Lecture Notes on Numerical Mathematics

Jan Bouwe van den Berg, Bob Planqué

January 31, 2017

What this is. These are lecture notes that we use for several overlapping courses. Not everything will be covered in your course, and we may discuss additional material (e.g. related to assignments) which is not represented here. We did not attempt to write in "textbook style". Most parts are probably not readable independently, i.e. without coming to the lectures. All feedback is welcome.

Contents

Chapter 1

The beginning

1.1 What do we mean by numerical mathematics?

The combination of mathematics and computer calculations is often referred to with different words. *Numerical Mathematics* suggests theorems and proofs. *Numerical Analysis* stresses continuous mathematics (i.e., analysis, as opposed to discrete mathematics, like cryptography, number theory, travelling salesman problem, etc). *Scientific Computing* is aimed at applications. We do not treat stochastic processes. Although there are numerical methods for those as well (Monte-Carlo techniques, etc), these will appear in other courses.

In this course, we focus on *Numerical Analysis* with more emphasis on applicability than on theorems.

We are thus concerned with the study of algorithms for problems in analysis. We want to calculate answers, fast and accurately. For example,

- solve $x^5 - 18x^3 - 7x + 3 = 0$,
- if A is a 1000×1000 matrix and b a vector: solve $Ax = b$.

These algorithms are often used in serious applications: for example

- fluid dynamics/flows: airplane (wing) design
- weather forecasting
- launching a satellite into space

We can distinguish two important classes of algorithms, *finite* and *iterative* algorithms. Finite algorithms are known/proven to give a solution to a problem in at most N "elementary operations", with rounding off errors of at most a small number ε. Iterative algorithms calculate approximations x_n, $n = 1, 2, \ldots$ to a problem, such that the x_n converge to the actual solution as $n \to \infty$.

Most methods are iterative! The proofs are generally of the form: you take N iterations (N large), or very small step size h in time or space, to approximate a solution, and then the numerical solution tends/converges to the actual solution as $N \to \infty$, or as $h \to 0$. Moreover, this convergence happens at a certain speed: the error is smaller than, say, N^{-k} or h^k, for some $k > 0$.

An example that you have seen before is a Riemann sum to approximate an integral.

Of course there are also going to be theorems and proofs. And the proofs are important. *But*, proofs often work under certain hypotheses, which are hard to verify, only in special, easy cases (for example: linear equations).

The proofs do give insight into whether the method is good in theory. Of course, a method should at the very least work in simple cases. In practice, you have to carefully *extrapolate* to more difficult problems.

1.2 What we do, and what we do not do

Our goals/aims are to

- Learn the background of methods
- Get experience with doing calculations

- You have to be able to use it
- You should learn what the dangers are
- Few proofs (as discussed before)

There are lots and lots of methods. In this course, we restrain ourselves, so that hopefully you can follow other (more advanced) courses to learn more, or look up more complicated algorithms in a book. We do not always deal with the most advanced or most modern method. In this course, it is all about the *ideas*, *and* how to implement them.

A small warning: it is not easy! It requires a different way of thinking compared to a definition-theorem-proof type course, and knowledge of different areas of mathematics in the background is also necessary or needs to be refreshed, e.g. real analysis, linear algebra, ODEs. Moreover, as you may already know, programming can be frustrating, and takes time.

1.3 Some terminology

Say we want to solve a problem $F(x,p) = 0$, where p is a (collection of) parameter(s). We solve the numerical approximation(s) $F_n(x_n, p_n) = 0$, where $p_n \to p$ and *hopefully* $x_n \to x$ as $n \to \infty$.

The problem, or a numerical method (for fixed n), is called *stable* if for every p_n near p there is a locally unique solution $x_n = x_n(p_n)$, and moreover, for every such p_n there exists a (small) $\delta > 0$ and a (large) $\overline{K} = \overline{K}_n(p_n)$ such that

$$|x_n(\tilde{p}_n) - x_n(p_n)| \leq \overline{K}|\tilde{p}_n - p_n| \qquad \text{for all } |\tilde{p}_n - p_n| \leq \delta.$$

In other words: the solution of $F_n = 0$ depends continuously on the parameters, in fact Lipschitz continuity is required ("linear dependence").

The idea is that $\overline{K}_n(p_n)$ does not blow up (stays bounded) as $n \to \infty$. Hence, let us now assume that $\overline{K}$ is independent of n. Then we can say something about the number $\overline{K}$:

- roughly $\overline{K} = |\frac{\partial F}{\partial p}|/|\frac{\partial F}{\partial x}|$ (by the implicit function theorem).
- For the scalar problem $F(x,p) = Ax - p$ we have $\overline{K} = 1/|A|$, or in the vector-valued case we have $\overline{K} = \|A^{-1}\|$ (with $\|A^{-1}\|$ measuring how big the inverse of A is (the matrix norm in fact, see §A.1)).
- $\overline{K}$ is called the *absolute condition number*. We will also consider *relative* condition numbers later (cf. §A.2).
- if $\overline{K}$ is large we call the problem *ill-conditioned*.

You should really avoid trying to solve ill-conditioned problems, or using ill-conditioned solution methods.

Chapter 2

What is MATLAB?

MATLAB is a software package specifically designed to do numerical (rather than symbolic) computations. It is accessible, easy to program, has large libraries ("toolboxes"; if you have access to them), and is particularly suited to work with matrices. Luckily, linear algebra is crucial in Numerical Methods. MATLAB is able to calculate with complex numbers, and it is fairly easy to make pictures.

Some disadvantages are that it costs money (there is a free alternative: Octave, but it is not as well developed, slightly behind, less strong at graphics, and there are small syntax differences with MATLAB). MATLAB is also not the fastest, but it is possible to program the routines that requires the most computations in a fast language such as C, and then incorporate those routines in MATLABprograms. We won't do that in this course.

This chapter contains a lot of example MATLAB commands. To really follow it, and learn the little lessons along the way, type in the commands yourself and inspect the outcomes!

2.1 Working with numbers

The following are commands in MATLAB. Just fire it up on a computer, and type in these lines one by one, pressing Enter each time.

Inputting numbers:

```
>> 2+2
```

Squaring a number:

```
>> 4^2
```

You need to add parentheses:

```
>> -3^2    gives −9
>> (-3)^2    gives 9
```

Standard functions are included, and don't forget `*` for multiplication:

```
>> sin(3(cos(9))
>> sin(3*cos(9))
>> sin(pi/2)
```

Dividing by zero evaluates to infinity:

```
>> 1/0    gives Inf
```

Complex numbers are included:

```
>> sqrt(-3)
>> exp(i*pi/2)
```

The number i is always the complex number i unless you redefine it (e.g., to use as a counter):

```
>> i=1
>> exp(i*pi/2)
```

To reset a variable, use `clear`. To clear all variables, use `clear all`.

```
>> clear i
>> exp(i*pi/2)    now re-evaluates to i
```

0/0 is not defined, and evaluates to NaN, or 'not a number':

`>> 0/0` gives `NaN`

Many functions are built in, such as taking square roots:

```
>> x=sqrt(3)
```

MATLAB variables are case-sensitive, so `x` is now defined, but `X` is not:

`>> x` gives $x = 1.7321$

`>> X` gives Undefined function or variable 'X'.

2.2 Matrices

This is how you make a row array:

`>> x=[1,2,3]` gives $(1, 2, 3)$

Using semicolons instead of commas gives a column array:

`>> y=[4;5;6]` gives $(4, 5, 6)^T$.

A 2x2 matrix A is defined here as the concatenation of two rows of length two:

`>> A=[1,2;3,4]` gives

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

To make a vector $x = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$, you can use the *colon* (`:`) separator. The following definition basically means the row of numbers starting at 1 and increasing to 10 by steps of 1 (try other step sizes yourself, even negative ones):

```
>> x=[1:1:10]
```

To transpose, use the ':

```
>> w=x'
```

To do matrix multiplication, define the following:

```
>> B = [1 1; 1 1];
>> z = [10;10]
```

To multiply, use `*`:

```
>> B*z
```

We can concatenate B and z with

`>> [B,z]` which gives

$$\begin{pmatrix} 1 & 1 & 10 \\ 1 & 1 & 10 \end{pmatrix}$$

If we multiply by a vector with the wrong dimensions, (y is a 1x3 vector), then

`>> B*y` gives `Error using *, Inner matrix dimensions must agree.`

Here are some matrix multiplication examples, for AB and A^2:

```
>> A*B
>> A^2
```

MATLAB also does *pointwise* multiplication, using the dot-operator, on a matrix $A = (a_{ij})$:

`>> A.^2` gives (a_{ij}^2)

To select entries from a matrix, use the following syntax for A_{12}:

```
>> A(1,2)
```

Selecting the first row of A is done by:

```
>> A(1,:)
```

Selecting the second column from A by:

```
>> A(:,2)
```

2.3 Complex numbers and transposing

Using i in any matrix definition makes it a complex-valued matrix:

```
>> A=[1,i;3,2*i]
```

Transpose now means 'transpose and complex conjugate':

```
>> A'
```

Transposing the elements without the conjugation is done by:

```
>> A.'
```

The same holds for vectors. Try the following:

```
x=[1,i]
x'
x.'
i'
```

2.4 And some more...

Let's take a vector from 0 to 20, in steps of 0.1.

```
>> x=[0:0.1:20];
```

Taking its sine, we can plot the graph as follows:

```
>> z=sin(x);
>> plot(x,z)
```

If we take a matrix A, we swap its elements left-right by the handy function

```
>> fliplr(A)
```

If you want to find out more about the inner workings and syntax of a function, try

```
>> help fliplr
```

`>> doc fliplr` is perhaps more detailed.

Also note the commands at the bottom of these help files. They are related commands, and are sometimes useful for related tasks.

Many commands are not precompiled, which means their code is available to the user! Try e.g.,

```
>> type fliplr
```

It may or may not give you much information.

To use comments in MATLAB, use the %. Everything on a line following % is not interpreted by MATLAB

```
% hoi
```

Normally, MATLAB doesn't show a lot of output.

`>> exp(-3)` gives `0.0498`

Changing the output format, using `format long`, results in `0.049787068367864`. Using `format long e` gives `4.978706836786394e-02`, To revert back to the short output format, use `format short` or `format short e` which gives `4.9787e-02`. Another handy command is `format compact` which suppresses the extra white lines seen in all the MATLAB output (in other words, you can fit more output on your screen).

2.5 matlab and functions

2.5.1 Linear equations

This is what MATLAB is made for (there is a general theme in this course: reduce every problem to solving lots of systems of linear equations). Here is how to solve a simple linear system, $Ax = b$.

```
A=[1 2 3; 4 5 6; 7 8 10]
b=[11 12 13]'
x=A\b
A*x
```

(Why not $A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$?)

An important note is to *use vector and matrix operations whenever possible.*
MATLAB is optimised for such operations, and often works much faster (than a `for` or `while` loop).

2.5.2 Programming

This is a reminder about basic programming and its syntax in MATLAB.

(1) Conditional
Example: absolute value $b = |a|$

```
a=-1;
if a>=0 b=a; else b=-a; end
b
```

(2) Loop

```
for k=1:12 p(k)=k^2; end
p
p=[]
for k=2:2:12 p(k)=k^2; end
p
p=[]
for k=[2 3 5 7 11] p(k)=k^2; end
p
```

(3) Alternative for a loop is often vectorisation:

```
k=[1:10]
r=k.^2
```

(4) Different loop

```
k=1;
while k<11 q(k)=k^2; k=k+1; end
q
```

2.5.3 Functions

In MATLAB it is handier to use functions that you have written yourself than "normal" m-files.

- They are more flexible
- They are faster (compiled once).
- They lead to better structured programs

We illustrate the use of functions using a simple example, the Fibonacci sequence, which is defined by $f_{n+2} = f_{n+1} + f_n$ with $f_0 = 1$, $f_1 = 1$.

Read the following file `fibonacci.m` line by line.

```
function [f,phi]=fibonacci(n,f01)
% FIBONACCI computes fibonacci sequence
%
% [F,PHI]=FIBONACCI(N,[F0 F1]) computes
% the first N terms in the Fibonacci sequence
% starting with F0 and F1
%
% Output is a vector F of length N consisting of the
% elements of the Fibonacci sequence and
% the approximation PHI=F(N)/F(N-1) of the golden ratio.
%
% [F,PHI]=FIBONACCI(N) uses default values F0=F1=1
```

```
% default values
if ~exist('f01')
  f01=[1 1];
end

f(1)=f01(1);
f(2)=f01(2);

% iterations
for j=3:n
  f(j)=f(j-1)+f(j-2);
end

% golden ratio
phi=f(n)/f(n-1);
```

We can open the file using `open fibonacci`, and edit it using the MATLAB editor. The first line defines the inputs and outputs of the program. The first lines following this definition is used for comments on the usage of the program, a kind of help file for the function. (This is also expected in the hand-in assignments.) The first block of comments is hence the documentation. It can be called using

```
>> help fibonacci
```

It gives the default values of optional inputs, the structure of the input and output, and notes which inputs are optional and which mandatory.

Now to use the function,

```
>> f=fibonacci(10)
>> [f,phi]=fibonacci(10)
```

The output ϕ is thus optional.

To see that the ratio of subsequent Fibonacci numbers converges to the golden mean $\frac{1}{2}(1+\sqrt{5})$,

```
>> format long
>> f
>> phi
>> [f,phi]=fibonacci(100);
>> phi
>> (sqrt(5)+1)/2
>> [f,phi]=fibonacci(100,[2 7]);
>> phi
```

See §A.4 for the theory that explains why this happens.

2.6 Be careful

Do not believe everything MATLAB tells you, e.g. try the following, to see what the problem is...

```
>> a=10^8
>> b=10^-8
>> b+a-a-b
```

now gives $4.9012e-09$

```
>> a-a+b
```

gives $-1.0000e-8$, but

```
>> b+a-a
```

gives $1.4901e-08$!

The order in which you do things makes a difference, especially when adding and subtracting large and small numbers!

Here is a more complicated example. Consider the differential equation $\frac{dx}{dt} = x(1-x)$. Its solution is given by $x = \frac{1}{1+Ce^{-t}}$, and example solutions are given in Figure 2.1, on the left. All the solutions are

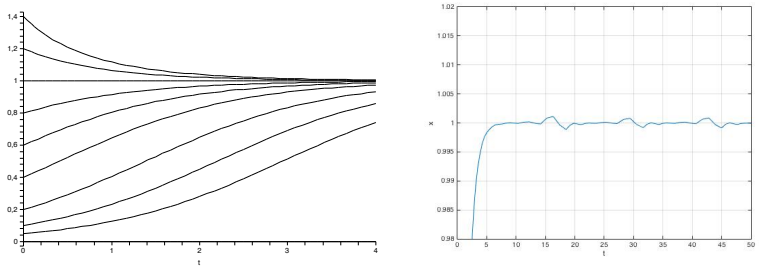

FIGURE 2.1. Integrating the logistic equation. One the left a few analytical solutions, on the right a numerical solution which is not monotonically increasing.

monotone increasing or decreasing (or completely constant). One might use a standard numerical method, built into MATLAB, to integrate the ODE, using e.g.

```
>> [t,x]=ode45(@(t,x) x.*(1-x),[0 50],0.8);
```

We plot the solutions using

```
>> plot(t,x)
```

and zoom in on the part of the solution where $x \approx 1$, see Figure 2.1, on the right,

```
>> axis([0 50 0.98 1.02])
```

The problem is in the default tolerance used in the computation (it is not strict enough), causing the ODE solver to take too large time steps. You will learn more on numerical integration of ODEs later on.

2.7 Approximations and Errors

A few things about errors. The "solution" is the actual number you want to know, Denote the actual/real solution by $\bar{x}$, and the calculated "solution" by $\tilde{x}$. Then the *absolute error* is defined by $|\tilde{x} - \bar{x}|$. The *relative error* is defined by $\frac{|\tilde{x}-\bar{x}|}{|\bar{x}|}$, provided $\bar{x} \neq 0$.

Of course, in estimating the absolute and relative errors in any real numerical computation, we have a problem... $\bar{x}$ is unknown! But of course you can test on analytically solvable problems, for which the solution *is* known in advance.

Alternatively, you can estimate the error analytically. For example, you can try to use Taylor series (with remainder),

$$f(x) = f(0) + f'(0)x + \frac{1}{2}f''(0)x^2 + \cdots + \frac{1}{n!}f^{(n)}(0)x^n + R_n(x),$$

but then you need to know how smooth the function f is. For instance, if f is n times continuously differentiable, then you only get $R_n(x) \to 0$ as $x \to 0$. When $f \in C^{n+1}([0, x])$, then the error/remainder $R_n(x)$ can be estimated (as in the Theorem of Taylor with remainder term):

$$|R_n(x)| \leq \frac{|x|^{n+1}}{(n+1)!} \max_{s\in[0,x]} |f^{(n+1)}(s)|.$$

If you know an upper bound on $|f^{(n+1)}|$, then you thus can estimate the absolute error of the Taylor series with respect to the function f.

On top of such estimates come rounding errors. In MATLAB: smallest relative error is `eps` (of the order 10^{-16}).

2.7.1 Floating point numbers

Numbers are represented in MATLAB as

- $(-1)^s \cdot (0.a_1a_2 \ldots a_t) \cdot \beta^e$
- $a_1 \neq 0$, $s \in \{0, 1\}$, $\beta \geq 2$, $L \leq e \leq U$

- The parameters β, t, L and U hence define a set of numbers, $\mathbb{F}(\beta, t, L, U)$. In the case of MATLAB, this set is $\mathbb{F}(2, 53, -1021, 1024)$.

This is an international IEEE standard: computer chips *must* operate correctly on $\mathbb{F}(2, 53, -1021, 1024)$. Note that 0 is not in $\mathbb{F}$! `0`, like `Inf` and `NaN`, are treated exceptionally by MATLAB.

An important number to keep in mind is the relative precision (round-off error) for any number in any computation: $\frac{|x-fl(x)|}{|x|} \leq \frac{1}{2}\varepsilon_M$, where $\varepsilon_M = \beta^{1-t} = 2^{-52} \approx 2.22 \cdot 10^{-16}$. MATLAB has lots of built-in variables, which are sometimes handy. Try e.g.

`>> eps`
`>> realmax` gives `1.7977e+308`, the largest number in $\mathbb{F}$.
`>> realmin` gives `2.2251e-308`, the smallest number in $\mathbb{F}$.

There are actually smaller numbers which MATLAB can work with, e.g.,
`>> 2^{-1060}` which gives `8.0948e-320`, but this number is not in $\mathbb{F}$, since $a_1 \neq 0$, and the number starts with a number of 0's. It has low relative precision.

Note that the computer computes in binary numbers, but we see decimal numbers. This becomes clear if we try the following.
`>> 0.3/0.1` gives `3.0000`, but
`>> 0.3/0.1-3` gives `-4.4409e-16`.

Another example, solve

$$\begin{aligned} 17x+\ \ 5y &= 22 \\ 1.7x+0.5y &= 2.2 \end{aligned}$$

A solution is for example $x = y = 1$, but if we try

```
A=[17,5;1.7,0.5]
b=[22;2.2]
c=inv(A)*b
A*c
d=A\b
A*d
```

we are presented with a number of warnings like
`Warning: Matrix is close to singular or badly scaled. Results may be inaccurate. RCOND = 2.293849e-18.` We get answers `c=[0 8]'`, but `A*c = [40 4]'`, and `d=[-1.8431, 10.6667]'`, but we do get `A*d = [22, 2]'`!

The following is an example in which the numerical evaluation of a polynomial is computed in two ways. One by writing out the polynomial in full, and the other by using its factorisation. Note that the factorised version works much better, see Figure 2.2 The code is given below.

```
f=@(x) x.^7-7*x.^6+21*x.^5-35*x.^4+35*x.^3-21*x.^2+7*x-1
t=[0.988:0.0001:1.012];
plot(t,f(t))
hold on
plot(t,(t-1).^7,'k')
hold off
```

2.7.2 On errors

There are three sources of errors in numerics:

- Modelling errors: 1st type of errors
- Discretisation of the model: 2nd type of errors, e.g. when integrating or solving differential equations
- Rounding errors: 3rd type of errors

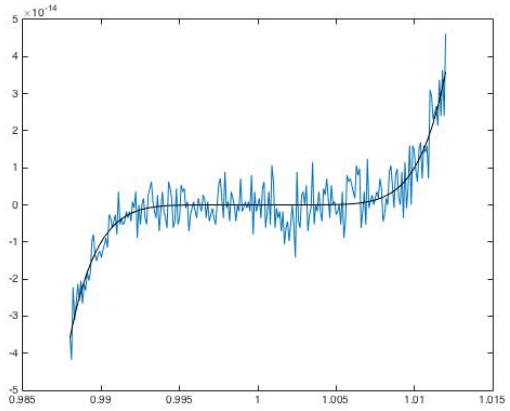

FIGURE 2.2. Polynomial evaluated in two ways

What is better in general, addition/subtraction, or multiplication/division, when it comes to errors? Let some quantity x by approximated by $\tilde{x}$. $\tilde{x} = x + \Delta x$, then the relative error in x is $\epsilon_x = \frac{\Delta x}{x}$. We keep the sign.

Now we want to know $f(x)$, but we only know $\tilde{x}$. Since

$$f(\tilde{x}) \approx f(x) + f'(x)\Delta x,$$

the relative error in $f(x)$ is

$$\epsilon_{f(x)} = \frac{f'(x)\Delta x}{f(x)} = \frac{f'(x)x}{f(x)}\frac{\Delta x}{x} = \frac{f'(x)x}{f(x)}\epsilon_x.$$

Equalities are only approximate here! For a function of two variables, we compute

$$f(\tilde{x}, \tilde{y}) = f(x, y) + \frac{\partial f}{\partial x}(x, y)\Delta x + \frac{\partial f}{\partial y}(x, y)\Delta y.$$

The relative error is

$$\epsilon_{f(x,y)} = \frac{f_x}{f} x\epsilon_x + \frac{f_y}{f} y\epsilon_y.$$

Some simple examples show that addition/subtraction is potentially *worse* than multiplication/division.

- $f(x, y) = xy$: $\epsilon_{xy} = \epsilon_x + \epsilon_y$.
- $f(x, y) = x/y$: $\epsilon_{x/y} = \epsilon_x - \epsilon_y$.
- $f(x, y) = x \pm y$: $\epsilon_{x\pm y} = \frac{x}{x\pm y}\epsilon_x + \frac{y}{x\pm y}\epsilon_y$. This does not bode well if $x \pm y \approx 0$.

Conclusion: adding/subtracting is more dangerous than division/multiplication, especially if the sum/difference is almost zero.

One might try to use *interval arithmetic* to keep track of the errors made. The idea is that you specify intervals in which the quantities lie that you compute with, and keep track of those intervals as you compute. For example, if $x \in [0.9, 1.1]$ and $y \in [1.1, 1.2]$ then $xy \in [0.99, 1.32]$. This needs special algorithms, and there are libraries for that. But we are not going to do it in this course.

2.8 Using functions

The following code describes the definition of a so-called *anonymous* function `f`.

```
f=@(x) x-x.^3
f(1)
f(2)
f([1 2])
```

It is written immediately in vector notation using the dot-operator for point-wise operations.

Here `f` is a *function handle* (kind of like a pointer to a function); the function itself does not have a name. This as opposed to a function in a file; there is the command `sin`, which refers to the name of the function in the file `sin.m`. The function handle to `sin` is `@sin`.

Function *handles* can be given as input variables to other functions. In the following example, `quad` is the Simpson's quadrature function, a routine to perform numerical integration of a function. I.e., we are approximating $\int_0^1 f(x)\,dx$ here. The `quad` function takes other functions as argument, and these should be specified using a function handle. Compare

```
quad(f,0,1)
quad(@sin,0,pi)
```

Here are some more examples of the definition of anonymous functions.

```
f=@(x,y) x.^2+y
f(1,2)
f(2,1)
f(0,[1 2 3])
f([0 10 100],[1 2 3])
g=@(x) f(x,1)
g(3)
h=@(x) x(1)+x(2)
h([10 100])
h([1 2 3])
h(1)
h=@(x) [x(1),x(2);x(3),x(4)]
h([1 2 3 4])
h=@(x) [x',x.^2']
h([1 2 3 4])
```

MATLAB allows for some very efficient notation. The following hopefully makes this immediately clear.

```
A=10*rand(4)
B=A<5
find(B)
A(find(A<5))
A(A<5)
```

and

```
A=rand(3,4)
sz=size(A)
B=A(:)
C=reshape(B,sz)
```

2.9 Debugging and optimisation

A few notes on debugging and optimisation. Consider the following code, which does nothing except fill a matrix with numbers. This code is written in the file `useless.m`:

```
function x = useless(n,m)
% USELESS calculates a little bit

if nargin<2
  m=n;
end
```

```
% x = zeros(n,m);
for i=1:n
  for j=1:m
    x(i,j)=i+j;
  end
end
```

If we call
`>> useless(10)` we find a 10×10 matrix filled with numbers. You can add *breakpoints* on a line of code. In MATLAB, click on `Editor > Breakpoints > Enable/Disable` to add a breakpoint on the line where the cursor is at present. Running the program now enters the debugger, shown with
`K>>`
If you want to see a particular variable's value at this moment in the program, you can type e.g.
`K>> m` which gives `10`. To continue the run, do `Editor > Continue`. MATLAB now exits the debugger.

After starting a program, and hitting a breakpoint, you can step through the code line by line using `Editor > Step`.

Your code may be really slow, for some reason. In this case, filling a 2000×2000 matrix takes a long time. The `tic`, `toc`, in the following code starts and stops a clock, giving the elapsed time as a result.
`>> tic;useless(2000);toc` gives `2.19` seconds on the authors' machine.

If we *pre-allocate* memory for the matrix to be filled, using `x=zeros(n,m);` in the useless.m file, then
`>> tic;useless(2000);toc` gives only `0.056` seconds! Forty times faster!

