

# Introduction to Matlab for Engineers

Are Mjaavatten

Bernt Lie

August, 2005



# Contents

Preface	xiii
<b>I Analysis of technical data with Matlab</b>	<b>1</b>
1 The Matlab workplace	3
<b>2 Arrays: Matlab's basic data structure</b>	<b>5</b>
2.1 Overview of learning goals	5
2.2 Arrays and data structures	5
2.3 Assignment: Naming data structures	6
2.4 Naming data structures: some technicalities*	7
2.5 Built-in constants in Matlab	8
2.6 Basic creation of arrays	9
2.6.1 Dimension of arrays	9
2.6.2 Element-wise assignment	10
2.6.3 Assigning row arrays	11
2.6.4 Assigning column arrays	12
2.6.5 Assigning two-dimensional arrays	12
2.6.6 Line continuation	13
2.6.7 One-dimensional addressing of array elements*	13
2.7 Exporting and importing data from files	14
2.7.1 Saving and loading data	14
2.7.2 File import	17
2.8 Functions and operations	19
2.9 Numeric format and accuracy	20
2.10 Functions for creating arrays	21
2.10.1 Creating arrays	21
2.10.2 Size of arrays	26
2.10.3 Rearranging arrays	27
2.11 Subarrays and superarrays	28
2.11.1 Picking subarrays	28
2.11.2 Building superarrays	29
2.12 Housekeeping	30
2.12.1 Variables	31
2.12.2 Matlab commands	32
2.12.3 OS commands	32
2.13 Basic functions	33
2.13.1 Overview of operations	33
2.13.2 Trigonometric functions	35
2.13.3 Exponential functions	36
2.13.4 Complex functions	37
2.13.5 Rounding and remainder functions	38
2.13.6 Number theoretic functions*	40
2.14 Help	41

<b>3</b>	<b>Basic plotting in Matlab</b>	<b>43</b>
3.1	Overview of learning goals . . . . .	43
3.2	Basic two dimensional plots . . . . .	43
3.2.1	The <code>plot</code> function . . . . .	43
3.2.2	Editing plots . . . . .	44
3.2.3	Multiple plots . . . . .	44
3.2.4	Editing axes properties . . . . .	47
3.2.5	Command line plot editing . . . . .	47
3.2.6	Fancy typesetting in plots* . . . . .	48
3.2.7	Closing . . . . .	48
3.3	Improving plots . . . . .	50
3.4	Array plots . . . . .	51
3.5	Presentation of experimental data . . . . .	52
3.5.1	Case study: distillation of water and ethanol . . . . .	52
3.5.2	Plotting data . . . . .	54
3.6	Further study . . . . .	55
<b>4</b>	<b>Simple data analysis*</b>	<b>57</b>
4.1	Overview of learning goals . . . . .	57
4.2	Interpolation . . . . .	57
4.2.1	Overview . . . . .	57
4.2.2	Polynomial interpolation . . . . .	58
4.2.3	Other interpolation functions . . . . .	60
4.2.4	Comparison of interpolation models . . . . .	61
4.3	Analysis . . . . .	63
4.3.1	Polynomials in Matlab . . . . .	63
4.3.2	Operations on polynomials . . . . .	64
4.3.3	Analysis of polynomial models . . . . .	66
4.4	More on interpolation and prediction models . . . . .	67
4.4.1	Case study: Saturated steam . . . . .	67
4.4.2	Multivariable interpolation . . . . .	71
<b>5</b>	<b>Introduction to automation of tasks</b>	<b>77</b>
5.1	Overview of learning goals . . . . .	77
5.2	Strings and basic string operations . . . . .	77
5.3	Motivating examples . . . . .	78
5.4	The basic repetition statement . . . . .	79
5.5	Scripts and the Matlab editor . . . . .	80
5.6	Examples: Repetition and Sums . . . . .	82
5.7	Example: Basic repetition over array* . . . . .	84
<b>6</b>	<b>Problems</b>	<b>87</b>
<b>II</b>	<b>Exploiting the power of Matlab</b>	<b>91</b>
<b>7</b>	<b>Revisiting Part I</b>	<b>93</b>
<b>8</b>	<b>More program flow control</b>	<b>95</b>
8.1	Introduction . . . . .	95
8.2	Logical variables in Matlab . . . . .	95
8.3	Relations: functions and operators . . . . .	95
8.4	Logical expressions . . . . .	96
8.5	Making logical choices: the <code>if</code> statement . . . . .	98
8.5.1	Motivating example . . . . .	98
8.5.2	Structure of <code>if</code> statements . . . . .	98
8.5.3	Example: use of <code>if</code> statement . . . . .	98
8.6	Logical choices: the <code>switch</code> statement . . . . .	99
8.7	Repetition loop: the <code>while</code> statement . . . . .	100

8.7.1	Motivating example . . . . .	100
8.7.2	Structure of <code>while</code> statement . . . . .	100
8.7.3	Example: use of <code>while</code> statement . . . . .	101
8.7.4	Breaking out from <code>for</code> loop . . . . .	101
<b>9</b>	<b>Writing your own functions</b>	<b>103</b>
9.1	Example: Distance to horizon . . . . .	103
9.1.1	Basic case . . . . .	103
9.1.2	Returning more than one result . . . . .	105
9.1.3	Exercises . . . . .	106
9.2	Workspaces and variables . . . . .	107
9.3	Loops vs. matrix operations . . . . .	108
<b>10</b>	<b>Advanced use of the Matlab editor</b>	<b>109</b>
10.1	Errors and the debugger . . . . .	109
10.2	De-linting the code . . . . .	111
10.3	Structuring the file . . . . .	111
<b>11</b>	<b>Data structures in Matlab</b>	<b>115</b>
11.1	Structures . . . . .	115
11.2	Cell arrays . . . . .	116
<b>12</b>	<b>Handle graphics</b>	<b>117</b>
<b>13</b>	<b>Problems</b>	<b>121</b>
<b>III</b>	<b>Becoming a Matlab Master</b>	<b>123</b>
<b>14</b>	<b>The once and future Matlab Master</b>	<b>125</b>
14.1	Revisiting Part II . . . . .	125
14.2	Overview of new material . . . . .	125
<b>15</b>	<b>Functions and function handles</b>	<b>127</b>
15.1	Function handles . . . . .	127
15.2	Function functions . . . . .	128
15.3	Anonymous functions . . . . .	129
15.4	Function functions: motivating example . . . . .	132
15.4.1	Implementation of Newton method . . . . .	132
15.4.2	Generalizing the Newton method . . . . .	134
15.4.3	Some final thoughts . . . . .	135
15.5	Function handles as function arguments . . . . .	135
15.6	Exercises . . . . .	137
<b>16</b>	<b>Simulation of dynamic systems</b>	<b>141</b>
16.1	Solution of ODEs using Euler integration . . . . .	141
16.2	Generalizing our Euler solver . . . . .	143
16.3	ODE solvers in Matlab . . . . .	146
16.4	Exercises . . . . .	147
<b>IV</b>	<b>Closing</b>	<b>149</b>
<b>17</b>	<b>Conclusions</b>	<b>151</b>

<b>V</b>	<b>Appendices</b>	<b>153</b>
<b>A</b>	<b>Arrays vs. vectors and matrices*</b>	<b>155</b>
A.1	Vectors . . . . .	155
A.2	Matrices . . . . .	156
<b>B</b>	<b>Manipulation of Arrays and Array Functions*</b>	<b>159</b>
B.1	Indexing arrays . . . . .	159
B.2	Building superarrays . . . . .	161
B.2.1	Toeplitz and Hankel arrays . . . . .	161
B.2.2	Kronecker product . . . . .	162
B.2.3	Block diagonal arrays . . . . .	163
<b>C</b>	<b>Hexadecimal numbers*</b>	<b>165</b>
<b>D</b>	<b>Three Dimensional Plots and Plot Housekeeping*</b>	<b>167</b>
D.1	Three dimensional plots . . . . .	167
D.2	Housekeeping . . . . .	169
<b>E</b>	<b>Automation and String Operations*</b>	<b>173</b>
E.1	Motivating example: generating sequence of plots . . . . .	173
E.2	Strings and string operations . . . . .	174
E.3	Example: automatic generation of figures . . . . .	176
E.4	Example: hex 2 floating point . . . . .	178
<b>F</b>	<b>Numerical methods with Matlab*</b>	<b>181</b>
F.1	Numerical functions . . . . .	181
F.2	Arrays vs. Vectors and Matrices . . . . .	182
F.3	Matrix operations . . . . .	183
F.3.1	Basic operations . . . . .	183
F.3.2	Matrix factorizations/decompositions <sup>1</sup> . . . . .	183
F.3.3	Vector space commands <sup>2</sup> . . . . .	184
<b>G</b>	<b>Extra parameters and anonymous functions*</b>	<b>185</b>
<b>H</b>	<b>DAE solvers in Matlab*</b>	<b>187</b>
<b>VI</b>	<b>References</b>	<b>191</b>
	<b>Bibliography</b>	<b>193</b>
	<b>Index</b>	<b>195</b>

---

<sup>1</sup>These methods are used extensively in linear algebra.

<sup>2</sup>These methods are used extensively in advanced linear algebra.

# List of Figures

1.1	The opening window in Matlab 6.5. It looks almost the same in Matlab 7.0. . . . .	4
1.2	The Matlab window after entering a command. . . . .	4
2.1	The result after loading a <code>.mat</code> file. . . . .	15
2.2	The content of the ASCII file. Note that the first two rows come from the $A$ array, while the three next rows come from the $B$ array. . . . .	16
2.3	Situation after loading file <code>myworkspaceA</code> (a <code>.mat</code> file), which only contains array $A$ . . .	16
2.4	Situation after loading file <code>myworkspaceB</code> (an <code>.ascii</code> file), which only contains array $B$ . . .	17
2.5	Excel spreadsheet with data that we want to import into Matlab. . . . .	18
2.6	The result of the Matlab Import Wizard. . . . .	18
2.7	The function $y = \sin x$ , where $x$ is the input argument, and $y$ is the output argument. This is an example of a relationship between $x$ and $y$ , which is also a function. . . . .	19
2.8	Implicit plot of $x^2 + y^2 = 1$ . This is an example of a relationship between $x$ and $y$ which is not a function. . . . .	19
2.9	The help browser window which opens when using the <code>doc</code> command in Matlab's command window, or by choosing <b>Help/MATLAB Help</b> in the Matlab window. . . . .	42
3.1	Function $y = \sin x$ plotted using command <code>plot(y)</code> . . . . .	44
3.2	Figure Toolbar (upper row) and Plot Edit Toolbar (lower row). Select which toolbar to see from the <b>View</b> menu of the figure window. . . . .	44
3.3	Function $y = \sin x$ , plotted using command <code>plot(x,y)</code> . . . . .	45
3.4	Function $y = \sin x$ and function $\cos x$ plotted using command <code>plot(x,y,x,cos(x))</code> . . . .	46
3.5	Example of edited line styles and line colors. To insert legend, click on the Insert Legend icon (fig. 3.2). To change the default legend text, double-click on the text and edit it. . .	46
3.6	The graph of $\exp(-x)$ has been added to the graphs of $\sin x$ and $\cos x$ . . . . .	47
3.7	Function $y = \tan x$ , plotted using command <code>plot(x,tan(x))</code> . . . . .	50
3.8	The function $y = \tan x$ , plotted using command <code>plot(x,y)</code> , but where the $y$ values greater than 3 in absolute value have been replaced by NaN (Not a Number). . . . .	51
3.9	The use of the <code>subplot</code> command to produce an array of plots. . . . .	52
3.10	The use of the <code>subplot</code> command to produce an array of plots, where there is a mixture of array sizes. . . . .	53
3.11	Sketch of a simple distillation unit, with feed stream ( $F$ ), bottom liquid stream ( $L$ ) with mole fraction $x$ of light component, and top vapor stream ( $V$ ) with mole fraction $y$ of light component. . . . .	53
3.12	Equilibrium data for water and ethanol, as found from experiments: $(x, y)$ are mole fractions of liquid and vapor at equilibrium. . . . .	54
3.13	The equilibrium correlation for content of ethanol in the liquid phase ( $x$ ) and in the vapor phase ( $y$ ). . . . .	55
3.14	Experimentally found equilibrium correlation for content of ethanol in the liquid phase ( $x$ ) and in the vapor phase ( $y$ ). . . . .	56
4.1	Experimental data $((y, x), \times)$ and model $((y_m(x), x), \text{solid})$ for water-ethanol equilibrium. Notice flawed model representation. . . . .	59
4.2	Experimental data $((y, x), \times)$ and model $((y_m(x), x), \text{solid})$ for water-ethanol equilibrium. . . . .	59
4.3	Experimental data $((y, x), \times)$ , interpolation model $((y_m(x), x), \text{solid})$ , and low order model $(y_{m6}(x), x, \text{dotted})$ for water-ethanol equilibrium. . . . .	60

4.4	Experimental data $((y, x), \times)$ and spline model $((y_m(x), x), \text{solid})$ for water-ethanol equilibrium. . . . .	61
4.5	Experimental data $((y, x), \times)$ and <b>pchip</b> model $((y_m(x), x), \text{solid})$ for water-ethanol equilibrium. . . . .	62
4.6	Comparison of various interpolation models. . . . .	63
4.7	6th order least squares polynomial model $y_m(x)$ of water-ethanol equilibrium, with the derivative $dy_m(x)/dx$ and the integral $\int y_m(x) dx$ of $y_m(x)$ . . . . .	68
4.8	Relationships between thermodynamic variables of vapor phase saturated steam. . . . .	68
4.9	Pressure ( $p$ , atm) as a function of temperature ( $T$ , °C) for saturated steam. . . . .	69
4.10	Experimental pressure data ( $p$ , atm, $\times$ ) as a function of temperature ( $T$ , °C) for saturated steam, and 4th order least squares model fit (solid line). . . . .	69
4.11	Experimental temperature ( $T$ , °C, $\times$ ) as a function of pressure ( $p$ , atm) for saturated steam, and polynomial least squares models of degrees 3–5. . . . .	71
4.12	Relationships between thermodynamic variables of superheated steam at $p = 0.5$ atm. . . . .	72
4.13	Interpolation model for $\rho(T, p)$ . . . . .	74
4.14	Contour plot of interpolation model for $\rho(T, p)$ . . . . .	74
5.1	Matlab Editor window, containing a sequence of Matlab commands. Note: if the file has been saved, the “Save and run” icon changes to the “Run” icon. . . . .	81
5.2	Matlab editor showing script file with comments and spaces. . . . .	82
5.3	Script <b>SumInvSquare</b> for computing $S_n$ . . . . .	83
5.4	Script <b>Fibonacci</b> for computing $f_k$ . . . . .	83
5.5	Matlab Editor with script where the repetition <b>array</b> has more than one row. . . . .	84
5.6	The result of running script <b>RepetitionDemo3</b> . . . . .	85
6.1	Experimental batch reactor data for reaction with stoichiometry $A \rightarrow R$ . Taken from (Levenspiel 1972), p. 117. . . . .	87
6.2	Extent of reaction $\xi(t)$ for reactor data based on trapezoidal integration. . . . .	88
8.1	Script for finding root of $ax^2+bx+c = 0$ , where $a, b, c$ are specified in the Matlab Command Window. . . . .	99
8.2	Script for finding $S_n = \sum_{k=1}^n = S_{n-1} + \frac{1}{n^2}$ to a prespecified accuracy of $ S_n - S_{n-1}  < 10^{-6}$ . . . . .	101
8.3	Script with <b>for</b> loop, where an <b>if</b> statement is used to <b>break</b> out of the loop. . . . .	102
9.1	Distance $x_h$ to visible horizon from height $h$ . $D$ and $R$ are the diameter and radius, respectively, of the earth. . . . .	103
10.1	Some debugger tools. . . . .	110
10.2	Example of Matlab code with “lint”: statement <b>Sn_new = 1;</b> is superfluous and should have been removed. . . . .	111
10.3	Report from checking the code of fig. 10.2 with M-Lint. . . . .	112
10.4	Example of text cells in the Matlab editor. . . . .	112
10.5	Tools for taking advantage of cells in Matlab files. The <b>Show cell titles</b> tool can be used for browsing through the cells: clicking this tool, gives a list of the cells. . . . .	112
12.1	Comparison of $\cos x$ , and two Taylor expansions of $\cos x$ . . . . .	117
15.1	Definition of function $g : x \rightarrow \sin^2 x$ in Computer Algebra System (CAS) MuPAD. Note in particular that the mapping $g$ is independent of what we choose to name the <i>argument</i> . . . . .	130
15.2	Plot of $\text{sinc } x = \frac{\sin x}{x}$ , $x \in [-6\pi, 6\pi]$ . . . . .	131
15.3	Plot of function $f(x) = \sin x - 0.5$ (black), and linear approximation at $x = 1 : f(x) \approx \sin 1 - 0.5 + (x - 1) \cos 1$ (gray). We want to find $x : f(x) = 0$ . . . . .	134
15.4	Function $f(x) = \cos x - 0.5$ . . . . .	137
16.1	Sketch of damped mass-spring system. . . . .	141
D.1	Graph of function $\sin x \cos y \exp\left(-\frac{1}{10}(x^2 + y^2)\right)$ generated using a CAS system (Computer Algebra System, here: MuPAD from within the word processor Scientific WorkPlace). . . . .	168
D.2	Graph of function $f(x, y) = \sin x \cos x \exp\left(-\frac{1}{10}(x^2 + y^2)\right)$ generated in Matlab. . . . .	168



D.3	In order to rotate 3D plots: click the rotation icon on the toolbar, click-and-hold in the plot, and “rotate” using the mouse. . . . .	169
D.4	Graph of function $f(x, y) = \sin x \cos x \exp\left(-\frac{1}{10}(x^2 + y^2)\right)$ generated in Matlab, with added contour lines. . . . .	170
D.5	Contour lines of function $f(x, y) = \sin x \cos x \exp\left(-\frac{1}{10}(x^2 + y^2)\right)$ . . . . .	170
D.6	Graph of function $f(x, y) = \sin x \cos x \exp\left(-\frac{1}{10}(x^2 + y^2)\right)$ , mapped on the $x$ - $z$ plane. . . . .	171
E.1	Script file <code>RepetitionDemoModelFit.m</code> . . . . .	177
E.2	The model fit of 15-th order polynomial model. Notice that although the model goes through most experimental data points, the model is very poor for interpolation at “high” pressures. . . . .	178



# List of Tables

2.1	Some built-in Matlab constants. . . . .	9
2.2	Example array which we want to name $A$ . . . . .	10
2.3	Available commands to format data structures in Matlab. . . . .	21
2.4	Some commands for creating and manipulating arrays in Matlab. . . . .	22
2.5	Some commands for basic housekeeping in Matlab. . . . .	31
2.6	Basic binary array operations in Matlab. It is assumed that array $A$ and $B$ have the same shape. . . . .	34
2.7	Selected trigonometric functions in Matlab. . . . .	36
2.8	Selected exponential functions in Matlab. . . . .	37
2.9	Selected complex functions in Matlab. . . . .	37
2.10	Selected rounding and remainder functions in Matlab. . . . .	39
2.11	Selected number theoretic functions in Matlab. . . . .	40
3.1	Basic plotting commands. . . . .	43
3.2	Modifying properties of graphs. . . . .	45
3.3	Modifying properties of axes. . . . .	48
3.4	Illustration of using $\TeX$ and $\LaTeX$ typesetting in plots. . . . .	49
3.5	Basic plotting commands. . . . .	55
4.1	Table Caption . . . . .	57
4.2	Data set for comparison of interpolation models. . . . .	61
4.3	Functions on polynomials. . . . .	64
8.1	Matlab relational functions and operators. In the examples, it is assumed that $x = 3$ , $y = 4$ . . . . .	95
8.2	Matlab logical functions and operators. In the examples, it is assumed that $a = [1, 0, 0]$ , $b = [1, 1, 0]$ . . . . .	97
15.1	Some built-in Matlab functions that require a function reference (function handle) as input argument. . . . .	129
16.1	Numerical values for damped mass-spring system. . . . .	142
16.2	Table Caption . . . . .	146
16.3	Parameters and nominal operating conditions for CSTR. . . . .	147
E.1	Some basic functions for operating on strings. . . . .	174
E.2	More advanced functions for operating on strings. . . . .	176
E.3	Some functions for number conversion. . . . .	178
F.1	Some numerical functions in Matlab. . . . .	181
F.2	Basic binary operations among scalars $s$ and matrices $M$ . . . . .	182
F.3	Basic Matrix operations. . . . .	183
F.4	Basic matrix factorizations in Matlab. . . . .	184



# Preface

These lecture notes are developed for the introductory course in Matlab for Master of Science students at Telemark University College. The lecture notes consists of three parts, and the presentation of each part is scheduled to take 8 hours and to be presented within a single week. Thus, a total of three weeks spread over the opening of the fall semester are used to present the content of the course, with a total of ca. 24 hours of arranged presentations (lectures + exercises). The presentations are integrated with exercises in the courses Numerical Methods, and Modeling of Dynamic Systems, but Matlab is also used in other courses to reinforce the learning.

The choice of course content is the result of a couple of meetings between Are Mjaavatten, Bernt Lie, and Randi Holta in the early summer of 2003. Our background experiences for composing the course are:

- Continuous use of Matlab for some 10 years,
- The arrangement of a NIF<sup>3</sup> course in the use of Matlab (BL), where the course content was highly inspired by talks with student representatives Marte S. Lerdal and Bjørn Erik Thorsteinsen in June 1999,
- Experience with using Matlab in courses in Numerical Methods (AM) and Modeling of Dynamic Systems (BL).

Compared to the lecture notes for 2004, we have moved some of the material which is not of immediate use to appendices. We have also marked material which is more advanced with an asterisk (\*). We encourage readers to give feedback on the course content and the presentation style.

Finally, it should be mentioned that there exist a useful freeware version of Matlab, named Octave. A brief introduction to the use of Octave and how it deviates from using MathWorks' Matlab version, see [www.techteach.no/octave/index.htm](http://www.techteach.no/octave/index.htm). For more information on Octave, see [www.octave.org](http://www.octave.org).

---

Are Mjaavatten, Bernt Lie  
Porsgrunn, August 27 2005

---

<sup>3</sup>NIF = Norske Sivilingeniørers Forening, which has been transformed into Tekna.



## Part I

# Analysis of technical data with Matlab





# Chapter 1

## The Matlab workplace

The Matlab graphical user interface is programmed in Java, and thus looks practically the same on all supported computer platforms. We assume that the reader knows how to start up a program on her/his respective computer. When Matlab is started, it looks more or less as in fig. 1.1 (shows Matlab v. 6.5).

The main window in fig. 1.1 is the *Command Window*. This is where commands are written, and we will show excerpts of such command list in these notes. As an introductory example, assume that we type the command:

```
>> sin(0.7)
```

Here, the symbol `>>` is the Matlab command prompt, and the user does not type this symbol. We simply typed `sin(0.7)`. Matlab responds with:

```
ans =  
  
    0.6442  
  
>>
```

Here, `0.6442` is Matlab's evaluation of `sin(0.7)`, and the symbol `>>` signifies that Matlab's Command Window is ready to accept another command. In the Matlab window, we now have the following situation, fig. 1.2.

Note that by double-clicking on the item `sin(0.7)` in the Command History window, this command is copied into the Command Window, and is immediately (re-) executed. Also, by double-clicking on the item `ans` in the Workspace, the so-called Array Editor is opened, containing the data of variable `ans`. The Array Editor is a spreadsheet-like window which makes it possible to interactively modify the content of the chosen variable.

