Introduction to numeric and symbolic computing

Antti Rasila, Susanna Liesipohja, Juha Kuortti

May 16, 2011

Contents

Chapter 1

Introduction to MATLAB

MATLAB is an interactive computing environment for doing numerical computations with vectors and matrices. It is suitable for a variety of different tasks involving scientific and technical computations. MATLAB was created by Cleve Moler in the 1970's as a small program for teaching matrix calculations. It was created by using Fortran. LINPACK and EISPACK library routines were used internally for computations. MATLAB was initially a shareware program, and it quickly spread to other universities. In 1984, Cleve Moler, Jack Little and Steve Bangert founded MathWorks and commersialized MATLAB. The current versions are commercial products, written in C. Several extensions have been added to the original MATLAB.

1.1 Basics

As mentioned above, MATLAB can be used interactively as a sort of calculator. In this case, the commands will be written directly into the MATLAB prompt `>>`. One can exit the prompt by writing `quit` or `exit`.
The program can also be written into a file ending with `.m`, for example `myfile.m`, and run from the prompt by typing the filename, in this case `myfile`.
The MATLAB commands are organized into different topics. Typing `help` will give a list of all the topics and typing `help [topic]` will give a list of all the commands grouped under that topic. Typing `help [command]` will give a short description of the specific command.
The MATLAB commands issued and the results obtained can be saved using

the `diary`-command. For example,

```
>> diary test.dry
>> a=1; b=0;
>> a+b
ans =
     1
>> diary off
>> type test.dry
a=1; b=0;
a+b
ans =
     1
diary off
```

Calculations are done in floating-point precision (approximately 16 digits in the decimal system). The output can be changed using the `format` command, but it will not change the precision of the calculations. The default output precision is `short`, which is of 5-digit precision. For example

```
>> pi
ans =
    3.1416
>> format long
>> pi
ans =
   3.141592653589793
```

A variable is given a value with the `=` operator. The most commonly used variable is `ans`, it always contains the result of the previous command. More precisely, if a command does not assign a value to a named variable then it is stored to the variable `ans`.
Some names are reserved for certain constants, such as `pi` (for π), and both `i` and `j` represent the imaginary unit. Other reserved names are, among others, `realmax`, `realmin`, `eps`, `Inf` and `NaN`. Different values can be assigned to these constants, but they will revert back to the default values after re-starting the program or using the command `clear`.
A short example:

```
>> 1/0
ans =
   Inf
```

```
>> 0/0
ans =
   NaN

>> NaN=5
NaN =
     5

>> clear
>> NaN
ans =
   NaN
```

1.1.1 Output

A simple way to output data is to display a variable. This can be accomplished by giving its name (without a semicolon) in interactive mode. Alternatively you can use the `disp` function, which shows values without the variable name, as in:

```
x=42;
>> disp(x)
42
```

For a fancier output, MATLAB has various functions for creating strings from numbers, formatting data, etc... One such is `fprintf`, which can also be used for printing into a file. The syntax for this is:
`fprintf([fileId],[format],[input values])`

If `[fileId]` is omitted, the function will print directly onto the screen. `[fileId]` refers to the file identifier returned when opening the file for writing with `fopen`. For example, `fileId=fopen('myfile.txt','w')` would open `myfile.txt` for writing. The command `fclose(fileId)` would close the file. `[format]` is a string in single quotation marks that describes the format of the output fields. It can include combinations of the following:

- A percent sign followed by a conversion character, such as `%s` for strings and `%d` for an integer. Floating-point numbers can be printed with `%f` for fixed notation and `%e` for exponential notation.

- Field width and precision. For example, `%6.2f` would refer to a floating-point number of field width 6 and precision 2.
- Flags, such as `-` for left-justified and `+` for printing a sign character (+ or −). For example, `%+-d` would print a signed integer justified to the left.
- Literal text to print.
- Escape characters, such as `\n` for a new line `\t` for tab and `%%` for the percent sign.

Below are some examples on the use of `fprintf`.

```
a=5; s='Hello world';

>> fprintf('%d is an integer and %s is a string\n',a,s)';
5 is an integer and Hello world is a string

>> fprintf('Now %+d is a signed integer\n',a)
Now +5 is a signed integer

b=1.23456789; c=0.0015;

>> fprintf('Printing with precision 2: %.2f\n',b)
Printing with precision 2: 1.23

>> fprintf('\t or with width 20: %20f\n',b)
    or with width 20:            1.234568

>> fprintf('Printing as %f and as %e\n',c,c)
Printing as 0.001500 and as 1.500000e-03
```

For printing into a file, one can do the following:

```
>> fid=fopen('output.txt','w');
>> fprintf(fid, '%s\n',s);
>> fclose(fid);
```

Now the sentence `Hello world` (and a row-change) can be found in the file `output.txt`.

1.2 Vectors and matrices

In MATLAB, the basic data structure is matrix. The most efficient way of programming MATLAB is to treat every variable as a vector or a matrix. Assigning vector values can be done in the followong ways:

```
>>x = 1:1:4; %  expression a:h:b produces a vector with
             % numbers from a to b with interval h. If
             % no h is provided, 1 is assumed, eg. 1:10

>>y = [0 1 0 1]; % Vector values can be
                 % given individually also.
```

Vector dimensions have to be taken into account when performing arithmetics. The product `x*y` is not defined for two n-vectors, but the pairwise operations `x.*y` and `x+y` are:

```
>>x.*y
ans =
    0 2 0 4
>>x.+y
ans =
    1 3 3 5
```

In the case of vectors, the product is defined as if they are $n \times 1$-matrices: hence we need to transform one vector from a row vector to a column vector. We do this with the transpose operator `'`.

```
>>x
x =
    1 2 3 4
>>x'
x =
    1
    2
    3
    4
>>x'*y
ans =
     0     1     0     1
     0     2     0     2
     0     3     0     3
     0     4     0     4
```

```
>>x*y'
ans =
    6
```

If your vector (or matrix) contains complex numbers, you need to take into account that the transpose operator will also change a complex number to its complement, i.e., if $z = a + bi$ then $\bar{z} = a - bi$.

```
>> xi=[2+i 2 -i 4];
>> xi'
ans =
   2.0000 - 1.0000i
   2.0000
        0 + 1.0000i
   4.0000
```

The power operator is ^, and again, it only works elementwise:

```
>>x.^y
ans =
     1 2 1 4
```

Elementary functions are also available for vectors:

```
>> sin(x)
ans =
    0.8415    0.9093    0.1411   -0.7568
>>exp(y)
ans =
    1.0000    2.7183    1.0000    2.7183
```

You can define a matrix just as you defined a vector: to indicate a row change, use ;

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


     1     2
     3     4
>>b= [5; 6]; % b must be a row vector

% You can now obtain inverse of A and multiply
% b with it
>>iA = inv(A)
```

```
iA =
   -2.0000    1.0000
    1.5000   -0.5000
>>x = iA*b
x =
   -4.0000
    4.5000

% It is generally faster and easier to use MATLABs
% built-in linear solver operator \
>> x = A\b
x =
   -4.0000
    4.5000
```

Some useful matrix commands are also: `eye` (produces an identity matrix), `zeros` (produces a matrix of all zeros) and `ones` (produces a matrix of all ones). It is also possible to select specific elements, rows och columns from a matrix. The command for this is `A[i,j]`, where `A` represents a matrix, `i` the row of that matrix and `j` the column. Here, `i` and `j` can be scalars or vectors.

```
% We create a 3x3-matrix of all ones
>> A=ones(3)
A =
     1     1     1
     1     1     1
     1     1     1
% To pick a specific element from matrix A, use A(i,j).
% To pick a whole row (or column), replace j (or i)
% with :
>> A(1,:)
ans =
     1     1     1

>> A(2,:)=[2, 3, 4]
A =
     1     1     1
     2     3     4
     1     1     1
```

```
>> A(3,2)=42
A =
     1     1     1
     2     3     4
     1    42     1
```

1.2.1 Random numbers

Random numbers can be generated by using the commands `rand` and `randn`. The command `rand(m,n)` will produce an $m \times n$-matrix of uniformly distributed random numbers on $(0, 1)$ and `randn(m,n)` will produce a matrix of normally distributed random numbers with mean 0 and standard deviation 1.

```
dist = zeros(6,1);
for j=0:99
    k = round(5*rand(1)+1);
    dist(k) = dist(k)+1;
end
disp(dist)
```

Output:

```
1: 14
2: 15
3: 13
4: 18
5: 18
6: 22
```

1.3 Branch and loop structures

The branch and loop structures available in matlab are: `for`, `while`, `if` and `switch`. The main principle is that you should only use these as a last resort. If possible, you should use efficient vector operations instead.
The syntax of the `for` statement is:

```
for [variable]=[vector]
...
end
```

If one wants to repeat the loop k times, it is handy to use the vector statement `1:k`, which produces a list of numbers $1, 2, \ldots, k$.

```
# Example: 2nd powers of positive integers
for x = 1:4
    xx = x*x
    fprintf('%d * %d = %d',x,xx)
end
```

Output:

```
1*1 = 1
2*2 = 4
3*3 = 9
4*4 = 16
```

The syntax of the `if` statement in MATLAB is:

```
if [condition]
...
elseif [condition]
...
else
...
end
```

The `elseif` and `else` branches may be omitted. The commands in the `if` branch are executed if the condition is satisfied, if not then the conditions in the `elseif` branches are evaluated. If none of the conditions given is satisfied, the commands in the `else` branch are executed.
The most common conditions used are the comparison operations `<`, `<=`, `==`, `~=`, `>=` and `>`. Note that for equality, the expression `==` is used in order to avoid confusion with the value assignment operator `=`[1]. The expression `~=` is used for inequality.
A `while` statement is used when one wishes to repeat the loop until some condition is no longer satisfied. This structure is very useful when reading input from a file or from the user.
The syntax of the `while` statement is:

```
while [condition]
...
end
```

[1]This is significant, as in e.g. the C language, the condition `if(x=1)...` is always true.

To avoid an infinite loop, inside the loop there must naturally be something to invalidate the condition when the desired number of loops is reached.

```
x = 5;
guess = 0;
while guess ~= x
    guess = input('Guess a number:');
    if (abs(guess - x)>10)
        disp('Your guess is very wrong')
    end
end
```

Output:

```
Guess a number:6
Guess a number:100
Your guess is very wrong
Guess a number:5
```

The syntax of the `switch` statement is:

```
switch [switch expression]
case [case expression 1]
...
case [case expression 2]
...
...
otherwise
...
 end
```

The statements associated with a certain case will be executed when the switch expression equals the case expression in question.

```
% Color evaluation
color='aqua';
switch color
    case {'red','pink','rose'} % multiple case expressions
        disp('The color is red.')
    case {'blue','turquoise','aqua'}
        disp('The color is blue.')
    case 'yellow'
        disp('The color is yellow.')
    otherwise
```

```
        disp('Unknown color.')
end
```

Remark. One should avoid comparing non-integers with the `==` operator. For example, `pi==3.14159265...` is actually false. The MATLAB `pi` is only calculated to a specific length, and thus, does not actually equal π. The following program will demonstrate this fact:

```
% Desired accuracy of approximation
tol=10^-4;
mypi=1;
while mypi~=0
    mypi=input('Guess the value of pi (0 exits):');
    if mypi==pi
        disp('Comparison to MATLAB pi is true')
    else
        disp('Comparison to MATLAB pi is false')
    end
    if (abs(pi-mypi)<tol)
        disp('Close to pi!')
    elseif (abs(pi-mypi)>1)
        disp('Far from pi!')
    else
        disp('Not close enough to pi!')
    end
end
```

cd Output:

```
Guess the value of pi (0 exits):3.141592653589793238
Comparison to MATLAB pi is true
Close to pi!
Guess the value of pi (0 exits):3.141592653589793258
Comparison to MATLAB pi is true
Close to pi!
Guess the value of pi (0 exits):3.14
Comparison to MATLAB pi is false
Not close enough to pi!
```

The first guess is an accurate approximation of π, but the second one is not (the second-to-last digit is wrong). However, the comparison to MATLAB `pi` is correct in both cases.

1.4 Defining functions

A function can be defined with the `function` statement. The syntax of this statement is:

```
function [output]=[function name]([input])
...
```

This function should be saved in an m-file and the name of the file must be the *same* as the function name. For example, the function below should be saved as `solve2.m`.

```
function x = solve2(a,b,c)
    D= b^2 - 4*a*c;
% Floating point number should not be directly
% compared to zero
    if(abs(a)<1e-6)
        disp('Error')
        return
    else if(abs(D)<1e-6)
            x = -b/2*a;
            return
    else
        x(1) = -b + sqrt(D)/2*a;
        x(2) = -b - sqrt(D)/2*a;
        end
    end
```

Output:

```
>> solve2(1,0,0)
ans =
     0
>> solve2(1,0,1)
ans =
        0 + 1.0000i         0 - 1.0000i
>> solve2(1,0,-1)
ans =
     1    -1
```

The above function `solve2` solves the roots of a given second order equation. The input parameters given for the function are three numbers a, b and c, corresponding to the coefficients of the equation to be solved. In the first

example, the equation $x^2 = 0$ is solved (one root at 0); in the second case, the equation is $x^2+1 = 0$ (only imaginary roots) and in the third case $x^2-1 = 0$ (two roots ± 1).
For simpler functions, it may be easier to define the functions "directly" into the program. This can be done with the `inline` command or, more recently, the function handle @.

```
>> f=inline('exp(x.^2)','x')
f =
     Inline function:
     f(x) = exp(x.^2)
>> g=@(x) x.^2
g =
    @(x)x.^2
```

1.5 Polynomials

In MATLAB, a polynomial is represented by a vector which consists of its coefficients. To create a polynomial one can simply enter each coefficient of the polynomial into the vector in descending order. For instance, consider the following polynomial:

$$p(x) = 2x^4 - x^2 + 5x + 17$$

To give this in MATLAB, just write the vector

```
>> p=[2 0 -1 5 17];
```

One may find the value of a polynomial using the `polyval` function. For example, to find the value of the above polynomial at $x = 2$,

```
>> polyval(p,2)
ans =
    55
```

The roots of a polynomial can be obtained with `roots([your polynomial])`. For example, the roots of the polynomial above are

```
>> r=roots(p)
r =
   1.2663 + 1.3591i
   1.2663 - 1.3591i
   -1.2663 + 0.9273i
   -1.2663 - 0.9273i
```

If one knows the roots already, the coefficients can be found using the inverse function `poly`. Two polynomials can be multiplied by using `conv([poly1],[poly2])` and dividing can be done in a similar way with the `deconv` function.

1.6 Plotting and drawing

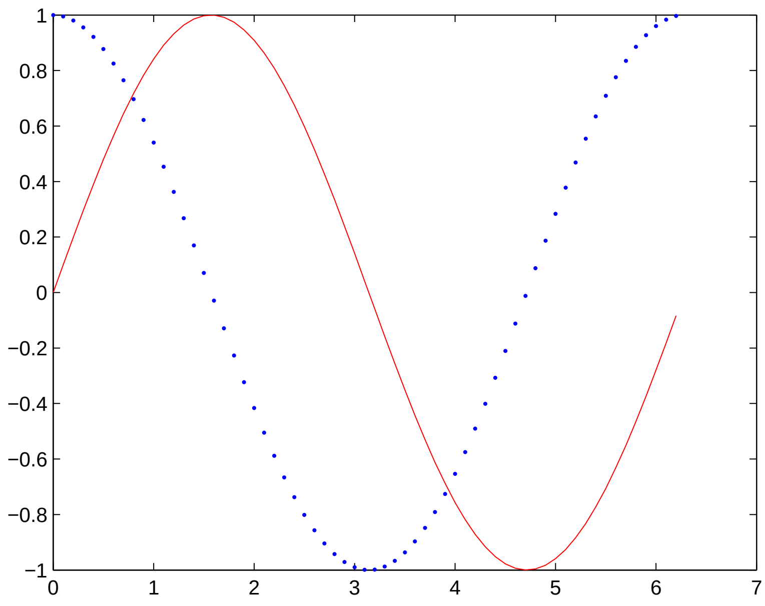
Curves can be drawn with the command `plot`. For example, to plot a sine curve, one can do the following:

```
>> x=0:.1:2*pi;
>> plot(x,sin(x))
```

It is possible to plot several curves at once. The appearance of the curves can be changed. For example, the command

```
>> plot(x,sin(x),'r',x,cos(x),'.b')
```

plots the sine curve in red and the cosine curve as blue dots.



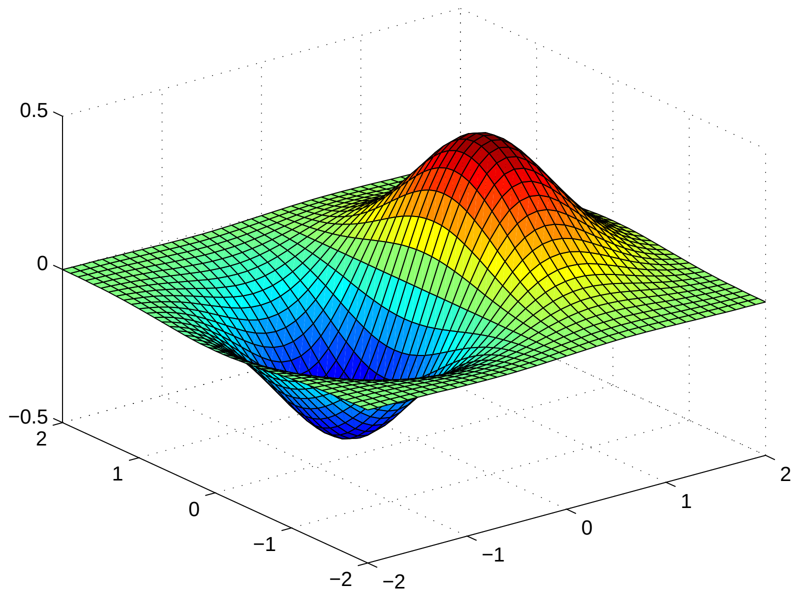
One may label the axes with the commands `xlabel('[labelname]')`, for the x-axis, and `ylabel('[labelname]')`, for the y-axis. A title can be added to the graph with `title('[title]')`. Curves can be labeled with the command `legend([curve1],[curve2],[curve3],[...  etc.])`.

1.6.1 Plotting 3D graphics

Spatial curves given in parameters can easily be plotted with the function `plot3` simply by adding z-coordinates.
Surface plotting can be done with the function `surf`. But first, one should generate the appropriate `X` and `Y` arrays using the function `meshgrid`.
In the example below, where we are plotting the function $f(x,y) = xe^{-x^2-y^2}$, `X` and `Y` represent the ”plane” and `Z` represents the ”height”.

```
>> x=-2:.1:2; y=x;
>> [X,Y]=meshgrid(x,y);
>> Z=X.*exp(-X.^2-Y.^2);
>> surf(X,Y,Z)
```

Output:



1.7 Useful links

Chapter 2

Linear algebra

2.1 Linear equations

An equation with variables $x_1 \dots x_n$ that can be written in the form

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = b, \qquad n \geq 1$$

is called *linear equation.* The coefficients $a_1 \dots a_n$ and b can be real or complex numbers.
A system of linear equations is a collection of one or more linear equations involving the same variables. Using matrix algebra, the linear system

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2, \\ \vdots \qquad\quad \vdots \qquad\quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m, \end{cases}$$

can be written in the form

$$\mathbf{Ax} = \mathbf{b}; \mathbf{A} = \begin{bmatrix} a_{11} & \dots & a_{1n} \\ a_{21} & \dots & a_{2n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ v_n \end{bmatrix}, \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}.$$

The matrix $\mathbf{A}$ is a $m \times n$ matrix, the vector $\mathbf{x}$ a vector with n components
Let $\mathbf{A} \in \mathbb{C}^{n \times n}$. A is said to be *invertible*, if there exists such $\mathbf{B} \in \mathbb{C}^{n \times n}$, that $\mathbf{AB} = \mathbf{I}$, where $\mathbf{I} \in \mathbb{C}^{n \times n}$ is the identity matrix. Then the matrix $\mathbf{B}$ is

called the inverse of matrix $\mathbf{A}$, and is denoted $\mathbf{A}^{-1}$. If $\mathbf{A}^{-1}$ exists, $\mathbf{A}$ is called invertible. Using this definition, we get following theorem.

Theorem 2.1. *The linear system of equation* $\mathbf{Ax} = \mathbf{b}, \mathbf{A} \in \mathbb{C}^{n \times n}, \mathbf{x} \in \mathbb{C}^n, \mathbf{b} \in \mathbb{C}^n$ *has a single solution only if* $\mathbf{A}$ *is invertible. The solution is* $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

Generally, system of linear equations can have an exact solution only if it has exactly as many linearily independent equations as it has unknowns. In this situation the system of linear equations

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2, \\ \vdots \qquad\quad \vdots \qquad\quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots + a_{nn}x_n = b_n, \end{cases}$$

translates into a $n \times n$ matrix and n vectors. However, number of equations and unknowns do not always coincide. In this case we get a system

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n = b_1, \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n = b_2, \\ \vdots \qquad\quad \vdots \qquad\quad \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n = b_m, \end{cases} .$$

This system can be written as a matrix equation

$$\mathbf{Ax} = \mathbf{b},$$

where $\mathbf{A}$ is a $m \times n$ matrix, x an n-vector, and b a m-vector.
If $m < n$, that is, if there are fewer equations than there are unknowns, system is called *underdetermined.* Solving an underdetermined system of equations will not usually produce an exact solution, but the solution will have degrees of freedom depending on the coefficient matrix: the number of which is determined by how many unknowns remain in the solution vector x. The solution can be interpreted as a space where the objects defined by the equations intersect.

Example 2.2. Solve an underdetermined system of equations

$$\mathbf{Ax} = \mathbf{b} = \begin{bmatrix} 1 & 3 & 3 & 2 \\ 2 & 6 & 9 & 5 \\ -1 & -3 & 3 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 1 \\ 5 \\ 5 \end{bmatrix}.$$

Using elementary row operations we get the solution:

$$\mathbf{x} = \begin{bmatrix} -2 - 3x_2 - x_4 \\ x_2 \\ 1 - \frac{x_4}{3} \\ x_4 \end{bmatrix}.$$

The vector $\mathbf{x}$ is a solution for the equation $\mathbf{Ax} = \mathbf{b}$ with arbitrary values of x_1 and x_4, thus giving an infinitely many solutions.

If in system **??** $m > n$, the system is called overdetermined; that is, there are more equations in the system than there are unknowns. Computing an exact solution to a overdetermined system may be possible. However, the more there are constraints (equations), the less likely it is that they all hold for a specific point. Thus solving overdetermined systems usually includes searching a best possible solution: a solution that does not necessarily hold for all the equations in the system, or any of them, but it *almost* holds for all of them. This is usually achieved by linear least squares, which will be discussed in depth in later chapters.
Later we will introduce some methods for finding the inverse of a matrix in MATLAB.

2.2 Matrices and vectors in MATLAB

The basic data type in MATLAB is a real valued matrix, and default assumption for every operation is that the operands are matrices. Some operations are, however defined also elementwise, so as to make certain operations easy and efficient, and it requires a certain amount of alertness to avoid any obvious pitfalls.
The basic multiplication, denoted by $*$, is matrix multiplication. It is defined for matrices $\mathbf{A}$ and $\mathbf{B}$, where $\mathbf{A} \in \mathbb{C}^{m \times n}$ and $\mathbf{B} \in \mathbb{C}^{n \times k}$. It also works on a scalar multiplication, that is $c\mathbf{A}$ is a legitimate operation, for $c \in \mathbb{C}$. If the

matrix dimensions do not match, MATLAB will produce an error. One can obtain an elementwise product with operator `.*`. The elementwise operator will produce the Hadamard product of two matrices of same dimensions.

```
% Define two square matrices and two
% non square matrices
>> A = [3 2 3; 3 4 3; 4 5 1]
A =
     3     2     3
     3     4     3
     4     5     1
>> B = [1 2 4; 1 4 6; 1 7 7];
>> D = [1 2; 4 3 ; 7 6]
D =
     1     2
     4     3
     7     6
>> K=[3 4 5; 5 6 7]
K =
     3     4     5
     5     6     7
% Multiplication of two square
% matrices works
>> A*B
ans =
     8    35    45
    10    43    57
    10    35    53
% Hadamard product of two square
% matrices
>> A.*B
ans =
     3     4    12
     3    16    18
     4    35     7
% Product of 3x2 and 2x3 matrices
>> D*K
ans =

    13    16    19
    27    34    41
    51    64    77
% Elementwise product doesn't work
>> D.*K
??? Error using ==> times
Matrix dimensions must agree.
% Multiply B by four
>> 4*B
ans =

     4     8    16
     4    16    24
     4    28    28
```

Same applies to the power operator: the operator ^ literally means that the first operand is multiplied by itself as many times as the second operand orders. If the first operand not a square matrix, the operation is not defined. Thus the the power operator should be used only as an elementwise operation: `.^`

```
% Examples of the power operator:
% First on real number
>> 3^5
ans =
   243
% then on a square matrix
% Note that this is a defined
% operation because A*A is a
% defined operation.
>> A^2
ans =
    27    29    18
    33    37    24
    31    33    28
% We try then the elementwise
% power operator. Notice the
% difference with the regular
% power operator.
>> A.^2
ans =
     9     4     9
     9    16     9
    16     25      1
% Power operator doesn't work on a non
% square matrix because D*D is not defined
>> D^4
??? Error using ==> mpower
Matrix must be square.
% However, an elementwise operator is defined:
>> D.^4
ans =
           1          16
         256          81
        2401        1296
```

The usual division sign - /, should be used only on matrices with a single value. In case of single value, it works as one would expect: it performs a division. However, if given matrix values, the values it produces are not what one would expect, and obtainable in much more intuitive way through the backslash-operator, which we will discuss later. Those interested in using it should familiarize themselves with the `mldivide` manual page. The elementwise version of division-operator is `./`, which is useful on a number of occasions. Because sometimes both elementwise operation, and matrix operation can be invoked, caution is required.
Addition and subtraction are elementwise operations: $\mathbf{A}+\mathbf{B}$ is the standard matrix addition, which requires that both **A** and **B** have the same dimensions. The addition and subtraction operators have also been overloaded to include operations like $2+\mathbf{A}$. This operation is defined as "add 2 to every element of **A**." There, however, is not an elementwise operation, that would allow one to add two matrices having the same number of elements, but different dimensions, together. All of the above holds for the subtraction as well.

```
% Examples of addition
% Sum of two matrices of equal sizes is ok
>> A+B
ans =

     4      4      7
     4      8      9
     5     12      8
% So is adding 2 to every element of A
```

```
>> 2+A
ans =
     5     4     5
     5     6     5
     6     7     3
% This doesn't work because K and L have
% different dimensions
>> K = [1 2 3]
K =
     1     2     3
>> L = [1;2;3]
L =
     1
     2
     3
>> K+L
??? Error using ==> plus
Matrix dimensions must agree.
```

Most of MATLAB's built-in functions, like `exp`, `sin` and `cos` are defined elementwise.

```
% Define an even spaced real valued vector H
>> H = 1:0.5:3
H =
    1.0000    1.5000    2.0000    2.5000    3.0000
% Take an sin of each element of the vector
>> sin(H)
ans =
    0.8415    0.9975    0.9093    0.5985    0.1411
```

Another topic that will require some attention is the matrix and vector dimensions. As mentioned, almost all the operations are dependent on the dimensions of the operands. Oftentimes, like when crafting a function, one does not wish to fix the matrix dimension, but dynamically adapt to the dimensions. The way to do this is to use functions `length` and `size`. Function `length` is primarily meant for work with vectors, and it returns the largest dimension of argument. For example `length(ones(4,2))` would return 4. The function `size` returns a vector containing all dimensions. It is more versatile than `length`, but to work, it requires an assignment. For example, if one wishes to know the number of rows in a vector a, this works:

```
>> dims = size(a);
>> rows = dims(1);
```

Another operation that is frequently needed in order to handle the dimensions is the transpose. MATLAB defaults the transpose to conjugate version, working as transpose on real matrices, but returning the conjugate transpose on complex matrices. The conjugate transpose operator is the '. If one wishes to obtain a non-conjugate transpose, a function `transpose` is available. For work on more complex structures than two-dimensional arrays, MATLAB provides the function `permute`.

```
% Create a complex matrix C
% Recall that i is overloaded
% to act as a complex coefficient
>> C = A+B*i
C =
   3.0000 + 1.0000i   2.0000 + 2.0000i   3.0000 + 4.0000i
   3.0000 + 1.0000i   4.0000 + 4.0000i   3.0000 + 6.0000i
   4.0000 + 1.0000i   5.0000 + 7.0000i   1.0000 + 7.0000i
% Notice the conjugate or hermitian transpose,
>> C'
ans =
   3.0000 - 1.0000i   3.0000 - 1.0000i   4.0000 - 1.0000i
   2.0000 - 2.0000i   4.0000 - 4.0000i   5.0000 - 7.0000i
   3.0000 - 4.0000i   3.0000 - 6.0000i   1.0000 - 7.0000i
% Should you ever need it, a non hermitian transpose
% is also available.
>> transpose(C)
ans =
   3.0000 + 1.0000i   3.0000 + 1.0000i   4.0000 + 1.0000i
   2.0000 + 2.0000i   4.0000 + 4.0000i   5.0000 + 7.0000i
   3.0000 + 4.0000i   3.0000 + 6.0000i   1.0000 + 7.0000i
% Transpose of a real valued vector
>> K'
ans =
     1
     2
     3
```

The matrices can be indexed with two numbers, as usual, the first being the row-index, the second being the column index, and indexes starting from 1.

This is the way matrices should be indexed. There is, however, an alternate way to index matrices. Matrices can be indexed with a single number, the index running down column wise. That is, `A(3) = A(3,1)`. While one may do this, for the sake of clarity, it is highly discouraged. The reason this option is available is due to the properties of computer architecture and C, the language that MATLAB is written with.
MATLAB allows accessing entire rows, columns, and submatrices of any matrix. This is achieved with the range operator `:`. If not given any range, it defaults to whole row or column, for example: the command `A(:,1)` returns the first column of **A**, while `A(2,:)` would return the entire second row of **A**. Instead of selecting the entire row or column, one can select only a part of it by giving the range operator parameters: `A(1:5,1)` would return the first five elements the first column of **A**. Selection of submatrices follows suit: instead of giving one range, we give two: `A(2:3,3:4)` would return a matrix that would contain **A**'s elements $a_{2,3}, a_{2,4}, a_{3,3}$ and $a_{3,4}$. This selection can be extended further: selection index can be any collection of positive integers, and the selection still works, as long as they are within index bounds of A, for example selection `A([1 3 5],2)` returns elements a_{12}, a_{32} and a_{52}.
Selection methods are not limited to numerical indexing; it is also possible to invoke so called logical indexing. Logical indexing is achieved through creating a logical array, and giving it as a index. Logical arrays are returned by logical operators, & ,| and ˜ , relational operators, such as `==`, `˜=`,`>` and `<`, as well as any logical functions, such as `any`, `isinf` and `isequal`. Using these operators and logical indexing, we can, for example, select all the positive elements of a matrix.