2.10 Some final matlab things

Not necessary yet, but may/will be useful later:

- Cells

```
clear A
A{1}=3;
A{2}=[2,4]
A{3}=[3,4;5,6]
A{4}='hoi'
A
A{3}
```

- Sparse matrix

```
x=[1:10];
y=[10:-1:1];
z=randn(1,10);
B=sparse(x,y,z);
B
spy(B)
```

- Logicals

```
A=[0,1;1,0];
B=A>0;
A
B
A(1,1)=0.5;
B(1,1)=0.5;
A
B
```

Chapter 3

Finding roots to nonlinear equations

One of the most important problems we need to solve all the time, in practically any numerical problem, is to find a root to a nonlinear equation. So let us consider a nonlinear function $f(x)$, and suppose we want to find a root, $f(x) = 0$.

The simplest method you could try to find a root of a continuous function, is *bisection*. You evaluate the function at the endpoints of an interval; if the function value have opposite signs, then by the Intermediate Value Theorem there must be a root in the interval. Now halve the interval, and check the endpoints of the two half intervals, and repeat this procedure.

This is a slow algorithm, but it cannot fail (if the initial interval was chosen well). It is good for proofs, but bad for actual applications. E.g., the consecutive estimates of the root do not necessarily converge to the root in a monotonous order: they may jump around. The error estimate is easy, since you halve the interval in each step. You therefore have excellent information how far your iterates are from the root.

3.1 Nonlinear equations: Newton's method

The best known, and also most widely used way to find roots is Newton's method. Let us first focus on $f : \mathbb{R} \to \mathbb{R}$. Suppose that we approximate the function f in the vicinity of a root α,

$$0 = f(\alpha) \approx f(x_0) + f'(x_0)(\alpha - x_0).$$

Then, starting from a guess x_0 we could try to approximate the root α by solving the above equation for the free variable α. Denoting the solution by x_1, we have

$$x_1 = x_0 - f(x_0)/f'(x_0).$$

Newton's method is now given by iterating this scheme:

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

Clearly, if α is a root of f, then α is a fixed point of this map.

Try to read through the following simple implementation of this idea in MATLAB. We will discuss it in class. Note that `varargin` allows you to carry additional parameters through to the function f. We will *not* deal with such issues in this course.

```
function [zero,res,niter]=newton(f,df,x0,tol,nmax,varargin)
%NEWTON Find function zeros.
%   ZERO=NEWTON(FUN,DFUN,X0,TOL,NMAX) tries to find the zero ZERO of the
%   continuous and differentiable function FUN nearest to X0 using the Newton
%   method. FUN and its derivative DFUN accept real scalar input x and returns
%   a real scalar value. If the search fails an error message is displayed.
%   FUN and DFUN can also be inline objects.
%
```

```
%   [ZERO,RES,NITER]= NEWTON(FUN,...) returns the value of the residual in ZERO
%   and the iteration number at which ZERO was computed.

x = x0;
fx = feval(f,x,varargin{:});
dfx = feval(df,x,varargin{:});
niter = 0;
diff = tol+1;
while diff >= tol & niter <= nmax
   niter = niter + 1;
   diff = - fx/dfx;
   x = x + diff;
   diff = abs(diff);
   fx = feval(f,x,varargin{:});
   dfx = feval(df,x,varargin{:});
end
if niter > nmax
    fprintf(['newton stopped without converging to the desired tolerance',...
    'because the maximum number of iterations was reached\n']);
end
zero = x;
res = fx;
```

Newton's method has a number of disadvantages.

- It does not always converge (the worst case is if $f'(x_n) = 0$).
- The error is hard to estimate (when do we need to stop the iteration?)
- The derivative is needed (has to be known)

But there are also some very good advantages.

- The method converges fast
- It is easy to extend to more dimensions: $f : \mathbb{R}^n \to \mathbb{R}^n$
  In this case, $0 = f(x) \approx f(x_0) + Df(x_0) \cdot (x - x_0)$, hence $x_1 \approx x_0 - DF(x_0)^{-1} \cdot f(x_0)$. Note that at every step a system of *linear* equations needs to be solved! This is one good reason why we will need to learn more about solving linear systems later on in the course.

3.1.1 Order of convergence

First recall some notation: little 'o' and big 'O':

$$f(x) = o(g(x)) \text{ as } x \to a \qquad \text{iff } \lim_{x\to a} \frac{f(x)}{g(x)} = 0,$$

$$f(x) = O(g(x)) \text{ as } x \to a \qquad \text{iff } \left|\frac{f(x)}{g(x)}\right| \leq C \text{ for } x \text{ close enough to } a.$$

We say that $x_n \to \alpha$ converges at *order* p if there is a C (independent of n) such that

$$\boxed{|x_{n+1} - \alpha| \leq C|x_n - \alpha|^p}$$

for large n. (And of course we take the largest p for which such a C exists.)

The convergence for the bisection method is roughly $p = 1$, $C = 1/2$.

For Newton, it is much better: $p = 2$ provided $f'(\alpha) \neq 0$ and $f \in C^2$. Then

$$\lim_{n\to\infty} \frac{x_{n+1} - \alpha}{(x_n - \alpha)^2} = \frac{f''(\alpha)}{2f'(\alpha)}.$$

We are going to look at convergence of maps in a bit more generality, namely by considering general iterated maps $x_{n+1} = \phi(x_n)$.

PROPOSITION 3.1.1. *Let ϕ be a continuous function on an interval $[a,b]$, such that $\phi(x) \in [a,b]$ for all $x \in [a,b]$. Suppose that $\phi'(x)$ exists and that there exists a $0 < \kappa < 1$ such that*

$$|\phi'(x)| \leq \kappa \quad \text{for all } x \in [a,b].$$

Then for any $x_0 \in [a,b]$, the sequence defined by $x_{n+1} = \phi(x_n)$, $n \geq 0$ converges to a unique fixed point α, $\phi(\alpha) = \alpha$.

The assumptions in the Proposition imply that ϕ is a contraction on $[a,b]$. Hence $x_n \to \alpha$, where $\phi(\alpha) = \alpha$.

Using Taylor expansions around α, we find

$$x_{k+1} = \phi(x_k) = \phi(\alpha) + \phi'(\alpha)(x_k - \alpha) + o(x_k - \alpha) = \alpha + \phi'(\alpha)(x_k - \alpha) + o(x_k - \alpha).$$

From this we get an *error estimate*. Since $x_{k+1} - \alpha = \phi'(\alpha)(x_k - \alpha) + o(x_k - \alpha)$, we have

$$\boxed{\frac{x_{k+1} - \alpha}{x_k - \alpha} = \phi'(\alpha) + o(1).}$$

The important conclusion is that convergence is order 1 unless $\phi'(\alpha) = 0$. When we want to design algorithms with fast convergence, we have to make them such that $\phi'(\alpha) = 0$, otherwise the algorithm will be slow!

Let us denote the specific choice of Newton's method by

$$\phi_N(x) := x - f(x)/f'(x).$$

We see that $\phi_N'(x) = 1 - 1 + f(\alpha)f''(\alpha)/f'(\alpha)^2 = 0$ if $f'(\alpha) \neq 0$. In other words, if f has a transversal intersection with the x-axis at $x = \alpha$, then Newton's methods convergence fast. But if $f'(\alpha) = 0$, e.g. for $f(x) = x^2$ at $x = 0$, then convergence will be slow.

We can say a little bit more about convergence for ϕ_N:

$$\begin{aligned} x_{k+1} = \phi(x_k) &= \phi(\alpha) + \phi'(\alpha)(x_k - \alpha) + \frac{1}{2}\phi''(\alpha)(x_k - \alpha)^2 + o((x_k - \alpha)^2) \\ &= \alpha + \frac{1}{2}\phi''(\alpha)(x_k - \alpha)^2 + o((x_k - \alpha)^2). \end{aligned}$$

and $\phi_N''(\alpha) = f''(\alpha)/f'(\alpha)$. The order of convergence of Newton's method is thus indeed 2 if $f'(\alpha) \neq 0$ and $f''(\alpha) \neq 0$.

Under these non-degeneracy conditions Newton's method has the property that *the number of correct digits roughly doubles at each step* (!): $|x_{n+1} - \alpha| \approx 10^{-\tilde{C}2^n}$. Namely, if the order of convergence is $p > 1$, then $\log|x_{n+1} - \alpha| \approx \log C + p \log|x_n - \alpha|$, from which you can derive (try it!) that $\log(x_{n+1} - \alpha) = O(p^n)$ as $n \to \infty$.

3.1.2 Estimating the order of convergence

If you do not know the order of convergence of your algorithm, how do you estimate the order of convergence numerically from the iterations?

(1) If you know the real solution α (test problem): $(x_{n+1} - \alpha) \approx C(x_n - \alpha)^p$, then

$$p \approx \frac{\ln(x_{n+1} - \alpha) - \ln C}{\ln(x_n - \alpha)} \approx \frac{\ln(x_{n+1} - \alpha)}{\ln(x_n - \alpha)}.$$

(2) An even better scheme, which does not ignore the constant C, is to use two consecutive iterations as data:

$$\begin{aligned} (x_{n+1} - \alpha) &\approx C(x_n - \alpha)^p, \\ (x_n - \alpha) &\approx C(x_{n-1} - \alpha)^p, \end{aligned}$$

from which we can deduce

$$\boxed{p \approx \frac{\ln(\frac{x_{n+1}-\alpha}{x_n-\alpha})}{\ln(\frac{x_n-\alpha}{x_{n-1}-\alpha})}.}$$

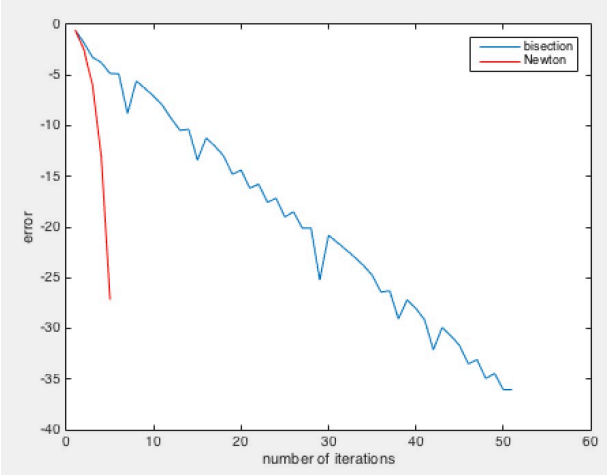

FIGURE 3.1. Comparing rates of convergence to the root $x = \sqrt{2}$ for the function $f(x) = x^2 - 2$, using the bisection (blue) and Newton (red) method.

3.1.3 Stopping criteria

One may also ask the question how to know when to stop the iteration. You cannot stop when x_k is close to α, since you do not know α! Intuitively, it seems reasonable that if the iterations are changing less and less, you should be closer and closer to the root. In other words, if $x_{k+1} - x_k$ becomes small, then so should $x_{k+1} - \alpha$.

So let us compute from the Taylor series again, but now we subtract x_k from both sides,

$$x_{k+1} - x_k = (\alpha - x_k) + \phi'(\alpha)(x_k - \alpha) + o(x_k - \alpha)$$

so that

$$\boxed{x_k - \alpha = \frac{x_k - x_{k+1}}{1 - \phi'(\alpha)} + o(1).}$$

We conclude that indeed you can look at the difference between successive iterations to determine when to terminate, provided you know that $|\phi'(\alpha)| \ll 1$ is. In the case of Newton's method, $\phi'(\alpha) = 0$ for roots α where $f'(\alpha) \neq 0$, so this fine indeed.

3.1.4 Modified Newton

To optimize speed of convergence we are looking for iteration methods $x_{n+1} = \phi(x_n)$ with $\phi'(\alpha) = 0$, guaranteeing order of convergence 2. For standard Newton, in general $\phi_N'(\alpha) = 1 - 1/m$, where m is the order of the zero. To see this, we know that if α is a root of order m, then f may be written as

$$f(x) = C(x - \alpha)^m + o((x - \alpha)^m).$$

So close to α, we have $f(x) \approx C(x - \alpha)^m$, and hence $f'(x) \approx Cm(x - \alpha)^m$. Therefore, $\phi_N(x) = x - \frac{f(x)}{f'(x)} = x - \frac{x-\alpha}{m} + o(x - a)$.
In other words $\phi_N'(\alpha) = 1 - \frac{1}{m}$.

But then we may modify Newton's method to

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

with $M = m$. This is called *modified Newton*, and let us denote it by $\phi_M(x)$. We check that $\phi_M(x) = x - \frac{M}{m}(x - \alpha) + o(x - \alpha)$, hence $\phi'(\alpha) = 1 - \frac{M}{m} = 0$ if $M = m$.

In one of the assignments, you are asked to think about how to estimate m from the iterations, since generally you do not know the order of the root in advance.

Chapter 4

Interpolation

4.1 General remarks

Interpolation is the art of assigning values between given data points. There are plenty of options how you could go about doing this, and as a result the first thing we note is that there is not *one* right method. It all depends on the interpretation or context, and the required smoothness.

To give a flavour, do we want one curve of a specific class (e.g., a polynomial) through all the points, or would we like a low degree polynomial that fits well with the points but doesn't necessarily go through all the points? Or do we wish to have a smooth curve through the points, which is glued from different parts, is simple gluing of straight lines an option?

This last option, piecewise-linear interpolation, is definitely the easiest. It is fast, uses only local data (joining two neighbouring points by a line segment), but of course it is not smooth, merely continuous.

We will see that through n datapoints (if the x coordinates are all different) there is a *unique* polynomial of degree (at most) $n-1$. This result has the advantage that the interpolation will be nicely unique, very smooth, but since we use all the data in one go, it is very "nonlocal".

We could take a middle ground, and try to find local polynomials that match up smoothly. This will result in a curve through all points, the result will be smoother than just using straight lines, and we have used local data.

A completely different approach is e.g. linear regression: one straight line through the cloud of data points.

It all depends on what you want!

4.2 Polynomials

Let us first try to find a polynomial of sufficient high degree through a set of points. The following theorem gives us the answer to this question, in a constructive way no less: it actually constructs the polynomial for you.

THEOREM 4.2.1. *Let $x_0, x_1, \ldots, x_n$ be $n+1$ distinct numbers and let $f(x_k)$ be given for some function f. Then there exists a unique polynomial $P_n(x)$ of degree less or equal to n such that*

$$P_n(x_k) = f(x_k) \quad \text{for all } k = 0, 1, \ldots, n.$$

In fact,

$$P_n(x) = f(x_0)L_{n,0}(x) + \cdots + f(x_n)L_{n,n}(x),$$

where

$$L_{n,k}(x) = \prod_{i=0, i\neq k}^{n} \frac{(x - x_i)}{(x_k - x_i)} = \frac{(x-x_0)\cdots(x-x_{k-1})(x-x_{k+1})\cdots(x-x_n)}{(x_k-x_0)\cdots(x_k-x_{k-1})(x_k-x_{k+1})\cdots(x_k-x_n)}$$

are the Legendre characteristic polynomials.

The idea of the proof is to evaluate $P_n(x_k)$,

$$P_n(x_k) = \sum_{i=0}^{n} f(x_i) L_{n,i}(x_k).$$

The only nonzero term in this sum is for $i = k$, but there we have

$$P_n(x_k) = f(x_k) \prod_{i=0, i\neq k}^{n} \frac{(x_k - x_i)}{(x_k - x_i)} = f(x_k).$$

So $L_{n,k}(x_j)$ has the property that

$$L_{n,k}(x_j) = \begin{cases} 0 & j \neq k, \\ 1 & j = k. \end{cases}$$

Also note that $f(x)$ is itself a polynomial of degree n, then $P_n(x) \equiv f(x)$ for all x.

The following Theorem on the error in the polynomial interpolation is instructive.

THEOREM 4.2.2. *Let $x_0, \ldots, x_n$ be distinct points in $[a, b]$, and let $f \in C^{n+1}([a, b]; \mathbb{R})$. Let $P_n(x)$ be the unique polynomial of degree $\leq n$ through the points $(x_k, f(x_k))$. Then for each $x \in [a, b]$ there exists a (generally unknown) $\xi(x) \in (a, b)$ such that*

$$f(x) - P_n(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!}(x - x_0)(x - x_1)\cdots(x - x_n).$$

Compare the result to a Taylor series (around $x = x_0$):

$$f(x) - T_n f(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}(x - x_0)^{n+1}.$$

The difference in the product terms in the error estimate resides in the fact that to make $P_n(x)$ we needed $n+1$ *different* points through which the polynomial had to be made, whereas in Taylor's theorem, you only use the behaviour of $f(x)$ in the vicinity of *one* base point x_0 (but you need derivatives).

The important conclusion from this theorem is that we *cannot* conclude that increasing the degree of $P_n(x)$ will improve the error. In fact

$$\lim_{n\to\infty} \max_{x\in[a,b]} \left| \frac{f^{(n+1)}(\xi(x))}{(n+1)!} \prod_{i=0}^{n} (x - x_i) \right| \to \infty$$

for some functions.

There does exist an optimal way of choosing the discretisation points $x_0, \ldots, x_n$ in such a way that the above error converges nicely. This follows from an analysis (not provided here) of how one can to minimise the product $\prod_{i=0}^{n}(x - x_i)$. These discretisation points are called Chebyshev-Gauss-Lobatto points, and the corresponding polynomials $p_n(x)$ Chebyshev polynomials. The points are constructed by setting

$$x_i = \frac{a+b}{2} + \frac{b-a}{2}\hat{x}_i \qquad \text{where} \qquad \hat{x}_i = -\cos\left(\frac{\pi i}{n}\right).$$

In words, you take a semi-circle with center in the midpoint of $[a, b]$, and divide the semi-circle in n equal radial slices. Then you take the x-coordinates of the points on the semi-circle of each slice.

These Chebyshev polynomials $p_n(x)$ have the property that for any C^k function f, we have the following convergence result: $|f(x) - p_n(x)| \leq Cn^{-k}$, where the constant C depends on f.

In practical situations, you are unlikely to have much influence on the choice of x_i (e.g. if these are data points from an experiment). But it is good to know that the choice of interpolation points makes a difference (if you have a computable function that is reasonably smooth).

4.3 Piecewise linear interpolation

Of course, it is easy to construct a piecewise linear approximation to a set of points. Formally, we can apply Theorem 4.2.1 with $n = 1$ on each discretisation interval. For $f \in C^2$ approximated by a piecewise linear interpolation f_{PL} on a uniform grid this gives an error estimate $|f(x) - f_{\mathrm{PL}}(x)| \leq \frac{1}{8}(\Delta x)^2 \max |f''|$, where Δx is the grid size.

4.4 Splines

The idea of splines is to glue pieces of 3rd order polynomials which match up to the 2nd derivative. In this way you get a C^2 curve through all the points, using only local information (well, not quite, as we will see below).

How do you calculate a spline?
We will use equidistant points here, for convenience, and set $s = x - x_k$, $h = x_{k+1} - x_k$. Let $y_{k+1} - y_k = f(x_{k+1}) - f(x_k) = h\delta_k$ for $k = 1, \ldots, n$. *Given* are the y_k (and thus δ_k). We need to find polynomials

$$P_k(s) = A_k(\frac{s}{h})^3 + B_k(\frac{s}{h})^2 + C_k\frac{s}{h} + D_k$$

such that

$$P_k(0) = y_k,\ P_k(h) = y_{k+1},\ P_k'(0) = d_k,\ P_k'(h) = d_{k+1},$$

where $d_1, \ldots, d_n$ are the *unknowns*.

This leads to the system

$$\left\{\begin{array}{rrrrcl} & & & D_k & = & y_k \\ A_k & +B_k & +C_k & +D_k & = & y_{k+1} \\ & & C_k & & = & hd_k \\ 3A_k & +2B_k & +C_k & & = & hd_{k+1} \end{array}\right.$$

from which we see that $D_k = y_k$, $C_k = hd_k$, $B_k = h(-d_{k+1} - 2d_k + 3\delta_k)$ en $A_k = h(d_{k+1} + d_k - 2\delta_k)$.

The condition $P_{k-1}''(h) = P_k''(0)$ implies

$$6A_{k-1}h^{-2} + 2B_{k-1}h^{-2} = 2B_kh^{-2},$$

and from that we obtain the equation for the unknowns δ_k:

$$d_{k-1} + 4d_k + d_{k+1} = 3\delta_{k-1} + 3\delta_k$$

for $k = 2, \ldots, n-1$.

We can close the system by prescribing d_1 and d_n (the derivatives in the endpoints), but other choices are also possible. A well-known choice is the "not-a-knot" condition: we demand $P'''(x_2)$ to be continuous, i.e. one polynomial on $[x_1, x_3]$, hence x_2 is not a "knot" (interpolation point) and idem ditto for $P'''(x_{n-1})$.

The latter choice leads to $A_2 = A_1$, i.e. $d_1 - d_3 = 2\delta_1 - 2\delta_2$, and we obtain the linear system

$$\begin{pmatrix} 1 & 0 & -1 & & & \\ 1 & 4 & 1 & & & \\ & 1 & 4 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & 4 & 1 \\ & & & -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \\ d_n \end{pmatrix} = \begin{pmatrix} 2\delta_1 - 2\delta_2 \\ 3\delta_1 + 3\delta_2 \\ 3\delta_2 + 3\delta_3 \\ \vdots \\ 3\delta_{n-2} + 3\delta_{n-1} \\ -2\delta_{n-2} + 2\delta_{n-1} \end{pmatrix},$$

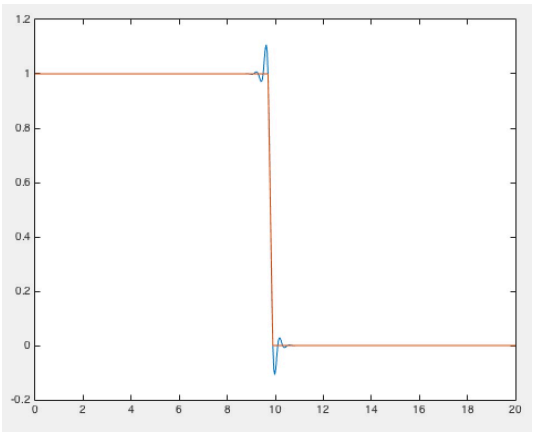

FIGURE 4.1. An example of a function (in orange) which is not well-approximated using splines (in blue).

which is equivalent with the tridiagonal system

$$\begin{pmatrix} 1 & 2 & & & & \\ 1 & 4 & 1 & & & \\ & 1 & 4 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & 4 & 1 \\ & & & & 2 & 1 \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \\ d_n \end{pmatrix} = \begin{pmatrix} \frac{5}{2}\delta_1 + \frac{1}{2}\delta_2 \\ 3\delta_1 + 3\delta_2 \\ 3\delta_2 + 3\delta_3 \\ \vdots \\ 3\delta_{n-2} + 3\delta_{n-1} \\ \frac{1}{2}\delta_{n-2} + \frac{5}{2}\delta_{n-1} \end{pmatrix}.$$

Tridiagonal systems are particularly easy and fast to solve, using LU-decomposition, see §**??**. This is one of the reasons splines are very popular. Another reason is that human eyes/brains enjoy curves without jumps in curvature (the second derivatives).

Reduction to a tridiagonal system is *not* true if you close the system by making it periodic (which would be another reasonable first guess to close the system of equations for the d_i):

$$\begin{pmatrix} 4 & 1 & & & & 1 \\ 1 & 4 & 1 & & & \\ & 1 & 4 & 1 & & \\ & & \ddots & \ddots & \ddots & \\ & & & 1 & 4 & 1 \\ 1 & & & & 1 & 4 \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \\ d_n \end{pmatrix} = \begin{pmatrix} 3\delta_{n-1} + 3\delta_1 \\ 3\delta_1 + 3\delta_2 \\ 3\delta_2 + 3\delta_3 \\ \vdots \\ 3\delta_{n-2} + 3\delta_{n-1} \\ 3\delta_{n-1} + 3\delta_1 \end{pmatrix}.$$

Splines have better convergence than piecewise linear interpolations, also for the derivatives (provided the original function was sufficiently smooth).

Of course splines are not always brilliant (note also the way you compute it in MATLAB), as the following example shows (see Figure 4.1)

```
x=linspace(0,20,100)';
y=ones(size(x));
y(end/2:end)=0;
t=linspace(0,20,500)';
q=spline(x,y,t);
plot(t,q,x,y)
```

4.4.1 Arclength parametrisation

If you want to interpolate a curve rather than a function, you can use pseudo-arclength parametrisation:

- points $\{(x_i, y_i)\}_{i=1}^n$ in the plane
- "arclength" parametrisation (pseudo!) with parameter t: $t_1 = 0$ and
$$t_{i+1} = t_i + \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}.$$
- And then put splines through (t_i, x_i) and (t_i, y_i).

4.5 Least square method

A completely different kind of interpolation is to find low-degree polynomials, such as straight lines, which should 'fit the data as well as possible'. The simplest such method is the one called Least Squares (which was also covered in your Linear Algebra course!) or Linear Regression (in Statistics).

Given data points (x_i, y_i), look for the line $y = Cx + D$ such that $y_i = Cx_i + D$. In matrix notation, we have

$$\begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix} \begin{pmatrix} C \\ D \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}.$$

We want to find the "best" solution to the overdetermined system $Ax = b$, i.e., the x which minimises $|Ax - b|_2$ (the Euclidean norm).

This best solution is found by MATLAB with `x=A\b` or `pinv(A)*b`. If A has a kernel, then `pinv(A)*b` is the smallest x that minimises $|Ax - b|_2$, whereas `x=A\b` minimises with many zero elements in the vector.

See Chapter 5 for more details.

4.6 A tiny bit on minimisation

The least square method is a very special case of a minimisation problem. In this course, we won't deal with minimisation problems in general! It *is* interesting in practise obviously: optimisation of business processes, profit, optimal solutions to routing problems, you name it. But the field is vast, and full of problems, for instance whether to do a local or global search for a minimum or maximum. Local problems might home in on a local minimum which is far from globally optimal, but global searches might give you plenty of these, and true global search is usually impossible.