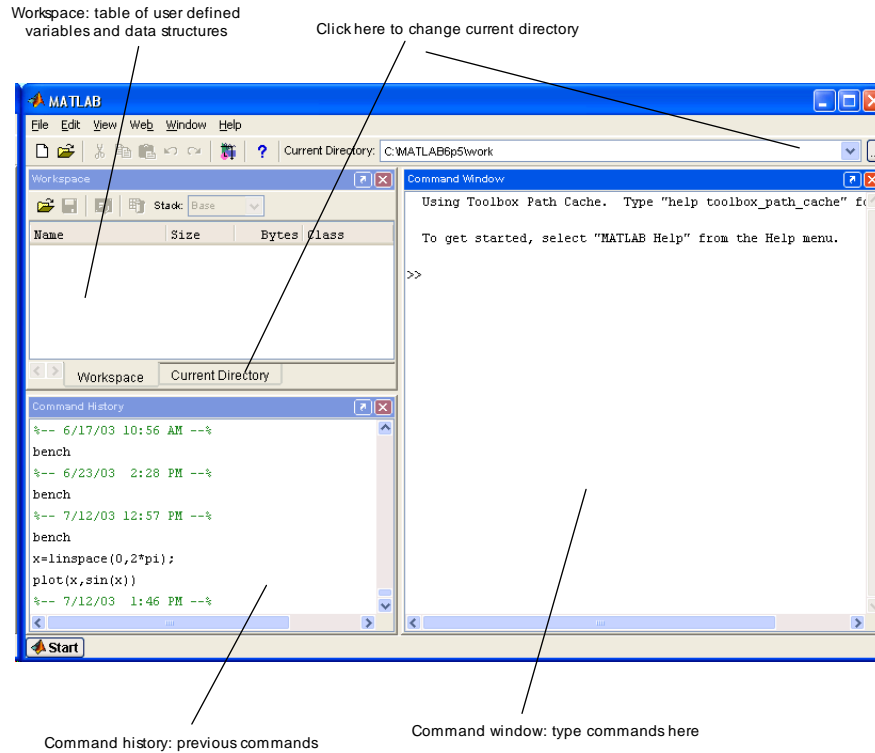


Figure 1.1: The opening window in Matlab 6.5. It looks almost the same in Matlab 7.0.

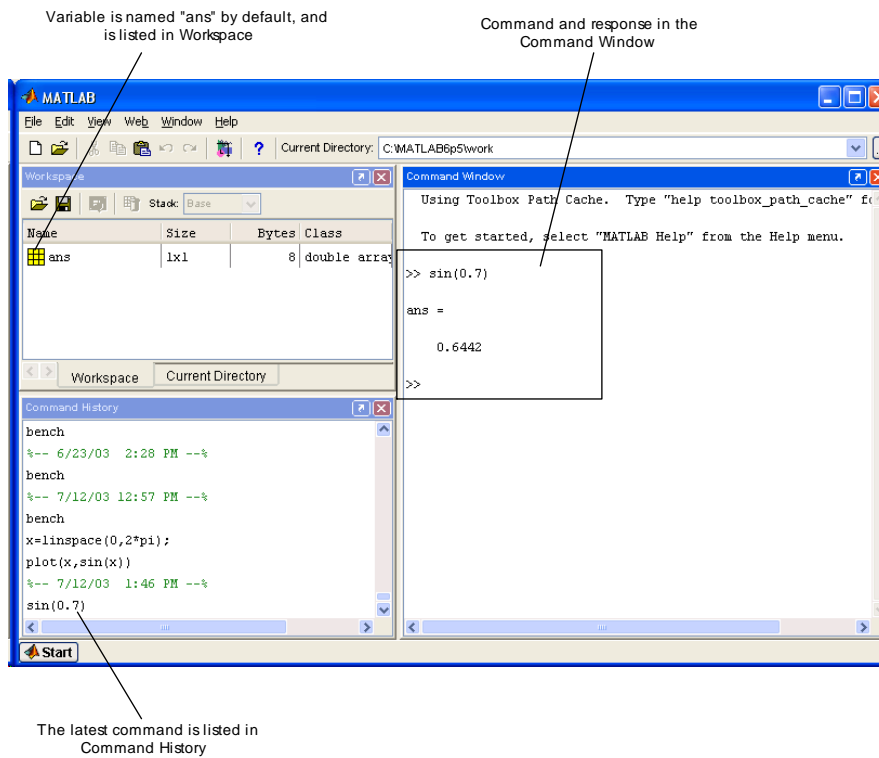


Figure 1.2: The Matlab window after entering a command.

## Chapter 2

# Arrays: Matlab's basic data structure

### 2.1 Overview of learning goals

After having completed this chapter, you should have a clear understanding of what an array is, what distinguishes an array from vectors and matrices, and how you can operate on arrays. In particular, you should master:

- How to define variables by assigning a value to a name, and how to save, load, and import data,
- The creation of arrays, how to pick out subarrays and how to build superarrays,
- How to operate on variables using functions and operators,
- How to adjust the presentation format of numbers,
- Basic housekeeping of variables, commands, and files,
- How to use Matlab's help system.

### 2.2 Arrays and data structures

By array, we will mean a *table of numbers* similar to a spreadsheet, where we require that every element in the table contains a number. Thus, there can be no vacant elements, and there can be no text strings or other objects in any of the elements. In principle, an array can hold a single element. In that case, we will use the term *scalar* to denote the array. If the array consists of a single row of numbers, we will denote it a *row array*. If the array consists of a single column of numbers, we will denote it a *column array*. A scalar is a special case of both a row array and of a column array. If the array contains several rows (or columns) of numbers where each row (column) does not have to be a scalar, then we will simply call it a (two dimensional) *array*. Obviously, both row and column arrays are arrays.

Even more complicated arrays could be imagined. We can consider arrays consisting of several layers of two dimensional arrays, much like a spreadsheet can contain several sheets of spreadsheets. In Matlab parlance, we say that the array consists of several *pages* of (two dimensional) arrays. Together these pages could be considered a volume. We can also consider arrays with several *volumes*. Together, several volumes could constitute a library. Then we can imagine arrays consisting of several libraries. And so on. Finally, we can also imagine arrays which contains zero elements; these we will denote *empty arrays*.

Originally, Matlab only supported two dimensional arrays (and character strings). From version 5 of Matlab, more general arrays are supported, as well as other *data structures* — these other data structures can be considered various types of tables. Often, row and column arrays are denoted row and column *vectors* in Matlab literature. Likewise, two dimensional arrays are often denoted *matrices* (Matlab was originally short for MATrix LABoratory). In these notes, we prefer to use the word array. The reason is that vectors are mathematical objects with very specific properties; vectors are not necessarily arrays. Likewise, matrices are mathematical operators with specific properties (matrices are arrays, however). Thus, we will use the term *array* when we do not imply such specific properties.

**Exercise 2.1** Write down examples of various types of arrays. ■

**Exercise 2.2** Write down examples of tables which are not arrays. ■

## 2.3 Assignment: Naming data structures

During computations, it is very important to be able to give names to data structures, e.g. to arrays. In practice, some data structures have constant value during these computations, while others vary/are changed. In this section, we will look at how we can give names (or: *assign* names) to data structures in Matlab. Some computer languages distinguish between how constants and variables are named — in Matlab, there is no such distinction

We have already seen that Matlab automatically gives a name to a result from a computation, see e.g. fig. 1.2 where the result of a computation was given the name `ans`. In practice, we need to be able to decide our own names for the data structures.

Various computer languages differ in how data structures are assigned a name. Symbolically, we could describe the assignment as `<data structure> -> <name>`, meaning “put the content of the data structure `<data structure>` into a computer memory location which bears the name `<name>`”. In most computer languages, this assignment is written the reverse way as `<name> <- <data structure>`, and other symbols may be used for the assignment. In e.g. the computer algebra system Maple, the assignment is written `<name> := <data structure>`.

In *Matlab*, the assignment symbol is identical to the symbol which is normally used for writing equality in mathematics: `<name> = <data structure>`:

```
>> a = 0.7
a =
    0.7000
>> b = sin(a)
b =
    0.6442
```

Here<sup>1</sup>, `a` and `b` are the names we want to use for the data structures, while `0.7` and `sin(a)` (where `a` is a replacement for `0.7`) are the data structures (they are arrays, or more precisely: scalars). In the assignment it is important both that the name we want to use is a valid name (see below), and that the data structure has a value (e.g. `0.7`, and `a` as a replacement for `0.7`). If we switch the sequence of the above commands, an error occurs:

```
>> clear all
>> b = sin(a)
??? Undefined function or variable 'a'.

>> a = 0.7
a =
    0.7000
```

At first, name `a` does not have a value, hence `sin(a)` does not have a value when we try to assign `sin(a)` to `b`. We'll come back to the effect of command `clear all` later.

The chosen assignment symbol in Matlab (and many other languages) *may* be confusing at first. Consider a common type of assignment:

```
>> i = i+1;
```

---

<sup>1</sup>To get a compact response in the Command Window, the command `format compact` has first been issued.

This assignment looks strange: how can `i` be equal to `i+1`? Well, it can not! The point is that `i=i+1` does not mean that `i` is equal to `i+1`. Instead, it means: add 1 to the value of data structure `i` (assuming that `i` already has a value), and put the result into data structure `i`. This is somewhat clearer if we write the assignment as `i+1 -> i`, or as in Maple: `i := i+1`. Still, as long as we accept that `=` means *assignment* and not equality in Matlab, the statement `i = i+1` makes sense:

```
>> i = 1;
>> i = i+1
i =
    2
```

Valid names for data structures in Matlab can consist of *any number of characters* as long as they *start with a letter*, and the *subsequent characters* are either letters in the English alphabet (`a`, ..., `z`, `A`, ..., `Z`), digits (`0`, ..., `9`), or underscore (`_`). Note that Matlab distinguishes between lowercase and uppercase letters. See Section 2.4 below for some more technical details about variable names.

**Exercise 2.3 (Assigning and changing the value of a variable)** Open Matlab and make sure you see the Command Window, the Workspace Window and the Command History Window. In the command window, write `x = 3`. Then press the “Enter” key and watch what happens in the workspace window. Next write:

```
x = x + 1
x = x + 1
```

Make sure you understand what is going on! Now write:

```
x = 3*x-4
```

Note that Matlab does not interpret this as an equation; interpreted as an equation, the answer would be `x = 2`. Instead, Matlab multiplies 3 with the current value of `x` (which is 5) and subtracts 4. The result (11) is then assigned to `x`. The interpretation of the `=` sign is thus: take the result of the instructions to the right and put the result into the variable at the left.

## 2.4 Naming data structures: some technicalities\*

Although names of data structures can consist of any number of characters, Matlab only uses the  $N$  first letters to distinguish between names. The number  $N$  may vary from computer platform to computer platform, and the number is found by issuing the command `namelengthmax` in Matlab’s Command Window. Here is the response from the author’s computer when issuing this command:

```
>> namelengthmax
ans =
    63
```

Thus, if you name one data structure by starting with 63 letters `a`, followed by digit 1, and another data structure by starting with 63 letters `a` followed by digit 2, Matlab will accept these names as valid names on the data structures, but Matlab will not be able to distinguish between the names. In practice, much shorter names are used.

There is a built in command `isvarname` in Matlab that can be used to check whether a name is valid. The following excerpt from the Matlab Command Window illustrates how to use the command in order to check the potential names `sin`, `namelengthmax`, and `5ball`:

```
>> isvarname('sin')
ans =
    1
>> isvarname('namelengthmax')
ans =
    1
>> isvarname('5ball')
```

```
ans =
     0
```

The response from Matlab is 1 (“true”) for legal names, and 0 (“false”) for illegal names.

Although both `sin` and `namelengthmax` are legal names of data structures, it is probably not a good idea to use these names: using the names may confuse Matlab as to how the names should be interpreted. Matlab also has a number of other built-in command names which should be avoided. In order to check that a potential name is not used by Matlab for other purposes (e.g. a built-in command, etc.), issue the command `which -all <name>`, where `<name>` is replaced by the potential name. Examples:

```
>> which -all sin
sin is a built-in function.
C:\MATLAB6p5\toolbox\matlab\elfun\sin.m % Shadowed
>> which -all namelengthmax
namelengthmax is a built-in function.
C:\MATLAB6p5\toolbox\matlab\lang\namelengthmax.m % Shadowed
>> which -all 5ball
5ball not found.
```

Note that `5ball` is not found; still `5ball` is an illegal name, and can not be used. It is very important to plan the names for data structures well, and to avoid names that are already used by Matlab. Consider e.g. that we have a data structure that contains the feed temperature of some process stream. We would like to denote the variable by `Tf`. Is this a good choice? Let us check:

```
>> which -all Tf
C:\MATLAB6p5\toolbox\control\control\@tf\tf.m % tf method
C:\MATLAB6p5\toolbox\control\control\@zpk\tf.m % zpk method
C:\MATLAB6p5\toolbox\control\control\@ss\tf.m % ss method
C:\MATLAB6p5\toolbox\control\control\@frd\tf.m % frd method
C:\MATLAB6p5\toolbox\ident\ident\@idmodel\tf.m % idmodel method
C:\MATLAB6p5\toolbox\ident\ident\@idfrd\tf.m % idfrd method
```

No, it is not a good choice. What about using `T_f` as name of the data structure instead?

```
>> which -all T_f
T_f not found
```

Hence, `T_f` is both unused and a legal variable name, and can thus be used as the name for the data structure of feed temperatures.

Note that the built-in functions may vary from Matlab setup to setup: Depending on which *Toolboxes*<sup>2</sup> are installed on your computer, you may or may not get the same response as above on your computer.

Although we have used scalars as values for data structures in examples above, the right hand side (rhs) in the assignment (`<name> = <data structure>`) can be any data structure with value.

## 2.5 Built-in constants in Matlab

Matlab has some built-in names for constants. We have already seen the constant `namelengthmax` (in reality, this is not an assigned scalar, but a so-called function — more about functions later). Usually, it is a bad idea to “overwrite” the names of built-in constants, i.e. to use names for our own variables/constants which coincide with built-in Matlab names. Some built-in Matlab constants/names are shown in Table 2.1.

<sup>2</sup>A Toolbox is a collection of Matlab functions (or: classes) with a somewhat specialized common topic. Most Toolboxes are for sale at a cost that comes in addition to the cost of basic Matlab. Examples of Toolboxes are the Control Toolbox, the Statistics Toolbox, etc.

Table 2.1: Some built-in Matlab constants.

Name	Math name	Value	Description
<code>i</code>	$i$	$\sqrt{-1}$	used in Mathematics
<code>j</code>	$j$	$\sqrt{-1}$	used in Electrical Engineering
<code>pi</code>	$\pi$	—	—
<code>eps</code>	—	$2.2402 \times 10^{-16}$	Smallest number in Matlab $> 0$
<code>inf</code>	$\infty$	—	e.g., $1/0$
<code>NaN</code>	—	—	e.g., $0/0$ , Not a Number

The following excerpt from Matlab's Command Window illustrates how these constants are used.

```
>> i
ans =
    0 + 1.0000i
>> j
ans =
    0 + 1.0000i
>> pi
ans =
    3.1416
>> eps
ans =
    2.2204e-016
>> NaN
ans =
    NaN
>> inf
ans =
    Inf
>> 1/0
Warning: Divide by zero.
ans =
    Inf
>> 0/0
Warning: Divide by zero.
ans =
    NaN
```

Although it is possible to re-assign these constant names, we strongly recommend that this is avoided. Whether a name is a built in constant or not, can also be checked by the statement `which -all <name>`, as illustrated earlier.

## 2.6 Basic creation of arrays

### 2.6.1 Dimension of arrays

When organizing arrays, one could consider a scalar to be a *zero-dimensional* array, row arrays and column arrays to be *one-dimensional* arrays, and two dimensional arrays in general to be *two-dimensional* arrays. What is the significance of such a notion of dimension?

Consider the two dimensional array in Table 2.2.

Why is this array two-dimensional? Because we (normally) need two numbers to address each element. In Matlab, rows and columns are numbered with integers starting from 1. Furthermore, when addressing the elements, the row number is given first, and then the column number, e.g. the content of element (1, 1) is 1, while the content of element (2, 2) is  $-2 \times 10^{-3}$ .

However, if we consider only the first column in Table 2.2, this column is one-dimensional because we only need a single number (the row number) to address each element, the content of element (2) of

Table 2.2: Example array which we want to name *A*.

	Column 1	Column 2
<b>Row 1</b>	1	2
<b>Row 2</b>	3.7	$-2 \times 10^{-3}$
<b>Row 3</b>	8	$1.3 \times 10^5$

the first column is 3.7. Similarly, a row array is one dimensional. If we consider the scalar in the element of the third row and the second column (value:  $1.3 \times 10^5$ ), we don't need any number to address this element.

Assuming that we want to name the two dimensional array of Table 2.2 by **A**, how do we assign value to **A**? There are (at least) two ways to do this. We can give the value of each element in separate command lines — it doesn't really matter much which element we start with. Or we can assign the whole data structure in one batch.

## 2.6.2 Element-wise assignment

Let us start by assigning element no. (3,1):

```
>> A(3,1) = 8
A =
    0
    0
    8
```

Notice that Matlab immediately realized that since the row and column numbers must start with digit 1, **A** must at least be a  $3 \times 1$  array. Since we, the user, so far has only specified the value of element (3,1), Matlab is designed to assume that all the other elements have value 0.

Next, let us give the value of element (2,2):

```
>> A(2,2) = -2e-3
A =
    0    0
    0 -0.0020
 8.0000    0
```

Notice that  $-2e-3$  is Matlab syntax for  $-2 \times 10^{-3}$ .<sup>3</sup> This time, Matlab realizes that the element must have two columns. Since Matlab already knows that there are three columns, the array is augmented to become a  $3 \times 2$  array, which is the final size in this case. Thus filling the remaining values will not change the size of the array:

```
>> A(1,1) = 1;
>> A(2,1) = 3.7;
>> A(1,2) = [2];
>> A(3,2) = 1.3e5
A =
 1.0e+005 *
    0.0000    0.0000
    0.0000   -0.0000
    0.0001    1.3000
```

Note:

---

<sup>3</sup> $-2 \times 10^{-3}$  can also be written as  $-2*10^(-3)$  in Matlab, but the notation  $-2e-3$  is clearly the preferred (and simplest!) notation.



- When we end a command line with a *semicolon* (“;”), the result is not printed in the Command Window. If we do *not* end the command line with semicolon, the result is printed.
- Since a scalar is a zero-dimensional array, there is no problem in specifying the scalar as either 2 or [2].
- Even though it may seem like only two elements in the final result contain values different from zero, this is not so: here, all elements have values that differ from zero. The reason why 4 out of 6 elements appear to be zero, is the way Matlab by default formats the numbers. We will return to how these numbers can be formatted.

Suppose we want to change the value of one element, e.g. assume that element (2,2) should be  $+2 \times 10^{-3}$  instead of the current value. We then simply rewrite the value:

```
>> A(2,2) = 2e-3
A =
 1.0e+005 *
 0.0000    0.0000
 0.0000    0.0000
 0.0001    1.3000
```

We can also check the value of a certain element, e.g. element (2,2), as follows:

```
>> A(2,2)
ans =
 0.0020
```

Finally, it should be commented that this element-wise way of giving values to **A** is somewhat inefficient in that we have changed the size of the array twice: the first time when we gave the value of element (3,1) and thus creating a  $3 \times 1$  array, and the second time when we gave the value of element (2,2) and thus augmented the array to a  $3 \times 2$  array.

### 2.6.3 Assigning row arrays

If we consider the two dimensional array of Table 2.2, we can define three row arrays:

```
>> A_r1 = [1,2]
A_r1 =
 1    2
>> A_r2 = [3.7, -2e-3]
A_r2 =
 3.7000   -0.0020
>> A_r3 = [8 1.3e5]
A_r3 =
 8    130000
>> A_r2(2)
ans =
 -0.0020
```

Here we notice that arrays are indicated by including the numbers in square braces [ and ]. Furthermore, we can separate the various elements of each row either by comma, or by a space. Since each of the row arrays are one-dimensional, it suffices to specify one number to indicate which element we want to check, e.g. `A_r2(2)`.

### 2.6.4 Assigning column arrays

Column arrays are constructed similar to the way row arrays are constructed. However, for column arrays, either semicolon or Carriage Return (**CR**) is used to separate the elements:

```
>> A_c1 = [1; 3.7; 8]
A_c1 =
    1.0000
    3.7000
    8.0000
>> A_c2 = [2
-2e-3
    1.3e5]
A_c2 =
 1.0e+005 *
    0.0000
   -0.0000
    1.3000
>> A_c2(2)
ans =
   -0.0020
```

Alternatively, we can pretend that the column array is a column matrix, and use the operation of *transposing* the matrix to turn the row array into a column array. The matrix transpose of a matrix  $M$ ,  $M^T$ , is performed using the notation  $M'$  in Matlab<sup>4</sup>. Thus, we would have defined e.g. `A_c2` by first creating it as a row array, and then transposing the result:

```
>> A_c2 = [2, -2e-3, 1.3e5]
A_c2 =
 1.0e+005 *
    0.0000
   -0.0000
    1.3000
```

The techniques of either separating elements by semicolon, or by transposing a row array, are normally preferred since fewer lines in the Command Window are needed. However, separating elements by **CR** makes the Command Window more readable.

### 2.6.5 Assigning two-dimensional arrays

We now consider the two dimensional array in Table 2.2 as a number of rows below each other. The following notation is used to assign the value of the array to the name of the array:

```
>> A = [1, 2; 3.7 -2e-3;
8, 1.3e5]
A =
 1.0e+005 *
    0.0000    0.0000
    0.0000   -0.0000
    0.0001    1.3000
```

Notice that we list the first row first (either separated by comma or space), then separate the first row from the second row (either by semicolon or **CR**), and so on. Often, the two dimensional array is easier to read if **CR** is used to separate rows.

<sup>4</sup>Actually, operator “'” takes both the transpose, and the conjugate of matrix  $M$ . We will return with a more precise description.

**Exercise 2.4 (Chaos)** Write:

```
x = 0.7
x = 2*x*(1-x)
```

Repeat the last instruction several times. (Use the up arrow!) Do you see any relationship between the result and the equation?

We can make a side-by-side comparison of the iteration sequence from two nearby initial values for  $x$  (e.g. 0.7 and 0.7001) by

```
x = [0.7 0.7001]
x = 2*x.*(1-x)
```

Note the `.*` operator, signifying element by element multiplication rather than the standard matrix multiplication! Again, repeat the last command several times to see how  $x$  changes.

Now, try with a bigger coefficient in the right-hand side:

```
x = [0.7 0.7001]
x = 4*x.*(1-x)
```

Note that when we repeat the last command, the values do not tend towards a constant, but rather flutter wildly about. This is an example of *chaotic behaviour*.

The results from the two close starting points diverge rapidly. In a chaotic system, a small change in the initial value may yield very different long-time results. This is called “the butterfly effect”: if the weather is a chaotic phenomenon, then the very small change in the motion of the air caused by the flapping of a butterfly’s wings in Brazil might in theory trigger a storm in the Indian Ocean some weeks or months later. This effect may indeed limit the feasibility of long-range weather forecasts. ■

### 2.6.6 Line continuation

For large arrays, it may be difficult to fit one row on one command line. We may then split the row across several command lines by using the line continuation operator `...`:

```
>> A = [1, 2; 3.7 ...
-2e-3; 8, 1.3e5]
A =
 1.0e+005 *
 0.0000    0.0000
 0.0000   -0.0000
 0.0001    1.3000
```

Here, Matlab will consider the assignment as having taken place on a single command line.

### 2.6.7 One-dimensional addressing of array elements\*

For any array, we can address the element element by a single number. For a scalar, the number is 1. For row or column arrays, the number is the location of the element in the array. For two-dimensional arrays, we imagine that the columns are stacked on top of each other, with the first column on the top, and the last column on the bottom.

```
>> A
A =
 1.0e+005 *
 0.0000    0.0000
 0.0000   -0.0000
 0.0001    1.3000
>> A(6)
ans =
    130000
>> A(5)
ans =
   -0.0020
```

Usually, it is more complicated to figure out the element address using a single number address.