```
% Define a large enoug a matrix
>> A = [-2 3 2 4 -4 0; -3 -4 -5 -11 2 4
3 -5 3 2 3 4; 1 -3 2 -4 5 -6; 1 2 3 -4 6 5]

A =

    -2     3     2     4    -4     0
    -3    -4    -5   -11     2     4
     3    -5     3     2     3     4
     1    -3     2    -4     5    -6
     1     2     3    -4     6     5
% Select the third row of the matrix
>> A(3,:)
```

```
ans =
     3    -5     3     2     3     4
% The alternate indexin way: A(12) is the
% the same as A(rem(12,5),mod(12,5)+1).
% While there are situations it can be
% more efficient than the usual way, readability
% suffers.
>> A(12)
ans =
    -5
% Selecting submatrices is quite similar to
% single elements or rows and columns: just
% give to ranges
% Here we have selected rows 2 3 4 and 5, and
% columns 3 4 5 and 6.
>> A(2:5,3:6)
ans =
    -5   -11     2     4
     3     2     3     4
     2    -4     5    -6
     3    -4     6     5
% Finally a look into the logical selection routines:
% select all the elemnts of A less than -4
>> A(A<-4)
ans =
    -5
    -5
   -11
    -6
% A more complicated logical condition: select
% elements of A smaller than 0 but greater than
% -5.
% Note that this requires the use of a elementwise
% logical operator &, which is defined for the use
% with logical matrices and vectors.
>> A((A<0)&(A>-5))
ans =
    -2
    -3
    -4
    -3
    -4
    -4
    -4
```

There are several matrices, that come up often in linear algebra, most notable being the unit matrix. Most of these are provided in MATLAB's matrix library, which generates them according to given parameters. The command `eye` produces the unit matrix of given dimensions. The command `ones` produces a matrix composed entirely of ones, and the command `zeros`, accordingly, produces a matrix made up of zeros. Some of the more exotic built-in matrices are, for example, the Hilbert matrix and the magic square. The Hilbert matrix is produced by command `hilb`. The Hilbert matrix is designed to have certain very poor numerical properties. The magic square is a square matrix with equal column, row and diagonal sums, and it is produced by the command `magic`.

2.2.1 Solving linear equations in MATLAB

The primary tool for solving linear equations in MATLAB is the \-operator. To solve a linear equation of the form $\mathbf{Ax} = \mathbf{b}$ we use the command `x = A \b`. The backslash operator is very versatile: if the matrix $\mathbf{A}$ is overdetermined, i.e, there are more rows than there are columns, a solution in least-square sense is provided. If the system is underdetermined, it finds the basic solution with at most m nonzeros. Here are a few examples:

```
% Example concerning the Hilbert matrix
>> A= hilb(10);
>> x = ones(10,1);
>> b = A*x;
>> sol = A\b;
>> norm(x-sol)

ans =

   8.7188e-04
% The previous lsq-example with the backslash-operator
>> A = [1 1; 2 1; 3 1 ; 5 1; 7 1; 9 1 ; 10 1];
>> b = [444 458 478 506 523 543 571];
>> b = b';
>> x = A\b

x =

   13.0798
  434.1498
```

2.3 Gaussian elimination

Probably the most famous method for solving an $n \times n$ system of linear equations is the *Gaussian elimination* algorithm, named after Carl Friedrich Gauss. The idea of the algorithm is to, for each column of the coefficient matrix, eliminate the elements below the diagonal using row operations, and when an upper triangular matrix is achieved, we do a *backward substitution*, solving x_n from the last equation, and substituting the solution to the second last equation, and thus gaining solution to x_{n-1}, and so forth.

Example 2.3. Solve a system of linear equations using Gaussian elimination, when the system is:

$$\begin{cases} 3x_1 - x_2 + x_3 &= 2, \\ -x_1 + 3x_2 - 2x_3 &= 1, \\ 2x_1 + 2x_2 - x_3 &= -3, \end{cases} .$$

Eliminate all the elements below the first element on the first column: we add the first row multiplied by $\frac{1}{3}$ to the second. Then add the first multiplied by $-\frac{2}{3}$ to the third row, and we get:

$$\begin{cases} 3x_1 - x_2 + x_3 &= 2, \\ 0x_1 + 8x_2 - 5x_3 &= 5, \\ 0x_1 + 8x_2 + x_3 &= -13, \end{cases} .$$

Then eliminate the second element of the third row by adding the second row multiplied by -1 to it, and you get:

$$\begin{cases} 3x_1 - x_2 + x_3 &= 2, \\ 0x_1 + 8x_2 - 5x_3 &= 5, \\ 0x_1 + 0x_2 + 6x_3 &= -18, \end{cases} .$$

We then obtain $x_3 = -\frac{18}{6} = -3$, and place it in the equation on the second row, and get $x_2 = -\frac{5}{4}$, and finally we get $x_1 = \frac{5}{4}$.

The algorithm for Gaussian elimination in MATLAB code is:

Listing 2.1: Algorithm for Gaussian elimination

```
function x = gauss_el(A,b)
n = length(A);
% part a - elimination
for i = 1:n-1
  for j = i+1:n
    % calculate scale factor
    m = A(j,i)/A(i,i);
    % perform row operation:
    % eliminate the elements below diagonal
    % on column i
    A(j,:) = A(j,:) - m*A(i,:);
    b(j) = b(j) - m*b(i);
  end
end
% part b -  backward substitution
x = zeros(n,1);
x(n) = b(n)/A(n,n);

for i = n-1:-1:1
  x(i) = (b(i) - A(i,i+1:n)*x(i+1:n))/A(i,i);
end
```

Gaussian elimination is prone to numeric instability when working on nearly singular matrices. The Hilbert matrix is one example of a nearly singular matrix. Problems rise if at some part of the algorithm the absolute value of the divisor $\hat{a}_{kk}$ (i.e. a diagonal element after k-steps of elimination) is very small. This easily leads to loss of precision due to the nature of floating point arithmetic, and causes the error to accumulate. These situations can generally be avoided through pivoting the matrix, that is, changing the order of rows and/or columns, and applying the same permutations both to the solution vector and the right-hand side of the equation.

Example 2.4. We now establish why Gaussian elimination without pivoting is not a stable algorithm. The function `gauss_elim` is the same as the previous one. We try to numerically solve a system of linear equations $\mathbf{Ax} =$

b where **A** is a Hilbert matrix (the Hilbert matrix is composed as follows: $H_{ij} = 1/(i + j - 1)$) using Gaussian elimination.

```
>>X = ones(13,1);
% We set up a synthetic solution to be a
% vector composed of ones
>>A = hilb(13);
% MATLAB provides some special matrices ready,
% Hilbert's is one of them
>>b = A*X
>>sol = gauss_elim(A,b);
% sol now holds the solution yielded by gauss_el
>> norm(sol-x)

ans =

   11.0527
```

As is obvious, the Gaussian elimination does not provide accurate results when dealing with matrices that are badly conditioned. In numeric cases, it is recommended to use the matrix decompositions, which we will discuss next.

2.4 Matrix decompositions

It is often difficult to solve the equation $\mathbf{Ax} = \mathbf{b}$. Therefore in numeric matrix computation we usually try to present **A** as a product of two or more matrices of some simpler form. This kind of representation is called *matrix decomposition*. As we will see, matrix decompositions will often give us not only an easier way to solve the linear system, but give us information about the decomposed matrix as well.

2.4.1 LU-factorization

In the Gaussian elimination the matrix **A** is first reduced into an upper triangular form, from which it is easy to obtain solutions through back substitution. The idea behind the LU-factorization is to present **A** as a product of two matrices, **L** and **U**, of which **U** is upper triangular, and **L** is lower triangular. We then can solve the equation $\mathbf{Ax} = \mathbf{b}$ by solving two triangular matrix equations:

$$\begin{cases} \mathbf{Ux} = \mathbf{z} \\ \mathbf{Lz} = \mathbf{b} \end{cases}, \text{ that is, } \mathbf{Ax} = \mathbf{LUx} = \mathbf{Lz} = \mathbf{b}$$

The working idea of the LU-algorithm is to perform the Gaussian elimination algorithm on matrix **A**, and take record of the multiplier that was used to zero the elements below the diagonal on each column. Here is a quick example:

Example 2.5.

$$\mathbf{A} = \begin{bmatrix} 1 & -1 & 3 \\ 3 & -5 & 12 \\ 0 & 2 & -10 \end{bmatrix}.$$

We see that in order to eliminate the elements $\mathbf{a}_{21}$ and $\mathbf{a}_{31}$ the first row must be multiplied by 3 and 0 respectively before subtracting from the second and third rows. Thus we get

$$\begin{bmatrix} 1 & -1 & 3 \\ (3) & 8 & 3 \\ (0) & 2 & -10 \end{bmatrix}$$

where numbers in parenthesis represent the recorded multipliers. These will form the lower triangular matrix **L**. On the second step we get

$$\begin{bmatrix} 1 & -1 & 3 \\ (3) & 8 & 3 \\ (0) & (\frac{1}{4}) & -\frac{43}{4} \end{bmatrix}.$$

The diagonal elements can be included either in **L** or **U**. The other matrix will have ones on the diagonal. Now we have the **L** and **U**,

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 0 & \frac{1}{4} & 1 \end{bmatrix}, \mathbf{U} = \begin{bmatrix} 1 & -1 & 3 \\ 0 & 8 & 3 \\ 0 & 0 & -\frac{43}{4} \end{bmatrix}$$

that satisfy

$$\mathbf{LU} = \mathbf{A}.$$

When doing calculations with paper and pen, it is generally easier to use the so called Doolittle algorithm. In this algorithm, the diagonal elements of **L** are fixed to ones.

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & \dots & & a_{2n} \\ & & \vdots & \\ a_{n1} & & \dots & a_{nn} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ l_{21} & 1 & & \\ \vdots & & 1 & \\ l_{n1} & \dots & l_{n(n-1)} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & u_{23} & \vdots \\ 0 & \dots & & u_{(n-1)n} \\ 0 & 0 & & u_{nn} \end{bmatrix}.$$

Here is an example.

Example 2.6. Let's form the LU decomposition for the matrix $\mathbf{A}$, when

$$\mathbf{A} = \begin{bmatrix} 6 & 5 & 12 \\ 30 & 18 & 51 \\ -24 & -76 & -98 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}.$$

The 3×3 matrix gives us 9 equations, each with only one unknown. From the first row we get

$$u_{11} = 6, u_{12} = 5, u = 13 = 12.$$

On the second row, we get

$$\begin{cases} l_{21}u_{11} = 30 \Leftrightarrow l_{21} = 5, \\ l_{21}u_{12} + 1 \cdot u_{22} = 18 \Leftrightarrow u_{22} = 18 - l_{21}u_{12} = -7, \\ l_{21}u_{13} + 1 \cdot u_{23} = 51 \Leftrightarrow u_{23} = 51 - l_{21}u_{13} = -9, \end{cases}$$

and on the third

$$\begin{cases} l_{31}u_{11} = -24 \Leftrightarrow l_{31} = -4 \\ l_{31}u_{12} + l_{32}u_{22} + u_{23} = -76 \Leftrightarrow l_{32} = (\frac{1}{u_{22}}) - (76 - l_{31}u_{12}) = (\frac{1}{-7})(-76 - (-4 \cdot 5)) = 8 \\ l_{31}u_{13} + l_{32}u_{23} + u_{33} = -98 \Leftrightarrow u_{33} = -98 - (l_{31}u_{13}) - (l_{32}u_{23}) = 22 \end{cases}$$

and thus:

$$\mathbf{A} = \mathbf{LU} = \begin{bmatrix} 6 & 5 & 12 \\ 30 & 18 & 51 \\ -24 & -76 & -98 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 5 & 1 & 0 \\ -4 & 8 & 1 \end{bmatrix} \begin{bmatrix} 6 & 5 & 12 \\ 0 & -7 & -9 \\ 0 & 0 & 22 \end{bmatrix}.$$

The Doolittle algorithm stops, if there appears a zero element on the diagonal of $\mathbf{U}$, but it is not limited to invertible matrices, in fact it can be computed on matrices $\mathbf{C} \in \mathbb{C}^{m \times n}$. In this case the $\mathbf{L} \in \mathbb{C}^{m \times m}$ and $\mathbf{U} \in \mathbb{C}^{m \times n}$, and the elements below $u_{kk}, k = 1 \dots m$, will be zeros.

LU-decompositions in MATLAB

MATLAB can compute the LU-factorization on any complex matrix **A** with the command `lu`. The result, however, is not true a lower triangular matrix: MATLAB permutes the parameter matrix **A** so as to achieve maximum efficiency, and the **L** it gives is the product of the permutation matrix and the actual **L**. To get true lower- and upper triangular matrices, we get a third return value: the permutation matrix **P**.

Example 2.7. Here is an example on how to use LU-decomposition in MATLAB.

```
>>A = [-1 1 4;-6 -4 0; 0 4 1]
A =

    -1     1     4
    -6    -4     0
     0     4     1
>>[l u ] = lu(A)
l =

    0.1667    0.4167    1.0000
    1.0000         0         0
         0    1.0000         0



u =

   -6.0000   -4.0000         0
         0    4.0000    1.0000
         0         0    3.5833
>>norm(l*u-A)
ans =

   1.1102e-16
>>[l u p ] = lu(A)
l =

    1.0000         0         0
         0    1.0000         0
    0.1667    0.4167    1.0000
```

```
u =

   -6.0000   -4.0000         0
         0    4.0000    1.0000
         0         0    3.5833


p =

     0     1     0
     0     0     1
     1     0     0
>>norm(l*u-p*A)
ans =

   1.1102e-16
```

2.4.2 Cholesky-decomposition

Another matrix decomposition is the Cholesky-decomposition, named after André-Louis Cholesky. It is not as general as LU-decomposition, but the number of computations required in order to do the decomposition is smaller. The matrices it can decompose are also common in real-life applications.

Definition 2.8. Matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ is said to be *Hermitian* if it holds true that $\mathbf{A}^* = \mathbf{A}$, i.e., $\mathbf{A}$ is its own conjugate transpose. This is analogical to the symmetry of the real matrices. Note that MATLAB's '-operator gives you the conjugate transpose.

Definition 2.9. A Hermitian matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ is *positive definite* if $\langle \mathbf{u}, \mathbf{Au}\rangle > 0 \quad$ for all $\mathbf{u} \in \mathbb{C}^n \backslash \{0\}$.

A matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ that is hermitian and positive definite can be presented with a single upper triangular matrix, as a product

$$\mathbf{A} = \mathbf{U}^*\mathbf{U}.$$

When you have this $\mathbf{U}$, you can simply use the method presented in LU-decomposition to solve the linear system $\mathbf{Ax} = \mathbf{U}^*\mathbf{Ux} = \mathbf{U}^*\mathbf{z} = \mathbf{b}$. Because $\mathbf{U}^*$ is a triangular matrix, this can be solved through back substitution.

The algorithm to produce $\mathbf{U}$ is:

$$u_{kk} = \sqrt{a_{kk} - \sum_{l=1}^{k-1} |u_{lk}|^2}$$

$$u_{kj} = \frac{1}{u_{kk}}\left(a_{kj} - \sum_{l=1}^{k-1} \overline{u}_{lk} u_{lk}\right).$$

If the number under the square root is ever negative, $\mathbf{A}$ is not positive definite and the algorithm halts, thus making the Cholesky-decomposition an efficient tool in studying the positive definity of the matrix.

Example 2.10. Compute the Cholesky-factorization of the matrix $\mathbf{A}$, when,

$$\mathbf{A} = \begin{bmatrix} 13 & 11 & 6 \\ 11 & 11 & 4 \\ 6 & 4 & 10 \end{bmatrix}.$$

Values of upper triangular matrix $\mathbf{U}$ can be computed with this table:

entry	general formula	value for $\mathbf{U}$
u_{11}	$\sqrt{a_{11}}$	$\sqrt{13}$
u_{12}	a_{21}/u_{11}	$\frac{11}{\sqrt{13}}$
u_{13}	a_{31}/u_{11}	$\frac{6}{\sqrt{13}}$
u_{22}	$\sqrt{a_{22} - u_{21}^2}$	$\sqrt{11 - (\frac{121}{13})}$
u_{23}	$(a_{32} - u_{12}u_{13})/u_{22}$	$(4 - \frac{6}{\sqrt{13}} \cdot \frac{11}{\sqrt{13}})/\sqrt{\frac{121}{13}}$
u_{33}	$\sqrt{a_{33} - u_{13}^2 - u_{23}^2}$	$\sqrt{10 - \frac{36}{13} - (11 - \frac{121}{13})}$

You get an upper triangular matrix $\mathbf{U}$:

$$\mathbf{U} = \begin{bmatrix} \sqrt{13} & \frac{11}{\sqrt{13}} & \sqrt{11 - (\frac{121}{13})} \\ 0 & \sqrt{11 - (\frac{121}{13})} & (4 - \frac{6}{\sqrt{13}} \cdot \frac{11}{\sqrt{13}})/\sqrt{\frac{121}{13}} \\ 0 & 0 & \sqrt{10 - \frac{36}{13} - (11 - \frac{121}{13})} \end{bmatrix},$$

having the property $\mathbf{A} = \mathbf{U}^T\mathbf{U}$.

Cholesky-decomposition in MATLAB

To obtain the Cholesky-decomposition in MATLAB, use the function `chol`:

```
>>A = [4 3 6; 4 7 6; 6 2 14]
A =

     4     3     6
     4     7     6
     6     2    14
>>A=A*A'
    61    73   114
    73   101   122
   114   122   236
>>u = chol(A)
ans =

    7.8102    9.3467   14.5962
         0    3.6931   -3.9062
         0         0    2.7735
>>norm(u'*u-A)
ans =

   1.8201e-14
```

2.4.3 QR-decomposition

Any complex square matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ can be decomposed as

$$\mathbf{A} = \mathbf{QR},$$

where $\mathbf{Q}$ is a unitary matrix, and $\mathbf{R}$ is a upper triangular matrix. If $\mathbf{A}$ is nonsingular, then the factorization is unique, if it is required that the diagonal elements of $\mathbf{R}$ are positive.

Definition 2.11. A matrix $\mathbf{A} \in \mathbb{R}^{n\times n}$ is *orthogonal* if $\mathbf{AA}^T = \mathbf{A}^T\mathbf{A} = \mathbf{I}$. A matrix $\mathbf{A} \in \mathbb{C}^{n\times n}$ is *unitary* if $\mathbf{AA}^* = \mathbf{A}^*\mathbf{A} = \mathbf{I}$.

QR-decomposition is often used to solve problems in leas square sense. It is the used in an algorithm for computing the eigenvalues of a matrix.

There are several methods to compute the QR-decomposition, such as Householder transformations and Givens rotations. We use the Gram-Schmidt process. The Gram-Schmidt process is applied to columns of the matrix $\mathbf{A}$ of full column rank, using the inner product $\langle \mathbf{u}, \mathbf{v} \rangle = \mathbf{u}^* \mathbf{v}$.

Definition 2.12. Let V be an n dimensional vector space. A projection of a vector $\mathbf{x} \in V$ onto the subspace spanned by a vector $\mathbf{b}$ is the vector $\mathbf{u}$ into the same direction as $\mathbf{b}$ with length $|\mathbf{u}| = |\mathbf{x}| \cos \theta$, where θ is the angle between the vectors $\mathbf{x}$ and $\mathbf{b}$. Because

$$\cos \theta = \frac{\mathbf{x} \cdot \mathbf{b}}{|\mathbf{x}||\mathbf{b}|}$$

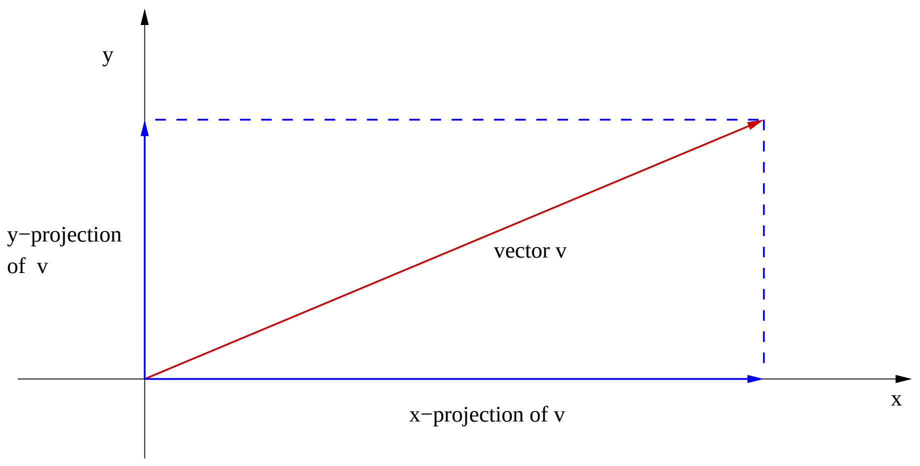
and because $\mathbf{u}$ is in the direction of $\mathbf{b}$ we get

$$\mathbf{u} = |\mathbf{x}| \frac{\mathbf{x} \cdot \mathbf{b}}{|\mathbf{x}||\mathbf{b}|} \frac{\mathbf{b}}{|\mathbf{b}|}.$$

Hence we can define: a projection of $\mathbf{a}$ onto the subspace spanned by $\mathbf{e}$ is

$$\text{proj}_e \mathbf{a} = \frac{\langle \mathbf{e}, \mathbf{a} \rangle}{\langle \mathbf{e}, \mathbf{e} \rangle} \mathbf{e},$$

where the *inner product* $\langle \cdot \rangle$ is defined as $\mathbf{x}^* \mathbf{x}$.



Orthonormalize the columns of $\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2 \dots \mathbf{a}_n]$.

$$\begin{array}{lcll}
\mathbf{u}_1 & = & \mathbf{a}_1, & \mathbf{e}_1 = \frac{\mathbf{u}_1}{||\mathbf{u}_1||}, \\
\mathbf{u}_2 & = & \mathbf{a}_2 - \mathrm{proj}_{e1}\mathbf{a}_2, & \mathbf{e}_2 = \frac{\mathbf{u}_2}{||\mathbf{u}_2||}, \\
\mathbf{u}_n & = & \mathbf{a}_3 - \mathrm{proj}_{e1}\mathbf{a}_3 - \mathrm{proj}_{e2}\mathbf{a}_3, & \mathbf{e}_3 = \frac{\mathbf{u}_3}{||\mathbf{u}_3||}, \\
\vdots & & & \\
\mathbf{u}_n & = & \mathbf{a}_n - \sum_{j=1}^{n-1} \mathrm{proj}_{e_j}\mathbf{a}_n, & \mathbf{e}_n = \frac{\mathbf{u}_n}{||\mathbf{u}_n||}.
\end{array}$$

By rearranging the above equations so that the $\mathbf{a}_i$'s are on the left hand side, and using the fact that $\mathbf{e}_i$'s are unit vectors you get:

$$\begin{array}{lcl}
\mathbf{a}_1 & = & \langle \mathbf{e}_1, \mathbf{a}_1 \rangle \mathbf{e}_1, \\
\mathbf{a}_2 & = & \langle \mathbf{e}_1, \mathbf{a}_2 \rangle \mathbf{e}_1 + \langle \mathbf{e}_2, \mathbf{a}_1 \rangle \mathbf{e}_2, \\
\mathbf{a}_3 & = & \langle \mathbf{e}_1, \mathbf{a}_3 \rangle \mathbf{e}_1 + \langle \mathbf{e}_2, \mathbf{a}_3 \rangle \mathbf{e}_2 + \langle \mathbf{e}_3, \mathbf{a}_3 \rangle \mathbf{e}_3, \\
 & \vdots & \\
\mathbf{a}_n & = & \sum_{j=1}^{n} \langle \mathbf{e}_j, \mathbf{a}_n \rangle \mathbf{e}_j.
\end{array}$$

This can be written in the matrix form

$$\mathbf{A} = \mathbf{QR},$$

where

$$\mathbf{Q} = [\mathbf{e}_1 \mathbf{e}_2 \dots \mathbf{e}_n] \text{ and } \mathbf{R} = \begin{bmatrix} \langle \mathbf{e}_1, \mathbf{a}_1 \rangle & \langle \mathbf{e}_1, \mathbf{a}_2 \rangle & \langle \mathbf{e}_1, \mathbf{a}_3 \rangle & \dots \\ 0 & \langle \mathbf{e}_2, \mathbf{a}_2 \rangle & \langle \mathbf{e}_2, \mathbf{a}_3 \rangle & \\ 0 & 0 & \langle \mathbf{e}_3, \mathbf{a}_3 \rangle & \dots \\ \vdots & & \vdots & \ddots \end{bmatrix}.$$

Example 2.13. Compute a QR-decomposition for the matrix A, when

$$\mathbf{A} = \begin{bmatrix} 2 & 1 & 3 \\ -1 & 0 & 7 \\ 0 & -1 & -1 \end{bmatrix}.$$

The columns of A are:

$$\mathbf{a}_1 = \begin{bmatrix} 2 \\ -1 \\ 0 \end{bmatrix}, \mathbf{a}_2 = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}, \mathbf{a}_3 = \begin{bmatrix} 3 \\ 7 \\ -1 \end{bmatrix}.$$

Then use the Gram-Schmidt process to orthonormalize the vectors:

$$\mathbf{q}_1 = \frac{\mathbf{a}_1}{||\mathbf{a}_1||} = \begin{bmatrix} \frac{2}{\sqrt{5}} \\ -\frac{1}{\sqrt{5}} \\ 0 \end{bmatrix},$$

$$\mathbf{q}_2 = \left(A_2 - \frac{\langle \mathbf{a}_2, \mathbf{q}_1\rangle}{\langle \mathbf{q}_1, \mathbf{q}_1\rangle}\right)\frac{1}{||\mathbf{a}_2||} = \begin{bmatrix} \frac{1}{\sqrt{30}} \\ \frac{2}{\sqrt{30}} \\ -\frac{5}{\sqrt{30}} \end{bmatrix},$$

$$\mathbf{q}_3 = \left(\mathbf{a}_3 - \frac{\langle \mathbf{a}_3, \mathbf{q}_1\rangle}{\langle \mathbf{q}_1, \mathbf{q}_1\rangle} - \frac{\langle \mathbf{a}_3, \mathbf{q}_2\rangle}{\langle \mathbf{q}_2, \mathbf{q}_2\rangle}\right)\frac{1}{||\mathbf{a}_3||} = \begin{bmatrix} \frac{1}{\sqrt{6}} \\ \frac{2}{\sqrt{6}} \\ \frac{1}{\sqrt{6}} \end{bmatrix}.$$

Thus you obtain the orthogonal matrix **Q**:

$$\mathbf{A} = \begin{bmatrix} \frac{2}{\sqrt{5}} & \frac{1}{\sqrt{30}} & \frac{1}{\sqrt{6}} \\ -\frac{1}{\sqrt{5}} & \frac{2}{\sqrt{30}} & \frac{2}{\sqrt{6}} \\ 0 & -\frac{5}{\sqrt{30}} & \frac{1}{\sqrt{6}} \end{bmatrix}.$$

The matrix **R** is

$$\mathbf{R} = \begin{bmatrix} \langle \mathbf{a}_1, \mathbf{q}_1\rangle & \langle \mathbf{a}_2, \mathbf{q}_1\rangle & \langle \mathbf{a}_3, \mathbf{q}_1\rangle \\ 0 & \langle \mathbf{a}_2, \mathbf{q}_2\rangle & \langle \mathbf{a}_3, \mathbf{q}_3\rangle \\ 0 & 0 & \langle \mathbf{a}_3, \mathbf{q}_3\rangle \end{bmatrix} = \begin{bmatrix} \sqrt{5} & \frac{2}{\sqrt{5}} & -\frac{1}{\sqrt{5}} \\ 0 & \frac{6}{\sqrt{30}} & \frac{22}{\sqrt{30}} \\ 0 & 0 & \frac{16}{\sqrt{6}} \end{bmatrix}.$$

The QR-decomposition is:

$$\mathbf{A} = \mathbf{QR} = \begin{bmatrix} 2 & 1 & 3 \\ -1 & 0 & 7 \\ 0 & -1 & -1 \end{bmatrix} = \begin{bmatrix} \frac{2}{\sqrt{5}} & \frac{1}{\sqrt{30}} & \frac{1}{\sqrt{6}} \\ -\frac{1}{\sqrt{5}} & \frac{2}{\sqrt{30}} & \frac{2}{\sqrt{6}} \\ 0 & -\frac{5}{\sqrt{30}} & \frac{1}{\sqrt{6}} \end{bmatrix} \begin{bmatrix} \sqrt{5} & \frac{2}{\sqrt{5}} & -\frac{1}{\sqrt{5}} \\ 0 & \frac{6}{\sqrt{30}} & \frac{22}{\sqrt{30}} \\ 0 & 0 & \frac{16}{\sqrt{6}} \end{bmatrix}.$$

The QR-decomposition can be computed more generally for an $m \times n$ matrix **A**, as long as $m \leq n$.

QR-decomposition in MATLAB

QR-decomposition is offered as a MATLAB function `qr`. Here is a brief example on how to solve linear systems using QR-decomposition. The function `triusolve` is a self-made function to do the back substitution; creating one is an exercise task.

Example 2.14. Here is an example on how to use QR-decomposition to solve a system of linear equations.