It is good to know that there is a MATLAB function

```
fminsearch(@(x) x^2-2*x,0)
fminsearch(@(x) x(1)^2+2*x(1)+x(2)^2-2*x(2),[0;0])
```

Only use this if there is no smarter way (as in least square method).

Chapter 5

The Least Square Method and Singular Value Decomposition

5.1 The least square method

We repeat the basics. The aim is to find the best fit to data points (x_i, y_i), using some model with linear coefficients. The most familiar example is of course to fit a straight line $y = Cx + D$ such that $y_i = Cx_i + D$.

In matrix notation we should find C and D such that

$$\begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix} \begin{pmatrix} C \\ D \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}.$$

The "best" solution of $Ax = b$ is found by MATLAB with `x=A\b` or `pinv(A)*b`. This actually finds the x which minimises $|Ax - b|_2$. If A has a kernel, there is a difference between the two commands: `pinv(A)*b` is the x that minimises $|Ax - b|_2$, whereas `x=A\b` minimises with many zero elements in the vector. We will now look at how this minimising works.

The method extends directly to general least square formulation, for *linear* objective functionals (i.e., those that are linear in the coefficients which need to be found!),

$$f(x) = c_1 f_1(x) + \cdots + c_n f_n(x).$$

If we wish to fit to the known data (x_i, y_i), then the unknowns c_j should be chosen such that $f(x_i) = \sum_j c_j f_j(x_i)$. Defining the matrix $A_{ij} = f_j(x_i)$, these equations still have the form $Ac = y$ (with A with many more rows than columns).

Geometrically ($n < m$), the best possible Ax is to orthogonally project b onto $R(A)$. That projection is given by $P_A = A(A^T A)^{-1} A^T$ (as you have seen in Linear Algebra 1, including the proof of this statement). There is one proviso, A has to have maximal rank (n). This means that the functions f_j should be linear independent (in the points x_i)). Hence $Ax = P_A b$, so that for $\mathrm{rank}(A) = n < m$

$$x = (A^T A)^{-1} A^T b.$$

To understand this more generally, we now discuss the SVD, which has other useful applications as well.

5.2 The Singular Value Decomposition

You may remember that diagonalisation of symmetric (or self-adjoint for complex-valued matrices) matrices was particularly fruitful. If A is square and self-adjoint, (i.e., $A^* = A$, which is `A'==A` in MATLAB), then A may decomposed as

$$A = U \Lambda U^{-1},$$

where Λ is a diagonal matrix with eigenvalues and U an orthogonal, or unitary, matrix of (normalised) eigenvectors. Since U is unitary, $U^{-1} = U^*$, i.e., $|Ux| = |x|$ for all x (in other words, the eigenvectors are orthonormal). For every self-adjoint linear map $A : \mathbb{R}^n \to \mathbb{R}^n$ there is thus a basis of $\mathbb{R}^n$ with respect to which A is diagonal (mapping basis vectors to multiples of themselves).

The SVD generalizes this idea of orthogonal diagonalisation to arbitrary (not even square) matrices. Let A be an $m \times n$ matrix. Then its Singular Value Decomposition is given by

$$\boxed{A = U\Sigma V^*,}$$

where U is $m \times m$ and V is $n \times n$, both are orthonormal (or complex unitary, $U^*U = I$) and Σ is $m \times n$ and has only diagonal elements.

The geometric meaning of all this is that for every linear map $A : \mathbb{R}^n \to \mathbb{R}^m$ you can find *two* orthogonal/normal bases, one of $\mathbb{R}^n$ and one of $\mathbb{R}^m$ (even if $n = m$ the two bases will be different), with respect to which the map is diagonal, i.e. maps basis vectors onto multiples of basis vectors.

We will not go into algorithms for calculating this decomposition in practise.

More precise notation: $\Sigma_{ij} = \sigma_i \delta_{ij}$, $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_{\min(m,n)} \geq 0$, so that

$$\Sigma = \begin{pmatrix} \sigma_1 & 0 & \cdots & 0 \\ 0 & \sigma_2 & & 0 \\ \vdots & & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_n \\ 0 & 0 & \cdots & 0 \\ \vdots & & & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix} \qquad \text{or} \qquad \Sigma = \begin{pmatrix} \sigma_1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & \sigma_2 & & 0 & 0 & & 0 \\ \vdots & & \ddots & \vdots & \vdots & & \vdots \\ 0 & 0 & \cdots & \sigma_m & 0 & \cdots & 0 \end{pmatrix},$$

$U = (u_1, u_2, \ldots, u_m)$ and $V = (v_1, v_2, \ldots, v_n)$ are systems of orthonormal vectors.

To get a sense of the shape of the matrices, consider

$$\begin{pmatrix} a & a \\ a & a \\ a & a \end{pmatrix} = \begin{pmatrix} u & u & u \\ u & u & u \\ u & u & u \end{pmatrix} \begin{pmatrix} \sigma & \sigma \\ \sigma & \sigma \\ \sigma & \sigma \end{pmatrix} \begin{pmatrix} v & v \\ v & v \end{pmatrix}.$$

U and V are real if A is real.

The remarkable fact is that such a decomposition always exists!

Let k be such that $\sigma_1, \ldots, \sigma_k > 0$ and $\sigma_{k+1} = 0$. In other words: $k = \operatorname{rank}(A)$.
Define $\Sigma^\dagger$ as the $(n \times m)$ matrix $\operatorname{diag}(\frac{1}{\sigma_1}, \ldots, \frac{1}{\sigma_k}, 0, \ldots, 0)$.
Hence $\Sigma\Sigma^\dagger = \operatorname{diag}(1, 1, 1, 1, 1, 0, 0, 0)$ with k 1's and $n - k$ 0's.

Also note that $A^*A = V\Sigma^T\Sigma V^*$ is self-adjoint, hence we must have the following. Let the singular values Σ be the square roots of the eigenvalues of A^*A, and let V be the corresponding eigenvectors of A^*A. Let $\tilde{U} = AV\Sigma^\dagger$; this is *the* candidate for U. Compute

$$\begin{aligned} \tilde{U}^*\tilde{U} &= {\Sigma^\dagger}^T V^* A^* A V \Sigma^\dagger \\ &= {\Sigma^\dagger}^T V^* V \Sigma^T \Sigma V^* V \Sigma^\dagger \\ &= {\Sigma^\dagger}^T \Sigma^T \Sigma \Sigma^\dagger \\ &= \begin{bmatrix} 1_k & 0 \\ 0 & 0 \end{bmatrix}. \end{aligned}$$

Hence the first k columns in $\tilde{U}$ are orthonormal.

If $k < m$, U is not yet square, and we need to "complete" it with any extra orthonormal vectors.

To be more precise, let $\Sigma_k = \operatorname{diag}(\sigma_1, \ldots, \sigma_k)$ and $V_k = (v_1, v_2, \ldots, v_k)$. Notice that Σ_k is square, while V_k is not (necessarily). Define $U_k = AV_k\Sigma_k^\dagger$. Then indeed, with $\overline{U}_k$ and $\overline{V}_k$ orthonormal completions (e.g.

use the Gram-Schmidt process from linear algebra),

$$U\Sigma V^* = \begin{bmatrix} U_k \, \overline{U}_k \end{bmatrix} \begin{bmatrix} \Sigma_k & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} V_k^* \\ \overline{V}_k^* \end{bmatrix} = U_k \Sigma_k V_k^* = A V_k \Sigma_k^\dagger \Sigma_k V_k^* = A.$$

To compute the SVD of a matrix in MATLAB use

```
[U,S,V]=svd(A)
```

The SVD has a long list of beautiful properties, making it one of the most widely used decompositions in numerical linear algebra.

- $\|A\|_2 = \|\Sigma\|_2 = \sigma_1$.
- Consider the *Frobenius norm*

  $$\|A\|_F = \sqrt{\sum_{i,j} |A_{ij}|^2} = \sqrt{\sum_i \sigma_i^2}.$$

  The Frobenius norm is the root of the sum of the squares of the lengths of the column vectors. The Frobenius norm is the root of the sum of the squares of the lengths of the row vectors. And it is (hence) invariant under multiplication (left or right) by unitary matrices.
- If A is square, then $\det(A) = \prod_{i=1}^n \sigma_i$, hence $\det(A) = 0 \iff \sigma_n = 0$.
- The SVD may be used to compute efficiently the columns and null spaces of a matrix. Let $\sigma_k \neq 0$, $\sigma_{k+1} = 0$, then
  - $R(A) = \mathrm{sp}(u_1, \ldots, u_k)$
  - $N(A) = \mathrm{sp}(v_{k+1}, \ldots, v_n)$.
- The SVD may be made more concise by focusing on the non-zero singular values. Suppose there are k of these. Then the matrices U and V may be reduced to U_k and V_k with k columns and k rows respectively, The matrices U_k, Σ_k (the square diagonal matrix with k non-zero singular values) and V_k give a *reduced SVD*, which satisfies

  $$A = U\Sigma V^* = U_k \Sigma_k V_k^*.$$

The SVD may also be used to define a generalisation of the inverse of a matrix for non-square matrices, called the *pseudo-inverse*. The pseudo-inverse is defined to be

$$A^\dagger = V \Sigma^\dagger U^*$$

- This is equivalent to $A^\dagger = V_k \Sigma_k^{-1} U_k^*$ using the reduced SVD.
- $A^\dagger A = \mathrm{diag}(1_k, O_{n-k})$ and $AA^\dagger = \mathrm{diag}(1_k, O_{m-k})$.
- A^*A (an $(n \times n)$ matrix) is invertible *provided* A has maximal rank (n).
- And we have $A^\dagger = (A^*A)^{-1}A^*$ *provided* $\mathrm{rank}(A) = n \leq m$:

  $$A = U\Sigma_n V_n^*$$
  $$A^* = V_n \Sigma_n U^*$$
  $$A^*A = V_n \Sigma_n^2 V_n^*$$
  $$(A^*A)^{-1} = V_n \Sigma_n^{-2} V_n^*$$
  $$(A^*A)^{-1}A^* = V_n \Sigma_n^{-1} U^* = A^\dagger$$

- And (if $\mathrm{rank}(A) = n$) then $A^\dagger$ is on $R(A)$ the inverse of $A : \mathbb{R}^n \to R(A)$: when $x = Ay$, then $AA^\dagger x = x$, and $A^\dagger A = 1_n$.
- In fact, $A^\dagger$ truly generalises the inverse: $A^\dagger = A^{-1}$ if $\mathrm{rank}(A) = n = m$.

Finally there is a direct connection between $A^\dagger$ and solving least squares problems. The least squares solution to $Ax = b$ is the orthogonal projection of b onto the column space of A, the range of A, $R(A)$. This projection is given by $P_A = A(A^TA)^{-1}A^T$, *provided* A has maximal rank (n).

Hence $Ax = P_A b$, so that for $\text{rank}(A) = n < m$

$$\begin{aligned} x &= A^\dagger A (A^T A)^{-1} A^T b \\ &= (A^T A)^{-1} A^T A (A^T A)^{-1} A^T b \\ &= (A^T A)^{-1} A^T b \\ &= A^\dagger b. \end{aligned}$$

In the complex case $x = (A^* A)^{-1} A^* b$.

If $\text{rank}(A) < n$, then you can obviously add elements of the kernel of A to x. But still, the *smallest* best solution is $x = A^\dagger b$ (proof to follow). Calculate the least square solution using the SVD (and pseudo inverse) [not by solving $A^* Ax = A^* b$, since $A^* A$ is much worse conditioned than A.]

THEOREM 5.2.1. *The smallest x that minimises $|Ax - b|$ is $x = A^\dagger b$.*

PROOF. Write $A = U\Sigma V^*$. Define k so that $\sigma_k > 0$ and $\sigma_{k+1} = 0$. Let $y = V^* x$, so that $|y| = |V^* x| = |x|$. Also $|U^*(Ax - b)| = |Ax - b|$.

Hence we can reformulate our problem as: find the smallest y $(= V^* x)$ such that $|(Ax - b)| = |(U^* AVy - U^* b)|$ is minimal.

In other words, the smallest y such that

$$|\Sigma y - U^* b|^2 = \sum_{i=1}^{k} (\sigma_i y_i - (U^* b)_i)^2 + \sum_{i=k+1}^{m} ((U^* b)_i)^2$$

is as small as possible.

It is immediate that in the minimum $y_i = (U^* b)_i / \sigma_i$ for $i = 1, \ldots, k$, and for the smallest of such y's all other components of y are equal to 0.

We conclude that the solution is $y = \Sigma^\dagger U^* b$ and $x = V\Sigma^\dagger U^* b = A^\dagger b$. □

5.3 Approximating matrices by ones of lower rank

The following theorem states that the SVD may be fruitfully applied to compute the best possible approximation of a certain matrix of a specified rank that is less than the rank of the original matrix.

THEOREM 5.3.1. *The best approximation of A by a rank $\leq l$ matrix is $A_l = U_l \Sigma_l V_l^*$.*

This means that A_l minimises the distance of to A in the Frobenius norm (and in the 2-norm) in the class of rank l matrices.

Here is a cool example, illustrating the point. The resulting images are found in Figure 5.1. In practise there are beter possibilities for data reduction. It does have (many) applications in signal processes and statistics. The technique known as *Principle Component Analysis* is essentially exactly this.

```
load detail

figure(1)

subplot(2,2,1)
image(X)
colormap(gray(64))
axis image, axis off
r=rank(X)
title(['rank = ',int2str(r)])

[U,S,V]=svd(X,0);
sigma=diag(S);
```

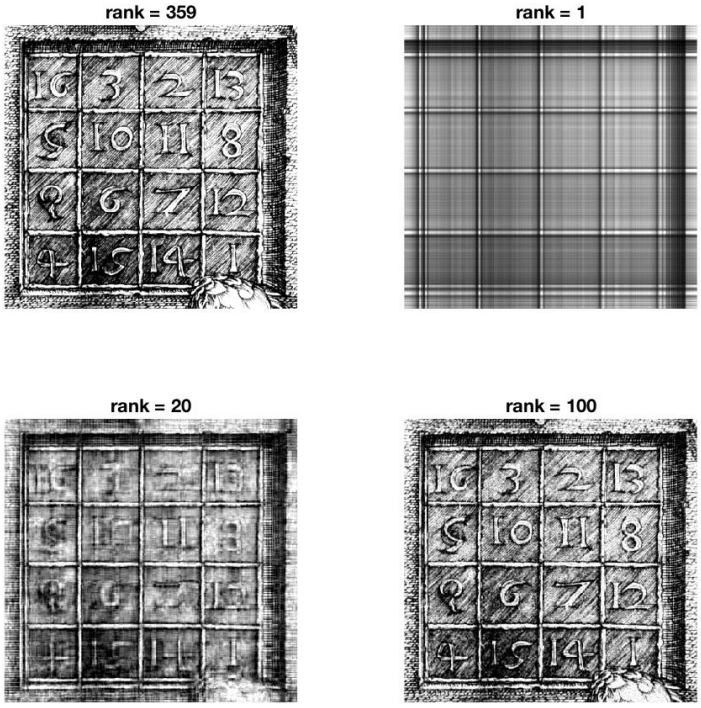

FIGURE 5.1. Finding lower-rank approximations to a matrix filled with grey values making up an image. The original matrix has full rank equal to 359. Note that the rank 20 matrix already gives a good impression of the original image.

```
subplot(2,2,2)
R1=U(:,1)*S(1,1)*V(:,1)';
image(R1)
axis image, axis off
title(['rank = 1'])

subplot(2,2,3)
R20=U(:,1:20)*S(1:20,1:20)*V(:,1:20)';
image(R20)
axis image, axis off
title(['rank = 20'])

subplot(2,2,4)
R100=U(:,1:100)*S(1:100,1:100)*V(:,1:100)';
image(R100)
axis image, axis off
title(['rank = 100'])
```

Finally, here is a proof of Theorem 5.3.1.

PROOF. We have

$$\|A - A_l\|_F^2 = \|\Sigma - \Sigma_l\|_F^2 = \sum_{i=l+1}^{\min(m,n)} \sigma_i^2.$$

Take an arbitrary matrix B of rank $\leq l$. Write $B = XY$ where X is an $(m \times l)$ matrix and Y is an $(l \times n)$ matrix (e.g. via the SVD of B). We will first optimise Y, and then we will optimise X.

We are thus first looking for the Y that minimizes $\|A - XY\|_F^2$ for fixed X. Since the square of the Frobenius norm is equal to sum of the squares of the length of the columns, we can minimise the length of each column separately: minimize $|a_i - Xy_i|^2$ for $i = 1, \ldots, n$.

Observe that the latter is the same as "least squares", hence the optimal choice $\hat{y}_i = (X^*X)^{-1}X^*a_i$ and $\hat{Y} = (X^*X)^{-1}X^*A$. In summary, so far we have

$$\|A - B\|_F^2 = \|A - XY\|_F^2 \geq \|A - X\hat{Y}\|_F^2 = \|A - X(X^*X)^{-1}X^*A\|_F^2 = \|A - P_XA\|_F^2 = \|A\|_F^2 - \|P_XA\|_F^2,$$

because column-wise $|(I - P_X)a_i|^2 = |a_i|^2 - |P_Xa_i|^2$.

Next we look for the best projection P_X which *maximizes* $\|P_XA\|_F^2$ over all possible $(m \times l)$ matrices X. We have

$$\|P_XA\|_F^2 = \|P_XU\Sigma\|_F^2 = \sum_{i=1}^{\min(m,n)} \sigma_i^2 |P_Xu_i|^2 = \sum_{i=1}^{\min(m,n)} \sigma_i^2 |p_i|^2,$$

where $p_i = P_Xu_i$ and hence $|p_i| \leq 1$ (projection of vector with length 1), and $\sum_{i=1}^n |p_i|^2 = \|P_XU\|_F^2 = \|P_X\|_F^2 = l$ [since for orthogonal projections $P_X^* = P_X$, hence $P_X = U\Lambda U^*$, with $\lambda = \mathrm{diag}(1, , \ldots, 1, 0, \ldots, 0)$, where the number of 1-s equals $\dim(X) = \mathrm{rank}(P_X) = l$, and thus $\|P_X\|_F^2 = \|\Lambda\|_F^2 = l$.]

The maximum is attained if the first l of the p_j-s are equal to 1, and the other ones are 0. In conclusion:

$$\|A - B\|_F^2 \geq \|A\|_F^2 - \|P_XA\|_F^2 \geq \|A\|_F^2 - \sum_{i=1}^{l} \sigma_i^2 = \sum_{i=l+1}^{\min(m,n)} \sigma_i^2 = \|A - A_l\|_F^2.$$

□

Chapter 6

The Fast Fourier Transform

Fast Fourier Transforms are used in many kinds of applications, particularly in signal processing, and image analysis. The acronym FFT actually stands for Finite Fourier Transform, but the FFT method is equally well known for being fast. The FFT is used to extract frequencies from a more or less periodic signal.

To get an idea of the method, let us start with an example, of sunspot activity, using some data from the web, to be found at `http://solarscience.msfc.nasa.gov/greenwch/spot_num.txt`. Table 1 gives some sample data on average number of sunspots (the third column) in different months.

YEAR	MON	SSN	DEV
1749	1	58.0	24.1
1749	2	62.6	25.1
1749	3	70.0	26.6
1749	4	55.7	23.6
1749	5	85.0	29.4
1749	6	83.5	29.2
1749	7	94.8	31.1
1749	8	66.3	25.9
1749	9	75.9	27.7
1749	10	75.5	27.7

TABLE 1. Some example data on sunspot activity, which has been collected on a monthly basis since 1749. The data above are for the first ten months of that year.

The following bit of code takes the sunspots data (already available in a file after copy-pasting it from the above URL), makes a time variable t calculated in years. The sunspot data is column three, and is put in variable a. The `polyfit` line calculates a linear fit to the data, see the left plot in Figure 6.1. We subtract the trend in the data from the data itself, to get a nearly periodic signal. Then we perform the FFT analysis using `fft`, and plot the extent to which certain frequencies are present in this data. The resulting plot can be found in the right plot in Figure 6.1.

```
sunspots=load('sunspots.txt');
t=sunspots(:,1)+sunspots(:,2)/12;
a=sunspots(:,3);
N=length(a)
N2=floor((N-1)/2)
plot(t,a)
c=polyfit(t,a,1)
trend=polyval(c,t);
plot(t,a,t,trend)
```

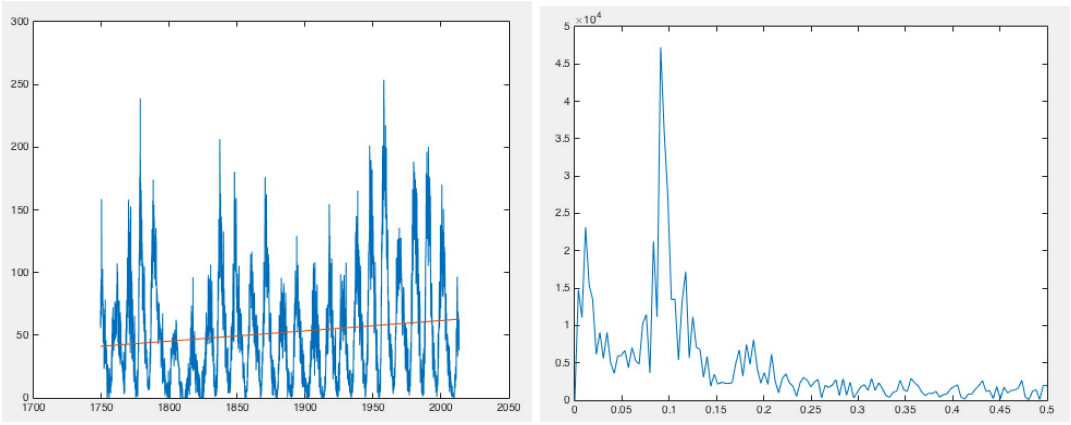

FIGURE 6.1. An illustration of using FFT on sunspots data. The left plot shows the raw data, which shows a clear periodic pattern that we wish to characterise, and also a clear positive trend. After subtracting the trend, we find that the FFT of the data is as shown on the right. The highest peak occurs at a frequency of just less than 0.1, which means there is an oscillation in the data with period just over 10 years. This corresponds nicely with eye-balling the picture on the left, which indeed shows a peak once every 12 years or so.

```
y=a-trend;
Y=fft(y);
Fs=12;
f=(0:N2)'*Fs/N;
Y2=Y(1:N2+1);
plot(f,abs(Y2))
axis([0 0.5 0 50000])
```

The details of the MATLAB computation will become clear later on.

6.1 Finite Fourier Transforms

The FFT is meant for periodic functions. The FFT algorithm is one of the most important mathematical algorithms (used extremely frequently in signal analysis)! Just why it is so fast and what is so special about it will hopefully be clear by the end of this section.

For $f(t)$ periodic on $[0, L]$ (i.e. period L, frequency $\frac{1}{L}$) we write it as a formal sum of complex exponentials (we could have used sines and cosines as well of course),

$$\boxed{f(t) = \sum_{k\in\mathbb{Z}} c_k e^{\frac{2\pi i k}{L} t}.}$$

Then Fourier analysis tells us that the coefficients c_k are found by

$$\boxed{c_k = \frac{1}{L}\int_0^L f(t) e^{-\frac{2\pi i k}{L} t} dt.}$$

Note that $c_k \in \mathbb{C}$! Also, we can see straight away that $c_{-k} = c_k^*$ for $f(t)$ real.

The bi-infinite vector $\{c_k\}_{k=-\infty}^{\infty}$ is called the Fourier transform of f. (for a periodic function; there is a different Fourier transform for functions defined on $\mathbb{R}$.) The c_k are also called *Fourier coefficients*. The interpretation of the coefficients is: $2|c_k|$ is the amplitude of the frequency k/L in the signal f.

The coefficients have a number of nice properties. Write $c_k(f) = \frac{1}{L}\int_0^L f(t)e^{-\frac{2\pi ik}{L}}dt$, then

$$\begin{aligned} c_k(f') &= \frac{1}{L}\int_0^L f'(t)e^{-\frac{2\pi ik}{L}t}dt \\ &= \frac{1}{L}\frac{2\pi}{L}ik\int_0^L f(t)e^{-\frac{2\pi ik}{L}t}dt \\ &= \frac{2\pi}{L}ikc_k(f) \\ c_k(f'') &= -(\tfrac{2\pi}{L})^2k^2c_k(f). \end{aligned}$$

(This means that one can sometimes turn ODE problems into algebraic problems!) It also turns convolutions $(f\star g)(x) = \int_0^L f(y)g(x-y)dy$ into products:

$$\begin{aligned} c_k(f\star g) &= \frac{1}{L}\int_0^L\int_0^L f(y)g(x-y)e^{-\frac{2\pi ik}{L}x}dx\,dy \\ &= \frac{1}{L}\int_0^L\int_y^{L+y} f(y)g(z)e^{-\frac{2\pi ik}{L}y}e^{-\frac{2\pi ik}{L}z}dz\,dy \\ &= Lc_k(f)c_k(g). \end{aligned}$$

Now for numerical purposes, we numerically need to approximate all integrals, and hence we turn them into finite sums. Since we would like to get a MATLAB implementation later on, we start all relevant counters at 1.

Let $\Delta t = \frac{L}{N}$, $t_j = \frac{L}{N}(j-1)$ and $f_j = f(t_j)$. Then as approximation for Nc_{k-1}, we have

$$\frac{N}{L}\sum_{j=1}^N f(t_j)e^{-\frac{2\pi i(k-1)}{L}t_j}\Delta t = \sum_{j=1}^N f_j e^{-\frac{2\pi i}{N}(k-1)(j-1)} \equiv F_k.$$

And this is *precisely* what `F=fft(f)` does.