## 2.7 Exporting and importing data from files

It is important to be able to save data from the workspace/a Matlab session to a file, so that we can easily import it again at a later time. It may also be important to be able to export the data to some file format that can be imported into other programs, or to import data from other programs such as Excel, etc. Here, we will only discuss saving data to and loading data from basic Matlab formats, and import of data from Excel.<sup>5</sup>

### 2.7.1 Saving and loading data

Suppose we have two arrays,  $A$  and  $B$ :

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 824 & -65 & -814 & -741 \\ -979 & -764 & 216 & 663 \\ 880 & 916 & 617 & -535 \end{pmatrix}.$$

We create these in Matlab as follows:

```
>> A = [1, 2; 3, 4];
>> B = [824, -65, -814, -741; -979, -764, 216, 663;
        880, 916, 617, -535];
```

Suppose these two data structures are the only ones in the workspace of Matlab at the time being. First, we want to save all data structures of the workspace to a file. Before doing so, we need to determine:

1. The location of the file on the computer. The default directory where the file will be placed, is the *Current Directory*, see fig. 1.1 on p. 4. Clicking the down-arrow (v) for the Current Directory brings up a list of previously used directories, which can be selected. Clicking the ellipsis icon (...) to the right of the Current Directory list, brings up a file browser that can be used to select the directory.
2. What file type to save the workspace in. The default file type is Matlab's own `.mat` type. The alternative is to save the workspace as an `.ascii` file. Note that the `.mat` type works both faster, and preserves more information.

The basic syntax for saving files is `save <filename> {-<filetype>}`, where `<filetype>` can be either `mat` or `ascii` (`mat` is the default), and `{}` indicates that what is between these braces is optional. Here is an example of saving the workspace to both file types:

```
>> save myworkspacem1 -mat
>> save myworkspacea -ascii
>> save myworkspacem2
```

Note that saving to `myworkspacem2` also leads to a file of type `.mat`, i.e. `myworkspacem2.mat`. The reason is that `.mat` is the default file type.

Next, imagine that we clear the workspace so that there is nothing there. This will typically be the case when we start up Matlab, but we can also enforce an empty workspace (more about that later). So, the workspace is empty. Let us then `load` the file containing the workspace we just saved. The syntax is `load {-<filetype>} <filename>`<sup>6</sup>. We first see what happens when we load the `.mat` files.

```
>> load myworkspacem1
```

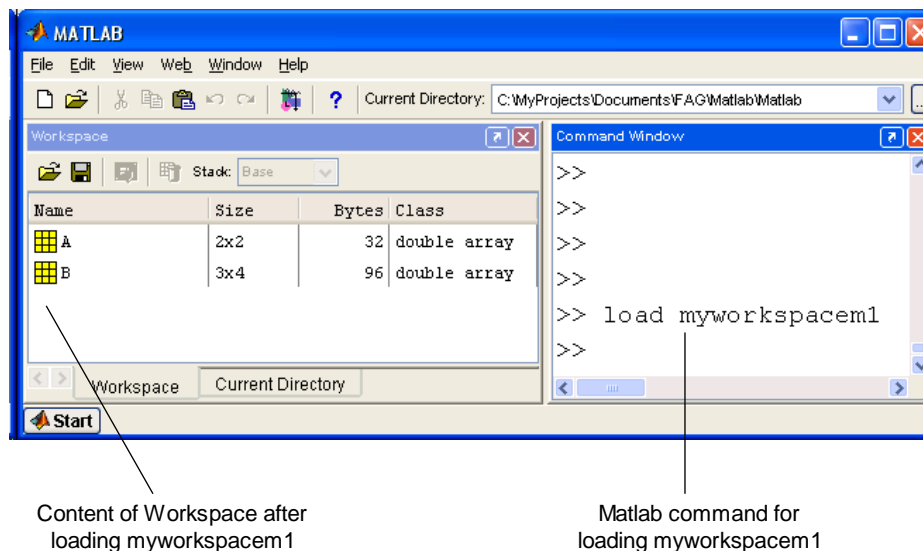


Figure 2.1: The result after loading a .mat file.

The result is shown in fig. 2.1. We see that the original variable names are preserved. What then if we load a .ascii file? It turns out that Matlab can not load the file `myworkspacem1`. The reason is that the arrays have a different number of columns:

```
>> load myworkspacem1
??? Error using ==> load
Number of columns on line 3 of ASCII file
C:\MyProjects\Documents\FAG\Matlab\Matlab\myworkspacem1 must be the same as
previous lines.
```

See FILEFORMATS for a list of known file types and the functions used to read them.

If we open the file in a text editor (e.g. Notepad, or the Matlab editor), the ASCII file looks as in fig. 2.2. This indicates that the ASCII format is not useful for storing more than one data structure, and perhaps it is not suitable at all for some data structures. Note also that there is no information about the array names in the file.

It is possible to use more details when saving and loading the workspace. We can choose to only save part of the workspace. The syntax is: `save <filename> {<variablename1> ... <variablenameN>} {-<filetype>}`, where `{<variablename1> ... <variablenameN>}` is meant to indicate that we can list any number of variable names, and that they should be separated by *space* (not comma!). Similarly, the syntax for loading files is `load {-<filetype>} <filename> {<variablename1> ... <variablenameN>}`. Obviously, since ASCII files do not preserve information about variable names, and are only useful for saving a single variable, for ASCII files it is not relevant to choose which variable to load. The following examples illustrate the use of `save` and `load`.

```
>> save myworkspacem1
>> save myworkspacem1 A
>> save myworkspacem1 B
>> save myworkspacem1 A -ascii
>> save myworkspacem1 B -ascii
```

<sup>5</sup>It is possible to import data using commands such as `fileread`, `fread`, etc.

<sup>6</sup>Note that for saving files, the file type is given after the filename, while for loading files, the file type is given before the filename.

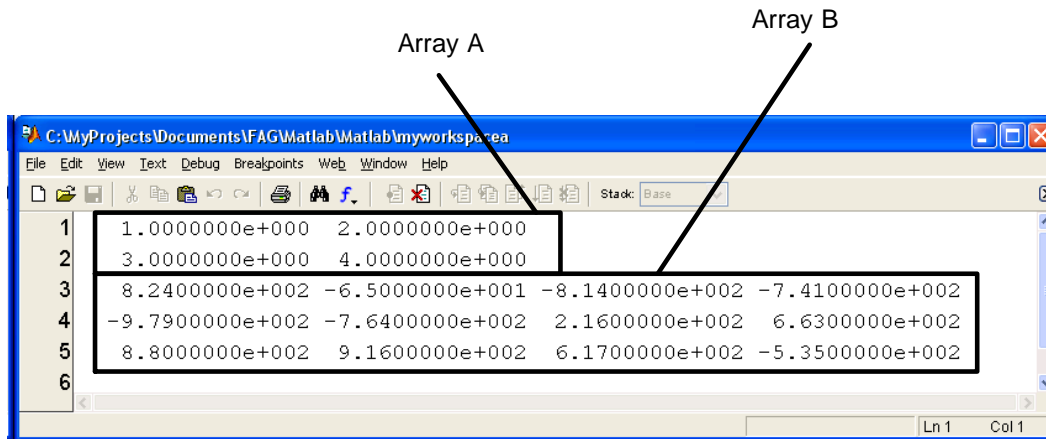


Figure 2.2: The content of the ASCII file. Note that the first two rows come from the *A* array, while the three next rows come from the *B* array.

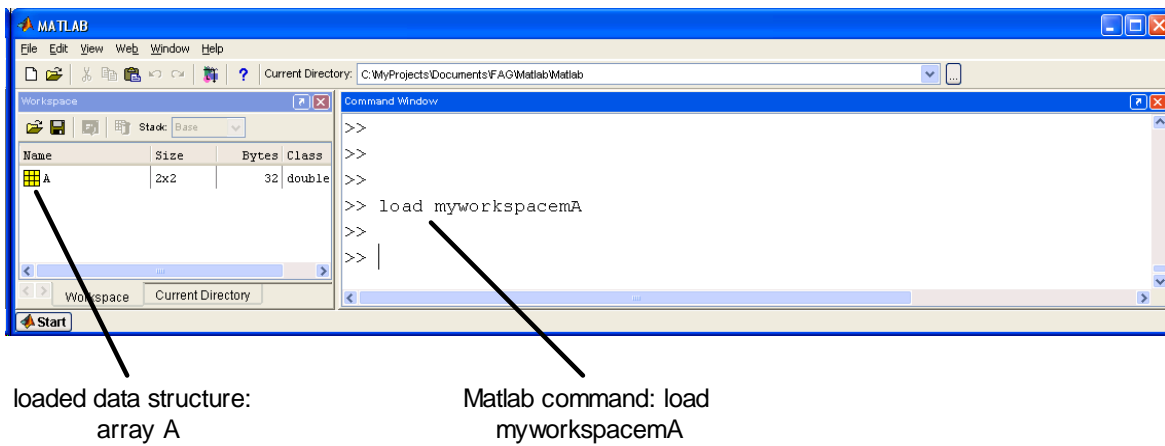


Figure 2.3: Situation after loading file *myworkspacemA* (a *.mat* file), which only contains array *A*.

This leads to the following files:

```

19.07.2003 16:01          66 myworkspaceA
19.07.2003 16:01        195 myworkspaceB
19.07.2003 16:01        264 myworkspacem.mat
19.07.2003 16:01        184 myworkspacemA.mat
19.07.2003 16:01        208 myworkspacemB.mat
                5 File(s)          917 bytes

```

What happens if we start with an empty workspace and load these files? The following examples illustrates what takes place.

```
>> load myworkspacemA
```

This operation leads to the situation in fig. 2.3. Similarly:

```
>> load myworkspaceB
```

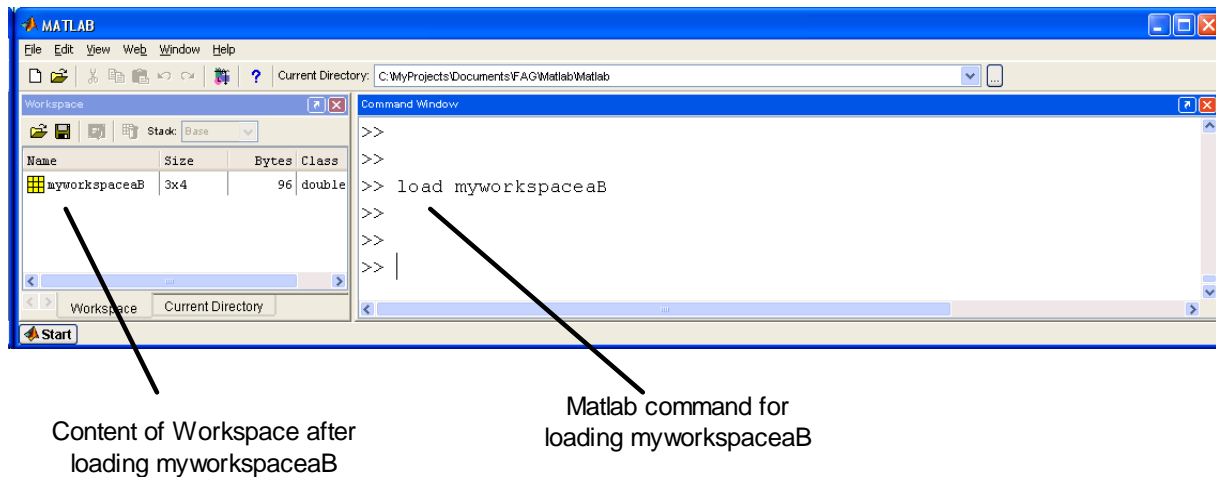


Figure 2.4: Situation after loading file `myworkspaceaB` (an `.ascii` file), which only contains array `B`.

This operation leads to the situation in fig. 2.4. Note in particular that since the ASCII file does not contain information about the name of the stored array, the loaded variable takes its name from the loaded file. Thus, what used to be array `B` now is named `myworkspaceaB`.

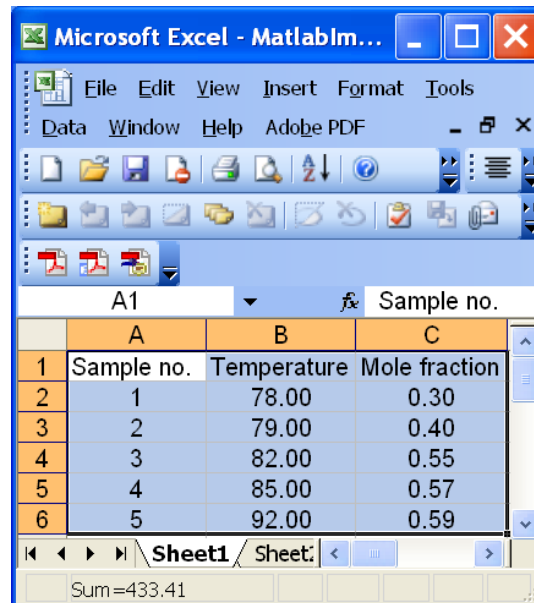
## 2.7.2 File import

The simplest way of importing data from Excel is via the Matlab `File/Import Data...` command. This command can also be used to import data saved using the `save` command of Matlab (see previous subsection). This command simply opens a file browser which shows files that Matlab recognizes. Suppose we have an Excel spreadsheet as in fig. 2.5. In the Matlab window, we choose the command `File/Data Import...`, which opens a file browser.<sup>7</sup> We choose the relevant file (`MatlabImportData.xls`, see fig. 2.5), and click `Open`. The Import Wizard is opened, and leads to the result in fig. 2.6. Clicking `Finish` in the Matlab Import Wizard populates the Matlab workspace with the variables `data`, `textdata`, and `colheaders`. These data look as follows:

```
>> data
data =
    1.0000    78.0000    0.3000
    2.0000    79.0000    0.4000
    3.0000    82.0000    0.5500
    4.0000    85.0000    0.5700
    5.0000    92.0000    0.5900
>> colheaders
colheaders =
    'Sample no.'    'Temperature'    'Mole fraction'
>> textdata
textdata =
    'Sample no.'    'Temperature'    'Mole fraction'
```

So far, we have learned how to *build* the data structure (array) of variable `data` (but not how to build the data structures `colheaders` and `textdata` — which are cell arrays). In order to analyze the data contained in variable `data`, we need to be able to deconstruct the variable, and pick out elements of the array. We'll get back to that problem soon.

<sup>7</sup>It is required that Excel is installed on your computer for this to work: Matlab communicates with Excel in order to import the data.



Microsoft Excel - MatlabIm...

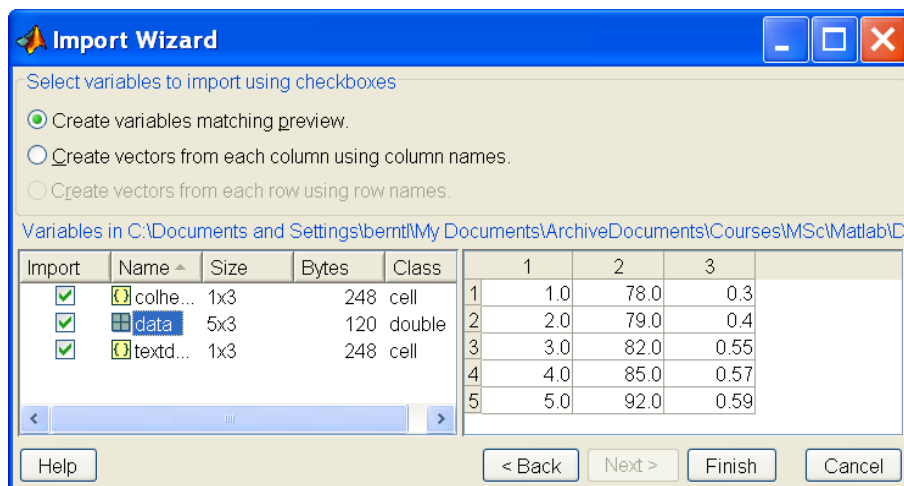
File Edit View Insert Format Tools  
Data Window Help Adobe PDF

A1 Sample no.

	A	B	C
1	Sample no.	Temperature	Mole fraction
2	1	78.00	0.30
3	2	79.00	0.40
4	3	82.00	0.55
5	4	85.00	0.57
6	5	92.00	0.59

Sheet1 Sheet: Sum=433.41

Figure 2.5: Excel spreadsheet with data that we want to import into Matlab.



Import Wizard

Select variables to import using checkboxes

Create variables matching preview.  
 Create vectors from each column using column names.  
 Create vectors from each row using row names.

Variables in C:\Documents and Settings\bernt\My Documents\ArchiveDocuments\Courses\MSc\Matlab\D...

Import	Name	Size	Bytes	Class
<input checked="" type="checkbox"/>	colhe...	1x3	248	cell
<input checked="" type="checkbox"/>	data	5x3	120	double
<input checked="" type="checkbox"/>	textd...	1x3	248	cell

	1	2	3
1	1.0	78.0	0.3
2	2.0	79.0	0.4
3	3.0	82.0	0.55
4	4.0	85.0	0.57
5	5.0	92.0	0.59

Help < Back Next > Finish Cancel

Figure 2.6: The result of the Matlab Import Wizard.



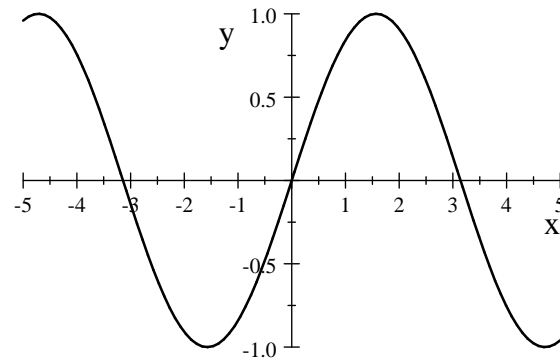


Figure 2.7: The function  $y = \sin x$ , where  $x$  is the input argument, and  $y$  is the output argument. This is an example of a relationship between  $x$  and  $y$ , which is also a function.

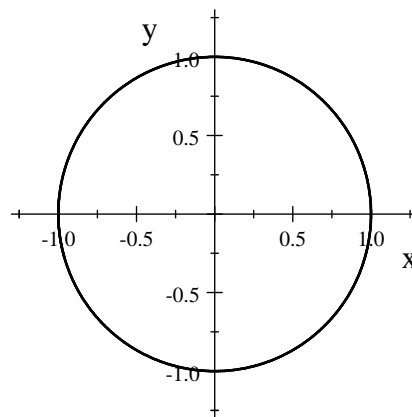


Figure 2.8: Implicit plot of  $x^2 + y^2 = 1$ . This is an example of a relationship between  $x$  and  $y$  which is not a function.

## 2.8 Functions and operations

The basic idea of *functions* is known from high school mathematics: A function is an operation that maps its input argument(s) uniquely into its output argument(s). The basic property is the uniqueness: with a specified (input) argument, the answer (output argument) is uniquely determined. There is no room for doubt or randomness. An example of a function is  $y = \sin x$ , which is shown in fig. 2.7. When we fix the input argument (the value of  $x$ ), the value for the output argument ( $y$ ) is given uniquely.

An example of a curve that does not satisfy the property of a function, is the implicit plot of  $x^2 + y^2 = 1$ , fig. 2.8. This is not a function simply because if we fix the input argument (e.g.  $x$ ), there is not a unique output argument ( $y$  when  $x$  is the input argument). This non-uniqueness causes the mapping between  $x$  and  $y$  to not be a function.

Matlab has many built-in functions. The typical structure of calling these functions are: `output_argument = function_name(input_argument)`. The function name must satisfy the requirements for Matlab variable names. The input and output arguments can be any data structure, and can also be lists (comma separated) of variables/data structures. There can even be a variable number of input and output arguments, depending on the properties of the input arguments and how much details of the output arguments the user is interested in. The requirement for making these functions is still that when the input arguments are given, there is a unique answer (output argument).

Let us consider what takes place during the common mathematical operation of multiplication, e.g. consider

$$y = 3x.$$

Here, the input argument  $x$  is multiplied by the number 3 in order to obtain the output argument  $y$ .

There is a unique relationship between  $x$  and  $y$ , so this is a function. In order to compute the output argument in Matlab, we can use e.g. the built-in function `c = mtimes(a,b)`:

```
>> x = 5;
>> y = mtimes(3,x)
y =
    15
```

This is a simple example of using a function in Matlab. There are some restrictions on what the input arguments can be for this function to work, but let us not worry about this now.

In the particular case of function `mtimes` (and a few other functions), we know from mathematics that it is more convenient to introduce the *operator* multiplication. Thus, we can instead express  $y = 3 \cdot x$  as follows:

```
>> x = 5;
>> y = 3*x
y =
    15
```

Every operation in Matlab has an equivalent function. But there are many functions that do not have an equivalent operator. Although we in the examples above have used scalar arguments, many functions accept or need other data structures as arguments (e.g. arrays). We will get back to this later.

**Exercise 2.5** Consider the mathematical expression  $z = 3x - 2y + \frac{(x-y)^2}{x+y}$ . Using the operator `+`, `-`, `*`, `/`, and `^`, this can be written as:

```
z = 3*x - 2*y + ((x-y)^2)/(x+y)
```

in Matlab. Check this, by first assigning values to `x` and `y`.

Next, try to repeat this using the corresponding functions `plus`, `-`, `mtimes`, `mrdivide`, and `mpower`:

```
z = plus(mtimes(3,x), plus(mtimes(-2,y), mrdivide(mpower(plus(x,-y),2), plus(x,y)))));
```

Do you get the same result? Which formulation is simplest? ■

## 2.9 Numeric format and accuracy

All commands in Matlab are carried out in double precision<sup>8</sup>. 8 bytes are used to store each element in an array, according to some IEEE standard<sup>9</sup>. In practice, this means that there is ca. 16 decimal digits accuracy in the computations, and a range of roughly  $[10^{-308}, 10^{+308}]$ . Many times, the accuracy in observed data that are used in computations is much lower, but still, Matlab always works with double precision, and the full accuracy is always stored to files. We can, however, influence how the data are presented to the user in the Matlab Command Window.

Table 2.3 shows the available commands for manipulating the format of the data as they are presented to the user.

The command `format compact` is useful for saving space when inserting Matlab commands/responses into a word processor. As an example, consider our familiar array:

```
>> format compact
>> a = [3, 2; 3.7, -2e-3; 8, 1.3e5]
a =
 1.0e+005 *
    0.0000    0.0000
    0.0000   -0.0000
    0.0001    1.3000
```

<sup>8</sup>Since ca. version 6.5 of Matlab, there is also support for other data types such as single precision, integer, boolean, etc.

<sup>9</sup>IEEE = The Institute of Electrical and Electronics Engineers, Incorporated, is an engineering organization with over 365 000 members in more than 105 countries; about 40% of the members are from outside of the USA.

Table 2.3: Available commands to format data structures in Matlab.

Command	Description	Comment
<code>format</code>	Default. Same as <code>short</code> (see below).	
<code>format short</code>	Scaled fixed point format with 5 digits.	default
<code>format long</code>	Scaled fixed point format with 15 digits.	
<code>format short e</code>	Floating point format with 5 digits.	
<code>format long e</code>	Floating point format with 15 digits.	
<code>format short g</code>	Best of fixed or floating point format with 5 digits.	
<code>format long g</code>	Best of fixed or floating point format with 15 digits.	
<code>format hex</code>	Hexadecimal format.	
<code>format +</code>	The symbols +, - and blank are printed for positive, negative and zero elements. Imaginary parts are ignored.	
<code>format bank</code>	Fixed format for dollars and cents.	
<code>format rat</code>	Approximation by ratio of small integers.	
<code>format compact</code>	Suppress extra line-feeds.	
<code>format loose</code>	Puts the extra line-feeds back in.	default

This (default) *formatting* is with 5 digits accuracy. Let's change this to 15 digits accuracy:

```
>> format long
>> a
a =
 1.0e+005 *
 0.000030000000000    0.000020000000000
 0.000037000000000   -0.000000020000000
 0.000080000000000    1.300000000000000
```

**Exercise 2.6** Experiment with each of the other `format` statements using the above array `a`. ■

Internally, computers use binary numbers to store the numbers. This means that Matlab introduces inaccuracies in the computations: e.g.,  $1/3$  is not represented as  $1/3$ , but rather as `3fd5555555555555` in Hexadecimal numbers:

```
>> format short
>> 1/3
ans =
 0.3333
>> format hex
>> ans
ans =
 3fd5555555555555
```

See Appendix C for a more detailed example on how Hexadecimal numbers work.

## 2.10 Functions for creating arrays

Table 2.4 shows some functions for creating matrices. Let us illustrate the basic use of some of these functions.

### 2.10.1 Creating arrays

**The colon operator** The colon operator is often used with integers, but can also be used with floating point numbers:

Table 2.4: Some commands for creating and manipulating arrays in Matlab.

