

# **An Introduction to MATLAB**

Winfried Auzinger  
Christoph Fabianek  
Peter Holy  
Stefan Pawlik

Department of Applied Mathematics  
and Numerical Analysis  
Vienna University of Technology

Version 1.2.1

June 16, 2002

*Life is too short to spend it writing do loops.*

---

Copyright (c) 2001, 2002 W. Auzinger, C. Fabianek, P. Holy, S. Pawlik

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation. A copy of the license can be found at <http://www.gnu.org/copyleft/fdl.html> or can be obtained from the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

The current version of this document including L<sup>A</sup>T<sub>E</sub>X-sources is available at: <http://www.math.tuwien.ac.at/~winfried/matlab.tgz>

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	About MATLAB . . . . .	1
1.2	MATLAB help facilities . . . . .	2
1.3	Getting started . . . . .	3
<b>2</b>	<b>MATLAB vectors, matrices, . . .</b>	<b>7</b>
2.1	Dense vectors and matrices . . . . .	7
2.2	Sparse vectors and matrices . . . . .	10
2.3	Special matrix functions . . . . .	11
2.4	Representation of polynomials . . . . .	13
2.5	Strings . . . . .	14
2.6	Cell and structure arrays . . . . .	14
<b>3</b>	<b>MATLAB arithmetic operators</b>	<b>15</b>
3.1	Computing with vectors and matrices . . . . .	15
3.2	Vectors, matrices and functions . . . . .	18
<b>4</b>	<b>MATLAB graphics</b>	<b>19</b>
4.1	Plotting $(x, y)$ data . . . . .	19
4.2	Function plots . . . . .	21
4.3	3D Plotting . . . . .	21
<b>5</b>	<b>Programming</b>	<b>23</b>
5.1	m-Files . . . . .	23
5.2	Input and output arguments in functions. . . . .	25
5.3	Error handling . . . . .	27
5.4	Function handles, inline functions, . . . . .	27
5.5	Control flow . . . . .	29
5.6	Input, output, and file handling . . . . .	34
5.7	Timing control . . . . .	36
5.8	Examples . . . . .	36

5.9	The symbolic math toolbox . . . . .	43
<b>A</b>	<b>Application: Linear Programming</b>	<b>46</b>
A.1	Basic feasible solutions . . . . .	46
A.2	Extreme points and extreme directions . . . . .	48
A.3	Solving the LP problem geometrically . . . . .	51
<b>B</b>	<b>Function Reference</b>	<b>55</b>
<b>C</b>	<b>Problems</b>	<b>67</b>

# Chapter 1

## Introduction

### 1.1 About MATLAB

MATLAB<sup>1</sup> (Matrix Laboratory) is an interactive software system for numerical computations and graphics. As the name suggests, MATLAB is especially designed for matrix computations: solving systems of linear equations, computing eigenvalues and eigenvectors, factorization of matrices, and so forth. In addition, it has a variety of graphical capabilities, and can be extended through programs written in its own programming language. Many such programs come with the system; these extend MATLAB's capabilities to a wide range of applications, like the solution of nonlinear systems of equations, the integration of ordinary and partial differential equations, and many others. For an overview see [3].

MATLAB is designed to solve problems numerically, that is, in finite-precision arithmetic. Therefore it produces approximate rather than exact solutions, and should not be confused with a symbolic computation system such as Mathematica or Maple. It should be understood that this does not make MATLAB better or worse than a symbolic system; it is a tool designed for different tasks and is therefore not directly comparable.

At the time when this document is created, MATLAB 6 is the most recent version. However, MATLAB 5.3 is in use on many systems. For this introduction, the difference between these versions is hardly relevant. Note, however, that MATLAB 6 comes with a graphical user interface (GUI), but its usage is not covered by this document. The GUI version is not supported on some UNIX platforms.

**Advantages:** MATLAB is an interpreted language for numerical computation. It can perform numerical calculations, and visualize the results

---

<sup>1</sup>MATLAB is a registered trademark of The MathWorks, Inc.

without the need for complicated and time consuming programming. MATLAB allows its users to accurately solve problems, produce graphics easily and produce code effectively.

**Disadvantages:** Because MATLAB is an interpreted language, it can be slow, and poor programming practices can make it unacceptably slow.

The main purpose of this document is to introduce the novel user to the MATLAB system. We give a number of examples, but do not provide a complete overview of all features MATLAB offers. However, after reading this introduction, you should be able to use MATLAB efficiently and to get along using the excellent online help facility. The books [3] and [6] provide a more comprehensive description, together with a large number of examples from various applications.

Basic knowledge about programming is assumed.

## 1.2 MATLAB help facilities

MATLAB provides different types of help with its help system.

**Online help** – to learn more about a function you wish to use, say `pascal`, type

```
>> help pascal
```

```
PASCAL Pascal matrix.
```

```
PASCAL(N) is the Pascal matrix of order N: a symmetric positive definite matrix with integer entries, made up from Pascal's triangle. Its inverse has integer entries.
```

```
PASCAL(N,1) is the lower triangular Cholesky factor (up to signs of columns) of the Pascal matrix. It is involutory (is its own inverse).
```

```
PASCAL(N,2) is a transposed and permuted version of PASCAL(N,1) which is a cube root of the identity.
```

If you do not remember the exact name of a function you want to learn more about, use the command `lookfor` followed by the incomplete name of a function. (Special note about help: The help pages use capitals to set

the MATLAB commands apart from other text. But remember: MATLAB commands are issued in lower case.)

**The Help Window** – The `helpwin` command, invoked without arguments, opens a new window on the screen. To find an information, double click on the name of the subdirectory and next double click on a function to see the help text for that function. You can go directly to the help text of your function invoking the `helpwin` command followed by an argument.

**Web based support and help** – From the MATLAB prompt, type `helpdesk`. This access the MATLAB help page that is part of the installed version of MATLAB. (See also: <http://www.mathworks.com>.) Finally, there is a newsgroup on Usenet, `comp.soft-sys.matlab`, which is devoted to MATLAB. Reading and participating in this group is a good way to improve your MATLAB skills.

**Demos** – To learn more about MATLAB capabilities you may execute the `demo` command. E.g., if you want to learn about matrices in MATLAB open the demo window. In the left panel select ‘Matrices’ and in the right panel select ‘Basic matrix operations’, then click on ‘Run Basic matrix’. Click on the Start>> button to begin the show.

## 1.3 Getting started

Start MATLAB, e.g. by executing the command `matlab` on the system prompt.

```
< M A T L A B >  
Copyright 1984-2000 The MathWorks, Inc.  
Version 6.0.0.88 Release 12  
Sep 21 2000
```

To get started, type one of these: `helpwin`, `helpdesk`, or `demo`. For product information, visit [www.mathworks.com](http://www.mathworks.com).

```
>>
```

‘>>’ is the prompt for interactive input. To perform a simple computation, type a command and press *Enter*. For instance,

```
>> s = 1 + 2  
s =
```

```
3
```

The result of this computation has been saved in a variable with the name `s` chosen by the user. ‘=’ is the assignment operator. If values of variables are needed during your current MATLAB session, you can recall their values typing their names and pressing *Enter*:

```
>> s
s =
     3
```

Variable names begin with a letter, followed by letters, numbers or underscores. Note that MATLAB is case sensitive and that only the first 31 characters of a variable name are recognized.

There are three kinds of numbers in MATLAB: integers, real numbers, and complex numbers. Integers are entered without, real numbers with the decimal point. Complex numbers are represented in Cartesian form. The imaginary unit  $\sqrt{-1}$  is denoted either by `i` or `j`. (Caution: If you have redefined `i` or `j`, e.g. by using these variables within a loop, this predefined value is no longer valid.)

Some examples:

```
>> 10
ans =
    10

>> real=10.01
real =
    10.0100

>> exp(i*pi)
ans =
   -1.0000+ 0.0000i
```

Note that the variable `ans` is always automatically assigned the result of the most recent computation.

### Basic arithmetic operations:

Operation	Symbol
addition	+
subtraction	-
multiplication	*
division	/ or \
exponentiation	^

MATLAB has two division operators: `/` – the right division and `\` – the left division. They do not produce the same results:

```
>> rd = 5 / 1
rd =
     5
```

```
>> ld = 5 \ 1
ld =
    0.2000
```

To interrupt a running program press *Ctrl-c*. Sometimes you have to repeat pressing these keys a couple of times to halt execution of your program. This is not a recommended way to exit a program. However, it may be necessary in certain circumstances.

To enter a statement that is too long to fit into one line, use three periods (`...`) followed by *Enter*. For instance,

```
>> x = sin(1) - sin(2) + sin(3) - sin(4) + sin(5) -...
sin(6) + sin(7) - sin(8) + sin(9) - sin(10)
x =
    0.7744
```

You can suppress output to the screen by adding a semicolon after the statement:

```
>> u = 2 + 3;
```

Variables can also take values via the `input` command. This prompts the user for an input from the keyboard and assigns the variable with the value entered:

```
>> x=input('Bitte x eingeben > ')
Bitte x eingeben > 2
x =
     2
```

In interactive mode, this is equivalent to typing `x=2`. The `input` command is extremely useful within function m-files (see chapter 5), when the execution of a function requires input from the user.

In MATLAB, all calculations are performed in double precision IEEE arithmetic, the accepted standard for floating point computation. By default, MATLAB variables are of the corresponding data type `double`, with

an accuracy of  $\approx 16$  significant decimal digits. However, numbers are by default displayed in 5-digit fixed point format. This can be changed via the `format` command. For instance, typing `format long` activates output in 16-digit fixed point format.

To check the workspace of your MATLAB session, i.e. to see a list of your active variables, use the command `who`, or `whos` to obtain a more detailed information:

```
>> whos
  Name      Size      Bytes  Class

  ls        1x1         8  double array
  rd        1x1         8  double array
  real      1x1         8  double array
  s         1x1         8  double array
  u         1x1         8  double array
  x         1x1         8  double array
```

```
Grand total is 6 elements using 48 bytes
```

The command `clear` removes all your variables from the workspace.

Naturally, MATLAB includes a library of elementary standard functions like `exp`, `sin`, `cos`, etc. (see appendix B).

With the `diary` command it is possible to make a copy of the terminal input and most of MATLAB's output to a text file. The command `diary filename` specifies that this diary is written to the specified file; the default is a file named `diary` in the MATLAB working directory (usually your home directory). `diary on` and `diary off` toggle the state of the diary.

To close MATLAB, type `exit`.

# Chapter 2

## MATLAB vectors, matrices, and further data types

As in the case of scalar variables, arrays need not to be declared, and MATLAB performs *automatic storage allocation*. In the following we describe the basic techniques for creating and operating with 1-dimensional arrays (vectors) and 2-dimensional arrays (matrices).

### 2.1 Dense vectors and matrices

This command creates a row vector:

```
>> a = [1 2 3]
a =
     1 2 3
```

Column vectors are specified in a similar way. However, semicolons must separate the components of a column vector:

```
>> b = [1;2;3]
b =
     1
     2
     3
```

Alternatively, the components of a column vector may be entered on separate lines (by pressing *Enter* instead of typing `;`).

The quote operator `'` is used to create the conjugate transpose of a vector (matrix) while the dot-quote operator `.'` creates the transpose vector (matrix). The command `length` returns the number of components of a vector:

```
>> length(a)
ans =
     3
```

This creates a 3-by-3 matrix:

```
>> A = [1 2 3;4 5 6;7 8 10]
A =
     1  2  3
     4  5  6
     7  8 10
```

Note that the semicolon operator ; separates the rows. To extract a single element of an array, use round braces:

```
>> A(3,2)
ans =
     8
```

The colon : stands for a full range of indices. It can e.g. be used to extract a whole row or column of an array:

```
>> A(2,:)
ans =
     4  5  6
```

```
>> A(:,3)
ans =
     3
     6
    10
```

A submatrix B consisting of rows 1 and 3 and columns 1 and 2 of the matrix A is achieved in the following way:

```
>> B = A([1 3], [1 2])
B =
     1  2
     7  8
```

To interchange rows 1 and 3 of A use the vector of row indices together with the colon operator:

```
>> C = A([3 2 1], :)
C =
     7  8 10
     4  5  6
     1  2  3
```

To delete a row (column) use the empty vector []:

```
>> A(:, 2) = []
A =
     1  3
     4  6
     7 10
```

The second column of A is now deleted. To insert a row (column) we use the technique for creating matrices and vectors:

```
>> A = [A(:,1) [2 5 8]'; A(:,2)]
A =
     1  2  3
     4  5  6
     7  8 10
```

The matrix A has now been restored to its original form.

In MATLAB, the colon operator `:` is used in several ways. It is especially useful for creating vectors of equally spaced values:

```
>> x = 1:5
x =
     1     2     3     4     5
```

Generally, `m:n` generates the vector with components `m`, `m+1`, `...`, `n`, and other increments can be specified using a third parameter, as in:

```
>> x = 2:3:9
x =
     2     5     8
```

This construct is often used in `for`-loops, see chapter 5.

## 2.2 Sparse vectors and matrices

MATLAB has several built-in functions for computations with sparse vectors and matrices. The command `sparse` is used to create a sparse form of a vector or matrix. Let

```
>> A = [0 0 1 1; 0 1 0 0; 0 0 0 1];
```

Then, application of `sparse` generates the sparse form, indicated in the following way:

```
>> B = sparse(A)
B =
      (2,2) 1
      (1,3) 1
      (1,4) 1
      (3,4) 1
```

The command `full` converts a sparse form of a matrix to its full form:

```
>> C = full(B)
C =
      0 0 1 1
      0 1 0 0
      0 0 0 1
```

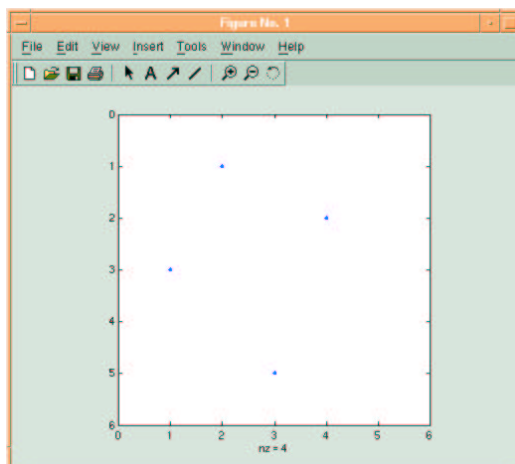
The command `sparse` has the following syntax:

`sparse(k,l,s,m,n)` where `k` and `l` are arrays of row and column indices, respectively, `s` is an array of nonzero numbers whose indices are specified in `k` and `l`, and `m` and `n` are the row and column dimensions, respectively:

```
>> S = sparse([1 3 5 2], [2 1 3 4], [1 2 3 4], 5, 5)
S =
      (1,2) 1
      (3,1) 2
      (5,3) 3
      (2,4) 4
```

The function `spy` creates a graph of the matrix. The nonzero entries are displayed as dots.

```
>> spy(S)
```

Figure 2.1: output of `spy(S)`

To create a sparse matrix with several diagonals parallel to the main diagonal one can use the command `spdiags`. Its syntax is:

```
spdiags(B,d,m,n)
```

The resulting matrix is an  $m \times n$  sparse matrix. Its diagonals are the columns of the matrix `B`. The location of the diagonals is described in the vector `d`.