The relation between the true coefficients c_m and the approximations `F=fft(f)` given by

$$\boxed{F_m = \sum_{j=1}^N f_j e^{-\frac{2\pi i}{N}(m-1)(j-1)}}$$

is as follows. First, we have

$$c_m \approx \frac{1}{N}F_{m+1}, \text{ for } m = 0,\ldots,N/2,$$

and

$$c_{-m} \approx \frac{1}{N}F_{N+1-m}, \text{ for } m = 1,\ldots,N/2,$$

since (using $\omega = e^{-\frac{2\pi i}{N}}$)

$$\begin{aligned} c_{-m} &\approx \frac{1}{N}\sum_{k=1}^N f(k)\omega^{(k-1)(-m)} \\ &= \frac{1}{N}\sum_{k=1}^N f(k)\omega^{(k-1)(-m+N)} \\ &= \frac{1}{N}\sum_{k=1}^N f(k)\omega^{(k-1)(N-m+1-1)} \\ &= \frac{1}{N}F_{N+1-m}. \end{aligned}$$

In summary, the ordering in the FFT of the approximations of the Fourier coefficients is as follows. If N is even,

$$\boxed{\begin{aligned} F &\approx N(c_0, c_1, \ldots, c_{N/2-1}, \tfrac{1}{2}(c_{N/2} + c_{-N/2}), c_{-N/2+1}, \ldots, c_{-2}, c_{-1}) \\ &\approx N(c_0, c_1, \ldots, c_{N/2-1}, c_{N/2}, c^*_{N/2-1}, \ldots, c^*_2, c^*_1), \qquad \text{for real } f, \end{aligned}}$$

with $F_{N/2+1} = \sum_{j=1}^{N} f_j(-1)^{j-1} \approx N c_{\pm N/2}$. And if N is odd,

$$\boxed{\begin{aligned} F &\approx N(c_0, c_1, \ldots, c_{(N-1)/2}, c_{-(N-1)/2}, \ldots, c_{-2}, c_{-1}) \\ &= N(c_0, c_1, \ldots, c_{(N-1)/2}, c^*_{(N-1)/2}, \ldots, c^*_2, c^*_1) \qquad \text{for real } f. \end{aligned}}$$

We note that the approximation $c_m \approx \frac{1}{N} F_{m+1}$ is not very good for m near $N/2$. Moreover, the FFT using N data points contains essentially *no* information on frequencies larger than $\frac{N}{2}\frac{1}{L}$ (this phenomenon is called "aliasing").

For real-valued f the first and second half of the FFT contain the same information. Indeed, there is a symmetry, given by $F^*_{N-m} = F_{m+2}$ for real f, since ($\omega_N = e^{-\frac{2\pi i}{N}}$)

$$\begin{aligned} F^*_{N-m} &= \sum_{k=1}^{N} f_k (\omega^*)^{(k-1)(N-m-1)} \\ &= \sum_{k=1}^{N} f_k \omega^{-(k-1)(N-m-1)} \\ &= \sum_{k=1}^{N} f_k \omega^{(k-1)[-(N-m-1)+N]} \\ &= \sum_{k=1}^{N} f_k \omega^{(k-1)(m+1)} = F_{m+2}. \end{aligned}$$

6.1.1 Sampling rate

Let us now consider the problem of sampling a continuous time signal with a sampling rate F^s. F^s is the number of measurements per time unit, so we have the following relation between the time variable t and sampling rate F^s, $t = (k-1)/\text{rate} = (k-1)/F^s$, with $k = 1, \ldots, N$. Comparing formulas, the sampling rate is therefore given by $F_s = N/L$. Since e^{iqt} has frequency $\frac{q}{2\pi}$, and the coefficient F_m in $f = f(t)$ multiplies the term $e^{\frac{2\pi(m-1)}{L}it} = e^{\frac{2\pi(m-1)F^s}{N}it}$, F_m thus corresponds to the (complex) amplitude of the frequency $\frac{q}{2\pi} = (m-1)F^s/N$.

Hence in the time domain we plot `t=(0:N-1)/rate` versus `f(1:N)`, and in the frequency domain we plot `freq=(0:N/2)*rate/N` versus `abs(F(1:N/2+1))`, where `F=fft(f)`. Recall in this light the sunspot example.

6.1.2 Inverse FFT

The inverse FFT, `f=ifft(F)`, is given by

$$\begin{aligned}
\texttt{ifft(F)} &\equiv \frac{1}{N}\sum_{k=1}^{N} e^{\frac{2\pi i}{N}(k-1)(j-1)} F_k \\
&= \frac{1}{N}\sum_{k=1}^{N} e^{\frac{2\pi i}{N}(k-1)(j-1)} \sum_{m=1}^{N} e^{-\frac{2\pi i}{N}(k-1)(m-1)} f_m \\
&= \frac{1}{N}\sum_{m=1}^{N} f_m \sum_{k=1}^{N} \left(e^{\frac{2\pi i}{N}(j-m)}\right)^{k-1} \\
&= \frac{1}{N}\sum_{m=1}^{N} f_m N\delta_{jm} \\
&= f_j.
\end{aligned}$$

This may be checked directly using the identity

$$\sum_{k=1}^{N} \left(e^{\frac{2\pi i}{N}(j-m)}\right)^{k-1} = \frac{1-e^{\frac{2\pi i}{N}(j-m)N}}{1-e^{\frac{2\pi i}{N}(j-m)}} = 0$$

for $j-m \neq 0$ or more precisely $j-m \neq 0$ modulo N, and since $\sum_{k=1}^{N} x^{k-1} = \sum_{k=0}^{N-1} x^k = \frac{1-x^N}{1-x}$ for $x \neq 1$.

6.2 Speed of the FFT algorithm

To review the main sums so far, define $\omega = e^{-\frac{2\pi i}{N}}$. Then $\omega^N = 1$ and $\omega^* = \omega^{-1}$. The FFT and iFFT are now simply given by

- `F=fft(f)`: $F_m = \sum_{k=1}^{N} f_k \omega^{(k-1)(m-1)}$
- `f=ifft(F)`: $f_k = \frac{1}{N}\sum_{m=1}^{N} F_m \omega^{-(k-1)(m-1)}$

Note that since ω is known (and so are all its powers), these are linear equations! To find the FFT of f, all we need to do is to multiply f with a matrix. But this, it quickly becomes clear, is absolutely impossible to larger datasets. Matrix-vector multiplication takes $O(N^2)$ operations. For a long time, therefore, the FFT was not applicable in real-life situations. So let us now turn to the question of speed in computing the FFT.

Since $F_k = \sum_{j=1}^{N} \omega^{(k-1)(j-1)} f_j$, we need to compute $F = \Omega f$ with $\Omega_{kj} = \omega^{(k-1)(j-1)}$, i.e.

$$\boxed{\Omega = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots \\ 1 & \omega & \omega^2 & \omega^3 & \cdots \\ 1 & \omega^2 & \omega^4 & \omega^6 & \cdots \\ 1 & \omega^3 & \omega^6 & \omega^9 & \\ \vdots & \vdots & \vdots & & \ddots \end{pmatrix}.}$$

However, in the Cooley-Tukey FFT algorithm it only costs $O(N \log_2 N)$ operations. (for $N = 1024$ this is a factor 100 smaller). So let us see why this is!

To make our lives a little easier, let us assume N is 2^p for suitable p. Then $\omega_N = e^{-\frac{2\pi i}{N}}$, so that $\omega_N^2 = \omega_{N/2}$.

For $m \leq N/2$, with notation $f(k) = f_k$,

$$\begin{aligned}
F_f(m) &= \sum_{k=1}^{N} f(k)\omega_N^{(k-1)(m-1)} \\
&= \sum_{k \text{ odd}}^{N} f(k)\omega_N^{(k-1)(m-1)} + \sum_{k \text{ even}}^{N} f(k)\omega_N^{(k-1)(m-1)} \\
&= \sum_{l=1}^{N/2} f(2l-1)\omega_N^{(2l-2)(m-1)} + \sum_{l=1}^{N/2} f(2l)\omega_N^{(2l-1)(m-1)} \\
&= \sum_{l=1}^{N/2} f(2l-1)(\omega_N^2)^{(l-1)(m-1)} + \omega_N^{m-1} \sum_{l=1}^{N/2} f(2l)(\omega_N^2)^{(l-1)(m-1)} \\
&= \sum_{l=1}^{N/2} f(2l-1)\omega_{N/2}^{(l-1)(m-1)} + \omega_N^{m-1} \sum_{l=1}^{N/2} f(2l)\omega_{N/2}^{(l-1)(m-1)} \\
&= F_{f_{\text{odd}}}(m) + \omega_N^{m-1} F_{f_{\text{even}}}(m).
\end{aligned}$$

If one interprets $F_{f_{\text{odd/even}}}$ as the *twice repeated* vectors, then the formula $F_f(m) = F_{f_{\text{odd}}}(m) + \omega_N^{m-1} F_{f_{\text{even}}}(m)$ holds for all $m = 1, \ldots, N$ (i.e. also for $m > N/2$). After all, for $m = N/2 + m'$ it holds that

$$\omega_{N/2}^{(l-1)(m-1)} = \omega_{N/2}^{(l-1)(N/2+m'-1)} = \omega_{N/2}^{(l-1)(m'-1)},$$

since $\omega_{N/2}^{N/2} = 1$.

Moreover, $\omega_N^{N/2} = -1$, and we see that $\omega_N^{m-1} = \omega_N^{N/2+m'-1} = -\omega_N^{m'-1}$. Hence in symbolic (MATLAB-like) notation

$$\boxed{F_{f(1:N)} = \left[F_{f(1:2:N)} + \omega^{(0:\frac{N}{2}-1)} * F_{f(2:2:N)}; F_{f(1:2:N)} - \omega^{(0:\frac{N}{2}-1)} * F_{f(2:2:N)} \right].}$$

Therefore, if you already have $F_{f_{\text{odd}}}$ and $F_{f_{\text{even}}}$ then it only costs of the order of N operations ($3N/2$ if you ignore the cost of computing ω^k, which we will not go into) to calculate F_f!

This process you repeat for $F_{f_{\text{odd}}}$ and $F_{f_{\text{even}}}$, etcetera, until you reach the first level.

In other words, the FFT algorithm works recursively, constructing the solution from the two "half-solutions", which in turn are defined in terms of *their* two half-solutions, and so on. When does process end? When the vector, which is continuously halved, cannot be halved any further, we are left with one specific vector element F_k. But at that level, the equation for this specific vector element reads $F_k = f_k$, and nothing needs to be done at all.

To get an idea of the number of computations involved for a dataset of $N = 2^p$ points:

level	$0:$	2^p	FFTs of length 1	0 operations
level	$1:$	2^{p-1}	FFTs of length 2	$O(2^{p-1} \cdot 2)$ operations
level	$2:$	2^{p-2}	FFTs of length 4	$O(2^{p-2} \cdot 4)$ operations
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
level $p-1:$		2	FFTs of length 2^{p-1}	$O(2 \cdot 2^p)$ operations
level	$p:$	1	FFT of length 2^p	$O(2^p)$ operations

In total $O(p \cdot 2^p) = O(\log_2 N \cdot N) = O(N \log N)$ operations to calculate the Fourier of a vector of length N. Remember this!
This is the reason it is called "Fast".

Chapter 7

Differentiation and integration

When derivatives or integrals of functions are not available directly for computations, we need to to approximate them numerically. We will first consider numerical differentiation, and then look at numerical integration.

7.1 Finite differences

The basic idea of finite differences is to use Taylor expansions. We could of course start with

$$f(x+h) \approx f(x) + hf'(x) + \frac{1}{2}h^2 f''(\xi)$$

so that

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

giving the forward finite difference formula

$$\boxed{f'(x) = \frac{f(x+h) - f(x)}{h},}$$

which is of first order in h (the error is $\frac{h}{2}f''(\xi)$). We could approximate $f'(x)$ using $f(x-h)$ and $f(x)$ as well, with the same precision. But we can easily do a bit better. Expanding one extra term,

$$f(x+h) \approx f(x) + hf'(x) + \frac{1}{2}h^2 f''(x) + \frac{1}{6}h^3 f'''(\xi),$$
$$f(x-h) \approx f(x) - hf'(x) + \frac{1}{2}h^2 f''(x) - \frac{1}{6}h^3 f'''(\eta),$$

then subtraction leads to

$$f(x+h) - f(x-h) = 2hf'(x) + \frac{1}{6}h^3(f'''(\xi) + f'''(\eta)).$$

Hence

$$\boxed{f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} - \frac{1}{6}h^2(f'''(\xi) + f'''(\eta)).}$$

The approximation

$$\boxed{f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}}$$

is called *centered differences*, and is of *second* order in h.

At the boundary of an interval $[x_0, x_n]$, you do not have $x_0 - h$ at your disposal. Now you can use $x_0 + h$ and $x_0 + 2h$ rather than points left and right of x_0. Set $x_1 = x_0 + h$ and $x_2 = x_0 + 2h$. Then,

$$f(x_1) = f(x_0 + h) = f(x_0) + f'(x_0)h + \tfrac{1}{2}f''(x_0)h^2 + O(h^3),$$
$$f(x_2) = f(x_0 + 2h) = f(x_0) + 2f'(x_0)h + 2f''(x_0)h^2 + O(h^3).$$

To get rid of the order h^2 terms, we take $4f(x_1) - f(x_2)$, which leads to $-3f(x_0) + 4f(x_1) - f(x_2) = 2hf'(x_0) + O(h^3)$, so that

$$f'(x_0) \approx \frac{1}{2h}[-3f(x_0) + 4f(x_1) - f(x_2)]$$

is the approximation of order h^2.

What is the finite difference approximation for the second derivative? The idea is much the same. Using first $f'(x) \approx \frac{f(x+h)-f(x-h)}{2h}$, and taking its derivative, we find

$$f''(x) \approx \frac{f'(x+h) - f'(x-h)}{2h} \approx \frac{\frac{f(x+2h)-f(x)}{2h} - \frac{f(x)-f(x-2h)}{2h}}{2h} = \frac{f(x+2h) - 2f(x) + f(x-2h)}{4h^2}.$$

Replacing $2h$ by $\tilde{h}$, and dropping the tilde, we obtain

$$\boxed{f''(x) \approx \frac{f(x+h) - 2f(x) + f(x-h)}{h^2}.}$$

The error is of order h^2 as can be seen from, again, suitable Taylor series expansions (try it!). Third and higher order derivatives of course work much the same.

MATLAB also has numerical differentiation routines, such as `numjac`. Its syntax is a bit weird though, since it was actually programmed to be used by MATLAB engineers internally rather than by end users. It assumes the function to be differentiated (numerically) with respect to a variable x is a function of t and x! (This function is used in ODE solvers, and ODEs generally have the structure $\dot{u} = f(t, u)$.) The following example will make this clear.

```
help numjac
f=@(t,x) x^2
numjac(f,0,2,f(0,2),eps)
f=@(t,x) [x(1)^2+x(2)^2;x(1)*x(2)]
numjac(f,0,[2;3],f(0,[2;3]),eps)
```

7.2 Integration

We would like to numerically approximate $I = \int_a^b f(x)dx$, for some given function f defined on an interval $[a, b]$. For simplicity, we will divide $[a, b]$ into M equal subintervals I_k. Specifically, write $[a, b] = \cup_{k=1}^{M} I_k$, where $I_k = [x_{k-1}, x_k]$, $k = 1, \ldots, M$, and $x_k = a + kH$, $k0, \ldots, M$, with $H = (b - a)/M$.

7.2.1 The midpoint rule

The simplest construction would be a Riemann sum. We assume that f is constantly equal to c_k on each interval I_k, so that

$$I(f) \approx H \sum_{k=1}^{M} c_k.$$

It seems straightforward to choose $c_k = f(\bar{x}_k)$ where $\bar{x}_k = \frac{1}{2}(x_{k-1} + x_k)$ is the midpoint of I_k. This *composite midpoint rule* thus yields

$$\boxed{I_{mp}(f) = H \sum_{k=1}^{M} f(\bar{x}_k).}$$

To investigate the error, we need to compute $I(f) - I_{mp}(f)$. If $M = 1$, then there is only one midpoint, $\bar{x} = \frac{1}{2}(a+b)$. Then, using Taylor's theorem, we get

$$\begin{aligned} I(f) - I_{mp}(f) &= \int_a^b [f(x) - f(\bar{x})]dx \\ &= \int_a^b f'(\bar{x})(x-\bar{x})dx + \frac{1}{2}\int_a^b f''(\eta(x))(x-\bar{x})^2 dx. \end{aligned}$$

But note that

$$\int_a^b f'(\bar{x})(x-\bar{x})dx = 0,$$

since it is equal to

$$f'(\bar{x})\int_a^b (x - \frac{1}{2}(a+b))dx = 0.$$

The Second Mean Value Theorem[1] states (roughly) that there exists a $\xi \in [a,b]$ such that

$$\int_a^b f(x)g(x)dx = f(\xi)\int_a^b g(x)dx.$$

If we apply this to the last term in the error, we find a $\xi \in [a,b]$ such that

$$\begin{aligned} \frac{1}{2}\int_a^b f''(\eta(x))(x-\bar{x})^2 dx &= \frac{1}{2}f''(\xi)\int_a^b (x-\bar{x})^2 dx \\ &= \frac{(b-a)^3}{24}f''(\xi). \end{aligned}$$

Generalising this computation to $M > 1$, so that $H < b - a$, we find

$$I(f) - I_{mp}(f) = \frac{b-a}{24}H^2 f''(\xi)$$

The conclusion is that the composite midpoint rule is of second order in the discretisation step H.

7.2.2 The trapezoidal rule

The trapezoidal rule uses piecewise-linear approximations of the function you want to integrate. The area of the trapezium spanned by the points $(x_{k-1}, 0)$, $(x_k, 0)$, $(x_k, f(x_k))$ and $(x_{k-1}, f(x_{x-1}))$ is given by $\frac{1}{2}(f(x_{k-1}) + f(x_k))(x_k - x_{k-1})$. Hence the trapezoidal approximation $I_t(f)$ is given by

$$\boxed{I_t(f) = \frac{H}{2}\sum_{k=1}^{M}(f(x_{k-1}) + f(x_k)).}$$

Since all points appear twice, except the end points $x = a$ and $x = b$, a lot of the terms in the above sum cancel, and the formula reduces to

$$I_t(f) = \frac{H}{2}(f(a) + f(b)) + H\sum_{k=1}^{M-1} f(x_k).$$

Without going to the proof, the error in this approximation is

$$I(f) - I_t(f) = -\frac{b-a}{12}H^3 f''(\xi).$$

In other words, it is one order better in H than the midpoint rule. In MATLAB, the trapezoidal rule is implemented using `trapz` and `cumtrapz` (the latter gives a cumulative array, so the n-th entry in the array is the sum $\frac{H}{2}f(a) + H\sum_{k=1}^{n} f(x_k)$.

[1]The first Mean Value Theorem states $\int_a^b f(x)dx = f(\xi)(b-a)$, so just use $g(x) = 1$ in the Second MVT.

7.2.3 Simpson's quadrature

One step better again is to use second order polynomials through the points. The rule becomes

$$\boxed{I_S(f) = \frac{H}{6} \sum_{k=1}^{M} \left(f(x_{k-1}) + 4f(\bar{x}_k) + f(x_k) \right),}$$

where $\bar{x}_k = \frac{1}{2}(x_{k-1} + x_k)$, the midpoint between x_{k-1} and x_k. The error is

$$I(f) - I_S(f) = -\frac{b-a}{180} \frac{H^4}{16} f''''(\xi).$$

So again, a step up in order of convergence, but also a step up in smoothness: the function you want to integrate has to be C^4.

7.2.4 Final remarks

Here's a smart, and very general, trick: take half the discretisation stepsize h, calculate again, and use order of convergence. Let $I_h(f)$ be a numerical approximation of the integral $I(f)$, with stepsize h, of order p in h. Then

$$I(f) - I_h(f) \approx Ch^p,$$
$$I(f) - I_{h/2}(f) \approx C(\tfrac{h}{2})^p.$$

Let $\Delta I = I_{h/2} - I_h$. Then

$$C \approx \frac{I_{h/2} - I_h}{h^p - (\frac{h}{2})^p} = \Delta I \frac{2^p}{(2^p - 1)h^p}$$

and the error is approximately

$$I(f) - I_{h/2}(f) \approx \frac{1}{2^p - 1} \Delta I.$$

This gives us a way to estimate the error of a numerical approximation of the integral by combining two different numerical approximations, without knowing the true value of the integral. This opens the foor for adaptive methods.

In addition, a little calculation shows that

$$I(f) - \frac{2^p}{2^p - 1} I_{h/2}(f) - \frac{1}{2^p - 1} I_h(f) = \mathcal{O}(h^{p+1}),$$

and a better approximation of the integral is now

$$I(f) \approx \frac{2^p}{2^p - 1} I_{h/2}(f) + \frac{1}{2^p - 1} I_h(f).$$

Chapter 8

Iterative methods for linear systems

Large linear systems occur frequently in applications. One way to solve them is through elementary row operations (Gauss elimination). or LU decomposition. For small matrices this is feasible, preferably combined with "pivoting" (intelligent choice of pivots) to improve numerical stability (battling rounding errors). When the number of equations and variables are in thousands or more, it is not realistic to compute solutions using elementary row operations. For such systems, iterative methods, in which an approximation to the solutions is computed in each step, are more convenient. Elementary row operations also do not work particularly well with badly conditioned (nearly singular, for instance) matrices. Also in such cases, iterative methods are commonly used.

We will start with reviewing some simple methods, and at the end treat two of the more modern methods.

8.1 Preconditioning

Let us start with an invertible matrix A, and consider $Ax = b$. We set up a linear iteration scheme, with, for now, arbitrary B and g:

$$x_{k+1} = Bx_k + g.$$

Fixed points then satisfy $x = Bx + g$. Moreover, we want the fixed point to solve our linear system, i.e. $x = A^{-1}b$, so we should take $g = (I - B)A^{-1}b$. Let $e_k = x - x_k$ be the error in the k-th step. Then $e_{k+1} = Be_k$, and the method converges iff $\|B\| < 1$. For symmetric B, this means that $|\lambda| < 1$ for all eigenvalues of B. Clearly, convergence is faster if $\|B\|$ is smaller.

Let us write things a little different, by introducing a new matrix P which we can later choose appropriately to suit our needs. Consider $A = P + (A - P)$, so that the problem now reads

$$[P + (A - P)]x = b,$$

which is equivalent to

$$x + P^{-1}(A - P)x = P^{-1}b,$$

and indeed to

$$x = P^{-1}(P - A)x + P^{-1}b.$$

This suggests that we should set an iteration scheme by setting

$$B = P^{-1}(P - A), \text{ and } g = P^{-1}b.$$

The choice $P = A$ is trivial, and of little use. To be useful in applications, P should be easy to invert. We can write the iteration scheme as

$$x_{k+1} = P^{-1}(P - A)x_k + P^{-1}b$$

or

$$P(x_{k+1} - x_k) = b - Ax_k = r_k,$$

where $r_k = b - Ax_k$ is the residue of the k^{th} iteration.

You can put another factor α in front of the residue: $P(x_{k+1} - x_k) = \alpha_k r_k$, which gives us another parameter to play with. This will be useful in Richardson's method later on.

The last equation may also be written in this form,

$$x_{k+1} = x_k + (P^{-1}b - P^{-1}Ax_k)$$

One way to interpret this is that instead of solving $Ax = b$ we are actually solving $P^{-1}Ax = P^{-1}b$. From this stems the term *preconditioning*. P is the preconditioning matrix.

Let us consider a few examples.

- The *Jacobi method* is found by taking $P = \text{diag}(A)$. It works for example if the matrix is diagonally dominant by row.
- The *Gauss-Seidel* (GS) method uses the lower triangular part of A as matrix P. GS works for symmetric positive-definite matrices and converges faster than Jacobi.
- $Ax = b$ could be solved using $A^TAx = A^Tb$, where A^TA is symmetric and positive definite (but with a worse conditioning number), hence one may apply GS. This should also remind you of least squares (on which more in Chapter 5).

8.2 When to stop iterating

There are several possible choices to terminate the iterative scheme.

Possibility 1: stop when the residue is small. Recall that $r_k = b - Ax_k$, or in other words x_k solves $Ax = b - r_k$. Hence (with the definition of the relative condition number $K(A)$ in §A.2),

$$\frac{|x_k - x|}{|x|} \leq K(A)\frac{|r_k|}{|b|}.$$

Hence we should stop at small residue if A is well conditioned.

Possibility 2: stop when the distance between consecutive iterations, $d_k = x_{k+1} - x_k$ is small enough. We should check that the error, $e_k = x - x_k$ becomes sufficiently small. Now note that

$$\begin{aligned} |e_k| = |x - x_k| &= |x - x_{k+1} + (x_{k+1} - x_k)| \\ &= |e_{k+1} + d_k| \\ &\leq |e_{k+1}| + |d_k| \\ &\leq \|B\||e_k| + |d_k|, \end{aligned}$$

so that

$$|e_k| \leq \frac{1}{1 - \|B\|}|d_k|.$$

We conclude that this second criterion works particularly well when $\|B\| \ll 1$.