Command	Notation	Description
<code>m:k:n</code> , <code>colon(m,k,n)</code>	$m:k:n$	Colon operator for creating row array where the first element has value $m$ , the last element has value $n$ , and the elements in between are evenly spaced with distance $k$ .
<code>linspace(x,y,n)</code>	—	Creates row array with $n$ elements, where the first element has value $x$ , the last element has value $y$ , and the elements in between are evenly (linearly) spaced.
<code>logspace(x,y,n)</code>	—	Creates row array with $n$ elements, where the first element has value $10^x$ , the last element has value $10^y$ , and the elements in between are logarithmically spaced.
<code>zeros(m,n)</code>	$0_{m,n}$	$m \times n$ -array filled with 0.
<code>ones(m,n)</code>	$1_{m,n}$	$m \times n$ -array filled with 1.
<code>eye(m,n)</code>	$I_{m,n}$	$m \times n$ -array with 1 on the main diagonal, and 0 elsewhere.
<code>diag(v)</code>	$\text{diag}(A, B)$	If $v$ is an array of length $n$ : creates an $n \times n$ -array with the elements of $v$ on the main diagonal.
<code>rand(m,n)</code>	—	$m \times n$ -array with uniformly distributed random numbers in the interval $(0, 1)$ .
<code>randn(m,n)</code>	—	$m \times n$ -array with normally distributed random numbers $\sim \mathcal{N}(0, 1)$ .
<code>size(A)</code> , <code>size(A,n)</code>	—	Produces a row vector where the first element is the number of rows of array $A$ , and the second number is the number of columns of $A$ .
<code>length(A)</code>	—	The maximum of the two numbers in <code>size(A)</code> .
<code>reshape(A,k,l)</code>	—	$m \times n$ -array $A$ is reshaped into a $k \times \ell$ -array; this is only possible if $m \cdot n = k \cdot \ell$ .
<code>rot90(A,k)</code>	—	Array $A$ is “rotated” $k \times 90^\circ$ counterclock-wise.
<code>blkdiag(A,B)</code>	$\text{diag}(A, B)$	Block-diagonal matrix with matrices $A$ and $B$ on the main diagonal blocks.
<code>toeplitz(c,r)</code>	—	Produces a Toeplitz array with $c$ as the first column, and $r$ as the <i>first</i> row. With a single argument, $c$ and $r$ are assumed equal.
<code>hankel(c,r)</code>	—	Produces a Hankel array with $c$ as the first column, and $r$ as the <i>last</i> row. With a single argument, $c$ is given, and the array is square.
<code>repmat(A,m,n)</code>	—	Replicates a matrix: matrix $A$ is stacked together into a super-matrix with $m$ block rows of copies of $A$ , and $n$ block columns of copies of $A$ .
<code>kron(A,B)</code>	$A \otimes B$	Produces the Kronecker-product of $A$ and $B$ , $A \otimes B$ .

```

>> 1:2:6
ans =
     1     3     5
>> 6:-3:-6
ans =
     6     3     0    -3    -6
>> pi:pi/10:2*pi
ans =
Columns 1 through 5
     3.1416     3.4558     3.7699     4.0841     4.3982
Columns 6 through 10
     4.7124     5.0265     5.3407     5.6549     5.969
Column 11
     6.2832

```

The colon operator is used extensively in operation on arrays, as we will see soon. Note that the end point ( $n$  of  $m:k:n$ ) is only included if it can be written as  $n = m + k \cdot i$  where  $i$  is some integer number. Also notice that when using the colon operator, we do *not explicitly* specify the number of elements in the row array.

The *function* for doing the same as the colon *operator*, is `colon(m,k,n)`:

```

>> colon(1,2,6)
ans =
     1     3     5

```

**Linspace and logspace** With the `linspace` command, we do not specify the distance between the numbers, but we do specify the number of elements in the row array:

```

>> linspace(0,2*pi,5)
ans =
     0     1.5708     3.1416     4.7124     6.2832

```

If the number of elements is left out (e.g., we write `linspace(x,y)`), then the default number of elements is 50.

When using the `logspace` command, the result is as follows:

```

>> format short g
>> logspace(-3,1,5)
ans =
     0.001     0.01     0.1     1     10

```

**Zeros and ones** Using function `zeros` is exemplified by the following Matlab session:

```

>> zeros(2,3)
ans =
     0     0     0
     0     0     0
>> zeros(2)
ans =
     0     0
     0     0

```

Notice that when we only use one argument in the function, the resulting array is square.

Function `ones` works similarly to function `zeros`:

```
>> ones(2,3)
ans =
     1     1     1
     1     1     1
>> ones(2)
ans =
     1     1
     1     1
```

**Eye and diag** Function `eye` also works similarly:

```
>> eye(2,3)
ans =
     1     0     0
     0     1     0
>> eye(2)
ans =
     1     0
     0     1
```

Function `eye` is typically used to generate the *identity matrix* in linear algebra.

Function `diag` works as follows:

```
>> diag([1,2,3])
ans =
     1     0     0
     0     2     0
     0     0     3
```

Note that `diag` can be used to create square “eye” arrays:

```
>> diag(linspace(1,1,3))
ans =
     1     0     0
     0     1     0
     0     0     1
```

Note also that function `diag` can have a second argument: `diag(v,k)`. The second argument is then an integer, and specifies whether the diagonal should be placed above the main diagonal (positive  $k$ ) or below the main diagonal (negative  $k$ ).

```
>> diag(1:2)
ans =
     1     0
     0     2
>> diag(1:2,0)
ans =
     1     0
     0     2
```

```
>> diag(1:2,-1)
ans =
     0     0     0
     1     0     0
     0     2     0
>> diag(1:2,2)
ans =
     0     0     1     0
     0     0     0     2
     0     0     0     0
     0     0     0     0
```

Finally, function `diag` can also be used to construct an array by *extracting* the diagonal from a matrix — the result is a column array:

```
>> A = [1, 2; 3.7 -2e-3; 8, 1.3e5]
A =
     1     2
    3.7   -0.002
     8   1.3e+005
>> diag(A)
ans =
     1
   -0.002
>> diag(A,-1)
ans =
     3.7
   1.3e+005
```

**Random numbers** *Uniformly distributed* random numbers are created using function `rand`:

```
>> rand(2)
ans =
    0.95013    0.60684
    0.23114    0.48598
>> rand(2)
ans =
    0.8913    0.45647
    0.7621    0.018504
>> rand(2,3)
ans =
    0.82141    0.61543    0.92181
    0.4447    0.79194    0.73821
```

Here, we notice that in the first two calls to the function (calling `rand(2)` twice), the result is different each time. The reason is that the numbers are supposed to be random, and thus they should not be the same. For comparisons, we often want to enforce the results of two drawings to be the same. We can achieve this by *resetting* the algorithm with command `rand('state',0)`:

```
>> rand(2)
ans =
    0.95013    0.60684
    0.23114    0.48598
>> rand('state',0)
```

```
>> rand(2)
ans =
    0.95013    0.60684
    0.23114    0.48598
```

The random generator for *normally distributed* numbers works in the same way, e.g.:

```
>> randn(2)
ans =
   -0.43256    0.12533
   -1.6656    0.28768
```

### 2.10.2 Size of arrays

**Size** Sometimes we don't know the size of an array, and it is useful to find the size. This can be done by using function `size`. Suppose we have an array  $A$  with an unknown size  $m \times n$ . We find the size as follows:

```
>> A = ones(20,30);
>> size(A)
ans =
    20    30
```

Here, we can argue that we already knew the size: we had just created matrix  $A$  to be of size  $20 \times 30$ . But we can easily get into a situation where we do not really know the size, or where it is too complicated to figure it out; thus it may be easier to use command `size` to find the size. Let us expand slightly on the example:

```
>> sA = size(A)
sA =
    20    30
>> m = sA(1)
m =
    20
>> n = sA(2)
n =
    30
```

We see that we can easily pick out the number of rows and the number of columns.

If we only care about a specific dimension (e.g. the number of rows, or the number of columns), then we can pick out this as follows:

```
>> size(A,1)
ans =
    20
>> size(A,2)
ans =
    30
```

**Length** Sometimes — typically for one-dimensional arrays, we want to find the length of the array. For this case, it is simpler to use the `length` function:



```
>> v = 1:3:20;
>> length(v)
```

```
ans =
```

```
7
```

```
>> size(v)
```

```
ans =
```

```
1 7
```

**Application of size** Let us finally consider a simple application of the `size` command: suppose we have an array of unit elements (1),  $A$ , and that we want to create an array  $B$  of the same size, but with uniformly distributed random numbers. This can be achieved as follows:

```
>> A = ones(2,3)
```

```
A =
```

```
1 1 1
1 1 1
```

```
>> B = rand(size(A))
```

```
B =
```

```
0.012863 0.68312 0.035338
0.38397 0.092842 0.6124
```

### 2.10.3 Rearranging arrays

**Reshape** Sometimes, it is useful to rearrange the shape of an array. Suppose we have an array  $A$  of dimension  $m \times n$ , and that we want to turn this array into an array  $B$  of dimension  $j \times k$ . As long as  $m \cdot n = j \cdot k$ , arrays  $A$  and  $B$  will contain the same number of elements. When this dimension requirement  $m \cdot n = j \cdot k$  is fulfilled, Matlab function `reshape(A,j,k)` can be used to change the shape of  $A$  from being a  $3 \times 2$  array, into being a  $2 \times 3$  array:

```
>> A = rand(3,2)
```

```
A =
```

```
0.81317 0.20277
0.0098613 0.19872
0.13889 0.60379
```

```
>> B = reshape(A,2,3)
```

```
B =
```

```
0.81317 0.13889 0.19872
0.0098613 0.20277 0.60379
```

Conceptually, this command may be carried out as follows:

1. First, matrix  $A$  is made into a column array with  $m \cdot n$  rows (and one column).
2. Then, the first  $j$  elements of the column array are placed in the first column of array  $B$ , the next  $j$  elements of the column array are placed in the next column of  $B$ , and so on, until  $B$  has  $k$  columns.

**Rotate** When considering the geometric layout of an array, it is sometimes of interest to rotate this layout. Function `rot90` rotates an array counterclockwise by  $90^\circ$ :

```
>> A = rand(2,3)
A =
    0.46599    0.84622    0.20265
    0.41865    0.52515    0.67214
>> rot90(A)
ans =
    0.20265    0.67214
    0.84622    0.52515
    0.46599    0.41865
>> rot90(ans)
ans =
    0.67214    0.52515    0.41865
    0.20265    0.84622    0.46599
>> rot90(ans)
ans =
    0.41865    0.46599
    0.52515    0.84622
    0.67214    0.20265
>> rot90(ans)
ans =
    0.46599    0.84622    0.20265
    0.41865    0.52515    0.67214
```

We see that after 4 rotations, we are back where we started. Sometimes, it may be useful to combine the `rot90` function with the operation of transposing an array.

## 2.11 Subarrays and superarrays

### 2.11.1 Picking subarrays

**Indexing the array** In this section, we only give a basic presentation of how to index arrays. See Appendix B.1 p. 159 for a more complete discussion.

Suppose array  $A$  is given. We often wish to pick out element  $(i, j)$  from array  $A$ . Let us denote element  $(i, j)$  of array  $A$  by  $A_{i,j}$ , which in Matlab is given as `A(i,j)`.

For array  $A$ , it can also be useful to be able to pick a subarray which has indices over the rectangle given by row  $i_1$  to row  $i_2$ , and column  $j_1$  to  $j_2$ . This submatrix can be denoted as  $A_{i_1:i_2, j_1:j_2}$ . In Matlab, this subarray is specified as `A(i1:i2, j1:j2)`:

```
>> rand('state',0)
>> A = rand(3,4)
A =
    0.95013    0.48598    0.45647    0.4447
    0.23114    0.8913    0.018504    0.61543
    0.60684    0.7621    0.82141    0.79194
>> A(1,3)
ans =
    0.45647
>> A(2:3,2:3)
ans =
    0.8913    0.018504
    0.7621    0.82141
```

**Properties of elements** Often, it is desirable to find elements of arrays with certain properties. We can use the function `find` to find the indices of such elements. Let us find the elements of array  $A$  with elements which satisfy  $A(i,j) > 0.5$ . This can be done as follows:

```
>> A
A =
    0.95013    0.48598    0.45647    0.4447
    0.23114    0.8913    0.018504    0.61543
    0.60684    0.7621    0.82141    0.79194
>> ind = find(A>0.5)
ind =
     1
     3
     5
     6
     9
    11
    12
```

Clearly, here the index must be interpreted as a one-dimensional addressing of the elements of  $A$ , i.e. the row number in  $\text{col}(A)$ . We can now find the value of these elements by using the following syntax:

```
>> A(ind)
ans =
    0.95013
    0.60684
    0.8913
    0.7621
    0.82141
    0.61543
    0.79194
```

We can also use the found index to “clip” the values of  $A$  such that the maximal value is 0.5:

```
>> A(ind) = 0.5
A =
    0.5    0.48598    0.45647    0.4447
    0.23114    0.5    0.018504    0.5
    0.5    0.5    0.5    0.5
```

### 2.11.2 Building superarrays

**Concatenation** It is possible to concatenate several arrays into superarrays. The concatenation operator is `[]`, and this operator works just as if the arrays we concatenate, are scalars:

```
>> A = rand(2,3)
A =
    0.41027    0.057891    0.81317
    0.89365    0.35287    0.0098613
>> B = [A, A; A, A]
B =
    0.41027    0.057891    0.81317    0.41027    0.057891    0.81317
    0.89365    0.35287    0.0098613    0.89365    0.35287    0.0098613
    0.41027    0.057891    0.81317    0.41027    0.057891    0.81317
```

```
0.89365    0.35287    0.0098613    0.89365    0.35287    0.0098613
```

There are some obvious restrictions on how to concatenate arrays.

- Every element of each block row must have the same number of rows.
- The total number of columns in each block row must be the same.

```
>> C = [[1,2;2,4], A;
        A, [4,7;2,2]]
C =
     1         2    0.41027    0.057891    0.81317
     2         4    0.89365    0.35287    0.0098613
 0.41027    0.057891    0.81317         4         7
 0.89365    0.35287    0.0098613         2         2
>> D = [ [1;2], A, [4;2];
        A, [4,7; 2,2]]
D =
     1    0.41027    0.057891    0.81317         4
     2    0.89365    0.35287    0.0098613         2
 0.41027    0.057891    0.81317         4         7
 0.89365    0.35287    0.0098613         2         2
```

Note here that the first array `[1,2;2,4]` of block row 1 of `C` has the same number of rows as array `A`. Likewise, array `A` has the same number of rows as array `[4,7; 2,2]`. This fulfills the requirement for the rows. Finally, the number of columns for `[1,2; 2,4]` and `A` are the same as the number of columns for `A` and `[4,7; 2,2]`. This fulfills the requirement for the columns. Similarly, the requirements are fulfilled for array `D`.

**Replication of arrays** Sometimes, we need to replicate an array many times. Function `repmat` (replicate matrix) makes a copy of the array in question: `E = repmat(A,m,n)` produces  $m \cdot n$  copies of array `A`, with  $m$  copies of `A` below each other, and then  $n$  copies of this stacked array beside each other:

```
>> A
A =
     0.41027    0.057891    0.81317
     0.89365    0.35287    0.0098613
>> repmat(A,3,2)
ans =
     0.41027    0.057891    0.81317    0.41027    0.057891    0.81317
     0.89365    0.35287    0.0098613    0.89365    0.35287    0.0098613
     0.41027    0.057891    0.81317    0.41027    0.057891    0.81317
     0.89365    0.35287    0.0098613    0.89365    0.35287    0.0098613
     0.41027    0.057891    0.81317    0.41027    0.057891    0.81317
     0.89365    0.35287    0.0098613    0.89365    0.35287    0.0098613
```

## 2.12 Housekeeping

It is important to be able to administer both variables and commands in a computational program like Matlab. The functions in Table 2.5 are useful for housekeeping in Matlab.

The use of these commands is discussed in more detail in the following sections.

Table 2.5: Some commands for basic housekeeping in Matlab.

Command	Description
<code>who</code>	List the variables defined in Matlab's workspace.
<code>whos</code>	List variables and type/size in Matlab's workspace.
<code>clear</code>	Delete all variables from Matlab's workspace; <code>clear a b</code> clears <code>a</code> and <code>b</code> .
<code>↑</code> , <code>↓</code>	Use arrows to navigate in Matlab's list of previous commands.
<code>→</code> , <code>←</code>	Use arrows to navigate in the current command line for editing the command.
<code>cd</code>	Change directory.
<code>delete</code>	Delete a file.
<code>dir</code>	Show the files in the directory.
<code>what</code>	Show the <i>Matlab</i> files in the directory.
<code>mkdir</code>	Create a directory.

### 2.12.1 Variables

We have already seen that the variables that have been introduced, are listen in the *Workspace window* — see fig. 1.1. We have also indicated how we can use the Array Editor to edit the variables. Let us now see how we can address the variables from the command line.

In order to get a list of variables, we can use the commands `who` and `whos`. Command `who` basically lists the variables, while command `whos` also gives some details about the variables:

```
>> a = linspace(1,10,20);
>> b = zeros(1,20);
>> c = ones(1,20);
>> d = eye(1,20);
>> rand(1,20);
>> who
```

Your variables are:

```
a    ans  b    c    d
```

```
>> whos
Name      Size           Bytes  Class

a         1x20           160   double array
ans       1x20           160   double array
b         1x20           160   double array
c         1x20           160   double array
d         1x20           160   double array
```

Grand total is 100 elements using 800 bytes

Remember that each element in an array occupies 8 bytes of memory. Since we have  $5 \cdot (1 \times 20) = 100$  elements, this means  $100 \cdot 8 = 800$  bytes.

The variable with name `ans` always contains the latest result that has been computed.

If we want to delete a variable from memory, this is done using command `clear`. By simply writing the command `>>clear`, every variable in the workspace is deleted. If we want to specify a specific variable to delete, or a set of variables to delete, this is done as follows:

```
>> clear ans
>> clear a b
>> who
```

Your variables are:

```
c d
```

Notice that when listing variables that are to be cleared, these must be separated by *space* and not by comma. If we write `>>clear a, b`, this is interpreted as `clear a` and then display `b` in the command window.

We can even use regular expressions to clear variables:

```
>> A1 = rand(2,30);
>> A2 = rand(2,40);
>> who
```

Your variables are:

```
A1 A2 c d
```

```
>> clear A*
>> who
```

Your variables are:

```
c d
```

### 2.12.2 Matlab commands

As we have seen, by double-clicking on a command in Matlab's Command History window, the command is copied to the Command Window and executed.

Another way to use previous commands, is to step through the previous commands starting with the most recent command, by using arrow-up ( $\uparrow$ ) or arrow-down ( $\downarrow$ ) on the keyboard. The previous commands are stored in a history list, which is seen in the Command History window. When the desired command has been found by stepping up and down in the list using arrow-up or arrow-down, the command is executed by hitting the carriage return (CR) key. It is also possible to edit the commands that are called back to the command window using the arrows: use the mouse to mark possible characters to delete using either the **Delete** or the **Backspace** keys. If you want to add new text in the command, either use the mouse to position the cursor, or use arrow-left ( $\leftarrow$ ) or arrow-right ( $\rightarrow$ ) to position the cursor. Then type in the missing characters.

Quite often, the history list is lengthy, and one may have to hit the arrow-keys many times to find the correct command. If so, the following possibility is useful: type the first characters in the command that you want to re-use, and hit the arrow-up key ( $\uparrow$ ). This way, the most recent command of the history list starting with exactly those characters, will be found.

### 2.12.3 OS commands

Sometimes, it is useful to issue operating systems (OS) commands from within Matlab. This used to be achieved by preceding the command with the character `!`. In the latest version of Matlab, preceding a command by `!` and ending the command by `&`, opens a console window where OS operation can be performed on a command line. However, in the latest version of Matlab, there are built in Matlab commands for issuing the most common OS commands, such as `cd` (Change Directory), `delete` (DELETE file), `dir` (list content of DIRectory), `mkdir` (ReMove DIRectory), etc.

```
>> dir
```

```
.          eulersim.m          reaksim1.m
..         losandregrad.m      reaksim2.m
MatlabImportData.xls matlab.mat      reaksim3.m
dampfjaer.m myworkspaceA          slett
```

```

data.m                myworkspaceB        slett.mat
dfsim1.m             myworkspacem.mat   temp
dfsim2.m             myworkspacemA.mat  temp.mat
etwadata.m           myworkspacemB.mat
etwaeqsol.m          odexeu.m
etweq.m              reakdata.m

>> cd ..
>> dir

.                    Intro-kurs.ind      Matlab-course.dmp
..                   Intro-kurs.lof      Matlab-course.dvi
FHQOYR00.bmp         Intro-kurs.log      Matlab-course.idx
Figurer              Intro-kurs.lot      Matlab-course.ilg
HJ3FG000.wmf         Intro-kurs.tex      Matlab-course.ind
HJ3FG001.wmf         Intro-kurs.toc      Matlab-course.lof
Intro-kurs.aaa       IntroFig            Matlab-course.log
Intro-kurs.aab       Matlab              Matlab-course.lot
Intro-kurs.aux       Matlab-course.aaa  Matlab-course.tex
Intro-kurs.bak       Matlab-course.aab  Matlab-course.toc
Intro-kurs.bbl       Matlab-course.aut  figs
Intro-kurs.blg       Matlab-course.aux  sw0000
Intro-kurs.dvi       Matlab-course.bak
Intro-kurs.idx       Matlab-course.bbl
Intro-kurs.ilg       Matlab-course.blg

>> cd Matlab
>> dir

.                    eulersim.m         reaksim1.m
..                   losandregrad.m      reaksim2.m
MatlabImportData.xls matlab.mat          reaksim3.m
dampfjaer.m         myworkspaceA        slett
data.m               myworkspaceB        slett.mat
dfsim1.m             myworkspacem.mat   temp
dfsim2.m             myworkspacemA.mat  temp.mat
etwadata.m           myworkspacemB.mat
etwaeqsol.m          odexeu.m
etweq.m              reakdata.m

```

## 2.13 Basic functions

### 2.13.1 Overview of operations

We have already used the built-in Matlab commands `eye`, `zeros`, `ones`, `linspace`, `colon`, etc. There are a number of mathematical functions in Matlab, e.g. `sin`, `cosh`, `tan`, `exp`, etc., as well as other functions.

Typically, operations/functions are either unary or binary. Unary functions operate on a single object, while binary functions operate on two objects. The very fundamental binary operations for arrays are array addition, array subtraction, array multiplication, and array division: in Table 2.6, it is assumed that both  $A$  and  $B$  are arrays of the same size.

Let us next take a look at the mathematical (unary) functions. If we in Matlab let an array be the argument of a mathematical function, how is the result to be interpreted? A simple example illustrates this the best. If we consider applying the sinus function `sin` on the row array  $(x_1, x_2, \dots, x_n)$ , this is interpreted as the following row array:

$$\sin(x_1, x_2, \dots, x_n) = (\sin x_1, \sin x_2, \dots, \sin x_n):$$

Table 2.6: Basic binary array operations in Matlab. It is assumed that array  $A$  and  $B$  have the same shape.

