k;A(n+1,n)=-4*k;
19 A(n+1,n+1)=3*k+2*h*C;
20
21 b(1)=Tleft;
22 b(2:n)=delta;
23 b(n+1)=2*h*C*Text;
24 b=b';
25 end

```

Lines 4 to 6 compute  $r_i$  and  $r'_i$  along the rod. The vector  $x$  contains  $x_0, x_1, \dots, x_n$ . We actually only need  $1 \leq i \leq n-1$  so the first and last values are filtered out on Line 5.

Lines 8 to 11 define  $\alpha_i$ ,  $\beta_i$ ,  $\gamma_i$  and  $\delta_i$ .

Lines 13 to 16 define the main part of the  $(n+1) \times (n+1)$  matrix  $A$ . For instance the  $\alpha_i$  are placed on the following slots of the matrix:  $(2,1)$ ,  $(3,2)$ , ...,  $(n,n-1)$ . Similarly the  $\delta_i$  are placed at  $(2,3)$ ,  $(3,4)$ , ...,  $(n,n+1)$ . It is here helpful to study the general shape of the matrix  $A$ , see the previous page.

Lines 17-19 correct for the boundary conditions. These lines are identical to the previous implementation.

Lines 20-23 define the vector  $b$ . It needs to be a column vector.

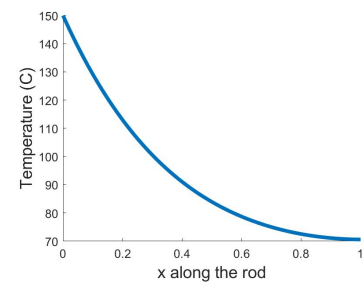
Finally the system is solved. We can use large values of  $n$  thanks to the sparse setup.

```

1 n=10^6;
2 [A,b]=finitediff3(n);
3 A1=full(A);
4 T=A\b;

```

The temperature distribution on the rod is shown on Figure 62.



**Figure 62:** Temperature distribution along the rod with left end maintained at  $150^\circ$  and convection through the rod's length and the right end. The rod gets wider as  $x$  increases. Minimum temperature is about  $70.6^\circ$ .

## Non-linear boundary value problems

### Theory

Non-linear problems can be cumbersome to set up, but the general idea stays the same. However the final step of solving the system

becomes more complex.

1. Use central difference approximations to convert the differential equation at an inner point  $x_i$ ,  $1 \leq i \leq n-1$  into  $n-1$  algebraic equations for the unknowns  $y_i \approx y(x_i)$ .
2. Boundary conditions are converted into two extra equations. First derivatives are estimated through forward or backward three-point approximations if needed.
3. The system is written as  $F(y) = 0$  where  $F : \mathbb{R}^{n+1} \rightarrow \mathbb{R}^{n+1}$ . We compute the Jacobi matrix of  $F$  as well.
4. The non-linear system is solved numerically using multivariate Newton's method.

Dirichlet boundary conditions are straightforward while Neumann and Robin conditions require to discretise the first derivatives.

### Implementation with mixed boundary conditions

We consider the non-linear differential equation

$$y'' - 10y' - 5y^2 = 0$$

on the interval  $x \in [0, 1]$  together with boundary conditions

$$10y(0) - y'(0) = 50 \quad y(1) = 0.$$

We follow the four-steps theoretical plan. We use both formulas

$$y''(x_i) \approx \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} \quad y'(x_i) \approx \frac{y_{i+1} - y_{i-1}}{2h}$$

to obtain for  $1 \leq i \leq n-1$ :

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} - 10 \frac{y_{i+1} - y_{i-1}}{2h} - 5y_i^2 = 0.$$

We simplify the equation by multiplying both sides with  $h^2$  and regrouping terms. Multiplying with  $h^2$  is highly recommended to avoid large numbers being introduced into the system.

$$(1 + 5h)y_{i-1} - 2y_i - 5h^2y_i^2 + (1 - 5h)y_{i+1} = 0.$$

The Robin boundary condition at  $x = 0$  is discretized as follows. We first use three-point forward difference approximation and obtain

$$y'(0) \approx \frac{-3y_0 + 4y_1 - y_2}{2h}$$

so the boundary condition at  $x = 0$  becomes

$$10y_0 - \frac{-3y_0 + 4y_1 - y_2}{2h} = 50 \Rightarrow 10y_0 - \frac{-3y_0 + 4y_1 - y_2}{2h} - 50 = 0.$$

This equation can be used to compute the concentration of a reactant in a specific chemical reaction within a reactor with dispersion.

The condition at  $x = 0$  is Robin, while the condition at  $x = 1$  is Dirichlet.

Step 1: turn the differential equation into  $n-1$  equations using central difference approximation.

The last term makes the equation non-linear.

$1/h^2$  becomes quite large when  $h$  is small.

The system is non-linear so it is impossible to rewrite it as  $Ay = b$ . The most important thing is to make sure to have all terms on the same side of the equation, since Newton's method requires a system of the form  $F(y) = 0$ .

Step 2: turn the boundary equations into two extra equations. We need to use forward difference approximation at  $x = 0$ .

It was  $10y(0) - y'(0) = 50$ .

Likewise we multiply both sides by  $2h$  to obtain:

$$(20h + 3)y_0 - 4y_1 + y_2 - 100h = 0.$$

The Dirichlet boundary condition at  $x = 1$  is simply

$$y_n = 0.$$

These  $n + 1$  equations form a system  $F(\mathbf{y}) = \mathbf{0}$  where

$$F(\mathbf{y}) = \begin{pmatrix} (20h + 3)y_0 - 4y_1 + y_2 - 100h \\ (1 + 5h)y_0 - 2y_1 - 5h^2y_1^2 + (1 - 5h)y_2 \\ (1 + 5h)y_1 - 2y_2 - 5h^2y_2^2 + (1 - 5h)y_3 \\ \vdots \\ \vdots \\ (1 + 5h)y_{n-2} - 2y_{n-1} - 5h^2y_{n-1}^2 + (1 - 5h)y_n \\ y_n \end{pmatrix}.$$

In order to use Newton's method we must compute the Jacobi matrix of  $F$ . Since the number of variables  $n + 1$  is not fixed we cannot completely rely on symbolic differentiation to obtain the answer. However calculations are simple enough to be done by hand. Consider the first row

$$f_0(\mathbf{y}) = (20h + 3)y_0 - 4y_1 + y_2 - 100h.$$

Only three partial derivatives are not 0:

$$\frac{\partial f_0}{\partial y_0} = 20h + 3, \quad \frac{\partial f_0}{\partial y_1} = -4, \quad \frac{\partial f_0}{\partial y_2} = 1.$$

Next we consider the general term ( $1 \leq i \leq n - 1$ ).

$$f_i(\mathbf{y}) = (1 + 5h)y_{i-1} - 2y_i - 5h^2y_i^2 + (1 - 5h)y_{i+1}.$$

Again only three partial derivatives matter:

$$\begin{aligned} \frac{\partial f_i}{\partial y_{i-1}} &= 1 + 5h, & \frac{\partial f_i}{\partial y_i} &= -2 - 10h^2y_i \\ \frac{\partial f_i}{\partial y_{i+1}} &= 1 - 5h. \end{aligned}$$

Finally the final equation

$$f_n(\mathbf{y}) = y_n$$

trivially only yields one non-zero partial derivative:

$$\frac{\partial f_n}{\partial y_n} = 1.$$

Step 3: write the system in vector form.

First line corresponds to the boundary condition at the left end. The next  $n - 1$  lines come from the discretization at  $x_1, x_2, \dots, x_{n-1}$ . The final line reflects the boundary condition  $y_n = 0$ .

Although we may consider a small value of  $n$  to get the idea.

These are very similar terms than in the linear case. The boundary condition at  $x = 0$  is linear after all.

The Jacobi matrix  $DF$  has therefore the following structure.

$$DF(y) = \begin{pmatrix} 20h+3 & -4 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ \alpha & \beta_1 & \delta & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & \alpha & \beta_2 & \delta & 0 & \cdots & 0 & 0 & 0 \\ \vdots & & \ddots & \ddots & \ddots & & & & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & & & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & & & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & \alpha & \beta_{n-1} & \delta \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 \end{pmatrix}$$

We use the shorthands:

$$\alpha = 1 + 5h$$

$$\beta_i = -2 - 10h^2 y_i$$

$$\delta = 1 - 5h$$

The first and last row reflect the boundary conditions.

A major difference to the linear case is that the  $\beta_i$  are now functions of the variable  $y_i$  - the matrix is not fixed. These two objects are coded as follows, first  $F$ :

```

1 function z=F(y)
2 n=length(y)-1;
3 h=1/n;
4 z(1)=(20*h+3)*y(1)-4*y(2)+y(3)-100*h;
5 for i=2:n
6     z(i) = (1+5*h)*y(i-1) - 2*y(i) - 5*h^2*y(i)
7           + (1-5*h)*y(i+1);
8 end
9 z(n+1)=y(n+1);
10 z=z';
end

```

We do not use a sparse setup here since  $F$  is a "full" vector.

It is rather annoying that Matlab cannot start vectors with the index 0 so we must shift all indices by one.

The outcome of  $F$  should be a column vector.

The Jacobi matrix  $DF$  is then saved as a sparse matrix.

```

1 function A=DF(y);
2 n=length(y)-1;
3 h=1/n;
4 alpha=(1+5*h)*ones(n-1,1);
5 beta=-2-10*h^2*y(2:end-1);
6 delta=(1-5*h)*ones(n-1,1);
7
8 i=[2:n , 2:n , 2:n]';
9 j=[1:n-1 , 2:n , 3:n+1]';
10 values=[alpha;beta;delta];
11 A=sparse(i,j,values);
12 A(1,1)=20*h+3;
13 A(1,2)=-4;
14 A(1,3)=1;
15 A(n+1,n+1)=1;
16 end

```

We are finally ready to run Newton's method on  $F$  and  $DF$ . For convenience here is the general algorithm for Newton's method:

```

1 function x=newtonmult(x0,tol)
2     x=x0;
3     oldx=x+2*tol;
4     while norm(x-oldx)>tol
5         oldx=x;
6         s=DF(x)\F(x);
7         x=x-s;
8     end
9 end

```

It requires an initial guess, and a tolerance value. It is wise not to have the tolerance too close to zero here since  $F$  and  $DF$  are already derived through a numerical approximation. A value of  $10^{-3}$  is sufficient. There is no easy way to determine an initial guess - our only certainty is that  $y_n = 0$ . The simplest guess is the zero vector. The main code is then simply