8.3 Richardson's method and preconditioned gradient method

This method is for matrices A and P that are positive definite and symmetric, and such that $P^{-1}A$ is also symmetric (and automatically positive definite). We now use the preconditioning, as well as the extra parameter α:

$$\begin{aligned} x_{k+1} &= x_k + P^{-1}\alpha_k r_k \\ &= x_k + \alpha_k P^{-1}(b - Ax_k) \\ &= (I - \alpha_k P^{-1}A)x_k + \alpha_k P^{-1}b. \end{aligned}$$

In Richardson's method we choose $\alpha_k = \alpha$, so independent of k. If $\lambda_i > 0$ are the eigenvalues of $P^{-1}A$, then $1 - \alpha\lambda_i$ are the eigenvalues of $I - P^{-1}A$. Hence for convergence it is necessary that $|1 - \alpha\lambda_i| < 1$. It is therefore necessary to at least require $|1 - \alpha\lambda_{\max}| < 1$, which we may also write as $0 < \alpha < 2/\lambda_{\max}$. The requirements $|1 - \alpha\lambda_i| < 1$ may be viewed graphically, as linear functions of α with slopes determined by λ_i, see Figure 8.1. We wish to make choose α so that $|1 - \alpha\lambda_i| < 1$ is minimal for all λ_i. The picture shows that

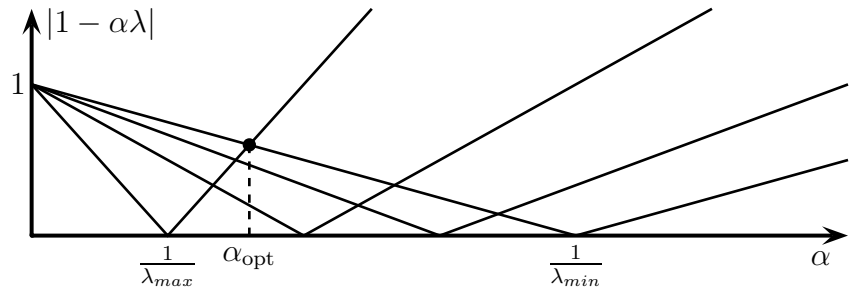

FIGURE 8.1. The constraints $|1-\alpha\lambda_i|<1$ illustrated as functions of α. The optimal α so that $|1-\alpha\lambda_i|$ are *all* as small as possible lies on the intersection of the lines $1-\alpha\lambda_{\min}$ and $\alpha\lambda_{\max}-1$.

the *optimal* α is found when $1-\alpha\lambda_{\min}=\alpha\lambda_{\max}-1$. Therefore, $\alpha_{\text{opt}}=2/(\lambda_{\min}+\lambda_{\max})$ follows immediately, and the rate of convergence is

$$\alpha_{\text{opt}}\lambda_{\max}-1=\frac{\lambda_{\max}-\lambda_{\min}}{\lambda_{\max}+\lambda_{\min}}=\frac{\frac{\lambda_{\max}}{\lambda_{\min}}-1}{\frac{\lambda_{\max}}{\lambda_{\min}}+1}=\frac{K(P^{-1}A)-1}{K(P^{-1}A)+1}.$$

Remarks

- For *every* preconditioner the method converges if α is small enough.
- The method converges faster if $K(P^{-1}A)$ is better conditioned (closer to the identity for example). That is why P is the preconditioner!
- P is not so easy to choose in general, but $P=\text{diag}(A)$ and $P=I$ are at least easy choices.

8.4 The gradient method

This next method is restricted to A symmetric positive-definite. Such matrices appear quite often in applications. You have seen such matrices in your Linear Algebra course, when studying quadratic forms. Recall that a quadratic form $Q(x)$ is a function of $x=(x_1,\dots,x_n)$ which is homogeneous of degree 2. In other words, it only contains terms of the form x_ix_j and x_i^2. A quadratic form $Q(x)$ may be written as $Q(x)=x^TAx$ for a suitable symmetric matrix A. The form Q is positive definite if $Q(x)>0$ for all $x\neq 0$.

For such matrices, we can associate a so-called energy function

$$\Phi(y)=\frac{1}{2}y^TAy-y^Tb.$$

Minima of Φ are solutions of $Ax=b$. To see this, we compute the gradient of Φ. First note that

$$\begin{aligned}\Phi(y+z)&=\frac{1}{2}(y+z)^TA(y+z)-(y+z)^Tb\\&=\frac{1}{2}y^TAy+\frac{1}{2}z^TAy+\frac{1}{2}y^TAz+\frac{1}{2}z^TAz-y^Tb-z^Tb.\end{aligned}$$

Using that $y^TAz=(y^TAz)^T=z^TA^Ty$ (this is a real number, after all) it follows that

$$\nabla\Phi(y)=\frac{1}{2}(A^T+A)y-b=Ay-b.$$

so that $\nabla\Phi(x)=0$ is equivalent to $Ax=b$ for symmetric A. For any other $y\neq x$,

$$\Phi(y)=\Phi(x+(y-x))=\Phi(x)+\frac{1}{2}(y-x)^TA(y-x)>\Phi(x).$$

So the goal is to take a starting y, and descend down the energy function until the minimum is reached.

Note that you can interpret $\Phi(y)-\Phi(x)=\frac{1}{2}|y-x|_A^2$ as the so-called A-norm (and the A-inner product: x^TAy, which you might also recall from Linear Algebra 1).

To get to the bottom of the well, we descend following the gradient at the current point, as the gradient vector is the direction in which Φ increases most. First note that

$$\nabla\Phi(x_k) = Ax_k - b = -r_k.$$

The idea is now to take a step from x_k to $x_k + \alpha r_k$. Φ will start to decrease first, but it will later increase again. The optimal α therefore satisfies $\frac{\partial\Phi}{\partial\alpha}(x_k + \alpha p) = 0$. By tuning α in this way, we make the biggest step forward to decrease Φ maximally in this one step, and then repeat the process.

Choose first a general direction p, and make a step in the direction $x_k + \alpha p$. We have

$$\Phi(x_k + \alpha p) = \frac{1}{2}(x_k + \alpha p)^T A(x_k + \alpha p) - (x + \alpha p)^T b.$$

Now

$$\begin{aligned} 0 &= \frac{\partial\Phi}{\partial\alpha}(x_k + \alpha p) \\ &= \frac{1}{2}p^T A(x_k + \alpha p) + \frac{1}{2}(x_k + \alpha p)^T Ap - p^T b \\ &= p^T Ax_k - p^T b + \alpha p^T Ap \end{aligned}$$

which means that the optimal α satisfies

$$\alpha = \frac{p^T(b - Ax_k)}{p^T Ap} = \frac{p^T r_k}{p^T Ap}.$$

If we now choose the gradient direction $p = r_k$ we obtain $\alpha_k = \frac{r_k^T r_k}{r_k^T Ar_k}$. You might also do some additional preconditioning with P and choose direction $P^{-1}r_k$. This leads to the algorithm

$$\boxed{\begin{aligned} r_k &= b - Ax_k \\ Py_k &= r_k \\ \alpha_k &= \frac{(y_k)^T r_k}{(y_k)^T Ay_k} \\ x_{k+1} &= x_k + \alpha_k y_k \\ &\text{repeat.} \end{aligned}}$$

A small example is given below. The script `gradillu` illustrates the gradient method, together with the next method, the conjugate gradient method, shown in Figure 8.2.

```
A=[3,2;2,6]
b=[1;2]
x0=[-2;3]
gradillu(A,b,x0)
```

8.5 Conjugate gradient method

The gradient method may be improved by choosing the direction a little better (so that the new directions are perpendicular in the right inner product to all previous directions).

Suppose that x_k is optimal with respect to a direction p. In other words,

$$\Phi(x_k) \leq \Phi(x_k + \lambda p) \quad \text{for all } \lambda \in \mathbb{R}.$$

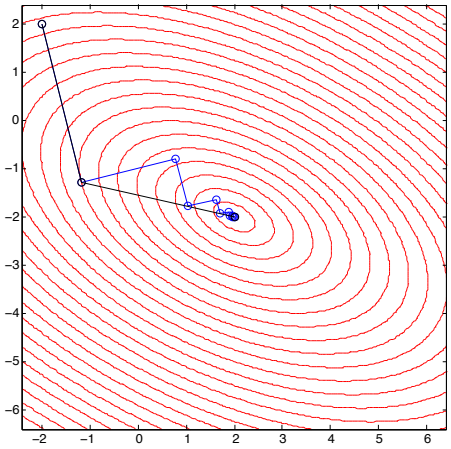

FIGURE 8.2. Illustration of the gradient (blue line) and conjugate gradient (grey line) methods. Note that the latter method converges to the solution in two steps, whereas the first in many. The red lines show contour lines of the function Φ associated with this example matrix A.

We observe that x_k is optimal with respect to a search direction p if and only if p is perpendicular to r_k. To see this, note that

$$\begin{aligned}\frac{\partial \Phi}{\partial \lambda}(x_k + \lambda p) &= p^T(Ax_k - b) + \lambda p^T A p \\ &= p^T r_k + \lambda p^T A p.\end{aligned}$$

Say now that, after the next step om our algorithm, $x_{k+1} = x_k + q$ (with q not necessarily in the direction r_k). We want that x_{k+1} remains optimal with respect to the previous direction p. That means you want that $0 = p^T r_{k+1}$. On the other hand

$$\begin{aligned}p^T r_{k+1} &= p^T(b - Ax_{k+1}) \\ &= p^T(b - (Ax_k + Aq)) \\ &= p^T(r^k - Aq) \\ &= -p^T Aq.\end{aligned}$$

Hence choose direction q which is A-orthogonal to p (with respect to the A-inner product $(x, y)_A = (x, Ay)$). Ideally you would like that it is A-orthogonal to all previous directions, but let's take $p = p_{k-1}$ for now. We will see that the orthogonality with respect to all previous directions will come for free (in the algorithm below).

The new search direction is $p_k = r_k - \beta_k p_{k-1}$, hence this search direction is a "correction" of the gradient r_k, with

$$\beta_k = \frac{(Ap_{k-1})^T r_k}{(Ap_{k-1})^T p_{k-1}}.$$

The initial direction is taken to be $p_0 = r_0$.

Next, compute $x_{k+1} = x_k + \alpha_k p_k$ with $\alpha_k = \frac{p_k^T r_k}{p_k^T A p_k}$ and thus also $r_{k+1} = r_k - \alpha_k A p_k$. The order in which the different numbers and vectors are computed is x_k, r_k, β_k, p_k, α_k, x_{k+1}, etcetera.

From the above it follows that if x_k was optimal for an arbitrary direction p, then x_{k+1} is optimal for direction p iff p_k is A-perpendicular to p.

We now claim that p_k is A-perpendicular to all previous directions $\{p_j\}_{j=0}^{k-1}$, not just p_{k-1}. Therefore, x_{k+1} is optimal with respect to all previous directions$\{p_j\}_{j=0}^{k}$. In other words $p_j^T r_{k+1} = 0$ for $j = 0, \ldots, k$. The proof is outlined below, and is by induction.

PROOF. We need to show that $(Ap_j)^T p_k = 0$ for $j = 0, \ldots, k-1$.

$(Ap_{k-1})^T p_k = 0$ by construction (choice β_k). In particular the assertion is true for $k = 1$. We still need to prove: $(Ap_j)^T p_k = 0$ for $j = 0, \ldots k-2$.

The induction hypothesis states that $\{p_j\}_{j=0}^{k-1}$ are all A-perpendicular, and hence $p_i^T r_j = 0$ for $i = 0, \ldots, j-1$ and $j = 1, \ldots, k$.

Because $p_0 = r_0$ it follows from $p_{i+1} = r_{i+1} - \beta_{i+1} p_i$ that

$$\mathrm{sp}(r_0, \ldots, r_j) = \mathrm{sp}(p_0, \ldots, p_j) \qquad \text{for } j = 0, \ldots, k.$$

For every $j = 0, \ldots, k-1$, since $Ap_j = \frac{1}{\alpha_j}(r_j - r_{j+1})$, it follows that

$$Ap_j \in \mathrm{sp}(r_0, \ldots, r_{j+1}) = \mathrm{sp}(p_0, \ldots, p_{j+1}).$$

From the induction hypothesis it now follows that

$$\begin{aligned}(Ap_j)^T p_k &= (Ap_j)^T r_k + (Ap_j)^T p_{k-1} \\ &= (Ap_j)^T r_k \qquad \text{for } j = 0, \ldots, k-2.\end{aligned}$$

Since $p_{j'}^T r_k = 0$ for $j' = 0, \ldots, k-1$, and $Ap_j \in \mathrm{sp}(p_0, \ldots, p_{j+1})$, we see that $(Ap_j)^T r_k = 0$ for $j = 0, \ldots, k-2$ (notice: $k-2$, not $k-1$, since $j+1 = j'$).

Taking the last points together we see that indeed $(Ap_j)^T p_k = 0$ for $j = 0, \ldots, k-2$. □

In summary, the CGM is described by the following algorithm, with initialisation $\beta_0 = 0$ (ignoring the initial calculation that involves the non-existent p_{-1}):

$$\boxed{\begin{aligned} r_k &= b - Ax_k \\ \beta_k &= \tfrac{(Ap_{k-1})^T r_k}{(Ap_{k-1})^T p_{k-1}} \\ p_k &= r_k - \beta_k p_{k-1} \\ \alpha_k &= \frac{(p_k)^T r_k}{(p_k)^T A p_k} \\ x_{k+1} &= x_k + \alpha_k p_k \\ &\text{repeat.}\end{aligned}}$$

This algorithm may be rearranged somewhat to obtain a computationally cheaper and more stable one. Of course you can do this with a preconditioner as well, giving us the *Preconditioned Conjugate Gradient Method*.

Since the number of A-orthogonal directions one can possibly choose is equal to dimension of the vector space, the Conjugate Gradient Method must terminate in a finite number of steps. That is nice, but the most important point is that it converges quickly.

Chapter 9

Eigenvalues

In this chapter we are going to have a look at how to compute eigenvalues of matrices. Given the central place of solving linear systems in numerical applications, and the importance of eigenvalues in linear algebra, this problem comes up over and over again.

We can generally ask two different questions: how do we estimate all the eigenvalues of a matrix, or how do we estimate a *particular* eigenvalue of interest (e.g., a zero eigenvalue, or an eigenvalue with norm 1, or the very largest or smallest eigenvalue among all eigenvalues). We will start with the last question, before turning to finding all eigenvalues of a matrix. Note that the commands in MATLAB to compute eigenvalues are `eig` and `eigs` (read the documentation!).

9.1 Single eigenvalues: the Power method

Let us introduce the so-called *Power method*. Let A be a given matrix with eigenvalues $|\lambda_1| > |\lambda_2| > \cdots$. For a given starting point y_0, make a sequence $\{(x_k, y_k, \mu_k)\}$, $k = 1, 2, \ldots$, by setting

$$\boxed{\begin{aligned} x_k &= Ay_{k-1}, \\ y_k &= \frac{x_k}{||x_k||}, \\ \mu_k &= \bar{y}_k^T A y_k = y_k^* A y_k. \end{aligned}}$$

Then μ_k converges to the maximal eigenvalue λ_1 at a rate $|\lambda_2/\lambda_1|$. The Power method just entails successively multiplying a vector by A and renormalising, to ensure the new vector has a reasonable norm. The behaviour of the vector x_k, as k increases, is therefore very reminiscent of a simple discrete time Markov chain, where x_k approaches a stationary distribution. An idea of the proof of the Power method, for diagonalisable matrices, may be found at the end of this section.

Of course, the Power method only works if there is a dominant eigenvalue ($|\lambda_1| > |\lambda_i|$).

The Power method may be modified slightly, to home in on the minimal eigenvalue rather than the maximal one. This is called the *inverse Power method*, and is just the Power method using the inverse of A:

$$\boxed{\begin{aligned} x_k &= A^{-1}y_{k-1}, \\ y_k &= \frac{x_k}{||x_k||}, \\ \mu_k &= \bar{y}_k^T A^{-1} y_k = y_k^* A^{-1} y_k. \end{aligned}}$$

(Evidently, A should be invertible. What if A is not invertible? How would you find the minimal eigenvalue (in norm) of a noninvertible matrix...?)

The Power method seems a bit limited in scope, since it only seems to allow us to find maximal or minimal eigenvalues. But note that the matrix $A - \lambda I$ has eigenvalues that are shifted exactly by λ. (Do the computation to convince yourself of this fact, and also that the eigenvectors do not change!) So if we would

like to find the eigenvalue of A closest to $\sqrt{2}$, say, then we can apply the inverse Power method to $A - \sqrt{2}I$. This is called the *inverse Power method with shift*.

To apply the inverse Power method with shift, it is clearly necessary to have a general idea of where the eigenvalues might be located. The following theorem on *Gershgorin disks* gives a useful indication. We make the following n *row disks* C_i^r and n *column disks* C_i^c for A,

$$C_i^r = \left\{ z \in \mathbb{C} \mid |z - a_{ii}| \leq \sum_{j=1, j\neq i}^{n} |a_{ij}| \right\}$$
$$C_i^c = \left\{ z \in \mathbb{C} \mid |z - a_{ii}| \leq \sum_{j=1, j\neq i}^{n} |a_{ji}| \right\}.$$

In other words, the circles are centered at diagonal elements of A and you take the sum of the absolute values of the remaining row (or column) entries of the matrix to define a radius of the circle for that row (or column). We now take the union of the row disks, and call it U^r and the union of the column disks, U^c. Now we have the following theorem.

THEOREM 9.1.1. *All the eigenvalues of A are in the region of the complex plane that is the formed by intersection of the row and column disk unions $U^r \cap U^c$.*

PROOF. Write $A = D + E$, where D is a diagonal matrix consisting of the diagonal of A. Then the eigenvalue equation is $[(D - \lambda I) + E]x = 0$. Choose k such that $|x_k|$ is maximal, then

$$(a_{kk} - \lambda)x_k = -\sum_{j\neq k} a_{kj} x_j,$$

hence

$$|a_{kk} - \lambda| = \left| \sum_{j\neq k} a_{kj} \frac{x_j}{x_k} \right| \leq \sum_{j\neq k} |a_{kj}|.$$

This shows that λ is in C_k^r. To get the same statement for the column disks, just observe that eigenvalues of A and A^T coincide. □

A stronger version of the theorem states that if there are m row (or column) disks whose union is disjoint of the remaining $n - m$ row (or column) disks, then this first union of m row (or column) disks contains exactly m eigenvalues.

Figure 9.1 shows an example of the Gershgorin row and column disks for the matrix

$$A = \begin{bmatrix} -3 & 0 & 2 & 2 \\ 2 & 1 & 1 & 1 \\ 0 & 1 & 5 & 0 \\ 1 & 1 & 2 & 11 \end{bmatrix}. \tag{9.1.1}$$

To give an idea why the Power method works at all, let us focus on self-adjoint matrices. If A is self-adjoint it can be diagonalised: $A = V\Lambda V^{-1}$, with Λ diagonal (the real eigenvalues) and $V = (v^1, \ldots, v^n)$ a unitary matrix with the orthonormal eigenvectors (i.e. $V^* = V^{-1}$). From the decomposition we see that

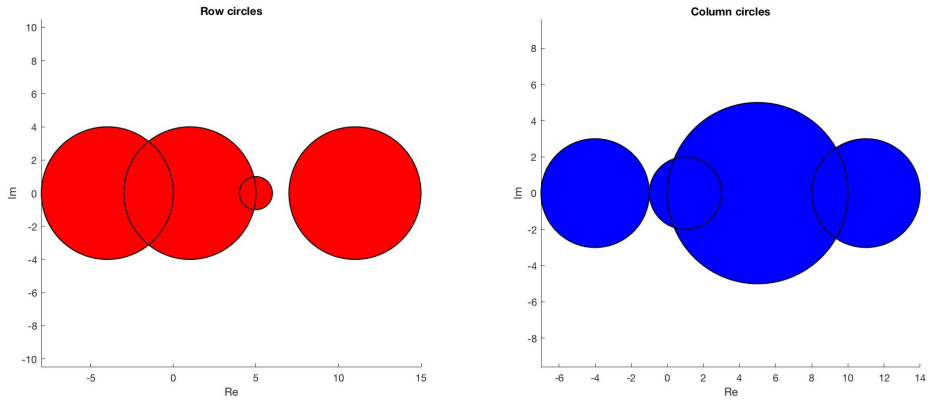

FIGURE 9.1. Example Gershgorin row (left) and column (right) disks for the matrix A given in 9.1.1.

$A^k = V\Lambda^k V^*$, so that $A^k x = \sum_{i=1}^n \lambda_i^k (v_i, x) v_i$. Hence, provided $(v_1, x) \neq 0$

$$\begin{aligned}
\lim_{k\to\infty} \frac{A^k x}{|A^k x|} &= \lim_{k\to\infty} \frac{\sum_{i=1}^k \lambda_i^k (v_i, x) v_i}{|\sum_{i=1}^k \lambda_i^k (v_i, x) v_i|} \\
&= \lim_{k\to\infty} \frac{\sum \lambda_i^k (v_i, x) v_i}{\sqrt{\sum \lambda_i^{2k} (v_i, x)^2}} \\
&= \lim_{k\to\infty} \frac{\sum (\frac{\lambda_i}{|\lambda_1|})^k (v_i, x) v_i}{\sqrt{\sum (\frac{\lambda_i}{\lambda_1})^{2k} (v_i, x)^2}} \\
&= \frac{\pm (v_1, x) v_1}{|(v_1, x)|} \\
&= \pm v_1.
\end{aligned}$$

To converge to v_1, we needed to require that $(v_1, x) \neq 0$ (x should have a v_1-component). So the Power method does not always work, but for a generic choice of starting vector x it does.

Computing also the next term (taking the + sign above, hence $\lambda_1 > 0$) gives us the rate of convergence:

$$\begin{aligned}
|A^k x| &= \sqrt{\lambda_1^{2k} (v_1, x)^{2k} + \lambda_2^{2k} (v_2, x)^{2k} + \dots} \\
&= \lambda_1^k |(v_1, x)| \left[1 + \frac{1}{2} \left(\frac{\lambda_2}{\lambda_1}\right)^{2k} \frac{(v_2, x)^2}{(v_1, x)^2} + \dots \right] \\
\frac{A^k x}{|A^k x|} &= \frac{\lambda_1^k (v_1, x) v_1 + \lambda_2^k (v_2, x) v_2}{\lambda_1^k |(v_1, x)| \left[1 + \frac{1}{2} \left(\frac{\lambda_2}{\lambda_1}\right)^{2k} \frac{(v_2, x)^2}{(v_1, x)^2} + \dots \right]} \\
&= v_1 + \left(\frac{\lambda_2}{\lambda_1}\right)^k \frac{(v_2, x)}{|(v_1, x)|} v_2 + \dots
\end{aligned}$$

This in fact also suggests a way to get our hands on the second eigenvector (and eigenvalue). Consider two starting vectors x_1 and x_2 (not linearly dependent). Let

$$y_1^k = \frac{A^k x_1}{|A^k x_1|},$$

be the vectors we obtain from applying the power method to x_1. Now apply a variant of the power method to x_2, where we orthogonalise with respect to y_1^k:

$$\begin{aligned}
\xi_2^k &= A^k x_2 - (y_1^k, A^k x_2) y_1^k \\
&= \lambda_1^k (v_1, x_2) v_1 + \lambda_2^k (v_2, x_2) v_2 + \dots \\
&\quad - \left\langle v_1 + \left(\frac{\lambda_2}{\lambda_1}\right)^k \frac{(v_2, x_1)}{|(v_1, x_1)|} v_2, \lambda_1^k (v_1, x_2) v_1 + \lambda_2^k (v_1, x_2) v_2 \right\rangle \left[v_1 + \left(\frac{\lambda_2}{\lambda_1}\right)^k \frac{(v_2, x_1)}{|(v_1, x_1)|} v_2 \right] \\
&= \lambda_2^k (v_2, x_2) v_2 - \lambda_2^k (v_1, x_2) \frac{(v_2, x_1)}{|(v_1, x_1)|} v_2 + \dots \\
&= \lambda_2^k \left[(v_2, x_2) - \frac{(v_1, x_2)(v_2, x_1)}{|(v_1, x_1)|} \right] v_2 + \dots .
\end{aligned}$$

Then $y_2^k = \frac{\xi_2^k}{|\xi_2^k|}$ generically tends to $\pm v_2$ as $k \to \infty$. Numerically, it is better to orthonormalise y_1^k and y_2^k after every step (multiplication by A). This is what happens in the QR algorithm which is discussed in the next section.

9.2 QR algorithm (all eigenvalues)

To compute all the eigenvalues of a matrix, the QR algorithm is used most often. It is in a sense a variation on the Power method.

First note the following: $B = UAU^*$ has the same eigenvalues as A for any unitary U. If $Av = \lambda v$, then $BUv = \lambda Uv$, hence the eigenvectors of A and B are also easily related.

The QR *algorithm* is based on the QR *decomposition*. The QR decomposition of A is as follows. Let A be an $(n \times n)$ matrix. Then we write $A = QR$, with Q Orthonormal (unitary) R Upper/Right triangular matrix. If A real, then Q and R real.

To calculate the QR decomposition, you may use Gram-Schmidt orthogonalisation

$$\begin{aligned}
\tilde{q}_1 &= a_1 \\
q_1 &= \tilde{q}_1 / |\tilde{q}_1| \\
\tilde{q}_2 &= a_2 - \langle a_2, q_1 \rangle \\
q_2 &= \tilde{q}_2 / |\tilde{q}_2| \\
\tilde{q}_3 &= a_3 - \langle a_3, q_1 \rangle - \langle a_3, q_2 \rangle \\
q_3 &= \tilde{q}_3 / |\tilde{q}_3|
\end{aligned}$$

with $r_{ik} = \langle a_k, q_i \rangle$ for $i = 1, \dots k-1$ and $r_{kk} = |\tilde{q}_k|$, since $R = Q^* A$. There are other (better) algorithms to determine the QR decomposition, and other applications, but we will not discuss those here. The QR algorithm is now defined as follows. Define a sequence of matrices by initialising $T_1 = A$ and then recursively:

$T_k = Q_k R_k$	(decomposition)
$T_{k+1} = R_k Q_k$	(definition)
repeat.	

Then

$$\begin{aligned}
T_{k+1} &= Q_k^* T_k Q_k \\
&= Q_k^{-1} T_k Q_k \\
&= (Q_1 Q_2 \cdots Q_k)^{-1} A (Q_1 Q_2 \cdots Q_k),
\end{aligned}$$

hence the eigenvalues of T_k and A are the same (the product of the Q_i is unitary). Let A be an $(n \times n)$ matrix with eigenvalues λ_i, such that $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$. Then

$$\lim_{k\to\infty} T_k = \begin{pmatrix} \lambda_1 & t_{12} & t_{12} & \dots & t_{1n} \\ 0 & \lambda_2 & t_{23} & \dots & t_{2n} \\ 0 & 0 & \lambda_2 & \dots & t_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \lambda_n \end{pmatrix}.$$

The eigenvalues are not necessarily in the right order, but in practice they are (for generic A, as well as numerically).