## 2.3 Special matrix functions

The function `diag` creates a diagonal matrix with diagonal entries taken from a given vector:

```
>> d = [1 2 3];
>> D = diag(d)
D =
     1  0  0
     0  2  0
     0  0  3
```

To extract the main diagonal of the matrix `D` we use the function `diag` again:

```
>> d = diag(D)
d =
     1
     2
     3
```

The function `inv` is used to compute the inverse of a matrix. Let, for instance, the matrix `A` be defined as follows:

```
>> A = [1 2 3;4 5 6;7 8 10]
A =
     1  2  3
     4  5  6
     7  8 10
```

Then,

```
>> B = inv(A)
B =
    -0.6667  -1.3333  1.0000
    -0.6667   3.6667  -2.0000
     1.0000  -2.0000  1.0000
```

The command `find` can be used for finding the indices of the nonzero entries of a vector or a matrix and is often used with sparse matrices.

`find(x)` returns an array containing the indices of the nonzero elements of the vector `x`. `[i,j]=find(A)` returns the row and column indices of the nonzero elements of the (sparse) matrix `A`. `[i,j,v]=find(A)` also returns the row and column indices of nonzero elements of `A` and as third output parameter it returns a vector containing the values of the nonzero elements. For example we could find the value of the absolute smallest nonzero element of a given matrix `A` like this:

```
>> [i,j,v]=find(A);
>> minimum=min(abs(v));
```

It is often useful to start with a predefined matrix providing only the dimension. A partial list of these functions is:

<code>zeros</code>	matrix filled with 0
<code>ones</code>	matrix filled with 1
<code>eye</code>	Identity matrix
<code>rand</code>	matrix with uniformly distributed random numbers
<code>pascal</code>	Pascal matrix
<code>hilb</code>	Hilbert matrix

## 2.4 Representation of polynomials

In MATLAB, polynomial functions of degree  $\leq n$ ,

$$p(x) = a_1 x^n + a_2 x^{n-1} + \dots + a_n x + a_{n+1}$$

are represented by a row vector of length  $n+1$  containing the coefficients,  $\mathbf{p} = [\mathbf{p}(1), \mathbf{p}(2), \dots, \mathbf{p}(n+1)]$ , where  $\mathbf{p}(i) = a_i, i = 1 \dots n+1$ . (Note that  $\mathbf{p}(1)$  is the coefficient of the *highest* power, and  $\mathbf{p}(n+1)$  is the constant coefficient.) There are a number of standard functions which use this representation. `polyval` evaluates a polynomial:

```
>> p=[1 3 0]      % represents p(x)=x^2+3*x
p =
     1     3     0

>> polyval(p,2)   % x^2+3*x=7 for x=2
ans =
     7
```

Some basic symbolic manipulations is also available `polyder` resp. `polyint` differentiate resp. integrate a polynomial. `conv` multiplies polynomials. The result of these operations is again a coefficient vector representing the corresponding result. For example:

```
>> polyder(p)     % differentiate p(x)=x^2+3*x
ans =
     2     3      % The derivative is 2*x+3
```

`polyfit` is used for polynomial curve fitting, i.e. it computes a polynomial approximation for given data points in a least square sense. In the following example, a fit of degree 3 is computed for the data points  $(x_i, \exp(x_i)), i = 1(0.1)1$ :

```
>> x=linspace(0,1,10);
>> y=exp(x);
>> n=3;
>> p=polyfit(x,y,n)
p =
    0.2792    0.4231    1.0160    0.9996
```

In the special case that  $n+1$  equals the number of given data points, with pairwise different abscissas, `polyfit` computes the coefficients of the corresponding unique interpolating polynomial.

## 2.5 Strings

Text data (character strings) can be generated using single quotes as delimiters and can be assigned to variables in the usual way:

```
>> s = 'GNU is Not Unix'
s =
GNU is Not Unix
```

Single quotes within a string are represented by a pair of quotes:

```
>> 'This introduction to MATLAB - it''s marvellous!'
ans =
This introduction to MATLAB - it's marvellous!
```

## 2.6 Cell and structure arrays

*Cell arrays* are arrays which contain elements of arbitrary types. They are identified by curly braces instead of square ones:

```
>> c = {[3,4],18.2,[1,2;2,1],'string'};
```

defines a cell array *c*. We can access elements of *c* in the following way:

```
>> c{3}
ans =
     1 2
     2 1
```

A *structure array* is an array consisting of several variables where each has its own type and identifier.

```
>> p.pol='x^2-3*x+2';p.coef=[1,-3,2];p.zeros=[1,2];p.min=[-1/4]
p =
    pol: 'x^2-3*x+2'
    coef: [1 -3 2]
    zeros: [1 2]
    min: -0.2500
```

defines a structure array *p*. The elements of a structure array are accessed by `array.identifier`, e.g.:

```
>> p.zeros
ans =
     1 2
```

# Chapter 3

## MATLAB arithmetic operators

### 3.1 Computing with vectors and matrices

We now explain how to perform arithmetic operations with vectors and matrices – the main strength of MATLAB. The matrix/vector - arithmetic operations are addition (+), subtraction (-) and multiplication (\*). Addition and subtraction are only defined if the matrices have the same dimensions. Multiplication only works if the matrices have equal inner dimensions: I.e., if  $A$  is an  $n \times m$  matrix and  $B$  is an  $p \times q$  matrix, then  $A*B$  is well-defined (and is calculated by MATLAB) if  $m = p$ . MATLAB also allows for powers (^) of square matrices.

The dot operator . plays a specific role in MATLAB. It is used for the *componentwise* application of the operator that follows the dot operator.

The conjugate transpose of a matrix  $a$  can be obtained by using the quote operator ', the transpose can be obtained with the dot-quote operator .'.

Some examples:

```
>> a=[1 2 3];
>> b=[1 2 3]';
>> (a+i*b')'
ans =
    1.0000 - 1.0000i
    2.0000 - 2.0000i
    3.0000 - 3.0000i
```

while

```
>> (a+i*b').'
```

```
ans =
```

```

1.0000 + 1.0000i
2.0000 + 2.0000i
3.0000 + 3.0000i

```

The dot product (inner product) and the outer product of vectors  $\mathbf{a}$  and  $\mathbf{b}$  are calculated as follows:

```

>> dotprod = a*b
dotprod =
    14

```

```

>> outprod = a'*b'
outprod =
    1 2 3
    2 4 6
    3 6 9

```

Componentwise application of the multiplication operator:

```

>> a.*b'
ans =
    1 4 9
>> a.^2
ans =
    1 4 9

```

Continuing with matrix operations:

```

>> A = [1 2 3;4 5 6];
>> A*A
??? Error using ==> * Inner matrix dimensions must agree.
>> A*A'
ans =
    14 32
    32 77

```

The  $\backslash$  operator solves linear systems of equations. If you desire the solution of  $\mathbf{Ax}=\mathbf{b}$ , then the most simple method using MATLAB to find  $\mathbf{x}$  is to set  $\mathbf{x} = \mathbf{A}\backslash\mathbf{b}$ . If  $\mathbf{A}$  is an  $n \times m$  matrix and  $\mathbf{B}$  is an  $p \times q$  matrix then  $\mathbf{A}\backslash\mathbf{b}$  is defined (and is calculated by MATLAB) if  $m = p$ . For non-square and singular systems, the operation  $\mathbf{A}\backslash\mathbf{b}$  gives the solution in the least squares sense.

*Example:* Let

```
>> A = [1 2 3;4 5 6;7 8 10]
```

```
A =  
     1 2 3  
     4 5 6  
     7 8 10
```

and let

```
>> b = ones(3,1);
```

Then,

```
>> x = A\b  
x =  
    -1.0000  
     1.0000  
     0.0000
```

In order to verify correctness of the computed solution, let us compute the residual vector  $r$ :

```
>> r = b - A*x  
r =  
    1.0e-015 *  
    0.1110  
    0.6661  
    0.2220
```

Theoretically, the entries of the computed residual  $r$  should all be equal to zero. This example illustrates an effect of the roundoff errors on the computed solution.

If  $m > n$ , then the system  $\mathbf{A}\mathbf{x}=\mathbf{b}$  is overdetermined and in most cases the system is inconsistent. A special solution to the system  $\mathbf{A}\mathbf{x}=\mathbf{b}$ , again obtained with the aid of the backslash operator  $\backslash$ , is the least-squares solution. Let

```
>> A = [2 1; 1 10; 1 2];
```

and let the vector of the right-hand sides be the same as the one in the preceding example. Then,

```
>> x = A\b  
x =  
     0.5849  
     0.0491
```

The residual of the computed solution is:

```
>> r = b - A*x
r =
    -0.1208
    -0.0755
     0.3170
```

## 3.2 Vectors, matrices and functions

Usually, standard functions may also take array arguments. The general rule is that the resulting array has the same shape as the input argument. For example:

```
>> sin([1 2 3])
ans =
    0.8415    0.9093    0.1411
```

Based on this behavior, the explicit coding of loops can be avoided in many situations where it would be necessary with a conventional programming language.

# Chapter 4

## MATLAB graphics

MATLAB comes with a rich variety of 2D and 3D plotting capabilities. The following sections provide a ‘quick tour’ to some useful commands, at the same time introducing the typical syntax.

### 4.1 Plotting $(x, y)$ data

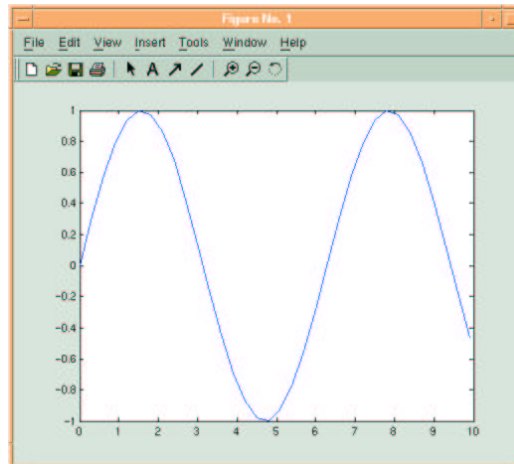
To generate a graph of the function  $y = \sin(t)$  on the interval  $[0, 10]$  we could do the following:

```
>> t = 0:.3:10;  
>> y = sin(t);  
>> plot(t,y)
```

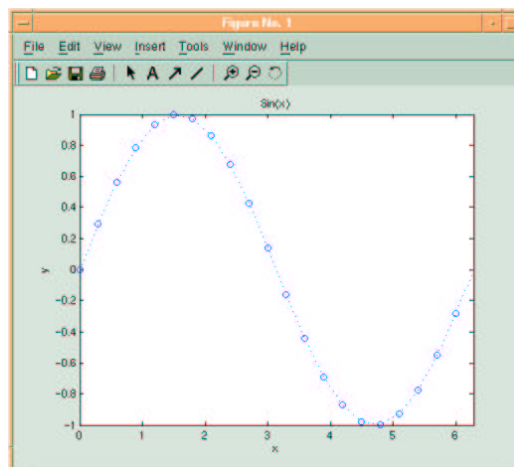
The command `t = 0:.3:10;` defines a vector with components ranging from 0 to 10 in steps of 0.3. Then, `y = sin(t);` defines a vector whose components are  $\sin(0)$ ,  $\sin(0.3)$ ,  $\sin(0.6)$ , etc. Finally, `plot(t,y)` use the vector of `t` and `y` values to construct the graph. Various line types, plot symbols and colors may be obtained with `plot(X,Y,S)` where the character string `S` is a combination of the following options:

color	data points	line type
r red	. point	- solid
g green	o circle	-- dashed
b blue	x x-mark	: dotted

To put labels on the plot, use the commands `xlabel`, `ylabel`, and `title`. With `axis` you can control the scaling and appearance of the axis. The following example demonstrates the use:

Figure 4.1: output of `plot(t,y)`

```
>> plot(t,y,'bo:'), title('Sin(x)'), xlabel('x'), ...
ylabel('y'), axis([0,2*pi,-1,1])
```

Figure 4.2: output of `plot(t,y,'bo:'),...`

It is also possible to plot parametrically defined curves. The following example plots the unit circle from its parametrical representation.

```
>> t=0:.01:2*pi;
>> x=sin(t);
>> y=cos(t);
```

```
>> plot(x,y);
```

## 4.2 Function plots

Another, more convenient possibility to generate the graph of a given function is to use the command `fplot`. To plot  $\sin(x)$  on the interval  $[0, 10]$  simply type

```
>> fplot('sin',[0,10]);
```

Here, the name (as a string) of the function to be plotted is used as the first parameter in `fplot`. This way of passing functions as arguments to other command or functions will be explained in more detail in section 5.4.

To plot more complicated functions using `fplot` you can either refer to the name of your own function m-file as parameter, or you can use inline notation for the expression defining the function:

```
>> fplot('x*sin(x)', [0,10]);
```

This plots the function  $y = x \cdot \sin(x)$  on the interval  $[0, 10]$ . `x` is just a placeholder in the above expression and has nothing to do with `x` as a variable.

## 4.3 3D Plotting

Plotting curves in three dimensions can be done with `plot3`, which works analogous to `plot` in two dimensions. For example

```
>> x=0:.01:30*pi;
>> y=x.^2.*sin(x);
>> z=x.^2.*cos(x);
>> plot3(x,y,z);
```

plots a spiral in three dimensions.

In the following example a bivariate function  $z = \sin(x^2 + y^2)$  is interpolated on the square  $-1 \leq x \leq 1$ ,  $-1 \leq y \leq 1$  using the 'linear' method for 2-D interpolation (`interp2`).

```
>> [x, y] = meshgrid(-1:.25:1);
>> z = sin(x.^2 + y.^2);
>> [xi, yi] = meshgrid(-1:.05:1);
>> zi = interp2(x, y, z, xi, yi, 'linear');
>> surf(xi, yi, zi), ...
title('Bilinear interpolant to sin(x^2 + y^2)')
```

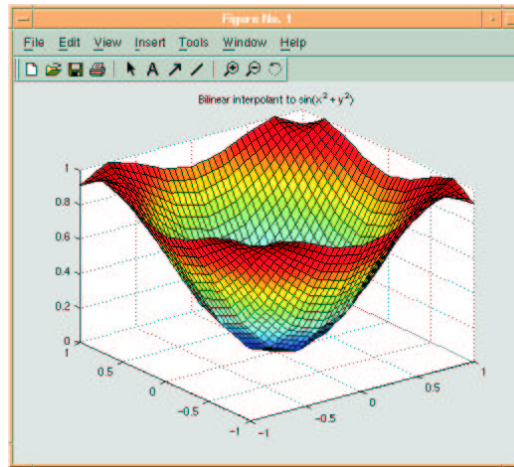


Figure 4.3: output of `surf(xi, yi, zi)`

`meshgrid(x,y)` is a very useful command which transforms the domain specified by vectors `x` and `y` into arrays `X` and `Y` that can be used for the evaluation of functions of two variables and 3D surface plots.