```

1 n=10000;tol=0.001;
2 y0=zeros(n+1,1);
3 y=newtonmult(y0,tol);

```

The solution  $y$  is plotted on Figure 63.

We could also carry  $n$  and  $h$  over from the main program. Easy enough to compute them from scratch.

Very similar setup to the previous example. We define the numbers  $\alpha$  and  $\delta$  and vectorize them, then the vector  $\beta$  is defined. Notice how we remove  $y_0$  and  $y_n$  since the  $\beta$  do not cover the whole diagonal.

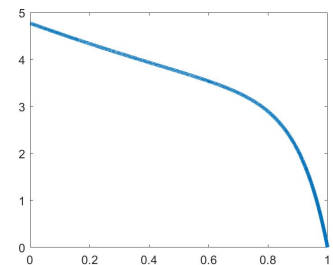
Lines 8-11 build the tridiagonal part of the matrix. This is virtually identical to the previous case. Lines 12-15 add the correct boundary equations.

Indices  $i = 1$  and  $i = n + 1$  need to be dealt with separately.

Step 4: use multivariate Newton's method to solve the non-linear system.

If it doesn't work, try another initial guess. It is difficult to guess what the solution will be like unless we have some physical intuition about our system.

We choose a large but not too large value for  $n$  - note that non-linear systems are much slower to solve since we must solve a large linear system in each iteration of Newton's method.



**Figure 63:** Solution to the non-linear boundary value problem.



# Partial differential equations

The finite difference methods of the previous chapter can also be used to solve partial differential equations (PDEs) subject to boundary conditions. The subject is vast and we will only consider two simple linear partial differential equations: the heat equation and the Poisson equation.

## *The heat equation - Parabolic PDEs*

We consider the heat equation in one spatial dimension  $x$ :

$$\frac{\partial u}{\partial t} = D \frac{\partial^2 u}{\partial x^2}$$

on the space interval  $x \in [a, b]$ . It describes the temperature  $u(x, t)$  measured along a one-dimensional rod of length  $L$ . The constant  $D$  is called the diffusion coefficient. In order to solve the heat equation we need extra conditions:

- An initial condition at  $t = 0$  which describes the initial temperature distribution along the rod. In mathematical terms  $u(x, 0)$  is a known function of  $x$ , say  $f(x)$  for  $0 \leq x \leq L$ .
- Boundary conditions at  $x = 0$  and  $x = L$  that have to be satisfied at any moment in time. These boundary conditions can be of the same three types as the previous chapter: Dirichlet, Neumann or Robin. Dirichlet conditions are used when the temperature at the endpoint is known. Neumann conditions are used when the flow out of the rod is known (e.g. the rod is insulated there). We will consider both cases.

The heat equation is a simple linear model for all sorts of diffusion phenomena.

A typical Dirichlet condition such as  $u(0, t) = 20$  describes a constant temperature at the left end  $x = 0$ .

A typical Neumann condition such as  $\frac{\partial u}{\partial x}(0, t) = 0$  describes insulation (no flow) at the left end  $x = 0$ .

## *discretization in time and space*

The main technical hurdle in applying finite difference methods is discretizing in both  $x$  and  $t$  directions. The spatial variable is discretized as in the previous chapter:

$$0 = x_0 < x_1 < x_2 < \cdots < x_m = L$$

with step size  $h = L/m$ . Although there is no natural upper bound for the time variable  $t$  we shall restrict ourselves to a bounded time interval  $[0, T]$ . This interval is then discretized as well:

$$0 = t_0 < t_1 < t_2 < \cdots < t_n = T$$

with step size  $k = T/n$ . Our goal is go to calculate an approximate value for the solution at  $x = x_i, t = t_j$ , which we denote by  $w_{i,j}$ , for all  $0 \leq i \leq m, 0 \leq j \leq n$ .

The central difference approximation for the second derivative is applied to  $\partial^2 u / \partial x^2$  at an inner point  $(x_i, t_j)$  of the grid. The general formula of Definition 23 is:

$$\frac{\partial^2 u}{\partial x^2}(x_i, t_j) \approx \frac{u(x_i + h, t_j) - 2u(x_i, t_j) + u(x_i - h, t_j)}{h^2}.$$

In terms of the values  $w_{i,j}$  this yields

$$\frac{\partial^2 u}{\partial x^2}(x_i, t_j) \approx \frac{w_{i+1,j} - 2w_{i,j} + w_{i-1,j}}{h^2}.$$

This formula is only valid at an inner point  $1 \leq i \leq m - 1$ . Our next step is to approximate the time derivative. Because of the natural flow of time in the positive direction, it makes sense to use a backward difference method:

$$\frac{\partial u}{\partial t}(x_i, t_j) \approx \frac{u(x_i, t_j) - u(x_i, t_{j-1})}{k}.$$

Although this formula is only of first order (see Theorem 22), its obvious advantage is to only use previously computed values instead of searching in the future. In terms of the grid values  $w_{i,j}$  we obtain

$$\frac{\partial u}{\partial t}(x_i, t_j) \approx \frac{w_{i,j} - w_{i,j-1}}{k}.$$

This formula is valid for  $1 \leq j \leq n$ . All in all the discretized heat equation becomes

$$\frac{w_{i,j} - w_{i,j-1}}{k} = D \frac{w_{i+1,j} - 2w_{i,j} + w_{i-1,j}}{h^2}$$

that is:

$$w_{i,j} - w_{i,j-1} = \frac{Dk}{h^2} (w_{i+1,j} - 2w_{i,j} + w_{i-1,j}).$$

We let  $\sigma = Dk/h^2$  and reorganize the terms to have all values at  $t = t_j$  on one side of the equal sign and  $w_{i,j-1}$  on the other side.

$$-\sigma w_{i-1,j} + (1 + 2\sigma)w_{i,j} - \sigma w_{i+1,j} = w_{i,j-1}$$

which is the discretized heat equation at an inner point  $1 \leq i \leq m - 1, 1 \leq j \leq n$ .

Note that we do not have any condition at  $t = T$ , only an initial value at  $t = 0$ .

We will shortly see that the time discretization needs to be finer to achieve good accuracy - hence it is a good idea to use two step sizes.

Inner meaning  $1 \leq i \leq m - 1$  i.e. not a boundary point.

Since this is a partial derivative in the  $x$  direction, the  $t$  variable stays constant throughout.

This is because  $x_i + h = x_{i+1}$  and  $x_i - h = x_{i-1}$  by definition of the step size.

Notice the different step size  $k$  in the time direction. We can't have  $j = 0$  so we can go one step back in time.

It turns out that using the theoretically superior central difference approximation leads to instability in the long run.

We plug into  $\frac{\partial u}{\partial t} = D \frac{\partial^2 u}{\partial x^2}$ .

We also reorder the  $w_{i,j}$  in order of increasing values of  $i$ . This makes it easier to see the structure of the coefficient matrix  $A$  later on.



### Implementation with Dirichlet boundary conditions

We assume Dirichlet boundary conditions on both sides of the rod, such that the left end is maintained at a higher temperature:

$$u(0, t) = 50 \quad u(L, t) = 20$$

for all  $t \geq 0$ . In terms of the  $w_{i,j}$  this means that

$$w_{0,j} = 50 \quad w_{m,j} = 20$$

for every  $0 \leq j \leq n$ . In addition to the boundary conditions we must know the initial temperature distribution at  $t = 0$ . For simplicity we assume a constant temperature  $u(x, 0) = 20$ . In terms of the  $w_{i,j}$  this means that

$$w_{i,0} = 20$$

for every  $0 \leq i \leq m$ . Because our two conditions do not agree on the value of  $w_{0,0}$  we have a decision to make - in general it is easier to agree on a homogeneous initial condition. This means we set

$$w_{0,0} = 20$$

and the boundary conditions need not apply for  $j = 0$ .

We now use the discretized heat equation to solve for  $w_{i,j}$  one time step at a time. The values for  $w_{i,0}$  are given by the initial values. Setting  $j = 1$  in the heat equation yields

$$-\sigma w_{i-1,1} + (1 + 2\sigma)w_{i,1} - \sigma w_{i+1,1} = w_{i,0} = 20$$

for all  $1 \leq i \leq m - 1$ . This together with the boundary values

$$w_{0,1} = 50 \quad w_{m,1} = 20$$

is a  $(m + 1) \times (m + 1)$  linear system for the  $m + 1$  unknowns  $w_{i,1}$  which we can solve. We write this system as  $Aw(:, 1) = \mathbf{b}_0$  with coefficient matrix

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\sigma & 1 + 2\sigma & -\sigma & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -\sigma & 1 + 2\sigma & -\sigma & 0 & \cdots & 0 & 0 & 0 \\ \vdots & & \ddots & \ddots & \ddots & & & & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & & & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & & & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & -\sigma & 1 + 2\sigma & -\sigma \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 \end{pmatrix}$$

We will consider a more general initial value  $u(x, 0) = f(x)$  in the next example.

Note the initial values and boundary conditions do not have to agree.

We imagine the boundary condition as applied immediately after time starts running.

$w_{i,0} = 20$  is the initial value.

We borrow from Matlab notation and let  $w(:, 1)$  be the vector with coordinates  $w_{0,1}, w_{1,1}, \dots, w_{m,1}$ .

First and last row reflect the boundary conditions  $w_{0,1} = 50$  and  $w_{m,1} = 20$ . The rest of the matrix is tridiagonal.

and right-hand side vector

$$\mathbf{b}_0 = \begin{pmatrix} 50 \\ 20 \\ 20 \\ \vdots \\ 20 \\ 20 \end{pmatrix}.$$

Solving it yields the solution at  $t = t_1$ , that is  $w_{i,1}$  for all  $0 \leq i \leq m$ .

The next iteration uses the heat equation at the next time step  $t_2$

$$-\sigma w_{i-1,2} + (1 + 2\sigma)w_{i,2} - \sigma w_{i+1,2} = w_{i,1}$$

together with the boundary conditions

$$w_{0,2} = 50 \quad w_{m,2} = 20.$$

Since the values for  $w_{i,1}$  are known from the previous step, we can solve for  $w_{i,2}$ . This is again a  $(m+1) \times (m+1)$  linear system

$$A\mathbf{w}(:,2) = \mathbf{b}_1$$

for  $m+1$  unknowns  $w_{0,2}, \dots, w_{m,2}$ . The matrix coefficient  $A$  is the same as before while the right-hand side is updated to

$$\mathbf{b}_1 = \begin{pmatrix} 50 \\ w_{1,1} \\ w_{2,1} \\ \vdots \\ w_{m-1,1} \\ 20 \end{pmatrix}$$

This process goes on until we reach the final time step  $j = n$ . Because the matrix  $A$  stays the same throughout, we can easily code these iterations with a for loop, only updating the right-hand side of the system.

Set  $j = 2$  in the discretized heat equation.

Our boundary conditions are time-independent - in general, they don't need to be.

The general shape of the discretized heat equation stays the same but the right hand side changes with  $j$ .