Matlab operator	Matlab function	Description
$A+B$	<code>plus(A,B)</code>	Element by element addition, $A_{ij} + B_{ij}$ .
$A-B$	<code>minus(A,B)</code>	Element by element subtraction, $A_{ij} - B_{ij}$ .
$A.*B$	<code>times(A,B)</code>	Element by element multiplication, $A_{ij} \cdot B_{ij}$ .
$A./B$	<code>rdivide(A,B)</code>	Element by element division, $A_{ij}/B_{ij}$ .

```
>> x=0.3

x =

    0.3000

>> sin(x)

ans =

    0.2955

>> x=[0.3,0.7,0.18]

x =

    0.3000    0.7000    0.1800

>> sin(x)

ans =

    0.2955    0.6442    0.1790

>> x=[0.3;0.7;0.18];
>> sin(x)

ans =

    0.2955
    0.6442
    0.1790
```

In general, let  $\text{fun}(x)$  denote a general mathematical function of a scalar  $x$ . If we use an  $m \times n$  array  $A$  as argument to the Matlab function, we (normally) get:

$$\text{fun}(A) = \begin{pmatrix} \text{fun}(A_{1,1}) & \text{fun}(A_{1,2}) & \cdots & \text{fun}(A_{1,n}) \\ \text{fun}(A_{2,1}) & \text{fun}(A_{2,2}) & \cdots & \text{fun}(A_{2,n}) \\ \vdots & \vdots & \ddots & \vdots \\ \text{fun}(A_{m,1}) & \text{fun}(A_{m,2}) & \cdots & \text{fun}(A_{m,n}) \end{pmatrix}.$$

In other words: the answer is found by copying the input argument array, and then mapping the function to every element of the copied input argument. This is valid for most mathematical functions, e.g. when  $\text{fun}$  is  $\sin$ ,  $\cos$ ,  $\tan$ ,  $\sinh$ ,  $\cosh$ ,  $\sec$ ,  $\exp$ , etc. As an example:

```
>> A = [1,2,3;2,3,5]

A =
```



```

      1     2     3
      2     3     5

>> sinh(A)

ans =

    1.1752    3.6269   10.0179
    3.6269   10.0179   74.2032

```

where we have used the hyperbolic sine function `sinh`.

One more example: suppose we want to compute  $\sin x$  for a number of values  $x$  in the interval  $[0, 2\pi]$ . This can be achieved as follows:

```

>> x = linspace(0,2*pi,10)

x =

Columns 1 through 6

    0    0.6981    1.3963    2.0944    2.7925    3.4907

Columns 7 through 10

    4.1888    4.8869    5.5851    6.2832

>> y = sin(x)

y =

Columns 1 through 6

    0    0.6428    0.9848    0.8660    0.3420   -0.3420

Columns 7 through 10

   -0.8660   -0.9848   -0.6428   -0.0000

```

In the sequel, some basic Matlab functions are discussed. More functions can be found e.g. in (?) or in the Matlab documentation.

### 2.13.2 Trigonometric functions

A partial list of available trigonometric functions is shown in Table 2.7.

Examples of the use of these trigonometric functions are shown below:

```

>> rand('state',0)
>> A = rand(2,3)

A =

    0.9501    0.6068    0.8913
    0.2311    0.4860    0.7621

>> sin(A(2,2))

```

Table 2.7: Selected trigonometric functions in Matlab.

Matlab function	Math function	Description
<code>cos(x)</code>	$\cos x$	$x$ is given in radians.
<code>cosh(x)</code>	$\cosh x = (e^x + e^{-x})/2$	—” —
<code>sin(x)</code>	$\sin x$	—” —
<code>sinh(x)</code>	$\sinh x = (e^x - e^{-x})/2$	—” —
<code>tan(x)</code>	$\tan x = \sin x / \cos x$	—” —
<code>tanh(x)</code>	$\tanh x = \sinh x / \cosh x$	—” —
<code>acos(x)</code>	$\cos^{-1} x$	Result is given in radians.
<code>acosh(x)</code>	$\cosh^{-1} x$	—” —
<code>asin(x)</code>	$\sin^{-1} x$	—” —
<code>asinh(x)</code>	$\sinh^{-1} x$	—” —
<code>atan(x)</code>	$\tan^{-1} x$	—” —
<code>atanh(x)</code>	$\tanh^{-1} x$	—” —

```

ans =

    0.4671

>> sin(A)

ans =

    0.8135    0.5703    0.7779
    0.2291    0.4671    0.6904

>> asin(ans)

ans =

    0.9501    0.6068    0.8913
    0.2311    0.4860    0.7621

```

Notice that  $\sin(A_{2,2}) = (\sin A)_{2,2}$ . Also notice that when we first take the sinus of a number, and next take the inverse sinus (arcsine), we will not necessarily get back the original result — the answer may be in a different quadrant:

```

>> cos(-1)

ans =

    0.5403

>> acos(ans)

ans =

    1

```

### 2.13.3 Exponential functions

A partial list of available exponential functions is shown in Table 2.8.

Some examples of the use of these functions:

Table 2.8: Selected exponential functions in Matlab.

Matlab function	Math function	Description
<code>x.^p</code>	$x^p$	Array exponentiation.
<code>exp(x)</code>	$\exp x = e^x$	Exponentiation of $e$ .
<code>log(x)</code>	$\ln x = \log_e x = \exp^{-1} x$	Natural logarithm.
<code>log10(x)</code>	$\log_{10} x = \ln x / \ln 10$	Briggs'/logarithm with base 10.
<code>log2(x)</code>	$\log_2 x = \ln x / \ln 2$	Logarithm with base 2.
<code>pow2(x)</code>	$\log_2^{-1} x$	Inverse of $\log_2 x$ .
<code>sqrt(x)</code>	$\sqrt{x} = x^{1/2}$	Square root of $x$ .

Table 2.9: Selected complex functions in Matlab.

Matlab function	Math function	Description
<code>abs(z)</code>	$ z $	Module/absolute value/magnitude of $z$ .
<code>angle(z)</code>	$\angle z$	Phase angle in radians of $z$ .
<code>conj(z)</code>	$\bar{z}$	Complex conjugate of $z$ .
<code>imag(z)</code>	$\Im z$	Imaginary part of $z$ .
<code>real(z)</code>	$\Re z$	Real part of $z$ .
<code>isreal(z)</code>	—	True for real variables, false otherwise.
<code>cplxpair(z)</code>	—	Sort array into complex conjugate pairs.
<code>complex(x,y)</code>	$x + i \cdot y$	Form complex number $z = x + i \cdot y$ .
<code>z', ctranspose(z)</code>	$z^*$	Complex conjugate and transpose of array $z$ .

```
>> A(2,2)^2
```

```
ans =
```

```
0.2362
```

```
>> A.^2
```

```
ans =
```

```
0.9027    0.3683    0.7944
0.0534    0.2362    0.5808
```

Notice that for scalars  $s$ , we may write  $s^2$ . This doesn't work for arrays though, or rather for array  $A$ , writing  $A^2$  has a different meaning than  $A.^2$ . Operation  $A^2$  makes sense when the array is a matrix with certain properties, but not for general arrays. Operation  $A.^2$  on the other hand, makes sense for any array, and produces the result discussed in the introduction.

### 2.13.4 Complex functions

A partial list of available complex functions is shown in Table 2.9.

Some example of the use of these functions:

```
>> B = A-eye(size(A))
```

```
B =
```

```
-0.0499    0.6068    0.8913
0.2311   -0.5140    0.7621
```

```
>> C = sqrt(B)
```

```

C =
    0 + 0.2233i    0.7790    0.9441
    0.4808        0 + 0.7170i    0.8730

>> abs(C)

ans =

    0.2233    0.7790    0.9441
    0.4808    0.7170    0.8730

>> angle(C)

ans =

    1.5708    0    0
    0    1.5708    0

>> conj(C)

ans =

    0 - 0.2233i    0.7790    0.9441
    0.4808        0 - 0.7170i    0.8730

>> imag(C)

ans =

    0.2233    0    0
    0    0.7170    0

>> real(C)

ans =

    0    0.7790    0.9441
    0.4808    0    0.8730

>> isreal(C)

ans =

    0

>> C'

ans =

    0 - 0.2233i    0.4808
    0.7790        0 - 0.7170i
    0.9441        0.8730

```

### 2.13.5 Rounding and remainder functions

A list of rounding and remainder functions is shown in Table 2.10.

Table 2.10: Selected rounding and remainder functions in Matlab.

Matlab function	Math function	Description
<code>fix(x)</code>	$\lfloor  x  \rfloor \cdot \text{sign } x$	Round toward zero.
<code>floor(x)</code>	$\lfloor x \rfloor$	Round toward $-\infty$ .
<code>ceil(x)</code>	$\lceil x \rceil$	Round toward $+\infty$ .
<code>round(x)</code>	$\lfloor x \rfloor$	Round toward nearest integer.
<code>mod(x,y)</code>	$\text{mod}(x,y) \cdot \text{sign } y$	Modulus or signed remainder.
<code>rem(x,y)</code>	$\text{mod}(x,y) \cdot \text{sign } x$	Remainder after division.
<code>sign(x)</code>	$\text{sign } x$	Signum function.

Some examples of how to use these functions:

```
>> D = 10*B
```

```
D =
```

```
   -0.4987    6.0684    8.9130
    2.3114   -5.1402    7.6210
```

```
>> fix(D)
```

```
ans =
```

```
    0     6     8
    2    -5     7
```

```
>> floor(D)
```

```
ans =
```

```
   -1     6     8
    2    -6     7
```

```
>> ceil(D)
```

```
ans =
```

```
    0     7     9
    3    -5     8
```

```
>> round(D)
```

```
ans =
```

```
    0     6     9
    2    -5     8
```

```
>> sign(D)
```

```
ans =
```

```
   -1     1     1
    1    -1     1
```

Here, function `round` is probably the one that is most useful.

Modulus and remainders are computed as follows:

Table 2.11: Selected number theoretic functions in Matlab.

Matlab function	Description
<code>factor(N)</code>	Prime factors of integer $N$ .
<code>isprime</code>	True for prime numbers, false otherwise.
<code>primes(N)</code>	Generate list of prime numbers $\leq N$ .
<code>gcd(M,N)</code>	Greatest common divisor of integers $M$ and $N$ .
<code>lcm(M,N)</code>	Least common multiple of integers $M$ and $N$ .
<code>rats(x)</code>	Rational approximation of floating point number $x$ .

```
>> mod(D,2)
```

```
ans =
```

```
    1.5013    0.0684    0.9130
    0.3114    0.8598    1.6210
```

```
>> rem(D,2)
```

```
ans =
```

```
   -0.4987    0.0684    0.9130
    0.3114   -1.1402    1.6210
```

### 2.13.6 Number theoretic functions\*

A partial list of number theoretic functions is shown in Table 2.11.

Prime factors are found as follows:

```
>> N = 133
```

```
N =
```

```
    133
```

```
>> factor(N)
```

```
ans =
```

```
     7     19
```

Notice that the answer is given as elements in a row array. Is e.g. 19 a prime number?

```
>> isprime(ans(2))
```

```
ans =
```

```
     1
```

```
>> isprime(N)
```

```
ans =
```

```
     0
```

What are the prime numbers smaller than 133?

```
>> primes(N)
```

```
ans =
```

```
Columns 1 through 11
```

```
    2    3    5    7   11   13   17   19   23   29   31
```

```
Columns 12 through 22
```

```
   37   41   43   47   53   59   61   67   71   73   79
```

```
Columns 23 through 32
```

```
   83   89   97  101  103  107  109  113  127  131
```

What is a good rational approximation of  $\pi$ ?

```
>> rats(pi)
```

```
ans =
```

```
    355/113
```

Suppose we want to simplify 144/244? We compute the greatest common divisor:

```
>> gcd(144,244)
```

```
ans =
```

```
    4
```

Thus,  $\frac{144}{4}/\frac{244}{4} = 36/61$  is the simplest equivalent expression where both numerator and denominator are integers. Because 4 is the GCD of (144, 244), we know that 144/4 and 244/4 are integer numbers.

Suppose we want to add  $1/12 + 1/16$ . What is the least common multiple of 12 and 16?

```
>> lcm(12,16)
```

```
ans =
```

```
   48
```

Thus, we have  $1/12 + 1/16 = \frac{48}{12}/48 + \frac{48}{16}/48 = 4/48 + 3/48 = (4 + 3)/48 = 7/48$ . Because 48 is the LCM of (12, 16), we know that 48/12 and 48/16 are integer numbers.

## 2.14 Help

It is a common problem that Matlab has so many functions, that it is impossible to keep track of how each of them work. And sometimes, we don't even know whether a certain function is available.

If we know the name of a certain function, and want to find out how to use it, we can either type `>>help <functionname>`, `>>helpwin <functionname>`, or `>>doc <functionname>` to get a description of how to use the function:

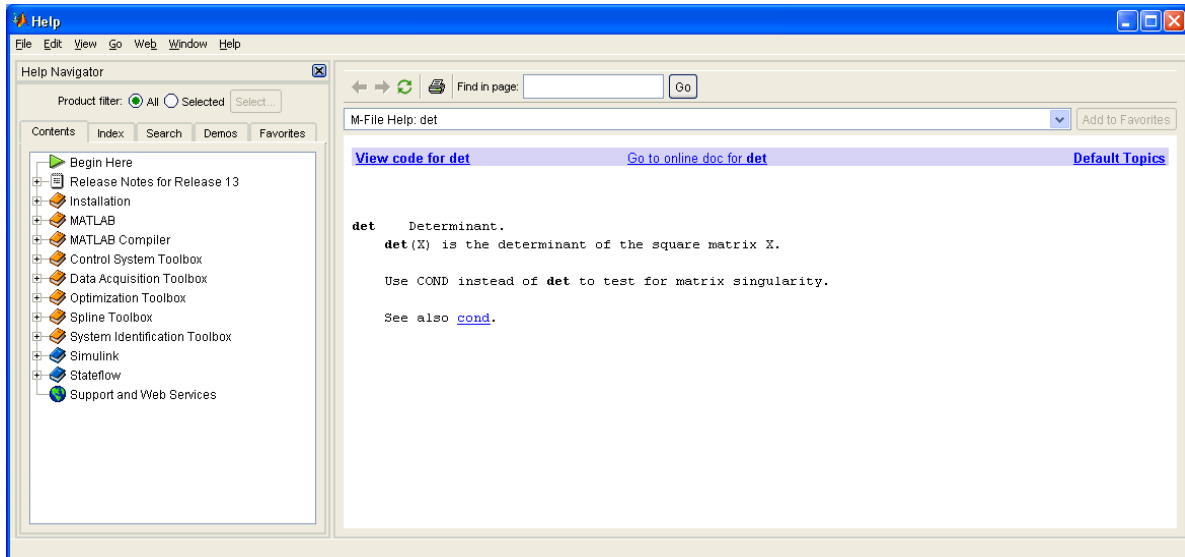


Figure 2.9: The help browser window which opens when using the `doc` command in Matlab's command window, or by choosing `Help/MATLAB Help` in the Matlab window.

```
>> help det
DET    Determinant.
      DET(X) is the determinant of the square matrix X.

      Use COND instead of DET to test for matrix singularity.

      See also cond.

Reference page in Help browser
      doc det.
```

Using `doc` instead of `help` (e.g. `doc det`) opens up a help browser in a separate window<sup>10</sup>, fig. 2.9. The `help` browser can also be used to search for keywords, etc., when one does not know the exact name of a function. The `help` browser can also be opened from the `HELP` menu of the Matlab window, by selecting `MATLAB Help`. From this help window, it is also possible to search the world wide web for information about Matlab functionality.

<sup>10</sup> As an alternative to command `doc`, command `helpwin` opens a help window. When the Matlab documentation files are installed, command `helpwin` appears to give the same response as command `doc`.



# Chapter 3

## Basic plotting in Matlab

### 3.1 Overview of learning goals

In order to study data from experiments, or results from computations, it is often useful to present the results graphically in plots. Matlab has very good support for plotting. Here, we will mainly go through the possibilities in two-dimensional plotting. Good references for plotting in Matlab are (Higham & Higham 2000) for a brief overview, and (Hanselman & Littlefield 2005) for more details.

After having completed this chapter, you should have a clear understanding of what a figure is, and how you can adjust properties of the figure interactively in Matlab. In particular, you should master:

- How to produce two-dimensional plots and how to modify plots,
- How to create arrays of plots,
- Presentation of experimental data, 3.5.

The description will indicate how you use the Matlab commands of Table 3.1.

### 3.2 Basic two dimensional plots

#### 3.2.1 The plot function

The basic plot command is `plot`:

```
>> x=linspace(0,2*pi,20);  
>> y = sin(x);  
>> plot(y)
```

`plot` is a built-in Matlab function. The result of this command is shown in fig. 3.1, where the `plot` function has been used with a single argument. Then, the elements of the (row or column array) argument is displayed as a function of the element number: because array `x` has 20 elements, array `y` has 20 elements, too. When using the `plot` command, Matlab draws straight lines between the data points. Thus, the result is a graph with 19 straight line segments between 20 data points:  $(1, 0), (2, \pi/10), \dots, (20, 2\pi)$ , and the *abscissa* (“*x*-axis”) ranges from 1 to 20.

Table 3.1: Basic plotting commands.

Matlab command	Description	Reference
<code>plot</code>	Plot 2D plots	p. 43
---	Interactive modification of plots	p. 44
<code>hold on / hold off</code>	Hold/un-hold plots	p. 46
<code>legend</code>	Insert legend	p. 47
---	L <sup>A</sup> T <sub>E</sub> X typesetting in plots	p. 48
<code>subplot</code>	Create array plots	p. 51

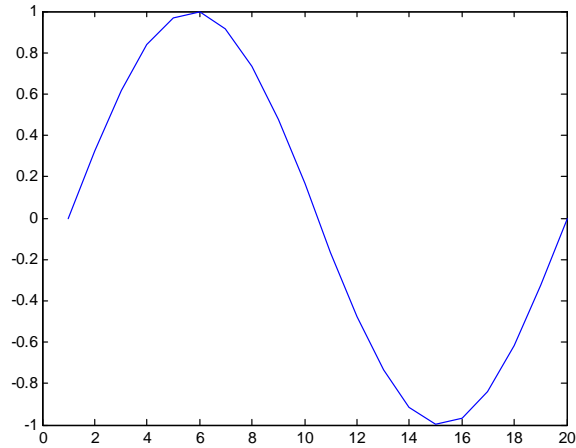


Figure 3.1: Function  $y = \sin x$  plotted using command `plot(y)`.

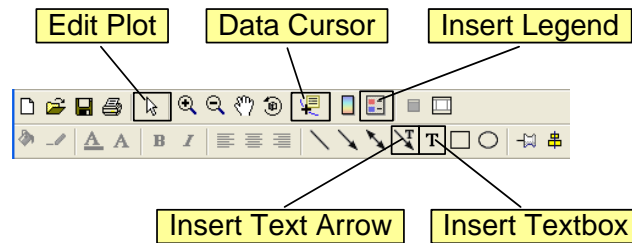


Figure 3.2: Figure Toolbar (upper row) and Plot Edit Toolbar (lower row). Select which toolbar to see from the **View** menu of the figure window.

### 3.2.2 Editing plots

In order to modify a plot, the Figure Toolbar and the Plot Edit Toolbar are useful, fig. 3.2. The Figure Toolbar is displayed by default, while the Plot Edit Toolbar is toggled on in the **View** menu of the figure window.

In order to edit a plotted graph, click on the Edit Plot icon (fig. 3.2), and then click on the curve to select the curve. Table 3.2 illustrates how we can modify some properties of the graph interactively.

Usually, we do not want to plot  $y$  as a function of the element number, but rather to plot  $y$  as a function of  $x$ :

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

The result is shown in fig. 3.3. Notice that the abscissa ranges from 0 to  $6.28 \approx 2\pi$ .

### 3.2.3 Multiple plots

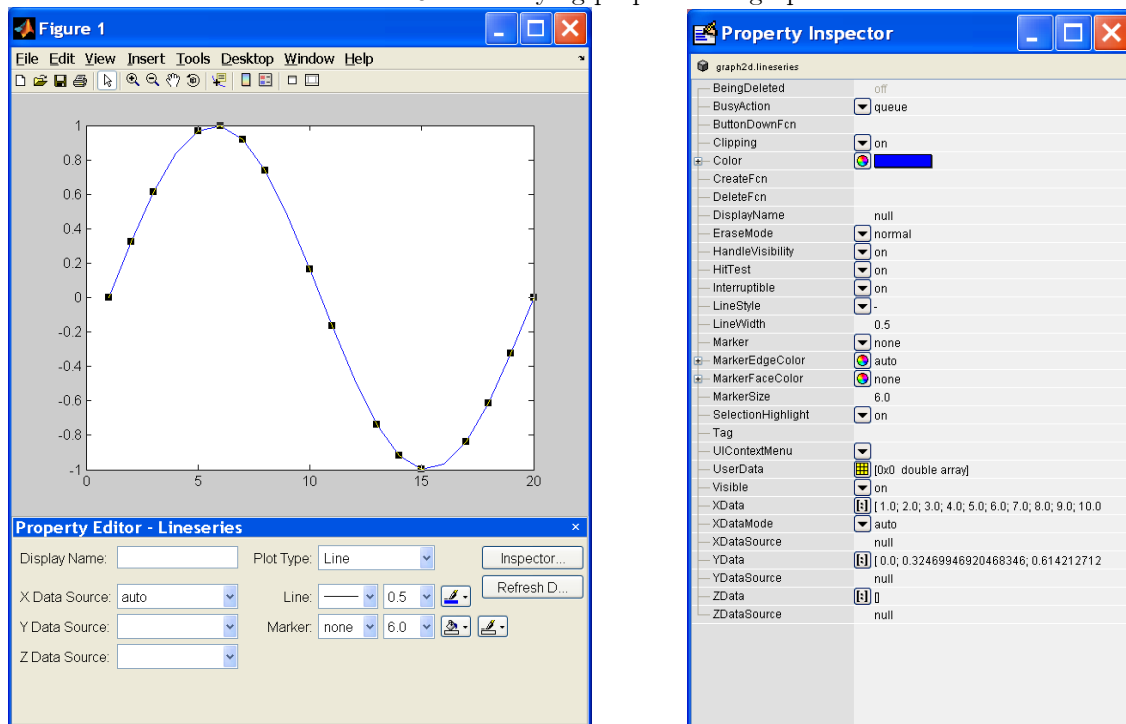
We can plot several functions in the same plot window:

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

The result is displayed in fig. 3.4.

**Remark 3.1** In order to prepare a plot for printing, it is *advisable* to use black color on each graph, and to distinguish among several graphs by varying the line type (solid, dotted, dashed, etc.). ■

Table 3.2: Mofding properties of graphs.



(a) Click **Edit Tool** (fig. 3.2), then click graph, right-click, and choose **Properties...**

(b) Click on **Inspector...** in Proper Editor (table element (a)).

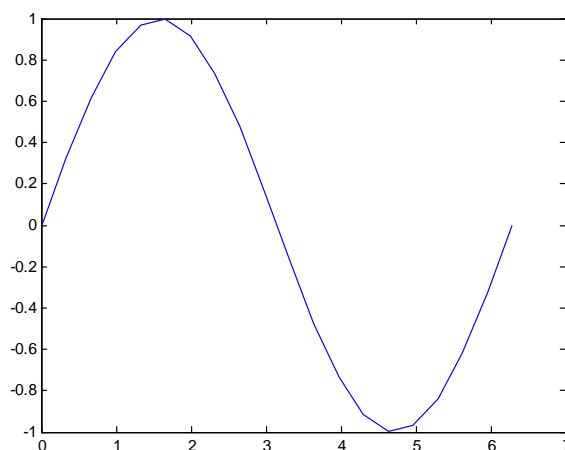


Figure 3.3: Function  $y = \sin x$ , plotted using command `plot(x,y)`.

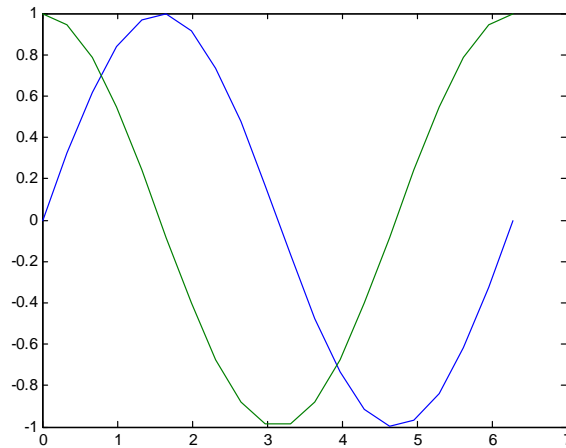


Figure 3.4: Function  $y = \sin x$  and function  $\cos x$  plotted using command `plot(x,y,x,cos(x))`.