```
>>A = [3 -5 7; 0 4 5; -6 -9 -8]
A =

     3    -5     7
     0     4     5
    -6    -9    -8
>>x = ones(3,1);
>>b = A*x
b =

     5
     9
   -23
>>[q r] = qr(A)
>> ba = q'*b;
>> xs = triusolve(r,ba);
>>norm(A*xs-b)
ans =

   6.6465e-15
```

2.4.4 Singular value decomposition

Every $m \times n$ matrix with complex entries can be presented as the product

$$\mathbf{A} = \mathbf{USV}^*,$$

where $\mathbf{U} \in \mathbb{C}^{m\times m}$ and $\mathbf{V} \in \mathbb{C}^{n\times n}$ are orthogonal matrices, and $\mathbf{S} \in \mathbb{C}^{m\times n}$ is a diagonal matrix with entries sorted by magnitude. If the matrix $\mathbf{A}$ is invertible, the inverse is

$$\mathbf{A}^{-1} = \mathbf{VS}^{-1}\mathbf{U}^*.$$

This is easy to compute, as the inverse of a diagonal matrix is just a diagonal matrix with inverses of the original diagonal elements.
MATLAB uses the QR-algorithm to compute the singular value decomposition. If one wishes to compute it manually, the following procedure is propably the easiest:

1. Find the eigenvalues and orthonormalized eigenvectors of $\mathbf{A}^*\mathbf{A}$, i.e.,

$$\mathbf{A}^*\mathbf{A} = \mathbf{V}\Lambda\mathbf{V}^*.$$

2. Sort the eigenvalues according to their magnitude, and let $\sigma_j = \sqrt{\lambda_j}$.
3. Find the first r columns of $\mathbf{U}$ via
$$\mathbf{u}_j = \sigma_j^{-1}\mathbf{A}\mathbf{v}_j, j = 1, \dots, r.$$
4. Pick the remaining columns so that $\mathbf{U}$ is unitary.

Example 2.15. Compute the singular value decomposition $\mathbf{U}\Sigma\mathbf{V}^T$ for matrix $\mathbf{A}$, when
$$\mathbf{A} = \begin{bmatrix} 1 & 2 \\ 2 & 2 \\ 2 & 1 \end{bmatrix}.$$
Begin by getting the $\mathbf{A}^*\mathbf{A}$:
$$\mathbf{A}^*\mathbf{A} = \begin{bmatrix} 9 & 8 \\ 8 & 9 \end{bmatrix}.$$
Continue by computing eigenvalues for the $\mathbf{A}^*\mathbf{A}$, and obtain $\lambda_1 = 17$ and $\lambda_2 = 1$. The corresponding eigenvectors are :
$$\mathbf{v}_1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \text{ and } \mathbf{v}_2 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$
Thus, by taking the square roots of the eigenvalues you get
$$\Sigma = \begin{bmatrix} \sqrt{17} & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}.$$
and by normalizing the eigenvectors you get
$$\mathbf{V} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}.$$
To get $\mathbf{U}$ compute
$$\mathbf{u}_1 = \frac{1}{\sqrt{17}}\mathbf{A}\mathbf{v}_1 = \frac{1}{\sqrt{17}}\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 2 \\ 2 & 2 \\ 2 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{34}}\begin{bmatrix} 3 \\ 4 \\ 3 \end{bmatrix},$$

and

$$\mathbf{u}_2 = 1\mathbf{A}\mathbf{v}_2 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 2 \\ 2 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}.$$

To determine u_3 you need only satisfy:

$$\mathbf{u}_j^* \mathbf{u}_3 = \delta_{j3}, \text{ where } j = 1, 2, 3.$$

With this in mind, you can pick

$$\mathbf{u}_3 = \frac{1}{\sqrt{17}} \begin{bmatrix} 2 \\ -3 \\ 2 \end{bmatrix},$$

and get

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^* = \begin{bmatrix} 1 & 2 \\ 2 & 2 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} \frac{3}{\sqrt{34}} & \frac{-1}{\sqrt{2}} & \frac{2}{\sqrt{17}} \\ \frac{4}{\sqrt{34}} & 0 & \frac{-3}{\sqrt{17}} \\ \frac{3}{\sqrt{34}} & \frac{1}{\sqrt{2}} & \frac{2}{\sqrt{17}} \end{bmatrix} \begin{bmatrix} \sqrt{17} & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}.$$

Condition number

When studying how well the matrix behaves numerically, its determinant, the usual method of determining, whether a system has solutions, does not give accurate estimates on the expected error. This is because matrices $\mathbf{A}$ and $\lambda\mathbf{A}$ have same numerical properties, but $\det(\lambda\mathbf{A} = \lambda^n \det(\mathbf{A}))$. A better estimate on numerical properties of a matrix is given by a *condition number*, defined by:

$$\text{cond}(\mathbf{A}) = \frac{\sigma_1}{\sigma_n},$$

where σ_1 and σ_n are the biggest and the smallest singular value of the matrix $\mathbf{A}$, respectively.
The bigger the condition number, are numerical properties of the matrix. For example the condition number of the Hilbert matrix presented earlier is approximately $1.5 \cdot 10^{10}$. In MATLAB the condition number is computed by the function `cond`.

SVD in MATLAB

MATLAB's built-in function `svd` gives out the singular value decomposition of any matrix given to it. As an example we solve a linear system involving the the Hilbert matrix. Recall that this didn't work with the Gaussian elimination algorithm.

```
>> a = hilb(8);
% We use MATLAB's matrix library to get the Hilbert matrix
>> cond(a)

ans =

   1.5258e+10
>> [u s  v] = svd(a);
>> x = ones(8,1);
% The predetermined solution is a vector consisting of ones.
>> b = a*x;
>> sol = v*(eye(8)/s)*u'*b;
>> norm(sol - x)

ans =

   3.8549e-06
```

The problem with this method of solution is inversing the diagonal matrix **S**. While this is easy from the theoretical point of view, it may numerically be extremely difficult, as it requires the computation of numbers $1/\alpha$ where α is very small.

2.5 Linear least squares

We have been given N pairs (x_i, y_i), and we believe that the y_i:s follow a model of the form $f(x, a_1, a_2, \dots a_M)$. The question now is: how do we best choose the parameters a_i, so that the model $f(x, a_1, a_2, \dots a_M)$ best fits the data (x_i, y_i). The model f is said to be *linear* if it is linearly dependent on the parameters a_i, otherwise it is *non-linear*. To fit the model we usually apply the least-square method, where we minimize the sum

$$S = \sum_{i=1}^{N} (y_i - f(x_i, a_1, \dots, a_M))^2.$$

To solve this, we needs to satisfy:

$$\frac{\partial S}{\partial a_i} = 0, i = 1, \dots M.$$

In case of a linear model, one can interpret the model f applied to observation points as a matrix, and the parameters a_i as unknowns, and thus gain the linear equation

$$\mathbf{Fa} = \mathbf{y}$$

where $\mathbf{F} \in \mathbb{C}^{M \times N}, \mathbf{a} \in \mathbb{C}^M$ and $\mathbf{y} \in \mathbb{C}^N$.

Theorem 2.16. *If $\mathbf{A} \in \mathbb{C}^{m \times n}$ then the equation $\mathbf{A}^*\mathbf{Ax} = \mathbf{A}^*\mathbf{y}$ has at least one solution $\mathbf{x} \in \mathbb{C}^n$, and*

$$||\mathbf{y} - \mathbf{Ax}|| \le ||\mathbf{y} - \mathbf{Az}|| \quad \forall z \in \mathbb{C}^n.$$

Example 2.17. Fit a linear model to the points:

y_i	x_i
444	1
458	2
478	3
506	5
523	7
543	9
571	10

Fitting a linear model means that you will minimize the sum

$$S(a) = \sum_{i=0}^{7} (y_i - (ax_i + b))^2$$

which yields the equations

$$\begin{array}{lcl} a + b & = & 444 \\ 2a + b & = & 458 \\ 3a + b & = & 478 \\ 5a + b & = & 506 \\ 7a + b & = & 523 \\ 9a + b & = & 543 \\ 10a + b & = & 571 \end{array} .$$

This linear system can be written in matrix form $\mathbf{A}\mathbf{x} = \mathbf{b}$ where

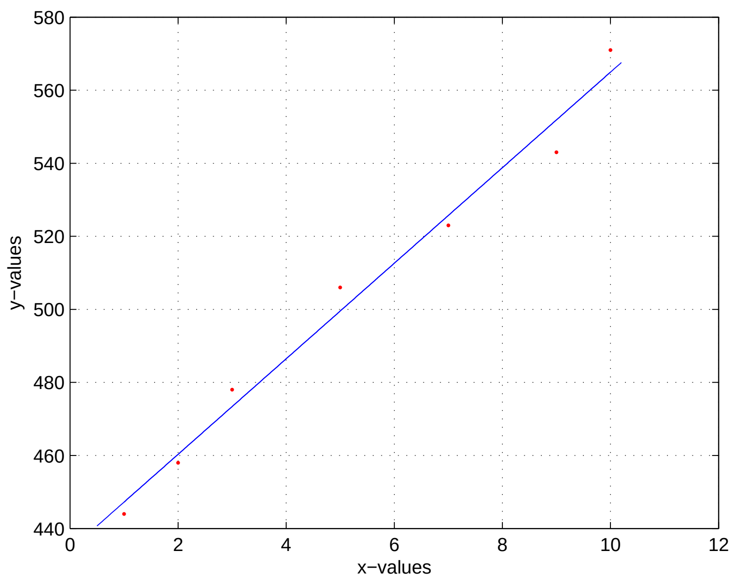
$$\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 3 & 1 \\ 5 & 1 \\ 7 & 1 \\ 9 & 1 \\ 10 & 1 \end{bmatrix}, \mathbf{x} = \begin{bmatrix} a \\ b \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 444 \\ 458 \\ 478 \\ 506 \\ 523 \\ 543 \\ 571 \end{bmatrix}.$$

You then obtain the LSQ-solution by solving $\mathbf{x} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$. The matrix $\mathbf{A}^T\mathbf{A}$ is

$$\begin{bmatrix} 269 & 37 \\ 37 & 7 \end{bmatrix}, \text{its inverse is } \begin{bmatrix} \frac{7}{514} & -\frac{37}{514} \\ -\frac{37}{514} & \frac{269}{514} \end{bmatrix}, \text{ and } \mathbf{A}^T\mathbf{b} \text{ is } \begin{bmatrix} 19582 \\ 3523 \end{bmatrix}.$$

Thus you gain the solution vector $\mathbf{x}$:

$$\mathbf{x} = \begin{bmatrix} \frac{6723}{514} \\ \frac{223153}{514} \end{bmatrix}, \text{ or numerically } \begin{bmatrix} 13.08 \\ 434.15 \end{bmatrix}.$$



Example 2.18. The linearity in linear least squares does not limit its uses to fitting lines: only the linearity of coefficients is required. This example will showcase this. We have some data points (x_k, y_k):

x_i	y_i
21	21
23	43
25	90
27	164
29	221

.

Quick study of the values shows, that fitting a line will not work this time. However, the distribution of the data points suggests, that a polynomial of second degree might work. Now, instead of fitting line $ax + b$, fit a quadratic polynomial $ax^2 + bx + c$.
The data points and quadratic polynomial give us a system of equations

$$\begin{cases} 441a + 21b + c &= 21 \\ 529a + 23b + c &= 43 \\ 625a + 25b + c &= 90 \\ 729a + 27b + c &= 164 \\ 841a + 29b + c &= 221 \end{cases}.$$

This yields an overdetermined linear system

$$\mathbf{Ax} = \mathbf{b}$$

where

$$\mathbf{A} = \begin{bmatrix} 441 & 21 & 1 \\ 529 & 23 & 1 \\ 625 & 25 & 1 \\ 729 & 27 & 1 \\ 841 & 29 & 1 \end{bmatrix}, \mathbf{x} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 21 \\ 43 \\ 90 \\ 164 \\ 221 \end{bmatrix}$$

Now solve this as in previous example: compute the $\mathbf{A}^T\mathbf{A}$ and seek solutions to equation

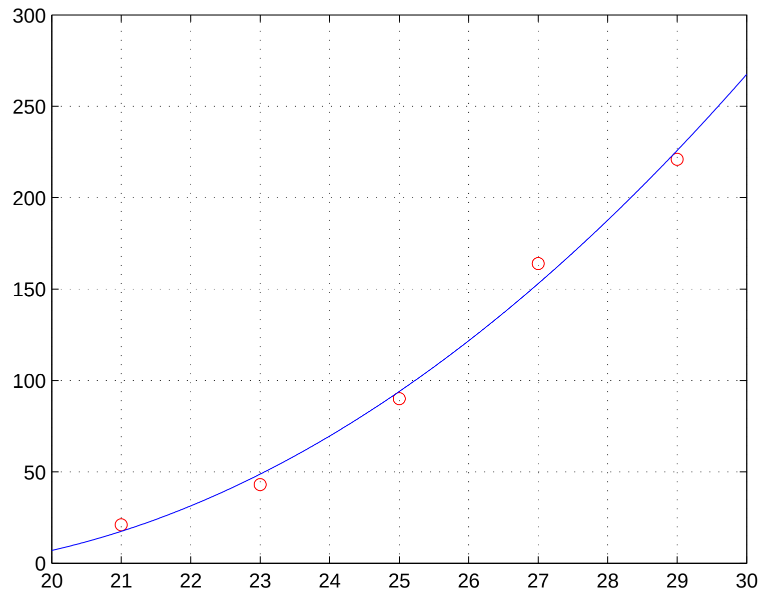
$$\mathbf{A}^T\mathbf{Ax} = \mathbf{A}^T\mathbf{b}$$

by obtaining the inverse $(\mathbf{A}^T\mathbf{A})^{-1}$ and multiply from left both sides of the equation with it.

For the final solution $\mathbf{x} = (\mathbf{A}^T\mathbf{A})^{-1}\mathbf{A}^T\mathbf{b}$ MATLAB's symbolic toolkit gives:

$$\mathbf{x} = \begin{bmatrix} 97/56 \\ -4239/70 \\ 147079/280 \end{bmatrix}.$$

Considering, that the data was synthetically generated by getting values of the function $f(x) = \frac{1}{2}x^2 - 200$ and adding some error, this falls quite far. However, graphical study indicates, that the solution fits the data quite nicely.



The least square method is not limited to fitting linear models. Though the linear interpretation of the model is lost, the premise of the problem does not change: one wishes to minimize the sum

$$S(\alpha) = \sum_{i=1}^{N} (y_i - f(x_i, \alpha))^2.$$

Doing this manually may turn out to be extremely difficult, but in numerical sense, it is possible to gain a good solution through standard minimization algorithms. Different methods of seeking function minimums are discussed later, but an example is given, that illustrates the idea of seeking the minimum.

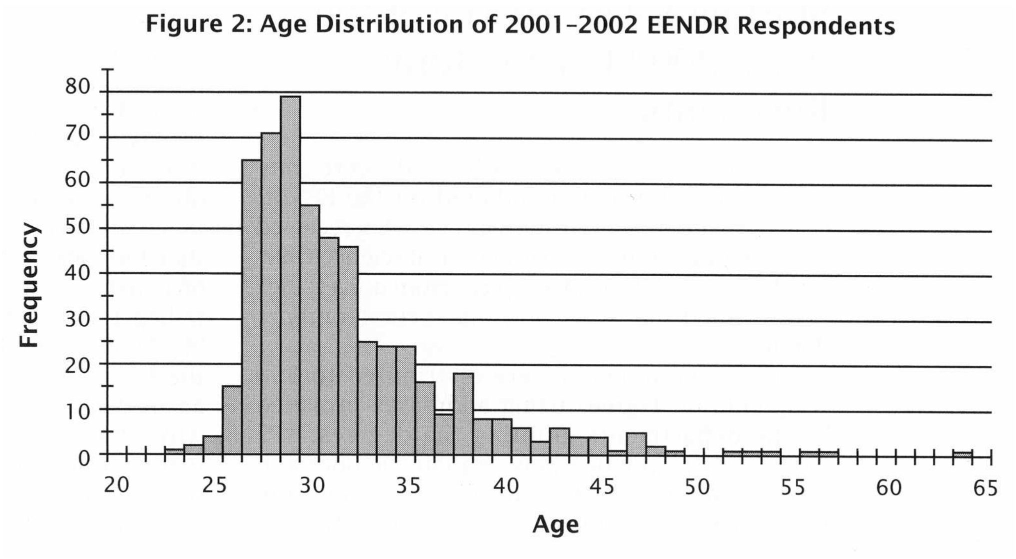
2.5.1 Least squares and MATLAB

In MATLAB one can use the '-operator, and form the matrices as in the previous example, or one can use the Moore-Penrose pseudo-invariant that yields the same results. It is given by MATLAB function `pinv`. Also the standard method for solving linear equations in MATLAB, discussed more previously, automatically gives the LSQ solution if the system is overdetermined or otherwise unsolvable. Here is the previous example in MATLAB:

```
>> A = [1 1; 2 1; 3 1 ; 5 1; 7 1; 9 1 ; 10 1];
>> b = [444 458 478 506 523 543 571];
>> b = b';
>> x = pinv(A)*b
x =

   13.0798
  434.1498
```

The fitted model need not be linear: the proper solution would be gained through computing the partial derivatives in respect to parameters, and solving the system of equations they give, but as this is usually cumbersome a process, it is possible, and oftentimes even preferable to use a function minimum seeking algorithm.

Example 2.19. In this example we wish to study the age doctorate students in math department complete the Ph.D. It is believed, that the function $f(x, \beta) = \beta_1 x^2 e^{-\beta_2 x}$ fits the data we have, and wish to find a β, that satisfies the least square condition.

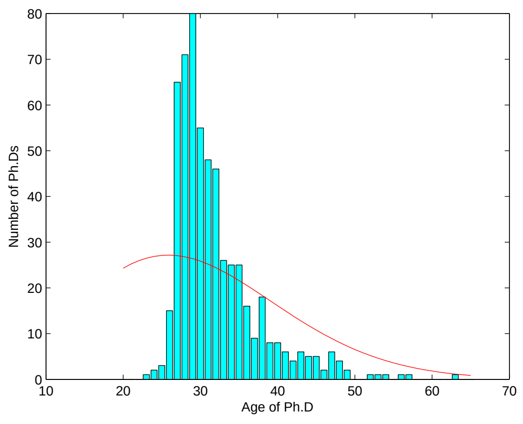


The following code does the minimum search.

Listing 2.2: Non-linear fit

```
clear; close all;
x = 20:65;
y = [0 0 0 1 2 3 15 65 71 80 55 48 46 26 25 25 16 9 18 ...
 8 8  6 4 6 5 5 2 6 4 2 0 0 1 1 1 0 1 1 0 0 0 0 0 1 0 0];
f = inline('beta(1)*x.^2 .* exp(-beta(2)*x.^2)','x','beta');
fobj= inline('norm (fmodel(x,lam)-y)','lam',...
    'fmodel', 'x', 'y');
beta0 = [2 0.01 ];
[beta fval eflag] = fminsearch(fobj,beta0,[],f,x,y);
bar(x,y,'c');
hold on;
plot(x,f(x,beta),'r');
xlabel('Age of Ph.D'); ylabel('Number of Ph.Ds');
```

When plotted, the $f(x, \beta)$, $x \in [20, 65]$, and β the vector produced by the previous algorithm, produces this graph.



2.6 Symbolic linear algebra in MATLAB

MATLAB's symbolic toolbox contains a number of tools with which to perform linear algebra symbolically. Here we present a short introduction to symbolic linear algebra with MATLAB. Most of the functionality of the numerical MATLAB is available in the symbolic toolbox as well. Now the focus is shifted on the symbolic matrix and vector operations.
A list of variables can be designated symbolic with the command `syms`, or for just a single variable, or number, `sym`. The variable designated symbolic can now be used to define a matrix just as usual. The symbolic matrix can now be operated just as a numerical one; most of the operations defined on numerical matrices are defined also on symbolic ones. One should keep in mind though, that fairly fast numeric operation does not translate into fairly fast symbolic one. For example, invoking decomposition algorithms on symbolic matrices can take an exorbitant amount of time. The full list of operations available in symbolic toolbox can be seen at help page `help symbolic`.
Here is an example of how to determine a symbolic matrix, and to obtain it's inverse.

```
>>  B =[sym(2) sym(3) sym(8);
sym(-13) sym(5) sym(6);
sym(-1) sym(13) sym(9)]

B =
[    2,    3,    8]
[  -13,    5,    6]
[   -1,   13,    9]


>> inv(B)

ans =

[       3/95,      -7/95,       2/95]
[ -111/1045,  -26/1045,  116/1045]
[  164/1045,   29/1045,  -49/1045]
```

It is also possible to include non-numeric symbols to matrices, thus gaining more general solutions. Here is a symbolic matrix, and its null space, characteristic polynomial, and determinant.

```
>> A = [2 b c ; 4 2*b 2*c ; a 1 b]

A =

[    2,    b,    c]
[    4, 2*b, 2*c]
[    a,    1,    b]


>> null(A)

ans =

    -(b^2-c)/(-2+b*a)
 -(-2*b+a*c)/(-2+b*a)
                    1

>> poly(A)

ans =

x^3-3*x^2*b+2*x*b^2-2*c*x-2*x^2+2*x*b-a*c*x
```

```
>> det(A)

ans =

0
```

To make use of the generalisations, we use the substitution function `subs` to replace the symbols with the values we wish calculate it with. Here is an example. A symbolic matrix is defined , and its symbolic determinant acquired, and used to compute the values of the determinant at $2, -1$ and 4.

```
>> syms a b c
>> A = [a b 3;
7*a -c 2*b;
c -2*a c]

A =

[     a,     b,     3]
[   7*a,    -c,   2*b]
[     c,  -2*a,     c]


>> d =det(A)

d =

-a*c^2+4*b*a^2-7*a*c*b-42*a^2+2*c*b^2+3*c^2


>> subs(d,{a,b,c},{2, -1, 4})

ans =

  -104
```

Chapter 3

Interpolation

Interpolation is a method of constructing new data points within the range of a discrete set of known data points. If the goal is to generate new data points outside of the range of the presented set of data points, we are discussing *extrapolation*, which is considerably more hazardous.
This chapter will serve as an introduction to a few of the more common methods of interpolation, such as *polynomial*, *linear* and *spline* (more specifically, *cubic spline*) interpolation.

3.1 Polynomial interpolation

Given n points in the plane, (x_k, y_k), $k = 1, 2, \dots, n$, with distinct x_k's, there is a unique polynomial in x of degree less than n whose graph passes through the points. There are many different formulas for this polynomial, but they all define the same function. The polynomial in question is called the *interpolating* polynomial because it exactly reproduces the given data

$$P(x_k) = y_k, \quad k = 1, \dots, n.$$

3.1.1 Lagrange interpolation

One representation on the interpolating polynomial is the *Lagrange* form

$$P(x) = \sum_{k=1}^{n} \left(\prod_{\substack{j=1 \\ j \neq k}}^{n} \frac{x - x_j}{x_k - x_j} \right) y_k.$$

Example 3.1. Let us consider the following data set

```
>> x = 0:3;
>> y = [-5 -6 -1 16];
```

The Lagrangian form of the polynomial interpolating this data is

$$\begin{aligned} P(x) = & \frac{(x-1)(x-2)(x-3)}{-6} \cdot (-5) + \frac{x(x-2)(x-3)}{2} \cdot (-6) \\ & + \frac{x(x-1)(x-3)}{-2} \cdot (-1) + \frac{x(x-1)(x-2)}{6} \cdot 16 \end{aligned}$$

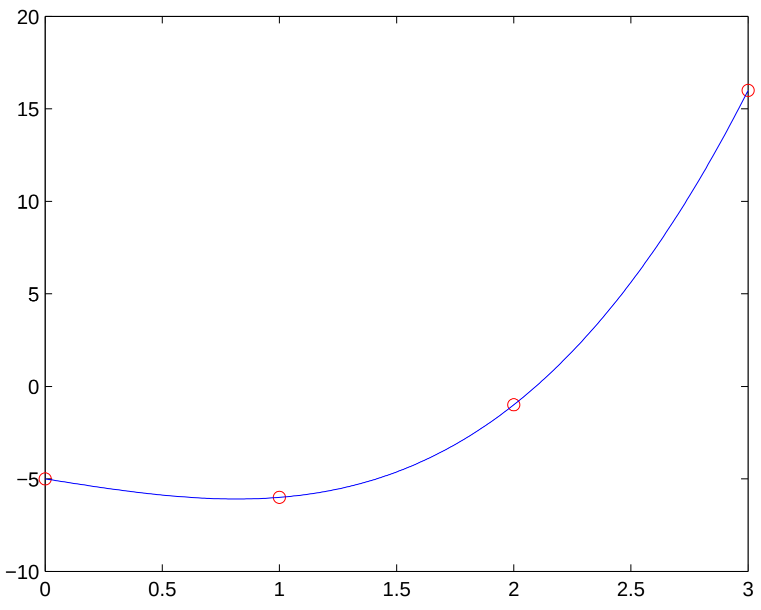
By defining

```
>> xi=0:.01:3;
>> yi=lagrange(x,y,xi);
```

where the function `lagrange` interpolates the values using the lagrange method (homework problem).
The resulting polynomial can now be plotted with the command

```
>> plot(x,y,'or',xi,yi,'-')
```

Output:



3.1.2 Determining coefficients

Polynomials are not usually represented in the Lagrange form, but in its *power form*,

$$P(x) = c_1 x^{n-1} + c_2 x^{n-2} + \cdots + c_{n-1} x + c_n.$$

The coefficients of the power form can, in principle, be computed by solving a system of simultaneous linear equations

$$\begin{pmatrix} x_1^{n-1} & x_1^{n-2} & \cdots & x_1 & 1 \\ x_2^{n-1} & x_2^{n-2} & \cdots & x_2 & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_n^{n-1} & x_n^{n-2} & \cdots & x_n & 1 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}.$$

The $n \times n$-matrix V in the linear system above is called the *Vandermonde* matrix. Its elements are

$$v_{k,j} = x_k^{n-j}.$$

Example 3.2. Define `x` and `y` as

```
>> x=0:3;
>> y = [-5 -6 -1 16];
```

The Vandermonde matrix can be generated in MATLAB with the command `vander`:

```
>> V=vander(x)
V =
     0     0     0     1
     1     1     1     1
     8     4     2     1
    27     9     3     1
```

Now, the linear equation `Vc=y'` can be solved with

```
>> c=V\y'
c =
    1.0000
    0.0000
   -2.0000
   -5.0000
```

In conclusion, the resulting interpolating polynomial is

$$P(x) = x^3 - 2x - 5.$$

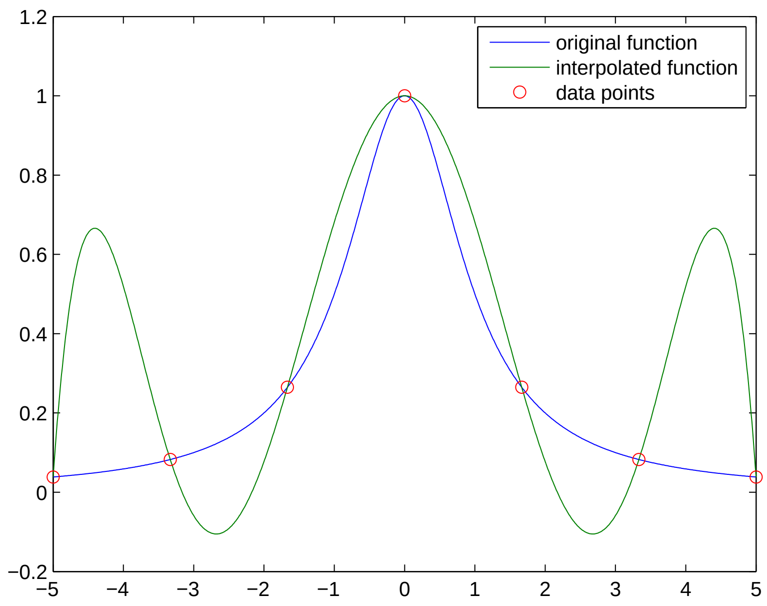
3.2 Runge's phenomenon

The idea with polynomial approximation is that the degree of the polynomial increases as the amount of sample points increases. This does not usually have the desired effect, and as the amount of sample points increase, the less accurate the approximation is. One such example is *Runge's phenomenon.* Observe the equally spaced polynomial approximation of the function $f(x) = 1/(1+x^2)$ in the interval $[-5, 5]$. As the amount n of sample points $x_k = -5 + (k-1) \cdot 10/(n-1)$, $(k = 1, \ldots, n)$, increases, the function starts to wildly oscillate close to the end points of the interval. Thus, the interpolated polynomial will only produce useless results.

Example 3.3. When plotting the polynomial interpolation of the function above for 7 sample points and comparing it with the graph of the original function, one can clearly see a difference. If the number of sample points is increased, the oscillations will become even wilder.