```

1 function w=heat1(T,m,n)
2     D=4;L=10;
3     h=L/m;k=T/n;sigma=(D*k)/h^2;
4     i=[2:m , 2:m , 2:m]';
5     j=[1:m-1 , 2:m , 3:m+1]';
6     values=[-sigma*ones(m-1,1);(1+2*sigma)*ones(
7         m-1,1);-sigma*ones(m-1,1)];
8     A=sparse(i,j,values);
9     A(1,1)=1;A(m+1,m+1)=1;
10
11    b=20*ones(m+1,1);b(1)=50;
12    w(:,1)=20*ones(m+1,1); %initial value
13
14    for j=1:n
15        w(:,j+1)=A\b;
16        b=[50;w(2:m,j+1);20];
17    end
end

```

Inputs are the length of the time interval  $T$ , as well as the number of steps in both space and time directions.

Lines 4-7 define the tridiagonal part of the coefficient matrix  $A$  as a sparse matrix. Line 8 then corrects the first and last row of  $A$ .

Line 10 defines the first  $b_0$  vector according to the initial values and boundary conditions (for the first and last entry). Line 11 saves the initial values as the first step of the solution.

The **for** loop solves the linear system and changes the value of  $b$  at the next time step. The boundary conditions stay fixed (first and last entry of  $b$ ) while the rest is updated.

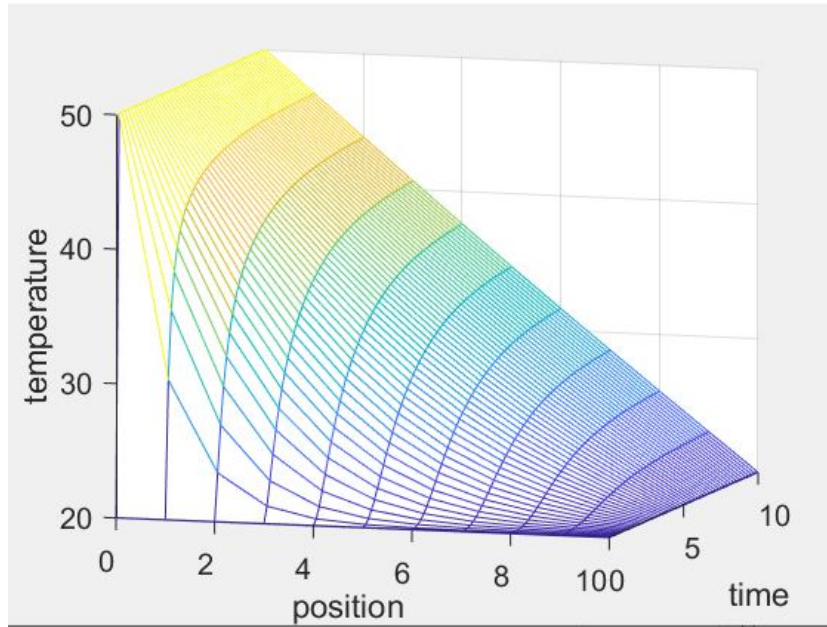
The outcome of this program is a large matrix  $w$ . It has  $m + 1$  rows (space steps) and  $n + 1$  columns (time steps). We can present this data in two ways. First off, we can draw a 3D picture with position on the  $x$ -axis, time on the  $y$ -axis and temperature on the  $z$ -axis using the mesh command:

```

1 L=10;T=10;m=10;n=50;
2 h=L/m;k=T/n;
3 w=heat1(T,m,n);
4 %plot
5 x=(0:m)*h;t=(0:n)*k;
6 mesh(x,t,w');

```

the result of which is displayed on the next page.



Solution to the heat equation on  $[0, 10]$  up to  $T = 10$ . We use  $m = 10$  steps in the space variable and  $n = 50$  in the time variable. Temperature is maintained at 50 degrees for  $x = 0$  and 20 degrees for  $x = 10$ .

The temperature distribution tends to a linear function taking values 50 and 20 at the endpoints, according to the boundary conditions. The time needed to reach this steady state depends on the diffusion coefficient  $D$ , but in our case we have almost reached this steady state when  $t = 10$ .

Another way to present the data is as an animated plot, showing how the temperature distribution along the rod changes with time, for example using the *animatedline* command.

```

1  tracerod = animatedline('Color','b','LineWidth'
    ,1);
2  for j = 1:n+1
3      clearpoints(tracerod)
4      for i=1:m+1
5          addpoints(tracerod,x(i),w(i,j));
6      end
7      %pause(0.1) use to slow down animation
8      drawnow
9  end

```

$m$ ,  $n$  and the matrix  $w$  are inherited from the main solver.

Line 1 defines the graph. We loop through all time steps (using  $jc$ ). At each time step we clear the figure on Line 3. Line 5 then adds all temperature data points  $w_{i,j}$  at the given time step.

The pause command may be used in order to slow down the animation if needed.

The reader is welcome to run this code on their own computer to see the resulting animation.

### Implementation with Neumann boundary conditions

We do two changes to the previous setup. First off we consider a general initial condition

$$u(x, 0) = f(x)$$

Instead of  $f(x) = \text{constant}$  which we used previously.

where  $f$  is a given function which describes the temperature along the rod at  $t = 0$ . Since the rod is discretized this amounts to prescribing  $m + 1$  values at  $x = x_i$  for  $0 \leq i \leq m$ :

$$w_{i,0} = f(x_i).$$

Second of all we will consider Neumann boundary conditions at  $x = 0$  and  $x = L$ . The conditions

$$\frac{\partial u}{\partial x}(0, t) = \frac{\partial u}{\partial x}(L, t) = 0$$

model a perfectly insulated rod. As in the one-dimensional case we will use three-points forward and backward difference method of Definition 22. At  $x = 0$  we use the formula

$$0 = \frac{\partial u}{\partial x}(0, t) = \frac{-3u(0, t) + 4u(h, t) - u(2h, t)}{2h}$$

which in terms of  $w_{i,j}$  means

$$0 = \frac{-3w_{0,j} + 4w_{1,j} - w_{2,j}}{2h}$$

and should hold for  $1 \leq j \leq n$ . Clearly we can cancel out  $2h$  and obtain

$$-3w_{0,j} + 4w_{1,j} - w_{2,j} = 0.$$

A similar analysis using

$$0 = \frac{\partial u}{\partial x}(L, t) = \frac{3u(L, t) - 4u(L - h, t) + u(L - 2h, t)}{2h}$$

yields the equation

$$w_{m-2,j} - 4w_{m-1,j} + 3w_{m,j} = 0$$

which holds for all  $1 \leq j \leq n$ . The discretized heat equation at an inner point stays the same as during the previous calculations, that is:

$$-\sigma w_{i-1,j} + (1 + 2\sigma)w_{i,j} - \sigma w_{i+1,j} = w_{i,j-1}$$

for  $1 \leq i \leq m - 1, 1 \leq j \leq n$ .

The iterative process functions as earlier. For  $j = 0$  the solution is given by the initial data:

$$w_{i,0} = f(x_i)$$

For  $j = 1$  the discretized heat equation gives  $m - 1$  equations

$$-\sigma w_{i-1,1} + (1 + 2\sigma)w_{i,1} - \sigma w_{i+1,1} = w_{i,0}$$

to which we add the two boundary conditions

Under these conditions we expect the temperature distribution to converge to a steady state where the temperature is everywhere the same.

We use the forward formula at  $x = 0$  in order to enter the rod towards the right.

Again we assume the boundary condition holds only when time starts running, otherwise we run into conflict with the initial condition. Hence the boundary condition isn't assumed to be true at  $t = 0$  that is  $j = 0$ .

At  $x = L$  we must use backward difference formula in order to enter the rod from the right.

We reorder the terms in order of ascending  $i$  - this helps us see the structure of the coefficient matrix  $A$ .

Recall  $\sigma = Dk/h^2$  is determined by our choice of  $m$  and  $n$ .

Both Neumann conditions from earlier with  $j = 1$ .

$$-3w_{0,1} + 4w_{1,1} - w_{2,1} = 0 \quad w_{m-2,1} - 4w_{m-1,1} + 3w_{m,1} = 0.$$

This is a total of  $m+1$  equations for the  $m+1$  unknowns  $w_{0,1}, \dots, w_{m,1}$ , that is a linear system

$$Aw(:,1) = b_0$$

The coefficient matrix  $A$  is almost the same as in the Dirichlet case, except we need to change its first and last row to accommodate the boundary conditions.

$$A = \begin{pmatrix} -3 & 4 & -1 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\sigma & 1+2\sigma & -\sigma & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -\sigma & 1+2\sigma & -\sigma & 0 & \cdots & 0 & 0 & 0 \\ \vdots & & \ddots & \ddots & \ddots & & & & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & & & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & & & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & -\sigma & 1+2\sigma & -\sigma \\ 0 & 0 & 0 & 0 & 0 & \cdots & 1 & -4 & 3 \end{pmatrix}$$

The vector  $b_0$  is determined by

$$b_0 = \begin{pmatrix} 0 \\ w_{1,0} \\ w_{2,0} \\ \vdots \\ w_{m-1,0} \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ f(x_1) \\ f(x_2) \\ \vdots \\ f(x_{m-1}) \\ 0 \end{pmatrix}$$

and will be updated along with time. Solving the system yields the solution at  $t = t_1$  that is  $w_{i,1}$  for all  $0 \leq i \leq m$ .

For the next iteration we again turn to the discretized heat equation

$$-\sigma w_{i-1,2} + (1+2\sigma)w_{i,2} - \sigma w_{i+1,2} = w_{i,1}$$

together with the Neumann boundary conditions

$$-3w_{0,2} + 4w_{1,2} - w_{2,2} = 0 \quad w_{m-2,2} - 4w_{m-1,2} + 3w_{m,2} = 0$$

Because the  $w_{i,1}$  have been previously computed, this is a linear  $(m+1) \times (m+1)$  system  $Aw(:,2) = b_1$  for the unknowns  $w_{i,2}$ . The

Top row reflects the equation

$$-3w_{0,1} + 4w_{1,1} - w_{2,1} = 0$$

and bottom row reflects

$$w_{m-2,1} - 4w_{m-1,1} + 3w_{m,1} = 0.$$

We could multiply either one by  $-1$ .

Again  $A$  is tridiagonal if we exclude the top and bottom row.

Recall that the  $w_{i,0}$  are determined by the initial condition  $u(x,0) = f(x)$ . That is  $w_{i,0} = f(x_i)$ .

The two zeroes stem from the boundary equations

$$-3w_{0,1} + 4w_{1,1} - w_{2,1} = 0$$

and

$$w_{m-2,1} - 4w_{m-1,1} + 3w_{m,1} = 0.$$

This step is exactly the same in all situations as it accounts for neither boundary nor initial conditions.

coefficient matrix  $A$  is the same as before, however we now have

$$\mathbf{b}_1 = \begin{pmatrix} 0 \\ w_{1,1} \\ w_{2,1} \\ \vdots \\ w_{m-1,1} \\ 0 \end{pmatrix}$$

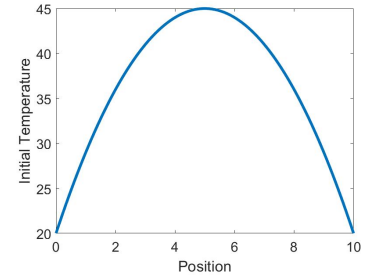
Solving this system yields the  $w_{i,2}$  and the next iteration can then be solved in a similar manner. The function  $f(x)$  which determines the initial condition can simply be defined in the same program in Matlab. As an example we use the function

$$f(x) = 20 + x(L - x), 0 \leq x \leq L.$$

The initial temperature distribution is shown on Figure 64.

The whole problem can be solved through a for loop.

Recall that the  $w_{i,1}$  are known as a result of the first iteration. Again both zeroes stem from our Neumann boundary conditions.



**Figure 64:** Initial temperature distribution  $f(x) = 20 + x(L - x)$  for  $L = 10$ .