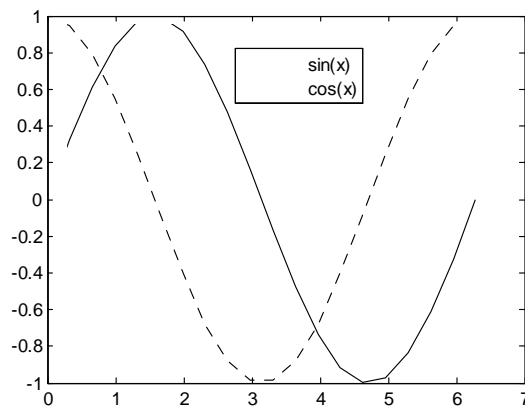


Figure 3.5: Example of edited line styles and line colors. To insert legend, click on the Insert Legend icon (fig. 3.2). To change the default legend text, double-click on the text and edit it.

By following the recipe in Table 3.2, the line styles and line colors of fig. 3.4 have been changed, and legends have been inserted, see fig. 3.5.

It is also possible to *add plots* to a figure window. By simply typing a new plot command:

```
>> plot(x,exp(-x))
```

the previous plot is deleted, and the new one is drawn. That is not what we want. To make sure that we preserve the original graphs in the figure window, we need to issue a command to **hold** the current graphs in the figure window:

```
>> plot(x,y,x,cos(x))
>> hold on
>> plot(x,exp(-x))
>> hold off
```

The result is shown in fig. 3.6.

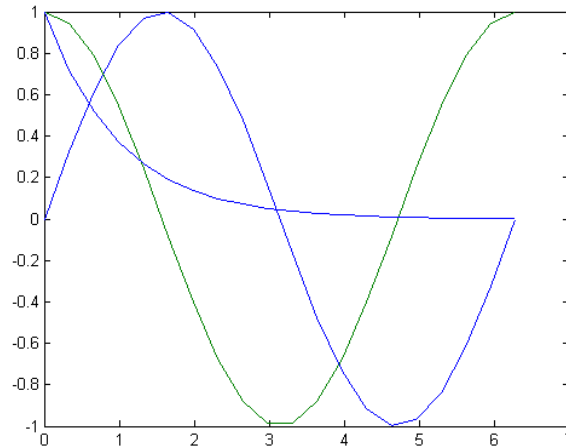


Figure 3.6: The graph of  $\exp(-x)$  has been added to the graphs of  $\sin x$  and  $\cos x$ .

Another way to plot several graphs in one figure, is to let the second argument be a two dimensional array. Either the number of rows, or the number of columns of the second argument, must be equal to the length of the first argument:

```
>> plot(x, [y; cos(x); exp(-x)])
>> plot(x', [y', cos(x)', exp(-x)'])
```

The command lines above give the same result; we could also have used the following command:

```
>> z = [y; cos(x); exp(-x)];
>> plot(x,z)
```

### 3.2.4 Editing axes properties

So far, we have seen how to change the properties of individual graphs. It is also possible to change the properties that are shared by all graphs. Table 3.3 shows how to select the so-called Axes Properties: Some of the Axes properties can only be used with three dimensional plots. Labels, etc. can be typeset using  $\text{\TeX}$  and  $\text{\LaTeX}$  commands.

### 3.2.5 Command line plot editing

All the settings for Graph and Axes properties, can also be set on the command line. This is very useful if we need to create a script for producing many figures overnight.

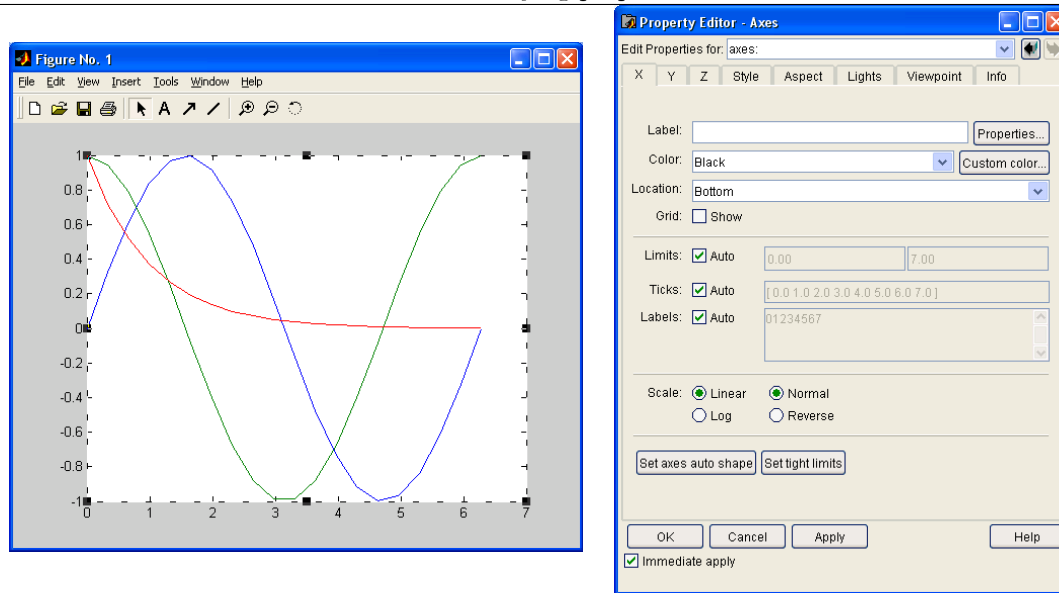
In the simplest case, we can produce the plot in fig. 3.5 as follows:

```
>> x = linspace(0,2*pi,20);
>> plot(x,sin(x),'k-',x,cos(x),'k--')
>> legend('sin(x)', 'cos(x)')
```

The only necessary interactive modification, is to use the cursor to move the legend box. In the `plot` statement above, the first character of the text strings indicate black color. For other colors and line styles, as well as markers, see `doc plot`, and click on the `LineStyle` hyperlink.

For more advanced modifications, it is possible to take advantage of the Handle Graphics data structure, see Chapter 12. A number of Matlab books contain more information, e.g. (Hanselman & Littlefield 2005).

Table 3.3: Modifying properties of axes.



(a) Click on the axes.

(b) Right-click, and choose **Properties...**

### 3.2.6 Fancy typesetting in plots\*

The basic typesetting in Matlab plots is rather simple. To achieve a more professional typesetting, it is possible to take advantage of  $\text{T}_{\text{E}}\text{X}$  and  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$  typesetting (which is used in this document). For an introduction to  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$ , see e.g. (Lamport 1986).

Table 3.4 illustrates some possibilities for  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$  typesetting in Matlab plots. The legend text is modified as indicated in the two upper plots of Table 3.4. By selecting the legend box, right-clicking, and selecting **Interpreter/latex**, the result in the lower left plot of Table 3.4 is achieved.

In  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$  typesetting, mathematical expressions are bracketed by the  $\$$  character, or by the  $\$\$$  symbol: using  $\$$  bracketing, indicates in-line math such as  $\sin(x)$  and  $\int_0^{2\pi} \sin(x) dx$ , while  $\$\$$  bracketing indicates displayed math such as

$$\sin(x)$$

$$\int_0^{2\pi} \sin(x) dx.$$

In Table 3.4, the lower left plot doesn't really illustrate the difference between in-line math and displayed math. To demonstrate the difference in a Matlab plot, we replace the  $\$\cos(x)\$$  and the  $\$\exp(-x)\$$  legend strings by  $\$\int_0^{2\pi} \sin(x) dx\$$  and  $\$\$\int_0^{2\pi} \sin(x) dx\$\$$ , respectively. When using the  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$  interpreter, the result is as shown in the lower right plot of Table 3.4.

Inserting typeset legends can also be done from the command line. For the legend in the lower right plot of Table 3.4, the following statement can be used:

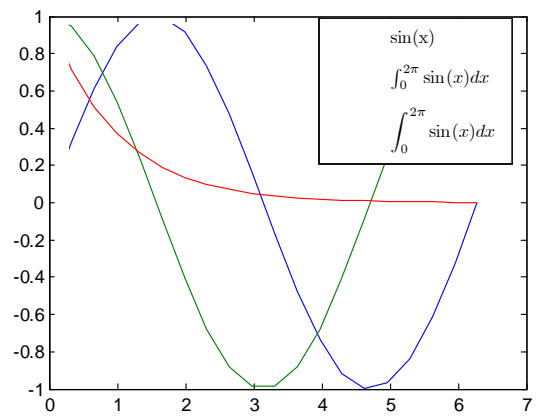
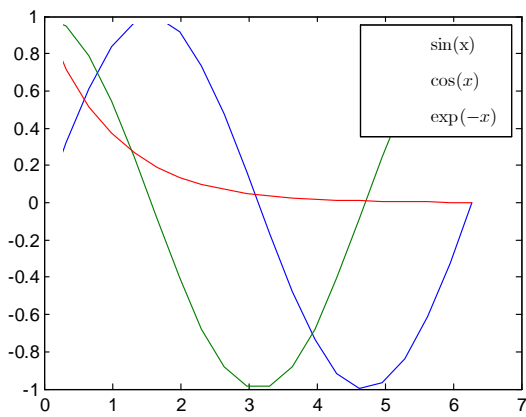
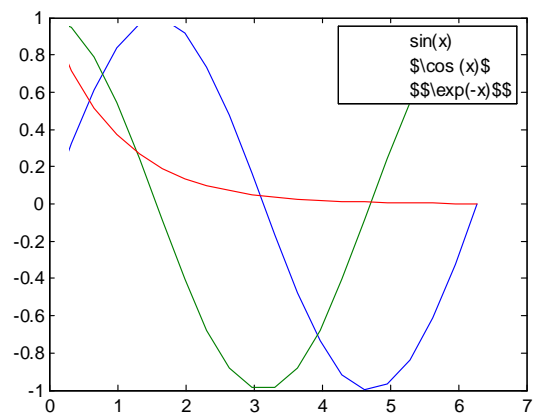
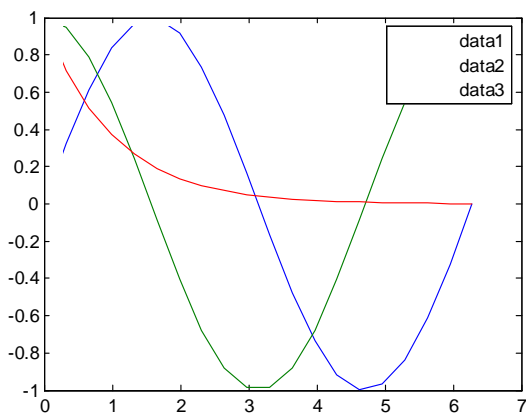
```
>> legend({'$\sin(x)$', '$\int_0^{2\pi}\sin(x)dx$', ...
          '$$\int_0^{2\pi}\sin(x)dx\$\$'}, 'Interpreter', 'LaTeX')
```

### 3.2.7 Closing

For more information about  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$  typesetting, see the Help browser, e.g. (Hanselman & Littlefield 2005), or any  $\text{L}_{\text{A}}\text{T}_{\text{E}}\text{X}$  book such as (Lamport 1986).

**Exercise 3.1** Describe what the following functions can do: `hold`, `axes`, `xlabel`, `ylabel`, `title`, `gtext`, `logspace`, `format`. What are the arguments of these functions? ■

Table 3.4: Illustration of using  $\TeX$  and  $\LaTeX$  typesetting in plots.



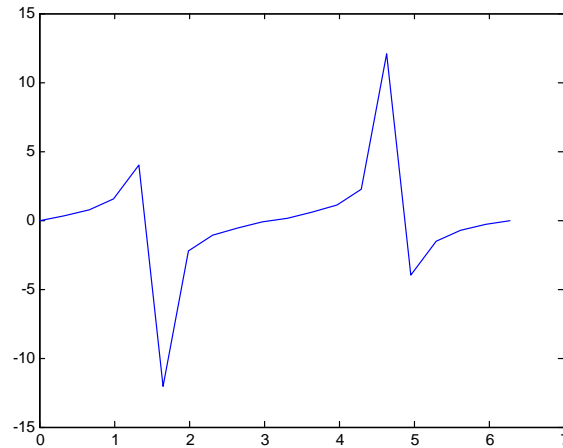


Figure 3.7: Function  $y = \tan x$ , plotted using command `plot(x, tan(x))`.

**Exercise 3.2** Check out the help files for the following Matlab 2D functions: `plot`, `loglog`, `semilogx`, `semilogy`, `plotyy`, `polar`, `fplot`, `fill`, `area`, `bar`, `barh`, `hist`, `pie`, `comet`, `errorbar`, `quiver`, `scatter`. ■

### 3.3 Improving plots

In some cases, the plots become “ugly”/erroneous — this is typically the case where the ordinate value goes to  $\pm\infty$  at some value of the abscissa, e.g. the function  $y = \tan x$ :

```
>> plot(x, tan(x))
```

The result is shown in fig. 3.7. The result is obviously incorrect: We know that there are asymptotes at  $x = \frac{\pi}{2}i$ ,  $i \in \{1, 3, 5, \dots\}$ , but the plotting algorithm draws straight lines between every point. We could wish to clip the function value e.g. at  $y = \pm 3$ , but in such a way that if  $|\tan x| \geq 3$ , nothing is plotted. We can achieve this as follows:

1. Find the indices for vector  $\tan x$  where  $|\tan x| \geq 3$ .
2. Replace the functional value of  $\tan x$  for the given indices ( $|\tan x| \geq 3$ ) with the value `NaN` (Not a Number). If the symbol `NaN` is found in an array that Matlab is asked to plot, no lines are drawn to this point.

The following Matlab sequence illustrates how this can be done:

```
>> y = tan(x);
>> ind = find(abs(y) >= 3)

ind =

     5     6    15    16

>> y(ind) = NaN

y =

Columns 1 through 7
```



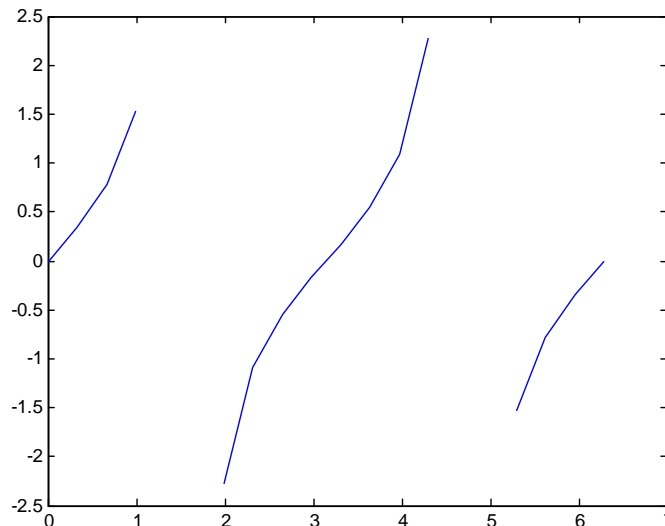


Figure 3.8: The function  $y = \tan x$ , plotted using command `plot(x,y)`, but where the  $y$  values greater than 3 in absolute value have been replaced by NaN (Not a Number).

```

0      0.3433    0.7783    1.5306      NaN      NaN    -2.2798

Columns 8 through 14
-1.0863   -0.5412   -0.1669    0.1669    0.5412    1.0863    2.2798

Columns 15 through 20
NaN      NaN    -1.5306   -0.7783   -0.3433   -0.0000

>> plot(x,y)

```

The result is shown in fig. 3.8. In fig. 3.8 we can clearly see the indication of asymptotes at  $x = \frac{\pi}{2}$  and at  $x = \frac{3\pi}{2}$ .

### 3.4 Array plots

In addition to plotting several graphs in a single axis system, it is possible to put graphs in an array of axes. The plots are considered as elements in a two dimensional array, and the command `subplot` is used to define which element the plot is put into. The the command has three integer arguments, `subplot(m,n,p)`, and this command is followed by e.g. the `plot` command which actually inserts the plot. In the `subplot` command, the two first arguments (`m`, `n`) indicate that the plot is to be inserted into an  $m \times n$  array. The third argument (`p`) specifies in which element of the array the plot will be placed — counting row-wise.

Suppose we want to produce an array with 1 column and 3 rows: the first element is specified as `subplot(3,1,1)`, the second element as `subplot(3,1,2)`, etc. The two first arguments define the size of the array, and the third argument specifies the location where the next plot should be put.

The following example illustrates the procedure:

```

>> subplot(1,3,1)
>> plot(x,y)

```

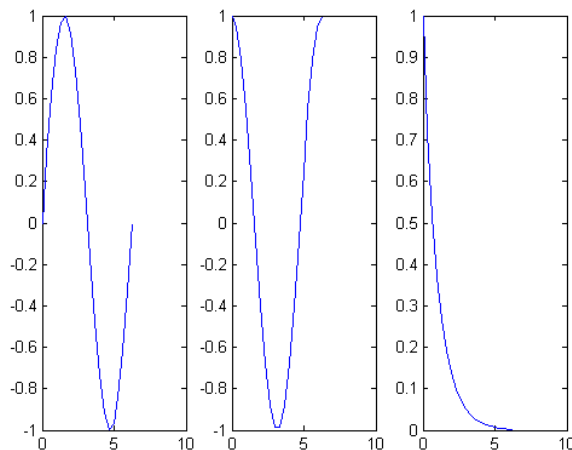


Figure 3.9: The use of the `subplot` command to produce an array of plots.

```
>> subplot(1,3,2)
>> plot(x,cos(x))
>> subplot(1,3,3);
>> plot(x,exp(-x))
```

This produces the plots in fig. 3.9.

It is possible to produce more complicated array plots if we “cheat” and mix the array division in the plot. The following example illustrates this:

```
>> subplot(2,2,1)
>> plot(x,y)
>> subplot(2,2,2)
>> plot(x,cos(x))
>> subplot(2,1,2)
>> plot(x,exp(-x))
```

Here, we have mixed freely between one plot array of size (2, 2), and another of size (2, 1). The result is displayed in fig. 3.10.

## 3.5 Presentation of experimental data

### 3.5.1 Case study: distillation of water and ethanol

For liquid mixtures of two components (e.g. water and ethanol), the unit operation of distillation can be used to separate the two components into their pure constituents. The distillation process takes advantage of the fact that the two components have different boiling points, i.e. by heating up the mixture, the component with the lowest boiling point will evaporate first. Thus, in principle, the vapor phase will contain the component with the lowest boiling point. In reality, part of the other component will also be found in the vapor phase.

In industrial practice, the mixture is continuously fed to a mixing vessel (feed stream  $F$ ), where the mixture is heated to the boiling point, fig. 3.11. The resulting vapor leaves at the top of the mixing vessel (vapor stream  $V$ ), while the (boiling) liquid leaves at the bottom of the vessel (liquid stream  $L$ ). Usually, one assumes that the vapor and liquid are at thermodynamic equilibrium. It is customary to measure the content of “light” component (i.e., the one with the lowest boiling point, e.g. ethanol in a water-ethanol mixture) as the mole fraction of light component, i.e. the number of moles of light component divided by the total number of moles in the mixture.  $\dot{Q}$  indicates the added heat.

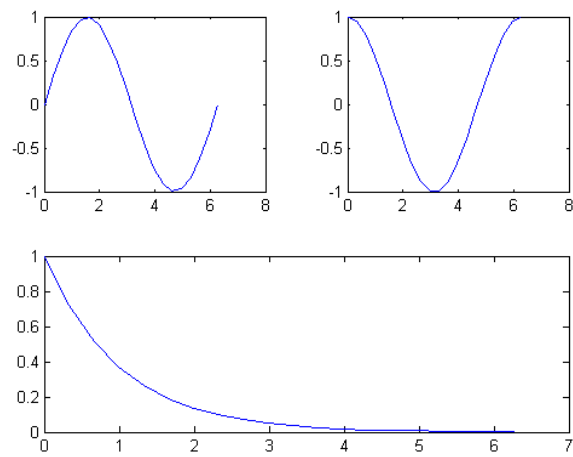


Figure 3.10: The use of the `subplot` command to produce an array of plots, where there is a mixture of array sizes.

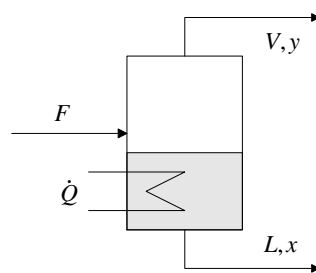


Figure 3.11: Sketch of a simple distillation unit, with feed stream ( $F$ ), bottom liquid stream ( $L$ ) with mole fraction  $x$  of light component, and top vapor stream ( $V$ ) with mole fraction  $y$  of light component.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1		Equilibrium data for water/ethanol: (x,y) are mole fractions of ethanol in the liquid (x) and vapor (y)														
2	x	0.0000	0.0714	0.1429	0.2143	0.2857	0.3571	0.4286	0.5000	0.5714	0.6429	0.7143	0.7857	0.8571	0.9286	1.0000
3	y	0.0000	0.3777	0.4913	0.5462	0.5814	0.6092	0.6353	0.6623	0.6921	0.7259	0.7648	0.8100	0.8631	0.9256	1.0000

Figure 3.12: Equilibrium data for water and ethanol, as found from experiments:  $(x, y)$  are mole fractions of liquid and vapor at equilibrium.

Through experiments with continuous distillation of a mixture of water and ethanol, assume that we have found the following correlation between the ethanol content of the liquid ( $x$ ) and the ethanol content of the vapor phase ( $y$ ), fig. 3.12.

### 3.5.2 Plotting data

The equilibrium data are stored in an Excel file. First we import the data into Matlab, using `File/Import Data...` in Matlab. The import filter interprets the first row of the spreadsheet (see fig. 3.12) as NaN, and produces a variable `data` consisting of 3 rows and 15 columns:

```
>> data
```

```
data =
```

```
Columns 1 through 8
```

NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
0.0000	0.0714	0.1429	0.2143	0.2857	0.3571	0.4286	0.5000
0.0000	0.3777	0.4913	0.5462	0.5814	0.6092	0.6353	0.6623

```
Columns 9 through 15
```

NaN	NaN	NaN	NaN	NaN	NaN	NaN
0.5714	0.6429	0.7143	0.7857	0.8571	0.9286	1.0000
0.6921	0.7259	0.7648	0.8100	0.8631	0.9256	1.0000

Clearly, the second row contains the  $x$ -data, while the third row contains the  $y$ -data:

```
>> x = data(2,:);
>> y = data(3,:);
>> plot(x,y,'k-', x,x,'k:');
>> xlabel('x')
>> ylabel('y')
>> title('Equilibrium data for water-ethanol mixture')
```

The data are displayed in fig. 3.13. Clearly, the vapor phase is richer in ethanol than the liquid phase as long as  $y(x) > x$ .

There is a problem with the presentation in fig. 3.13: Matlab has drawn straight lines between each of the 15 data points. This is *not considered a proper presentation* of experimental data, since it may give the *false impression* that we actually know something about the value of  $y$  for other values of  $x$  than the experimental values. It is thus common to present experimental data as isolated data points:

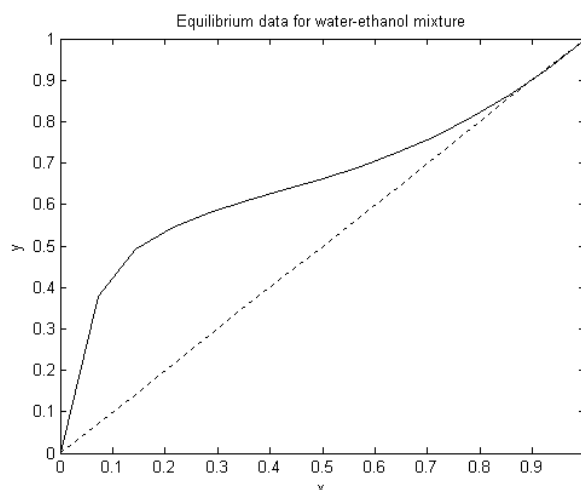


Figure 3.13: The equilibrium correlation for content of ethanol in the liquid phase ( $x$ ) and in the vapor phase ( $y$ ).

Table 3.5: Basic plotting commands.

Matlab command	Description	Reference
<code>meshgrid</code>	Create array argument for 3D plot	p. 167
<code>mesh / meshc</code>	Basic 3D plotting of surfaces	p. 167
<code>contour</code>	Contour plot	p. 169
<code>clf</code>	Clear figure	p. 171
<code>figure</code>	open figure	p. 171
<code>saveas</code>	save plot	p. 171
<code>open</code>	open saved plot	p. 171
<code>close</code>	close figure window	p. 171

```
>> plot(x,y,'kx', x,x,'k:')
>> xlabel('x')
>> ylabel('y')
>> title('Experimental equilibrium data for water-ethanol mixture')
>> legend('Experimental data: y(x)', 'y = x')
```

The result is shown in fig. 3.14.