```
% Runge's phenomenon
% Files needed: lagrange.m
xi=-5:.01:5;
n=7; %number of sample points
k=1:n;
x=-5+(k-1).*10./(n-1); % sample points
f=@(x) 1./(1+x.^2); %Runge's function
yi=lagrange(x,f(x),xi); % interpolated values
plot(xi,f(xi), xi,yi, x, f(x), 'or')
legend('original function', 'interpolated function', ...
    'data points')
```

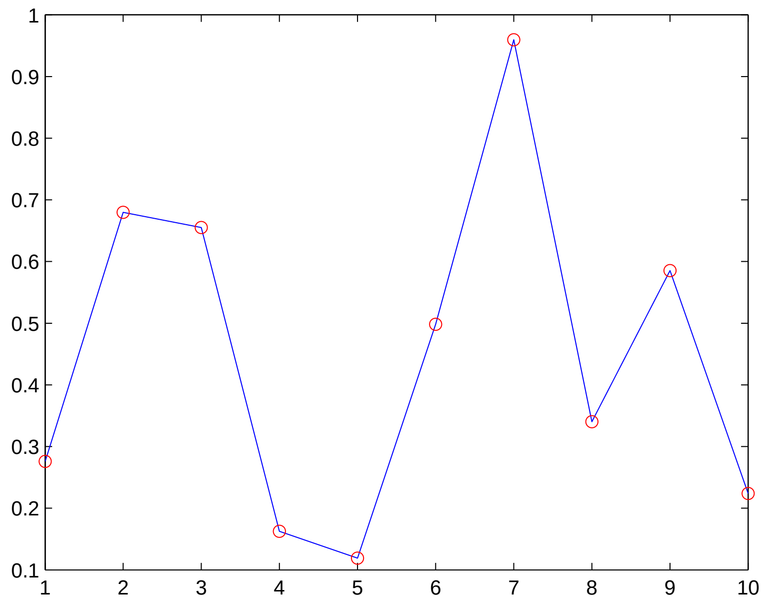
Output:



Lesson: Methods for polynomial approximation (like Lagrange interpolation) should not be used for large values of n ($n \geq 6$). If there are many sample points, one could, for example, use a piecewise cubic interpolation method (like *cubic spline*).

3.3 Piecewise linear interpolation

A simple picture of a data set can be created by plotting the data twice, once with circles at the data points and once with straight lines connecting the points.



To create the lines, MATLAB uses *piecewise linear* interpolation. First, the interval index k must be determined, so that

$$x_k \leq x \leq x_{k+1}.$$

Now, a line between the points (x_k, y_k) and (x_{k+1}, y_{k+1}) can be mapped using analytical geometry. The interpolant between the points can be written as:

$$L(x) = y_k + (x - x_k)\frac{y_{k+1} - y_k}{x_{k+1} - x_k} = Ay_k + By_{k+1}, \tag{3.4}$$

where

$$A = \frac{x_{k+1} - x}{x_{k+1} - x_k} \quad \text{and} \quad B = \frac{x - x_k}{x_{k+1} - x_k}. \tag{3.5}$$

The points x_k are sometimes called *breakpoints* or *breaks*.
The piecewise linear interpolant $L(x)$ is continuous in reference to x, but its derivate is not continuous. The derivate is

$$L'(x) = \frac{y_{k+1} - y_k}{x_{k+1} - x_k}$$

for all $x \in [x_k, x_{k+1}]$, and it jumps at the breakpoints.

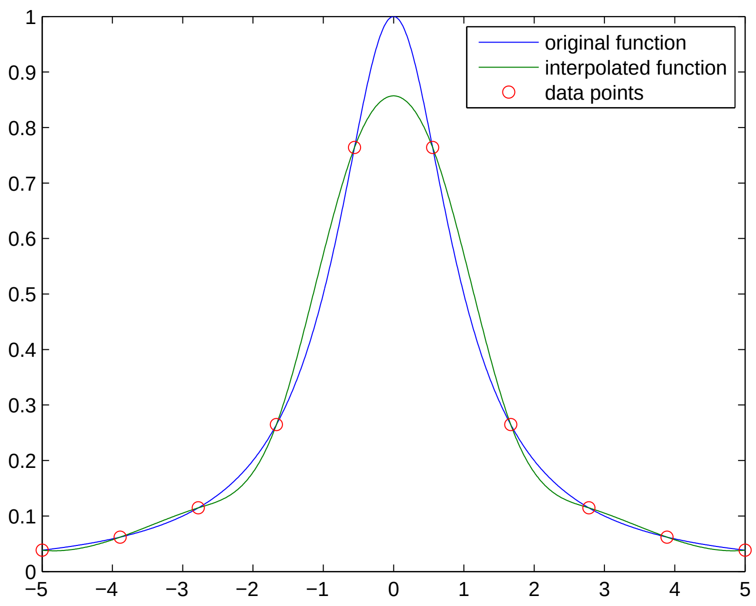
3.4 Splines

Spline interpolation is a built-in function in MATLAB and can be accessed with the command `spline([datapoints],[datapoint values], [interpolant])`. The function returns the interpolated values.
There are several methods for spline interpolation, but what all the methods have in common, is its piecewise polynomial nature. It works in a similar way as piecewise linear, but instead of linear functions, one uses polynomial functions of a fixed degree whose derivatives are continuous at the breakpoints (called *knots* when discussing spline).
The classical approach is to use polynomial functions of degree 3, this is the case of *cubic spline*, which MATLAB also uses.

Example 3.6. Plot the spline interpolation of Runge's function (presented in the section on Runge's phenomenon).

```
xi=-5:.01:5;
n=10; %number of data points
k=1:n;
x=-5+(k-1).*10./(n-1); % data points
f=@(x) 1./(1+x.^2); %Runge's function
yi=spline(x,f(x),xi); % interpolated values
plot(xi,f(xi), xi,yi, x, f(x), 'or')
legend('original function', 'interpolated function', ...
    'data points')
```

Output:



In contrast to polynomial interpolation, here the accuracy will increase as the amount of data points increases.

We will now have a closer look at the theory behind cubic spline.

3.4.1 Cubic spline

The polynomials used in cubic spline are of third degree, and must have continuous second derivatives and satisfy the interpolation constraints.
Suppose, that in addition to the tabulated values of y_k one would also have the tabulated values to the function's second derivatives, that is, a set of numbers y_k''. Now, one can add to the right-hand side of the equation for piecewise linear interpolation, i.e.

$$L(x) = Ay_k + By_{k+1}, \tag{3.7}$$

where

$$A = \frac{x_{k+1} - x}{x_{k+1} - x_k} \quad \text{and} \quad B = \frac{x - x_k}{x_{k+1} - x_k}, \tag{3.8}$$

a cubic polynomial whose second derivative varies from y''_k at the left of the interval and y''_{k+1} at the right. This will produce the desired continuous second derivative. By also constructing the cubic polynomial so that it has values of zero at x_k and at x_{k+1}, then adding it in will not change the behaviour at the knots (i.e. the value y_k at x_k in the interval $[x_{k-1}, x_k]$ is equal to the value y_k at x_k in the interval $[x_k, x_{k+1}]$).
This can be achieved with

$$y = Ay_k + By_{k+1} + Cy''_k + Dy''_{k+1}, \tag{3.9}$$

where A and B are defined as above in (3.8), and

$$C = \frac{1}{6}(A^3 - A)(x_{k+1} - x_k)^2 \quad \text{and} \quad D = \frac{1}{6}(B^3 - B)(x_{k+1} - x_k)^2. \tag{3.10}$$

One can easily check that y'' is in fact the second derivative of the interpolating function. The derivatives of equation (3.9) with respect to x can be taken by using the definitions of A, B, C and D to compute dA/dx, dB/dx, dC/dx and dD/dx.
The first derivative is now

$$\frac{dy}{dx} = \frac{y_{k+1} - y_k}{x_{k+1} - x_k} - \frac{3A^2 - 1}{6}(x_{k+1} - x_k)y''_k + \frac{3B^2 - 1}{6}(x_{k+1} - x_k)y''_{k+1} \tag{3.11}$$

and the second derivative is

$$\frac{d^2y}{dx^2} = Ay''_k + By''_{k+1}. \tag{3.12}$$

In the calculations above, it was assumed that the y''_k's were known. In order to calculate them, one must require that the first derivative of the polynomial is also continuous. Now, the required equations can be obtained from (3.11) by setting the value for $x = x_k$ in the interval $[x_{k-1}, x_k]$ to be equal to the value for $x = x_k$ in the interval $[x_k, x_{k+1}]$. With some rearrangement, this gives

$$\frac{x_k - x_{k-1}}{6}y''_{k-1} + \frac{x_{k+1} - x_{k-1}}{3}y''_k + \frac{x_{k+1} - x_k}{6}y''_{k+1} = \frac{y_{k+1} - y_k}{x_{k+1} - x_k} - \frac{y_k - y_{k-1}}{x_k - x_{k-1}}$$

for all $k = 2, \ldots, N - 1$.
Now y_k, where $k = 1, \ldots N$, can be solved from this system of $N - 2$ linear equations. In order for the solution to be unique, the boundary conditions at x_{k+1} and x_k must be specified. The most common ways of doing this is to either

- set one or both of y_1'' and y_N'' to zero, which will give us the, so called, *natural cubic spline*, or
- set either of y_1'' and y_N'' to values calculated from (3.11) so as to give the first derivative of the interpolating function at either or both boundaries a specific value.

Example 3.13. Let the set of sample points (x_k, y_k) be $(1,2), (2,1), (3,5), (4,3)$. Using equations (3.8) for A and B, we get the following

$$A = \begin{bmatrix} 2-x \\ 3-x \\ 4-x \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} x-1 \\ x-2 \\ x-3 \end{bmatrix}.$$

Using these values for A and B, and equations (3.10) for C and D, we get

$$C = \begin{bmatrix} \frac{1}{6}((2-x)^3 - 2 + x) \\ \frac{1}{6}((3-x)^3 - 3 + x) \\ \frac{1}{6}((4-x)^3 - 4 + x) \end{bmatrix} \quad \text{and} \quad D = \begin{bmatrix} \frac{1}{6}((x-1)^3 - x + 1) \\ \frac{1}{6}((x-2)^3 - x + 2) \\ \frac{1}{6}((x-3)^3 - x + 3) \end{bmatrix}.$$

In equation (3.9) the piecewise cubic polynomial was defined as

$$y = Ay_k + By_{k+1} + Cy_k'' + Dy_{k+1}''.$$

By using the derivate (3.11) of this polynomial and rearranging it, as described, and investigating it at the knots, we receive the following linear system equations

$$\begin{cases} \frac{1}{6}y_1'' + \frac{2}{3}y_2'' + \frac{1}{6}y_3'' = 5 \\ \frac{1}{6}y_2'' + \frac{2}{3}y_3'' + \frac{1}{6}y_4'' = -6 \end{cases}$$

which has the solutions

$$\begin{cases} y_1'' = t_1 \\ y_2'' = -\frac{4}{15}t_1 + \frac{1}{15}t_2 + \frac{52}{5} \\ y_3'' = \frac{1}{15}t_1 - \frac{4}{15}t_2 - \frac{58}{5} \\ y_4'' = t_2 \end{cases} \qquad t_1, t_2 \in \mathbb{R}.$$

The values for t_1 and t_2 can now be set to zero and the piecewise cubic polynomial is

$$y = \begin{cases} \frac{26}{15}x^3 - \frac{26}{5}x^2 + \frac{37}{15}x + 3, \text{ when } x \in [1,2] \\ -\frac{11}{3}x^3 + \frac{136}{5}x^2 - \frac{187}{3}x + \frac{231}{5}, \text{ when } x \in [2,3] \\ \frac{29}{15}x^3 - \frac{116}{5}x^2 + \frac{1333}{15}x - 105, \text{ when } x \in [3,4] \end{cases}$$

3.5 Additional methods for interpolation in MATLAB

One function in MATLAB, that allows the user to specify the desired interpolation method, is `interp1`. It can be accessed with the command

```
interp1([datapoints],[datapoint values],[interpolant],...
        ...[method],[extrapolation]).
```

The argument `[method]` specifies the specific interpolation method, available methods are

- `'linear'`, which specifies linear interpolation. This is the default method, and will be used if no method is specified.
- `'nearest'`, which uses nearest neighbor interpolation. The interpolated value in a specific point will be the same as the value of the nearest datapoint.
- `'spline'`, which uses piecewise cubic spline interpolation.
- `'pchip'`, which uses shape-preserving piecewise cubic interpolation, also known as piecewise cubic Hermite interpolation.
- `'cubic'`, which is the same as `'pchip'`.
- `'v5cubic'`, which is cubic interpolation used in MATLAB 5. This method does not extrapolate and if the datapoints are not equally spaced, `'spline'` is used instead.

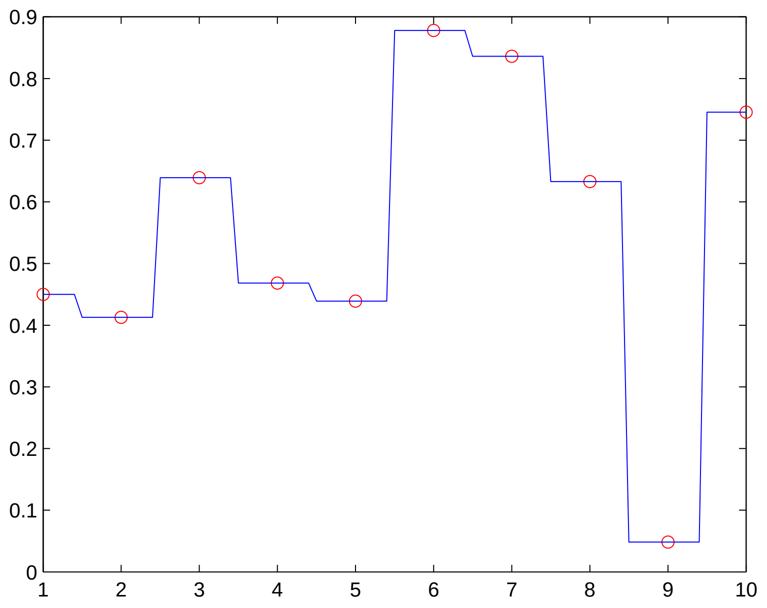
Example 3.14. Define the datapoints and the interpolant as

```
>> x=1:10;
>> y=rand(1,10);
>> xi=1:.1:10;
```

The nearest neighbor method:

```
>> yi=interp1(x,y,xi,'nearest');
>> plot(x,y,'or',xi,yi)
```

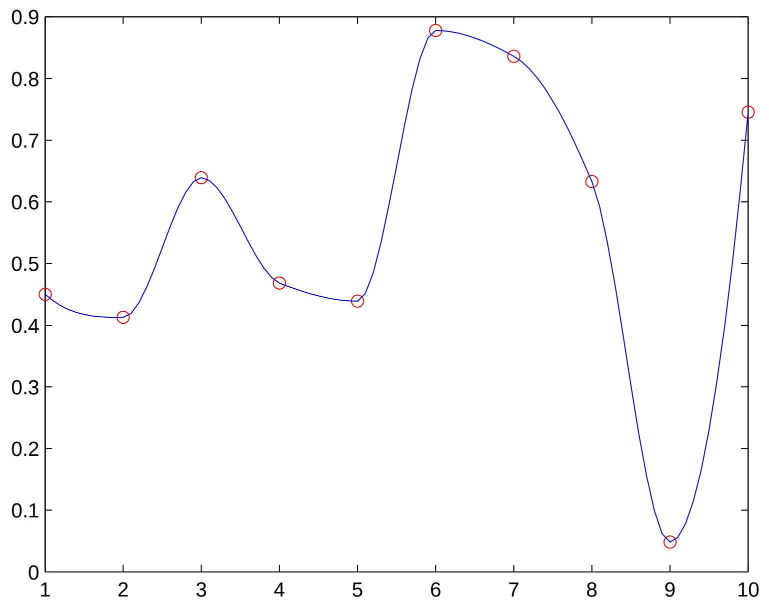
Output:



The shape-preserving piecewise cubic method:

```
>> yi=interp1(x,y,xi,'cubic');
>> plot(x,y,'or',xi,yi)
```

Output:



The argument [extrapolate] can be used to evaluate points outside of the given interval of data points.
If the argument is specified as 'extrap', the function will use the specified method to evaluate any out of range values in [interpolant].

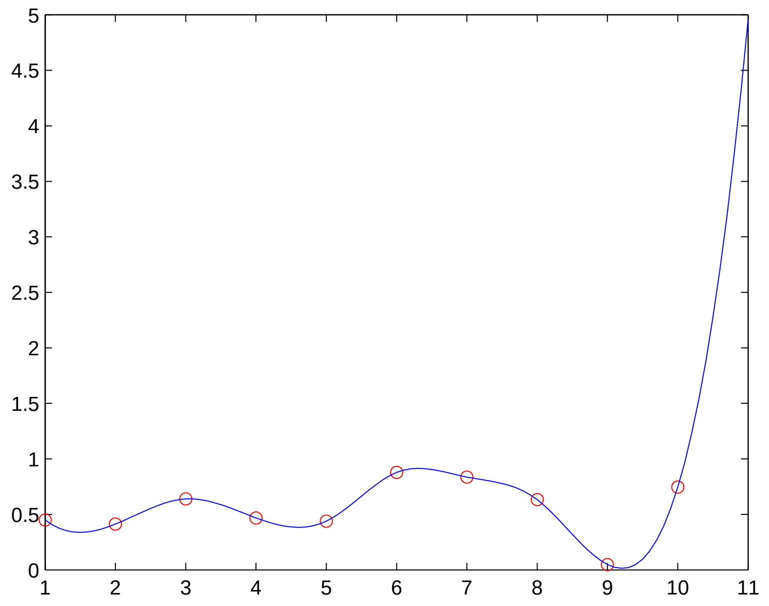
Example 3.15. Let x and y be defined as in the last example, and define the interpolant as

```
>> xi=1:.1:11;
```

Now, the interpolant is defined outside of the range of datapoints, and the points outside of the range must be extrapolated.

```
>> yi=interp1(x,y,xi,'spline','extrap');
>> plot(x,y,'or',xi,yi)
```

Output:



The [extrapolation] argument can also be specified as a scalar to be returned for any out of range values. Here, 0 and NaN are often used.

The function can also be defined, for example, as
pp=interp1(x,y,[method],'pp'),
which will use the method specified in the arguments (except for 'v5cubic') to generate the piecewise polynomial form of the datapoint values. Then ppval can be used to evaluate that piecewise polynomial. For example, ppval(pp,xi), where pp is defined as above, is equivalent to
interp1(x,y,xi,[method],'extrap').

Chapter 4

Numerical differential and integral calculus

4.1 Numerical derivation

The derivative of a function measures how its values changes as its parameters change. It is defined via limiting values of difference quotient.

Definition 4.1. The derivative of function f at x_0 is the limit

$$f'(x) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}.$$

When f is a function of one real variable, the derivative is the slope of the tangent line drawn to the graph of the function at real number x_0.

From the perspective of the numerical computation the definition is skewed: it tells the behaviour of the function either before or after the derivation point. Applying it numerically will give results with error term proportional to h.
One wishes to know behaviour of the function both before and after the derivation point. This is achieved by fitting a secant line travelling through the points $((x_0 - h), f(x_0 - h))$ and $((x_0 + h), f(x_0 + h))$, and computing its slope. As h approaches 0, the secant line approaches the tangent line of the function at x_0:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}, \text{ if } h \approx 0. \tag{4.2}$$

In numerical sense, using this so called *three point rule*, will yield results with error terms proportional to h^2.

In order for the formula 4.2 to work, the parameter h must be selected appropriately; while the intuition says, that the smaller the $|h|$, the better the results, the truth is, that a small value of h will result in extremely bad loss of precision. Literature on the subject suggests that usually selection $h = (meps)^{0.5}$ yields the best results. Furthermore, it may be necessary to ensure that the selected h is presentable in floating point arithmetic. If it is not, then the difference of x_0 and $x_0 + h$ is not exactly h, which will lead to additional accumulation of error. Considering this phenomenon in MATLAB is not necessary because of the optimization procedures, but in compiled languages one should take steps to ensure proper representation of h. The formula 4.2 is susceptible to bad properties of function: if the values of the function f vary widely on the interval $(x - h, x + h)$, the results it provides are not accurate. Here is an example code on how to implement this in MATLAB:

Listing 4.1: Numerical derivative

```
function df = numdif(f,x,h)
% x is an n-vector
[m,n] = size(x);  one = ones(m,n);
df = (feval(f,x+h*one)-feval(f,x-h*one))/(2*h);
```

Here is an example on how to use the function `numdif`, and then an example, why this method of derivation should be only applied with care.

Example 4.3. We numerically derivate a function whose derivative is easy to define:

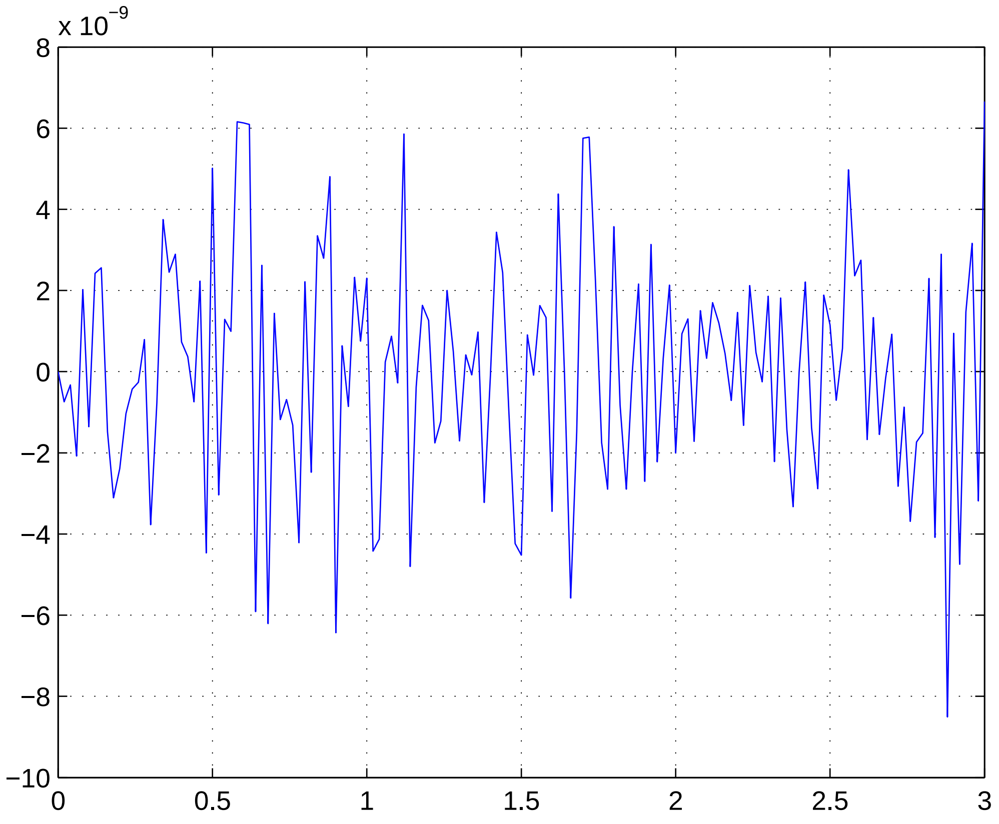
$$f(x) = \cos(4x) - \sin(2x),$$

and then compare it to the real derivative,

$$f'(x) = -4\sin(4x) - 2\cos(2x).$$

Listing 4.2: Example of numerical derivative

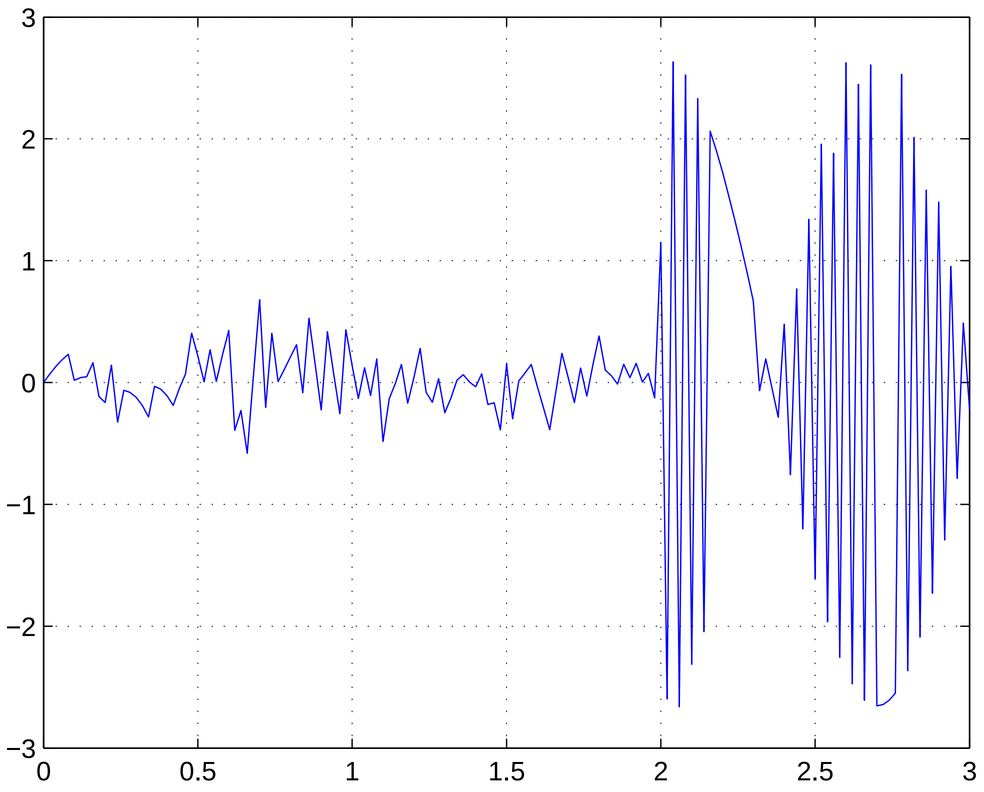
```
>> f = inline('cos(4*x)-sin(2*x)','x');
>> x = 0:0.02:3;
% We now compute the numerical
% derivative on the interval x
% eps is MATLAB built-in value
% for machine epsilon
>> df = numdif(f,x,eps.^0.5);
% To establish the accuracy of
% the numeric derivative,
% we compare it to the actual derivative
>> y = -4*sin(4*x) - 2*cos(2*x);
% we plot the difference of the numeric
% an the actual derivative
>> plot(x, y-df);
```



As can seen, the maximum error seems to be of magnitude $6 \cdot 10^{-9}$, which is comparably tolerable. Next presented is a warning example on the effects of a poor choice of h:

Listing 4.3: Consequences of poorly selected h

```
>> f = inline('cos(4*x)-sin(2*x)','x');
>> x = 0:0.02:3;
% We now compute the numerical derivative
% on the interval x
>> df = numdif(f,x,eps);
% To establish the accuracy of the
% numeric derivative,
% we compare it to the actual derivative
>> y = -4*sin(4*x) - 2*cos(2*x);
% we plot the difference of the numeric
% an the actual derivative
>> plot(x, y-df);
```



As is obvious, selecting too small an h can yield staggeringly bad results.

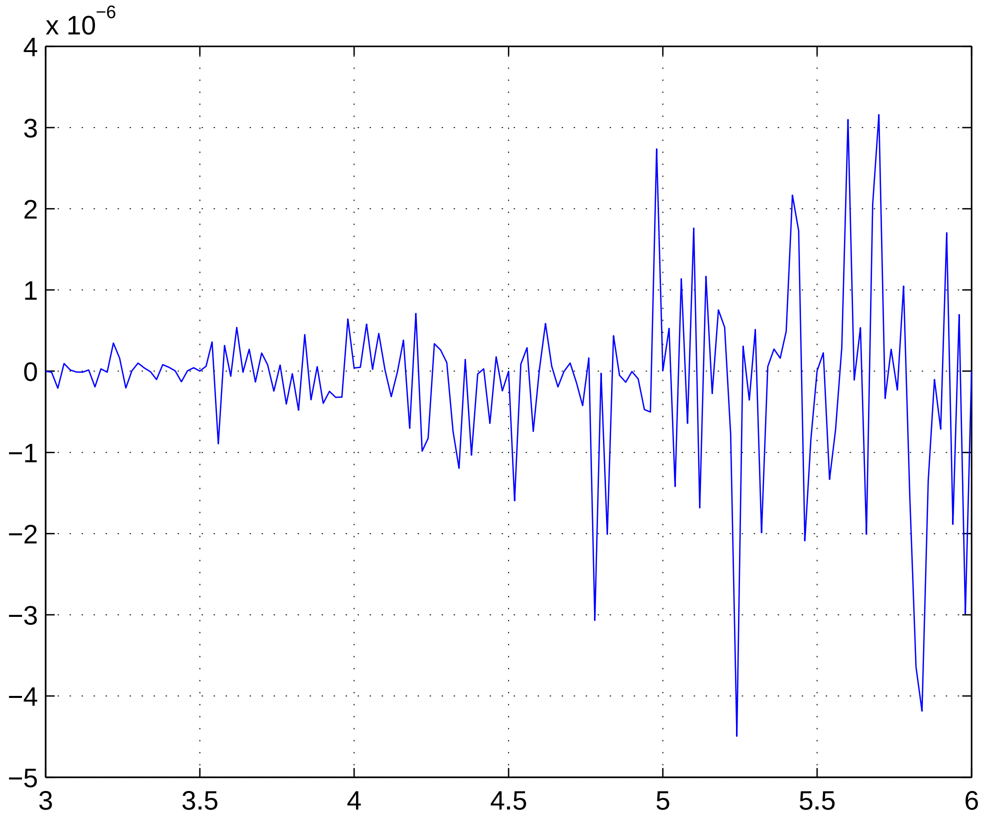
Example 4.4. There are situations where not even a proper choice of parameters can help to salvage the accuracy of numerical derivative. To showcase this the behaviour of numerical derivative of function

$$f(x) = \sin(x^4)$$

on the interval $(3, 6)$ is studied. It is then compared to the true derivative of f,

$$f'(x) = x^3 \cos(x^4)$$

```
>> f = inline('sin(x.^4)','x');
>> fd = inline('(4*x.^3).*(cos(x.^4))','x');
>> x = 3:0.02:6;
>> plot(x,numdif(f,x,eps.^0.5)-fd(x));
```



The picture shows, that the errors are of magnitude $4 \cdot 10^{-6}$, which, while not unbearable, can in certain situations be meaningful. One should also note, that the function's variation in values continues to increase in frequency, thus making numerical derivation highly suspect.

In addition to poorly behaving functions, there are functions that are not differentiable, either at specific points or at all, but whose numerical derivatives can be obtained. For example, it is well known that the function $f(x) = |x|$ does not have derivative at 0. However, when computed with the function `numdif` this is not instantly obvious.