```

1 function w=heat2(T,m,n)
2     D=4;L=10;
3     h=L/m;k=T/n;sigma=(D*k)/h^2;
4     i=[2:m , 2:m , 2:m]';
5     j=[1:m-1 , 2:m , 3:m+1]';
6     values=[-sigma*ones(m-1,1);(1+2*sigma)*ones(
7         m-1,1);-sigma*ones(m-1,1)];
8     A=sparse(i,j,values);
9
10    A(1,1)=-3;A(1,2)=4;A(1,3)=-1; %Neumann cond.
11
12    A(m+1,m-1)=1;A(m+1,m)=-4;A(m+1,m+1)=3;
13
14    x=linspace(0,L,m+1); %initial value
15    w(:,1)=f(x);
16    b=[0;w(2:m,1);0]
17
18    for j=1:n
19        w(:,j+1)=A\b;
20        b=[0;w(2:m,j+1);0];
21    end
22 end
23
24 function y=f(x)
25     L=10;
26     y=20+x.*(L-x);
27 end

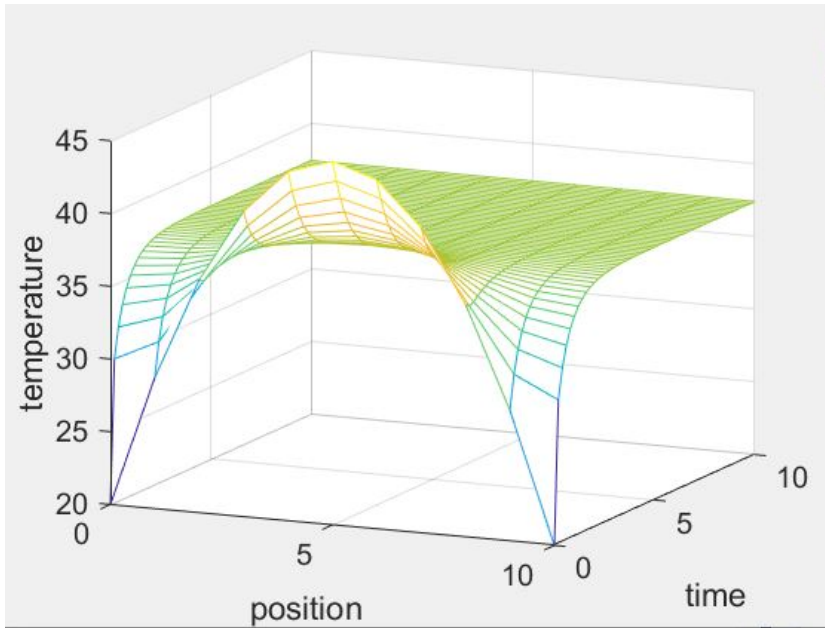
```

Lines 9-10 account for the Neumann boundary conditions by coding the top and bottom row of the matrix  $A$  manually.

Lines 12-14 define the first iteration vector  $\mathbf{b}_0$ . First and last entry are 0, the rest is  $w_{i,0} = f(x_i)$ . The function  $f$  which describes the initial temperature distribution is defined in Lines 22-25.

Lines 17-18 update the vector  $\mathbf{b}_j$  and solve the resulting linear system. First and last entry of the vector are always 0 according to our Neumann conditions.

The data contained in the matrix  $w$  can then be displayed in two ways exactly as before, first as a 3D plot, and also as an animated plot. The code is virtually the same as earlier, except we need to call the `heat2` function to obtain the data.



**Figure 65:** Solution to the heat equation on  $[0, 10]$  up to  $T = 10$ . We use  $m = 10$  steps in the space variable and  $n = 50$  in the time variable. Initial temperature distribution is shown on Figure 64. The rod is insulated at both ends (Neumann condition).

At  $t = 0$  we clearly see the initial temperature distribution displayed on Figure 64. As time runs the system evolves towards a steady state of constant temperature visible at the far end of the plot. The rod is insulated so no energy can exit the system. The limit value can be computed beforehand as the mean value of the initial temperature distribution. The speed at which we converge to the steady state depends on the diffusion constant  $D$ .

### *Crank-Nicholson method*

Our numerical method relies heavily on the backward difference formula

$$\frac{\partial u}{\partial t}(x_i, t_j) \approx \frac{u(x_i, t_j) - u(x_i, t_{j-1}))}{k}$$

in order to discretize the heat equation in the time direction. As we learned this formula is rather inaccurate, and according to Theorem 22 the error is  $O(k)$ . Since all other involved formulas are of order 2, this means that  $k$  needs to be much smaller than  $h$  in order to maintain accuracy. The so-called Crank-Nicholson method fixes the accuracy and lets the error be of order 2 in both time and space. The formulas are however slightly more complicated.

Too small values of  $k$  can then lead to other problems, for instance floating-point errors when dividing by  $k$ .

We go on using the backward difference formula

$$\frac{\partial u}{\partial t}(x_i, t_j) \approx \frac{w_{i,j} - w_{i,j-1}}{k}$$



while the second-derivative term is approximated by a mixed formula

$$\frac{\partial^2 u}{\partial x^2}(x_i, t_j) \approx \frac{1}{2} \frac{w_{i+1,j} - 2w_{i,j} + w_{i-1,j}}{h^2} + \frac{1}{2} \frac{w_{i+1,j-1} - 2w_{i,j-1} + w_{i-1,j-1}}{h^2}$$

Plugging these formulas into the heat equation then leads to

$$\frac{w_{i,j} - w_{i,j-1}}{k} = \frac{D}{2} \frac{w_{i+1,j} - 2w_{i,j} + w_{i-1,j}}{h^2} + \frac{D}{2} \frac{w_{i+1,j-1} - 2w_{i,j-1} + w_{i-1,j-1}}{h^2}$$

Again setting  $\sigma = Dk/h^2$  we end up with a new discretized heat equation at an inner point where  $1 \leq i \leq m-1$ ,  $0 \leq j \leq n$ :

$$-\sigma w_{i-1,j} + (2+2\sigma)w_{i,j} - \sigma w_{i+1,j} = \sigma w_{i-1,j-1} + (2-2\sigma)w_{i,j-1} + \sigma w_{i+1,j-1}$$

The right-hand side is always known from the previous iteration. We consider the Dirichlet boundary conditions

$$w_{0,j} = 50 \quad w_{m,j} = 20$$

of our first example, along with the initial condition

$$w_{i,0} = 20$$

This leads to a linear system  $A\mathbf{w}(:,j) = \mathbf{b}_{j-1}$  where

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 \\ -\sigma & 2+2\sigma & -\sigma & 0 & 0 & \cdots & 0 & 0 & 0 \\ 0 & -\sigma & 2+2\sigma & -\sigma & 0 & \cdots & 0 & 0 & 0 \\ \vdots & & \ddots & \ddots & \ddots & & & & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & & & \vdots \\ \vdots & & & & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & & & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \cdots & -\sigma & 2+2\sigma & -\sigma \\ 0 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 \end{pmatrix}$$

and

$$\mathbf{b}_{j-1} = \begin{pmatrix} 50 \\ \sigma w_{0,j-1} + (2-2\sigma)w_{1,j-1} + \sigma w_{2,j-1} \\ \sigma w_{1,j-1} + (2-2\sigma)w_{2,j-1} + \sigma w_{3,j-1} \\ \vdots \\ \sigma w_{m-1,j-1} + (2-2\sigma)w_{m,j-1} + \sigma w_{m+1,j-1} \\ 20 \end{pmatrix}.$$

The values of  $w_{i,0}$  are determined by the initial conditions. This leads to the following code:

That is  $\frac{\partial u}{\partial t} = D \frac{\partial^2 u}{\partial x^2}$ .

As before we gather all terms a time  $t_{j-1}$  on one side of the equation, and terms at  $t = t_j$  on the other side.

For  $j = 1$  it is obtained from the initial condition.  
Where  $0 \leq j \leq n$

Where  $0 \leq i \leq m$ . We agree upon  $w_{0,0} = 20$ .