In fig. 3.14, we do not make any assumptions about the values of  $y(x)$  outside of the experimental data.

## 3.6 Further study

Appendix D contains a discussion of more “advanced” topics such as

- How to produce three-dimensional plots, see accompanying Appendix D,
- Basic housekeeping of plots, see accompanying Appendix D,
- etc.

Some commands that are treated in Appendix D, are listed in Table 3.5.

Other useful plots not treated directly, are `semilogx`, `semilogy`, `loglog`, `plot3`, `contourf`, `contour3`, `meshz`, `surf`, `surfc`, `waterfall`, `bar3`, `bar3h`, `pie`, `fill3`, `comet3`, `scatter3`, `stem3` — see the Matlab help system. Yet other useful commands are `axis`, `title`, `xlabel`, `ylabel` — see help system as well as Section 3.5.

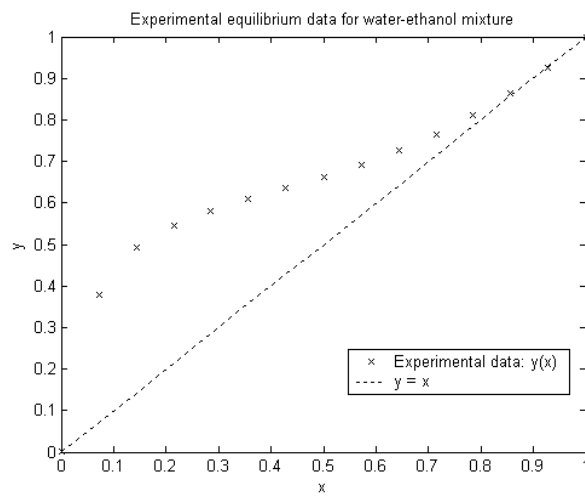


Figure 3.14: Experimentally found equilibrium correlation for content of ethanol in the liquid phase ( $x$ ) and in the vapor phase ( $y$ ).

# Chapter 4

## Simple data analysis\*

### 4.1 Overview of learning goals

After having completed this chapter, you should have a clear understanding of what is meant by interpolation vs. data fitting, what are some advantages and disadvantages of standard interpolation models and fitted models, and basic use of these models. In particular, you should master:

- How to use functions for polynomial and other types of interpolation,
- How polynomials are used in Matlab, basic functions for operating on polynomials, and how these functions can be used to analyze models,
- How to build monovariate and multivariate interpolation models in Matlab.

In addition to using plotting functions that we studied in Chapter 3, see also Table 3.1 p. 43, we will use some additional functions to analyze the data, see Table 4.1.

In addition, Matlab has some very rudimentary functions for data analysis such as `max`, `min`, `sum`, `mean`, `std`, `var`. See the Matlab help system for the rudimentary data analysis tools.

### 4.2 Interpolation

#### 4.2.1 Overview

In interpolation, we create a mathematical model that goes through every experimental data point. Interpolation is used to compute values of  $y(x)$  for  $x$ -values for which we do not have experimental values, i.e. in between the experimental values. The structure of such models is typically:

$$y_m(x) = \sum_{i=1}^n c_i \phi_i(x)$$

where  $\phi_i(x)$  is a chosen function, and  $c_i$  is a constant, but the models can also be more complex. Typically in interpolation, an implicit assumption is that the experimental results are perfect.

Table 4.1: Table Caption

Matlab command	Description	Reference
<code>polyfit</code>	Single-input polynomial model fit	p. 58
<code>polyval</code>	Evaluate polynomial model	p. 58
<code>spline</code>	Build spline single-input interpolation	p. 60
<code>ppval</code>	Evaluate interpolation models	p. 60
<code>pchip</code>	Build Hermitian cubic polynomials single-input interpolation	p. 60
<code>interp1</code>	Build various types of interpolation models	p. 61
---	polynomial functions	Table 4.3 p. 64
<code>interp2</code>	Build various two-input interpolation models	Section 4.4.2

Various interpolation models have different sets of functions  $\phi_i(x)$ , which may be termed basis functions or trial functions. Depending on the chosen basis functions, the resulting interpolation models will have different properties (e.g. smoothness, etc.), and will predict different values  $y_m(x)$  in between experimental data points.

The simple interpolation strategy of drawing straight lines between data points, is achieved by selecting  $\phi_i(x)$  to be so-called first order spline basis functions. Another possibility is to use polynomial models, e.g.

$$y_m(x) = \sum_{i=1}^n c_i x^{i-1},$$

i.e. we have chosen  $\phi_i(x) = x^{i-1}$ . Sometimes, we want the interpolation formulae to be differentiable at every  $x$ ; obviously, we can not achieve this if we use first order spline basis functions.

In the study, we use the experimental data from Section 3.5.

## 4.2.2 Polynomial interpolation

Let us experiment with polynomial interpolation. In order to find the interpolation formulae, we use the Matlab function `polyfit(x,y,n)`, where  $x$  is the input variable,  $y$  is the output variable, and  $n$  is the order of the polynomial. Note that with  $N = 15$  data points, it is necessary to use a model with 15 parameters  $c_i$ , i.e. to build a 14-th order polynomial in order to have true interpolation, i.e. that the model is  $y_m(x) = \sum_{i=1}^{15} c_i x^{i-1}$ . If we choose  $n < 15$ , then the model will not go through every data point. If we choose  $n > 15$ , then it is not possible to find unique values  $c_1$ .

```
>> c = polyfit(x,y,14);
Warning: Polynomial is badly conditioned. Remove repeated data points
        or try centering and scaling as described in HELP POLYFIT.
(Type "warning off MATLAB:polyfit:RepeatedPointsOrRescale" to suppress this warning.)
> In C:\MATLAB6p5\toolbox\matlab\polyfun\polyfit.m at line 75
```

For now, we do not worry about the Matlab Warning.

If we want to plot the result, we do this by using command `polyval(c,x)`:

```
>> ym = polyval(c,x);
```

Let us see how the interpolation function fits the data:

```
>> plot(x,y,'kx', x,x,'k:', x,ym,'k-')
```

The result is shown in fig. 4.1. At first sight, the result looks good. But wait — the model appears to be composed of straight lines between the experimental data points! That can not be correct for a polynomial model!

The problem is that we only computed model values for experimental values of  $x$ . Drawing a (solid) line through the computed values  $(x, y_m)$ , Matlab simply uses a straight line between the computed data points  $(x, y_m)$ . To get a better view of the model, we need to compute the model value  $y_m$  for abscissa values in between the experimental points!, e.g. as follows:

```
>> xm = linspace(0,1,100);
>> ym = polyval(c,xm);
>> plot(x,y,'kx', x,x,'k:', xm,ym,'k-')
>> legend('Experimental data: y(x)', 'y = x', 'Model of data: ym(x)')
>> xlabel('x')
>> ylabel('y')
>> title('Experimental data and model data for water-ethanol equilibrium')
```



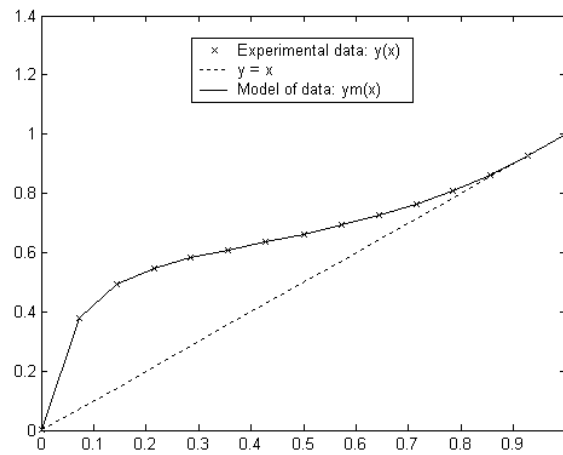


Figure 4.1: Experimental data  $((y, x), \times)$  and model  $((y_m(x), x)$ , solid) for water-ethanol equilibrium. Notice flawed model representation.

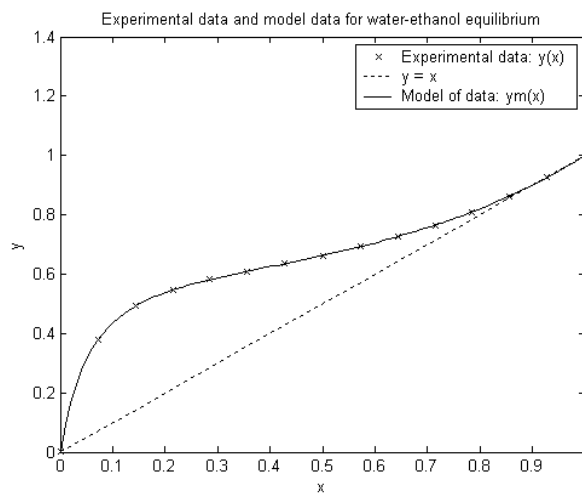


Figure 4.2: Experimental data  $((y, x), \times)$  and model  $((y_m(x), x)$ , solid) for water-ethanol equilibrium.

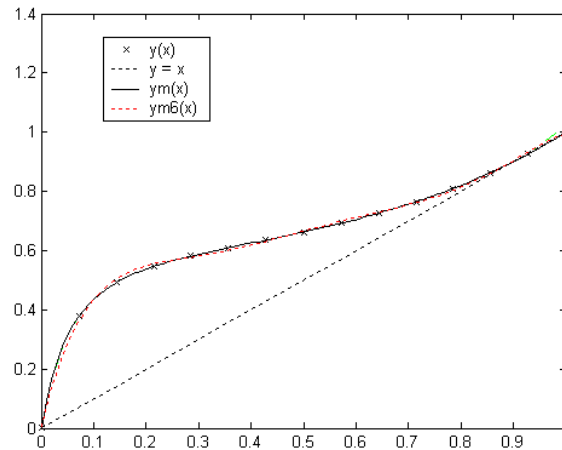


Figure 4.3: Experimental data  $((y, x), \times)$ , interpolation model  $((y_m(x), x), \text{solid})$ , and low order model  $((y_{m6}(x), x), \text{dotted})$  for water-ethanol equilibrium.

The resulting plot is shown in fig. 4.2.

The Matlab warning is caused by the fact that in this case, we could have found almost as good a model by using fewer parameters  $c_i$ :

```
>> c6 = polyfit(x,y,6);
>> ym6 = polyval(c6,xm);
>> plot(x,y,'kx', x,x,'k:', xm,ym,'k-', xm,ym6,'r:')
>> legend('y(x)', 'y = x', 'ym(x)', 'ym6(x)')
```

The result is shown in fig. 4.3. Since model `ym6` is not a true interpolation model (the model does not go through every experimental data point), we denote it a fitted model — in fact it is the model of 6-th order with the minimal squared error, i.e. the least squares model of 6-th order.

### 4.2.3 Other interpolation functions

Similar to function `polyfit`, function `spline(x,y)` calculates the data structure associated with cubic spline interpolation of the data set  $(x, y)$ . Next, similar to function `polyval`, function `ppval(c,xx)` calculates the model value according to spline interpolation. Note that array `xx` must be different from array `x` if we want to compute the value  $y_m(x)$  in between experimental data points `x`.

```
>> c = spline(x,y);
>> ym = ppval(c,xm);
>> plot(x,y,'kx', x,x,'k:', xm,ym,'k-')
>> legend('Experimental data y(x)', 'y = x', 'Spline model ym(x)')
>> xlabel('x')
>> ylabel('y')
>> ym = ppval(c,xm);
>> title('Experimental data and model data for water-ethanol equilibrium')
```

The result is shown in fig. 4.4.

Another interpolation function is `pchip(x,y)` which works similarly to `polyfit` and `spline`, but which is based on Hermitian cubic polynomials:

```
>> c = pchip(x,y);
```

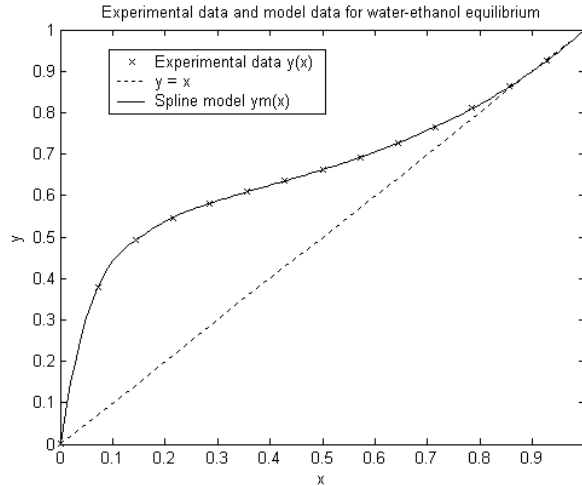


Figure 4.4: Experimental data  $((y, x), \times)$  and spline model  $((y_m(x), x), \text{solid})$  for water-ethanol equilibrium.

Table 4.2: Data set for comparison of interpolation models.

$i$	1	2	3	4	5
$x_i$	0.8	1.2	2.5	5.1	10
$y_i$	5.1	2.7	1.4	0.85	0.55

```
>> ym = ppval(c,xm);
>> plot(x,y,'kx', x,x,'k:', xm,ym,'k-')
>> xlabel('x')
>> ylabel('y')
>> title('Experimental data and model data for water-ethanol equilibrium')
>> legend('Experimental data y(x)', 'y = x', 'pchip model ym(x)')
```

The interpolation model is shown in fig. 4.5.

Finally, there is a generic interpolation function `interp1(x,y,xx)`, which combines the functionality of `spline` and `ppval` in one function. An optional fourth argument specifies the type of interpolation: either `'nearest'` for nearest neighbor interpolation, `'linear'` for the default linear interpolation, `'spline'` for cubic spline interpolation, and `'cubic'` for cubic interpolation.

For the data set we have studied so far, there is not much difference to see in the different interpolation models, see figs. 4.2 – 4.5. In the next section, we will consider a new data set which better illustrates the properties of the various interpolation techniques.

#### 4.2.4 Comparison of interpolation models

We consider the data set in Table 4.2.

We first create variables in Matlab to hold the data:

```
>> x = [0.8,1.2,2.5,5.1,10];
>> y = [5.1,2.7,1.4,0.85,0.55];
>> plot(x,y,'kx')
```

We want to interpolate the data in 100 data points between  $\min x$  and  $\max x$ :

```
>> x_int = linspace(min(x),max(x),100);
```

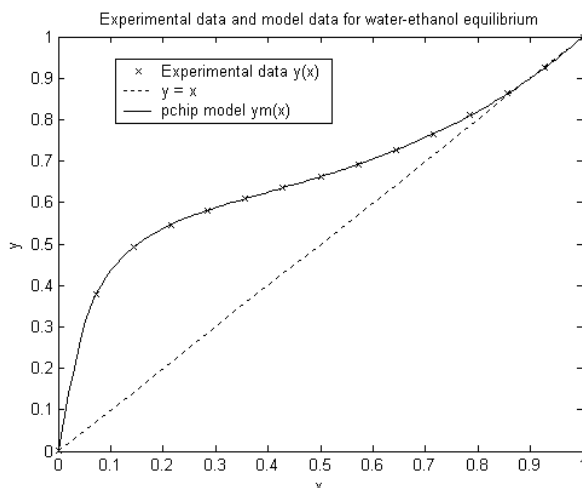


Figure 4.5: Experimental data  $((y, x), \times)$  and `pchip` model  $((y_m(x), x)$ , solid) for water-ethanol equilibrium.

Next, we compute the interpolation values from some interpolation models:

```
>> cint_pol = polyfit(x,y,4);
>> ymint_pol = polyval(cint_pol,x_int);
>>
>> cint_spl = spline(x,y);
>> ymint_spl = ppval(cint_spl,x_int);
>>
>> cint_pch = pchip(x,y);
>> ymint_pch = ppval(cint_pch,x_int);
>>
>> ymint_i1n = interp1(x,y,x_int,'nearest');
>>
>> ymint_i1l = interp1(x,y,x_int);
>>
>> ymint_i1s = interp1(x,y,x_int,'spline');
>>
>> ymint_i1c = interp1(x,y,x_int,'cubic');
```

We also include a 3-order polynomial least squares model:

```
>> cint_pls = polyfit(x,y,3);
>> ymint_pls = polyval(cint_pls,x_int);
```

Next, we plot the various models:

```
>> plot(x_int,ymint_pol1,'k--', x_int,ymint_spl,'k:', x_int,ymint_pch,'k-', ...
x_int,ymint_i1n,'k-', x_int,ymint_i1l,'k--', x_int,ymint_i1s,'k:', ...
x_int,ymint_i1c,'k-.', x_int,ymint_pls,'k:')
>> legend('polynom interp.', 'spline interp.', 'pchip interp.', ...
'nearest neigh. interp.', 'linear interp.', 'spline interp.', 'cubic interp.',...
'polynom least squares')
>> xlabel('x')
>> ylabel('y')
```

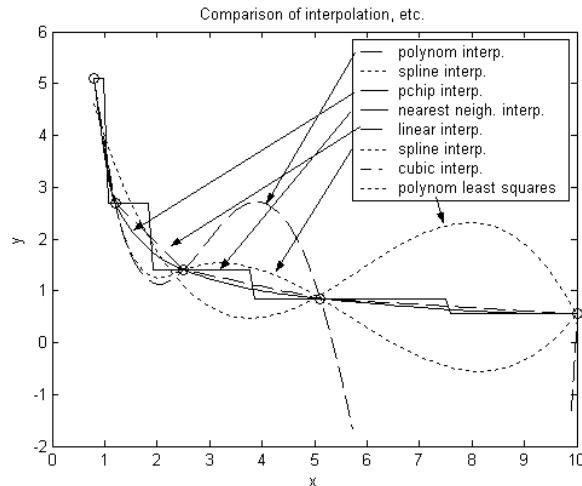


Figure 4.6: Comparison of various interpolation models.

```
>> title('Comparison of interpolation, etc.')
>> hold on;
>> plot(x,y,'ko')
```

The result is shown in fig. 4.6.

From the figure, it is somewhat difficult to compare the various models. We do, however see that the polynomial interpolation model (4-th order) probably is not very good at interpolating the data. Also the spline interpolation model appears to vary quite a lot between the data. The smoothest interpolation model appears to be the Hermitian cubic polynomial model (`pchip`). Although they are not differentiable, the nearest neighbor interpolation method and the linear interpolation method appear to give result similar to the Hermitian cubic polynomial model.

In sum, the various interpolation methods have different properties, and may give very different interpolation results, in particular if the number of available experimental data are few. We cannot really decide which interpolation method is “best”, simply because we do not know the true values between the experimental data points. If we have some notion that the variation is relatively smooth, then we can say more, however. It should also be noted that the smoothness of various models may depend on the available data: in the distillation data, all methods give essentially the same result.

## 4.3 Analysis

### 4.3.1 Polynomials in Matlab

Polynomial models have certain advantages over other models when it comes to the ease at which we can analyze the resulting model. Let us see how polynomials are dealt with in Matlab.

Let us start by looking at a polynomial model as found by function `polyfit`. We consider the data given in fig. 3.12:

```
>> c = polyfit(x,y,4)

c =

    -6.3220    15.5972   -12.8875     4.5495     0.0432

>> length(c)

ans =
```

Table 4.3: Functions on polynomials.

Function	Description
<code>conv(p,q)</code>	Multiplication of polynomials, $p(x) \cdot q(x)$
<code>[q,r] = deconv(b,a)</code>	Division of polynomials, $b(x) = a(x) \cdot q(x) + r(x)$
<code>polyder(p)</code>	Differentiation of polynomials, $\frac{d}{dx}p(x)$
<code>polyint(p)</code>	Integration of polynomials, $\int p(x) dx$
<code>polyval(p,x)</code>	Evaluate polynomial, $y = p(x)$
<code>roots(p)</code>	Roots of polynomials, $x : p(x) = 0$
<code>cplxpair(x)</code>	Sort complex array $\mathbf{x}$ into complex conjugate pairs

5

MATLAB represents polynomials as row arrays containing coefficients ordered by descending powers. Thus, row array  $\mathbf{c}$  represents the polynomial:

$$p(x) = -6.322x^4 + 15.5972x^3 - 12.8875x^2 + 4.5495x^1 + 0.0432.$$

### 4.3.2 Operations on polynomials

Matlab supports a number of functions for polynomials, see Table 4.3.

Suppose we want to multiply the polynomials  $(1 + x - x^2)$  and  $(2 + x^3)$ . Using a CAS<sup>1</sup>, we find:

$$(1 + x - x^2) \cdot (2 + x^3) = -x^5 + x^4 + x^3 - 2x^2 + 2x + 2.$$

Using Matlab, we find:

```
>> p = [-1, 1, 1];
>> q = [1,0,0,2];
>> conv(p,q)

ans =

    -1     1     1    -2     2     2
```

We see that the result is identical in Matlab and the CAS.

We can also divide polynomials:

```
>> deconv(ans,p)

ans =

     1     0     0     2
```

The answer is obviously correct: we have divided  $p \cdot q$  by  $p$ , and the answer is  $q$ . We can also divide  $q$  by  $p$  to get:

```
> [a,b] = deconv(q,p)

a =

    -1    -1
```

<sup>1</sup>CAS = Computer Algebra System, here: MuPAD through Scientific WorkPlace.

`b =`

```
0    0    2    3
```

The result means that

$$\frac{2 + x^3}{1 + x - x^2} = -x - 1 + \frac{2x + 3}{1 + x - x^2}.$$

It is trivial to show that this is the correct result.

We can differentiate a polynomial, e.g.  $q(x) = 2 + x^3$ :

```
>> polyder(q)
```

`ans =`

```
3    0    0
```

The result is  $q'(x) = 3x^2$ , which is correct. We can integrate a polynomial, e.g.  $p(x) = 1 + x - x^2$ :

```
>> polyint(p)
```

`ans =`

```
-0.3333    0.5000    1.0000    0
```

The result is  $\int p(x) dx = -0.3333x^3 + 0.5x^2 + x$ , which is correct.

We have previously used function `polyval` to evaluate polynomials. As an example,  $p(x = 0.5)$  can be found as follows:

```
>> polyval(p,0.5)
```

`ans =`

```
1.2500
```

Finally, we can find the roots polynomials, e.g.  $q(x) = 0$ :

```
>> roots(q)
```

`ans =`

```
-1.2599
 0.6300 + 1.0911i
 0.6300 - 1.0911i
```

where  $i = \sqrt{-1}$ . It can be shown that the roots of polynomials with real number coefficients, must occur in complex conjugate pairs if the root is complex, e.g. if  $z = 0.6300 + 1.0911i$  is a root, then  $\bar{z} = 0.6300 - 1.0911i$  must also be a root. It is possible to sort the roots in such a way that complex conjugate roots are grouped together using command `cplxpair`; in the example above, the `ans` array is already sorted.

### 4.3.3 Analysis of polynomial models

We consider the 6-th order least squares polynomial model for water-ethanol equilibrium:

```
>> c6 = polyfit(x,y,6)

c6 =

-34.5134  119.0827 -161.7170  110.1621  -39.3068    7.2848    0.0049
```

In the original data, fig. 3.12, we see that the value of  $y(x)$  dips below  $y = x$  for some value  $x_1$ , and we want to find an estimate of  $x_1$ . We can do that as follows: find the roots of  $y_m(x) - x = 0$ :

```
>> length(c6)

ans =

    7

>> roots(c6 - [0,0,0,0,0,1,0])

ans =

    0.9915
    0.8757 + 0.0497i
    0.8757 - 0.0497i
    0.3541 + 0.3384i
    0.3541 - 0.3384i
   -0.0008
```

What takes place here? First, we must find the polynomial description of  $y_m(x) - x$ .  $y_m(x)$  is represented by the polynomial of array `c6`. We must find the array that represents polynomial  $x$ ; this array must have the same length as array `c6`, or else we cannot subtract them.