Via the option `shading` you can control the appearance of the surface (turn on/off the mesh lines and interpolated shading).

```
>> surf(xi, yi, zi), title('Bilinear interpolant to ...  
sin(x^2 + y^2)'), shading interp
```

# Chapter 5

## Programming

MATLAB includes a modern, interpreted programming language.

### 5.1 m-Files

Files that contain MATLAB source code are called m-files (extension `.m`). There are two kinds of m-files: script files and function files.

- *Script m-files* do not process any arguments. Upon typing the name of the file (without the extension `.m`), the commands contained in the file are executed as if they had been entered at the keyboard.
- *Function m-files* contain a line with a `function` definition. These may take input arguments and return output arguments. They can be called in the same way as built-in functions.

You create an m-file with your favorite editor (e.g. `pico`, `emacs`, `vi`) or using the editor built in to MATLAB, and save the file in MATLAB's search path. (It is recommended to create a directory for your MATLAB files and start `matlab` in this directory. MATLAB's search path usually contains the current directory. A list of these directories can be obtained by the command `path`.)

Here is an example of a small script m-file - let us save it under the name `firstprog.m`:

```
% Script file firstprog.m
x = pi/100:pi/100:10*pi;
y = sin(x)./x;
plot(x,y)
grid
```

Let us analyze the contents of this file. The first line begins with the percentage sign `%`. This is a comment. All comments are ignored by MATLAB. They are added to improve readability of the code. In the next two lines arrays `x` and `y` are created. Note that the semicolon follows both commands. This suppresses display of the content of both vectors to the screen. The array `x` holds 1000 evenly spaced numbers in the interval  $[\frac{\pi}{100}, 10\pi]$  while the array `y` holds the values of the function  $y = \frac{\sin(x)}{x}$  at these points. Recall that the dot operator `.` before the right division operator `/` specifies the component-wise division of the arrays `sin(x)` and `x`. The command `plot` creates the graph of the sin function using the points previously generated. Finally, the command `grid` is executed. This adds a grid to the graph. We invoke this file by typing its name in the Command Window and next pressing *Enter*.

```
>> firstprog
```

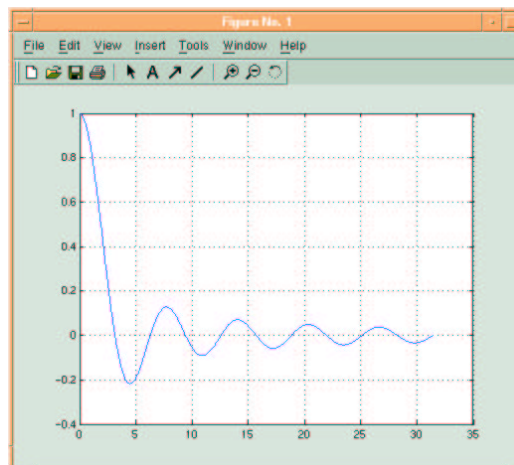


Figure 5.1: output of `firstprog`

Here is an example of a function m-file.

```
function f_result=fakultaet(n)
%FAKULTAET berechnet n!
%          FAKULTAET(n) berechnet n!=1*2*...*n
%          in rekursiver Weise.
if n==0
    f_result=1;
else
    f_result=n*fakultaet(n-1);
end
```

Note the difference between the function name `fakultaet` and the result variable, or output argument, `f_result`. This example also shows that a MATLAB function may be recursive.

The function name must coincide with the name of the m-file in which the function is stored. The function can be then called in the same way as a predefined function, e.g.

```
>> x = fakultaet(5)
x =
    120
```

## 5.2 Input and output arguments in functions. Vector syntax

In the example above, the function takes one input argument and returns the function value (= output argument). In general, several input arguments and also several output arguments are possible.

Moreover, it is often useful to define a function that can be called with a variable number of input or output arguments passed. For this purpose, the MATLAB functions `nargin` and `nargout` return the number of arguments actually passed. The following example illustrates the definition of a function with a variable number of arguments (it uses the built-in functions `sum` and `cumsum`):

```
function [s,cs] = vecsum(x,absval)
% VECSUM flexible vector element addition.
% VECSUM(X) returns the sum of the elements of X.
% [S,CS] = VECSUM(X) returns the sum and the
% cumulative sum of X.
% If VECSUM is called with two input arguments
% (X,ABSVAL) and a value of ABSVAL not equal to 0,
% it works with the vector ABS(X) instead of X.
if (nargin > 1 & absval ~= 0)
    s = sum(abs(x));
    if nargout > 1 cs = cumsum(abs(x)); end
else
    s = sum(x);
    if nargout > 1 cs = cumsum(x); end
end
```

(See section 5.5 for the explanation of the `if`-construct used here.)

The function `vecsum` may e.g. be called in the following ways:

```
>> x=[1 -2 3];
>> vecsum(x,1)
ans =
     6

>> [summe,ksumme]=vecsum(x)
summe =
     2
ksumme =
     1     -1     2
```

A very important technique in writing MATLAB functions is *vectorization*. Similarly as for standard functions (cf. section 3.2), user-defined functions can be written in such a way that they automatically accept array arguments and return arrays of the corresponding shape. Consider the following function:

```
function f=quot(x)
f=1/(1+x);
```

Function `quot` can only be called with a scalar argument `x`. Evaluation of this function for several elements of an array is only possible using a `for` loop. An efficient alternative is the vectorized version `quotv`:

```
function f=quotv(x)
f=1./(1.+x);
```

This can be called with array arguments:

```
>> x=linspace(1,5,5)
x =
     1     2     3     4     5

>> quotv(x)
ans =
    0.5000    0.3333    0.2500    0.2000    0.1667
```

## 5.3 Error handling

Error handling can be conveniently realized by means of the `error` function. For instance, `error` is often used to check the validity of arguments passed to a function, as in the following example:

```
function f=factorial(n)
if n<0 error('error: n must be nonnegative') end
...
```

A call of `error` issues the specified error message and terminates the procedure.

## 5.4 Function handles, inline functions and the feval command

Sometimes it is handy to define a function that will be used during the current MATLAB session only. For this purpose, MATLAB has a command `inline` used to define the so-called inline functions. Let

```
>> f = inline('sqrt(x.^2+y.^2)', 'x', 'y')
f =
    Inline function:
    f(x,y) = sqrt(x.^2+y.^2)
```

You can evaluate this function in the usual way.

```
>> f(3,4)
ans =
    5
```

Since vector syntax has been used in the definition of `f`, it can also be used with arrays:

```
>> A = [1 2;3 4];
>> B = ones(2);
```

then

```
>> C = f(A, B)
C =
    1.4142  2.2361
    3.1623  4.1231
```

Some functions take other functions as input parameters, for example `fplot` in section 4.1. There we specified the function to be passed by a string containing its name:

```
>> fplot('sin',[0,10]);
```

If a function defined in a function m-file is to be passed, use the name of this m-file (without extension `.m`).

In MATLAB 6, another more efficient and flexible way to specify functions as parameters is to use ‘function handles’, for example:

```
>> fplot(@sin,[0,10]);
```

If you want to use an inline function as parameter, you have to specify it in the following way:

```
>> f = inline('x.^2','x');
>> fplot(f,[0,5]);
```

In section 5.1 we have learned how to create function m-files. In order to execute functions specified by a parameter (string or function handle) in your own functions you should use the command `feval` as shown in the example below:

```
function [m1,m2]=fminmax(f,p);
% evaluates minimum and maximum of a
% function (f) over a given array of points (p)
% using MATLAB functions min and max
if (nargin==2 & nargout==2)
    t=feval(f,p);
    m1=min(t);
    m2=max(t);
else
    error('fminmax requires two input and two output arguments');
end
```

In general,

```
feval(f, input parameters for f)
```

evaluates the function specified by the first parameter (`f`) with the given input parameters for `f`.

Note the use of the `error` function in the above example (cf. section 5.3).

Examples for calling the above function during your MATLAB-session:

```
>> [m1,m2] = fminmax('sin',0:1:10);

or

>> [m1,m2] = fminmax(@sin,0:1:10);

or for an inline function:

>> f = inline('abs(x.^2-2.*x+1)', 'x');
>> [m1,m2] = fminmax(f,-5:1:5);
```

## 5.5 Control flow

To control the flow of commands you can use the following structures: **for**-loops, **while**-loops, **if-else-end** and **switch-case**. These can be used in interactive mode, but their main purpose is programming using m-files.

- **Repeating with for-loops** – the syntax of the **for**-loop is shown below:

```
for k = array
    commands
end
```

The commands between the **for** and **end** statements are executed for all values stored in the array. Note that the **end** statement is necessary in any case, even if the body of the loop consists of a single command only.

As an example, suppose that we need the values of the **sin** function at 11 evenly spaced points  $\pi \frac{n}{10}$ , for  $n = 0, 1, \dots, 10$ . To this end we use:

```
for n=0:10
    x(n+1) = sin(pi*n/10);
end

x =
    Columns 1 through 7
    0 0.3090 0.5878 0.8090 0.9511 1.0000 0.9511
    Columns 8 through 11
    0.8090 0.5878 0.3090 0.0000
```

Loops may be nested:

```
H = zeros(5);
for k=1:5
    for l=1:5
```

```

        H(k,1) = 1/(k+1-1);
    end
end

H =
    1.0000  0.5000  0.3333  0.2500  0.2000
    0.5000  0.3333  0.2500  0.2000  0.1667
    0.3333  0.2500  0.2000  0.1667  0.1429
    0.2500  0.2000  0.1667  0.1429  0.1250
    0.2000  0.1667  0.1429  0.1250  0.1111

```

The matrix  $H$  created here is called the Hilbert matrix. The first command assigns a space in memory for the matrix to be generated. This is added here to reduce the overhead that is required by loops in MATLAB. It is called *memory preallocation* and is an important technique to improve the efficiency when dealing with large matrices.

A general rule is that `for`-loops should only be used when other, vectorized methods cannot be applied. Consider the following problem. Generate a  $10 \times 10$  matrix  $A$ , with  $a_{k\ell} = \sin(k)\cos(\ell)$ . Using nested loops one can compute entries of the matrix  $A$  using the following code:

```

A = zeros(10);
for k=1:10
    for l=1:10
        A(k,l) = sin(k)*cos(l);
    end
end

```

A loop free version might look like this:

```

k = 1:10; A = sin(k)'*cos(k);

```

The first command generates a row vector  $\mathbf{k}$  consisting of integers  $1, 2, \dots, 10$ . Application of a standard function to such a vector works in an elemental sense; thus the command `sin(k)'` creates a column vector while `cos(k)` is a row vector (like  $\mathbf{k}$ ). This small piece of code again illustrates the powerful vectorization feature of MATLAB. This technique should be used whenever possible.

- **Repeating with while-loops** – the syntax of the `while`-loop is

```

while expression
    statements
end

```

This loop is used when the number of repetitions is not a priori determined. Here is a small problem that requires such a loop: The piece of code below performs a Newton iteration to find the value  $\sqrt[3]{2}$ , i.e. the solution of the equation  $x^3 - 2 = 0$ . Naturally, the number of iterations necessary to satisfy the tolerance requirement  $|x^3 - 2| < \text{tol}$  is not a priori known.

```
format long;
x=input('Startwert fuer die Iteration > ');
tol=input('Toleranz fuer Abbruch > ');
disp(x);
while (abs(x^3-2) > tol)
    x=(2+2*x^3)/(3*x^2);
    disp(x);
end
```

Let us start the script `iter.m` from the command line:

```
>> iter
Startwert fuer die Iteration > 1
Toleranz fuer Abbruch > 1e-12
1.0000000000000000
1.3333333333333333
1.2638888888888889
1.25993349344998
1.25992105001777
1.25992104989487
```

- **The if-else-end construct** – the syntax of the simplest form of the construct under discussion is:

```
if expression
    commands
end
```

This is used if there is one alternative only. Two alternatives require the following construct:

```
if expression
    commands (evaluated if expression is true)
else
    commands (evaluated if expression is false)
end
```

If there are several alternatives, we write

```

if expression1
    commands (evaluated if expression 1 is true)
elseif expression 2
    commands (evaluated if expression 2 is true)
elseif ...
    .
    .
    .
else
    commands (executed if all previous expressions evaluate to
    false)
end

```

Comparisons are performed with the aid of the following operators:

Operator	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
~=	not equal to

When dealing with logical expressions, note that 0 represents ‘false’ and 1 represents ‘true’:

```

[3==4,5==5]
ans =
     0     1

```

The logical operators are:

Operator	Description
&	logical and
	logical or
~	logical not
xor	logical exclusive or
all(x)	1 (true) if all elements of vector <b>x</b> are zero
any(x)	1 (true) if any element of vector <b>x</b> is zero

*Example:* Chebyshev polynomials  $T_n(x)$ ,  $n = 0, 1, \dots$  of the first kind are of great importance in numerical analysis. They are defined recursively as follows  $T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x)$ ,  $n = 2, 3, \dots$ ,  $T_0(x) = 1$ ,  $T_1(x) = x$ .

The implementation of this definition is easy. The following function `ChebT` computes the coefficients of Chebyshev polynomials and stores them in row vectors according to the standard MATLAB representation of polynomials (cf. section 2.4).

```
function T = ChebT(n)
% Coefficients T of the n-th Chebyshev polynomial of the first
% kind. They are stored in the descending order of powers.
t0 = 1; t1 = [1 0];
if n == 0
    T = t0;
elseif n == 1;
    T = t1;
else
    for k=2:n
        T = [2*t1 0] - [0 0 t0];
        t0 = t1;
        t1 = T;
    end
end
```

The coefficients of the cubic Chebyshev polynomial of the first kind are:

```
>> coeff = ChebT(3)
coeff =
    4  0 -3  0
```

Thus,  $T_3(x) = 4x^3 - 3x$ .

- **The switch-case construction** – with syntax:

```
switch expression (scalar or string)
    case value1 (executes if expression evaluates to value1)
        commands
    case value2 (executes if expression evaluates to value2)
        commands
    . . .
    otherwise
statements end
```

`switch` compares the input expression to each case value. Once the match is found it executes the associated commands. In the following example a random integer number  $x$  from the set  $1, 2, \dots, 10$  is generated. If  $x = 1$  or  $x = 2$ , then the message ‘Probability = 20%’ is displayed to the screen. If  $x = 3$  or 4 or 5, then the message ‘Probability = 30%’ is displayed, otherwise the message ‘Probability = 50%’ is generated. The script file `fswitch` utilizes a switch as a tool for handling all cases mentioned above:

```
% Script file fswitch.
x = ceil(10*rand); % Generate a random integer in {1,2,...,10}
switch x
    case {1,2}
        disp('Probability = 20%');
    case {3,4,5}
        disp('Probability = 30%');
    otherwise
        disp('Probability = 50%');
end
```

Note the use of curly braces after the word `case`. This creates a cell array rather than a one-dimensional array, which requires square braces.