```
>> numdif(@abs,0,1e-8)
ans =
     0
```

Only times the numerical derivative is not technically obtainable are at discontinuity intervals of the evaluated function. One should however keep in mind that even if a derivative is numerically obtainable, it does not mean that it exists.

4.1.1 Estimating derivative with polynomial

The previous estimate for a derivative of a function was based on linear approximation of the function on the interval $(x - h, x + h)$. This leads one to wonder, whether it is possible to increase the accuracy of the derivation through better approximation of the function.
If a function f is approximated with a polynomial, basing the approximation on points $x_i = x + ih, i = -n, \ldots, n$, one can acquire a polynomial using the Lagrange interpolation method. Suppose then that $n = 2$ has been chosen, and been used to created the estimate p_2, with $p_2(x_i) = f(x_i)$. The derivative can now be approximated:

$$f'(x_0) \approx p_2'(x_0).$$

Different approximations to functions derivatives, and their accuracy have been widely discussed in literature. We rest the matter by giving the result the previously presented polynomial estimate yields, though without proof.

$$f'(x_0) \approx \frac{1}{h}\left(\frac{1}{12}f(x_{-2}) - \frac{2}{3}f(x_{-1}) + \frac{2}{3}f(x_2) - \frac{1}{12}f(x_2)\right).$$

Here is the MATLAB implementation to the five point rule:

```
function dy = diff(y,h)
% The 5-point rule
% the parameter y is
% the values of the
% function on the interval
% we wish to obtain the
% derivatives, h is the step factor
% To compute dy at single point x0
% set interval x = x0-2*h:h:x0+2*h
% y=f(x), and invoke diff
% dy = diff(y,h);
for p =-2:2
    a= (2*p^3-3*p^2-p+1)/12; b= (4*p^3-3*p^2-8*p+4)/6;
    c= (2*p^3-5*p)/2;
    d= (4*p^3+3*p^2-8*p-4)/6; e= (2*p^3+3*p^2-p-1)/12;
    coe=[coe; [a -b c -d e]];
end;
% We now make sure that y is of proper size
[d1,d2]=size(y);
if ((min(d1,d2)>1) | (max(d1,d2) <5))
    error('Argument error in numder');
end;
dy =y;
dy(1)=(1/h)*sum(coe(1,:).*y(1:5));
dy(2)=(1/h)*sum(coe(2,:).*y(1:5));
for p=3:d2-2
    dy(p)=(1/h)*sum(coe(3,:).*y(p-2:p+2));
end;
dy(d2-1)=(1/h)*sum(coe(4,:).*y(d2-4:d2));
dy(d2)=(1/h)*sum(coe(5,:).*y(d2-4:d2));
```

4.2 Jacobian matrix

When studying functions with more than one component and variable, a best tool to observe the differentiation of a function is the *Jacobian matrix.* Jacobian matrix contains all first-order partial derivatives of a vector- or scalar-valued function on it's columns.

Suppose $F : \mathbb{R}^n \to \mathbb{R}^m$ has components

$$F(x_1 \dots x_n) = (F_1(x_1 \dots x_n)), F_2(x_1 \dots x_n) \dots F_m(x_1 \dots x_n)).$$

Then its Jacobian matrix is

$$\begin{bmatrix} \frac{\partial F_1}{\partial x_1} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial x_1} & \cdots & \frac{\partial F_m}{\partial x_n} \end{bmatrix}.$$

As one can see, if $(x_1, \dots, x_n)$ are the orthogonal Cartesian coordinates, as usual, the k:th row of Jacobian is the *gradient* of the k:th component of the function F.
To numerically compute the Jacobian matrix we use the method in one direction at time, filling the Jacobian matrix column wise.

Listing 4.4: Algorithm for numerical Jacobi matrix

```
function Jf = jacobian_matrix(f,x,m,n)
% here f is the function to be derivated,
% x is the point of derivation,
% m is the number of component functions,
% and n is the number of parameters.
% we begin by initializing Jf
Jf = zeros(m,n);
h = eps.^0.5;
% e will define the direction we wish to partially derivate
e = zeros(n,1);
% function f will produce m partial derivatives,
% thus filling the column
for j=1:n
    %set the direction
    e(j) = 1;
    Jf(:,j) = (f(x+e*h) - f(x-e*h))/(2*h);
    e(j)=0;
end
```

Jacobian matrix describes the orientation of the tangent plane of the function at a given point; one can think it a generalized gradient.
Jacobian matrix can through the *inverse function theorem* say, whether a function has an inverse at some point or not. The inverse function theorem

states, that matrix inverse of the Jacobian matrix of an invertible function is the Jacobian matrix of the inverse function. Hence,

$$J_{f^{-1}}(f(p)) = J_f(p)^{-1}.$$

Because the existence of inverse function is usually more interesting than actually determining what it is, it is often enough to compute the determinant of the J_f, called *Jacobian determinant*, or just *Jacobian*. The Jacobian plays a large role in many fields of mathematics, such as partial differential equations. Jacobian matrix can also be used to linearly approximate the function on short intervals, and it is essential when applying the Newton method on vector functions.

4.3 Numerical derivation on complex plane

It is often desirable to perform numeric differential calculus on complex functions. This is possible in MATLAB using the built-in complex variables i and j.
Complex functions are functions that map complex variables into complex plane. Any complex number can be separated in to real and imaginary parts:

$$z = x + yi,$$

where $z \in \mathbb{C}, x, y \in \mathbb{R}$. Similarily any complex function can be divided into real and separate parts:

$$f(z) = u(x, y) + iv(x, y),$$

where $u, v : \mathbb{R}^2 \to \mathbb{R}$ and $x, y \in \mathbb{R}$.
Complex derivation at point $z_0 \in \mathbb{C}$ is defined as a limiting value on a complex function f

$$f'(z_0) = \lim_{h \to 0} \frac{f(z_0 + h) - f(z_0)}{h},$$

where $h \in \mathbb{C}$.
One should notice, that while this definition seems very much like like its counterpart on the real line, the fact that $h \in \mathbb{C}$ makes matters a bit complicated. Instead of two possible directions of approach, there are now in fact infinitely many directions from where h can approach 0. It usually pays to

express the complex number in polar coordinates to make the determination of the limiting value easier.
Evaluating the derivative numerically may sometimes be deceivingly easy: while the method need not necessarily be different than the one we observed before for real functions. As a rule MATLAB does not need any special instructions on how to deal with complex variables:

```
>> f = inline('z.^2','z')
f =
     Inline function:
     f(z) = z.^2
>> numdif(f,2+2*i,1e-8)
ans =
   4.0000 + 4.0000i
```

Problems rise when we encounter functions that are not differentiable; when dealing with complex functions these are not always easy to identify. For example the complex conjugate: $f(z) = f(x + iy) = x - iy = \overline{z}$ is not differentiable anywhere on complex plane, but `numdif` still provides the numerically evaluated derivative. This is somewhat deceiving, because the partial derivatives for the similar real valued function $f(x, y) = (x, -y)$ exist and are continous at every point of $\mathbb{R}^2$.
Usually differentiability at any one single point is not an interesting property. If U is some open disk of $\mathbb{C}$ and a complex function f is differentiable at every point of U, f is called *holomorphic* in U. The holomorphity of a complex function, while similar in nature to differentiability of a real valued function, is much more strict a requirement. There is a link between the two, however. If we separate the real and imaginary components of a complex function $f(x + iy) = u(x, y) + iv(x, y)$, in order for f to be holomorphic, the real valued functions u and v must satisfy the partial differential equations

$$\frac{\partial u}{\partial x} = \frac{\partial v}{\partial y} \text{ and } \frac{\partial u}{\partial y} = -\frac{\partial v}{\partial x}.$$

These are called *Cauchy-Riemann equations.* Holomorphism is an important concept in function theory. It will be revisited when complex integration is studied.

4.4 Numerical integration

Integrals are an area of mathematics where numerical solutions are often sought out, because for many functions it is impossible to define an exact integral. Even if it is possible, oftentimes it takes far less work and yields good enough results to make numerical solutions sufficient. The term numerical quadrature, or just quadrature, is more or less synonym for numerical integration.
The basic problem considered by numerical integration is to approximate a solution to a definite integral

$$\int_a^b f(x)dx.$$

First thought would probably be to count function's Riemann sums with sufficiently dense partition, but while Riemann sums provide a good theoretical tool for defining the integrals, in applications the skewed results they provide are usually insufficient.

4.4.1 Trapezoid rule

The idea of partitioning the interval is a useful one, but instead of approximating the function on the short interval in partition by a constant value at either end, like the Riemann sums do, function's values are approximated with a line drawn through the functions values at the endpoints of the interval. This method produces us a number of trapezoids, whose area is easily determined, and the sum of those areas is, depending on the smoothness of the function, and the selected partition, a good approximation of the integral.
Using the knowledge that area of any trapezoid is defined as

$$A = (b-a)\frac{f(a)+f(b)}{2},$$

approximate the defined integral:

$$\int_a^b f(x)dx \approx \frac{1}{2}\sum_{i=2}^{N}(x_i - x_{i-1})(f(x_i) + f(x_{i-1})).$$

In the formula N is the number of intervals studied. It depends on the integrand and the interval $[a, b]$ what the N should be, and should the intervals $[x_{i-1}, x_i]$ be of uniform length or not.

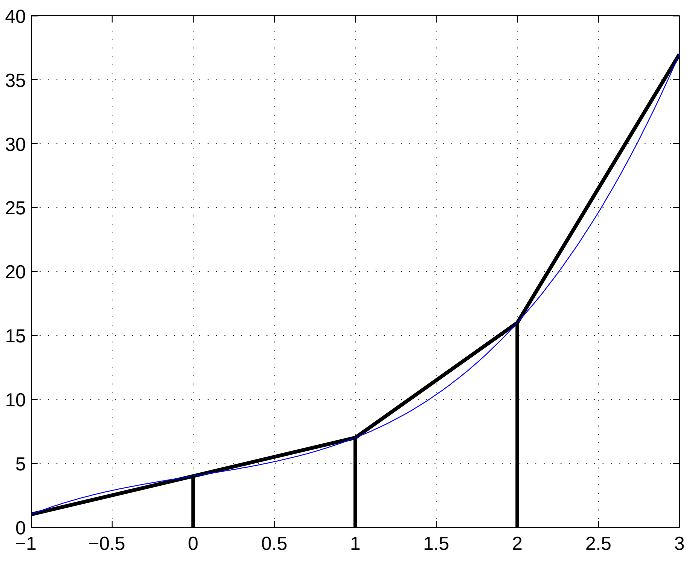


Figure 4.1: Trapezoids drawn on curve $y = x^3 + 2x + 4$.

Trapezoid rule in MATLAB

MATLAB has a built-in function called `trapz`. It takes two vectors as arguments, containing the values x_i and y_i.
Here is another way to implement the trapezoid rule: one that uses uniform interval length, and takes a function as an argument.

Listing 4.5: Algorithm for trapezoid rule

```
function A = trapez(f,a,b,n)
h = (b-a)/n;
A = 0;
for i = 1:n-1
    x = a + h*i;
    A = A + 2*f(x);
end
A = A + f(a) + f(b);
A = 0.5*A*h;
```

The function `trapez` is used like this

```
>>trapez(@sin, 0, pi, 10)
```

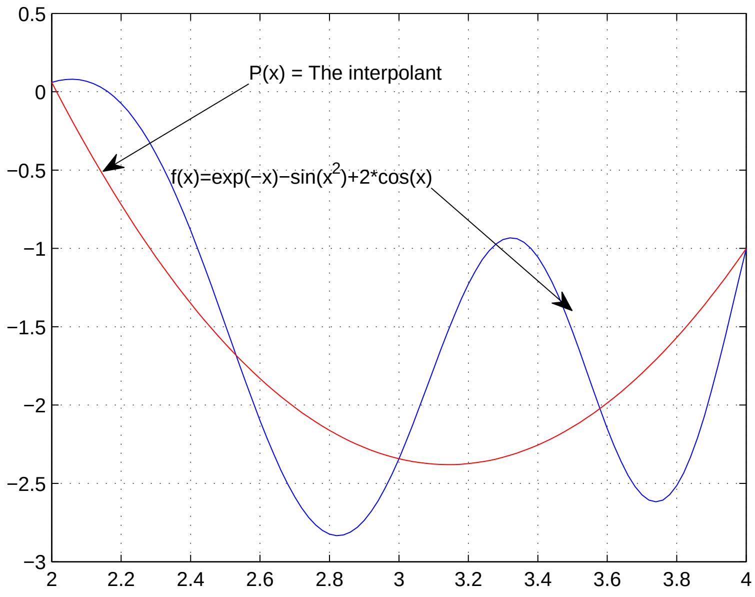
```
ans =
    1.9835
% Since integral of sin from 0 to pi is
% cos(0) - cos(pi) = 2
% this is quite accurate with as few as 10
% intervals..
```

Expanding the formula to more dimensions is not impossible, or difficult, though one should keep track of the quantity the trapezoids present.

4.4.2 Simpson's rule

Simpson's rule is based on interpolation of the integrand function with a quadratic polynomial $P(x)$. The polynomial $P(x)$ takes the same values as integrand at the endpoints a and b, and at the midpoint $m = \frac{a+b}{2}$. Using Lagrange interpolation method, it is discovered, that

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



The interpolant $P(x)$ is polynomial of second degree, and hence easy to integrate:

$$\int_a^b P(x)dx = \frac{b-a}{6}\left(f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right).$$

One should notice, that in order for the Simpson's rule to produce good approximations, the integrand function should be relatively smooth over the interval $[a, b]$; relatively meaning that the quadratic interpolant is accurate some acceptable degree. However, if the integrand function oscillates heavily or it lacks derivatives at some points, or it has some other "bad" property, an accurate interpolation over a long interval may be impossible.
To correct the situation where integrand function behaves badly the usual approach is to break the interval $[a, b]$ into a number of subintervals. The Simpson's rule can then be applied to each subinterval individually, and the sum of these approximations is usually a good approximation of the definite integral over the entire interval.
Suppose that f is the integrand function, and the interval $[a, b]$ is divided into n subintervals, n being an even number. Then the composite Simpson's rule gives

$$\int_a^b f(x)dx \approx \frac{h}{3}\left(f(x_0) + 2\sum_{j=1}^{n/2-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(x_n)\right),$$

where $x_j = a + jh$ for $j = 0 \dots n$ and $h = (b-a)/n$. Here is an implementation in MATLAB code

```
function S = simpson(f,a,b,n)
% f is the name of the integrand,
% a and b define the interval
% n is the desired number of
% subintervals

% Here's the first term of the sum
S = f(a);

n = 2*n;
% make sure n is even

% h is the length of the
% subdivision.
```

```
l = (b - a)./n;
% the uneven additions
for j = 1:2:n-1
    x = a + l*j;
    S = S + 4*f(x);
end
% the even additions
for j = 2:2:n-2
    x = a + l*j;
    S = S + 2*f(x);
end
S = S+f(b);
S = h*S/3;
```

Simpson's rule can be extended to more than one dimensions, but is limited to studying rectangular shapes.

4.4.3 Numerical integration in MATLAB

MATLAB offers a range of built-in functions to numerically calculate definite integrals. Most of them are based on adaptive Simpson's rule, so they can be expected to produce accurate results on functions that are relatively well behaved.
The simplest one to use is the function `quad`. It uses the Simpson's rule to estimate the definite integral of a function of single variable on an interval $[a, b]$. It's variant, `quadl` takes the same parameters, but uses the Lobatto-quadrature instead. The previously mentioned function `trapz` essentially computes the integral using trapezoid rule. Here are a few examples on how to use these functions.

```
% First we set up the integrand functions,
% and the integral functions to observe the
% accuracy of different methods.
f = inline('sin(2*x) + 4*cos(2*x)','x');
g = inline('x.^3 + 2*x -5','x');
F = inline('2*sin(2*x)- 0.5*cos(2*x)','x');
G = inline('0.25*x.^4 + x.^2 - 5*x','x');

% first the trapezoid rule
x = 0:0.2:4;
```

```
s1 = trapz(x,f(x));
s2 = trapz(x,g(x));
%
disp(s1 - (F(4) - F(0)) )
%   prints
%   -0.0341
disp(s2 - (G(4) - G(0)) )
%   prints
%   0.1600

% we then use the quad function
s1 = quad(f,0,4);
s2 = quad(g,0,4);
disp(s1 - (F(4) - F(0)) )
%   prints
%   3.7260e-09
disp(s2 - (G(4) - G(0)) )
%   prints
%   -7.1054e-15

% Finally we observe the quadl - the Lobatto rule
s1 = quadl(f,0,4);
s2 = quadl(g,0,4);
disp(s1 - (F(4) - F(0)) )
%   prints
%   8.5916e-11
disp(s2 - (G(4) - G(0)) )
%   prints
%   -7.1054e-15
```

MATLAB's integration methods are not limited only to functions of single variable: the functions `dblquad` and `triplequad` compute the integrals over rectangular planes and volumes respectively. The syntax they use is `dblquad(f,xmin,xmax,ymin,ymax)`, where f is a function handle to function that takes two parameters, a vector x and a scalar y, and returns values in a vector `V` containing the values of the integrand.

4.4.4 Numerical integration on complex plane

Integral of complex function f is called *complex integral.* It is notated as

$$F(z) = \int_C f(z)dz$$

where C is a path on complex plane. Before discussing what this notation actually means, one needs to define a path and a integral of a complex function over real interval.

Definition 4.5. Let $[a, b]$ be an interval on real line, U be an open subset of $\mathbb{C}$, and let $\gamma : [a, b] \to U$ be continous. Then γ is called a *path.* If $\gamma(a) = \gamma(b)$ path γ is called a *closed path.* For the purposes of complex integrals paths are usually chosen so that they are also differentiable.

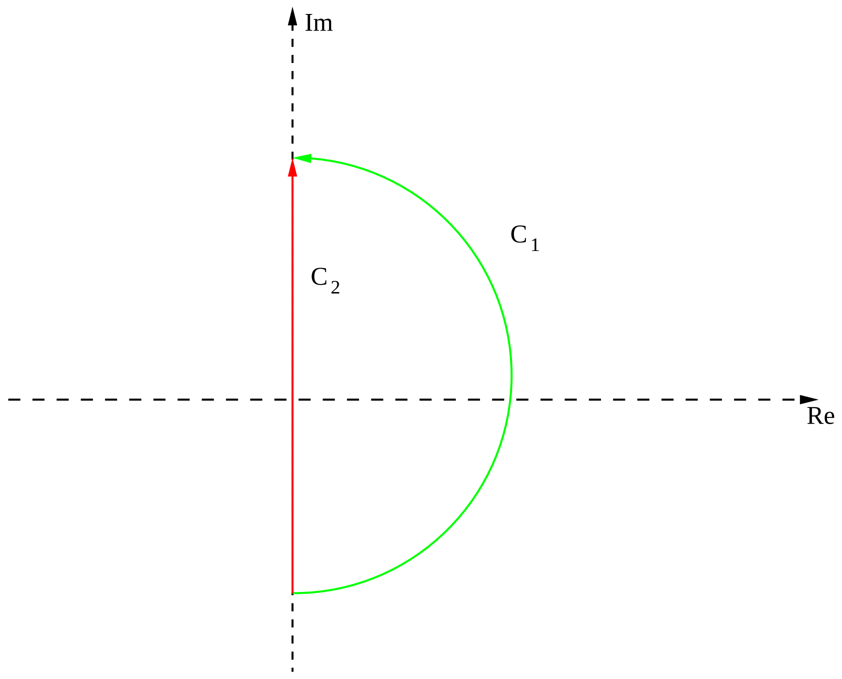
Definition 4.6. Let $[a, b]$ be an interval on real line , and let $f : [a, b] \to \mathbb{C}$ be a continous function $f(t) = v(t) + iu(t)$, where v and u are real valued functions. Integral of function f over the interval

$$\int_a^b f(s)ds = \int_a^b v(s)ds + i\int_a^b u(s)ds.$$

With the two previous definitions we can define a complex integral over a curve C. Let γ map some real interval $[a, b]$ to the path C. Now the complex integral is

$$\int_C f(z)dz = \int_a^b f(\gamma(t))\gamma'(t)dt.$$

Example 4.7. Integrate $f(z) = 2z + 3i$ over paths C_1 and C_2, when $C_1 : [-1, 1] \to \mathbb{C}$, $C_1(x) = ix$, and $C_2 : [-\frac{\pi}{2}, \frac{\pi}{2}] \to \mathbb{C}$, $C_2(x) = \cos(x) + i\sin(x)$.



Derivative of the path $C_1(x) = ix$ is simply i. This means that the complex integral is:

$$\int_{C_1} 2z + 3idz = \int_{-1}^{1} (2is + 3i)ids = \int_{-1}^{1} -2s - 3ds = -6.$$

Derivative of the path $C_2 = \cos(x) + i\sin(x)$ is $C_2'(x) = -\sin(x) + i\cos(x) = i(\cos(x) + i\sin(x))$. Using this yields the complex integral

$$\int_{C_2} 2z + 3i = \int_{\frac{\pi}{2}}^{-\frac{\pi}{2}} (2(\cos(s) + i\sin(s)) + 3i)i(\cos(s) + i\sin(s))ds =$$

$$i\left(\int_{\frac{\pi}{2}}^{-\frac{\pi}{2}} 2(\cos(s) + i\sin(s))^2 ds + \int_{\frac{\pi}{2}}^{-\frac{\pi}{2}} 3i(\cos(s) + i\sin(s))ds\right) =$$

$$i(3i \cdot 2) = -6.$$

The reason the two complex integrals yield the same result is that the integrand function, $f(z) = 2z + 3i$ is holomorphic. It also means, that it is path independent: the value of the complex integral does not depend on the selected path, provided that the function is holomorphic on the entire path. Observe now a complex integral of $g(z) = |z|$ along the paths

C_1 and C_2. We use Euler's formula to make the C_2 more manageable: $C_2(x) = \cos(x) + i\sin(x) = e^{ix}$, and $C_2'(x) = ie^{ix}$.

$$\int_{C_1} g(z)dz = \int_{-1}^{1} |is|i \quad ds = i,$$

while

$$\int_{C_2} g(z) = \int_{\frac{\pi}{2}}^{-\frac{\pi}{2}} |e^{is}|ie^{is}ds = \int_{\frac{\pi}{2}}^{-\frac{\pi}{2}} ie^{is} = 2i.$$

Because g is not holomorphic, the integrals along different paths differ.

Computing complex integrals in numerically does not differ greatly from real integrals: essentially the idea of dividing the interval and summing the trapezoids works in complex case as well.

```
>> f = inline('2*z+3*i','z')
f =
      Inline function:
      f(z) = 2*z+3*i
% function trapez is the same one
% defined in the section on trapezoid
% integrals
>> trapez(f,-i,i,100)
ans =
     -6
% MATLAB's own integral tool has no
% problems either
>> quad(f,-i,i)
ans =
     -6
```

Numerically the real and complex integrals do not differ, when dealing with holomorphic functions. When the only the endpoints of the integral path matter, the integral can always be evaluated along the straight line from the beginning of the path to the end of the path. However, if the integrand function is not holomorphic, this is not the case, as was seen in the previous example. In these cases it is necessary use the definition to compute the integral.

```
>> quad(@abs,-i,i)
ans =
        0 + 1.0000i
>> g=inline('abs(cos(x)+i*sin(x))*i.*(cos(x)+i*sin(x))')
>> quad(g,-0.5*pi,0.5*pi)
ans =
        0 + 2.0000i
```

4.4.5 More advanced integration methods

Boole's rule

Boole's rule approximates the integral

$$\int_{x_1}^{x_5} f(x)dx$$

by computing values of f at five equally spaced points, so that $x_k = x_1 + (k-1)h$ and $h = \frac{x_5 - x_1}{4}$. In the *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, the estimate is expressed as:

$$\int_{x_1}^{x_5} = \frac{2h}{45}(7f(x_1) + 32f(x_2) + 12f(x_3) + 32f(x_4) + 7f(x_5)) + \text{error}.$$

The error term is:

$$-\frac{8}{945}h^7 f^{(6)}(c)$$

where $c \in [x_1, x_5]$. Here is an example of Boole's method implemented in MATLAB.

```
function I = boole(f,a,b)
h = (b-a)/4;
S = [f(a) f(a+h) f(a+2*h) f(a+3*h) f(a+4*h)];
I = 2*h/45*(7*S(1)+32*S(2)+12*S(3)+32*S(4)+7*S(5));
```

Because the only factor that can be affected in the error term is the length of the integration interval, it might be a good idea to adapt the method for intervals too long. Here is an example on how to implement the adaptation. A word of warning though: this example makes use of recursion. Recursion, as a rule, is extremely resource consuming, and should be avoided at all costs.

```
function S = boole_rec(f,a,b)
if(abs(a-b)<0.5)
    S = boole(f,a,b);
    return;
end
middle = (a+b)/2;
S = boole_rec(f,a,middle) + boole_rec(f,middle,b);
```

Romberg's method

Romberg's method creates a triangular array consisting of numerical estimates of the definite integral it approximates. It applies Richardson extrapolation continuously on the trapezoid rule, until desired accuracy is achieved. The method can be defined inductively:

$$\begin{array}{lcl} R(0,0) & = & \frac{1}{2}(b-a)(f(a)+f(b)) \\ R(n,0) & = & \frac{1}{2}R(n-1,0)+h_n\sum_{k=1}^{2^{n-1}} f(a+(2k-1)h_n) \\ R(n,m) & = & R(n,m-1)+\frac{1}{4^m-1}\Big(R(n,m-1)-R(n-1,m-1)\Big) \end{array}$$

where $h_n = \frac{a+b}{2^n}$.
With n and m sufficiently large,

$$\int_a^b f(x)dx \approx R(m,n)$$

with the maximal error estimate for the $R(m,n)$ being $O(h_n^{2m+2})$.
First column of this triangular array, that is, values $R(i,0), i = 0 \dots n$, are the trapezoidal integrals calculated with 2^n+1 points. The first extrapolation is equivalent to the integral approximation using the Simpson's rule with 2^n+1 points.
As far as computation is concerned, the recursive calls within the loops are not efficient at all. A better solution is to table the values, and update the table as we move along the algorithm. While some small inefficiency is suffered by not being able to preallocate the matrix **R**, it is a small price to pay for avoiding the deep recursions that would otherwise be necessary.
Here is an example implementation of Romberg integral in MATLAB.

```
function q = romb(f,a, b, tol)

% Approximates the integral from a to b of f(x)dx
% to tolerance of tol by using the trapezoidal
% rule with repeated Richardson
% extrapolation.

%  Make first estimate using one interval.

n = 1;  h = b-a;
fval = [f(a); f(b)];
R(1,1) = .5*h*(fval(1)+fval(2));
% Keep doubling the number of subintervals until
% desired tolerance is achieved or max no. of
%subintervals (2^10 = 1024) is reached.
% The array R will hold the triangular
%array of estimates for F(b)-F(a)

err = tol+80;
% Initialize err to something > tol.
disp('          q            error est')

s = 0;
while err > tol && s < 10,
  s = s+1;  n = 2*n;  h = h/2;
  fvalnew = zeros(n+1,1);
% Store computed values of f to reuse
% when h is cut in half. We preallocate
% for speed.
   for i=1:2:n+1
    fvalnew(i) = fval((i-1)/2 + 1);
  end;
% Compute f at midpoints of previous intervals
  for i=2:2:n
    fvalnew(i) = f(a+(i-1)*h);
  end;

  fval = fvalnew;
  trap = .5*(fval(1)+fval(n+1));
  for i=2:n
    trap = trap + fval(i);
  end;
% Use trapezoidal rule with new h value
% to estimate integral. fval holds the
% endpoints of
```

```
  R(s+1,1) = h*trap;
% Store new estimate in first column of tableau.