```

1 function w=heatCN(T,m,n)
2     D=4;L=10;
3     h=L/m;k=T/n;sigma=(D*k)/h^2;
4     i=[2:m , 2:m , 2:m]';
5     j=[1:m-1 , 2:m , 3:m+1]';
6     values=[-sigma*ones(m-1,1);(2+2*sigma)*ones(m-1,1);-sigma*ones(m-1,1)];
7     A=sparse(i,j,values);
8     A(1,1)=1;A(m+1,m+1)=1;
9
10    w(:,1)=20*ones(m+1,1); %initial value
11    b(1)=50;
12    b(2:m)=sigma*w(1:m-1,1)+(2-2*sigma)*w(2:m,1)
13    +sigma*w(3:m+1,1);
14    b(m+1)=20;b=b';
15
16    for j=1:n
17        w(:,j+1)=A\b;
18        b(1)=50;
19        b(2:m)=sigma*w(1:m-1,j+1)+(2-2*sigma)*w(2:m,j+1)+sigma*w(3:m+1,j+1);
20        b(m+1)=20;
21    end
end

```

This code is mostly the same as **heat1**. The matrix  $A$  has slightly different coefficients which are coded on Line 6. The main difference is the structure of the right-hand side  $b$ .

Other types of initial and/or boundary conditions can be handled as they were earlier.

### *The Poisson equation - elliptic PDEs*

We now consider the Poisson equation in two spatial dimensions  $x$  and  $y$ :

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y)$$

on a rectangle  $a \leq x \leq b, c \leq y \leq d$ , where  $f(x, y)$  is a given function. The main difference to the heat equation is that the Poisson equation is of second degree in both variables  $x$  and  $y$ . This forces us to use a central difference formula in both directions and does not allow for a simple iterative process. Because we need to find all discretized values at once, we need a new discretization scheme in both space directions.

Poisson's equation describes a vast array of physical phenomena in electrostatics and fluid dynamics.

We will also need boundary on all four walls of the rectangle in order to solve uniquely for  $u(x, y)$ . Since the equation is independent of time there is no need for initial conditions

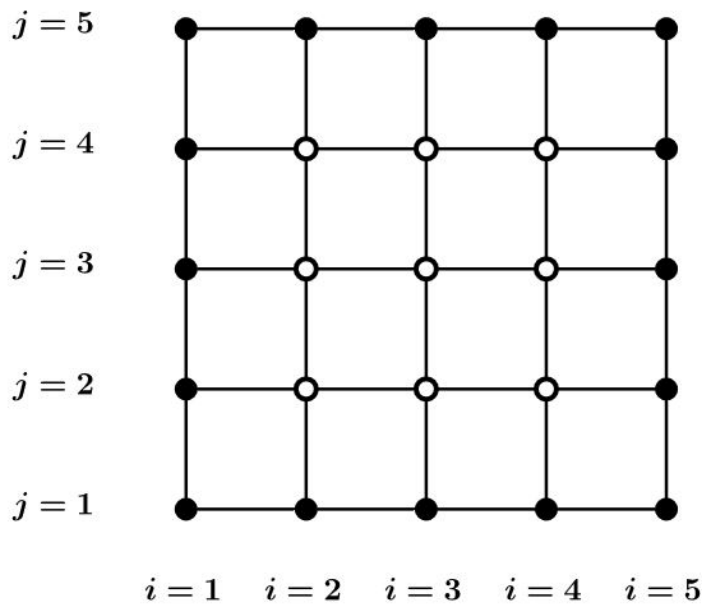
Indeed there is no preferred direction for either  $x$  and  $y$ , so it does not make sense to progress forward in time.

*discretization in two space directions*

We discretize in the  $x$  direction using  $m - 1$  steps such that with step size  $h_x = (b - a)/(m - 1)$ . Similarly the  $y$ -axis is discretized in  $n - 1$  steps with step size  $h_y = (d - c)/(n - 1)$ . Our objective is to calculate an approximated value for the solution at all points of the grid. However an annoying technicality arises: we do not particularly want to number the grid points using two indices  $i$  and  $j$  - rather we need them to be arranged in a single vector so that we can solve a single system for all unknowns at once. There are many possible ways to do so, and a common solution is to order the grid points in lexicographical order. The mechanism is easy to explain on a small example where  $m = n = 5$ .

It will be much simpler later on to have  $m$  points rather than  $m$  steps.

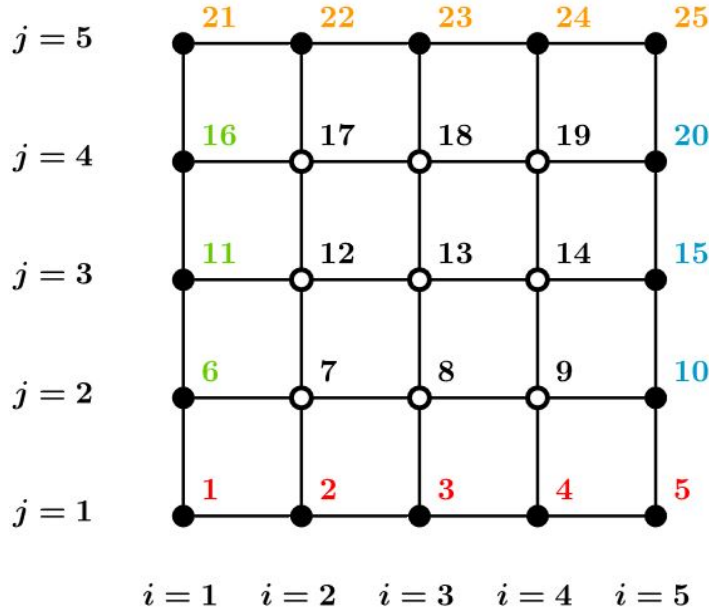
For instance bottom to top and left to right.



discretized grid with  $m = n = 5$  steps in each direction. Solid black dots are boundary points while the open dots are inner points of the domain.

Instead of numbering these  $mn = 25$  points by their  $xy$  coordinates, we will start numbering at the bottom left, going along one row, and then moving up and back to the left in a snakelike trajectory. The next figure shows an updated grid with lexicographical numbering.

Using double coordinates such as  $(2, 3)$  would prove complicated when wanted to set up a linear system for the 25 unknowns.



Lexicographical ordering of the grid vertices with respect to a left to right and bottom to top ordering. Different types of boundary conditions are indicated in different colors. There is some leeway as to whether the corners belong to a vertical or a horizontal boundary. Our choice is that they belong to the top and bottom boundaries.

For unspecified values of  $m$  and  $n$ , the ordering is shown in the table below.

$(n-1)m+1$	$(n-1)m+2$	$\cdots$	$(n-1)m+i$	$\cdots$	$nm$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$jm+1$	$jm+2$	$\cdots$	$jm+i$	$\cdots$	$(j+1)m$
$(j-1)m+1$	$(j-1)m+2$	$\cdots$	$(j-1)m+i$	$\cdots$	$jm$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$2m+1$	$2m+2$	$\cdots$	$2m+i$	$\cdots$	$3m$
$m+1$	$m+2$	$\cdots$	$m+i$	$\cdots$	$2m$
1	2	$\cdots$	$i$	$\cdots$	$m$

**Table 11:** Lexicographical ordering of a  $m \times n$  grid.

Our objective is now to find the solution to Poisson's equation at all  $m \times n$  points. The approximate value to the solution will be denoted by  $v_k$ ,  $1 \leq k \leq mn$ . A clear disadvantage of this numbering is that we lose all insight connecting coordinates to actual physical position. First off we need to locate where boundary points and inner points.

- Bottom boundary conditions at indices  $1 \leq k \leq m$ .
- Top boundary conditions at indices,  $(n-1)m+1 \leq k \leq nm$ .

A final step in the implementation will be to restore the grid shape of these  $mn$  unknowns which are computed as a single vector.

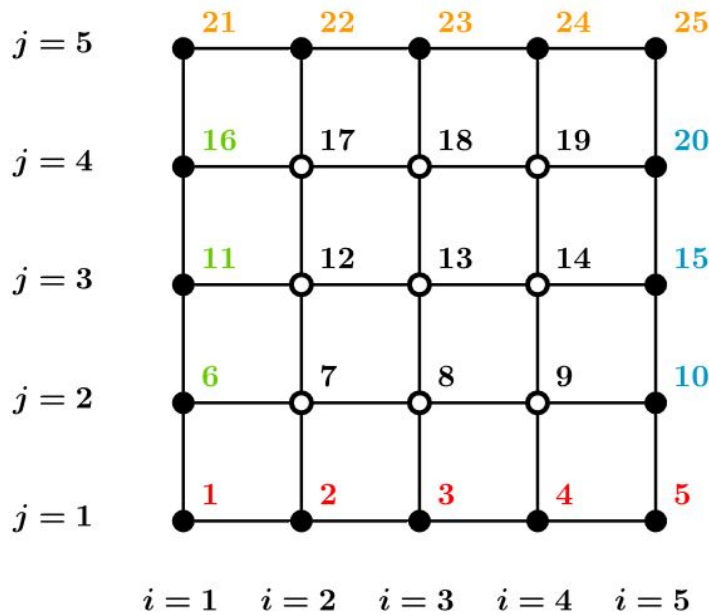
In total there are  $m$  bottom and  $m$  top boundary points, as well as  $n-2$  left and  $n-2$  right boundary points. The rest constitutes inner points, that is a total of  $mn - 2m - 2(n-2)$  inner points.

- Left boundary conditions at indices  $k = (j - 1)m + 1$  where  $2 \leq j \leq n - 1$ .
- Right boundary conditions at indices  $k = jm$  where  $2 \leq n - 1$ .
- The indices which are left describe inner points.

Secondly in order to efficiently implement finite difference methods, we should know what it means to move one step in any direction in terms of the lexicographical numbering.

- Moving right is an increase from  $k$  to  $k + 1$ .
- Moving left is a decrease from  $k$  to  $k - 1$ .
- Moving up is an increase from  $k$  to  $k + m$ .
- Moving down is a decrease from  $k$  to  $k - m$ .

Finally we need a way to converge the single index  $k$  into the  $(x, y)$  coordinates of the point. Let us again study the  $5 \times 5$  grid as an example:



Consider for instance the index  $k = 14$ . We compute the two integers

$$\text{mod}(14, 5) = 4 \quad \text{ceil}(14/5) = 3$$

retrieving the coordinates  $i$  and  $j$  in the process. We run into a slight problem at the right boundary, say  $k = 15$  as Matlab would compute

$$\text{mod}(15, 5) = 0 \quad \text{ceil}(15/5) = 3$$

Corners can be thought of belonging to either type of boundary coefficient. Usually this means little trouble but there might be physical reasons for preferring one type rather than the other.

Of course in some cases such movements are illegal - we end up outside of the grid. Recall that we use different approximations at boundary points to precisely avoid this.

Mainly this is so we can handle the right-hand side  $f(x, y)$  of Poisson's equation. Boundary conditions may be position-dependent as well.

**mod(a,b)** is the remainder of the division of  $a$  by  $b$ . **ceil(a/b)** is the nearest integer greater or equal than  $a/b$ . Both functions are native in Matlab.

which forces us to manually convert a 0 remainder to  $i = 5$ . A final rather straightforward step converts  $(i, j)$  into actual  $x$  and  $y$  coordinates.