## 5.6 Input, output, and file handling

The `disp` command can be used to display expressions or contents of variables to the screen during the execution of functions or script files:

```
disp('string expression');
```

```
    string expression
```

```
A=[3,2;2,3];
disp(A);
```

```
    3    2
    2    3
```

The `input` function can be used for requesting user input. For example, `r=input('value for r: ');`

displays `value for r:` to the screen and waits for the user to enter an expression which is then assigned to `r`.

To display formatted output to the screen you can use the function `fprintf`, with a syntax similar as in C:

```
fprintf ('approximation for pi: %6.4f\n',pi)
```

```
approximation for pi: 3.1416
```

For further variants of formatting consult the online help. Further examples for its usage can be found in section 5.8.

It is also possible to handle I/O on external files. An example for handling an external file within a function can be found in the *Ausgleichsgerade* example in section 5.8. Here we give a short description of some of the most important MATLAB file handling functions. For more details please refer to MATLAB's help facility.

- `ID=fopen(name,permission)` opens the file specified by `name` with the specified `permission` (`'r'`..read, `'w'`..write, `'a'`..append, ...) and a value is stored to `ID` with which the file can be accessed later.
- `fclose(ID)` closes the file specified by `ID`.
- `fprintf(ID,format,A,...)` writes formatted data to the file specified by `ID`. `format` is a string containing C conversion specifications. If no parameter `ID` is passed, `fprintf` writes to the screen.
- `[A,count]=fscanf(ID,format,size)` reads formatted data from the file specified by `ID`. `format` is a string containing C conversion specifications. The `size` parameter puts a limit on the number of elements to be read from the file (optional). The data read from the file are stored to `A`. If the optional output parameter `count` is used, it is assigned the number of elements successfully read.
- `fseek(ID,offset,origin)` changes the file position inside the file specified by `ID` to the location specified by the `offset` parameter relative to the location specified by the `origin` parameter. There are three possible values for `origin`: `'bof'` or `-1` refer to the beginning of the file, `'cof'` or `0` refer to the actual file position, `'eof'` or `1` refer to the end of the file. If the value of `offset` is  $> 0$ , the position is moved towards the end of the file, if it is  $< 0$ , the position is moved towards the beginning of the file.

## 5.7 Timing control

To compare different algorithms designed for solving the same problem it is often useful to evaluate the time required for a computation. This can be done by using the stopwatch timers `tic` and `toc`. `tic` starts a stopwatch timer, `toc` reads the stopwatch timer and returns the elapsed time (in seconds).

To perform [an] operation[s] and display the elapsed time, use:

```
tic;
...
operation[s];
...
toc
```

## 5.8 Examples

The examples *Quadratische Gleichung* and *Ausgleichsgerade* below are the analoga of the Fortran programs from [1].

- **Quadratic Equation**

```
function quadratische_gleichung

% Eingabe der Koeffizienten im Dialog, Kontrollausgabe:

disp('Dieses Programm loest quadratische Gleichungen');
disp('a x^2 + b x +c = 0');
disp('-----');
disp('');
disp('Koeffizienten a,b,c (reell) eingeben:');
a=input('a=');
b=input('b=');
c=input('c=');
fprintf('a = %e, b= %e, c=%e\n\n',a,b,c);

% Berechnung der Loesung(en) mit Fallunterscheidung, Ausgabe:

if a==0.0          % quadratischer Term 0

    if b==0.0      % auch linearer Term 0
```

```

    if c==0.0 % auch konstanter Term 0
        disp(['Gleichung entartet zu 0=0: '...
            'alle Koeffizienten 0']);
    else
        disp(['Gleichung lautet c==0, wobei c~=0 '...
            'spezifiziert:']);
        disp('widerspruechlich, keine Loesung');
    end

else % b~=0: echt linearer Fall
    x=-c/b;
    disp('Gleichung linear. Die Loesung lautet:');
    fprintf('x = %e\n',x);
end

else
    diskriminante=b^2-4*a*c;
    if diskriminante==0.0 % Signum der Diskriminante ermitteln
        signum=0;
    else
        signum=sign(diskriminante);
    end

switch signum
case 1 % 2 verschiedene reelle Loesungen x1, x2
    wurzel=sqrt(diskriminante);
    x1=0.5*(-b+wurzel)/a;
    x2=0.5*(-b-wurzel)/a;
    disp('Gleichung hat 2 verschiedene reelle Loesungen:');
    fprintf('x1 = %e\n',x1);
    fprintf('x2 = %e\n',x2);
case 0 % eine (doppelte) reelle Loesung x
    x=0.5*(-b)/a;
    disp('Gleichung hat eine (doppelte) reelle Loesung:');
    fprintf('x = %e\n',x);
case -1 % konjugiert komplexe Loesungen x +- i y
    wurzel=sqrt(-diskriminante);
    x=0.5*(-b)/a;
    y=0.5*wurzel/a;
    disp(['Gleichung hat ein konjugiert komplexes '...
        'Loesungspaar:']);

```

```

        fprintf('x1 = %e + i * %e\n',x,y);
        fprintf('x1 = %e - i * %e\n',x,y);
    end % switch
end %if

```

- **Linear fit**

```
function ausgleichsgerade
```

```
n=0;
```

```
% Eingabe des Dateinamens im Dialog, Kontrollausgabe:
```

```

disp('Dieses Programm berechnet eine Ausgleichsgerade');
disp('zu gegebenen Datenpaaren (x_i,y_i)');
disp('-----');
disp('');
datei=input('Name der Eingabedatei: ','s');

```

```

% Datei oeffnen, und Daten einlesen und Koeffizienten des
% Normalgleichungssystems berechnen

```

```

UNIT=fopen(datei,'r');
if UNIT== -1
    fprintf('FEHLER: %s NICHT GEFUNDEN\n',datei);
    break;
end

```

```

while 1
    [x,Count]=fscanf(UNIT,'%f',1);

    if Count==0
        break;
    end

    fseek(UNIT,1,'cof');
    y=fscanf(UNIT,'%f',1);
    fseek(UNIT,1,'cof');

    n=n+1;
    xi(n)=x;

```

```

        yi(n)=y;
end

fclose(UNIT);

fprintf('Anzahl der Datenpaare: %d\n',n);

% Koeffizienten der Ausgleichsgerade berechnen
% (Ausgleichspolynom vom Grad 1)

sol=polyfit(xi,yi,1);
a=sol(1);
b=sol(2);

% Ausgabe

disp('Ausgleichsgerade: y(x)=a*x+b, mit');
fprintf('a = %f, b = %f',a,b);

% Plot
[xsorted,index]=sort(xi);
ysorted=yi(index);
plot(xsorted,ysorted,'o:',xsorted,a*xsorted+b);

```

- **Orthogonal polynomials**

This is an example program which illustrates the manipulation of polynomial data. It performs the following task:

Let  $P_n$  be the vector space of polynomials with degree  $\leq n$ .

$(p, q) := \int_0^1 p(x)q(x)dx$  defines a scalar product in  $P_n$ .

Find a polynomial  $p_n$  of degree  $n$  which is orthogonal to  $p_0, p_1, \dots, p_{n-1}$ , where  $p_k(x) = x^k$  for  $x = 0 \dots n - 1$ .

Recall from section 2.4 that MATLAB provides some functions for the handling of polynomials, using the representation of polynomials as a row vector of coefficients with respect to the powers  $x^k$ . For example, `[1,2,4]` represents the polynomial  $x^2 + 2x + 4$ . In the next example we use MATLAB's `conv` function for multiplying polynomials. We also need to integrate polynomials. In MATLAB 6, there is a predefined function `polyint`. In MATLAB 5, we have to define `polyint` for ourselves, which is of course very easy:

```
function pint=polyint(p);
% POLYIINT(p) integrate polynomial
pint=[p 0]./[(length(p):-1:1) 1];
```

Now we have everything we need for solving our problem:

```
function opol=orthogonal1(n)
% ORTHOGONAL(n) find polynomial orthogonal to monomials
%           of degree <n

% compute the coefficients a(i) of a polynomial of degree =n,
% p(x) = a(1)*x^n + a(2)*x^(n-1) + ... + a(n)*x + a(n+1),
% such that int(x^k*p(x),x=0...1)=0 for k=0...n-1
% With the a-priori choice a(1)=1, p(x) is uniquely determined
% and a(2:n+1) can be computed by solving a system of n
% linear equations, namely:
%
% a(1)*int(x^(k+n)) + a(2)*int(x^(k+n-1)) + ... +
% + a(n)*int(x^(k+1)) + a(n+1)*int(x^k) = 0,      k=0...n-1

% preallocation of memory:
L=zeros(n,n+1);
coeff=zeros(1,2*n);

% compute the integrals x^(j), j=0...2*n-1 which occur as
% coefficients in the above equations:
for j=0:2*n-1
    monomial=[1, zeros(1,j)];
    coeff(j+1)=polyval(polyint(monomial),1);
end

% set up coefficients of linear equations
for k=0:n-1
    L(k+1,:)=coeff(k+n+1:-1:k+1);
end

% fix a(1)=1 and solve for a(2:n+1):
opol=[1,(L(:,2:n+1)\(-L(:,1)))'];
```

Since the integration of a monomial  $x^k$  is a trivial symbolic process,  $\int_0^1 x^k = 1/(k+1)$ , we may further simplify and optimize the function `orthogonal`.

For instance, the coefficients of the linear system to be solved can be extracted from the predefined ‘Hilbert matrix’  $H = \text{hilb}(n)$  which is defined as  $H_{ij} = 1/(i+j-1)$ . The following function realizes this:

```
function opol=orthogonalh(n);
% ORTHOGONAL(n) find polynomial orthogonal to monomials
%           of degree <n using Hilbert matrix

% find normed orthogonal polynomial to p(0),...p(n-1),
% with p(k)(x)=x^k for k=0..(n-1) with respect to the
% scalar product int(p(x)*q(x),x=0..1) by solving
% a system of linear equations

H=hilb(n+1);

% extract coefficients and column of solutions from
% Hilbert matrix
L=H(1:n,[n:-1:1]);
b=-H(1:n,n+1);

% solve linear equation
ps=L\b;
opol=[1,ps'];
```

- **A recursive searching function**

The following is an example of a recursive function. It checks whether a given number  $x$  is contained in a sorted array (increasing order assumed) and returns its position. The search is realized by recursive halving of the array, comparing  $x$  with the number in the middle position, and continuing with the left or right subarray, respectively.

```
function position=binary_search(sorted_array,x)
% BINARY_SEARCH find position of X in a SORTED_ARRAY
%           by recursive search
% If X is not contained in SORTED_ARRAY, 0 is returned
%
len=length(sorted_array);
if (len<=0)
    error('error: array is empty');
end
if (sum(size(sorted_array))>len+1)
```

```

        error('error: array must be 1D');
    end
    mindex=floor(len/2+1);
    mentry=sorted_array(mindex);
    position=0;
    if x==mentry
        position=mindex;
    elseif x<mentry
        if mindex>1
            position=binary_search(sorted_array(1:mindex-1),x);
        end
    else
        if mindex<len
            position=binary_search(sorted_array(mindex+1:len),x);
        end
        if position>0
            position=mindex+position;
        end
    end
end

```

- **Vector and matrix norms**

```

function normen(A,x);
% gegeben: A...quadratische Matrix, x...Vektor

% Betragssummen-Norm:
x_norm_1=norm(x,1);
% Euklid'sche Norm:
x_norm_2=norm(x,2);
% Maximum-Norm:
x_norm_max=norm(x,'inf');

% Spaltensummen-Norm:
A_norm_1=norm(A,1);
% Zeilensummen-Norm:
A_norm_max=norm(A,'inf');
% Frobenius-Norm:
A_norm_f=norm(A,'fro');
% Euklid'sche Norm:
A_norm_2=norm(A,2)
% letzteres ist identisch mit:

```

```
A_norm_2=max(sqrt(eig(A'*A)))
```

## 5.9 The symbolic math toolbox

It is also possible to perform symbolic computations with MATLAB using the 'Symbolic Math Toolbox', which uses a MAPLE kernel.

First we have to explicitly declare symbolic variables using the functions `sym` or `syms`:

```
>> syms x
```

Now we can construct arithmetic expressions using your symbolic variable(s), like

```
>> f=exp(x)*x^2
f =
exp(x)*x^2
```

and perform symbolic calculations, like

```
>> g=int(f,x)
g =
exp(x)*x^2-2*x*exp(x)+2*exp(x)
```

```
>> factor(g)
ans =
exp(x)*(x^2-2*x+2)
```

The `factor` function can also be used for factorizing integers:

```
>> factor(sym('384'))
ans =
(2)^7*(3)
```

We could also compute a definite integral of the expression `f`:

```
>> a = int(f,x,0,1)
a =
exp(1)-2
```

Note that the results of these operations are again symbolic expressions. Conversion of results of symbolic operations into MATLAB floating-point numbers can be done with the MATLAB function `double`:

```
>> a
a =
    exp(1)-2;

>> af = double(a)
af =
    0.7183
```

To display a symbolic expression in easier-to-read form, you can use the command `pretty`:

```
>> pretty (g)
                2
    exp(x) (x  - 2 x + 2)
```

The symbolic math toolbox can also be used for calculating exact rational expressions,

```
>> simplify(sym('3/8+7/12'))
ans =
    23/24
```

or for calculating expressions with arbitrary precision using `vpa`:

```
>> vpa('log(2)',50)
ans =
.69314718055994530941723212145817656807550013436026
```

displays the value of the logarithm accurate to 50 decimals. Note that this is not the same as

```
>> vpa(log(2),50);
```

because here `ln(2)` is first calculated with default MATLAB precision and then displayed with 50 digits, but the result will only be accurate to about 16 digits. So it is very important to pass over a *string* or a *symbolic expression* to `vpa`, or it will not produce the results expected.

There are many other functions for handling expressions containing symbolic variables. A short list of some of the most important ones is:

<code>sym / syms</code>	define symbolic variable
<code>diff</code>	differentiate symbolic expression
<code>int</code>	integrate symbolic expression
<code>pretty</code>	display symbolic expression
<code>double</code>	convert symbolic expr. to MATLAB float
<code>vpa</code>	convert symb. expr. to variable precision float
<code>limit</code>	compute symb. limit of symb. expr.
<code>taylor</code>	calculate taylor pol. for symb. expr.
<code>factor</code>	factorize symbolic expression
<code>simplify</code>	simplify symbolic expression
<code>simple</code>	find most simple equivalent to symb. expr.
<code>expand</code>	expand symbolic expression
<code>convert</code>	convert symbolic expression
<code>ezplot</code>	plot graph of symbolic expression
<code>solve</code>	solve equation of symb. expressions

Many MATLAB functions for numeric expressions work for symbolic expressions as well.