%    Perform Richardson extrapolations.
%    That is, we fill the slots R(s,2) to R(s,s+1)
  for j=2:s+1,
   R(s+1,j)=((4.^(j-1))*R(s+1,j-1)-R(s,j-1))/(4.^(j-1)-1);
  end;
  q = R(s+1,s+1);

% Estimate error.  This is usually an overestimate of
% the error in q.
% It is a more appropriate approximation for the
% error at previous stage.
% The error will decrease, as either n or m grows.
  err = max([abs(q-R(s,s)); abs(q-R(s+1,s))]);

% Print out approximation to integral and error at each
% step, for monitoring convergence. For industrious
% use, comment away. (Print is a costly operation)
  disp([q err])

end;
```

Monte Carlo - methods

Monte Carlo methods form a class of computational algorithms, that rely on evaluating repeated random samples to compute an approximate result to the given problem. Because of their reliance on large number of pseudo-random numbers, they are almost uniformly suited for computers, and tend to be used when acquiring the deterministic solution is impossible or unfeasible. The term itself was invented in Los Alamos National Laboratory by physicists working on nuclear weapon project during the second world war.
Our interest in Monte Carlo methods concern numerical integration. The previously presented methods of numerical integration are based on taking a number of evenly spaced sample points, and determining the quadrature. However, there are cases when computing the definite integral in some deterministic way, even numerically, may turn out to be too difficult. In these cases Monte Carlo integration method may prove to be a good choice.

Informally, the idea of Monte Carlo integration is to approximate definite integral over domain D, by picking a simple domain E, whose area is easily determined, and which contains C. Random points are then selected in E, knowing that some of these will also fall in C. The estimate for the integral D is the area of E multiplied by the fraction of random samples in D.

$$\int_D f(x)dx \approx \text{area}(E)\frac{n}{N}$$

where n is the number of random samples that fell within D and N is the total number of random samples. As the number N grows, the approximation converges towards the definite integral.
Monte Carlo integration methods are very well suited to situations, when there is little or no mathematical structure behind the integrand: for example integration of a noisy experimental data. For this reason Monte Carlo methods are eminently used in computational physics, while in other areas of mathematics deterministic methods are used.
In order for Monte Carlo- integration to produce good results, the method for producing random points must be selected with care; traditionally the random points are uniformly distributed over the domain E, though other methods have been suggested to decrease the error.

Example 4.8. In this example Monte Carlo integration is used to estimate the volume of a cube with a radius of one. To do this, take a number of random samples from $[-1, 1] \times [-1, 1] \times [-1, 1]$, and perform the evaluation.

```
% we estimate the volume of a sphere
% with radius of one
% using Monte Carlo integration

% we get n random triples from cube
% [-1,1]x[-1,1]x[-1,1]
% it contains the sphere with radius
% of one
n=input('How many random samples do you want?');
hit = 0;
for i = 0:n
    a = 2*rand(1)-1;
    b = 2*rand(1)-1;
    c = 2*rand(1)-1;
    if(a^2+b^2+c^2 <= 1)
        hit = hit + 1;
    end
end
disp('Estimated volume')
disp(8*hit/n);
disp('Real volume')
disp(4/3*pi);
```

Testing shows, that 3000 samples seems to produce quite good results, with error of magnitude of 10^{-4}.

Gaussian quadrature

The concept of orthogonal functions, first defined in theory of vector spaces, gives us a useful tool to approximate a definite integral numerically.

Definition 4.9. First, define a vector space with continuous functions defined on the interval $[a, b]$. Let f and g be such functions, and let W be a third function, a *weight function.* Define then the inner product for functions f and g with

$$\langle f, g\rangle = \int_a^b W(x)f(x)g(x)dx.$$

If $\langle f, g\rangle = 0$, the functions are orthogonal. If $\langle f, f\rangle = 1$,, f is said to be normalized. If every function in a set of normalized functions is orthogonal with each other, it is said to be orthonormal.

Using the previous definition, one can create a set of polynomials having exactly one polynomial $p_j(x)$ of the degree $j, j = 0, 1, 2, \ldots$.
The construction is as follows. First set

$$p_{-1}(x) \equiv 0, p_0(x) \equiv 1,$$

then

$$p_{j+1}(x) = (x - a_j)p_j(x) - b_j p_{j-1}(x)$$

where

$$a_j = \frac{\langle xp_j, p_j\rangle}{\langle p_j, p_j\rangle}, b_j = \frac{\langle p_j, p_j\rangle}{\langle p_{j-1}, p_{j-1}\rangle}, j = 1, 2, 3\ldots$$

The factor b_0 can be selected arbitrarily, usual choice is zero.

Now, when approximating definite integral

$$\int_a^b W(x)f(x)dx \approx \sum_{j=1}^{N} w_j f(x_j)$$

one can select the weights w_j and abscissas x_j so, that the formula holds as equivalence for all polynomials of at most $2N-1$ degree; the evaluation points are the roots of the orthogonal polynomials, constructed as shown before. The weights depend on the polynomials as well.
One of the most commonly used set of polynomials are the Legendre polynomials $P_n(x)$. It can be defined as a contour integral

$$P_n(z) = \frac{1}{2\pi i}\oint (1-2tz+t^2)^{-1/2}t^{-n-1}dt.$$

The contour should enclose the origin, and no other singular points, and it is traversed counterclockwise.
Scale the integrand function to interval $[-1,1]$, and select the weight function $W(x) \equiv 1$. The evaluation points, or Gauss nodes, x_i will then be the i:th root of the P_n, where n is the degree of the Legendre polynomial to be applied. The weights w_i will be

$$w_i = \frac{2}{(1-x_i^2)(P_n'(x_i)^2)}.$$

Here is an example code for Gaussian quadrature using Legendre polynomials.

```
function I = gauss_quad2(f,a,b,n)
I = 0;
% The function must be scaled to
% [-1 1]. wp and ws are scaling
% weights.
wp = (b-a)/2;
ws = (a+b)/2;
% Find the abscissas
R = legroots(n);
% define a step for derivation
h = sqrt(eps);
for i= 1:length(R)
    r = R(i);
    % Built in Legendre function:
    % subsequent rows represent
    % increasing order: the first
    % row is the 0th order Legendre
    % function, i.e. Leg. polynomial.
    y = legendre(n,[r-h r+h]);
    dy = (y(1,2)-y(1,1))/(2*h);
    % Determine the weights
    w = 2/( (1-r.^2) * (dy.^2) );
    I = I + f(r*wp+ws)*w;
end

function r = legroots(N)
% The function r = legroots(N) computes the roots of the
% Legendre polynomial of degree N. For the purposes of
% this course, just have faith that it does what it
% promises.
n = 1:N-1;                          %  Indices
d = n./sqrt(4*n.^2-1);              %  Create subdiagonals
J = diag(d,1)+diag(d,-1);           %  Create Jacobi matrix
r = eig(J);                         %  Compute eigenvalues
```

The Gaussian method can be made more accurate by increasing the degree of Legendre polynomial, or by selecting a different set of orthogonal polynomials and weight function altogether. In case of the latter, common choices include weight function $\frac{1}{\sqrt{1-x^2}}$ with Chebysev polynomials, and e^{-x} with Laguerre polynomials.

4.5 Symbolic differential and integral calculus

While MATLAB is designed to be primarily a tool for numerical computing, since 2008 MATLAB symbolic math toolkit has included the MuPAD computer algebra system, capable of performing symbolic computations. It is somewhat inferior to its more famous competitors, Maple and Mathematica, but it provides a good enough foundation on which to perform symbolic operations.
If used from MATLAB command line, the MuPAD functionality is accessed through defining a variable symbolic with the command `sym`. After declaring a variable symbolic it does not hold a numeric value, like variable usually

would. It is now considered a symbol, and all operations performed on it are now done through the MuPAD kernel, rather than MATLAB. Here is an example:

```
% First we define two symbolic variables, x and a.
>> x = sym('x');
>> a = sym('a');
% We now test the arithmetics
>> a * x*a

ans =

a^2*x
>> a + a + a +a + a

ans =

5*a
% Numbers can also be given symbolic
% representation.
>> sym(11)/sym(22)

ans =

1/2
```

In addition to the basic operators, the symbolic toolbox offers a wide variety of different operators. In this section we, however concentrate on those that have to do with basic calculus, starting with the obvious ones: derivation and integration. Symbolic operators `diff` and `int` perform the derivation, or integration, if possible. They must be given a symbolic expression as a parameter in order for them to work. Here are examples.

```
% First define a symbolic variable.
>> x=sym('x');
% Then define a symbolic function:
% it's only parameters are symbolic
% variables.
>> t = 8*x^3 + 15*x^2 - 56*x + 8;
% We integrate the polynomial with
% respect to x
>> int(t,x)
```

```
ans =

2*x^4+5*x^3-28*x^2+8*x
% We then derivate in respect to x
>> diff(t,x)

ans =

24*x^2+30*x-56
% Then something more complex
>> t = 1/(1+x^2);

>> int(t,x)

ans =

atan(x)
% Integration over areas works also
>> y = sym('y');
>> t = x^2+y^2;
>> int(t,x,y)

ans =

1/3*y^3-1/3*x^3+y^2*(y-x)

% partial derivation works also...
>> diff(t,x)

ans =

2*x
% so do second derivatives
>> diff(diff(t,x),x)

ans =

2
```

```
% and finally the gradients
>> h = x^3+ 4*y;
>> A=[t h];
>> diff(A,x)

ans =

[   2*x, 3*x^2]
```

In addition to integration and derivation operators, MuPAD offers tools to observe limits of functions, convergence of series, and finally, to find Jacobian matrices and Taylor series for given functions. Here are examples.

```
% We start with simple limit:
% the value of Napier's constant e
>> n = sym('n');
>> s = (1+1/n)^n
% The limit defaults to
% 0 if no value is given
>> limit(s, n, inf)

ans =

exp(1)
% Then another limit, this time at 0
>> x = sym('x');
>> f = sin(x)/x;
>> limit(f)

ans =

1

% Now we shall attempt to find a Taylor series
%for a complicated function at x_0 = 0;
% the command syms is shorthand for creating
% lists of symbolic variables.
>> syms x y
>> f = sin(x)*x + exp(x) + 8
>> taylor(f)
```

```
ans =

9+x+3/2*x^2+1/6*x^3-1/8*x^4+1/120*x^5
% Without specifications the function
% taylor finds the Taylor polynomial
% at 0, and computes five first terms
>> taylor(exp(-x),3,6)

ans =

exp(-6)-exp(-6)*(x-6)+1/2*exp(-6)*(x-6)^2
% Here we specified that we want the first
% three terms computed at x_0 = 6

% Finally we take a look at the symbolic
% Jacobian matrix
>> f = [x^2+y*x; x*y+x; exp(x+y)];
>> jacobian(f,[x y])
ans =

[    2*x+y,         x]
[      y+1,         x]
[ exp(y+x), exp(y+x)]
```

Chapter 5

Nonlinear equations

In previous chapters different methods of solving systems of linear equations were studied. Now more general types of equations are studied. Generally, object is to find a vector $\mathbf{x} = (x_1, \ldots x_n), \mathbf{x} \in \mathbb{R}^n$ that satisfies the system of equations

$$\begin{cases} f_1(\mathbf{x}) & = b_1 \\ & \vdots \\ f_n(\mathbf{x}) & = b_n \end{cases}, \tag{5.1}$$

where the functions f_j are non-linear. If the vector $\mathbf{x}$ satisfies the system of equations, it is called *root*. The methods that were available for solving linear systems of equations are no longer generally valid, and one must find other methods of solutions.

Before trying to seek exact solutions to a non-linear system of equations, you must make sure the solution exists. In case of linear algebra this was easily gleaned from theorems of linear algebra, in non-linear case there is no single way of determining the existence of a solution.

There is also no general algorithm of solving a system of non-linear equations, if there are more than on equation. In case of just one equation, the bracketing method is general, since it requires knowledge only about the values of function. For a system of equations, there are algorithms, that work, if some fairly light assumptions can be made about the functions f_j. In order for several of these algorithms to work, somewhat accurate initial guess is required.

Most of the algorithms to find the root of 5.1 are based on iterative methods. Since it is usually impossible to numerically find the exact root, one needs to have some preset condition to halt the iteration once the desired accuracy is achieved. It should also be noted, that the iterations do not always converge toward the root, and to avoid infinite loops, a halting condition should be set for this eventuality as well.

5.1 Root finding algorithms

5.1.1 Bracketing

Bracketing, or bisection method, is a very general algorithm for discovering the roots of a function of one variable. Only thing it requires is, that there exist an interval $[a, b]$, where the function is continuous, and that the function changes sign on the interval, i.e. $f(a)f(b) < 0$. Bracketing makes use of the intermediate value theorem, which says, that a function f is continuous on the interval $[a, b]$, it gets at least all the values $[f(a)f(b)]$. Should the $f(a)$ and $f(b)$ have different signs, it implies that there is a value $c, a \leq c \leq b$, so that $f(c) = 0$. The basic idea of the bracketing is this: first check that interval endpoints have different signs. Then evaluate the function at the midpoint $m = \frac{b-a}{2}$. If the $f(m) = 0$ or numerically close enough, stop the algorithm and return m. If not check the signs of $f(m)f(a)$ and $f(m)f(b)$. If $f(m)f(a)$ is positive, it is known that the root lies on the interval $[m, b]$, and if it is not, it's known that the root lies on the interval $[a, m]$. Then select the appropriate interval, and repeat the iteration, and keep repeating it until you reach the root.
Bracketing is very robust algorithm: it produces good results and does not require complex procedures to acquire the root. It is not without its weaknesses, however. As a rule, the bracketing method converges slowly when compared to other root finding methods. Also, it finds only one root; and only that root. Finding other roots requires a priori knowledge where the roots lie, or adaptive implementation of the algorithm. Here is an example implementation of bracketing in MATLAB.

```
function x0 = bracket(f,xmin,xmax)
% finds a root of the function f on the
% interval xmax, xmin. f should change its
% sign on this interval at least once
if(f(xmax)*f(xmin)>0)
    error('Positive or negative endpoints');
```

```
end
m= (xmax-xmin)/2;
m = xmin+m;
while abs(f(m))>1e-8
    disp(f(m));
    if(f(m)*f(xmin)>0)
        xmin =m;
    else
        xmax = m;
    end
    m = (xmax-xmin)/2;
    m = xmin + m;
end
x0=m;
```

5.1.2 Fixed point iteration

The iterative methods to solve the system 5.1 are almost uniformly based on the fixed points of function. The point x is said to be a fixed point of function f, if $x = f(\mathbf{x})$. The idea is to write the iteration in the form

$$x_{k+1} = f(x_k).$$

The x is not restricted into being a real or complex number: it can be a vector, or even a function. If the sequence (x_k) converges towards some value x_0, and the function f is continuous, it holds that

$$x_0 = f(x_0).$$

This method for finding the root of equation $x_0 = f(x_0)$ is called *fixed point iteration.* Next sufficient and necessary properties for function f to have in order for the sequence (x_k) to converge are studied

Banach's fixed point theorem

In 1922 a polish mathematician named Stefan Banach proved a theorem that stipulates when a function has fixed points, and guarantees that they are unique. He presented his theorem for metric spaces, which allows the fixed point iteration to be used in not only real- and complex spaces, but, for example, in the space defined by continuous functions on some interval.

Definition 5.2. Let B be a complete vector space with scalar field $\mathbb{C}$. B is a *Banach space*, if it has a norm $||\cdot||$ so that

1. $||x|| \geq 0 \quad \forall x \in B$,
2. $||x|| = 0 \Leftrightarrow x = 0$,
3. $||\gamma x|| = |\gamma| ||x||, \quad \forall \gamma \in \mathbb{C}, \forall x \in B$,
4. $||x + y|| \leq ||x|| + ||y|| \quad \forall x, y \in B$.

Definition 5.3. Let (X, d_X) and (Y, d_Y) be metric spaces. The function

$$f : X \rightarrow Y$$

is called *Lipschitz-continuous*, if there exists a real constant $K \geq$ so that for all $x_1, x_2 \in X$

$$d_Y(f(x_1), f(x_2)) \leq K d_X(x_1, x_2).$$

If $0 < K < 1$, the function f is called *contraction*.

Theorem 5.4. *(Banach's fixed point theorem). Let A be a closed subset of Banach space B, and let the function f be Lipschitz continuous contraction. Then the function f admits one, and only one fixed point x_0. Furthermore, the iterative sequence $x_n = F(x_{n-1})$ converges to x_0 regardless of the selection of the initial point.*

Proof. First remember, that B is a complete vector space and hence, every Cauchy sequence converges, and that the sequence (x_k) is a Cauchy-sequence if for every $\epsilon > 0$ there exists a number n_ϵ so that

$$||x_m - x_n|| < \epsilon, \text{ when } m, n \geq n_\epsilon.$$

The fixed point iteration $x_k = f(x_{l-1})$ gives us:

$$||x_{k+1} - x_k|| = ||f(k) - f(x_{k-1})|| \leq K||x_k - x_{k-1}|| =$$

$$K||f(x_{k-1}) - f(x_{k-2})|| \leq K^2||x_{k-1} - x_{k-2} = \dots$$

this gives us inductively

$$||x_{k+1} - x_k|| \leq K^{k-l}||x_{l+1} - x_l||.$$

Then show that (x_k) is a Cauchy-sequence:

$$||x_{k+m} - x_k|| = ||x_{k+m} - x_{k+m-1} + x_{k+m-1} - \ldots - x_k||$$

$$\leq \sum_{j=k}^{k+m-1} ||x_j|| \leq K^k(K^{m-1} + K^{m-2} + \ldots + K + 1)||x_1 - x_0||$$

$$= K^k \frac{1 - K^m}{1 - K} ||x1 - x0||.$$

Because $0 < K < 1$, (x_k) is a Cauchy-sequence. Therefore the closed subset A contains the limiting value

$$s = \lim x_k, \quad s \in A.$$

Furthermore $f(s) = f(\lim x_k) = \lim f(x_k) = \lim x_{k+1} = s$, so s is a fixed point of f. Show then, that this fixed point is unique through counter assumption: suppose that s_1 and s_2 are fixed points of f, and $||s_1 - s_2|| > 0$, we get

$$||s_1 - s_2|| = ||f(s_1) - f(s_2)|| \leq K||s_1 - s_2||$$

which leads to situation $K \geq 1$, which contradicts the supposition that the function f is a contraction. Therefore $s_1 = s_2$, and fixed points are unique. $\square$

You can now use the fixed point iteration to solve equations of the form $f(\mathbf{x}) = \mathbf{x}$, if the function f satisfies the required conditions. Checking the contraction-condition is can be simplified in Euclidean spaces: function f is a contraction, if $|f'(x)| < c, c \in (0, 1)$.
Here is a simple implementation of how fixed point iteration could be implemented in MATLAB.

```
function fp = banach(f)
% f is assumed to be a function
% of single vector variable.
% A fixed point of f is
% returned if it was found
% in less than 100 iterations.
% Otherwise 0 is returned.
fp = 0;
ctr = 0;
```

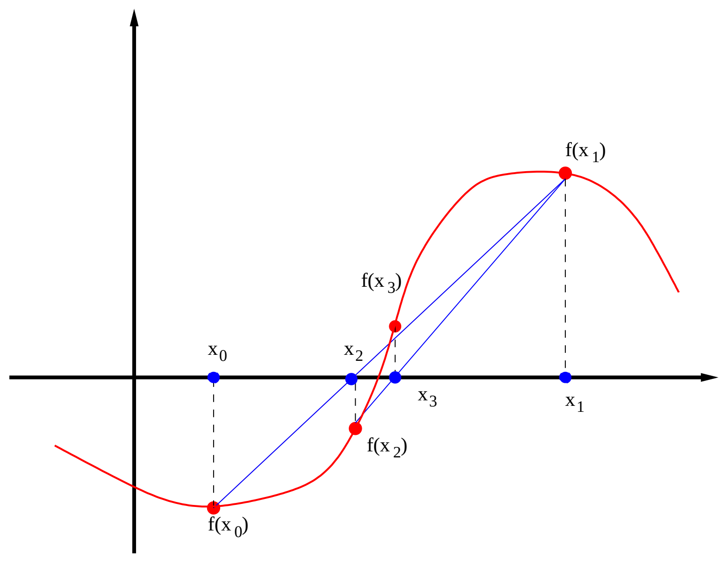
```
h = sqrt(eps);
% We make an elementary check of
% Lipschitz property
if((f(fp+h)-f(fp-h)/2*h)>1)
    error('Not a contracting function')
end
while((abs(fp-f(fp))>1e-8) && (ctr<100))
    fp = f(fp);
end
```

5.1.3 Secant method

Secant method uses sequence of secant lines drawn to the graph of the studied function. Roots of these lines will, given good enough an initial guess, converge towards the root of the function. Good enough guess means, that one must have knowledge, that a root exists on some interval (a, b).
Secant line of a curve is a line that locally intersects with curve at two different points. Secant method uses the line that is drawn to intersect the curve of the function at the points of initial guess ass interpolant for the function on this interval. It then makes a new estimate on a new interval, using the root of the secant line as a new endpoint. Here is the recurrence formula for the secant method:

$$x_n = x_{n-1} - f(x_{n-1})\frac{x_{n-1} - x_{n-2}}{f(x_{n-1}) - f(x_{n-1})}.$$

The two values, x_0 and x_1, required for the first recursion are the initial guess, and ideally should lie close to the root.



Secant method, when it converges, is somewhat slow, but usually better than bracketing method. There are, however, cases when bracketing will prove to be more efficient: especially if a smooth function's second derivative changes sign near the root. Secant method can be extended to more than on dimensions: it is then call *Broyden's method*. Here is an example MATLAB implementation of the secant method in one dimension.

```
function x = secant(f,x0,x1)
% Secant method for MATLAB
% parameter f is a function
% handle or inline function.
% x0 and x1 are the initial
% guess points.
xold = x0;
xnew = x1;
ctr =0;
% initialize xnew as x0
% for convenience purposes
% We set up a halting conditions
% both for finding the root
% and for the case that
% the series (x_n) does not
% converge.
while(abs(f(xnew))>1e-8)
    aux = xnew;
    xnew = xnew - f(xnew)*((xnew-xold)/(f(xnew)-f(xold)));
    xold = aux;
    ctr = ctr+1;
end
x = xnew;
```

One should note, that the algorithm makes no suppositions for the initial values: the root does not have to lie between them. This means, that the method does not necessarily converge at all. Next an algorithm is presented that requires the root to be bracketed between the interval's endpoints.

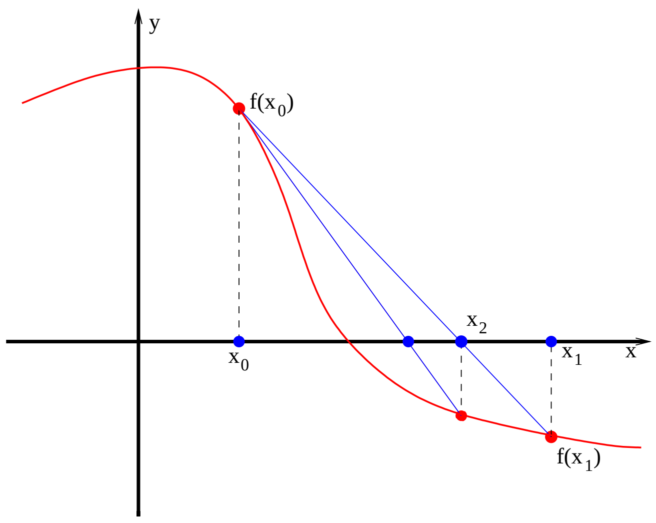
5.1.4 False position method

False position method (sometimes called Regula Falsi-method) combines bracketing and secant methods. It begins as the bracketing method does: by selecting an interval, with function values at interval ends having opposite signs. Then, instead of choosing the midpoint for new interval endpoint, choose the root of secant line drawn at these two points, and then choosing the new interval so, that the function values at the endpoints have different signs. Formally:

$$c_k = \frac{f(b_k)a_k - f(a_k)b_k}{f(b_k) - f(a_k)}$$

$$\begin{cases} a_{k+1} = c_k \text{ if } f(c_k)f(a_k) < 0 \\ b_{k+1} = c_k \text{ if } f(c_k)f(b_k) < 0 \end{cases} .$$

In case the studied function is continuous and the initial condition $f(a_0)f(b_0) < 0$ holds, one will always find a root with this method. This method is generally faster than bracketing, but as with secant method, there are cases when finding the functions roots requires many iterations.



Here is a MATLAB implementation of method of false position.

```
function x = regfalsi(f,a,b)

if(f(a)*f(b)>0)
    error('no sign change on interval');
end
xnew = a;
xold = b;
ctr = 0;
while(abs(f(xnew))>1e-8 && ctr<100)
    c = (f(xold)*xnew-f(xnew)*xold)/(f(xold)-f(xnew));
    if(f(xnew)*f(c)<0)
        xold = xnew;
        xnew = c;
    else
        xnew = c;
    end
end
x = xnew;
```

5.1.5 Newton's method

Probably the most famous method for finding roots of a function is the Newton's method, named after sir Isaac Newton. The method will find successively better approximations for roots of a real valued function using tangent

lines fitted to the function. Newton's method requires that the studied function is differentiable.
The idea behind the method is to use approximation gained by calculating the functions Taylor series:

$$T(f;x_0) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2 + \dots$$

at point $x_0 + \epsilon$. Obtain

$$f(x_0 + \epsilon) = f(x_0) + f'(x_0)\epsilon + \frac{f''(x_0)}{2}\epsilon^2 \dots$$

When ϵ is very small, one can approximate the function value by keeping terms only to the first order:

$$f(x_0 + \epsilon) \approx f(x_0) + f'(x_0)\epsilon.$$

If you now set $f(x + \epsilon) = 0$, and use the previous approximation to compute the ϵ, you get:

$$\epsilon_0 = -\frac{f(x_0)}{f'(x_0)}.$$

One can see, that the approximation is the equation of the tangent line of the function f at the point $(x_0, f(x_0))$. It intercepts the x-axis at point $(x_1, 0)$. Set now, that $x_1 = x_0 + \epsilon_0$. This gives an idea for an algorithmic approach for finding a root: set

$$\epsilon_n = -\frac{f(x_n)}{f'(x_n)}$$

and calculate x_n by

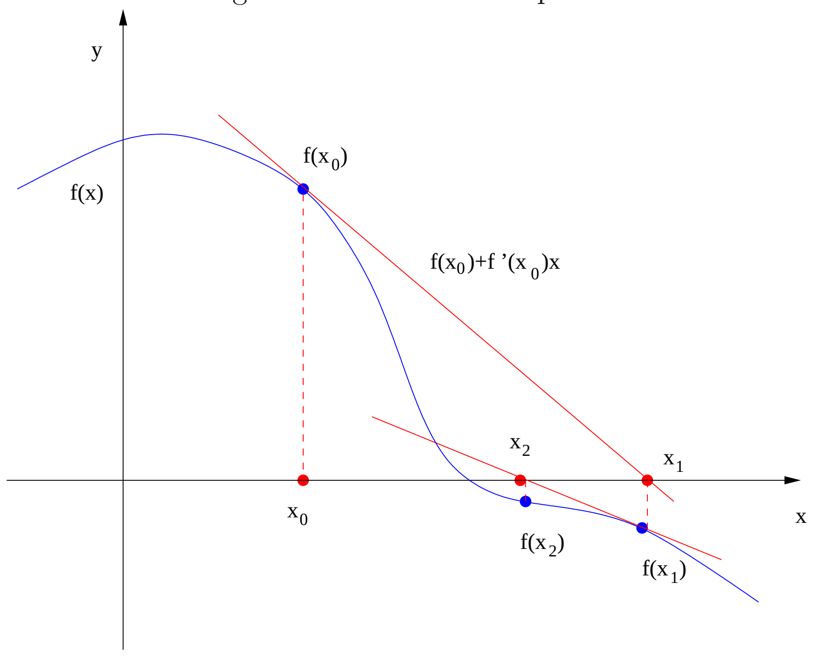
$$x_n = x_{n-1} - \epsilon_n.$$

If the obtained sequence (x_n) converges, it converges towards a fixed point, which is precisely the root. This gives us the traditional formula for Newton's iteration:

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

Whether the sequence given by Newton's iteration converges is a complicated question; however for the purposes of this course it is enough to say, that in a sufficiently small neighborhood of a simple root of a twice differentiable function, Newton's method converges quadratically to that root.

Figure 5.1: First two steps of Newton iteration



Another interesting question is that if a function f has more than one root, which one will it converge towards, if it converges at all. The answer is somewhat unexpected: on complex plane roots of functions with more than two roots yield a rational map of $\mathbb{C}$, and the Julia set of this map is a fractal, or to put it more poetically: this is a manifestation of chaos.
Here is an example MATLAB implementation of Newton's method in single dimension.

```
function root = mynewt(f,x0)
% f is the function we whose
% roots we wish to find, x0 is
% the initial guess.
% mynewt uses the numdif function
% that was introduced in the
% numerical calculus section.
% If a suitable solution is
% not found in 100 iterations
% attempt is abandoned.
```

```
ctr =0;
x =x0;
h = sqrt(eps);

while(abs(f(x))>1e-8 && ctr<100)
    df = numdif(f,x,h);
    x = x - (f(x)/df);
    ctr = ctr+1;
end
root = x;
```

Newton's method can be generalized for vector functions $F : \mathbb{R}^n \rightarrow \mathbb{R}^n$ by substituting the the functions derivative by Jacobian matrix of the function. This puts somewhat more requirements for the function, as the Jacobian matrix must be invertible at the evaluation points, and as we remember from the calculus section, this means the function must have an inverse in some small environment near the evaluation point.
Searching for the root of the function F is analogous to solving a system of equations

$$\begin{cases} f_1(x_1 \dots x_n) & = 0 \\ f_2(x_1 \dots x_n) & = 0 \\ \vdots & \vdots \;\; \vdots \\ f_n(x_1 \dots x_n) & = 0 \end{cases}.$$

Assuming that the function $F = (f_1(\mathbf{x}), \dots f_n(\mathbf{x}))^T, \mathbf{x} = (x_1 \dots x_n)$ is differentiable, following holds:

$$F(\mathbf{x}_0 + \delta) \approx F(x_0) + J_f(\mathbf{x}_0)\delta,$$

where $J_F(x_0)$ is the Jacobian matrix of F evaluated at $\mathbf{x}_0$.
As in one dimensional case, using successive linearisation approximations for the function F yield:

$$0 \approx F(\mathbf{x}_{n+1}) \approx F(\mathbf{x}_n) + J_F(\mathbf{x}_n)(x_{n+1} - \mathbf{x}_n).$$

This gives us the Newton-iteration step:

$$\mathbf{x}_{n+1} = \mathbf{x}_n - J_F(\mathbf{x}_n)^{-1}F(\mathbf{x}_n).$$

If the initial guess $\mathbf{x}_0$ is located close enough to the root, the sequence $(\mathbf{x}_n)$ converges to the root.

Here is an one MATLAB implementation of the Newton's method in multiple dimensions.