If A is self-adjoint and $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$, then $\lim_{k\to\infty} T_k$ becomes a diagonal matrix in the limit (and convergence is quadratic):

$$\lim_{k\to\infty} T_k = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \lambda_n \end{pmatrix}$$

Try the following example in MATLAB.

```
A=[1,2,3;4,5,6;7,8,10]
[Q,R]=qr(A); A=R*Q
[Q,R]=qr(A); A=R*Q
[Q,R]=qr(A); A=R*Q
[Q,R]=qr(A); A=R*Q
[Q,R]=qr(A); A=R*Q
```

gives

$$A = \begin{pmatrix} 16.7075 & -4.7606 & 1.1520 \\ -0.0000 & -0.9058 & 0.0993 \\ -0.0000 & -0.0006 & 0.1983 \end{pmatrix}.$$

The eigenvalues of A (computed numerically using `eig(A)`) are $16.7075, -0.9057, 0.1982$. For a self-adjoint example, (e.g., take $B = A + A^T$ for the original A above), the resulting sequences of matrices becomes diagonal. With the above choice of $B = A + A^T$, the QR algorithm yields

$$\begin{pmatrix} 34.0848 & -0.0002 & -0.0000 \\ -0.0002 & -2.4656 & -0.0002 \\ -0.0000 & -0.0002 & 0.3808 \end{pmatrix}.$$

The diagonal elements are numerically identical to the eigenvalues up to at least four decimal places.

The algorithm may take a long time to converge in some cases. E.g., for the matrix $A = \begin{pmatrix} 1 & \varepsilon \\ \varepsilon & 2 \end{pmatrix}$, the QR algorithm has not converged after 50 iterations.

```
A=[1,eps;eps,2]
[Q,R]=qr(A); A=R*Q
[Q,R]=qr(A); A=R*Q
[Q,R]=qr(A); A=R*Q
for k=1:50 [Q,R]=qr(A); A=R*Q; end
```

Now the matrix A (which is T_k in the QR scheme), is given by

$$A = \begin{pmatrix} 1.8000 & -0.4000 \\ -0.4000 & 1.2000 \end{pmatrix}.$$

A few more times yields

$$A = \begin{pmatrix} 2.000 & -0.0000 \\ -0.0000 & 1.000 \end{pmatrix}.$$

9.2.1 Why does the QR algorithm work?

For those who really want to know, a little background why the QR algorithm actually works.

Let X be a generic invertible $(n \times n)$ matrix, and consider $A^k X$, or better, orthonormalised at every step. Taking $X_0 = I$ and

$$\begin{aligned} Y_k &= \text{ orthonormalise the vectors in } X_k \\ X_{k+1} &= AY_k \end{aligned}$$

with $X_k = \tilde{Q}_k R_k$ we get $\tilde{Q}_k = Y_k$. You can see that this is very similar to the Power method, see also the discussion at the end of §9.1. We get the sequence $X_0 = I = \tilde{Q}_0 = Y_0$, $X_1 = A = \tilde{Q}_1 R_1$, $Y_1 = \tilde{Q}_1$, $X_2 = A\tilde{Q}_1 = \tilde{Q}_2 R_2$, $X_3 = A\tilde{Q}_2 = \tilde{Q}_3 R_3$, and more generally $X_k = \tilde{Q}_k R_k$ and $X_{k+1} = A\tilde{Q}_k$.

$Y_k = \tilde{Q}_k$ converges to a basis of eigenvectors V, and $\tilde{Q}_k^* A \tilde{Q}_k$ converges to $V^* A V = \Lambda$.

This is the QR algorithm in disguise, because if we define $S_k = \tilde{Q}_{k-1}^* A \tilde{Q}_{k-1}$ and $Q_k = \tilde{Q}_{k-1}^* \tilde{Q}_k$, then $S_k \to \Lambda$ if $k \to \infty$, and from the preceding it follows that $A\tilde{Q}_{k-1} = \tilde{Q}_k R_k$. Hence

$$\begin{aligned} S_k &= \tilde{Q}_{k-1}^* \tilde{Q}_k \tilde{Q}_k^* A \tilde{Q}_{k-1} = Q_k R_k \\ S_{k+1} &= \tilde{Q}_k^* A \tilde{Q}_{k-1} \tilde{Q}_{k-1}^* \tilde{Q}_k = R_k Q_k \end{aligned}$$

We conclude that $T_k = S_k$ (since $T_1 = S_1 = A$).

9.3 Sensitivity of eigenvalues

You may wonder whether it is reasonable to compute eigenvalues numerically at all! After all, you never work with the real matrix, always with an approximation of it. So why should the eigenvalues of this approximate matrix be close to the eigenvalues of the real one? We first look at this problem from a general perspective, and then try to get some insight for which matrices the computation of eigenvalues is actually a poorly-defined problem.

There are two major reasons why for generic matrices, computing the eigenvalues is a reasonable problem.

- The Implicit Function Theorem (IFT): the eigenvalues are the roots of the characteristic equation of A. The coefficients in that equations are defined by matrix entries. In fact, they are *continuous functions* of those matrix entries. The IFT now tells you that 'locally' a real 'transverse' root λ of the characteristic polynomial may be written as a continuous function of those coefficients. That means that the roots, provided they are real and of multiplicity 1, will not change much when the coefficients are changed, and hence when the matrix entries are slightly perturbed. But what happens for degenerate (double) roots? And complex roots?
- A more general way to look at this, is to recall Cauchy's argument principle from Complex Analysis. It says that $\frac{1}{2\pi i} \oint_C \frac{f(z)}{f'(z)} = N - P$, where N is the number of zeros of $f(z)$ within the domain bounded by the curve C, and P the number of poles. If we apply this to the characteristic polynomial, then we see that the number of poles is zero. By considering a curve C that forms a small circle around a zero, we see that fiddling with the coefficients in the polynomial will mean that the number of roots within this circle, counted with multiplicity, cannot change abruptly. This gives continuous dependence of the eigenvalues with respect to changes in the coefficients of the matrix.

To get a more quantitative understanding of the sensitivity of the eigenvalues when the coefficients in the matrix are varied, we argue as follows. If A is diagonalisable then $\Lambda = X^{-1} A X$. Now let δA be a change in A, then naively $\|\delta\Lambda\| \leq \|X^{-1}\| \|\delta A\| \|X\| = K(X) \|\delta A\|$. *However*, $\delta\Lambda$ is not diagonal (or better, $A + \delta A$ is not necessarily diagonalisable, and even if it is, there is also a δX). Nevertheless the conclusion that the sensitivity of the eigenvalues depends on the condition number of the matrix of eigenvectors (not of A) is correct.

To see this, it is better to also consider the left-eigenvectors $y^* A = \lambda y^*$. If λ is a simple eigenvalue (depends continuously on A), then there exist x and y such that $Ax = \lambda x$ and $y^* A = \lambda y^*$. We write out the perturbed problem: $(A + \delta A)(x + \delta x) + \text{h.o.t.} = (\lambda + \delta\lambda)(x + \delta x) + \text{h.o.t.}$, hence $\delta A x + A \delta x \approx \delta\lambda x + \lambda \delta x$.

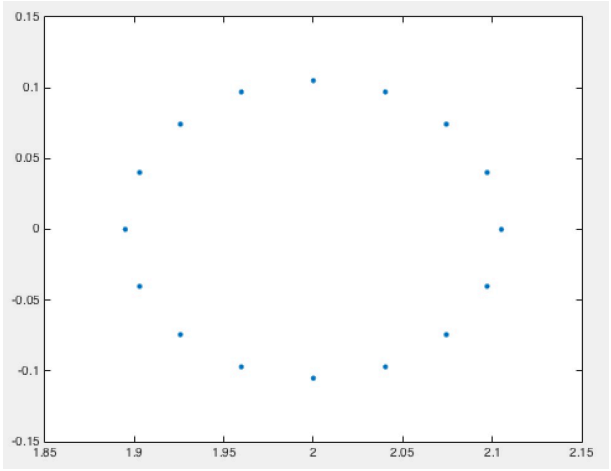

FIGURE 9.2. An example of eigenvalues for a badly conditioned matrix.

Now multiply on the left with y^*, $y^*\delta A x + y^* A\delta x = y^*\delta\lambda x + y^*\lambda\delta x$. Two terms cancel:

$$\delta\lambda = \frac{y^*\delta A x}{y^* x} \leq \frac{|y||x|}{|y^*x|}\|\delta A\|$$

This leads us to introduce the *eigenvalue condition number* $\kappa(\lambda, A) = \frac{|y||x|}{|y^*x|} \geq 1$, which is independent of the lengths of the eigenvectors.

If X is a matrix of right eigenvectors (A diagonalisable) then you can take $Y^* = X^{-1}$ (because $A = X\Lambda X^{-1}$, hence $Y^*A = \Lambda Y^*$).

We obtain $Y^*X = I$ and $y^*x = 1$, as well as $\kappa(\lambda, A) = |y||x| \leq \|X^{-1}\|\|X\| = K(X)$, where we used that $|x| \leq \|X\|$ and $|y| \leq \|Y\|$.

If $A^* = A$ (self-adjoint) then $y = x^*$, hence $k(\lambda, A) = 1$. We conclude that for symmetric (self-adjoint) matrices the eigenvalue problem is well conditioned.

If the algebraic multiplicity is not equal to the geometric multiplicity, then computing eigenvalues numerically is badly conditioned: $\det(A - \lambda I) = (\lambda - \bar{\lambda})^m q(\lambda) \approx C(\lambda - \bar{\lambda})^m = \delta$ then $\lambda = \bar{\lambda} + O(\delta^{1/m})$. An example $(n \times n)$ matrix is

$$A = \begin{pmatrix} 2 & 1 & & & \\ & 2 & 1 & & \\ & & \ddots & \ddots & \\ & & & 2 & 1 \\ \delta & & & & 2 \end{pmatrix}$$

The characteristic polynomial is $(\lambda - 2)^n \pm \delta$, hence with $\delta = 10^{-n}$ we get $|\lambda - 2| \approx \frac{1}{10}$. Trying to compute eigenvalues in MATLAB is done e.g. by

```
A=2*eye(16)+diag(ones(1,15),1)
eig(A)
A(16,1)=eps
q=eig(A)
plot(q,'.');
```

The resulting numerical estimate of the eigenvalues are shown in Figure 9.2.

Chapter 10

Ordinary Differential Equations

10.1 Introduction to differential equations

An (ordinary) differential equation (ODE) is an equation for a function $y(t)$, in which both the function and its derivative(s) appear. We will focus on the most common type of ODEs. For $y : \mathbb{R} \to \mathbb{R}^n$, we will consider the *initial value problem*

$$\begin{cases} y'(t) = f(t, y(t)), \\ y(t_0) = y_0, \end{cases}$$

with $f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ and the initial data $y_0 \in \mathbb{R}^n$ given. The question is what we can say about functions $y(t)$ that solve such equations. Does a solution exist, does it exist for all time, what it the behaviour of the solution as time continues, how does the solution depend on the chosen starting point y_0? These are some of the basic questions.

ODEs have their origin in physical laws, and in book keeping. A familiar example is e.g., Newton's law, $F = ma$, which states that a force F acting on a mass m causes an acceleration a in such a way that $F = ma$. If we denote the position of this mass at time t by $y(t)$, then $a = y''(t)$. In many cases, the force acting on the mass is somehow related to where the mass is. In the simplest example of a mass hanging from a spring, the magnitude of the force with which the spring pulls the mass depends on the extent to which the spring is extended (and thus changes when the mass itself changes position). In a linear spring, the force is proportional to this extension, and then the force of the spring will counteract displacement and thus have opposite sign (compared to the displacement). All this brings us to

$$F_s = -ky(t),$$

so that $F = ma$ reads

$$-ky(t) = my''(t).$$

We have ourselves an ODE (and we can see now why the mass goes up and down: example solutions $y(t)$ to this equation are sines and cosines, in particular $\sin(\sqrt{\frac{k}{m}}t)$ and $\cos(\sqrt{\frac{k}{m}}t)$).

You may have see ODE theory before, but for those who have not, we highlight again the important problem of well-posedness: for which functions f does the initial value problem have a unique solution (for some time)?

Assume that f is Lipschitz continuous in y with a constant L independent of t, i.e.,

$$|f(t, y_1) - f(t, y_2)| \leq L|y_1 - y_2| \qquad \text{for all } t \in \mathbb{R} \text{ and } y_1, y_2 \in \mathbb{R}^n.$$

Then the IVP has a unique solution $y(t) \in C^1$ for $t \in \mathbb{R}$.

Of course in applications, it is easier to assume that f is a C^1 function.

An example for which Lipschitz continuity may be checked very quickly is $u' = u/(1 + t^2)$, $u(0) = 1$. (Try it!)

The conclusion of the existence of a unique solution also holds if the Lipschitz constant (or continuous differentiability) is not for all y or all t. For example, if f is Lipschitz on $I \times D$, and $t_0 \in \text{int}(I)$ and $y_0 \in \text{int}(D)$. Then there is a small piece of solution ($|t - t_0| < \delta$) and it is unique.

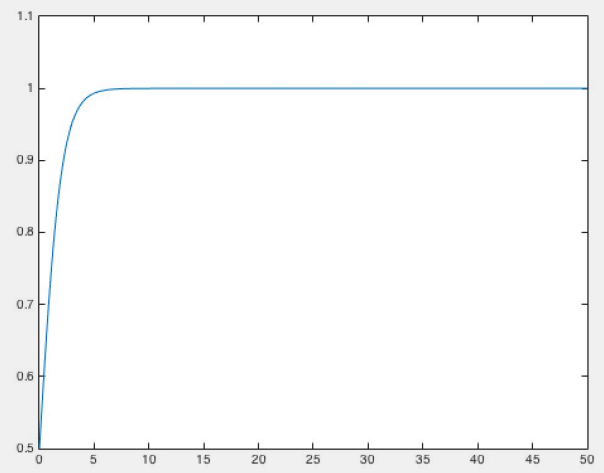

FIGURE 10.1. An example simulation of the logistic equation.

The solution can only terminate (or stop being unique) when $t \to \pm\infty$, or $|y| \to \infty$, or if the solution approaches a singularity of f (in t or y), where f is no longer Lipschitz.

The Lipschitz condition is locally satisfied if f is a C^1-function. That is also the easiest way to see that a function is Lipschitz.

Let us consider a few examples.

- $u' = u^2$ with $u(0) = 1$, solution $u = \frac{1}{1-t}$ exists for $t \in [0, 1)$ and $u \to \infty$ as $t \to 1$.
- $u' = -1/u$ with $u(0) = 1$, solution $u = \sqrt{1-2t}$ exists for $t \in [0, \frac{1}{2})$ and u approached a singularity of f as $t \to \frac{1}{2}$.
- $u' = \sqrt{u}$ with $u(0) = 1$, solution $\frac{1}{4}(t+2)^2$, but uniqueness does not hold for not $u' = \sqrt{u}$ with $u(0) = 0$, since both $u = 0$ and $u = \frac{1}{4}t^2$ are solutions for $t \geq 0$ (and there are many more).

For higher order equations, such as $v'' = g(v)$, we can still include these in our general class of ODEs $y' = f(t, y(t))$. We introduce $w = v'$, then $v' = w$ and $w' = g(v)$, so we have system of two first-order equations instead of one second-order equations. Indeed, set $u = (u_1, u_2)^T$ and

$$\frac{d}{dt}\begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} u_2 \\ g(u_1) \end{pmatrix} = f(u).$$

Similarly, for higher order: $v''' + v'' - v' = v^2$ becomes $u_1' = u_2$, $u_2' = u_3$ and $u_3' = -u_3 + u_2 + u_1^2$. Hence $f(u) = (u_2, u_3, -u_3 + u_2 + u_1^2)^T$.

In this course we will cover a number of numerical algorithms to find numerical solutions to initial value problems. Of course, MATLAB has some of these *solvers* already built in, making it often very easy to integrate ODEs. To get a first flavour, let's do an example of an easy equation for which we know the solution analytically (by separation of variables).

The following bit of code defines the ODE $x' = x(1-x)$, sets some `options` which increase the accuracy compared to the default settings, and then integrates the ODE from the starting point $x(0) = 0.5$ using the built-in solver `ode45`

```
f=@(t,x) x.*(1-x)
options=odeset('AbsTol',1e-8,'RelTol',1e-8);
[t,x]=ode45(f,[0 50],0.5,options);
plot(t,x)
```

Note that the ODE solver wants a function of t and x, even if the function itself is dependent only on x. Hence `f = @(t,x) x.*(1-x)`. We had a similar situation with the function `numjac`, which is actually used by ODE solvers under the hood. The result of this simulation is found in Figure 10.1.

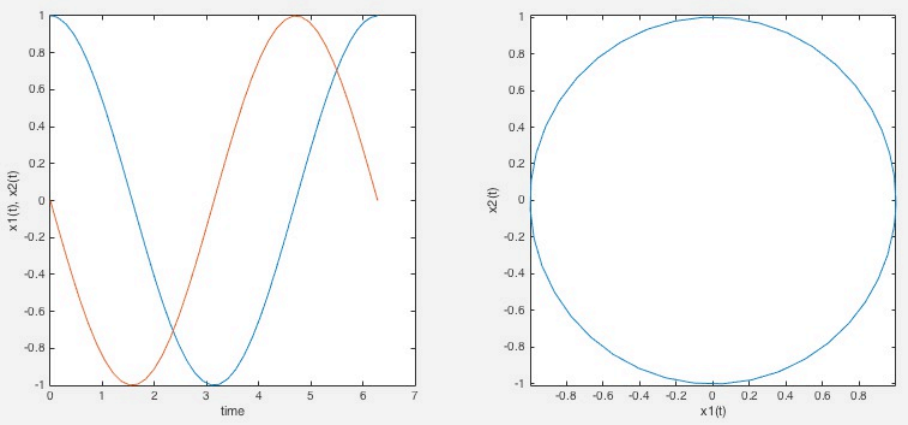

FIGURE 10.2. An example simulation $x'' = -x$.

Here is another example, which shows you how to simulate $x'' = -x$, by converting it into a system of two first-order equations.

```
g=@(t,x) [x(2);-x(1)]
[t,x]=ode45(g,[0 2*pi],[1;0]);
subplot(1,2,1)
plot(t,x)
xlabel('time')
ylabel('x1(t), x2(t)')
subplot(1,2,2)
plot(x(:,1),x(:,2),'-')
xlabel('x1(t)'); ylabel('x2(t)')
```

We can of course plot the solutions $x_1(t)$ and $x_2(t) = x_1'(t)$ over time, but it is often more instructive to draw a phase plot, in which we plot x_1 against x_2. Both options are shown in Figure 10.2.

You may wonder, perhaps, why you have to learn those algorithms, if it is so easy to integrate ODEs numerically. Well, not is as easy as it seems... Let us do last example again, but now integrate much further (see Figure 10.3).

```
[t,x]=ode45(g,[0 2000*pi],[1;0]);
subplot(1,2,1)
plot(x(:,1),x(:,2))
subplot(1,2,2)
plot(x(:,1).^2+x(:,2).^2)
```

Clearly, something has gone wrong! We know that the solution to the ODE has to be periodic, but the simulation shows a solution which spirals inwards. The left picture is a bit fuzzy because lines overlap, but in the right picture, you can see the distance of the point $(x_1(t), x_2(t)$ to the origin. In a perfect circle (the real solution) this distance should remain 1, but in the simulation the distance decreases.

This shows that you cannot always be certain that the simulation gives reasonable results. In any case, it shows that you have to think about what those solvers are doing, what they are good at and when they fail.

Use for the assignments always `ode45` or `ode15s` with a smaller error (make a good choice) than the default. To work with these solvers, it is often useful for the function f in the ODE to be a MATLAB function (that you code yourself). Then you have to use a function handle to this function. In other words,

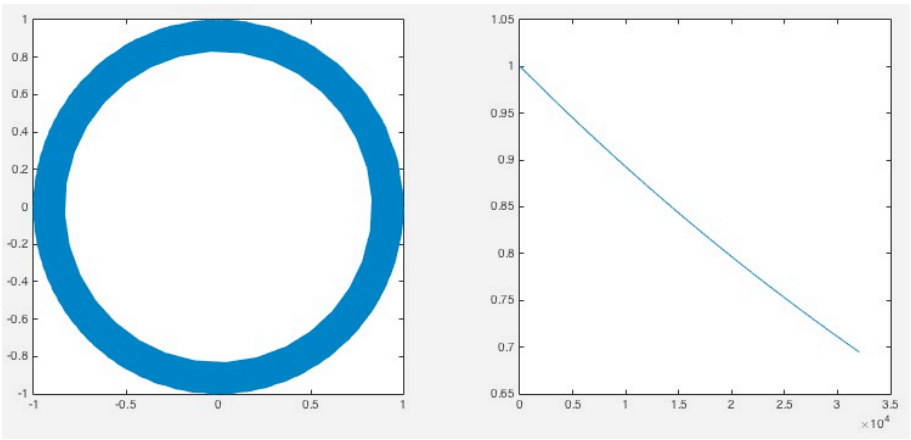

FIGURE 10.3. An example simulation $x'' = -x$, in which time has run on and on. Left, the numerical solution (it is hard to see, but it spirals inwards over time), and right the distance of the solution to the origin over time. It should not decrease, but it clearly does.

```
[t,x] = ode45(@rhs,[0,T],x0,options);
...
function dx = rhs(t,x);
% definition of the right hand side of x'=f(t,x(t))
```

10.2 Simple methods

Let's first introduce some notation. Let $N_h = T/h$, $u_n = u(nh)$ and $f_n = f(nh, u_n)$.

If we start with

$$u'(t) \approx \frac{u(t+\Delta t) - u(t)}{\Delta t} = f(t, u(t)),$$

then

$$u(t+\Delta t) = u(t) + \Delta t f(t, u(t)) + \mathcal{O}((\Delta t)^2).$$

This leads to the *explicit Euler* scheme

$$\boxed{u_{n+1} = u_n + hf_n = u_n + hf(nh, u_n).}$$

Of course, we could have also approximated right hand side,

$$u'(t) \approx \frac{u(t+\Delta t) - u(t)}{\Delta t} = f(t+\Delta t, u(t+\Delta t)).$$

This yields the *implicit Euler* scheme

$$\boxed{u_{n+1} = u_n + hf((n+1)h, u_{n+1}) (+\mathcal{O}(h^2)).}$$

Note that this is not an expression for u_{n+1} in terms of the n and u_n, but it is an *equation* for u_{n+1} (it also appears in the right hand side). Hence the name.

As a third starting point, we could have made the following observation. If $u' = f(t, u(t))$ for $t \in [0, T]$, and $y(t_0) = y_0$ for some $t_0 \in (0, T)$, then

$$\int_{t_0}^{t_0+\Delta t} f(t, y(t))dt = \int_{t_0}^{t_0+\Delta t} y'(t) = y(t_0 + \Delta t) - y(t_0).$$

Now we can approximate the integral on the left using one of our numerical integration schemes. For instance, if we use the trapezium rule, then

$$\int_{t_0}^{t_0+\Delta t} f(t,y(t))dt = \frac{\Delta t}{2}\left(f(t_0,y(t_0)) + f(t_0+\Delta t, y(t_0+\Delta t)) + \mathcal{O}((\Delta t)^3).\right)$$

A bit of rewriting gives the *Crank-Nicolson* scheme

$$\boxed{u_{n+1} = u_n + h\left(\frac{1}{2}f_n + \frac{1}{2}f_{n+1}\right),}$$

where $f_k = f(kh, u_k)$. This method therefore already has better convergence than the two Euler methods.

If $|y(T) - u_{N_h}| \leq Ch^p$, then we say that p is the order of convergence. A rough estimate is: if $u_n = y(nh)$ implies that $u_{n+1} - y((n+1)h) \leq Ch^q$, than the order of convergence is $p = q - 1$, since you have to sum over the number of steps $N = 1/h$.

Implicit solvers are of course more expensive computationally than explicit solvers. But hopefully you can take larger steps.

The following theorem is quite worrisome. It states that solutions to differential equations may diverge exponentially.

THEOREM 10.2.1. *Let y^1 and y^2 be solutions of $y' = f(t,y)$ with $y^1(0) = y_0^1$ and $y^2(0) = y_0^2$, and let f have Lipschitz constant L, then*

$$|y^2(t) - y^1(t)| \leq e^{Lt}|y_0^2 - y_0^1|$$

How bad is this? Real bad! This theorem is sharp, as the following example with the simplest ODE already shows. Take $f(t,y) = y$. Then $y^i = y_0^i e^{Lt}$. The conclusion is that errors grow exponentially in general, even *without* numerical approximations.

A few words on nonlinear problems (which is where all the fun is). Nonlinear problems are *hard*, even without numerical approximation. To test a new integration method, test it on linear problems since they give good intuition about the method. Extrapolation to nonlinear problems usually does not give precise theorems. Well, in fact it is fine for finite time intervals and infinitesimal step sizes (we can then still control the order of convergence). But statements about long time intervals are always difficult. One general conclusion that may be drawn is: implicit methods are often better to simulate stable dynamics. But unstable dynamics and/or chaos is much harder, and there are no easy solutions. Just to give one example, the Lorenz system (a small (3 dimensional) set of nonlinear differential equations)

$$\begin{cases} x' = 10(y-x) \\ y' = x(28-z) - y \\ z' = xy - \frac{8}{3}z \end{cases}$$

shows seemingly chaotic behaviour. It was shown only very recently that the numerical simulations could be trusted, in the sense that the strange butterfly-like orbit called the Lorentz-attractor is not a numerical artefact: it actually exists, and the orbit is indeed chaotic. Here is some MATLAB code that illustrates (in Figure 10.4) the chaotic dynamics of the Lorenz system.

```
sigma=10;
beta=8/3;
rho=28;
f=@(t,x) [sigma*(x(2)-x(1));x(1)*(rho-x(3))-x(2);x(1)*x(2)-beta*x(3)];
options=odeset('AbsTol',1e-8,'RelTol',1e-8);
[t,x]=ode45(f,[0 100],[1;1;1],options);
plot3(x(:,1),x(:,2),x(:,3))
```

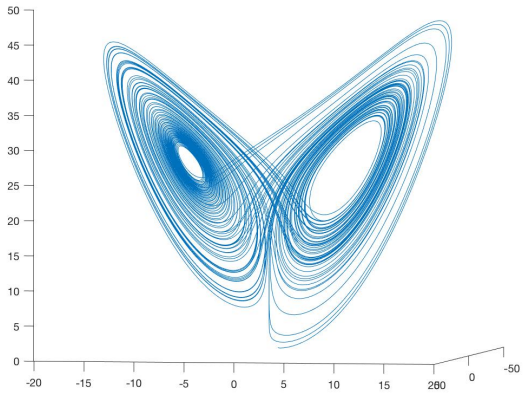

FIGURE 10.4. The chaotic attractor of the Lorenz system looks like a butterfly.