# Appendix A

## A nontrivial application: Linear Programming

### A.1 Basic feasible solutions

The *standard form* of a linear programming problem is formulated as follows. Given matrix  $A \in \mathbb{R}^{m \times n}$  and two vectors  $c \in \mathbb{R}^n$  and  $b \in \mathbb{R}^m$ ,  $b \geq 0$ , find a vector  $x \in \mathbb{R}^n$  such that

$$\begin{aligned} \min \quad & z = c^\top x \\ \text{subject to} \quad & Ax = b, \\ & x \geq 0. \end{aligned}$$

The MATLAB functions that are presented in this sections make calls to functions named `vr` and `delcols`. These functions should be saved in the directory holding other m-files that are used in this tutorial.

```
function e = vr(m,i)
% The ith coordinate vector e in the m-dimensional Euclidean
% space.
e = zeros(m,1);
e(i) = 1;

function d = delcols(d)
% Delete duplicated columns of the matrix d.
d = union(d',d', 'rows')';
n = size(d,2);
j = [];
for k =1:n
    c = d(:,k);
```

```

    for l=k+1:n
        if norm(c - d(:,l),'inf') <= 100*eps
            j = [j l];
        end
    end
end
end
if ~isempty(j) j = sort(j);
    d(:,j) = [];
end

```

The function `vert=feassol(A,b)` computes all basic feasible solutions, if any, to the system of constraints in standard form.

```

function vert = feassol(A, b)
% Basic feasible solutions vert to the system of constraints
%
%  $Ax = b, x \geq 0.$ 
% They are stored in columns of the matrix vert.
[m, n] = size(A);
warning off
b = b(:);
vert = [ ];
if (n >= m)
    t = nchoosek(1:n,m);
    nv = nchoosek(n,m);
    for i=1:nv
        y = zeros(n,1);
        x = A(:,t(i,:))\b;
        if all(x >= 0 & (x ~= inf & x ~= -inf))
            y(t(i,:)) = x;
            vert = [vert y];
        end
    end
end
else
    error(['Number of equations is greater than the number '...
        'of variables.'])
end
if ~isempty(vert)
    vert = delcols(vert);
else
    vert = [ ];
end

```

To illustrate functionality of this code consider the following system of constraints:

$$\begin{aligned}x_1 + x_2 &\leq 6 \\x_2 &\leq 3 \\x_1, x_2 &\geq 0\end{aligned}$$

To put this system in standard form two *slack variables*  $x_3$  and  $x_4$  are added. The constraint matrix  $A$  and the right hand side  $b$  are:

```
>> A = [1 1 1 0; 0 1 0 1];
>> b = [6; 3];
>> vert = feassol(A, b)
vert =
    0    0    3    6
    0    3    3    0
    6    3    0    0
    3    0    0    3
```

To obtain values of the legitimate variables  $x_1$  and  $x_2$  it suffices to extract rows one and two of the matrix `vert`:

```
>> vert = vert(1:2, :)
vert =
    0    0    3    6
    0    3    3    0
```

## A.2 Extreme points and extreme directions of the constraint set

The type of problem discussed in this section is formulated as follows. Given a polyhedral set  $X = \{x : Ax \leq b \text{ or } Ax \geq b, x \geq 0\}$  find all *extreme points*  $t$  of  $X$ . If  $X$  is unbounded, then the addition to finding the extreme points  $t$  its *extreme directions*  $d$  should be determined as well. To this end we will assume that the constraint set does not involve the equality constraints. If the LP problem has the equality constraint, then one can replace it by two inequality constraints. This is based on the following trivial observation: the equality  $a = b$  is equivalent to the system of inequalities  $a \leq b$  and  $a \geq b$ . Knowledge of the extreme points and extreme directions of the polyhedral set  $X$  is critical for a full mathematical description of this set. If set  $X$  is bounded, than a *convex combination* of its extreme points gives a point in this set. For the unbounded sets, however, a convex combination of its extreme points and a *linear combination*, with positive coefficients, of its

extreme directions gives a point in set  $X$ . For more details, the interested reader is referred to [4], Chapter 2.

Extreme points of the set in question can be computed using the function `extrpts`:

```
function vert = extrpts(A, rel, b)
% Extreme points vert of the polyhedral set
%           X = {x: Ax <= b or Ax >= b, x >= 0}.
% Inequality signs are stored in the string rel, e.g.,
% rel = '<<>' stands for <= , <= , and >= , respectively.
[m, n] = size(A);
nlv = n;
for i=1:m
    if(rel(i) == '>')
        A = [A -vr(m,i)];
    else
        A = [A vr(m,i)];
    end
    if b(i) < 0
        A(i,:) = -A(i,:);
        b(i) = -b(i);
    end
end
warning off
[m, n]= size(A);
b = b(:);
vert = [];
if (n >= m)
    t = nchoosek(1:n,m);
    nv = nchoosek(n,m);
    for i=1:nv
        y = zeros(n,1);
        x = A(:,t(i,:))\b;
        if all(x >= 0 & (x ~= inf & x ~= -inf))
            y(t(i,:)) = x;
            vert = [vert y];
        end
    end
else
    error(['Number of equations is greater than the number '...
        'of variables'])
end
```

```
end
vert = delcols(vert);
vert = vert(1:nlv,:);
```

Consider the polyhedral set  $X$  given by the following inequalities:

$$\begin{aligned} -x_1 + x_2 &\leq 1 \\ -0.1x_1 + x_2 &\leq 2 \\ x_1, x_2 &\geq 0 \end{aligned}$$

To compute its extreme points we define:

```
>> A = [-1 1; -0.1 1];
>> rel = '<<';
>> b = [1; 2];
```

and next run the m-file `extrpts`

```
>> vert = extrpts(A, rel, b)
vert =
     0         0     1.1111
     0     1.0000     2.1111
```

Recall that the extreme points are stored in columns of the matrix `vert`.

Extreme directions, if any, of the polyhedral set  $X$  can be computed using the function `extrdir`:

```
function d = extrdir(A, rel, b)
% Extreme directions d of the polyhedral set
%      X = {x: Ax <= b, or Ax >= b, x >= 0}.
% Matrix A must be of the full row rank.
[m, n] = size(A);
n1 = n;
for i=1:m
    if(rel(i) == '>')
        A = [A -vr(m,i)];
    else
        A = [A vr(m,i)];
    end
end
end
[m, n] = size(A);
A = [A;ones(1,n)]
b = [zeros(m,1);1]
```

```

d = feassol(A,b)
if ~isempty(d)
    d1 = d(1:n1,:);
    d = delcols(d1);
    s = sum(d);
    for i=1:n1
        d(:,i) = d(:,i)/s(i);
    end
else
    d = [];
end

```

The function `extrdir` returns the empty set operator `[]` for bounded polyhedral sets.

We will test this function using the polyhedral set that is defined in the last example of this section. We have:

```

>> d = extrdir(A, rel, b)
d =
    1.0000    0.9091
         0    0.0909

```

Again, the directions in question are stored in columns of the output matrix `d`.

### A.3 Solving the LP problem geometrically

A solution to the LP problem with two legitimate variables  $x_1$  and  $x_2$  can be found geometrically in three steps. First, the feasible region described by the constraint system  $Ax \leq b$  or  $Ax \geq b$  with  $x \geq 0$  is drawn. Next, a direction of the level line  $z = c_1x_1 + c_2x_2$  is determined. Finally moving the level line one can find easily vertex (extreme point) of the feasible region where the minimum or maximum value of  $z$  is attained or conclude that the objective function is unbounded.

The function `drawfr` implements the two initial steps of this method :

```

function drawfr(c, A, rel, b)
% Graphs of the feasible region and the line level
% of the LP problem with two legitimate variables
%
%           min (max) z = c*x
%           Subject to Ax <= b (or Ax >= b),
%                   x >= 0

```

```

clc
[m, n] = size(A);
if n ~= 2
    str = ['Number of the legitimate variables must be equal '...
          'to 2'];
    msgbox(str, 'Error Window', 'error')
    return
end
vert = extrpts(A,rel,b);
if isempty(vert)
    disp(sprintf('\n Empty feasible region'))
    return
end
vert = vert(1:2,:);
vert = delcols(vert);
d = extrdir(A,rel,b);
if ~isempty(d)
    msgbox('Unbounded feasible region', 'Warning Window', 'warn')
    disp(sprintf('\n Extreme directions of the constraint set'))
    d
    disp(sprintf('\n Extreme points of the constraint set'))
    vert
    return
end
t1 = vert(1,:);
t2 = vert(2,:);
z = convhull(t1,t2);
hold on
patch(t1(z),t2(z), 'r')
h = .25;
mit1 = min(t1)-h;
mat1 = max(t1)+h;
mit2 = min(t2)-h;
mat2 = max(t2)+h;
if c(1) ~= 0 & c(2) ~= 0
    s1 = -c(1)/c(2);
    if s1 > 0
        z = c(:)'*[mit1;mit2];
        a1 = [mit1 mat1];
        b1 = [mit2 (z-c(1)*mat1)/c(2)];
    end
end

```

```

else
    z = c(:)'*[mat1;mit2];
    a1 = [mit1 mat1];
    b1 = [(z-c(1)*mit1)/c(2) mit2];
end
elseif c(1) == 0 & c(2) ~= 0
    z = 0;
    a1 = [mit1 mat1];
    b1 = [0,0];
else
    z = 0;
    a1 = [0 0];
    b1 = [mit2 mat2];
end
h = plot(a1,b1);
set(h,'linestyle','--')
set(h,'linewidth',2)
str = ['Feasible region and a level line with the '...
       'objective value = '];
title([str,num2str(z)])
axis([mit1 mat1 mit2 mat2])
h = get(gca,'Title');
set(h,'FontSize',11)
xlabel('x_1')
h = get(gca,'xlabel');
set(h,'FontSize',11)
ylabel('x_2')
h = get(gca,'ylabel');
set(h,'FontSize',11)
grid
hold off

```

To test this function consider the LP problem with five constraints

```

>> c = [-3 5];
>> A = [-1 1;1 1;1 1;3 -1;1 -3];
>> rel = '<><<<';
>> b = [1;1;5;7;1];

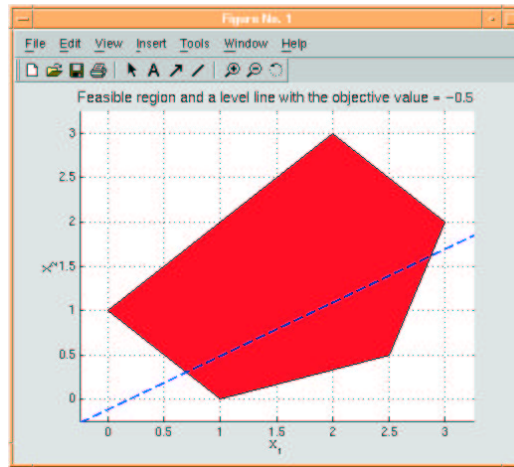
```

Graphs of the feasible region and the level line are shown below.

```

>> drawfr(c, A, rel, b)

```

Figure A.1: output of `drawfr(c, A, rel, b)`

Let us note that for the minimization problem the optimal solution occurs at

$$x = \begin{bmatrix} 2.5 \\ 0.5 \end{bmatrix},$$

while the maximum of the objective function is attained at

$$x = \begin{bmatrix} 2 \\ 3 \end{bmatrix}.$$

# Appendix B

## Function Reference

This purpose of this chapter is to give an overview of functions built-in to MATLAB, with a short description to each command. The commands are grouped by subject as listed below. For detailed information please refer to the MATLAB-Helpdesk.

- General Purpose Commands
- Math Functions
- Matrix Function
- Language Constructs and Debugging
- Sparse Matrix Functions
- Matrix Functions - Numerical Linear Algebra
- Data Analysis and Fourier Transform Functions
- Polynomial and Interpolation Functions
- Function Functions - Nonlinear Numerical Methods
- Character String Functions
- Low-Level File IO Functions
- Plotting and Data Visualization
- Graphical User Interface Creation
- Symbolic Math Toolbox
- Miscellaneous

## General Purpose Commands

- Managing Commands and Functions
  - `help` Online help for MATLAB functions and M-files
  - `helpdesk` Display Help Desk page in Web browser, giving access to extensive help
  - `helpwin` Display Help Window, providing access to help for all commands
  - `lookfor` Keyword search through all help entries
  - `type` List file
  - `what` Directory listing of M-files, MAT-files, and MEX-files
  - `whatsnew` Display README files for MATLAB and toolboxes
  - `which` Locate functions and files
- Managing Variables and the Workspace
  - `clear` Remove items from memory
  - `disp` Display text or array
  - `length` Length of vector
  - `load` Retrieve variables from disk
  - `save` Save workspace variables on disk
  - `saveas` Save figure or model using specified format
  - `size` Array dimensions
  - `who, whos` List directory of variables in memory
- Controlling the Command Window
  - `clc` Clear command window
  - `format` Control the output display format
  - `more` Control paged output for the command window
- Working with Files and the Operating Environment
  - `cd` Change working directory
  - `copyfile` Copy file
  - `delete` Delete files and graphics objects
  - `diary` Save session in a disk file
  - `dir` Directory listing
  - `mkdir` Make directory
  - `open` Open files based on extension
  - `pwd` Display current directory
  - `!` Execute operating system command
- Starting and Quitting MATLAB
  - `matlabrc` MATLAB startup M-file
  - `quit` Terminate MATLAB

## Math Functions

- Operators and Special Characters

*	Matrix multiplication
~	Matrix power
\	Backslash or left division
/	Slash or right division
:	Colon
%	Comment
=	Assignment
==	Equality
< >	Relational operators
&	Logical and
	Logical or
~	Logical not
xor	Logical exclusive or

- Elementary Math Functions

abs	Absolute value and complex magnitude
acos, acosh	Inverse cosine and inverse hyperbolic cosine
acot, acoth	Inverse cotangent and inverse hyperbolic cotangent
acsc, acsch	Inverse cosecant and inverse hyperbolic cosecant
angle	Phase angle
asec, asech	Inverse secant and inverse hyperbolic secant
asin, asinh	Inverse sine and inverse hyperbolic sine
atan, atanh	Inverse tangent and inverse hyperbolic tangent
atan2	Four-quadrant inverse tangent
ceil	Round toward infinity
complex	Construct complex data from real and imaginary components
conj	Complex conjugate
cos, cosh	Cosine and hyperbolic cosine
cot, coth	Cotangent and hyperbolic cotangent
csc, csch	Cosecant and hyperbolic cosecant
exp	Exponential
fix	Round towards zero
floor	Round towards minus infinity
gcd	Greatest common divisor
imag	Imaginary part of a complex number
lcm	Least common multiple