```
function root = vectorNewton(f,x0)
% f is a inline function or a function handle.
% f should an nx1 vector as a parameter, and
% it should return an nx1 vector.
x = x0;
ctr = 0;
while((abs(norm(f(x)))>1e-8) && (ctr<100))
    jf  = jacob(f,x,length(x0), length(x0));
    x = jf\(jf*x-f(x));
    ctr = ctr +1;
end
root = x;

function Jf = jacob(f,x,m,n)
% f is a function with m components,
% x is a vector with n components,
% the result is an m by n matrix.
Jf = ones(m,n);   h = 1e-4;
for j =1:n
    e = zeros(n,1); e(j) = 1;
    Jf(:,j) = (feval(f,x+h*e)-feval(f,x-h*e) )/(2*h);
end;
```

5.1.6 Brent's method

Since there exist situations where secant method and false position - method lose in efficiency to the bracketing method, one can pose a question: can these methods be combined in a way which makes best use of the best properties of all three methods? It turns out that there is: Brent's method combines the secant method, bracketing and inverse quadratic interpolation.
The idea is as follows: you wish to solve an equation of the form $f(x) = 0$. As with bracketing method, you need two points, a and b, so that $f(a)f(b) < 0$. This means that if f is continuous, according to intermediate value theorem, it must have a root between a and b.
Before presenting the Brent's method, we will study the so called Dekker's iteration, on which the the Brent's method is based on. Dekker's iteration

uses three points at each step of the iteration: b_n, the most recent estimate for the root of f, a_n, is a point for which $f(a_n)f(b_n) < 0$, and $|f(b_n)| \leq |f(a_n)|$, and the previous iterate, b_{k-1}. For the first iteration set $b_{-1} = a_0$.
At each step of iteration, two possible values for the next iterate are computed; first one by the secant method:

$$s = b_n - \frac{b_n - b_{n-1}}{f(b_n) - f_{n-1}},$$

and the second using the bracketing method:

$$m = \frac{a_n + b_n}{2}.$$

If $b_k < s < m$, then $b_{n+1} = s$, otherwise, $b_{n+1} = m$. Then a new contra point is selected. If $f(a_n)f(b_{n+1} < 0)$, no change is necessary, and $a_{n+1} = a_n$, otherwise $a_{n+1} = b_n$. finally test, if $|f(a_{n+1})| < |f(b_{n+1})|$. If the inequality holds, then a_{n+1} is (probably) a better estimate for the function root, so swap the values a_{n+1} and b_{n+1}.
Brent's method introduces several additional tests to ensure a fast convergence. First, if $f(a_n)$, $f(b_n)$ and $f(b_{n-1})$ are distinct, the method uses *inverse quadratic interpolation* instead of secant method.
Inverse quadratic interpolation is another root finding method for function $f(x)$, using Lagrange's quadratic interpolation to approximate the inverse of f. The quadratic inverse formula is a recurrence relation:

$$x_{n+1} = \frac{f(x_{n-1})f(x_n)}{(f(x_{n-2}) - f(x_{n-1}))(f(x_{n-2}) - f(x_n))}x_{n-2} +$$
$$\frac{f(x_{n-2})f(x_n)}{(f(x_{n-1}) - f(x_{n-2}))(f(x_{n-1}) - f(x_n))}x_{n-1} +$$
$$\frac{f(x_{n-2})f(x_{n-1})}{(f(x_n) - f(x_{n-2}))(f(x_n) - f(x_{n-1}))}x_n.$$

Second, set some tolerance δ, and, if previous step used bracketing, an inequality

$$|\delta| < |b_n - b_{n-1}|$$

must hold. If it doesn't, next iteration will also use bracketing.
If previous iteration used inverse quadratic interpolation or secant method, an inequality

$$|\delta| < |b_{n-1} - b_{n-2}|$$

must hold in order for another interpolation to be made: otherwise bracketing will be used.
These tests are performed, because in Dekker's method a situation may arise, where $|b_{n+1} - b_n|$ will be very small, leading to extremely slow convergence of (b_n).
Additionally, in order for an interpolation to be performed at step n of algorithm, if step $n-1$ used bracketing this inequality has to hold:

$$|s - b_n| < \frac{1}{2}|b_n - b_{n-1}|$$

in order to perform a interpolation at step n. If step $n-1$ used interpolation, an inequality

$$|s - b_n| < \frac{1}{2}|b_{n-1} - b_{n-2}|$$

has to hold to continue performing interpolations.
These inequalities ensure, that consecutive interpolation step sizes halve every two iterations, and furthermore, ensure that interpolation step size will be less than δ, thus forcing the use of bisection method, once the root has been localised to a small enough an interval.
Brent's method is somewhat complicated, but it is very popular method of finding roots: for example MATLAB's function `fzero` uses it. Here is an example implementation in MATLAB.

```
function root = brent(f,x0,x1)
% The function brent will attempt
% to find the function root on
% a given interval. The function
% must a single variable real valued
% function, and it must change sign
% on the given interval
% The parameter f is function handle
% or a string holding the function name
% x0 and x1 must be real numbers that
% satisfy f(x0)*f(x1)<0.

% Check the initial condition
if(f(x0)*f(x1)>0)
    error('no sign change on (a,b)');
end
```

```
a= x0;
b = x1;
% make sure the endpoints
% are in right order
if(abs(f(x0))< abs(f(x1)))
    b = x0;
    a = x1;
end
c = a;
s = a;
% mflag keeps track of the previous step:
% if true(1) previous step was bisection
% if false(0) it was an interpolation
% or secant step.
mflag = true;

delta = 1e-4;
d = 0;
% conditions for ending the iteration:
% small enough a function value or small
% enough a an interval
while(abs(f(b))>1e-8||abs(f(s))>1e-8||abs(b-a)<1e-10 )
    % Do we use interpolation or the secant rule ?
    if(norm(f(a)-f(c))>1e-11 && norm(f(b)-f(c))>1e-11)
        s = inversequadratic(f,a,b,c);
    else
        % Secant rule
        s = b-f(b)*((b-a)/(f(b)-f(a)));
    end
% Now a list of conditions that define,
% if we take a bisection rule instead
    c1 = (0.25*(3*a+b)<s || s<b);
    c5 = (mflag == 1 && abs(s-b)>= abs(b-c)/2);
    c2 = (mflag == 0 && abs(s-b)>= abs(b-c)/2);
    c3 = (mflag == 1 && abs(b-c)<delta);
    c4 = (mflag == 0 && abs(c-d)<delta);
    if(c1||c2||c3||c4||c5)
        s = (a+b)/2;
        mflag = true;
    else
```

```
        mflag = false;
    end
    d = c;
    c = b;
    % Define a new interval: determine
    % the enpoints
    if(f(a)*f(s)<0)
        b = s;
    else
        a = s;
    end
    % put the points in right order
    if( abs(f(a))<abs(f(b)) )
        aux = a;
        a = b;
        b = aux;
    end
end
root = b;

function s = inversequadratic(f,a,b,c)
s = (a*f(b)*f(c))/((f(a)-f(b))*(f(a)-f(c)));
s = s + (b*f(a)*f(c))/((f(b)-f(a))*(f(b)-f(c)));
s = s + (c*f(a)*f(b))/((f(c)-f(a))*(f(c)-f(b)));
```

5.1.7 Roots of polynomials

The root finding methods presented thus far have not made little distinction on the functions whose roots we have wished to find: there have been requirements to be sure, but finding the roots has been based on the functions derivatives, or Lipschitz-continuity or intermediate value theorem. If the studied function is a polynomial, one can take advantage of the properties of the function itself.
The fundamental theorem of algebra states, that nth polynomial $p(x)$ has n roots in the complex plane, so a root will always be found. Also, polynomials are differentiable and continuous on entire real line. Using these properties allows us to develop algorithms for finding the roots of polynomials. While one can use any of the previous algorithms to find roots of polynomials, as well as any other function, methods crafted for polynomials tend to be more

accurate and converge faster than the more general ones. As an example of an root finding algorithm for polynomials, we present the *Laguerre's method.*

Laguerre's method

According to the fundamental theorem of algebra, we can write every polynomial $p(x)$ of nth degree in form

$$p(x) = C(x - x_1)(x - x_2)\dots(x - x_n),$$

where $x_i, i = 1 \dots n$ are the roots of p. To get the Laguerre's method, study the natural logarithm, and logarithmic derivatives of the p.

$$\log|p(x)| = \log|C| + \log|x - x_1| + \log|x - x_2| + \ldots + \log|x - x_n|,$$

$$\begin{aligned}\frac{d\log|p(x)|}{dx} &= \frac{1}{x-x_1} + \frac{1}{x-x_2} + \ldots + \frac{1}{x-x_n},\\ \frac{d^2\log|p(x)|}{dx^2} &= -\frac{1}{(x-x_1)^2} + \frac{1}{(x-x_2)^2} + \ldots + \frac{1}{(x-x_n)^2}.\end{aligned}$$

Denote the first and second derivatives of p with

$$F(x) = \frac{d\log|p(x)|}{dx}, \quad G(x) = \frac{d^2\log|p(x)|}{dx^2}.$$

Now some assumptions are required: assume, that the root we are currently looking for, x_1 is a certain distance a away from our current estimate x, while all other roots are at same distance b away from our current best estimate. Denote : $a = x - x_1$ and $b = x - x_i, i = 2 \dots n$. This allows you to express F and G in terms of a and b:

$$F \equiv \frac{1}{a} + \frac{n-1}{b},$$

$$G(x) \equiv \frac{1}{a^2} + \frac{n-1}{b^2}.$$

Solving these equations for a gives

$$a = \frac{n}{F \pm \sqrt{(n-1)(nH - G^2)}}$$

where the sign is selected to give the largest magnitude for the denominator.

This gives an approach for an algorithm: Select an initial guess x_0, and on every iteration k, compute $F = \frac{p'(x_k)}{p(x_k)}$ and $G = F^2 - \frac{p''(x_k)}{p(x_k)}$. Then set a as previously:

$$a = \frac{n}{F \pm \sqrt{(n-1)(nH - G^2)}}$$

and choose the sign appropriately. Finally we set $x_{k+1} = x_k - a$.
One should note that if the set of assumptions made in the derivation of the method does not hold for some polynomial P, P can be transformed into polynomial Q for which the assumptions do hold true. Finding all the roots can be derived through finding one root: if $a + ib$ is a root, $a - ib$ is also a root; if x_0 is a root of a polynomial P, $P = (x - x_0)Q(x)$ for some polynomial Q of $(n-1)$th degree, and factor $(x - x_0)$ can be reduced away.
Here is an example implementation of Laguerre method to find one root of a polynomial in MATLAB.

```
function z = laguerre ( p, x0, tol, itmax )
% the parameter p should be 1xn or nx1
% vector holding the coefficients of
% polynomial P(x).
% x0 should be an initial guess for the
% root, and it determines, towards which
% root the method converges
% tol and itmax determine the halting
% condition for the method: it halts
% if f(xn)<tol or if number of iterations
% exceed the itmax.
n = length(p)-1;
dp = polyder(p);
dp2 = polyder(dp);
ctr = 0;
% Here we use the built-in function
% polyder that provides a derivative
% for polynomial.
while(ctr<itmax)
    px = polyval(p, x0);
    dpx = polyval(dp, x0);
    dp2x = polyval(dp2, x0);
    %is the current guess ok?
    if(abs(px) < tol)
        z = x0;
```

```
          return
      end
      % Here we compute the F
      F = dpx/px;
      % And here the G
      G = F*F-dp2x/px;
      % here's the square root part of
      % a
      disc = sqrt((n-1)*(n*G-F*F));
      % Here we decide if we choose
      % positive or negative sign
      if (abs(F-disc) < abs(F+disc))
         denom = F+disc;
      else
         denom = F-disc;
      end
      dx = n / denom;
      % update the x0
      x0 = x0 - dx;
      % if the change is very small,
      % there is no point in
      % continuing.
      ctr = ctr +1
      if ( abs(dx) < tol )
         z = x0;
         return
      end
end
z = x0;
```

5.1.8 Root finding in MATLAB

MATLAB offers tools for finding roots of finding single variable functions. First one is the function `fzero`, which is a built-in implementation of the Brent's method. It attempts to find a root located near a parameter location `x0`. The function `fzero` works on any single variable functions, but returns only one root. For polynomials there exist the function `roots`, which will compute all the roots of given polynomial; remember that MATLAB handles polynomials as vectors containing the coefficients $a_0 \ldots a_n$, in order $a_n, a_{n-1}, \dots a_0$. The function `roots` is based on the companion matrix method.

5.2 Minimization algorithms

Minimization algorithms, or more generally, optimization, is a field of mathematics that studies selecting the best possible element from some set of alternatives. Usually this can be reduced to finding minimums and maximums of a real valued function. Finding maximums can in turn be reduced to minimizing problem: finding maximum of function f is same as finding minimum of $-f$.
As with finding roots of non-linear functions, minimization finds its basis in great theorems of calculus: the extremal value theorem, proved Karl Weierstrass in 1860, says, that a continuous, real valued function on a compact set attains its maximum and minimum value. These values are local to the compact set. This allows the bracketing idea we already presented with the root-finding algorithms: if there exist points x, y, z so that $f(y) < \min\{f(x), f(z)\}$, then there exists a minimum at some y_0, $x < y_0 < z$.
Optimization problem is called constrained, if the variable have some a priori restrictions. Generally this makes the problem easier, as it makes possible to apply the extremal value theorem. Also, since problems faced in real world are also usually constrained, it is not an unreasonable supposition. As will be seen, there are minimum search methods, that require an unconstrained space to work.
If the studied function is differentiable, the problem of minimums becomes easier: calculus teaches us, that functions extremal values are located either at the functions critical points, or at the boundary of the domain. Critical points of function $f : \mathbb{R}^n \to \mathbb{R}$ are points $\mathbf{x}_0$ where partial derivatives $\frac{\partial f(\mathbf{x_0})}{\partial x_i} = 0$ for all $i = 1 \dots n$. However, a critical points may be minimum, it is not necessarily so: it might be a local optima, or a saddle point. If the function is twice differentiable, it is possible to distinguish minima, maxima and saddle points using the second derivatives test (so called Hessian matrix). Twice differentiability is a restricting condition, and even if function f were twice differentiable, finding the critical points can be difficult. In these cases a numerical study of the problem is called for.
Optimization is a area of mathematics, that, while it has been widely studied, is based on heuristics. Many of the methods are extremely complex, and the

proofs of their convergence, if they even exist, even more so. The methods presented here are only the proverbial tip of the iceberg, and meant only to serve as an example: there are many more, most of them guaranteed to work better in some situation than those presented here.

5.2.1 Golden section search

Golden section search is a method for finding the minima of a unimodal single variable function f. It is based on the idea of bracketing by successively narrowing the interval on which the extremum is known exist. This is possible due the unimodality requirement: it means that there exists an $a < m < b$, and that for all $a < x \leq m$ f is monotonically decreasing, and that for all $m \leq x < b$ f is increasing. The algorithm gets its name from maintaining triples of points, whose distances form a golden ratio.
The algorithm works as follows: you have points x_1, x_2, x_3 so that $x_1 < x_2 < x_3$ and $f(x_2) < \min\{f(x_1), f(x_3)\}$. This means, that the minimum must lie on the interval (x_1, x_3). Then select the new interval by considering two cases:

1. If $x_2 - x_1 > x_3 - x_2$ select $x_0 \in (x_1, x_2)$ fulfilling the golden ratio requirement.

   If $f(x_1) < f(x_0)$, the new interval is defined by (x_0, x_3) with x_2 being the best estimate for minimum.

   If $f(x_0) < f(x_2)$ the new interval is (x_1, x_2) and x_0 is the new best estimate for the minimum.

2. If $x_2 - x_1 < x_3 - x_2$ we select $x_0 \in (x_2, x_3)$ so, that the distances form a golden ratio.

   If $f(x_1) < f(x_0)$, the new interval is defined by (x_0, x_3) with x_2 being the best estimate for minimum.

   If $f(x_0) < f(x_2)$ the new interval is (x_1, x_2) and x_0 is the new best estimate for the minimum.

Keep iterating the steps 1 and 2 until the length of the interval (x_1, x_3) is very small.
The easiest way to implement the golden section search with computers is to use recursion. As has been stated previously, as a rule recursion should be

avoided, but since depth of the recursion is unlikely to be very deep, hence its use in this example:

```
function m = golden(f,x1,x2,x3)
% The golden section search finds
% the minimum of unimodal function
% f,
% parameter f should be an inline
% function or a function handle,
% x1 and x3 should define the
% interval known to contain a
% minimum and x2 should be the
% initial guess for the minimum.
tol = 1e-8;
phi = 2- ((1+sqrt(5))/2);
% 2 - the golden ratio

x4 = x2 + phi*(x3-x2);
% a value between x2 and x3,
% the new guess for the function
% minimum.


% This will end the algorithm, when the interval
% is small enough. Please note that there is no
% check on the recursion depth, but MATLAB defaults
% to maximum of 500 recursions.

if(abs(x3-x1)<tol*(abs(x2)+abs(x4)))
    m = (x3+x1)/2;
    return
end

% Select the new interval for the
% search, and call recursively.
if(f(x4)<f(x2))
    m = golden3(f,x2,x4,x3);
    return
else
    m = golden3(f,x4,x2,x1);
    return
```

```
end
```

5.2.2 Brent's method

Brent's method presented previously as a root finding tool can be modified for use in optimization tasks. The method can be roughly summarized like this: on each iteration a quadratic polynomial is fitted on three existing points, gained through either previous iterations, or initial guess. The minimum of this parabola is then taken as a guess for the functions minimum. If it lies between the interval that we know holds the minimum, then it is accepted as an interpolating point, and used to generate a new, smaller interval that holds the minimum. If the point is unacceptable, then a regular golden section step is taken.
The idea is very much like in the root finding version: we attempt to speed the algorithm by interpolation. In this case fitting a parabola to three existing points, and taking the minimum of the parabola as the best guess for the function minimum. Then a test is made: if the point lies within the bounds of the current interval, it is accepted and used to generate a new, shorter interval. If it is not accepted, a golden section search step is taken.

```
function [xmin fxmin] = brentmin(F,ax,bx,cx)
itmax = 100;
% The 1-1/(golden ratio) for
% golden section search
golden = (1/sqrt(5))/2;
gold = 1-1/gold;

xmin =0;
fxmin= 0;

zeps = eps*1e-6;
iter =0 ;
tiny = 1e-8;
f = fcnchk(F);
% Distance moved on the last step
d = 0;
% Distance moved on the step before
% last
e = 0;
```

```
% Set up the bracket limits correctly
if(ax<cx)
    a = ax;
    b = cx;
else
    a = cx;
    b = ax;
end

% Set up the initial guess for the
% function minimum location and
% value
x = bx;w = bx; v =bx;
xm = 0.5*(a+b);

% Set up the numerical tolerance
tol1 = abs(x)*tiny+zeps;

% The search loop checks for maximum iterations
% and the lenght of search interval
while(iter<itmax && (abs(x-xm)<= abs(2*tol1-0.5*(b-a))))
    xm = 0.5*(a+b);
    tol1 = abs(x)*tiny+zeps;
% Check if step before last was big enough to try a
% parabolic step. Note that this will fail on first
% iteration, which must be a golden section step.
    if(abs(e)>tol1)
     % Construct a trial parabolic fit through x, v and w
        r = (x-w)*(f(x)-f(v));
        q = (x-v)*(f(x)-f(w));
        p = (x-v)*q-(x-w)*r;
        q = 2*(q-r);
        if(q<0)
            p = -p;
        end
        q = abs(q);
        etemp = e;
        e = d;
        % Let's check if the parabola minimum is indeed
        % on the interval
        if(abs(p)>=abs(0.5*q*etemp)||p<=q*(a-x)||p>=q*(b-x))
            % The parabola minimum is not on our interval
            % so we take a golden section step instead
            if(x>=xm)
                e = a-x;
            else
                e = b-x;
            end
            d = gold*e;
        else
            % The minimum IS on our current interval
            % so we take a parabolic step
            d = p/q;
            u = x+d;
            if (u-a < 2*tol1 || b-u < 2*tol1)
                d = sign(xm-x)*tol1;
            end
        end
    else
        % The step before was not big enough, so
        % we take a golden section step
        if(x>=xm)
            e = a-x;
        else
            e = b-x;
        end
            d = gold*e;
    end
    % Now we make sure the step is big enough.
    if (abs(d)>= tol1 )
        u = x+d;
    else
        u = x+sign(d)*tol1;
    end
    % At this point u holds our best estimate for
    % function minimum location. Now we evaluate
    % function at u and judge, if it really is.
    % Remember, x is the old best estimate
    if(f(u)<=f(x))
```

```
        % The current estimate was better than old
        % so we stick with it
        if(u>=x)
            a = x;
        else
            b = x;
        end
        v = w;w = x;x= u;
    else
        % The newer estimate wasn't better, so
        % we can limit the search to the interval
        % it did not cover
        if(u<x)
            a = u;
        else
            b = u;
        end
        if(f(u)<=f(w)||w==x )
            v=w;
            w=u;
        elseif(f(u)<=f(v)||v==x||v==w)
            v=u;
        end
    end
    xmin = x;
    fxmin = f(x);
    % If one wishes to observe the
    % convergence or non-convergence
    % uncomment
    %disp([xmin fxmin])
end
```

5.2.3 Search methods for multivariable functions

Powell's method

Powell's method is one method of finding minimums of multivariable real valued functions. It is based on the fact that if the function $f(x_0 \dots x_n)$ has a minimum at $(x_{0_0} \dots x_{n_0})$, then the function f reaches its minimum in the direction of the vector e_i at $f(0, \dots x_{i_0} \dots x_n)$. Simplistically, the idea of

Powell method is to perform n single dimension minimizations along each of the axes.
The algorithm proceeds as follows:

- Set $u_i = e_i, \quad i = 1, \dots, N$
- Save the initial point P_0.
- While $= 1, \dots, N$ move from P_{i-1} to (P_{i-1}, u_i) minimum P_i
- While $= 1, \dots, N-1$, set $u_i := u_{i-1}$.
- Set $u_N = P_N - P_0$.
- Move from P_N to (P_N, u_N) minimum, and denote the point with P_0.
- Repeat as long as function values get smaller.

Powell's method is useful when trying to find local optima of functions, that are continuous but whose derivatives are either difficult or impossible to obtain. The efficiency of the algorithm itself is very much dependent on the method used to find the minimums along the search vectors. One can choose between any search algorithms made for functions of one variable.

Steepest descent method

If the studied function is differentiable, but the zeros of derivatives are either difficult to find or there are none, one option is to use geometric intuition: the local minimum is probably in the direction of the function's deepest descent. The idea deepest descent method is to determine the direction of deepest descent at initial point, determine the minimum on this point, move to that point, and iterate, until a local minimum is found.
The convergence of this method is very much dependent on the good numerical properties of the function, as well as the properties of the derivative and the method of finding the minimum on the direction of the deepest descent. There are examples when this algorithm takes extremely long time to find the function minimum of a differentiable function. The Rosenbrock function is one such example.

Quasi-Newton methods

Quasi-Newton methods are a set of algorithms, that use the Newton's method to find a stationary point of the function where the gradient of the function is 0. These algorithms assume, that the function can be approximated with a quadratic polynomial in some area around the minimimum. It then uses the gradient and Hessian matrix (first and second derivatives in single dimension) to find the stationary point.
The idea is built on the second order expansion of Taylor series of function f at $x_0 + \delta$.

$$f(x_0 + \delta) \approx f(x_0) + f'(x)\delta + \frac{1}{2} f''(x)\delta^2.$$

The function f attains its minimum when δ satisfies the equation

$$f'(x) + f''(x)\delta = 0.$$

The left hand side of the second order Taylor expansion gives

$$f'(x - \delta) = f'(x) + \delta f''(x).$$

If the function f is twice differentiable and well enough behaved, and provided the initial guess x_0 is reasonably close to the function's critical point, usually denoted by x^*, the sequence yielded by previous equations:

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

will converge towards the critical point of f.
One should bear in mind that gradient of 0 at some point x_0 does not guarantee that there exists a local optimum at x_0. For differentiable functions it is a necessary but not sufficient condition.
Like its root finding relative, quasi-Newton methods can be generalized into handling functions of more than one variable. It is achieved by substituting the first order derivative with its generalization, the gradient vector, and the inverse of the second derivative by the inverse of the Hessian matrix. With these substitutions the sequence gets the form

$$\mathbf{x}_{n+1} = \mathbf{x}_n - (Hf(\mathbf{x}_n))^{-1}\nabla f(\mathbf{x}_n),$$

where $Hf(\mathbf{x}_n)$ is the Hessian matrix of the function f evaluated at $\mathbf{x}_n$. The quasi-Newton methods avoid computing the Hessian matrix, and use different approximations for it instead.

Here is a very simple implementation of Newton's minimum search in MATLAB.

```
function [fmin,xmin ] = newtmin(f,n,x0)
% Minimization using Newtons method.
% Function will attempt to find the
% root of gradient(f).
% parameter f should be a function handle
% of the studied function.
% f should take only one argument: a vector
% with n components, and it should return a
% real value.
% n is the size of the argument vector
% x0 is the initial guess for the minimum.

x = x0;
for i = 1:20
    x = x-inv(Hessian(f,n,x))*numgrad(f,x,n);
end
fmin = f(x);
xmin = x;


function H =Hessian(f,n,x0)
% Function Hessian attempts to
% compute the Hessian matrix of
% the argument function at point
% x0.
% f should be a function handle of a
% function which takes vector arguments,
% n should be the size of the argument
% vector,
% x0 is the point at which the Hessian
% is determined.
% Note this function uses of the
% the Symbolic toolkit.

% Define a symbolic vector to use as
% a parameter
for i = 97:97+n-1
    A(i-96) = sym(char(i));
```

```
end
H = sym(zeros(n,n));
% First take a Jacobian matrix
J = jacobian(f(A));
% Then derivate each column again
for i=1:n
    H(:,i)= (diff(J,A(i)));
end
% finally do the substitution
H =subs(H,A,x0);

function D=numgrad(f,x0,n)
D = zeros(n,1);
h = 1e-6;
e = zeros(n,1);
for i = 1:n
    e(i) = 1;
    D(i) = (f(x0+e*h)-f(x0-e*h))/2*h;
    e(i) =0;
end
```

5.2.4 Searching minimum in MATLAB

MATLAB provides some very sophisticated tools for finding the functions minima: first and foremost is the function `fminsearch`, that attempts to find the functions minimum using the Nelder-Mead algorithm. The number of variables is not constrained, but there must be a clearly definable minimum. A fairly accurate initial guess is required. The function `fminbnd` attempts to find a function minimum on the interval $[x_0 x_1]$. At the most simple form `fminsearch` takes as a parameter only the function handle and the initial guess. However, as was discussed in linear algebra section, it is possible to use the `fminsearch` to fit parameters to a model so that it will fit the given data.
Here is one idea how to implement the parameter fit using the `fminsearch`.