10.3 Runge-Kutta methods

The most commonly used standard methods to integrate an ODE system are the *Runge-Kutta methods*, most often the explicit ones.

The main idea is to divide the integration step from t_i to t_{i+1} into s steps, approximate $f(t, y)$ at these steps, and choose some introduced constants in such a way as to get a high order of convergence.

The general setup is $y_{n+1} = y_n + h\sum_{i=1}^{s} b_i K_i$ and s the number of steps, where K_i are approximations of $y' = f(t, y)$ in intermediate points (τ_i, ξ_i) where

$$\tau_i = t_n + c_i h,$$

(with $0 \leq c_1 \leq \cdots \leq c_s \leq 1$), and

$$\xi_i = y_n + h\sum_{j=1}^{s} a_{ij} K_j,$$

so that

$$K_i = f(\tau_i, \xi_i) = f(t_n + c_i h, y_n + h\sum_{j=1}^{s} a_{ij} K_j).$$

We take $\sum_{i=1}^{s} b_i = 1$ (a logical choice, otherwise it does not even work for $y' = 1$, and is not consistent, see below), and need to require that $\sum_{j=1}^{s} a_{ij} = c_i$ to have any chance of finding a consistent solution. The method is explicit if $a_{ij} = 0$ for $j \geq i$.

As an example, an explicit 3-step method would be given as follows. Choose values b_1, b_2, b_3 and c_1, c_2, c_3 and a_{21}, a_{31}, a_{32}, such that $b_1 + b_2 + b_3 = 1$ and

$$A = \begin{pmatrix} 0 & 0 & 0 \\ a_{21} & 0 & 0 \\ a_{31} & a_{32} & 0 \end{pmatrix} \qquad \begin{aligned} c_1 &= 0, \\ c_2 &= a_{21}, \\ c_3 &= a_{31} + a_{32}. \end{aligned}$$

The main question is: what are good choices for all those constants? We consider the autonomous case $f(t, y) = f(y)$.

Denote $y = y_n$. We obtain (starting from ξ_3 below we do not explicitly write the evaluation in y anymore)

$$\begin{aligned}
\xi_1 &= y + a_{11}h = y;\\
K_1 &= f(\xi_1) = f(y);\\
\xi_2 &= y + ha_{21}K_1 = y + hc_2 f(y);\\
K_2 &= f(\xi_2) = f(y + hc_2 f(y))\\
&= f(y) + hc_2 f(y) f_y(y) + \frac{1}{2}h^2 c_2^2 f(y)^2 f_{yy}(y) + O(h^3);\\
\xi_3 &= y + ha_{31}K_1 + ha_{32}K_2\\
&= y + h(c_3 - a_{32})f + ha_{32}[f + hc_2 f f_y] + O(h^3)\\
&= y + hc_3 f + h^2 a_{32} c_2 f f_y + O(h^3);\\
K_3 &= f(\xi_3) = f(y + hc_3 f + h^2 a_{32} c_2 f f_y + O(h^3))\\
&= f + hc_3 f_y f + h^2[\frac{1}{2}c_3^2 f^2 f_{yy} + a_{32} c_2 f f_y^2] + O(h^3).
\end{aligned}$$

Substitution in $y_{n+1} = y_n + h\sum_{i=1}^3 b_i K_i$ gives

$$\begin{aligned}
y_{n+1} &= y + hb_1 f + hb_2[f + c_2 f_y f + \frac{1}{2}h^2 c_2^2 f^2 f_{yy}]\\
&\quad + hb_3[f + hc_3 f_y f + h^2(\frac{1}{2}c_3^2 f^2 f_{yy} + a_{32} c_2 f f_y{}^2)] + O(h^4)\\
&= y_n + h(b_1 + b_2 + b_3)f + h^2(b_2 c_2 + b_3 c_3) f_y f + h^3[\frac{1}{2}(b_2 c_2^2 + b_3 c_3^2) f^2 f_{yy} + b_3 a_{32} c_2 f f_y{}^2] + O(h^4)
\end{aligned}$$

The true solution satisfies

$$\begin{aligned}
\tilde{y}_{n+1} = y(t_{n+1}) &= y(t_n + h) = y(t_n) + hy'(t_n) + \frac{1}{2}h^2 y''(t_n) + \frac{1}{6}h^3 y'''(t_n) + O(h^4)\\
&= y_n + hf + \frac{1}{2}h^2 f_y f + \frac{1}{6}h^3(f_y^2 f + f_{yy} f^2) + O(h^4)
\end{aligned}$$

since $y'' = f(y)' = f_y y' = f_y f$ and $y''' = (f_{yy} f + f_y^2) y'$. Comparison of the coefficients gives

$$b_1 + b_2 + b_3 = 1, \quad b_2 c_2 + b_3 c_3 = \frac{1}{2}, \quad b_2 c_2^2 + b_3 c_3^2 = \frac{1}{3}, \quad b_3 a_{32} c_2 = \frac{1}{6}.$$

Then the method is of order 3. For example (the classical choice): $c_2 = \frac{1}{2}$, $c_3 = 1$, $b_2 = \frac{2}{3}$, $b_3 = \frac{1}{6}$, $b_1 = \frac{1}{6}$ and $a_{32} = 2$, $a_{31} = -1$.

More often used is RK4, an explicit 4-step Runge-Kutta method of order 4, which, when combined with an error estimator, so that the step size can be adapted automatically, yields the by now familiar MATLAB ODE solver `ode45`.

It is good to remember that for $s \geq 5$ there exists no s-step explicit Runge-Kutta method which has order of convergence s! The equations for the constants simply can never be solved in these cases.

Chapter 11

Partial Differential Equations

11.1 Introduction

The field of Partial Differential Equations (PDEs) is vast, and much less cohesive than that of ODEs. Numerous methods (analytic and numeric) exists for specific equations, and there is little general theory other than for linear PDEs. A proper treatment of numerical methods for PDEs is therefore definitely beyond this course. But it is still useful to know that some PDEs may be integrated numerically as if they were big systems of coupled ODEs. Before we illustrate this point, we give a very rudimentary overview of PDEs.

There are basically three types of problems (with physical interpretations).

- The Initial value problem

$$\begin{cases} u_t = u_{xx} \\ u(0,x) = u_0(x) \\ u(t,0) = u(t,1) = 0 \end{cases}$$

  (this is called the *heat* or *diffusion* equation). It often has a unique solution.
- The Boundary value problem (sometimes written as $-u_{xx} = f$),

$$\begin{cases} u_{xx} = f(x) \\ u_x(0) = u_x(1) = 0 \end{cases}$$

  (called the *Laplace* equation.) It does not always have a unique solution, but sometimes it does.
- Then there are Eigenvalue problems

$$\begin{cases} u_{xx} = \lambda u \\ u(0) = u(1),\ u_x(0) = u_x(1) \end{cases}$$

  These generally have many solutions (eigenvalues with corresponding eigen*functions*).

In the above examples, you also saw the three types of *boundary conditions*: Dirichlet (zero on the boundary), Neumann (derivative zero on the boundary), and periodic (beginning and end points coincide, as do the derivatives).

The order of the equation and the precise form may vary, as well as the number of spatial dimensions.

11.1.1 A first Boundary Value Problem: Finite differences

Now let us solve $u''(x) = f(x)$ with $u(a) = u(b) = 0$. This seems like an ODE problem, but note that we did not prescribe two initial conditions, such as $u(a) = 0, u'(a) = 1$, but the beginning and end point! We can therefore not start integrating the solution from $x = a$ to $x = b$. We have to find the entire solution $u(x)$ in one go! This makes it more of a PDE problem.

We introduce some notation again. Set $h = (b-a)/(N-1)$ with $x_i = a + (i-1)h$, $i = 1, \ldots N$, u_i the approximation of $u(x_i)$, and $f_i = f(x_i)$.

Write the equations in matrix form and remove the last row/column, since $u_1 = u_N = 0$ are not unknowns. This leads to an $(N-2)\times(N-2)$ symmetric matrix

$$\begin{pmatrix} -2 & 1 & 0 & \cdots & 0 & 0 \\ 1 & -2 & 1 & & & 0 \\ 0 & 1 & -2 & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & 1 & 0 \\ 0 & & \cdots & 1 & -2 & 1 \\ 0 & 0 & \cdots & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} u_2 \\ u_3 \\ u_4 \\ \vdots \\ u_{N-2} \\ u_{N-1} \end{pmatrix} = h^2 \begin{pmatrix} f_2 \\ f_3 \\ f_4 \\ \vdots \\ f_{N-2} \\ f_{N-1} \end{pmatrix}$$

This equation is accurate up to order h^4. This linear system can be solved (the matrix is invertible) and the numerical solution turns out to have an error of order h^2.

We may apply different boundary conditions (BCs).

- With boundary conditions $u(a) = \alpha$ and $u(b) = \beta$:
  replace f_2 by $f_2 - \alpha/h^2$ and f_{N-1} by $f_{N-1} - \beta/h^2$.
- For Neumann BCs, i.e., $u'(a) = 0$ and $u'(b) = 0$, the easiest way is to set $u_1 = u_2$ and $u_N = u_{N-1}$. Then the matrix becomes $N \times N$

$$A = \begin{pmatrix} -1 & 1 & 0 & \cdots & 0 & 0 \\ 1 & -2 & 1 & & & 0 \\ 0 & 1 & -2 & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & 1 & 0 \\ 0 & & \cdots & 1 & -2 & 1 \\ 0 & 0 & \cdots & 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-1} \\ u_N \end{pmatrix} = h^2 \begin{pmatrix} 0 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-1} \\ 0 \end{pmatrix}$$

  You can again remove the first and last row and column (since you can infer u_1 from u_2, etc.), hence it suffices to solve the following $(N-2)\times(N-2)$ system (which has exactly the same matrix shape as A, don't be confused!):

$$\begin{pmatrix} -1 & 1 & 0 & \cdots & 0 & 0 \\ 1 & -2 & 1 & & & 0 \\ 0 & 1 & -2 & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & 1 & 0 \\ 0 & & \cdots & 1 & -2 & 1 \\ 0 & 0 & \cdots & 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} u_2 \\ u_3 \\ \vdots \\ u_{N-1} \end{pmatrix} = h^2 \begin{pmatrix} f_2 \\ f_3 \\ \vdots \\ f_{N-1} \end{pmatrix}$$

- An alternative way to implement Neumann BCs is to recall how to choose the finite difference approximation for a point on the boundary: use $u_1 = \frac{4}{3}u_2 - \frac{1}{3}u_3$. This has the advantage of being more accurate, but the disadvantage that the matrix is no longer symmetric.
- For periodic BCs, set $u_N = u_1$. Now we are left with a $(N-1)\times(N-1)$ matrix:

$$\begin{pmatrix} -2 & 1 & 0 & \cdots & 0 & 1 \\ 1 & -2 & 1 & & & 0 \\ 0 & 1 & -2 & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & 1 & 0 \\ 0 & & \cdots & 1 & -2 & 1 \\ 1 & 0 & \cdots & 0 & 1 & -2 \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ u_{N-1} \end{pmatrix} = h^2 \begin{pmatrix} (f_1 + f_N)/2 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-1} \end{pmatrix}$$

For both Neumann and periodic BCs, the solution is not unique, since the kernel is $\mathrm{sp}(1, 1, \ldots, 1)$.

11.2 PDEs as ODEs

As hinted in the introduction to PDEs, ODE methods are also useful for some real *Partial differential equations*. Consider

$$u_t = f(u, u_x, u_{xx}, \ldots, x, t).$$

discretise the *spatial* variable, $x = i\Delta x$, and introduce $u^i(t) = u(t, i\Delta x)$. Then, with centered differences, we can set up a large system of coupled ODEs for the u_i,

$$u_t^i = f\left(u^i, \frac{u^{i+1} - u^{i-1}}{2\Delta x}, \frac{u^{i+1} - 2u^i + u^{i-1}}{(\Delta x)^2}, \ldots, i\Delta x, t\right).$$

for $i = 0, 1, 2, \ldots$.

We again need to specify some boundary conditions to complete the PDE problem.

As an example, take the heat equation $u_t = u_{xx}$ on $x \in [0, 1]$ with periodic boundary conditions $u(t, 0) = u(t, 1)$, and initial conditions $u(0, x) = u_0(x)$.

Let us now take N discretisation intervals, so that $\Delta x = 1/N$. Set $u^i = u(i\Delta x)$ for $i = 1, \ldots, N$ (numerically handy for MATLAB), and $u^0 = u^N$, but u^0 is thus not involved in the computation. Then the coupled system of ODEs for the u_i variables is given by

$$u_t^i = \frac{u^{i-1} - 2u^i + u^{i+1}}{(\Delta x)^2}$$

for $i = 1, \ldots, N$, if we agree that $u^0 = u^N$ and $u^{N+1} = u^1$.

Put everything into one vector (i.e. $u = (u^1, \ldots, u^N)^T$), so that

$$u_t^j = f(u) = \begin{cases} \frac{u^N - 2u^1 + u^2}{(\Delta x)^2} & j = 1 \\ \frac{u^{j-1} - 2u^j + u^{j+1}}{(\Delta x)^2} & j = 2, \ldots, N-1 \\ \frac{u^{N-1} - 2u^N + u^1}{(\Delta x)^2} & j = N. \end{cases}$$

Here

$$u^j(0) = u_0(j\Delta x) \qquad j = 1, \ldots, N$$

Of course, do not forget to add the boundary point at $x = 0$ before you plot!

The following bit of code implements the scheme above, and the result may be found in Figure 11.1. The command `eye` makes an identity matrix, `triu` gives an upper triangular matrix, and `ones` fill a matrix with ones.

```
n=50;
B=triu(ones(n),1) - triu(ones(n),2);
A = -2*eye(n) + B + B';
A(1,n) = 1; % for the BCs
A(n,1) = 1; % for the BCs
f = @(t,u) A*u;
IC = zeros(n,1);
IC(n/4:3*n/4) = 1; % a block of mass in the middle of the domain as initial condition
[t,u] = ode45(f,[0,10],IC);
surf(t,[1:n],u');
xlabel('time');
ylabel('space');
zlabel('u(t,x)');
```

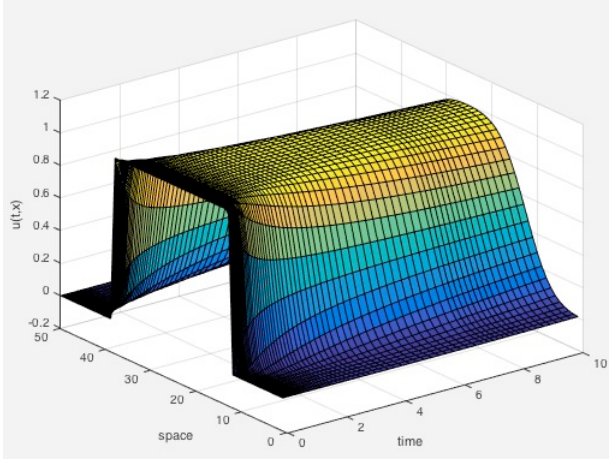

FIGURE 11.1. An example simulation of the heat equation with periodic boundary conditions.

11.3 Finite element methods

Finite element methods form a counterpart to finite differences. In general they are a better approach, and are more widely used, but they are more time-intensive to implement. It is a vast subject lying on the boundary of partial differential equations, functional analysis en numerical analysis.

The first step is to rewrite your equation in the so-called *weak formulation*. Let us consider the toy problem $u'' = f$ with $u(a) = 0$ en $u(b) = 0$, Multiply the differential equation by a test function $\phi(x)$, and integrate over the domain $[a, b]$. Then perform one integration by parts and use the boundary conditions to conclude

$$-\int_a^b u'(x)\overline{\phi}'(x)dx = \int_a^b f(x)\overline{\phi}(x)dx$$

where the bar denotes complex conjugation (only for the complex case), and ϕ for example an *arbitrary* C^1 function with $\phi(a) = \phi(b) = 0$.

Choose a "simple" space X of functions that are 0 on the boundary (containing $C_0^\infty([a, b])$), and find the function $u \in X$ in this space which solves the equation for all $\phi \in X$.

Numerically, of course, X is finite dimensional.

Take a basis of X: $\phi_1, \ldots \phi_N, \ldots$, do a finite dimensional truncation, so that

$$u = \sum_{k=1}^{N} a_k \phi_k.$$

The problem is therefore to solve for a_k

$$-\sum_{k=1}^{N} a_k \int_a^b \phi_k'(x)\overline{\phi_j}'(x) = \int_a^b f(x)\overline{\phi_j}(x)dx \qquad \text{for } j = 1, \ldots, N.$$

This is of the form

$$-\sum_k c_{jk}a_k = b_j$$

with unknowns a_k and known integrals c_{jk} and b_j, in other words, a matrix equation (linear system)!

You would like to make choices such that the integrals are easy to calculate and such that in the truncated X there is a good approximation of the solution.

Important example of classes of functions (the "elements") that make up X are

- Taylor polynomials
- Piecewise linear functions ("tent functions")
- Fourier series, for period boundary conditions.

In general, you should triangulate your domain and use functions with small support. This way you are very flexible (in terms of resolution) and can do very general domain (which is particularly hard with finite differences). Now imagine trying to make Max Verstappen's car go 1% faster by reducing air resistance, and you realise that finite element methods (for air flow) are *the* way to numerically investigate aerodynamic properties of new car designs. Computational Fluid Dynamics is an important field using FEMs.

We will focus here on periodic boundary conditions as an illustration, and thus express solutions in terms of Fourier series. Hence, we set

$$\phi_k = e^{\frac{2\pi}{L} ikx},$$

and with

$$u = \sum_{k=-N}^{N} a_k \phi_k,$$

we have

$$-\sum_{k=-N}^{N} a_k \int_0^L \phi_k'(x)\overline{\phi_j}'(x) = \int_0^L f(x)\overline{\phi_j}(x)dx,$$

$$-a_k \frac{4\pi^2}{L^2} L = L\,\mathcal{F}(f)(k).$$

Note that this is precisely taking Fourier transformations! Solving through Fourier transformation is thus an example of a finite element method. Fourier transformation then leads to an *algebraic problem* (if we consider PDEs with constant coefficients), which we can solve *by hand* in this exceptional case, since the matrix (c_{jk}) is diagonal and can be computed analytically.

In more dimensions, let us introduce the following notation for squares/cubes:

$$\mathcal{F}(u)(k) = \frac{1}{L^n} \int_{\text{periodic cell}} u(\vec{x}) e^{-\frac{2\pi}{L} i\vec{k}\cdot\vec{x}} d\vec{x}.$$

Or

$$\hat{u}(k_x, k_y) = \mathcal{F}(u(x,y)) \qquad \hat{f}(k_x, k_y) = \mathcal{F}(f(x,y)).$$

To solve $(\Delta = \partial_x^2 + \partial_y^2)$

$$\Delta u - \mu u = f,$$

this boils down to

$$\mathcal{F}(\Delta u) = -4\pi^2 (\frac{k_x^2}{L_x^2} + \frac{k_y^2}{L_y^2}) \mathcal{F}(u).$$

We obtain, with $k = (k_x, k_y)$ and $\ell(k) = 2\pi(k_x/L_x, k_y/L_y)$,

$$\hat{u}(k) = -\frac{\hat{f}(k)}{|\ell(k)|^2 + \mu}$$

and transforming back we arrive at the actual solution,

$$u = \mathcal{F}^{-1}(\hat{u}).$$

Notice that this gives, analytically, the unique(!) solution of the periodic problem.

Some numerical implementation remarks:

- In MATLAB, take note of the ordering in the vector that is the output of `fft` or `fftn`.
- For a periodic function on $[0, L]$ you can best choose the discretisation points for the FFT in 2^q points with $x_i = \frac{i}{2^q} L$ for $i = 1, \ldots, 2^q$.
- Use reflection for the other boundary conditions (bigger domain (factor 2 or 4)), as explained below.

11.4 Reflections

Partial differential equations require boundary conditions, and these need to be implemented in the numerical schemes. Here we focus on the two most common types of boundary conditions: Dirichlet and Neumann problems. We consider $u''(x) - \mu u(x) = f(x)$ for $x \in [a, b]$ with $\mu > 0$. Let $a = 0$ without loss of generality.

For Neumann BCs, we should extend by reflection $f(-x) = f(x)$ for $x \in [0, b]$; for Dirichlet BCs, extend anti-symmetrically: $f(-x) = -f(x)$. Now think of u and f as periodic functions on $[-b, b]$ (or $[0, 2b]$), i.e., extend them periodically in your mind. Then there is a unique solution on $[-b, b]$ for the problem with periodic boundary conditions. Hence u satisfies original the differential equation, and $u(x)$ satisfies the requested boundary conditions.

The reason behind these reflections or "antireflections" is as follows. Consider the Dirichlet problem

$$\begin{cases} u'' - \mu u = f \\ u(0) = u(b) = 0 \end{cases}$$

Consider the *unique*(!) solution of

$$\begin{cases} \tilde{u}'' - \mu \tilde{u} = \tilde{f} \\ \tilde{u} \text{ periodic with period } 2b \end{cases}$$

where $\tilde{f}(-x) = -\tilde{f}(x)$ and $\tilde{f}$ is $2b$-periodic.

We claim that $u(x) = \tilde{u}(x)$ for $x \in [0, b]$ solves the Dirichlet problem. To prove this, define

$$\hat{u}(x) = -\tilde{u}(-x).$$

Then

$$\hat{u}''(x) - \mu \hat{u}(x) = -\tilde{u}''(-x) + \mu \tilde{u}(-x) = -\tilde{f}(-x) = \tilde{f}(x).$$

Hence $\hat{u}$ solves the same problem as $\tilde{u}$ and because the solution is unique: $\hat{u}(x) = \tilde{u}(x)$, or $\tilde{u}(x) = -\tilde{u}(-x)$. Therefore $\tilde{u}(0) = -\tilde{u}(0)$, or $\tilde{u}(0) = 0$; and $\tilde{u}(b) = -\tilde{u}(-b) = -u(b)$ ($2b$-periodicity), or $\tilde{u}(b) = 0$ and we have thereby proven the assertion (that it satisfies the Dirichlet boundary conditions).

For Neumann similar arguments work with $\tilde{f}(-x) = \tilde{f}(x)$ and extending periodically. Define $\hat{u}(x) = \tilde{u}(-x)$ then

$$\hat{u}''(x) - \mu \hat{u}(x) = \tilde{u}''(-x) - \mu \tilde{u}(x) = \tilde{f}(-x) = \tilde{f}(x),$$

hence $\hat{u}$ solves also the problem that $\tilde{u}$ solves and since the solution is unique: $\hat{u}(x) = \tilde{u}(x)$, or $\tilde{u}(x) = \tilde{u}(-x)$. Hence $\tilde{u}'(0) = -\tilde{u}'(0)$, or $\tilde{u}'(0) = 0$; and $\tilde{u}'(b) = -\tilde{u}'(-b) = -u'(b)$ ($2b$-periodicity), or $\tilde{u}'(b) = 0$ and we have thereby proven the assertion (that it satisfies the Neumann boundary conditions).

11.5 Finite differences in more dimensions

To extend the techniques of finite differences to more spatial dimensions does not require new ideas, just more bookkeeping! We will treat periodic boundary conditions first. Consider $u_{i,j} = u((i-1)h_x, (j-1)h_y)$, with $x \in [0, L_x]$ and $y \in [0, L_y]$ and $h_x = L_x/N_x$ and $h_y = L_y/N_y$.

For the Laplacian, we take the 5-point approximation

$$\begin{aligned} \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}\right) u(ih_x, jh_y) &\approx \frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{h_x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{h_y^2} \\ &= \frac{u_{i-1,j} + u_{i,j-1} - 4u_{i,j} + u_{i+1,j} + u_{i,j+1}}{h^2} \qquad \text{if } h_x = h_y = h. \end{aligned}$$

We wish to rewrite the entire system of equations into one linear system, so we need to convert the square set of variables into one long vector.

The usual ordering corresponds with how MATLABcounts the elements in a matrix: column-wise. (Sometimes the following command, `rot90`, is useful for visualisations, by rotating over 90 degrees.).