<code>log</code>	Natural logarithm
<code>log2</code>	Base 2 logarithm and dissect floating-point numbers into exponent and
<code>log10</code>	Common (base 10) logarithm
<code>mod</code>	Modulus (signed remainder after division)
<code>nchoosek</code>	Binomial coefficient or all combinations
<code>real</code>	Real part of complex number
<code>rem</code>	Remainder after division
<code>round</code>	Round to nearest integer
<code>sec, sech</code>	Secant and hyperbolic secant
<code>sign</code>	Signum function
<code>sin, sinh</code>	Sine and hyperbolic sine
<code>sqrt</code>	Square root
<code>tan, tanh</code>	Tangent and hyperbolic tangent
• Specialized Math Functions	
<code>erf, erfc, erfcx, erfinv</code>	Error functions
<code>expint</code>	Exponential integral
<code>factorial</code>	Factorial function
<code>gamma, gammainc, gammaln</code>	Gamma functions
<code>pow2</code>	Base 2 power and scale floating-point numbers
<code>rat, rats</code>	Rational fraction approximation
• Function Functions - Nonlinear Numerical Methods	
<code>dblquad</code>	Numerical double integration
<code>fminbnd</code>	Minimize a function of one variable
<code>fminsearch</code>	Minimize a function of several variables
<code>fzero</code>	Zero of a function of one variable
<code>ode45, ode23, ode113, ode15s, ode23s, ode23t, ode23tb</code>	Solve differential equations
<code>odefile</code>	Define a differential equation problem for ODE solvers
<code>odeget</code>	Extract properties from options structure created with <code>odeset</code>
<code>odeset</code>	Create or alter options structure for input to ODE solvers
<code>quad, quad8</code>	Numerical evaluation of integrals

## Matrix Functions

- Elementary Matrices
  - `eye` Identity matrix
  - `linspace` Generate linearly spaced vectors
  - `logspace` Generate logarithmically spaced vectors
  - `ones` Create an array of all ones
  - `rand` Uniformly distributed random numbers and arrays
  - `randn` Normally distributed random numbers and arrays
  - `zeros` Create an array of all zeros
  - `:` (colon) Regularly spaced vector
- Matrix Manipulation
  - `cat` Concatenate arrays
  - `diag` Diagonal matrices and diagonals of a matrix
  - `tril` Lower triangular part of matrix
  - `triu` Upper triangular part of matrix
  - `reshape` Reshape array
  - `:` (colon) Index into array, rearrange array
- Specialized Matrices
  - `compan` Companion matrix
  - `gallery` Test matrices
  - `hilb` Hilbert matrix
  - `invhilb` Inverse of the Hilbert matrix
  - `magic` Magic square
  - `toeplitz` Toeplitz matrix

## Language Constructs and Debugging

- MATLAB as a Programming Language
  - `feval` Function evaluation
  - `function` Function M-files
  - `global` Define global variables
- Logical Functions
  - `all` Test to determine if all elements are nonzero
  - `any` Test for any nonzeros
  - `exist` Check if a variable or file exists
  - `find` Find indices and values of nonzero elements
  - `logical` Convert numeric values to logical

- Control Flow
  - `break` Terminate execution of for loop or while loop
  - `case` Case switch
  - `catch` Begin catch block
  - `else` Conditionally execute statements
  - `elseif` Conditionally execute statements
  - `end` Terminate for, while, switch, try, and if statements or indicate last
  - `error` Display error messages
  - `for` Repeat statements a specific number of times
  - `if` Conditionally execute statements
  - `otherwise` Default part of switch statement
  - `return` Return to the invoking function
  - `switch` Switch among several cases based on expression
  - `try` Begin try block
  - `warning` Display warning message
  - `while` Repeat statements an indefinite number of times
- Interactive Input
  - `input` Request user input
  - `pause` Halt execution temporarily

### Sparse Matrix Functions

- Elementary & Sparse Matrices
  - `spdiags` Extract and create sparse band and diagonal matrices
  - `speye` Sparse identity matrix
  - `sprand` Sparse uniformly distributed random matrix
  - `sprandn` Sparse normally distributed random matrix
  - `sprandsym` Sparse symmetric random matrix
- Full to Sparse Conversion
  - `full` Convert sparse matrix to full matrix
  - `sparse` Create sparse matrix
- Working with Nonzero Entries of Sparse Matrices
  - `nnz` Number of nonzero matrix elements
  - `nonzeros` Nonzero matrix elements
- Visualizing Sparse Matrices
  - `spy` Visualize sparsity pattern
- Norm, Condition Number, and Rank
  - `condest` 1-norm matrix condition number estimate
  - `normest` 2-norm estimate

- Sparse Systems of Linear Equations
  - `bicg` BiConjugate Gradients method
  - `bicgstab` BiConjugate Gradients Stabilized method
  - `cgs` Conjugate Gradients Squared method
  - `cholinc` Sparse Incomplete Cholesky and Cholesky-Infinity factorizations
  - `cholupdate` Rank 1 update to Cholesky factorization
  - `gmres` Generalized Minimum Residual method (with restarts)
  - `luinc` Incomplete LU matrix factorizations
  - `pcg` Preconditioned Conjugate Gradients method
  - `qmr` Quasi-Minimal Residual method
  - `qr` Orthogonal-triangular decomposition
- Sparse Eigenvalues and Singular Values
  - `eigs` Find eigenvalues and eigenvectors
  - `svds` Find singular values

## Matrix Functions - Numerical Linear Algebra

- Matrix Analysis
  - `cond` Condition number with respect to inversion
  - `det` Matrix determinant
  - `norm` Vector and matrix norms
  - `null` Null space of a matrix
  - `orth` Range space of a matrix
  - `rank` Rank of a matrix
  - `rcond` Matrix reciprocal condition number estimate
  - `subspace` Angle between two subspaces
  - `trace` Sum of diagonal elements
- Matrix Functions
  - `expm` Matrix exponential
  - `funm` Evaluate functions of a matrix
- Linear Equations
  - `chol` Cholesky factorization
  - `inv` Matrix inverse
  - `lu` LU matrix factorization
  - `lsqnonneg` Nonnegative least squares
  - `pinv` Moore-Penrose pseudoinverse of a matrix
  - `qr` Orthogonal-triangular decomposition

- Eigenvalues and Singular Values
  - `eig` Eigenvalues and eigenvectors
  - `gsvd` Generalized singular value decomposition
  - `hess` Hessenberg form of a matrix
  - `poly` Polynomial with specified roots
  - `schur` Schur decomposition
  - `svd` Singular value decomposition

### Data Analysis and Fourier Transform Functions

- Basic Operations
  - `convhull` Convex hull
  - `cumprod` Cumulative product
  - `cumsum` Cumulative sum
  - `factor` Prime factors
  - `max` Maximum elements of an array
  - `mean` Average or mean value of arrays
  - `median` Median value of arrays
  - `min` Minimum elements of an array
  - `primes` Generate list of prime numbers
  - `prod` Product of array elements
  - `sort` Sort elements in ascending order
  - `std` Standard deviation
  - `sum` Sum of array elements
  - `trapz` Trapezoidal numerical integration
  - `var` Variance
- Finite Differences
  - `del2` Discrete Laplacian
  - `diff` Differences and approximate derivatives
  - `gradient` Numerical gradient
- Correlation
  - `corrcoef` Correlation coefficients
  - `cov` Covariance matrix
- Vector Functions
  - `cross` Vector cross product
- Filtering and Convolution
  - `conv` Convolution and polynomial multiplication
  - `conv2` Two-dimensional convolution
  - `deconv` Deconvolution and polynomial division

- Fourier Transforms
  - `abs` Absolute value and complex magnitude
  - `angle` Phase angle
  - `cplxpair` Sort complex numbers into complex conjugate pairs
  - `fft` One-dimensional fast Fourier transform
  - `fft2` Two-dimensional fast Fourier transform

## Polynomial and Interpolation Functions

- Polynomials
  - `conv` Convolution and polynomial multiplication
  - `deconv` Deconvolution and polynomial division
  - `poly` Polynomial with specified roots
  - `polyder` Polynomial derivative
  - `polyeig` Polynomial eigenvalue problem
  - `polyfit` Polynomial curve fitting
  - `polyint` Integral of polynomial
  - `polyval` Polynomial evaluation
  - `polyvalm` Matrix polynomial evaluation
  - `roots` Polynomial roots
- Data Interpolation
  - `griddata` Data gridding
  - `interp1` One-dimensional data interpolation (table lookup)
  - `interp2` Two-dimensional data interpolation (table lookup)
  - `interp3` Three-dimensional data interpolation (table lookup)
  - `spline` Cubic spline interpolation

## Character String Functions

- String Manipulation
  - `strcat` String concatenation
  - `strcmp` Compare strings
- String to Number Conversion
  - `char` Create character array (string)
  - `int2str` Integer to string conversion
  - `num2str` Number to string conversion
  - `sprintf` Write formatted data to a string
  - `sscanf` Read string under format control
  - `str2double` Convert string to double-precision value
  - `str2num` String to number conversion
- Radix Conversion
  - `bin2dec` Binary to decimal number conversion
  - `dec2bin` Decimal to binary number conversion

### Low-Level File IO Functions

- File Opening and Closing
  - `fclose` Close one or more open files
  - `fopen` Open a file or obtain information about open files
- Unformatted IO
  - `fread` Read binary data from file
  - `fwrite` Write binary data to a file
- Formatted I/O
  - `fgets` Return the next line of a file as a string with line terminator(s)
  - `fprintf` Write formatted data to file
  - `fscanf` Read formatted data from file
- File Positioning
  - `feof` Test for end-of-file
  - `frewind` Rewind an open file
  - `fseek` Set file position indicator
  - `ftell` Get file position indicator
- String Conversion
  - `sprintf` Write formatted data to a string
  - `sscanf` Read string under format control

### Plotting and Data Visualization

- Basic Plots and Graphs
  - `bar` Vertical bar chart
  - `hist` Plot histograms
  - `hold` Hold current graph
  - `loglog` Plot using log-log scales
  - `pie` Pie plot
  - `plot` Plot vectors or matrices.
  - `polar` Polar coordinate plot
  - `semilogx` Semi-log scale plot
  - `semilogy` Semi-log scale plot
- Three-Dimensional Plotting
  - `bar3` Vertical 3-D bar chart
  - `plot3` Plot lines and points in 3-D space

- Plot Annotation and Grids
  - `grid`     Grid lines for 2-D and 3-D plots
  - `title`    Titles for 2-D and 3-D plots
  - `xlabel`   X-axis labels for 2-D and 3-D plots
  - `ylabel`   Y-axis labels for 2-D and 3-D plots
  - `zlabel`   Z-axis labels for 3-D plots
- Surface, Mesh, and Contour Plots
  - `contour`   Contour (level curves) plot
  - `surf`      3-D shaded surface graph
  - `fplot`     Plot a function

### Symbolic Math Toolbox

<code>sym/syms</code>	Define symbolic variable
<code>diff</code>	Differentiate expression
<code>int</code>	Integrate expression
<code>pretty</code>	Pretty-print
<code>double</code>	Convert expression to MATLAB float
<code>vpa</code>	Convert expression to variable precision float
<code>limit</code>	Compute limit
<code>taylor</code>	Calculate taylor polynomial
<code>factor</code>	Factorize expression
<code>simplify</code>	Simplify expression
<code>simple</code>	Find most simple equivalent to expression
<code>expand</code>	Expand expression
<code>collect</code>	Collect coefficients in expression
<code>convert</code>	Convert expression
<code>subs</code>	Perform substitution
<code>ezplot</code>	Plot graph of an expression
<code>solve</code>	Solve equation
<code>dsolve</code>	Solve ordinary differential equation
...	

**Miscellaneous**

- Special Variables and Constants
 

<code>ans</code>	The most recent answer
<code>eps</code>	Floating-point relative accuracy
<code>i</code>	Imaginary unit
<code>Inf</code>	Infinity
<code>j</code>	Imaginary unit
<code>NaN</code>	Not-a-Number
<code>nargin, nargout</code>	Number of function arguments
<code>pi</code>	Ratio of a circle's circumference to its diameter
<code>realmax</code>	Largest positive floating-point number
<code>realmin</code>	Smallest positive floating-point number
<code>varargin, varargout</code>	Pass or return variable numbers of arguments
- Time and Dates
 

<code>cputime</code>	Elapsed CPU time
<code>now</code>	Current date and time
<code>tic, toc</code>	Stopwatch timer
- Coordinate System Conversion
 

<code>cart2pol</code>	Transform Cartesian coordinates to polar or cylindrical
<code>cart2sph</code>	Transform Cartesian coordinates to spherical
<code>pol2cart</code>	Transform polar or cylindrical coordinates to Cartesian
<code>sph2cart</code>	Transform spherical coordinates to Cartesian
- Graphical User Interface Creation
 

<code>dialog</code>	Create a dialog box
<code>menu</code>	Generate a menu of choices for user input

# Appendix C

## Problems

- Suppose  $n$  is a positive integer. Write a one-liner (i.e. a single MATLAB statement  $\mathbf{v} = \dots$  that generates the vector  $[2 \ 4 \ 8 \ \dots \ 2^n]$ .
  - Suppose  $n$  is a positive integer and  $w$  is a scalar parameter. Write a one-liner (i.e. a single MATLAB statement  $\mathbf{A} = \dots$  that generates the  $n \times n$ -matrix  $A$  with  $a_{ij} = w^{(i-1)(j-1)}$ .
- Suppose a vector  $x = [x_1 \ \dots \ x_n]^T$  of length  $n$  is given. Write a sequence of MATLAB statements that generates its so-called ( $n \times n$ ) *companion matrix*

$$\begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ -x_n & -x_{n-1} & -x_{n-2} & \dots & -x_1 \end{pmatrix}.$$

- Write MATLAB functions `schur(A,B)` and `lie(A,B)` that compute
  - the *Schur product* (componentwise product)  $[A \circ B]$  of the matrices  $A$  and  $B$ ,

$$[A \circ B]_{ij} = a_{ij}b_{ij},$$

- the *Lie product*  $[A, B]$  of the matrices  $A$  and  $B$ ,

$$[A, B]_{ij} = \sum_{k=1}^n a_{ik}b_{kj} - b_{ik}a_{kj}.$$

An error message should be printed if the corresponding product is not well-defined due to dimension conflicts.

4. By using the `while` construct, compute approximate function values for  $\sin x$ ,  $x = 0.01, 0.02, \dots, 1$ , using the series

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots$$

Consider only the terms necessary to guarantee accuracy up to the fourth digit and compare the result with the one given by the MATLAB function `sin`.

5. You cannot use `for` or `while` loops:
- Write a MATLAB function `[in, fr] = infr(x)` that takes an array  $x$  of real numbers and returns arrays  $in$  and  $fr$  holding the integer and decimal parts, respectively, of all numbers in the array  $x$ .
  - Write a MATLAB function `d = rep(b, m)`. It takes an array of numbers  $b$  and the array  $m$  of positive integers and returns an array  $d$  whose each entry is taken from the array  $b$  and is duplicated according to the corresponding value in the array  $m$ . For instance, if  $b = [1\ 2]$  and  $m = [2\ 3]$ , then  $d = [1\ 1\ 2\ 2\ 2]$ .
6. Write a MATLAB function `d = dsc(c)` that takes a one-dimensional array of numbers  $c$  and returns an array  $d$  consisting of all numbers in the array  $c$  with all neighbouring duplicate numbers removed. For instance, if  $c = [1\ 2\ 2\ 2\ 3\ 1]$ , then  $d = [1\ 2\ 3\ 1]$ . You cannot use `for` or `while` loops.
7. You cannot use `for` or `while` loops:
- To find the largest (smallest) entry of a vector you can use the function `max` (`min`). Suppose that these functions are not available. How would you calculate
    - the largest entry of a vector ?
    - the smallest entry of a vector?
  - Let  $a$  be a vector of integers.
    - Create a vector  $b$  whose components are the even entries of the vector  $a$ .