```
function lam = paramfit(fm, xdata, ydata, initguess)
% function paramfit attempts to find
% the parameters that best fit the
% model fm to data (x,y).
% fm should be a function handle, inline
% function or a string containing the
% function. It must be in form
% f(x,parameters), and parameters must
% be contained in one single vector.
% xdata and ydata must hold two vectors
% of equal length.
% The minimum is searched around initguess
% which must be of proper length, and close
% enough to

% make sure fm is function
fmodel = fcnchk(fm);
% set up the object function...
% fobj = S(lambda) = Sum(f(x_i,lambda)-y_i)^2
% We wish to find the lambda that provides the
% smallest value of fobj.
fobj = inline('norm((fmodel(xdata,lambda)-ydata))',...
    'lambda','fmodel','xdata','ydata');
lam = fminsearch(fobj,initguess,[],fmodel,xdata,ydata);
```

Chapter 6

Differential equations

Differential equation is an equation for some unknown function y, that relates the values of the function with its derivatives. If function y has one variable, the equation will be called *ordinary differential equation.* If y has more than one variables, it will be called *partial differential equation.* Order of differential equation is decided by the highest order of derivatives that is present in the equation.
In order to obtain unique solutions for any differential equation, one needs some a priori knowledge of the problem. These are usually given in the form of *initial values*: $y(x_0) = y_0$. If differential equation has set initial value, it is called *initial value problem.*
Sometimes solution for a differential equation is only wanted on some given interval. In these situations initial conditions are usually given at the end-points of the interval. Differential equation with these kinds of constraints is called *boundary value problem.*
Ordinary differential equation of first order with initial values can be written in as

$$y'(x) = f(x, y(x)); \quad y(x_0) = y_0, \tag{6.1}$$

where $f : \mathbb{R}^2 \to \mathbb{R}$ is dependent on both x and $y(x)$. The goal is to find a function $y(x)$, that realizes both the differential equation and the initial value problem. Solutions are sought by integrating both sides of the equation with respect to x. This gives us

$$y(x) = y_0 + \int_{x_0}^{x} f(s, y(s))ds.$$

However, finding an exact integral for arbitrary is usually impossible. For this reason, numerical solutions play a large role in applications concerning differential equations.
Other things to consider are the existence and uniqueness of the solution. The Picard-Lindelöf theorem states, that a initial value problem has a unique solution, if the right-hand side of the 6.1 is Lipschitz-continuous contraction.

Example 6.2. Solve an initial value problem

$$y'(x) = x^2 - 2 - y(x); \quad y(0) = 2.$$

Integrating both sides directly will not work, but by multiplying both sides with an *integrating factor* e^x, one gets

$$e^x y'(x) + e^x y(x) = e^x(x^2 - 2).$$

Using the product rule of differentiation reversely simplifies the equation to

$$\frac{d}{dx}(y(x)e^x) = e^x(x^2 - 2).$$

Now integrating both sides gives

$$y(x)e^x = \int e^x(x^2 - 2) = e^x(x^2 - 2x) + C,$$

where $C \in \mathbb{R}$ is the integration constant. By multiplying this equation with e^{-x} one gets

$$y(x) = x^2 - 2x + e^{-x}C$$

We apply y to initial value condition $y(0) = 2$ and get

$$y(0) = C = 2$$

and finally one gets

$$y(x) = x^2 - 2x + 2e^{-x}.$$

This y fills both the differential equation and the initial value condition.

As one can see, solving even a fairly simple differential equation can be an effort consuming project.
Solutions for differential equations in MATLAB can be obtained symbolically using the MuPad kernel, or numerically.
Symbolically the solution happens like this

```
>> dsolve('Dy = x^2-2-y','y(0)=2','x')
ans =
-2*x+x^2+2*exp(-x)
```

It is also useful to study systems of differential equations, where both y and f are vectors: $y = (y_1 \dots y_n), f = (f_1 \dots f_n)$. Systems of differential equations are important when one considers differential equations of higher order: one can reduce solving the differential equation

$$y^{(n)} + g_1(x)y^{(n-1)} + \dots g_{n-1}y' = g_n(x)$$

into solving a system of equations

$$y^{(n)} = f(x, y, y', \dots y^{(n-1)}).$$

Before moving on to numerical solution to differential equations, consider the problem for a moment; to be well posed the problem must have solution, and the solution must be unique. There are differential equations, that do not have solutions at all; if a solution exists, there is little guarantee, that it is unique. For the purposes of this course the existence and uniqueness theorem of Picard and Lindelöf is sufficient.

Theorem 6.3. *Let function f be continuous in strip $S = \{(x, y) : a \leq t \leq b, y \in \mathbb{R}\}$ with $a, b \in \mathbb{R}$. Let there exist a constant L so that*

$$|f(x, y_1) - f(x, y_2)| < L|y_1 - y_2|,$$

when $x \in [a, b]$ and $y_1, y_2 \in \mathbb{R}$. If these conditions hold, and the initial values $(x_0, y_0) \in S$ (with $y_0 = y(x_0)$), the initial value problem

$$y'(x) = f(x, y(x)); \quad y(x_0) = y_0,$$

has a solution, and it is unique.

6.1 Numerical solutions to ODE's

The methods available for solving ordinary differential equations are numerous, but most are based on discretization the initial value problem 6.1, and creating an estimate for the values of y at $x_1 < x_2 < \dots < x_n$, where $x_{n+1} = x_n + h_n$. The selection of discretization largely dictates the accuracy

of solution: later on methods will be introduced that have built-in discretization.
Generally the estimate values $y_k \approx y(x_k)$ depend on the values $y_{k-1} \dots y_{k-j}$. If $j = 1$, the method in question is *single step method*, and if not, it is *multistep method*.
Single step methods can always be written either in form

$$y_{n+1} = y_n + \phi(x_n, y_n, h_n),$$

when the method is explicit, or in form

$$y_{n+1} = y_n + \phi(x_n, y_n, y_{n+1}, h_n),$$

when the method is implicit.

6.1.1 Euler's method

Probably the most famous explicit single step method for obtaining numerical solutions for initial value problems is the Euler's method, named after Leonhard Euler. The goal of the method is to estimate values of the function at discrete points $x_1 \dots x_n$, $x_{i-1} < x_i$, $x_i = x_{i-1} + h$. The increment h is called *step size*.
To derive the Euler's method, consider initial value problem

$$y'(x) = f(x, y(x)), \quad y(x_0) = y_0.$$

Taylor series of function gives estimates for function's values in the vicinity of its origin based on its derivative. Computing the first two terms of the Taylor expansion of function $y(x)$ at x_0 yields

$$T(y, x_0) = y(x_0) + y'(x_0)(x - x_0).$$

Using the ODE gives form to the derivative:

$$T(y, x_0) = y_0 + f(x, y(x))(x - x_0).$$

Euler's method makes the assumption that this is a good estimate for the behavior of the y, and uses this to compute the estimate for the $y(x_1) = y(x_0 + h)$:

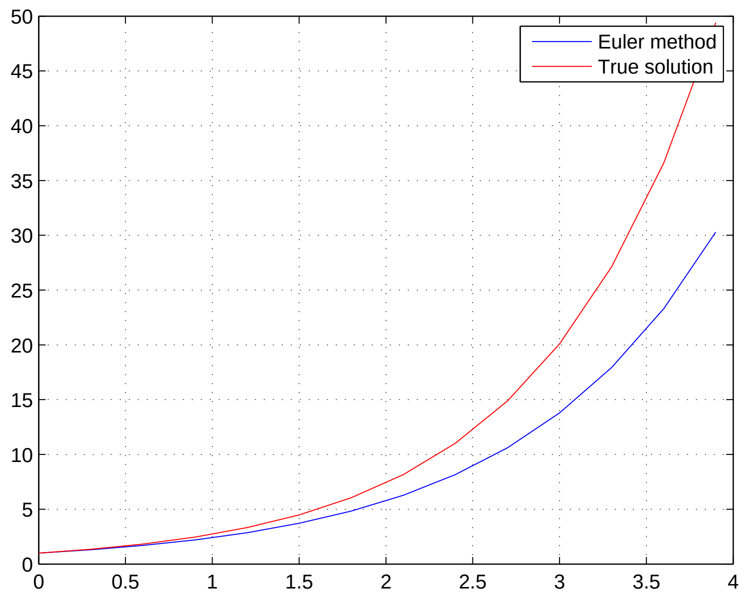
$$y_1 = y_0 + f(x, y(x_0))(x_1 - x_0) = y_0 + hf(x_0, y(x_0)).$$

This gives is the general iteration step for the Euler's method:

$$y_{n+1} = y_n + h(f(x_n), y_n).$$

Here is an example of Euler's method in MATLAB.

```
% An example using Euler's method to
% solve a differential equation
% numerically
% The example equation :
% dy/dx = y, y(0) = 1
% h is selected to be 0.3,
% and the solution interval
% is [0, 4]

h = 0.3;
X = 0:h:4;
Y = zeros(size(X));
Y(1) = 1; % the initial value
for i = 2:length(X)
    Y(i) = Y(i-1) + h*Y(i-1);
end

% Then compare it to the real solution

YR = exp(X);
plot(X,Y,'b',X,YR,'r')
```



Criticism of the method

As the previous example shows, Euler's method is susceptible to error, when the study interval is big: this is due to the fact, that second degree Taylor polynomial is not very accurate method of estimating values of the function, and whatever error it produces, is accumulated into the next iteration. Hence, the estimates produced by the Euler's estimate invariably deteriorate as one moves further away from the initial value point. To combat the deterioration, the step size must be usually set quite small, thus requiring many iterations It is mostly because of this phenomena that Euler's method serves mostly as a historical curiosity, rather than a viable method for actually solving a differential equation numerically.

Backward Euler method

Instead of finite difference approximation, backward Euler estimates the derivative with

$$y'(t) \approx \frac{y(t) - y(t-h)}{h}.$$

This leads to following iteration step:

$$y_{n+1} = y_n + hf(x_{n+1}y_{n+1}).$$

Backward Euler method is an example of implicit method: in order to complete the iteration step n, one needs to solve the given equation for y_n. There are several ways to do this numerically: you may find suitable methods in previous chapter. While computational requirements are considerably more than that of regular Euler's method, the numerical stability is notably better.

Exponential Euler method

Another example of explicit single step methods is the exponential Euler method. If it so happens, that the ODE of the initial value problem takes the form

$$y'(x) = K - Ly(x),$$

then a approximate numerical solution can be obtained through iteration

$$y_{n+1} = y_n e^{-Lh} + \frac{K}{L}(1 - e^{-Lh}).$$

In some specific situations this method can be very accurate, but generally the error term is comparable to that of the Euler's method.

6.1.2 Runge-Kutta methods

Runge-Kutta method is not so much a one single method, rather than a collection of both explicit and implicit multistep methods. They were developed at the end of 19th century by German mathematicians C. Runge and M.W. Kutta.

The idea behind the Runge-Kutta methods is to increase the number of evaluation points in the interval $[x_n, x_{n+1}]$. This is achieved by using a test step at the middle of the interval to cancel out error terms of lower order. The method introduced here is the "classical Runge-Kutta method", or the fourth order method, usually known simply as RK4.

Given an initial value problem

$$y'(x) = f(x, y(x)), \quad y(x_0) = y_0$$

define terms

$$\begin{aligned}k_1 &= f(x_n, y_n),\\ k_2 &= f(x_n + \tfrac{h}{2}, y_n + \tfrac{1}{2}hk_1),\\ k_3 &= f(x_n + \tfrac{h}{2}, y_n + \tfrac{1}{2}hk_2),\\ k_4 &= f(x_n + h, y_n + hk_3).\end{aligned}$$

Terms k_i define the slope of the estimated solution during the interval: k_1 estimates the slope at the beginning of the interval $[x_n, x_n + h]$. The term k_2 estimates the slope at midpoint of the interval $[x_n, x_n + h]$ using k_1 to determine a value for the y at $x_n + \frac{h}{2}$ using the Euler's method; k_3 does the same, but using k_2 as the slope. Term k_4 is the estimate for the slope at the end of the interval.
The final estimate for the slope on the interval $[x_n, x_n + h]$ is obtained as a weighted sum of the estimates for the slope: slope k will be:

$$k = \frac{1}{6}(k_1 + 2k_2 + k_3 + k_4).$$

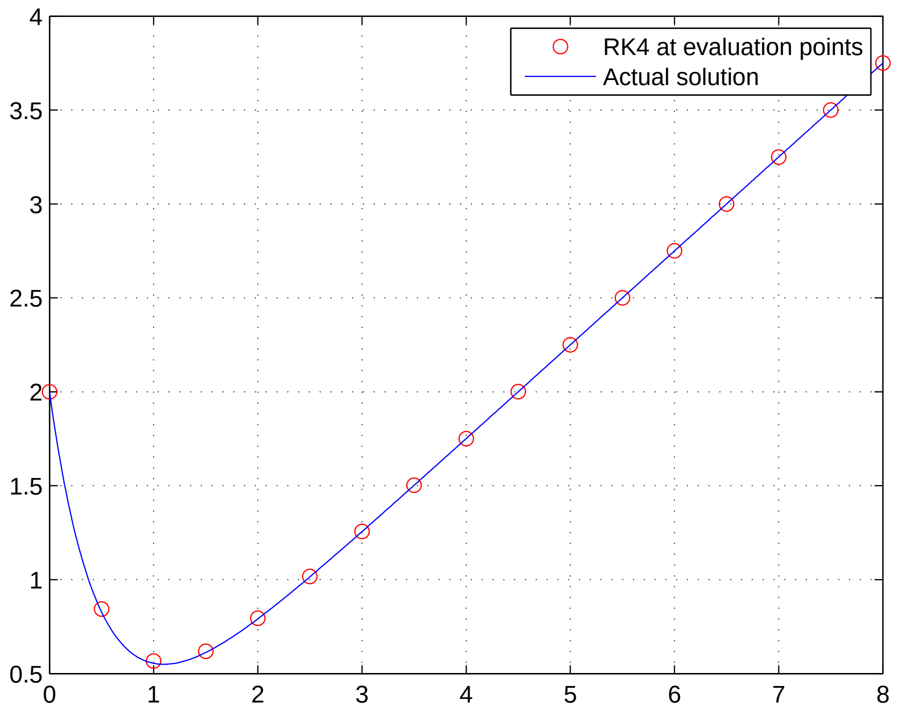
The iteration step will be same as in Euler's method, only instead of using just f to estimate the function progression, use the k.

$$y_{n+1} = y_n + hk = y_n + \frac{h}{6}(k_1 + 2k_2 + k_3 + k_4).$$

RK4 is a fourth order method, meaning that the value y_n is dependent on four previous values of y. It also means, that the error term of this method will be of order $O(h^4)$.
Note the similarity between the numerical methods: if f is independent in respect to $y(x)$, then the RK4 is the Simpson's numerical integration method.
Here is an example implementation of Runge-Kutta method for a sample function. In actuality, though, there is little reason to implement Runge-Kutta methods yourself. There are many functions to achieve this in MATLAB function library.

```
% Example of using Runge-Kutta method
% of the fourth order to solve a
% differential equation
% dy/dx = -2y+x, y0 = 2.

h = 0.5;

X = 0:h:8;
```

```
Y = zeros(size(X));
% for simplicity's sake, define
% f as inline function
f = inline('-2*y+x','y','x');

Y(1) = 2;
for i = 2:length(X)
    k1 = f(Y(i-1),X(i-1));
    k2 = f(Y(i-1)+0.5*h*k1,X(i-1)+0.5*h);
    k3 = f(Y(i-1)+0.5*h*k2,X(i-1)+0.5*h);
    k4 = f(Y(i-1)+h*k3,X(i-1)+h);
    Y(i) = Y(i-1)+h/6*(k1+2*k2+2*k3+k4);
end

% Check the solution versus symbolic
% result.
syms x y;
y = dsolve('Dy = -2*y+x, y(0)=2','x');
x = 0:0.02:8;
y = subs(y,x);

plot(X,Y,'r.',x,y,'b');
```



As one can see, the numerical estimates fall nicely alongside the actual solution. This particular function, however, is of well behaved variety; computing the solution with Euler method will yield nearly identical solution. This means, that the sample equation, $y' = -2y + x$ is not *stiff*. A stiff ODE is an equation, that will work particularly poorly under numerical solution methods.

6.2 Solving ODE's in MATLAB

MATLAB has a range of functions dedicated to solving differential equations numerically. There are methods of high and low orders, implicit and explicit and for stiff and non-stiff equations. The fourth order Runge-Kutta method that was introduced earlier, can be found in the function `ode45`. All of the `ode` methods are invoked similarly: for example `ode45(f,[0,8],5.5)`. First argument is the right-hand side of the ODE, second argument defines the beginning and endpoints of the interval where the solutions are sought, and the final obligatory argument is the initial value at the beginning of the interval.

Example 6.4. Solve a second order initial value problem $y'' + 3.5y' + 4y = 0$ with initial values $y(0) = 2, y'(0) = 0$ and give a numeric approximation for solution in at $x = 2$. Manual solutions would lead to computing the characteristic equations for the y, but for MATLAB solution, the ODE is written in form $y'' = -3.5y' - 4y$. By denoting $y_1 = y, y_2 = y'$ solving the second order differential equation is equivalent to solving the system

$$\begin{cases} y_1' = y_2 \\ y_2' = -3.5y_2 - 4y_1 \end{cases}.$$

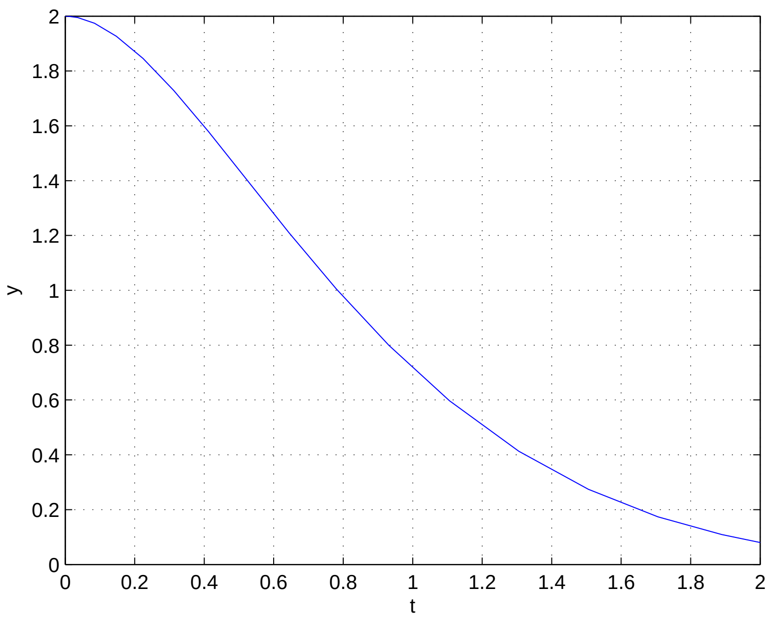
To get MATLAB solution, first one needs to create the differential equation, or rather, the right hand side of one:

```
% First set up the right hand side of the differential
% equation
>> dy = inline('[y(2);-3.5*y(2)-4*y(1)]','x','y')
dy =
      Inline function:
      dy(x,y) = [y(2);-3.5*y(2)-4*y(1)]
% Note the inclusion of x in the equation, even though
% it is not used in it: this is the requirement of the
% ode functions
```

After that, solve it and plot the solution. In this case, use `ode23` function.

```
[t y] = ode23(@deqex, [0 2], [2 0])
% In this example, t is the variable
% Remember the initial values:
% y'(0) = 0, y(0)=2.
% After this there is a n-vector t, and
% nx2 vector y. y(:,2) holds the solution
% for the y', the y(:,1) for the y.
plot(t,y(:,1));grid
% Numeric estimate for the y(2) is the
% last element of the y(:,1) = 0.0801.
```

Plotted solution on the interval $[0, 2]$ looks like this. Note that solving this equation symbolically is not possible in MATLAB.



6.3 Boundary value problems

Initial value problems are not the only type of problems that face differential equations: in real life applications a more common situation is, when a solution is sought on some finite interval, and initial values are given at the endpoints of the interval. Differential equations with these kinds of restraints are called *boundary value problems*, and the restraining conditions *boundary conditions*.
Solving boundary value problems is significantly more difficult than initial value problems, even in numerical sense.
Numerical solutions for boundary value problems are obtained through finite difference methods. Let's observe a boundary value problem of form

$$y''(x) = p(x)y'(x) + q(x)y(x) + r(x) \quad a < x < b; \quad y(a) = \alpha, y(b) = \beta.$$

Denote $h = (b-a)/N$ with some $N \in \mathbb{N}$ and $x_j = a + jh$ with $j = 0, 1, \dots N$. Approximate the derivatives using Taylor polynomials:

$$\begin{cases} y(x+h) = y(x) + hy'(x) + \frac{h^2}{2}y''(x) + O(h^3) \\ y(x-h) = y(x) - hy'(x) + \frac{h^2}{2}y''(x) + O(h^3) \end{cases}.$$

Summing up $y(x-h)$ and $y(x+h)$ yields

$$y(x-h)+y(x+h)=2y(x)+h^2y''(x)+O(h^3),$$

from which you can solve y'':

$$y''(x)=\frac{y(x+h)+y(x-h)-2y(x)}{h^2}+O(h).$$

Likewise, one can solve $y'(x)$, and get the usual finite difference:

$$y'(x)=\frac{y(x+h)-y(x-h)}{2h}+O(h^2).$$

Using these approximations one can write:

$$\begin{cases} y'(x)=\frac{y(x_{j+1})-y(x_{j-1})}{2h}+O(h^2) \\ \\ y''(x)=\frac{y(x_{j+1})-2y(x_j)+y(x_{j-1})}{h^2}+O(h). \end{cases}$$

Having done denote $y_j=y(x_j)$, and substitute into the problem:

$$y_{j+1}+2y_j+y_{j+1}=p(x_j)\frac{h}{2}(y_{j+1}-y_{j-1})h^2(q(x_j)y_j+r(x_j)). \tag{6.5}$$

Solving this equation for $j=1,\dots N-1$, and using the boundary conditions $y_0=\alpha, y_n=\beta$, leads to tridiagonal system of linear equations.

$$\begin{bmatrix} a_1 & c_1 & & & 0 \\ b_2 & a_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ & & b_{n-2} & a_{n-2} & c_{n-2} \\ & & & b_{n-1} & a_{n-1} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-2} \\ y_{n-1} \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_{n-2} \\ w_{n-1} \end{bmatrix}.$$

The coefficients a_j, b_j, c_j, as well as the right hand side are obtained from the equation 6.5.

6.4 Partial differential equations

If a differential equation concerns a function of more than one variable, and its partial derivatives, it is called *partial differential equation.* Partial differential

equations are often used to formulate problems concerning many variables, such as propagation of heat or sound.
Solutions to partial differential equations in classical sense are difficult to obtain, and thus the numerical methods play a huge role in seeking solutions to these kinds of problems. The difficulty arises from the fact, that unlike in the case of one variable problems, there is no universal theorem to state when there exists a solution, and whether it is unique.
Partial differential equations are usually classified into prototypes. Some of the most important prototypes are the wave equation

$$u_{tt} = c^2 u_{xx},$$

the heat equation

$$u_t = \alpha u_{xx},$$

and the Laplace equation, and it's inhomogeneous variant, the Poisson equation

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = f(x,y); \quad u|\partial D = g(x,y).$$

Classic study of partial differential equations has concentrated on study of characteristics of these prototypes, classification of equations according to these prototypes. In this course the theory behind these equations is not discussed, just the numerical solutions.

6.4.1 Wave equation

Wave equations are the archetype of a hyperbolic partial differential equations. It concerns a function $u(\mathbf{x}, t)$, where variable t parametrizes the time, and the vector $\mathbf{x}$ the location on the plane. Function u is a solution for a wave equation, if it satisfies an equation

$$\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2},$$

and whatever boundary conditions have been specified. Usual boundary conditions include at least

$$\begin{cases} u(x,0) = f(x), \\ u_t(x,0) = g(x) \end{cases} .$$

These conditions mean, that state of the studied system is known at moment $t = 0$ with respect to $\mathbf{x}$, and the speed of change in system is known at the moment $t = 0$.
Wave equations, as the name suggests, descripts the behavior of wave-like motion, be it light, sound (three-dimensional equations), some liquid (two-dimensional) or a vibrating string (one-dimensional). The constant coefficient c in the equation is the speed of the wave, and solution u will be the magnitude of the wave in location specified by $\mathbf{x}$ at the time t.
Simplicity of solutions depend largely on the dimension of $\mathbf{x}$, and on whether $\mathbf{x}$ is constrained or not. If the equation is posed in single dimension with unrestricted x, then the solution is yielded by the D'Alembert's formula:

$$u(x,t) = \frac{1}{2}(f(x-ct) + f(x+ct)) + \frac{1}{2c}\int_{x-ct}^{x+ct} g(s)ds.$$

There are similar formulas available in higher dimensions. If the solution is limited to some finite area, no formula exists: rather, the solutions are obtained through separation of variables which will lead to Fourier series. If restricted to some finite interval of real line, and for some finite duration, the wave equation will take form

$$\begin{cases} \frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}, 0 < x < L, 0 < t < T \\ u(x,0) = f(x), \\ u_t(x,0) = g(x), \\ u(L,t) = u(0,t) = 0 \end{cases}.$$

To numerically solve the equation, denote by $R = \{(x,t) : 0 < x < L, 0 < t < T\}$. R is a rectangle on plane. The idea is now to subject R to same kind of finite difference study that was introduced with ordinary differential equations. This is achieved by dividing the R into $(n-1)(m-1)$ rectangles of equal size. Denote the division interval of x-axis with $\Delta x = h$ and of t-axis with $\Delta t = k$. Also, denote with $x_i = ih$ and $t_j = jt$; the real function value at $(x_i, t_j) = u(x_i, t_j)$ and the numerical estimate u_{ij}.
To move forward use the familiar formula to approximate the second partial derivatives:

$$u_{xx}(x,t) \approx \frac{u(x+h,t) - 2u(x,t) + u(x-h,t)}{h^2},$$

$$u_{tt}(x,t) \approx \frac{u(x,t+k) - 2u(x,t) + u(x,t-k)}{k^2}.$$

Then replace the exact function values with estimates u_{ij}, and the original equation gives:

$$\frac{u_{i,j+1} - 2u_{ij} + u_{i,j-1}}{k^2} = c^2 \frac{u_{i+1,j} - 2u_{ij} + u_{i-1,j}}{h^2}.$$

Then denote $r = ck/h$, and substitute:

$$u_{i,j+1} - 2u_{ij} + u_{i,j-1} = r^2(u_{i+1,j} - 2u_{ij} + u_{i-1,j}).$$

This equation gives an explicit formula for $u_{i,j+1}$:

$$u_{i,j+1} = (2 - r^2)u_{ij} + r^2(u_{i+1,j} + u_{i-1,j}) - u_{i,j-1}.$$

To compute values on row $j = 2$, one needs both the rows $j = 1$ and $j = 0$. These are obtained from the boundary conditions:

$$u(x_i, k) \approx u(x_i, 0) + u_t(x_i, 0)k = f(x_i) + kg(x_i) = u_{i,1}$$

and with these one can compute the $u_{i,2}$. Numerical solution is now obtained by iteratively computing the rows of the lattice.

Example 6.6. As an example, solve a wave equation concerning a vibrating string

$$\begin{cases} u_{tt} = 4u_{xx}; \quad 0 < x < 3; 0 < t < 2, \\ u(0,t) = u(3,t) = 0; \quad 0 < t < 2, \\ u(x,0) = f(x) = sin(\pi x) + sin(2\pi x); \quad 0 \le x \le 3, \\ u_t(x,0) = g(x) = 0. \end{cases}$$

Select $h = 0.1, k = 0.05$. The constant r, required for the formula, is $r = ck/h = 2 \cdot 0.05/0.1 = 1$. Thus the linear equation for u_{ij} becomes $u_{i,j+1} = u_{i+1,j} + u_{i-1,j} - u_{i,j-1}$.

```
% attempt to numerically estimate solutions to
% wave equation u_(tt) = 4u_(xx), 0<x<3, 0<t<2
% with boundary conditions
% u(x,0) = sin(x*pi)+sin(2*x*pi)
% u(0,t) = u(3,t) = 0;


clear; clc;close all;
```

```
% function defining boundary values
f = inline('sin(x*pi)+sin(2*pi*x)','x');
h = 0.1;k=0.05;
t1 = 2;
x1 = 3;
M = zeros(t1/k+1,x1/h+1);
x = 0:h:x1;
[m n ] = size(M);
% these come from initial conditions
M(1,:) = f(x);
M(2,2:n-1) = 0.5*(f(x(1:n-2))+f(x(3:n)));
% fill the mesh, retain boundary values.
for l = 3:m
    M(l,2:n-1) = M(l-1,3:n)+M(l-1,1:n-2)-M(l-2,2:n-1);
end
% draw
mesh(fliplr(M));
```

6.4.2 Heat equation

Heat equation is the primary prototype for the parabolic differential equation. It describes the heat distribution or temperature variation in a determined object over time. One dimensional heat equation has the form

$$u_t - c^2 u_{xx}.$$

Heat equation can be generalized into more dimensions by replacing the second x-derivatives by spatial Laplacian operator:

$$u_t = c^2 \Delta_x u.$$

To obtain any but the most general solutions, one needs to set some boundary conditions: initial values at boundaries must be known, as must be the initial heat distribution in the object. Thus we gain the equation:

$$\begin{cases} u_t - c^2 u_{xx}; \quad 0 < x < L, 0 < t < T \\ u(0,t) = u(L,t) = 0; \\ u(x,0) = f(x) \end{cases}.$$

Note that this equation assumes, that the temperature at the boundaries of the studied object is constant at all times. For more realistic model one should replace the constant expression with time dependent functions $g(t)$ and $h(t)$. This does, however, make the symbolic solution much more complicated, and is therefore disregarded in this presentation. Solutions for this equations are usually sought through separation of variables, giving access to solutions with the form $u(x,t) = X(x)T(t)$. This will lead to solution:

$$u(x,t) = \sum_{n=1}^{\infty} c_n \sin\left(\frac{n\pi x}{L}\right) e^{-c^2(n\pi/L)^2 t},$$

where

$$c_n = \frac{2}{L}\int_0^L \sin\left(\frac{n\pi x}{L}\right) f(x)dx,$$

that is, c_n's are coefficients of the Fourier sine series.
To numerically solve a heat equation, use the familiar finite difference method: define a rectangle R in which you wish to obtain the solution, then create the discretization by dividing R into $(m-1)(n-1)$ rectangles of equal size hk. Let h be Δx, that is, the height of one rectangle on x-axis, and k the length of the rectangle on t-axis. Denote points $x_i = ih$ and $t_j = jk$, and $u_{i,j}$ the numerical approximation for $u(x_i, t_j)$. Then one can approximate the derivatives.

$$u_t(x, t_i) \approx \frac{u(t_{i+1}, x) - u(t_i, x)}{k}.$$

Time derivative is the forward looking version instead of the usual three point rule.

$$u_{xx}(x_i, t) \approx \frac{u(x_{i+1}, t) - 2u(x_i, t) + u(x_{i-1}, t)}{h^2}.$$

By substituting these into the heat equation, and replacing the true function values with estimates, you get

$$\frac{u_{i,j+1} - u_{i,j}}{k} = c^2 \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}.$$

By solving this equation in respect to $u_{i,j+1}$ you get

$$u_{i,j+1} = u_{i,j} + c^2 k \frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{h^2}$$

By denoting $r = c^2k/h^2$ you get an equation

$$u_{i,j+1} = ru_{i+1,j} + (1-2r)u_{i,j} + ru_{i-1,j}.$$

This equation is called the *forward time, centered space* approximation to the heat equation, because of the forward looking approximation to derivative. It also means, that this approximation only yields good solutions, if solved forward in time.