Counting goes as follows:

$$\begin{pmatrix} 1 & N_x+1 & 2N_x+1 & \cdots & (N_y-1)N_x+1 \\ 2 & N_x+2 & 2N_x+2 & \cdots & (N_y-1)N_x+2 \\ \vdots & \vdots & \vdots & & \vdots \\ N_x & 2N_x & 3N_x & \cdots & N_yN_x \end{pmatrix}.$$

For the matrix

$$u = \begin{pmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \cdots & u_{1,N_y} \\ u_{2,1} & u_{2,2} & u_{2,3} & \cdots & u_{2,N_y} \\ \vdots & \vdots & \vdots & & \vdots \\ u_{N_x,1} & u_{N_x,2} & u_{N_x,3} & \cdots & u_{N_x,N_y} \end{pmatrix}$$

we get a vector U by `U=u(:)`. This leads to a system of equations (with periodic boundary conditions), involving an $N_xN_y \times N_xN_y$ matrix

$$\Delta u \qquad \longrightarrow \qquad \begin{bmatrix} T & D & 0 & \cdots & 0 & D \\ D & T & D & \cdots & 0 & 0 \\ 0 & D & T & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & D & 0 \\ 0 & 0 & & D & T & D \\ D & 0 & \cdots & 0 & D & T \end{bmatrix} U$$

made up of two $N_x \times N_x$ matrices

$$D = \begin{pmatrix} \frac{1}{h_y^2} & 0 & \cdots & 0 \\ 0 & \frac{1}{h_y^2} & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \frac{1}{h_y^2} \end{pmatrix}$$

and

$$T = \begin{pmatrix} -\frac{2}{h_x^2}-\frac{2}{h_y^2} & \frac{1}{h_x^2} & 0 & \cdots & 0 & \frac{1}{h_x^2} \\ \frac{1}{h_x^2} & -\frac{2}{h_x^2}-\frac{2}{h_y^2} & \frac{1}{h_x^2} & & 0 & 0 \\ 0 & \frac{1}{h_x^2} & -\frac{2}{h_x^2}-\frac{2}{h_y^2} & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \frac{1}{h_x^2} & 0 \\ 0 & 0 & & \frac{1}{h_x^2} & -\frac{2}{h_x^2}-\frac{2}{h_y^2} & \frac{1}{h_x^2} \\ \frac{1}{h_x^2} & 0 & \cdots & 0 & \frac{1}{h_x^2} & -\frac{2}{h_x^2}-\frac{2}{h_y^2} \end{pmatrix}$$

For *Dirichlet* BCs: delete the corners, and make it $(N_x-1)(N_y-1)$; don't forget to add the boundary at the end!

For *Neumann* BCs, you may try to figure it out yourself. You won't need it for for the assignments.

11.5.1 An example

Consider the non-linear equation

$$\Delta u = -u^2$$

in two dimensions on a square with Dirichlet boundary conditions. Then the following code gives Figure 11.2.

```
N=21;N2=N-2;

% The discretised 5-point Laplacian
B=ones(N2^2,1)*[1,1,-4,1,1];
```

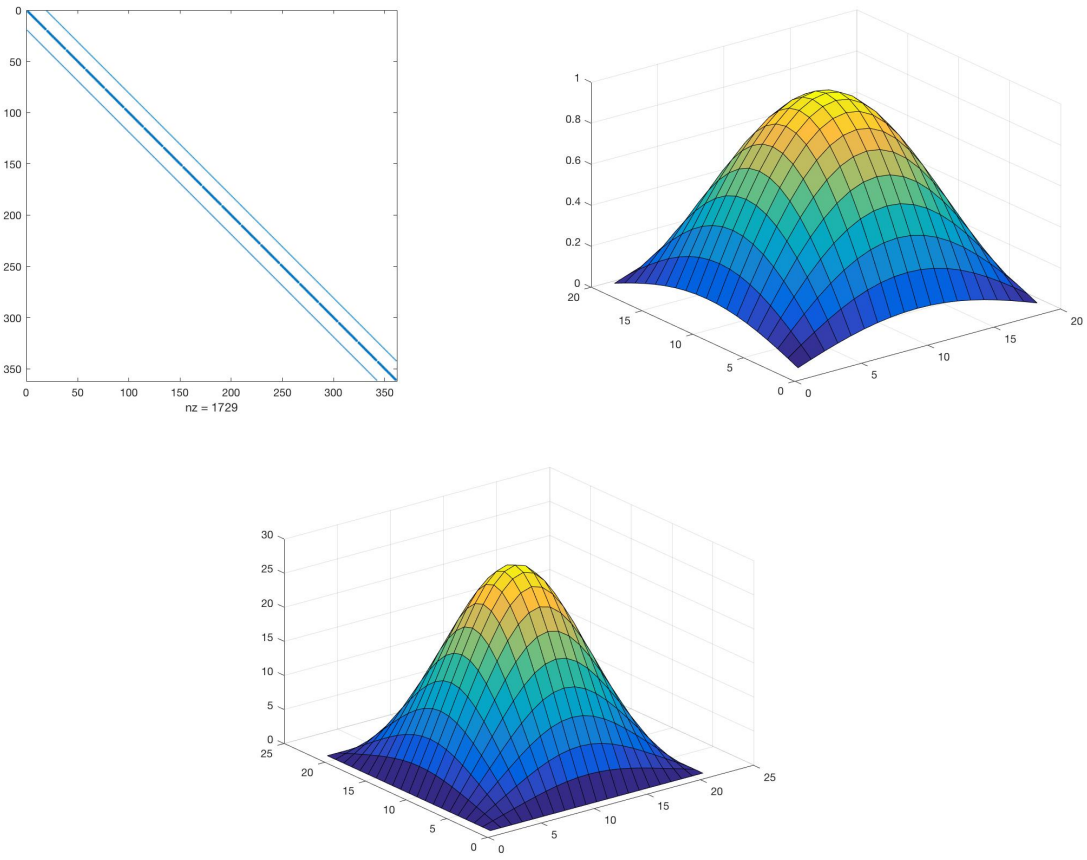

FIGURE 11.2. Using finite differences in two spatial dimensions. Left-top: the pattern of non-zero elements in the complete discretised Laplacian, matrix A in the code. Top-right: the initial condition, used to start the Newton method to solve the nonlinear algebraic equations. Bottom: the actual solution; note that it contains the boundary again, which was deleted in the computation of the solution.

```
A=spdiags(B,[-N2,-1,0,1,N2],N2^2,N2^2);

% have a look at the spatial position of the non-zero elements in A
figure(1)
spy(A)

% for the Dirichlet BCs
for k=1:N2-1
A(N2*k+1,N2*k)=0;
A(N2*k,N2*k+1)=0;
end
spy(A)

% The PDE in finite-difference form
f=@(u) (N-1)^2*A*u+u.^2
```

```
% we will use Newton's method to solve it, so require the derivative
df=@(u) (N-1)^2*A+2*diag(u)

% a starting point for the Newton solver, which kind of
% resembles the real solution
x=sin(linspace(0,pi,N))
u0=x'*x;

figure(2)
surf(u0);

u0(:,[1,end])=[];
u0([1,end],:)=[];
surf(u0)

% now actually solve the problem
u1=multinewton(f,df,50*u0(:))
% make sure you convert the column vector u1 to a matrix again
u2=reshape(u1,N2,N2);

% include the boundaries again!
u3=zeros(N,N);
u3(2:N-1,2:N-1)=u2
figure(3)
surf(u3)
```

Chapter A

Selected linear algebra topics

A.1 Matrix norms

The matrix/operator norm is given by

$$\|A\| = \sup_{x\neq 0} \frac{|Ax|}{|x|} = \sup_{|x|=1} \frac{|Ax|}{|x|} = \max_{|x|=1} \frac{|Ax|}{|x|}.$$

Properties:

(1) If $y = Ax$ then $|y| \leq \|A\|\,|x|$,
and if invertible $|x| \leq \|A^{-1}\|\,|y|$,
hence $\frac{1}{\|A^{-1}\|}\,|x| \leq |y| \leq \|A\|\,|x|$.

(2)
$$\|A^{-1}\| = \sup_{x\neq 0} \frac{|A^{-1}x|}{|x|} = \sup_{y\neq 0} \frac{|y|}{|Ay|} = \sup_{|y|=1} \frac{1}{|Ax|} = \frac{1}{\min_{|y|=1}|Ay|}.$$

(3) Different norms for $x = (x_1, \ldots, x_n) \in \mathbb{R}^n$:
$|x|_2 = \sqrt{\sum_{i=1}^n |x_i|^2}$,
$|x|_1 = \sum_{i=1}^n |x_i|$,
$|x|_\infty = \max_{1\leq i\leq n} |x_i|$,
lead to different matrix norms for $A = (a_{ij})$:
$\|A\|_2$ hard to compute,
$\|A\|_1 = \max_{1\leq j\leq n} \sum_{i=1}^n |a_{ij}|$,
$\|A\|_\infty = \max_{1\leq i\leq n} \sum_{j=1}^n |a_{ij}|$.

(4) $\|A\|_\infty = \max_{1\leq i\leq n} \sum_{j=1}^n |a_{ij}|$. Namely, write $y = Ax$, then $|y_i| \leq \sum_{j=1}^n |a_{ij}||x_j|$.
It follows that
$$|y|_\infty \leq \max_{1\leq i\leq n} \sum_{j=1}^n |a_{ij}||x_j| \leq |x|_\infty \max_{1\leq i\leq n} \sum_{j=1}^n |a_{ij}|,$$
and equality for $x_j = \operatorname{sign}(a_{i_0 j})$ when the maximum is attained for $i = i_0$.

(5) $\|A\|_1 = \max_{1\leq j\leq n} \sum_{i=1}^n |a_{ij}|$. Namely
$$\begin{aligned} |y|_1 &\leq \sum_{i=1}^n \sum_{j=1}^n |a_{ij}||x_j| = \sum_{j=1}^n \sum_{i=1}^n |a_{ij}||x_j| \\ &\leq \max_{1\leq j\leq n} \sum_{i=1}^n |a_{ij}| \sum_{j=1}^n |x_j| = |x|_1 \max_{1\leq j\leq n} \sum_{i=1}^n |a_{ij}| \end{aligned}$$
and equality for $x_j = \delta_{jj_0}$ when the maximum is attained for $j = j_0$.

(6) These are all norms on the space of matrices:
$\|A\| \geq 0$ and $\|A\| = 0$ if and only if $A = 0$;

and triangle inequality: $\|A+B\| \le \|A\| + \|B\|$
holds point-wise, i.e. $|(A+B)x| = |Ax+Bx| \le |Ax| + |Bx|$ for all x.

(7) Furthermore: $\|AB\| \le \|A\|\,\|B\|$.

A.2 Relative condition number

Let us look at the solutions of $Ax = b$ and of the perturbed problem $A(x+\delta x) = b + \delta b$. Recall

$$\|A\| = \max_{x\neq 0} \frac{|Ax|}{|x|} = \max_{|x|=1} |Ax|$$

and for invertible matrices

$$\|A^{-1}\| == \max_{x\neq 0} \frac{|A^{-1}x|}{|x|} = \max_{y\neq 0} \frac{|y|}{|Ay|} = \max_{|y|=1} \frac{1}{|Ay|} = \frac{1}{\min_{|y|=1}|Ay|}$$

so that $\min_{|x|=1}|Ax| = \frac{1}{\|A^{-1}\|}$. We find

$$\delta x = A^{-1}\delta b, \quad \text{hence} \quad |\delta x| \le \|A^{-1}\||\delta b,$$

and

$$|b| \le \|A\||x|, \quad \text{so that} \quad |x| \ge \|A\|^{-1}|b|.$$

All in all

$$\frac{|\delta x|}{|x|} \le \|A\|\|A^{-1}\|\frac{|\delta b|}{|b|} = K(A)\frac{|\delta b|}{|b|},$$

where the *relative* condition number is defined as

$$K(A) = \|A\|\|A^{-1}\|.$$

Properties of the relative condition number:

- $K(A) \ge 1$
- It depends on choice of norm. For the 2-norm: $K_2(A) = \frac{\sigma_1}{\sigma_n}$, where the σ_i are the singular values (Chapter 5). For the special case of positive definite symmetric matrices: $K_2(A) = \frac{\lambda_{\max}}{\lambda_{\min}}$.
- Define the "relative distance to being singular":

  $$\text{dist}(A) = \min\{\frac{\|\delta A\|}{\|A\|} : A + \delta A \text{ is singular}\}.$$

  Then this relative distance is the inverse of the relative condition number: $\text{dist}(A) = \frac{1}{K(A)}$. Here is the proof of this statement:

  (1) On the one hand: if $\|\delta A\| < \frac{1}{\|A^{-1}\|}$ then $A+\delta A$ is invertible: $|(A+\delta A)x| \ge |Ax| - |\delta Ax| \ge (\frac{1}{\|A^{-1}\|} - \|\delta A\|)|x|$; thus $A+\delta A$ is injective, hence invertible. We conclude that $\text{dist}(A) \ge \frac{1}{\|A\|\|A^{-1}\|} = \frac{1}{K(A)}$.

  (2) On the other hand, let $\tilde{x}$ with $|\tilde{x}| = 1$ be a minimiser of $\min|Ax|$, i.e. $|A\tilde{x}| = \frac{1}{\|A^{-1}\|}$.
  Define B through $Bx = -\langle \tilde{x}, x\rangle A\tilde{x}$, then $(A+B)\tilde{x} = 0$ and $|B(x)| \le \frac{1}{\|A^{-1}\|}|x|$, i.e. $\|B\| \le \frac{1}{\|A^{-1}\|}$ (in fact equal). Hence $\text{dist}(A) \le \frac{\|B\|}{\|A\|} \le \frac{1}{K(A)}$.
- calculating $K(A)$ is computationally expensive, but the precise value is not so important; a rough estimate suffices.

Here is an example in MATLAB.

```
load Amatrix.mat
A
det(A)
b1=[1;2;3]
b2=b1+0.01
x1=A\b1
x2=A\b2
```

```
A*x1
A*x2
A\b1
(A+0.01)\b1
```

Note that the final computation is for a problem where the matrix A is perturbed.

Finally, here is a geometric interpretation. The solution of

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

is the intersection of the lines

$$a_{11}x + a_{12}y = b_1 \qquad \text{and} \qquad a_{21}x + a_{22}y = b_2.$$

If these lines are almost parallel, then the residue of an almost solution is small. Picture of two lines that cross at a small angle. Points that almost lie on both lines can be far away from the intersection.

A.3 Tridiagonal systems

Any tridiagonal matrix

$$A = \begin{pmatrix} a_1 & c_1 & & & \\ b_2 & a_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & b_{n-1} & a_{n-1} & c_{n-1} \\ & & & b_n & a_n \end{pmatrix}$$

may be decomposed as $A = LU$ with

$$L = \begin{pmatrix} 1 & & & \\ \beta_2 & 1 & & \\ & \ddots & \ddots & \\ & & \beta_n & 1 \end{pmatrix} \qquad \text{en} \qquad U = \begin{pmatrix} \alpha_1 & c_1 & & & \\ & \alpha_2 & c_2 & & \\ & & \ddots & \ddots & \\ & & & \alpha_{n-1} & c_{n-1} \\ & & & & \alpha_n \end{pmatrix}$$

Working this out gives $\alpha_1 = a_1$ and

$$\alpha_1\beta_2 = \beta_2, \;\; \alpha_2 + \beta_2 c_1 = a_2, \;\; c_2 = c_2,$$

and more generally

$$\alpha_{i-1}\beta_i = b_i, \;\; \alpha_i + \beta_i c_{i-1} = a_i, \;\; c_i = c_i.$$

Hence you can calculate them one by one via

$$\beta_i = \frac{b_i}{\alpha_{i-1}}, \quad \text{en} \quad \alpha_i = a_i - \beta_i c_{i-1} \qquad \text{for } i = 2, 3, \ldots n.$$

To solve $Ax = z$, we subsequently solve first $Ly = z$ and then $Ux = y$:

$$y_1 = z_1, \quad y_i = z_i - \beta_i y_{i-1} \qquad \text{for } i = 2, \ldots, n,$$

and

$$x_n = \frac{y_n}{\alpha_n}, \quad x_i = \frac{y_i - c_i x_{i+1}}{\alpha_i} \qquad \text{for } i = n-1, \ldots, 1.$$

This takes only of the order of n operations!

A.4 Fibonacci sequences and the golden mean

The Fibonacci sequence is given by $f_0 = 1$, $f_1 = 1$, and the recursively

$$f_{n+1} = f_n + f_{n-1}.$$

Our focus here is to explain, using a bit of linear algebra, why for large n the ratio of successive Fibonacci numbers approaches the golden mean (divine proportion) $1 : \phi = \phi : (1 + \phi)$, or $\phi^2 - \phi - 1 = 0$, or explicitly

$$\phi = \frac{1 + \sqrt{5}}{2}.$$

The iteration can be rewritten as

$$\begin{pmatrix} f_{n+1} \\ f_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix}$$

Hence if you take $u_n = (f_n, f_{n-1})^T$ with $u_1 = (1, 1)^T$ then

$$u_{n+1} = Au_n = A^2 u_{n-1} = \cdots = A^n u_1.$$

The matrix $A^* = A$ is self-adjoint (hermitian), or simply $A^T = A$ since it is real-valued, hence A can be diagonalised

$$A = V \Lambda V^{-1},$$

with Λ the diagonal matrix of eigenvalues

$$\Lambda = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix} = \begin{pmatrix} \frac{1+\sqrt{5}}{2} & 0 \\ 0 & \frac{1+\sqrt{5}}{2} \end{pmatrix} = \begin{pmatrix} \phi & 0 \\ 0 & 1 - \phi \end{pmatrix}$$

and V the matrix with the eigenvectors v_1 and v_2 as columns:

$$V = (v_1, v_2) = \begin{pmatrix} \frac{\phi}{\sqrt{1+\phi^2}} & \frac{1}{\sqrt{1+\phi^2}} \\ \frac{1}{\sqrt{1+\phi^2}} & \frac{-\phi}{\sqrt{1+\phi^2}} \end{pmatrix}$$

with $V^{-1} = V^*$ unitary, or real symmetric in our real case: $V^{-1} = V^T$. Hence

$$\begin{aligned} A^n = (V \Lambda V^{-1})^n = V \Lambda V^{-1} V \Lambda V^{-1} \cdots V \Lambda V^{-1} = V \Lambda^n V^{-1} \\ = V \begin{pmatrix} \lambda_1^n & 0 \\ 0 & \lambda_2^n \end{pmatrix} V^{-1} = (v_1, v_2) \begin{pmatrix} \lambda_1^n & 0 \\ 0 & \lambda_2^n \end{pmatrix} \begin{pmatrix} v_1^T \\ v_2^T \end{pmatrix} \end{aligned}$$

so

$$u_{n+1} = A^n u_1 = v_1 \lambda_1^n \langle v_1, u_1 \rangle + v_2 \lambda_2^n \langle v_2, u_1 \rangle = \lambda_1^n \left[\langle v_1, u_1 \rangle v_1 + \left(\frac{\lambda_2}{\lambda_1} \right)^n \langle v_2, u_1 \rangle v_2 \right] \approx \lambda_1^n \langle v_1, u_1 \rangle v_1.$$

Therefore the later iterations are dominated by the largest eigenvalue (in absolute value) and the corresponding eigenvector, whose two elements have the golden ratio.

A.5 The finite difference matrix for the second derivative

Consider the $n \times n$ matrix

$$A = \begin{pmatrix} -2 & 1 & 0 & \cdots & 0 & 0 \\ 1 & -2 & 1 & & & 0 \\ 0 & 1 & -2 & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & 1 & 0 \\ 0 & & \cdots & 1 & -2 & 1 \\ 0 & 0 & \cdots & 0 & 1 & -2 \end{pmatrix}$$

which appears when one applies centered finite differences to the problem $u'' = f$ with Dirichlet boundary conditions (using $n + 2$ discretisation points).

The problem $Ax = b$ (or $Au = h^2 f$) has a unique solution since A is invertible: writing out $x^T Ax$ with $x = (x_1, \ldots, x_n)$ gives

$$\begin{aligned} x^T Ax &= -2x_1^2 + 2x_1x_2 - 2x_2^2 + 2x_2x_3 - \cdots + 2x_{n-1}x_n - x_n^2 \\ &= -x_1^2 - (x_1 - x_2)^2 - (x_2 - x_3)^2 - \cdots - (x_{n-1} - x_n)^2 - x_n^2, \end{aligned}$$

hence A is negative-definite, so A is invertible.

Let us now look at the relative condition number of A. The eigenvalues of A are

$$\lambda_j = -2(1 - \cos(j\theta)) = -4\sin^2\left(\tfrac{j\theta}{2}\right) \qquad \text{for } j = 1, \ldots n, \qquad \text{with } \theta = \tfrac{\pi}{n+1},$$

with corresponding eigenvectors

$$v_j = (\sin(j\theta), \sin(2j\theta), \ldots, \sin nj\theta)^T.$$

Namely, writing out $Av_j = \lambda_j v_j$ gives

$$\begin{aligned} -2\sin(j\theta) + \sin(2j\theta) &= -2(1 - \cos(j\theta))\sin(j\theta), \\ \sin((k-1)j\theta) - 2\sin(kj\theta) + \sin((k+1)j\theta) &= -2(1 - \cos(j\theta))\sin(kj\theta) \qquad \text{for } k = 2, \ldots n-1 \\ \sin((n-1)j\theta) - 2\sin(nj\theta) &= -2(1 - \cos(j\theta))\sin(nj\theta). \end{aligned}$$

The first line is an identity and the other lines as well since

$$\begin{aligned} \sin((k-1)j\theta) &= \sin(kj\theta)\cos(j\theta) - \cos(kj\theta)\sin(j\theta); \\ \sin((k+1)j\theta) &= \sin(kj\theta)\cos(j\theta) + \cos(kj\theta)\sin(j\theta), \end{aligned}$$

while the last one you get by taking $k = n$ and using that $\sin((n+1)j\theta) = \sin(j\pi) = 0$.

This implies that the problem is badly conditioned:

$$K(A) = \frac{1 - \cos(\frac{n}{n+1}\pi)}{1 - \cos(\frac{1}{n+1}\pi)} \approx \frac{2}{\frac{\pi^2}{2(n+1)^2}} = O(n^2) = O(h^{-2}).$$

Next, we compare the eigenvalues of A to the ones of the Dirichlet problem $u'' = \bar{\lambda}_j u$ with $u(0) = 0$, $u(1) = 0$. It is well known (and can be checked easily) that the eigenvalues are $\bar{\lambda}_j = -\pi^2 j^2$. Let $h = \frac{1}{n+1}$ (i.e. $n+2$ grid points) and set $\theta = \frac{\pi}{n+1}$. Then

$$\lim_{n\to\infty} \frac{-2(1 - \cos(\pi j/(n+1)))}{h^2} = \lim_{n\to\infty} \frac{-\frac{\pi^2 j^2}{(n+1)^2}}{\frac{1}{(n+1)^2}} = -\pi^2 j^2 = \bar{\lambda}_j,$$

hence the eigenvalues of A indeed converge suitably to the eigenvalues of the elliptic problem as the grid size tends to zero.

There is a more elegant way to analyse the eigenvalues of the matrix A. The Dirichlet and Neumann problems can also be analysed by extension to twice as large domain and restricting (out of the periodic solutions) to solutions that satisfy the correct boundary conditions.

We first solve the periodic problem (reading periodically)

$$u_{k-1} - 2u_k + u_{k+1} = \lambda u_k \qquad \text{for } k = 1, \ldots, N.$$

Now define the Fourier transform

$$\hat{u}_m = \sum_{k=1}^{N} e^{-2\pi ikm/N} u_k \qquad \text{with inverse} \qquad u_k = \frac{1}{N}\sum_{m=1}^{N} e^{2\pi ikm/N}\hat{u}_m.$$

By Fourier transforming the equation we obtain

$$\left[e^{-2\pi im/N} - 2 + e^{2\pi im/N}\right]\hat{u}_m = \lambda \hat{u}_m \qquad \text{for } m = 1, \ldots, N.$$

The solutions are straightforward:

$$\lambda = \lambda_j = e^{-2\pi ij/N} - 2 + e^{2\pi ij/N} = -2 + 2\cos(2\pi j/N) \qquad \text{for } j = 1, \ldots, N,$$

with corresponding $\hat{u}^j_m = \delta_{jm}$ (Kronecker delta). This transforms back into the eigenvector

$$u^j_k = \frac{1}{N} e^{2\pi i kj/N}.$$

Since each eigenvalues appear twice: $\lambda_j = \lambda_{N-j}$, one obtains real-valued eigenvectors

$$u^j_k = \cos(2\pi kj/N) \qquad \text{and} \qquad v^j_k = \sin(2\pi kj/N)$$

corresponding to eigenvector λ_j.

To recover the result for Dirichlet boundary conditions, note that $v^j_0 = v^j_{N/2} = 0$ and $N = 2n + 2$ since we have extended the domain.

Finally, let us consider in this context the heat equation

$$u_t = u_{xx}$$

with Dirichlet boundary conditions. Discretising with forward Euler in time (just to analyse the method, not as a suggestion for best practice) we obtain for $u^n_i \approx u(n\Delta t, i\Delta x)$ the equations

$$\frac{u^{n+1} - u^n}{\Delta t} = \frac{1}{(\Delta x)^2} A u_n$$

with A the centered second derivative finite difference matrix defined at the start of this section. This leads to the explicit time integration scheme

$$u_{n+1} = (I + \frac{\Delta t}{(\Delta x)^2} A) u_n.$$

For this to converge to 0, as we know it should, the eigenvalues of $I + \frac{\Delta t}{(\Delta x)^2} A$ must have absolute values smaller than 1. The eigenvalues are

$$\lambda_j = 1 + \frac{\Delta t}{(\Delta x)^2} \cdot (-2(1 - \cos(j\theta)) = 1 - \frac{2\Delta t}{(\Delta x)^2}(1 - \cos(j\pi/(N-1))).$$

Hence $|1 - \frac{4\Delta t}{(\Delta x)^2}| < 1$ leads to the condition

$$\Delta t < \frac{\Delta x)^2}{2}$$

to have any form of stability for this method. This shows that the time steps (in forward Euler) must be very small (compared to the grid size in the spatial variable), hence the problem is "stiff". This is a good reason to consider more advanced methods such as finite elements.

An illustration in MATLAB with $\mu = \frac{\Delta t}{(\Delta x)^2}$

```
B=ones(20,1)*[1,-2,1]
A=spdiags(B,[-1,0,1],20,20)
spy(A)
mu=0.7
x=ones(20,1)
x=(eye(20)+mu*A)*x
x=(eye(20)+mu*A)*x
for k=1:40 x=(eye(20)+mu*A)*x; end
x
plot(x)
mu=0.3
x=ones(20,1)
x=(eye(20)+mu*A)*x
x=(eye(20)+mu*A)*x
for k=1:40 x=(eye(20)+mu*A)*x; end
x
plot(x)
```