- ii. Repeat part (i) where now  $b$  consists of all odd entries of the vector  $a$ .

Hint: Function `logical` is often used to logical tests. Another useful function you may consider to use is `rem(x, y)` – the remainder after division of  $x$  by  $y$ .

8. The Pascal triangle holds coefficients in the series expansion of  $(1+x)^n$ , where  $n = 0, 1, 2, \dots$ . Write a MATLAB function `t = pasctri(n)` that generates the Pascal triangle  $t$  up to level  $n$ . Remark: Two-dimensional arrays in MATLAB must have the same number of columns in each row. In order to avoid error messages you have to add a certain number of zero entries to the right of last nonzero entry in each row of  $t$  but one.
9. The  $n \times n$  Frank matrix  $F_n = (f_{ij})$  is an upper Hessenberg matrix whose nonzero elements are given by  $f_{ij} = n-j+1$  if  $i > j$ , and by  $f_{ij} = n-j$  whenever  $j = i-1$ . Write a MATLAB function `frank(n)` which returns  $F_n$  for given  $n$  and test it for several values of  $n$ .
10. Write a MATLAB function `posdef(A,b)` which checks whether, for a given real  $n \times n$ -matrix  $A$  and a given column vector  $b$  of length  $n$ , the  $n \times n$ -matrix  $M := B*A + A^T*B - b*b^T$  is positive definite. (Here,  $B$  denotes the diagonal matrix  $B = \text{Diag}(b_1, \dots, b_n)$ .)  
Return the matrix  $M$  and the information whether it is positive definite in an appropriate way. Check the data  $A$  and  $b$  for compatibility.
11. If  $A$  is a symmetric  $n \times n$  matrix, then `R = chol(A)` computes its Cholesky factorization, i.e., an upper triangular matrix  $R$  such that  $A = R^T*R$ . Use this factorization in a MATLAB segment which computes  $L$  and  $D$  in the factorization  $A = L*D*L^T$ , with  $D$  diagonal and  $L$  unit lower triangular (i.e., with 1 along the diagonal).
12. Write MATLAB function `[nonz, mns] = matstat(A)` that takes as the input argument a real matrix  $A$  and returns all nonzero entries of  $A$  in the column vector `nonz`. Second output parameter `mns` holds values of the unweighted arithmetic means of all columns of  $A$ .
13. Let  $A$  be an  $m$ -by- $n$  and let  $B$  be an  $n$ -by- $p$  matrix. Computing the product  $C = AB$  requires  $mnp$  multiplications. If either  $A$  or  $B$  has a special structure, then the number of multiplications can be reduced drastically. Let  $A$  be a full matrix of dimension  $m$ -by- $n$  and let  $B$  be an upper triangular matrix of dimension  $n$ -by- $n$  whose nonzero entries are equal to one. The product  $AB$  can be calculated without using

a single multiplication. Write an algorithm for computing the matrix product  $C = A*B$  that does not require multiplications.

14. Write a MATLAB function `prod3(B,C,D)` which computes the product  $A = B*C*D$  for given matrices  $B, C$  and  $D$ . The order of multiplication should be determined to minimize the number of arithmetic operations required. An error message should be printed if either  $B*C$  or  $C*D$  is not well-defined due to dimension conflicts.
15. For given matrices  $A$  ( $m \times m$ ),  $B$  ( $m \times n$ ),  $C$  ( $n \times n$ ) and column vectors  $g$  (dimension  $m$ ),  $h$  (dimension  $n$ ), write a MATLAB segment which solves the linear system

$$\begin{bmatrix} A & B \\ 0 & C \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} g \\ h \end{bmatrix}$$

by means of solving two smaller systems.

Choose appropriate data and check your solution by direct solution of the full system.

16. Consider the linear system  $A*x = b$  where

$$A = \begin{bmatrix} 4 & 1 & 1 \\ 1 & 4 & 1 \\ 1 & 1 & 4 \end{bmatrix}, \quad b = \begin{bmatrix} 9 \\ 12 \\ 15 \end{bmatrix}.$$

Write a MATLAB statement performing a step of

- (a) the Jacobi iteration

$$x^{(i+1)} := x^{(i)} - D^{-1}*(A*x^{(i)} - b)$$

(here  $D$  is the diagonal matrix formed from the main diagonal of  $A$ ),

- (b) the Gauss-Seidel iteration

$$x^{(i+1)} := x^{(i)} - L^{-1}*(A*x^{(i)} - b)$$

(here  $L$  is the lower diagonal part of  $A$  with zeros above the main diagonal),

starting from  $x^{(0)} = [0 \ 0 \ 0]^T$ . Use a `for` or `while` loop and observe its convergence behavior towards the exact solution of the given system after  $n$  steps.

17. The MATLAB function `inv(A)` computes the inverse of a given  $n \times n$ -matrix  $A$ . To compute the solution  $x$  of a linear system  $A*x = b$  we could compute `inv(A)` and multiply this inverse with the right hand side  $b$ . However, a better way, from an execution time (and also numerical accuracy!) standpoint, is to use matrix division, i.e.  $x = A \backslash b$  (what is the underlying algorithm?).

Write a MATLAB segment which compares these two versions of linear systems solving for different randomly generated data (using `rand`), with dimensions  $n$  from 1 to 100. Use stopwatch timing (`tic/toc`) to generate two vectors of length  $n$  containing the corresponding processing times. Plot this information.

18. Compute the coefficients of the *Legendre polynomials* of degree  $n = 1, 2, 3, 4$  defined by the formula

$$P_n(x) = \frac{1}{2^n n!} \frac{d^n}{dx^n} (x^2 - 1)^n.$$

Plot these polynomials.

*Hint:* Use `conv`, `polyder` and `polyval`.

19. The Legendre polynomials  $P_n(x)$ ,  $n = 0, 1, \dots$  are defined recursively as follows

$$\begin{aligned} P_0(x) &= 1, P_1(x) = x \\ nP_n(x) &= (2n-1)xP_{n-1} - (n-1)P_{n-2}(x), \quad n = 2, 3, \dots \end{aligned}$$

Write a MATLAB function `P = LegendP(n)` that takes an integer  $n$  – the degree of  $P_n(x)$  and returns its coefficients stored in descending order of powers.

20. Interpolation problem: Write a MATLAB segment which computes the coefficients of  $p(x)$  of degree 4 which satisfies  $p(0) = p'(0) = p(1) = p'(1) = 1$  and  $p(1/2) = -1$ . Plot the function  $p(x)$ .

*Hint:* Solve an appropriate system of linear equations. Check your result.

21. Write a MATLAB segment which orthonormalizes the sequence of  $n \times n$ -matrices  $A(k) := (a_{i,j}(k))$  where

$$a_{i,j}(k) := (n(i-1) + j)^k \quad k = 1, \dots, m$$

(with  $n = 5$ ,  $m = 5$ ) w.r.t. the inner product<sup>1</sup>

$$\langle A, B \rangle := \text{trace}(B^T A)$$

( $A, B \dots n \times n$  - matrices).

*Hint:* Use the Gram-Schmidt method.

22. Compare the plots, in the interval  $x \in [-1, 1]$ , of the *Chebyshev polynomials*  $T_n(x)$  of degree  $n = 0, 1, 2, 3, 4$  defined by the formula

$$T_n(x) = \cos(n \arccos(x))$$

Then, by using `polyfit`, find the coefficients of these polynomials.

23. Use `polyfit` to approximate the error function,  $\text{erf}(x)$ , by a single polynomial (`polyfit` works in a least squares sense). Start by generating an equally spaced vector of  $x$ -points (e.g.  $x = 0.1, 0.2, \dots, 2.5$ ) and fit the data  $y = \text{erf}(x)$  with a polynomial of degree 6. Plot  $\text{erf}(x)$  and its polynomial fit to check the approximation quality inside the interval  $[0, 2.5]$  and outside (say, up to  $x = 5$ ).
24. In this exercise, implement Euclid's Algorithm for computing the greatest common divisor (`gcd`) of two integer numbers  $a$  and  $b$ :

$$\text{gcd}(a, 0) = a, \text{gcd}(a, b) = \text{gcd}(b, \text{rem}(a, b)).$$

Here  $\text{rem}(a, b)$  stands for the remainder in dividing  $a$  by  $b$ . MATLAB has function `rem`. Write a MATLAB function `gcd = mygcd(a, b)` that implements Euclid's Algorithm.

25. Write a MATLAB function `aqx(A, x, t)` which computes the vector  $y = A^q * x$ , where  $x$  is a given vector of length  $n$ ,  $A$  is an  $n \times n$ -matrix, and  $q = 2^t$ . Let  $A = \text{rand}(n)$  and  $x = \text{rand}(n, 1)$  for some values of  $n$  (e.g. 5, 10, 20). Print `T(1:10)` with the property that  $T(t)$  is the length of time required to execute `aqx(A, x, t)`.

*Hint:* There is more to this problem than just matrix-vector multiplication. Several algorithmic versions are possible, e.g.

- Computation of  $A^q$  via repeated matrix multiplication (generating  $A^2, A^4, \dots, A^q$ ) and computation of  $y$  by one final matrix-vector multiplication

---

<sup>1</sup> $\langle A, B \rangle$  is an inner product if the matrices are interpreted as "long vectors" (row by row or column by column in  $\mathbb{R}^{n^2}$ ).

- $q$  matrix-vector multiplications  $y_1 := A*x$ ,  $y_2 := A*y_1$ , etc.
- Computation of  $A^2$  followed by  $q/2$  matrix-vector multiplications
- ...

How to compute  $A^q*x$  in the most efficient way depends on  $n$  and  $q$ . Try different versions, compare processing times and plot the results.

26. *Collocation* is a method for approximating the solution of differential equations: Consider an initial value problem  $y' = f(y)$ ,  $y(0) = y_0$ . Choose a stepsize  $h$  and a vector  $c$  of  $s$  distinct nodes  $c_i \in [0, 1]$  and find a polynomial  $p(t)$  of degree  $\leq s$  by requiring that  $p(t)$  satisfies

- the initial condition, i.e.,  $p(0) = y_0$ , and
- the differential equation at the nodes  $c_i h$  i.e.,  $p'(c_i h) = f(p(c_i h))$ ,  $i = 1, \dots, s$ .

Write a MATLAB function `colloc(lambda,y0,h,c)` which computes the coefficients of the corresponding collocating polynomial for the scalar ODE  $y' = \lambda y$ .

Make a particular choice for the problem data  $\lambda$  and  $y_0$ , let  $h = 0.1$  and check the accuracy of the resulting approximation  $p(h)$  for the exact solution value  $e^{\lambda h} y_0$  for equidistant nodes  $c_i$ , with different values of  $s$ .

27. Consider an ordinary differential equation (ODE) with initial condition: A function  $y(t)$  (with values in  $\mathbb{R}^n$ ) is to be determined such that  $y(0)$  equals a given initial vector  $y_0 \in \mathbb{R}^n$  and such that, for  $t > 0$ ,  $y(t)$  satisfies the differential equation  $y'(t) = f(y(t))$  ( $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$  a given function).

*Runge-Kutta* methods for numerically approximating the solution of such problems work in the following way. A Runge-Kutta scheme is defined by the parameter  $s \geq 1$  (number of stages), and particular choices for the  $s \times s$  coefficient matrix  $A = (a_{ij})$  and the vector  $b$  of  $s$  weights  $b_i$ . Choose a stepsize  $h$  and solve the following system of

equations (unknowns  $Y_1, \dots, Y_s$ ):

$$\begin{aligned} Y_1 &:= y_0 + h \sum_{j=1}^s a_{1j} f(Y_j) \\ Y_2 &:= y_0 + h \sum_{j=1}^s a_{2j} f(Y_j) \\ Y_s &:= y_0 + h \sum_{j=1}^s a_{sj} f(Y_j) \end{aligned}$$

The resulting Runge-Kutta approximation to the solution value  $y(h)$  is then defined as

$$\tilde{y}(h) = y_0 + h \sum_{j=1}^s b_j f(Y_j).$$

Write a MATLAB function `rk(M,y0,h,A,b)` which computes this approximation to  $y(h)$  for the linear ODE system  $y'(t) = M * y(t)$  ( $M$  a given  $n \times n$ -matrix,  $n \geq 1$  arbitrary). Test this approximation for concrete data  $y_0, M$ , with  $n > 1$ , using the following Runge-Kutta scheme:

$$A = \begin{bmatrix} \frac{1}{4} & \frac{1}{4} - \frac{1}{6}\sqrt{3} \\ \frac{1}{4} + \frac{1}{6}\sqrt{3} & \frac{1}{4} \end{bmatrix}, \quad b = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \end{bmatrix}$$

(the so-called 2-stage Gauss-Legendre method).

28. Write and test a MATLAB function `bisect` which solves a nonlinear equation  $f(x) = 0$  using the *bisection method*:

Given an interval  $[a, b]$  where a zero of a given (continuous) function  $f$  is to be found, do:

- Check if  $f(a)$  and  $f(b)$  have opposite sign, otherwise exit.
- Compute  $f((a+b)/2)$ , check its sign and continue in that subinterval where a zero of  $f$  must lie.
- Stop if the actual search interval has become smaller than a given tolerance `tol` or if a maximum number `maxit` of iterations is reached.

The specification of `bisect` should read as follows:

```

function [x,y] = bisect(fun,a,b,tol,max)
%BISECT Find a zero using bisection method
%
% fun           specifies the function f
% [a,b]         interval containing zero
% tol           allowable tolerance in computed zero
% maxit         maximum number of iterations allowed
%
% x             vector of approximations to zero
% y             vector of function values fun(x)

```

29. Write and test a MATLAB function `falsi` which solves a nonlinear equation  $f(x) = 0$  using the *regula falsi* method:

Given an interval  $[a, b]$  where a zero of a given (continuous) function  $f$  is to be found, do:

- Check if  $f(a)$  and  $f(b)$  have opposite sign, elsewhere exit.
- Compute an approximation  $z$  for the zero searched for by

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

- Determine the sign of  $f(z)$  in order to decide in which subinterval to continue.
- Stop if the actual search interval has become smaller than a given tolerance `tol` or if a maximum number `maxit` of iterations is reached.