```

1 function [x,y]=findposition(k,a,c,hx,hy,m)
2     i=mod(k,m); j=ceil(k/m);
3     if i==0
4         i=m;
5     end
6     x=a+(i-1)*hx;
7     y=c+(j-1)*hy;
8 end

```

In general to  $i = m$ .

For this subroutine we need to carry over physical information about the grid, that is the rectangle boundaries  $x = a, y = c$  as well as  $m$  and the size steps.

This dictionary will prove very useful in the next steps. From now on we completely forget about the  $(i, j)$  coordinates and use the **findposition** subroutine when it is necessary to compute a position from the index  $k$ .

### Inner points

We begin by discretizing the Poisson equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x, y)$$

at an inner point of the grid. This is done independently of any boundary conditions we might have. Both second derivatives are approximated using the central difference approximation of Definition 23:

$$\frac{\partial^2 u}{\partial x^2}(x, y) \approx \frac{u(x+h, y) - 2u(x, y) + u(x-h, y)}{h_x^2}$$

$$\frac{\partial^2 u}{\partial y^2}(x, y) \approx \frac{u(x, y+k) - 2u(x, y) + u(x, y-k)}{h_y^2}$$

The  $x$  second derivative moves directly to the left and to the right, while the  $y$  second derivative moves up and down. We have characterized these movements just above. At a given inner point  $k$  we may therefore use the approximations

$$\frac{\partial^2 u}{\partial x^2}(x_k, y_k) \approx \frac{v_{k+1} - 2v_k + v_{k-1}}{h_x^2}$$

$$\frac{\partial^2 u}{\partial y^2}(x_k, y_k) \approx \frac{v_{k+m} - 2v_k + v_{k-m}}{h_y^2}$$

Recall we have different step sizes  $h_x$  and  $h_y$  along the  $x$  and  $y$  axis.

This was established in the dictionary.

Recall that  $v_k$  is the approximated solution at the point with lexicographical number  $k$ . We let  $(x_k, y_k)$  be the corresponding point in the 2D grid, that is the output of the **findposition** subroutine.

Poisson's equation therefore becomes

$$\frac{v_{k+1} - 2v_k + v_{k-1}}{h_x^2} + \frac{v_{k+m} - 2v_k + v_{k-m}}{h_y^2} = f(x_k, y_k)$$

We regroup the terms involving  $v_k$ , arrange terms in increasing index order and obtain:

$$\frac{1}{h_y^2}v_{k-m} + \frac{1}{h_x^2}v_{k-1} - \left(\frac{2}{h_x^2} + \frac{2}{h_y^2}\right)v_k + \frac{1}{h_x^2}v_{k+1} + \frac{1}{h_y^2}v_{k+m} = f(x_k, y_k)$$

which is the discretized Poisson's equation at an inner point of the grid. The equation holds for any  $k$  between 1 and  $mn$  which isn't attached to any boundary. In the rather common special case where  $h_x = h_y = h$ , it pays off to multiply both sides by  $h^2$  to obtain the much simpler:

$$v_{k-m} + v_{k-1} - 4v_k + v_{k+1} + v_{k+m} = h^2 f(x_k, y_k)$$

We have worked out precisely which indices  $k$  belong to the boundary.

### Dirichlet boundary conditions

We now assume Dirichlet boundary conditions on all four sides of the rectangle. For simplicity, let us say that we have

$$u(x, y) = 0 \quad \text{on the left side } x = a$$

$$u(x, y) = 0 \quad \text{on the right side } x = b$$

$$u(x, y) = 0 \quad \text{on the top side } y = d$$

while

$$u(x, y) = \varphi(x) \quad \text{on the bottom side } y = c$$

where  $\varphi$  is a known function of  $x$ . Our first task is to translate these conditions into equations for the  $v_k$ .

- Left boundary corresponds to  $k = (j-1)m + 1$  where  $2 \leq j \leq n-1$ . For such  $k$  we have  $v_k = 0$ . We have  $u(a, y) = 0$ .
- Right boundary corresponds to  $k = jm$  where  $2 \leq j \leq n-1$ . Again we have  $v_k = 0$  for such  $k$ . We have  $u(b, y) = 0$ .
- Top boundary corresponds to  $(n-1)m + 1 \leq k \leq mn$ . For such  $k$  we have  $v_k = 0$ . We have  $u(x, d) = 0$ .
- Bottom boundary corresponds to  $1 \leq k \leq m$ . For such  $k$  we have  $v_k = \varphi(x_k)$  where  $x_k$  is the corresponding  $x$  coordinate according to the **findposition** subroutine. We have  $u(x, c) = \varphi(x)$ .

We now have a complete  $(mn) \times (mn)$  linear system for the unknowns  $v_k$ . The structure of the  $(mn) \times (mn)$  coefficient matrix  $A$  is somewhat complicated. As an example, consider the case  $m = n = 4$  and assume also  $h_x = h_y = h$  for simplicity. We now need to arrange the equations in order of increasing  $k$  rather than by boundary type.

- First four equations come from the bottom boundary,  $k = 1, 2, 3, 4$ .

$$\begin{cases} v_1 = \varphi(x_1) \\ v_2 = \varphi(x_2) \\ v_3 = \varphi(x_3) \\ v_4 = \varphi(x_4) \end{cases}$$

- The fifth equation is at the left boundary:

$$v_5 = 0$$

- Equations 6 and 7 concern inner points. We use our general equation with  $k = 6$  and  $k = 7$  (and  $m = 4$ ).

$$v_2 + v_5 - 4v_6 + v_7 + v_{10} = h^2 f(x_6, y_6)$$

$$v_3 + v_6 - 4v_7 + v_8 + v_{11} = h^2 f(x_7, y_7)$$

- Equation 8 is at the right boundary:

$$v_8 = 0$$

- Equations 9 through 12 repeat this pattern, now with  $k = 9, 10, 11, 12$ :

$$v_9 = 0$$

$$v_6 + v_9 - 4v_{10} + v_{11} + v_{14} = h^2 f(x_{10}, y_{10})$$

$$v_7 + v_{10} - 4v_{11} + v_{12} + v_{15} = h^2 f(x_{11}, y_{11})$$

$$v_{12} = 0$$

- Final four equations are the top boundary:

$$v_{13} = 0$$

$$v_{14} = 0$$

$$v_{15} = 0$$

$$v_{16} = 0$$

We have established one equation for each point of the  $(mn) \times (mn)$  grid. The inner equations were dealt with in the previous subsection.

13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

**Table 12:** Lexicographical ordering of a  $4 \times 4$  grid with boundary types indicated by different colors. The four inner points are in black.

General form of the equation is

$$v_{k-m} + v_{k-1} - 4v_k + v_{k+1} + v_{k+m} = h^2 f(x_k, y_k).$$

In the general case where  $h_x \neq h_y$  these equations rather become

$$\begin{aligned} \frac{1}{h_y^2} v_2 + \frac{1}{h_x^2} v_5 - \left( \frac{2}{h_x^2} + \frac{2}{h_y^2} \right) v_6 \cdots \\ \cdots + \frac{1}{h_x^2} v_7 + \frac{1}{h_y^2} v_8 = f(x_6, y_6) \end{aligned}$$

and

$$\begin{aligned} \frac{1}{h_y^2} v_3 + \frac{1}{h_x^2} v_6 - \left( \frac{2}{h_x^2} + \frac{2}{h_y^2} \right) v_7 \cdots \\ \cdots + \frac{1}{h_x^2} v_8 + \frac{1}{h_y^2} v_9 = f(x_7, y_7) \end{aligned}$$

The general form  $h_x \neq h_y$  of the inner equations corresponding to  $k = 10$  and  $k = 11$  goes as above.



Our coefficient matrix for  $m = n = 4$  is therefore

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -4 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

while the right-hand side vector is

$$\mathbf{b} = \begin{pmatrix} \varphi(x_1) \\ \varphi(x_2) \\ \varphi(x_3) \\ \varphi(x_4) \\ 0 \\ h^2 f(x_6, y_6) \\ h^2 f(x_7, y_7) \\ 0 \\ 0 \\ h^2 f(x_{10}, y_{10}) \\ h^2 f(x_{11}, y_{11}) \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

We code  $A$  and  $\mathbf{b}$  using a series of **for** loops. We consider immediately the more general case where  $h_x$  and  $h_y$  are not necessarily equal and refer to the notes in the margin above for explanations.

It is a  $16 \times 16$  matrix since we have  $mn = 16$  unknowns  $v_1, \dots, v_{16}$ .

In the case where  $h_x \neq h_y$  the non-zero coefficients on the rows corresponding to inner points ( $k = 6, 7, 10, 11$ ) would rather be:

$$1/h_y^2 \quad 1/h_x^2 \quad -2/h_x^2 - 2/h_y^2 \quad 1/h_x^2 \quad 1/h_y^2$$

instead of

$$1 \quad 1 \quad -4 \quad 1 \quad 1.$$

We skip over the zeroes in this explanation.

$\mathbf{b}$  is a vector with 16 components and fits to the size of  $A$ .

In the general case  $h_x \neq h_y$ , our equations become e.g.

$$\begin{aligned} \frac{1}{h_y^2} v_2 + \frac{1}{h_x^2} v_5 - \left( \frac{2}{h_x^2} + \frac{2}{h_y^2} \right) v_6 \cdots \\ \cdots + \frac{1}{h_x^2} v_7 + \frac{1}{h_y^2} v_8 = f(x_6, y_6) \end{aligned}$$

so we would need to drop the  $h^2$  factor on rows 6, 7, 10 and 11.

The following code constructs both  $A$  and  $b$ .

```

1 function [A,b]=constructmatrix(m,n,hx,hy,a,c)
2     A=zeros(m*n,m*n);b=zeros(m*n,1);
3
4     %bottom boundary
5     for k=1:m
6         A(k,k)=1;
7         [x,y]=findposition(k,a,c,hx,hy,m);
8         b(k)=phi(x);
9     end
10
11    %top boundary
12    for k=(n-1)*m+1:m*n
13        A(k,k)=1;
14        b(k)=0;
15    end
16
17    %left and right boundaries
18    for j=2:n-1
19        k=(j-1)*m+1;
20        A(k,k)=1;
21        b(k)=0;
22        k=j*m;
23        A(k,k)=1;
24        b(k)=0;
25    end
26
27    %inner points - jump over boundaries
28    for k=m+1:(n-1)*m
29        if mod(k,m)>1
30            A(k,k-m)=1/hy^2;
31            A(k,k-1)=1/hx^2;
32            A(k,k)=-2/hx^2-2/hy^2;
33            A(k,k+1)=1/(hx^2);
34            A(k,k+m)=1/(hy^2);
35            [x,y]=findposition(k,a,c,hx,hy,m);
36            b(k)=f(x,y);
37        end
38    end
39 end

```

Solving the Poisson equation subject to our boundary conditions is then achieved simply by solving the linear system  $Aw = b$ . However

Inputs are grid sizes  $m, n$ , step sizes  $h_x$  and  $h_y$  as well as endpoints of the grid  $a$  and  $c$ . These are needed for the **findposition** subroutine.

Boundary points are considered first. We refer to the dictionary for their characterization.

Bottom boundary corresponds to  $1 \leq k \leq m$ . Our equations there are of the form

$$v_k = \phi(x_k).$$

The **findposition** subroutine is used in order to compute  $x_k$  which is then plugged into the function  $\phi$ .

Top boundary corresponds to  $(n-1)m+1 \leq k \leq mn$ . Our equations there are of the form  $v_k = 0$ .

Left boundary are indices  $k = (j-1)m+1, 2 \leq j \leq n-1$ . Our equations there are of the form  $v_k = 0$ .

Right boundary are indices  $k = jm, 2 \leq j \leq n-1$ . Our equations there are of the form  $v_k = 0$ . It is simpler to treat left and right boundaries in one loop.

The main loop defines the equations at inner points. We need to be careful not to overwrite the boundary equations. Two measures ensure that:

- Starting at  $k = m+1$  skips the bottom boundary and stopping at  $k = (n-1)m$  skips the top boundary, see Table 11.
- A value  $\text{mod}(k, m) = 0$  means a index  $k$  on the right boundary. Likewise if  $\text{mod}(k, m) = 1$  we are at the left boundary. We skip both cases using an if condition.

If done correctly we should visit each value of  $k$  exactly once - something we can check easily with a counter.

The **findposition** subroutine is again used in order to compute  $(x_k, y_k)$  at an inner point and plug into the right-hand side  $f(x_k, y_k)$ .

We assume both functions  $f(x, y)$  and  $\phi(x)$  to be defined somewhere else.

the data will come out as a vector with  $mn$  components and we still need to shape it as a two-dimensional grid for plotting purposes. The **reshape** command in Matlab does precisely that. As an example we shall solve Poisson's equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -(2 + \pi^2 x(1-x)) \cos(xy)$$

on the rectangle  $0 \leq x \leq 1, 0 \leq y \leq 0.5$ , subject to four Dirichlet boundary conditions

$$u(x, 0) = x(1-x)$$

$$u(x, 1) = 0$$

$$u(0, y) = 0$$

$$u(0.5, y) = 0$$

The right-hand side is what we have called  $f(x, y)$  so far.

We use  $\varphi(x) = x(1-x)$ . Boundary conditions are listed in the following order: bottom, top, left, right.

Both functions  $f$  and  $\varphi$  need to be defined:

```

1 function z=f(x,y)
2 z=-(2+pi^2*x*(1-x))*cos(pi*y);
3 end
4
5 function z=phi(x)
6 z=x*(1-x);
7 end

```

The main code is then very short.

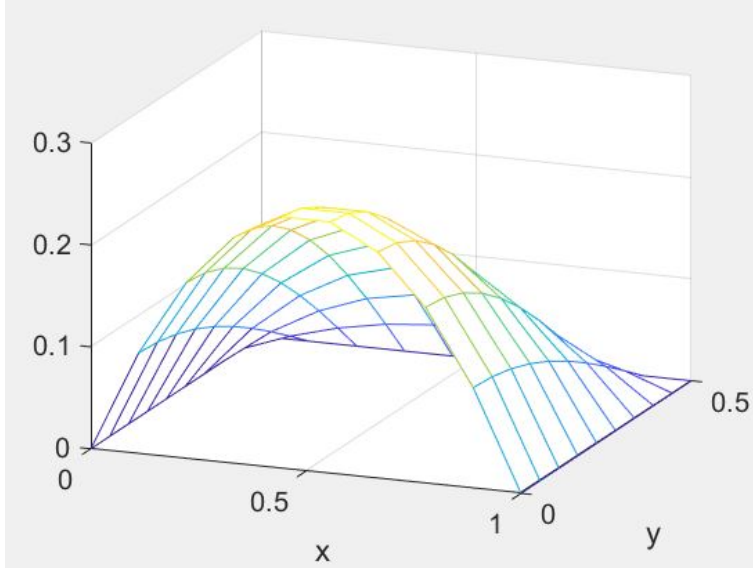
```