The specification of `falsi` should read as follows:

```

function [x,y] = falsi(fun,a,b,tol,max)
%FALSI Find a zero using regula falsi method
%
% fun           specifies the function f
% [a,b]         interval containing zero
% tol           allowable tolerance in computed zero
% maxit         maximum number of iterations allowed
%
% x             vector of approximations to zero
% y             vector of function values fun(x)

```

30. Write and test a MATLAB function `secant` which solves a nonlinear equation  $f(x) = 0$  using the *secant method*:

Given an interval  $[a, b]$  where a zero of a given (continuous) function  $f$  is to be found, choose  $x_1 := a$ ,  $x_2 := b$  and iterate:

- $(x_{i-1}, x_i) \rightarrow x_{i+1}$  via

$$x_{i+1} := x_i - \frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})} f(x_i).$$

- Stop if  $|x_{i+1} - x_i|$  has become smaller than a given tolerance `tol` or if a maximum number `maxit` of iterations is reached.

The specification of `secant` should read as follows:

```
function [x,y] = secant(fun,a,b,tol,max)
%SECANT Find a zero using the secant method
%
% fun           specifies the function f
% [a,b]         interval containing zero
% tol           allowable tolerance in computed zero
% maxit         maximum number of iterations allowed
%
% x             vector of approximations to zero
% y             vector of function values fun(x)
```

31. Write and test a MATLAB function `newton` which solves a nonlinear equation  $f(x) = 0$  using *Newton's method*:

Given an initial approximation  $x_0$ , iterate:

- $x_i \rightarrow x_{i+1}$  via

$$x_{i+1} := x_i - \frac{f(x_i)}{f'(x_i)}.$$

- Stop if  $|x_{i+1} - x_i|$  has become smaller than a given tolerance `tol` or if a maximum number `maxit` of iterations is reached.

The specification of `newton` should read as follows:

```
function [x,y] = newton(fun,fun_pr,x1,tol,maxit)
%NEWTON Find zero near x0 using Newton's method
%
```

```

% fun           specifies the function f
% fun_pr       specifies the derivative of f
% x0           starting estimate
% tol          allowable tolerance in computed zero
% maxit        maximum number of iterations
%
% x            (row) vector of approximations to zero
% y            (row) vector of function values fun(x)

```

32. Write and test a MATLAB function `newton_sys` which solves a system  $F(x) = 0$  of  $n$  nonlinear equations in  $n$  variables using *Newton's method*:

Given an initial approximation  $x_0$ , iterate:

- $x_i \rightarrow x_{i+1}$  via

$$x_{i+1} := x_i - \delta_i,$$

where  $\delta_i$  is the solution of the linear system  $DF(x_i)\delta_i = F(x_i)$  ( $DF \dots$  Jacobian of  $F$ ).

- Stop if  $\|x_{i+1} - x_i\|$  has become smaller than a given tolerance `tol` or if a maximum number `maxit` of iterations is reached.

The specification of `newton_sys` should read as follows:

```

function x = newton_sys(F,DF,x0,tol,maxit)
%NEWTON_SYS Solve the nonlinear system F(x) = 0 using
%           Newton's method
%
% F         specifies the function F (column vector)
% DF        specifies the Jacobian of F
% x0        starting vector
% tol       allowable tolerance in computed zero
% maxit     maximum number of iterations
%
% x         vector of resulting approximation to zero

```

33. Write and test a MATLAB function `fixed_pt_sys` which solves a nonlinear system  $x = G(x)$  ( $G : \mathbb{R}^n \rightarrow \mathbb{R}^n$ ) using *fixed point iteration*:

Given an initial approximation  $x_0$ , iterate:

- $x_i \rightarrow x_{i+1}$  via

$$x_{i+1} := G(x_i), \quad i = 0, 1, \dots$$

- Stop if  $\|x_{i+1} - x_i\|$  has become smaller than a given tolerance `tol` or if a maximum number `maxit` of iterations is reached.

The specification of `newton_sys` should read as follows:

```
function x = fixed_pt_sys(G,x0,tol,maxit)
%FIXED_PT_SYS Solve the nonlinear system x=G(x) using
%           fixed point iteration
%
%   G       specifies the function G (column vector)
%   x0      starting vector
%   tol     allowable tolerance in computed zero
%   maxit   maximum number of iterations
%
%   x       vector of resulting approximation to zero
```

The convergence of this iteration requires that  $G$  is Lipschitz continuous with a Lipschitz constant  $< 1$  in a neighborhood of the fixed point.

*Example:* For  $n = 3$ ,  $G(x_1, x_2, x_3) = (-0.02x_1^2 - 0.02x_2^2 - 0.02x_3^2 + 4, -0.05x_1^2 - 0.05x_3^2 + 2.5, 0.025x_1^2 + 0.025x_2^2 - 1.875)^T$  the iteration converges in a sufficiently small neighborhood of the fixed point  $(3.6328\dots, 1.7321\dots, -1.4701\dots)^T$ . Make experiments with different initial approximations  $x_0$ .

34. The minor  $A_{ij}$  of an element  $a_{ij}$  of an  $n \times n$ -matrix  $A$  is the determinant of the matrix obtained by removing the  $i$ -th row and the  $j$ -th column of  $A$ .

Write and test a function `inverse(A)` which computes  $A^{-1}$  using minors via the formula

$$(A^{-1})_{ij} = (-1)^{i+j} A_{ji} / \det(A).$$

`inverse` should use a subfunction `minor(A,i,j)` to compute the corresponding minors.

Check your result using the standard MATLAB inverse `inv`. An error message should be issued if the given matrix is singular or not quadratic.

*Warning:* The above method is very inefficient for large matrices.

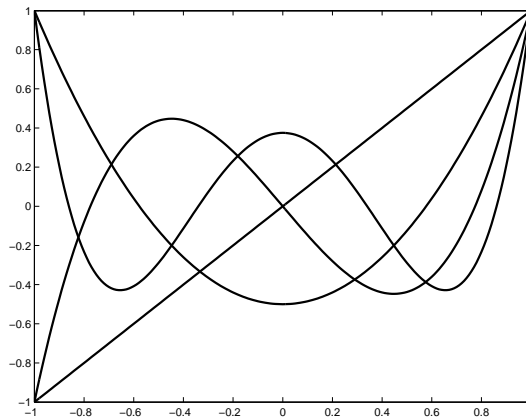


Figure C.1: Legendre polynomials up to degree 4

35. (cf. example 33) The *Legendre polynomials*  $P_n(x)$  can be defined by the recurrence relation

$$(n+1)P_{n+1}(x) - (2n+1)xP_n(x) + nP_{n-1}(x) = 0$$

with  $P_0(x) = 1$  and  $P_1(x) = x$ .

Write and test a MATLAB script which recursively computes the Legendre polynomials up to a given degree (represented as a coefficient vector as usual in MATLAB) and plot these polynomials up to degree 10 on the interval  $[-1, 1]$ . (cf. Figure C.1).

36. • Write and test a script that asks for an integer  $n$  and then performs the following iteration:

While the value of  $n$  is greater than 1, replace it with

$$n := \begin{cases} n/2, & \text{if } n \text{ is even,} \\ 3n+1, & \text{if } n \text{ is odd.} \end{cases}$$

The iteration stops if a value  $n = 1$  occurs. (It has been conjectured that this iteration terminates for arbitrary given  $n$ , but a proof seems not to exist.)

Save a protocol of this iteration in an array, i.e. the different values of  $n$  occurring during this iteration, and plot.

- Furthermore, write a script which protocols and plots the *number of iterations* performed for different starting values for  $n$  ( $n = 1 \dots n_{max}$ ) for given  $n_{max}$ .

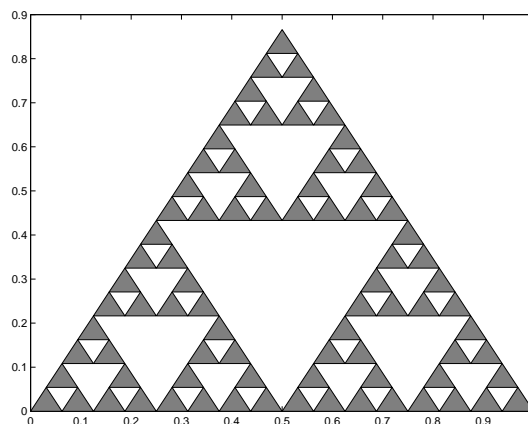


Figure C.2: Sierpinski gasket (level=4, 81 triangles)

37. Write and test a recursive function `gasket(a,b,c,level)` which draws a *Sierpinski gasket* (cf. figure C.2).

This is based on the following process: Given a triangle with vertices  $a$ ,  $b$  and  $c$ , we remove the subtriangle with vertices at the midpoints of the edges,  $(a+b)/2$ ,  $(b+c)/2$  and  $(c+a)/2$ . This removes the ‘middle quarter’ of the triangle (level 1). Effectively, we have replaced the original triangle with three subtriangles. We can now apply this removal process to each of these triangles to generate nine subtriangles (level 2), and so on, up to a given maximal level, generating a pattern consisting of  $3^{\text{level}}$  triangles. These remaining triangles are to be colored using the MATLAB function `fill`.

38. Write and test a recursive function `rgasket(a,b,c,d,level)` which draws a rectangular version of a *Sierpinski gasket* (cf. figure C.3).

This is based on the following process: Given a rectangle with vertices  $a$ ,  $b$ ,  $c$  and  $d$ , we subdivide it into 9 congruent subrectangles and remove the 4 subrectangles located at the middle of the boundaries (level 1). Effectively, we have replaced the original rectangle with five subrectangles. We can now apply this removal process to each of the remaining 5 subrectangles to generate 25 subrectangles (level 2), and so on, up to a given maximal level, generating a pattern consisting of  $5^{\text{level}}$  rectangles. These remaining rectangles are colored using the MATLAB function `fill`.

39. The *Chebyshev polynomials*  $T_n(x)$  can be defined by the recurrence

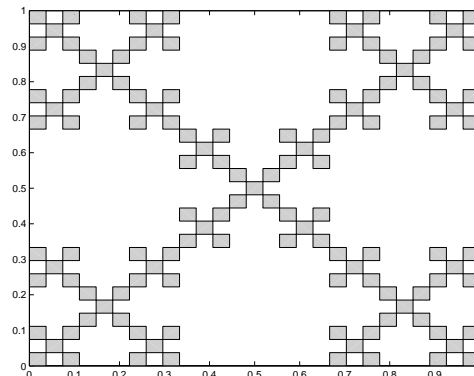


Figure C.3: Rectangular Sierpinski gasket (level=3, 125 rectangles)

relation

$$T_{n+1}(x) := 2xT_n(x) - T_{n-1}(x)$$

with  $T_0(x) = 1$  and  $T_1(x) = x$ .

Write and test a function `cheby(x,n)` which evaluates the Chebyshev polynomials  $T_n(x)$  up to  $n = p$  at a given vector  $\mathbf{x}$  of  $x$ -values in the interval  $[-1, 1]$ . Use vectorization!

Use this function to plot  $x$ -values in the Interval the  $T_n(x)$  for  $n = 1 \dots 10$ .

40. In this example you will solve linear systems of equations to find sums of the form  $\sum_{i=1}^n i^k$ . You have probably seen a few sums of this form, such as

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} = \frac{1}{2}n^2 + \frac{1}{2}n, \text{ or}$$

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

One way to find the sum  $\sum_{i=1}^n i^k$  for an integer  $k$  is to use the fact that such a sum can be expressed as a polynomial of degree  $k+1$  in  $n$  (proof by induction w.r.t.  $k$  – try it):

$$\sum_{i=1}^n i^k = a_{k+1}n^{k+1} + a_k n^k + \dots + a_1 n + a_0.$$

We can find the coefficients  $a_{k+1}, \dots, a_0$  by solving a linear system of  $k+2$  equations in these  $k+2$  unknowns; simply substitute the values

$n = 0, 1, \dots, k+1$  in the above formula. For example, to find the sum  $\sum_{i=1}^n i^2$ , use the fact that  $\sum_{i=1}^n i^2 = a_3 n^3 + a_2 n^2 + a_1 n + a_0$  and substitute the values  $n = 0, 1, 2, 3$  into this equation. We thereby obtain the system of equations

$$\begin{aligned} 0 a_3 + 0 a_2 + 0 a_1 + a_0 &= 0 \\ a_3 + a_2 + a_1 + a_0 &= 1 \\ 8 a_3 + 4 a_2 + 2 a_1 + a_0 &= 5 \\ 27 a_3 + 9 a_2 + 3 a_1 + a_0 &= 14 \end{aligned}$$

- (a) Use MATLAB to solve the above system and thus confirm that

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

(*Hint:* The coefficient matrix is a Vandermonde matrix and therefore can be constructed quickly using MATLAB's `vander` function. After solving the system, apply the `rats` function to your answer to convert it into a rational number.)

- (b) Use MATLAB to calculate the coefficients of the sum  $\sum_{i=1}^n i^k$  with  $k$  arbitrary.
41. *Singular values and numerical rank of a matrix.* The  $n \times n$  Hilbert matrix  $H$  is a matrix whose  $(i, j)$  entry  $h_{ij}$  is given by

$$h_{ij} = \frac{1}{i+j-1}.$$

It can be generated by the MATLAB command `H = hilb(n)`.

- (a) Generate the  $n \times n$  Hilbert matrix  $H$  for  $n = 10, 12, 14$  and  $16$ . In each case compute the inverse and rank of the matrix using MATLAB's `inv` and `rank` commands. Which of these Hilbert matrices appear to be singular?
- (b) The MATLAB command `rank` determines the *numerical rank* of a matrix rather than its exact rank. The numerical rank of a matrix is determined by its singular values. If  $k$  of the singular values of an  $n \times n$ -matrix are very close to 0, then for finite precision numerical computations the matrix will behave just as if those singular values were 0. For computational purposes, the matrix is

indistinguishable from a rank  $n-k$  matrix. Consequently, we say that its *numerical rank* is  $n-k$ .

In actuality, all of the Hilbert matrices are nonsingular; however, the larger  $n$  is, the closer the matrix will be to one of lower rank. Compute the singular values (using `svd`) of the  $12 \times 12$  Hilbert matrix  $H$  and display the results using `format long`. If all displayed digits of a singular value are 0, then the computed singular value is 0 to machine precision even though it would be nonzero if calculated in exact arithmetic. How many of the computed singular values of  $H$  were nonzero? Use the `rank` command to determine the numerical rank of  $H$ . Repeat this experiment with the  $16 \times 16$  Hilbert matrix. What relationship do you observe between the computed singular values and the numerical rank?

# Bibliography

- [1] W. Auzinger. *Einführung in die EDV für TM (Vorlesungsskriptum)*. TU Wien, 2000.
- [2] Kermit Sigmon, Timothy A. Davis. *MATLAB Primer*. Chapman & Hall/CRC, sixth edition, 2002.
- [3] D.J. Higham, N.J. Higham. *MATLAB Guide*. SIAM, Philadelphia, 2000.
- [4] M.S. Bazaraa, J.J. Jarvis, and H.D. Sherali. *Linear Programming and Network Flows*. John Wiley & Sons, New York, second edition, 1990.
- [5] Edward Neuman, editor. *MATLAB Tutorials (Southern Illinois University)*. <http://www.math.siu.edu/matlab/tutorials.html>.
- [6] S. Katzenbeisser, C. Ueberhuber. *MATLAB 6, Eine Einführung*. Springer, Wien, NewYork, 2000.