1 m=10;n=10;
2 a=0;b=1;c=0;d=0.5;
3 hx=(b-a)/(m-1);hy=(d-c)/(n-1);
4 [A,b]=constructmatrix(m,n,hx,hy,a,c);
5 w=A\b;
6 %plot
7 u=reshape(w,m,n);
8 x=linspace(a,b,m);
9 y=linspace(c,d,n);
10 mesh(x,y,u')

```

Values for  $m$  and  $n$  are rather arbitrary here. More on errors and speed below.

from which we obtain a two-dimensional plot of the solution  $u(x, y)$ .



### Error analysis and speed issues

Since the finite difference method relies on the second order central difference approximation (see Theorem 23), we fully expect it to be of second order as well. For simplicity we assume  $m = n$  and run the program for a range of values. The Poisson problem as stated has the exact solution

$$u_{\text{exact}} = x(1-x)\cos(\pi y)$$

which we can compare our approximated solution to. It is customary to use the Root Mean Square Error (RMSE) in order to account for all  $x$  and  $y$  values:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^m \sum_{j=1}^n (u_{\text{exact}}(x_i, y_j) - u_{\text{approx}}(x_i, y_j))^2}{mn}}$$

We use the following range of values for  $m = n$

$$m \in \{5, 7, 10, 15, 20, 30, 40, 50\},$$

and use the finite difference method as above to find  $u_{\text{approx}}$  for that value of  $m = n$ . The exact solution is then computed as

```

1  for i=1:m
2      for j=1:n
3          uactual(i,j)=x(i)*(1-x(i))*cos(pi*y(j));
4      end
5  end

```

It trivially satisfies all four boundary conditions. A bit of work is required to show it satisfies

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = -(2 + \pi^2 x(1-x))\cos(\pi y)$$

as well.

The square root is so that the error has the dimension of  $u$ . Dividing by the total number of points  $mn$  calculates the mean error. Here  $(x_i, y_j)$  denotes all points of the  $xy$ -grid.

and we finally compute the RMSE. The results are best displayed in a log-log plot and convince us that our finite difference method is indeed second order.

However accuracy is only one half of the problem here as speed quickly becomes an issue. For  $m = n = 100$  our code needs a few seconds to run (these are not especially large values for  $m$  and  $n$ !). We have  $mn = 10^4$  which we know is near the upper limit for the size of linear systems that Matlab can handle, see Theorem 8. Two solutions are available to us.

- For moderate values of  $m$  and  $n$  (such that we can store the matrix  $A$  but solving the system directly is prohibitively long) we can use iterative methods such as Jacobi method (see Definition 6) to speed up the process. This is a technique known as relaxation.
- For larger values of  $m$  and  $n$  the only realistic possibility is to define  $A$  as a sparse matrix. However the more complex structure of  $A$  makes the process rather complicated. Note that  $A$  is far from being tridiagonal.

### Mixed boundary conditions

The code of the previous section is very adaptable and only needs minor changes in order to accommodate different types of boundary conditions.

We consider a long bar with  $4 \times 6$  m cross section. The top and bottom edges are maintained at a constant temperature of  $200^\circ\text{C}$  while the left side is insulated. On the right side cooling is provided by convective heat transfer with a fluid of temperature  $30^\circ\text{C}$ . Our goal is to compute the steady state temperature distribution  $T(x, y)$  within the bar. It satisfies the Poisson problem with  $f(x, y) = 0$ :

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

on the rectangle  $0 \leq x \leq 4, 0 \leq y \leq 6$  together with Dirichlet boundary conditions the sides  $y = 0$  and  $y = 6$ :

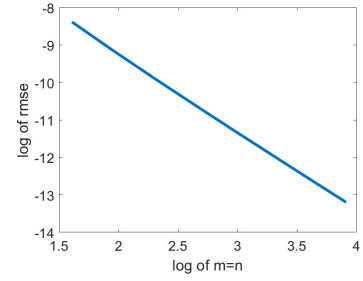
$$T(x, 0) = T(x, 6) = 200^\circ\text{C},$$

one Neumann boundary condition on the insulated side  $x = 0$ :

$$\frac{\partial T}{\partial x}(0, y) = 0$$

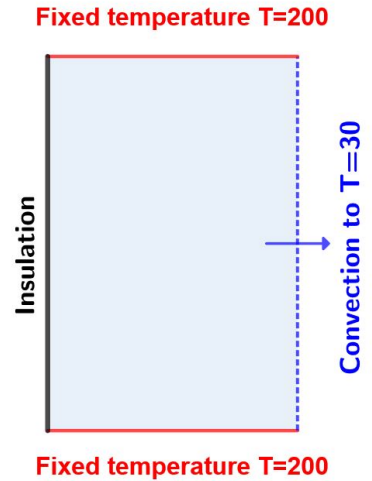
and finally one Robin condition modelling heat convection on the right side  $x = 4$ :

$$C \frac{\partial T}{\partial x}(4, y) + hT(4, y) = 30h$$



**Figure 66:** Log-log plot of the RMSE against log of  $m = n$  for our Poisson problem. According to `polyfit` the line's slope is close to  $-2$  which is consistent with a second order numerical method.

The main problem is storage rather than speed. For slightly larger sizes Matlab simply refuses to define the  $(mn) \times (mn)$  matrix  $A$  and displays an error message.



**Figure 67:** Schematics of the cross section of our bar. Boundary conditions are indicated.

$\frac{\partial T}{\partial x}$  models temperature flow through the boundary.

where  $C = 1.5 \text{ W/m} \cdot \text{K}$  is the thermal conductivity of the bar and  $h = 50 \text{ W/m}^2 \cdot \text{K}$  is the convective heat transfer coefficient, see Figure 67.

The setup is similar to the previous example, we only need to change boundary conditions as well as letting  $f(x, y) = 0$ . The two Dirichlet boundary conditions are simple and lead to the following equations:

$$\begin{aligned} v_k &= 200, & 1 \leq k \leq m & \quad \text{Bottom side} \\ v_k &= 200, & (n-1)m+1 \leq k \leq nm & \quad \text{Top side} \end{aligned}$$

This brings no changes to the coefficient matrix  $A$ , however the vector  $\mathbf{b}$  will now contain 200 at the entries  $k$  which we have specified. The Neumann condition is handled by a forward three point difference formula of Definition 22:

$$\frac{\partial u}{\partial x}(0, y) \approx \frac{-3u(0, y) + 4f(h_x, y) - f(2h_x, y)}{2h_x}$$

Forward so we enter the box towards the right.

which in our grid numbering means gives the equation

$$0 = \frac{\partial u}{\partial x}(0, y_k) \approx \frac{-3v_k + 4v_{k+1} - v_{k+2}}{2h_x}$$

for all indices  $k = (j-1)m+1$  where  $2 \leq j \leq n-1$ . Clearly we can cancel  $2h_x$  out and obtain the equations

$$-3v_k + 4v_{k+1} - v_{k+2} = 0$$

We refer to the dictionary where we worked out which indices  $k$  correspond to the left boundary.

for  $k = (j-1)m+1$  where  $2 \leq j \leq n-1$ . This equation is written at row  $k$  in the matrix  $A$ .

The Robin condition is similarly handled through a backward three point difference formula

$$\frac{\partial u}{\partial x}(4, y) \approx \frac{3u(4, y) - 4f(4 - h_x, y) + f(4 - 2h_x, y)}{2h_x}$$

Backward so that we enter the box towards the left.

In light of the equation

$$C \frac{\partial T}{\partial x}(4, y) + hT(4, y) = 30h$$

This is our convection condition in this part of the boundary.

we then obtain

$$C \frac{3v_k - 4v_{k-1} + v_{k-2}}{2h_x} + hv_k = 30h$$

for all  $k = jm$  where  $2 \leq j \leq n-1$ . This equation is slightly simplified and reordered:

Again we worked out in the dictionary which values of  $k$  lie on the right boundary.

$$\frac{C}{2h_x}v_{k-2} - \frac{2C}{h_x}v_{k-1} + \left(\frac{3C}{2h_x} + h\right)v_k = 30h$$

Again this is equation number  $k = jm$ ,  $2 \leq j \leq n-1$  in the system.

It does not quite pay off to write down the fine structure of the matrix  $A$  and we shall instead change the code where needed.

```

1 function [A,b]=constructmatrix2(m,n,hx,hy,C,h)
2     A=zeros(m*n,m*n);b=zeros(m*n,1);
3     %bottom boundary
4     for k=1:m
5         A(k,k)=1;
6         b(k)=200;
7     end
8
9     %top boundary
10    for k=(n-1)*m+1:m*n
11        A(k,k)=1;
12        b(k)=200;
13    end
14
15    %left and right boundaries
16    for j=2:n-1
17        k=(j-1)*m+1;
18        A(k,k)=-3;
19        A(k,k+1)=4;
20        A(k,k+2)=-1;
21        b(k)=0;
22        k=j*m;
23        A(k,k-2)=C/(2*hx);
24        A(k,k-1)=(2*C)/hx;
25        A(k,k)=(3*C)/(2*hx)+h;
26        b(k)=30*h;
27    end
28
29    %inner points - jump over boundaries
30    for k=m+1:(n-1)*m
31        if mod(k,m)>1
32            A(k,k-m)=1/hy^2;
33            A(k,k-1)=1/hx^2;
34            A(k,k)=-2/hx^2-2/hy^2;
35            A(k,k+1)=1/(hx^2);
36            A(k,k+m)=1/(hy^2);
37            b(k)=0;
38        end
39    end
40 end

```

We need to carry  $C$  and  $h$  over from the main program. However since we do not find the **findposition** subroutine anywhere we can skip  $a$  and  $c$ . All boundary conditions are position independent.

There is a number of changes to the way the matrix  $A$  is coded.

- Left boundary condition now reflects the equation

$$-3v_k + 4v_{k+1} - v_{k+2} = 0$$

- Right boundary condition now reflects the equation

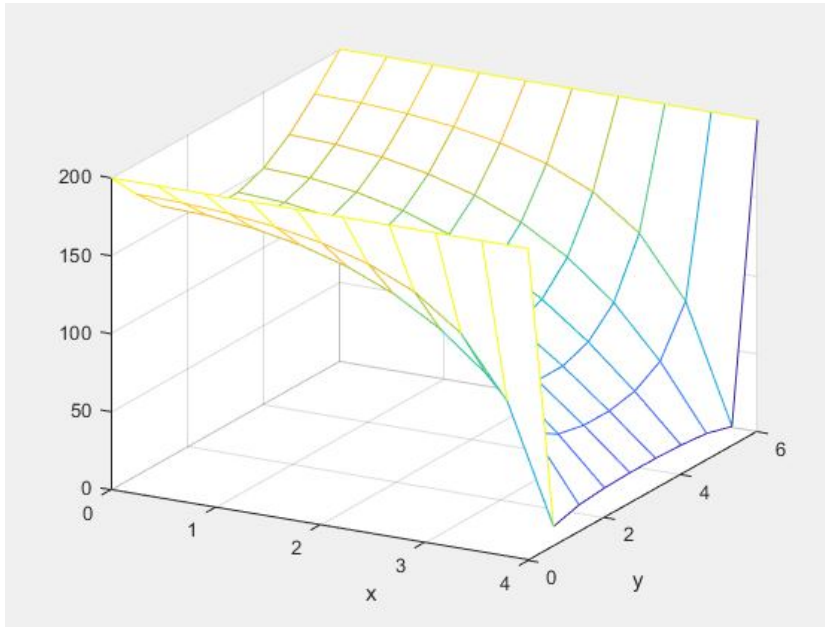
$$\frac{C}{2h_x}v_{k-2} - \frac{2C}{h_x}v_{k-1} + \left(\frac{3C}{2h_x} + h\right)v_k = 30h$$

Inner points are left unchanged. Top and bottom points are still Dirichlet conditions, the changes are only visible in the right-hand side  $b$ .

Changes regarding  $b$  are as follows:

- Both Dirichlet conditions (top and bottom) are of the form  $v_k = 200$ .
- The Neumann condition (left) has right-hand side 0.
- The Robin condition (right) has right-hand side  $30h$ .
- The main equation has right-hand side  $f(x,y) = 0$ .

The resulting temperature distribution is plotted in the figure below.



The Dirichlet conditions maintaining the sides where  $y = 0$  and  $y = 6$  are clearly visible. Heat exits the bar through the right side while the left side is insulated.

### *Sparse setup*

Because our coefficient matrix  $A$  is not quite tridiagonal it is more challenging to code it as a sparse matrix. This would however allow for much larger values for  $m$  and  $n$  and hence higher stability and accuracy. For  $m = n = 4$  the matrix  $A$  is shown below. For simplicity we let at inner points

$$\alpha = \frac{1}{h_y^2}, \quad \beta = \frac{1}{h_x^2}, \quad \gamma = -\frac{2}{h_x^2} - \frac{2}{h_y^2}$$

and

$$\delta = \frac{C}{2h_x}, \quad \varepsilon = \frac{2C}{h_x}, \quad \zeta = \frac{3C}{2h_x} + h$$

at right boundary points.

This matrix corresponds to our last example. Different boundary conditions would alter the structure of the matrix at the boundary points - although never dramatically so.



$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -3 & 4 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \alpha & 0 & 0 & \beta & \gamma & \beta & 0 & 0 & \alpha & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \alpha & 0 & 0 & \beta & \gamma & \beta & 0 & 0 & \alpha & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \delta & \varepsilon & \zeta & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -3 & 4 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \alpha & 0 & 0 & \beta & \gamma & \beta & 0 & 0 & \alpha & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \alpha & 0 & 0 & \beta & \gamma & \beta & 0 & 0 & \alpha & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \delta & \varepsilon & \zeta & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

The non-zero coefficients of  $A$  all lie on six diagonal lines within the matrix:

1. the **main diagonal** of length  $mn$
2. the **off by one diagonals**, each of length  $mn - 1$
3. the **off by two diagonals**, each of length  $mn - 2$
4. the **off by  $m$  diagonals**, each of length  $mn - m$ .

The main diagonal has three distinct parts:

- a series of  $m = 4$  ones,
- the vector  $[-3 \ \gamma \ \cdots \ \gamma \ \zeta]$  being repeated  $m - 2$  times,
- another series of  $m = 4$  ones.

The **repmat** in Matlab helps us to code the main diagonal in two quick lines:

```
1 x = [-3 ; gamma*ones(m-2,1) ; zeta];
2 maindiag = [ones(m,1) ; repmat(x,m-2,1) ; ones(m,1)];
```

Similar strategies can be applied to the other five diagonals. For instance the upper off by one diagonal consists of:

- a series of  $m = 4$  ones,
- $m - 2$  repetitions of the vector  $[4 \ \beta \ \cdots \ \beta \ 0]$

The **off by two diagonals** appear solely because of Neumann/Robin boundary conditions.

The **off by  $m$  diagonals** are caused by the discretized Poisson equation at inner points.

In general there would be a total  $m - 2$  gammas instead of only two.  $m - 2$  corresponds to the total number of values along the  $x$ -axis of the grid minus the two boundary points.

The parameter 1 in **repmat** tells Matlab to build a vector of width 1. The default (rather incomprehensibly) constructs a square matrix.

Note that the off by one diagonal is one shorter at its lower-right side.

- a final series of  $m - 1 = 3$  ones.

which we code using:

```
1 x=[4 ; beta*ones(m-2,1) ; 0 ];
2 upperdiag=[zeros(m,1) ; repmat(x,m-2,1) ; zeros(
    m-1,1)];
```

Having defined what coefficients the matrix contains, it remains to decide where to put them in the matrix. Again we strive to avoid loops. A common strategy is to define two large index vectors **i** and **j** containing all the coordinates, and a large vector **values** containing all coefficients. A full code defining  $A$  as a sparse matrix is as follows:

```
1 m=50;n=50;
2 a=0;b=4;c=0;d=6;C=1.5;h=50;
3 hx=(b-a)/(m-1);hy=(d-c)/(n-1);
4 %six coefficients
5 alpha=1/(hy^2);
6 beta=1/(hx^2);
7 gamma=-2/(hx^2)-2/(hy^2);
8 delta=C/(2*hx);
9 epsilon=(2*C)/(hx);
10 zeta=(3*C)/(2*hx)+h;
11
12 %main diagonal
13 i=[1:m*n];j=[1:m*n];
14 x=[-3 ; gamma*ones(m-2,1) ; zeta ];
15 maindiag=[ones(m,1) ; repmat(x,m-2,1) ; ones(m
    ,1)];
16
17 %upper diagonal one off
18 i=[i 1:m*n-1]; j=[j 2:m*n];
19 x=[4 ; beta*ones(m-2,1) ; 0 ];
20 upperdiag=[zeros(m,1) ; repmat(x,m-2,1) ; zeros(
    m-1,1)];
21
22 %upper diagonal two off
23 i=[i 1:m*n-2]; j=[j 3:m*n];
24 x=[-1 ; zeros(m-1,1) ];
25 upperdiag2=[zeros(m,1) ; repmat(x,m-2,1) ; zeros
    (m-2,1)];
26
```

Lines 1-10 establish all necessary parameters.

Line 13 tells us we consider the points  $(1,1), (2,2), \dots, (mn, mn)$  which correspond to the main diagonal. The coefficients to be assigned there are defined in **maindiag** as before. These three objects need to have the same size.

The process is repeated five other times with variations. For instance the two-off upper diagonal of  $A$  consists of indices  $(1,3), (2,4), \dots, (mn-2, mn)$ . Most values there are 0 excepts for a single  $-1$  repeated  $m-2$  times.

The vectors  $i$  and  $j$  are updated (concatenated) in each paragraph. The vector **values** is likewise a simple concatenation of all the diagonal coefficients.

The **sparse** commands then defines a sparse matrix with values being set at all  $(i,j)$  couples. Other values are defaulted to 0 and are not stored actively.

```

27 %lower diagonal one off
28 i=[i 2:m*n]; j=[j 1:m*n-1];
29 x=[0 ; beta*ones(m-2,1) ; epsilon];
30 lowerdiag=[zeros(m-1,1) ; repmat(x,m-2,1) ;
    zeros(m,1)];
31
32 %lower diagonal two off
33 i=[i 3:m*n]; j=[j 1:m*n-2];
34 x=[zeros(m-1,1) ; delta];
35 lowerdiag2=[zeros(m-2,1) ; repmat(x,m-2,1) ;
    zeros(m,1)];
36
37 %alpha diagonal left - m to the left of main
    diagonal, total length is mn-m
38 i=[i m+1:m*n]; j=[j 1:m*n-m];
39 x=[0; alpha*ones(m-2,1) ; 0];
40 alphadiag1=[repmat(x,m-2,1) ; zeros(m,1)];
41
42 %alpha diagonal right - m to the right of main
    diagonal, total length is mn-m
43 i=[i 1:m*n-m]; j=[j m+1:m*n];
44 x=[0; alpha*ones(m-2,1) ; 0];
45 alphadiag2=[zeros(m,1) ; repmat(x,m-2,1)];
46
47 %construct sparse matrix
48 values=[maindiag;upperdiag;upperdiag2;lowerdiag;
    lowerdiag2;alphadiag1;alphadiag2];
49 A=sparse(i',j',values);

```

We might as well define the right-hand side  $\mathbf{b}$  as a sparse vector, although this is less relevant in term of speed and storage space.

```

1 i=[1:m 2*m:m:(n-1)*m (n-1)*m+1:n*m]';
2 j=ones(1,1);
3 values=[200*ones(m,1); 30*h*ones(n-2,1); 200*ones(
    m,1)];
4 bsparse=sparse(i,j,values)

```

Solving Poisson's equation is then simply achieved by solving the linear system as usual:

```

1 w=A\sparse;

```

and plotting the results as we have done. This setup is very flexible

It is very much recommended to compare the sparse versions of  $A$  and  $\mathbf{b}$  with their full counterparts defined through loops, at least for small values of  $m$  and  $n$ .

and only needs minor tweaking were we to use different boundary conditions or an another equation related to Poisson's equation.

# Index

- 3D plot, 131
- animated plot, 132
- bisection method, 5, 49
- central difference approximation, 108, 113, 119
- composite Newton-Cotes methods, 77
- composite Simpson's method, 82
- composite trapezoid rule, 78
- curve fitting, 51
- diagonally dominant matrix, 37
- differential system, 100, 103
- Dirichlet condition, 112, 113, 149
- elliptic PDEs, 138
- endpoint rule, 72, 90
- Euler's method, 90
- forward difference approximation, 107
- function handle, 11
- fundamental theorem of analysis, 72, 74, 90
- Gauss-Newton method, 65
- Gaussian elimination, 32
- golden ratio, 24, 29
- golden section search, 22
- heat equation, 127
- Heun's method, 96
- initial value problem, 89
- intermediate value theorem, 5
- interpolation (three points), 52, 74
- inverse matrix theorem, 31
- Jacobi method, 37
- Lagrange polynomial, 53
- large linear systems, 36
- linear finite difference method, 113
- linear least squares, 59
- linear rate of convergence, 12, 19, 23
- linear system, 31
- local convergence, 15, 27, 42
- log-log plot, 93, 97, 109, 149
- matrix norm, 38
- mean value theorem, 16, 94
- nested form, 57
- Neumann condition, 112, 132, 149
- Newton's divided differences, 54
- Newton's method (multivariate), 41, 48, 122
- Newton's method (one variable), 14
- Newton-Cotes formulas, 71
- non-linear boundary value problem, 121
- non-linear least squares, 64
- non-linear system, 41
- non-singular matrix, 31
- normal equations, 61, 65
- numerical differentiation, 107
- order of a numerical method, 80, 82, 93, 97, 100, 108, 149
- parabolic PDEs, 127
- partial derivatives, 112
- phase-space plot, 102
- Poisson equation, 138
- quadratic rate of convergence, 20, 42
- residual error, 59, 64
- RMSE, 64, 148
- Robin condition, 112, 149
- root, 5
- Runge phenomenon, 58
- Runge-Kutta method (RK4), 98
- secant method, 26
- second degree differential equation, 103
- second derivatives, 111
- Simpson's rule, 76, 98
- sparse matrix, 36, 114, 117, 152
- superlinear rate of convergence, 27
- symbolic differentiation, 18, 120
- Taylor approximation, 21, 28, 94, 108
- three-point forward difference approximation, 110, 116, 122
- tolerance, 6, 16, 24, 39, 49, 84
- trapezoid rule, 73, 95
- tridiagonal matrix, 36, 114, 129
- unimodal function, 22
- vector norm, 39, 104