We see that the complex conjugate roots are grouped already. The complex roots do not have a physical interpretation, and we are left with 2 real roots. Most likely, these roots,  $x \in \{0.0008, 0.9915\}$  are related to the trivial roots  $x = 0$  and  $x = 1$ . Thus, we can not find the location of  $x_1$  from polynomial `c6`.

What if we choose a polynomial of odd order? In that case, there must be an odd number of real roots! We find:

```
>> c5 = polyfit(x,y,5)

c5 =

    15.5423  -45.1778   49.6511  -25.1126    6.0893    0.0157

>> roots(c5 - [0,0,0,0,1,0])

ans =

    0.9892
    0.8571
    0.5317 + 0.3306i
    0.5317 - 0.3306i
   -0.0030
```



Now, it is reasonable to associate the root 0.8571 with  $x_1$ . From the data of fig. 3.12, it seems reasonable to bracket the root to lie in the interval  $x_1 \in [0.8631, 1]$ , and this doesn't really support the root we have found. Let us try with a higher dimensional polynomial of odd order:

```
>> c9 = polyfit(x,y,9);
>> x9 = zeros(size(c9));
>> x9(end-1) = 1

x9 =

    0    0    0    0    0    0    0    0    1    0

>> roots(c9-x9)

ans =

    0.9047 + 0.1562i
    0.9047 - 0.1562i
    0.9999
    0.9000
    0.5249 + 0.3435i
    0.5249 - 0.3435i
    0.1139 + 0.2661i
    0.1139 - 0.2661i
   -0.0000
```

From the 9-th order model, it appears as if the root  $x_1 \approx 0.9000$ , which fits better into bracket of the solution. Still, we have too few data to really be sure about the value of  $x_1$ .

We can also differentiate and integrate polynomials:

```
>> c6 = polyfit(x,y,6);
>> c6der = polyder(c6);
>> c6int = polyint(c6);
>> xx = linspace(0,1);
>> plot(xx, polyval(c6,xx),'k-', xx, polyval(c6der,xx),'k:', ...
xx, polyval(c6int,xx),'k-.' )
>> legend('y_m(x)', 'd y_m(x)/dx', '\int y_m(x) dx')
>> xlabel('x')
```

The result is shown in fig. 4.7.

Although it should be possible to write similar functions as those of Table 4.3 for spline models, etc., such functions are not currently available in Matlab.

## 4.4 More on interpolation and prediction models

### 4.4.1 Case study: Saturated steam

The data in fig. 4.8 are taken from <http://webbook.nist.gov>.

It is of interest to build a model that correlates pressure and temperature,  $p(T)$ . We start by importing the data in file `SaturatedSteamVapor.xls` into Matlab, and defining variables  $T$  and  $p$ :

```
>> T = data(3:end,1);
>> p = data(3:end,2);
>> plot(T,p,'kx')
>> xlabel('T [C^{\circ}]')
>> ylabel('p [atm]')
```

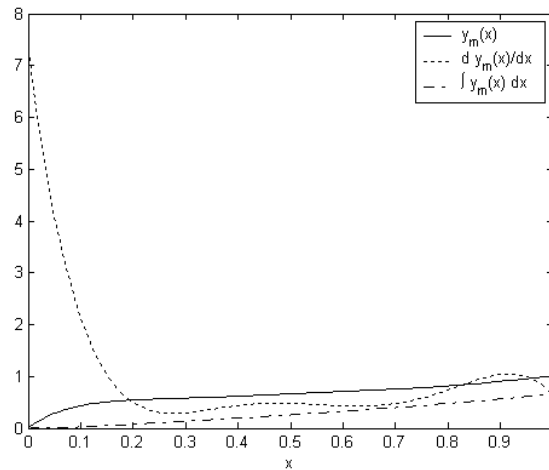


Figure 4.7: 6th order least squares polynomial model  $y_m(x)$  of water-ethanol equilibrium, with the derivative  $dy_m(x)/dx$  and the integral  $\int y_m(x) dx$  of  $y_m(x)$ .

Saturated steam: Vapor Phase data. Taken from <a href="http://webbook.nist.gov">http://webbook.nist.gov</a>													
Temperature (C)	Pressure (atm)	Density (kg/m <sup>3</sup> )	Volume (m <sup>3</sup> /kg)	Internal Energy (kJ/kg)	Enthalpy (kJ/kg)	Entropy (J/g <sup>o</sup> K)	Cv (J/g <sup>o</sup> K)	Cp (J/g <sup>o</sup> K)	Sound Spd. (m/s)	Joule-Thomson (F/atm)	Viscosity (uPa*s)	Therm. Cond. (W/m <sup>o</sup> K)	Phase
0.01	0.0060366	0.004855	205.99	2374.9	2500.9	9.1555	1.4184	1.8844	409	108.09	9.2163	0.017071	vapor
5.01	0.0066177	0.006807	146.91	2381.8	2510.1	9.0246	1.4226	1.8894	412.61	92.224	9.336	0.017339	vapor
10.01	0.01213	0.009413	106.24	2388.7	2519.2	8.8995	1.4269	1.8947	416.17	78.898	9.4614	0.017622	vapor
15.01	0.016846	0.012849	77.828	2395.5	2528.4	8.78	1.4314	1.9002	419.7	67.754	9.5921	0.017918	vapor
20.01	0.023102	0.017324	57.723	2402.3	2537.5	8.6658	1.4359	1.9059	423.19	58.461	9.7275	0.018228	vapor
25.01	0.031303	0.023088	43.313	2409.1	2546.5	8.5564	1.4405	1.9118	426.63	50.723	9.8671	0.018551	vapor
30.01	0.041938	0.030432	32.86	2415.9	2555.6	8.4518	1.4453	1.918	430.04	44.279	10.011	0.018887	vapor

Figure 4.8: Relationships between thermodynamic variables of vapor phase saturated steam.

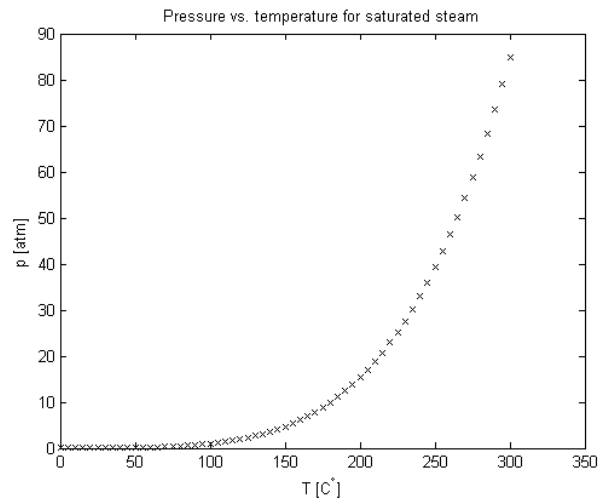


Figure 4.9: Pressure ( $p$ , atm) as a function of temperature ( $T$ , °C) for saturated steam.

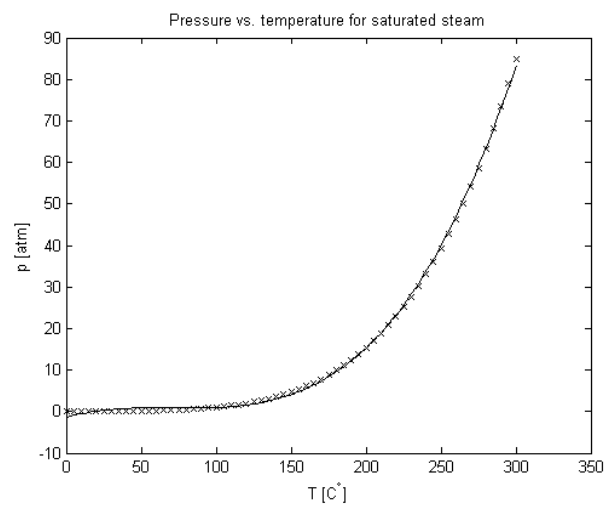


Figure 4.10: Experimental pressure data ( $p$ , atm,  $\times$ ) as a function of temperature ( $T$ , °C) for saturated steam, and 4th order least squares model fit (solid line).

```
>> title('Pressure vs. temperature for saturated steam')
```

The result is shown in fig. 4.9.

We choose to build a polynomial least squares model:

```
>> ppar = polyfit(T,p,3);
>> hold on;
>> TT = linspace(min(T),max(T));
>> plot(TT,polyval(ppar,TT), 'k-')
```

The result is not perfect, see fig. 4.10. In particular, the model is relatively poor at temperatures below ca. 80–90 °C.

Based on the model, what is the pressure at 100 °C? Using the polynomial least squares model, Matlab suggests:

```
>> polyval(ppar,100)
```

```
ans =
```

```
0.9650
```

The results is somewhat surprising — we would have expected that  $p^{\text{sat}}(100) = 1 \text{ atm}^2$  — the error is probably due to the relatively low quality of the polynomial model at low temperatures.

Let us turn the question around: at a pressure of 1 atm, what is the *temperature*? To answer this question, we must solve the equation  $p^{\text{sat}}(T) = 1$ . We must thus use the `roots` function for finding roots in Matlab:

```
>> length(ppar)
```

```
ans =
```

```
4
```

```
>> fpol = ppar - [0,0,0,1]
```

```
fpol =
```

```
0.0000 -0.0015 0.1035 -2.4529
```

```
>> roots(fpol)
```

```
ans =
```

```
1.0e+002 *
```

```
1.0223
```

```
0.5597 + 0.1790i
```

```
0.5597 - 0.1790i
```

```
>> format short g
```

```
>> ans
```

```
ans =
```

```
102.23
```

```
55.968 + 17.904i
```

```
55.968 - 17.904i
```

First, we created polynomial `fpol` which is  $p(T) - 1$ . Next, we used the `roots` function to find the roots of  $p(T) - 1 = 0$ . The answer is two roots which are complex (a complex conjugate pair  $55.968 \pm 17.904i$ ) and hence irrelevant, and the relevant real root  $T^{\text{sat}}(1) = 102.23 \text{ }^\circ\text{C}$ . Again, we are somewhat puzzled — we know that the answer should have been  $100 \text{ }^\circ\text{C}$ , but again, the inaccuracy is probably due to the poor quality of the model at “low” temperatures.

An interesting question is: could we have avoided the process of finding the roots of  $p(T) = 1$ ? The answer is yes! We could simply have built another model, where we fit a polynomial  $p(T)$  to the data! However, with the current data, least squares polynomial models  $T(p)$  are very poor — see fig. 4.11. Let us check the “disastrous” prediction qualities of the 5-th order model:

```
>> Tpar = polyfit(p,T,5);
```

---

<sup>2</sup>The boiling point temperature for water at pressure  $p = 1 \text{ atm}$ , is  $T = 100 \text{ }^\circ\text{C}$ .

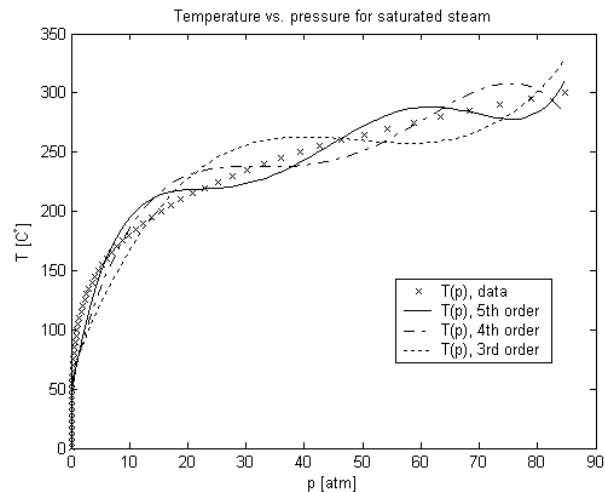


Figure 4.11: Experimental temperature ( $T$ , °C,  $\times$ ) as a function of pressure ( $p$ , atm) for saturated steam, and polynomial least squares models of degrees 3–5.

```
>> polyval(Tpar,1)
```

```
ans =
```

```
74.067
```

The correct answer should be 100 °C.

Let us see if the situation improves by using the `spline` function:

```
>> ppar = spline(T,p);
```

```
>> ppval(ppar,100)
```

```
ans =
```

```
1.0009
```

```
>> Tpar = spline(p,T);
```

```
>> ppval(Tpar,1)
```

```
ans =
```

```
99.974
```

Clearly, true interpolation using the `spline` function gives very good predictions.

In this section, we have tweaked around with mono variable functions, i.e. models where one variable is a function of another variable. It is often of interest to consider models where one variable is a function of more than one variable — so-called multivariable functions.

#### 4.4.2 Multivariable interpolation

In this section, some of the plotting techniques from Appendix D are used.

To illustrate the ideas, we use superheated steam as a case study. Superheated steam is vapor which is “above” the saturation state of steam. Because of this, the thermodynamic properties are functions of two variables. Figure 4.12 shows the variables at  $p = 0.5$  atm; data at other pressures can easily be created, e.g. for  $p \in \{1, 2, 5, 10, 20, 50\}$  atm; these data are taken from <http://webbook.nist.gov>. Note

Superheated steam: Taken from <a href="http://webbook.nist.gov">http://webbook.nist.gov</a>														
Pressure p = 0.5 atm														
	Temperature (C)	Pressure (atm)	Density (kg/m3)	Volume (m3/kg)	Internal Energy (kJ/kg)	Enthalpy (kJ/kg)	Entropy (J/g*K)	Cv (J/g*K)	Cp (J/g*K)	Sound Spd. (m/s)	Joule-Thomson (F/atm)	Viscosity (uPa*s)	Therm. Cond. (W/m*K)	Phase
4	100	0.5	0.29642	3.3736	2511.4	2682.3	7.689	1.4856	1.9755	474.89	11.897	12.311	0.024483	vapor
5	150	0.5	0.26051	3.8387	2585.7	2780.2	7.9351	1.4734	1.9483	506.03	6.6802	14.208	0.028526	vapor
6	200	0.5	0.23261	4.299	2659.9	2877.7	8.153	1.4886	1.9576	534.48	4.2898	16.192	0.03309	vapor
7	250	0.5	0.2102	4.7575	2735.1	2976.1	8.3507	1.513	1.9792	561.03	3.0028	18.23	0.038053	vapor
8	300	0.5	0.19176	5.2149	2811.5	3075.7	8.5325	1.5413	2.006	586.04	2.2249	20.296	0.043342	vapor
9	350	0.5	0.17631	5.6717	2889.4	3176.8	8.7015	1.5717	2.0355	609.76	1.7147	22.374	0.048908	vapor

Figure 4.12: Relationships between thermodynamic variables of superheated steam at  $p = 0.5$  atm.

that for higher pressures, the fluid will liquidize for low temperatures. Since we are interested in vapor behavior, liquid results will be deleted in Excel, and will appear as NaN (not a number) when imported into Matlab.

Assume that we want to find a model for  $\rho(T, p)$ , i.e. how the vapor density  $\rho$  varies with temperature  $T$  and pressure  $p$ . We import the data, and associate the imported data to descriptive variable names — note that the data starts in the fourth row:

```

Import Wizard created variables in the current workspace.
>> T0_5 = data(4:end,1);
>> p0_5 = data(4:end,2);
>> rho0_5 = data(4:end,3);
Import Wizard created variables in the current workspace.
>> T1 = data(4:end,1);
>> p1 = data(4:end,2);
>> rho1 = data(4:end,3);
Import Wizard created variables in the current workspace.
>> T2 = data(4:end,1);
>> p2 = data(4:end,2);
>> rho2 = data(4:end,3);
Import Wizard created variables in the current workspace.
>> T5 = data(4:end,1);
>> p5 = data(4:end,2);
>> rho5 = data(4:end,3);
Import Wizard created variables in the current workspace.
>> T10 = data(4:end,1);
>> p10 = data(4:end,2);
>> rho10 = data(4:end,3);
Import Wizard created variables in the current workspace.
>> T20 = data(4:end,1);
>> p20 = data(4:end,2);
>> rho20 = data(4:end,3);
Import Wizard created variables in the current workspace.
>> T50 = data(4:end,1);
>> p50 = data(4:end,2);
>> rho50 = data(4:end,3);

```

Next, we construct the temperature vector:

```

>> T = T0_5';
>> p = [0.5,1,2,5,10,20,50];
>> rho = [rho0_5, rho1, rho2, rho5, rho10, rho20, rho50]

rho =

Columns 1 through 6

    0.29642    0.59761         NaN         NaN         NaN         NaN
    0.26051    0.52326    1.0559         NaN         NaN         NaN
    0.23261    0.46645    0.93791    2.3849    4.9225         NaN
    0.2102     0.42113    0.84523    2.1362    4.3557    9.099
    0.19176    0.38399    0.76986    1.9391    3.9289    8.0794
    0.17631    0.35294    0.70713    1.7774    3.5876    7.3144
    0.16318    0.32658    0.65401    1.6416    3.3053    6.7032
    0.15187     0.3039    0.60841    1.5256    3.0667    6.1973
    0.14204    0.28418    0.5688     1.4253    2.8617    5.7687

Column 7

    NaN
    NaN
    NaN
    NaN
    22.405
    19.528
    17.537
    16.013
    14.782

```

Here, both  $T$  and  $p$  are row arrays, while  $\rho$  is an array where each row holds the pressure variation for a fixed temperature. Finally, we can interpolate:

```

>> [TT,pp] = meshgrid(T,p);
>> size(TT)

ans =

     7     9

>> size(rho)

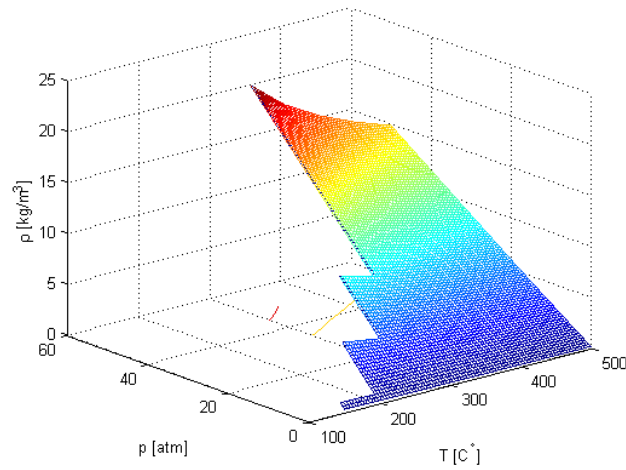
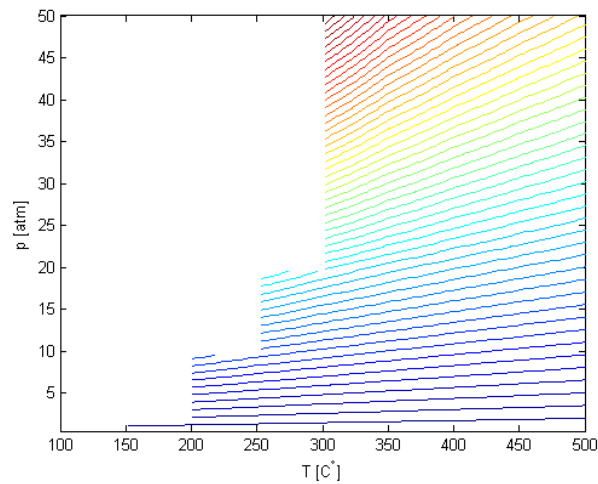
ans =

     9     7

>> Tvar = linspace(min(T),max(T));
>> pvar = linspace(min(p),max(p));
>> [TTvar, ppvar] = meshgrid(Tvar,pvar);
>> rhovar = interp2(TT,pp,rho',TTvar,ppvar);
>> meshc(TTvar,ppvar,rhovar)
>> xlabel('T [C^\circ]')
>> ylabel('p [atm]')
>> zlabel('\rho [kg/m^3]')

```

The resulting surface  $\rho(T, p)$  is displayed in fig. 4.13.

Figure 4.13: Interpolation model for  $\rho(T, p)$ .Figure 4.14: Contour plot of interpolation model for  $\rho(T, p)$ .

It is difficult to see from fig. 4.13 the valid region for  $(T, p)$  where  $\rho$  can be computed. A contour plot better illustrates the legal region:

```
>> contour(TTvar, ppvar, rho, 50)
>> xlabel('T [C^\circ]')
>> ylabel('p [atm]')
```

Figure 4.14 illustrates the valid region for  $(T, p)$  for the interpolation model that we have developed:

Suppose we want to find the vapor density of superheated steam at  $T = 375^\circ\text{C}$  and  $p = 3\text{ atm}$  — the point  $(T = 375^\circ\text{C}, p = 3\text{ atm})$  is well within the valid region for  $(T, p)$ , see fig. 4.14:

```
>> interp2(TT, pp, rho, 375, 3)
```

```
ans =
```

```
1.0235
```



The answer is:  $\rho_{375^\circ\text{C}, 3\text{ atm}} = 1.0235 \text{ kg/m}^3$ .



# Chapter 5

## Introduction to automation of tasks

### 5.1 Overview of learning goals

After having completed this chapter, you should have a clear understanding of why automation of tasks is useful, and how this can be achieved in Matlab. In particular, you should master:

- How to operate on strings,
- How to use the `for`-loop in Matlab,
- How to use the Matlab Editor, and restrictions on where you save files,
- What script files are, and know why they are useful.

### 5.2 Strings and basic string operations

It is useful to have some knowledge of strings in Matlab, since these are important for some types of task automation. A string is a sequence of characters. When defining the string, the sequence of characters must be surrounded by an apostrophe `'`:

```
>> clear
>> mystr1 = 'a';
>> mystr2 = 'abc';
>> whos
  Name          Size          Bytes  Class
  mystr1        1x1             2  char array
  mystr2        1x3             6  char array
```

Grand total is 4 elements using 8 bytes

We see that each character in a string takes up 2 bytes of memory.

For our purpose, the most useful string operations are:

- Concatenation of strings `s1` and `s2`: `[s1, s2]`,
- Conversion of number `n` to a string `s`: `s = num2str(n)`.

An example can be:

```
>> I = 2;
>> ['counter I has the value ', num2str(I)]
ans =
counter I has the value 2
```

A more thorough discussion of string operations is given in Appendix E.

### 5.3 Motivating examples

Suppose we want to find the sum

$$S_n = \sum_{k=1}^{n-1} \frac{1}{k^2}$$

for  $n = 1, 2, 3, \dots$ . This is a quite laborious task if we are to do it manually. Although we can write the expression as

$$S_n = \sum_{k=1}^{n-1} \frac{1}{k^2} + \frac{1}{n^2} = S_{n-1} + \frac{1}{n^2},$$

this doesn't help much in saving work:

```
>> S1 = 1/1^2
S1 =
    1
>> S2 = S1 + 1/2^2
S2 =
    1.2500
>> S3 = S2 + 1/3^2
S3 =
    1.3611
>> S4 = S3 + 1/4^2
S4 =
    1.4236
```

etc. ( $S_n$  converges to  $\frac{\pi^2}{6} \approx 1.6449$ ).

As another example, suppose we want to find the Fibonacci numbers  $f_k$  defined by:

$$\begin{aligned} f_1 &= 1 \\ f_2 &= 1 \\ f_{k+1} &= f_k + f_{k-1} \text{ for } k \geq 2. \end{aligned}$$

Straight ahead manual use of Matlab leads to:

```
>> f1 = 1;
>> f2 = 1;
>> f3 = f2 + f1
f3 =
    2
>> f4 = f3 + f2
f4 =
    3
>> f5 = f4 + f3
f5 =
    5
>> f6 = f5 + f4
f6 =
    8
```

etc.

In both of the examples above, we see that we repeat assignments that are almost the same. In the next section, we will introduce repetition statements which can be used to automate the computation of e.g.  $S_n$  and  $f_k$ .

**Exercise 5.1** Before proceeding to the repetition statements, let us just note that we can compute  $S_n$  *without* using repetition loops. As an example, we can compute  $S_n$  as follows:

```
>> n = 20;
>> N = 1:n;
>> S = 1./N.^2;
>> cumsum(S)
ans =
Columns 1 through 5
1.0000 1.2500 1.3611 1.4236 1.4636
Columns 6 through 10
1.4914 1.5118 1.5274 1.5398 1.5498
Columns 11 through 15
1.5580 1.5650 1.5709 1.5760 1.5804
Columns 16 through 20
1.5843 1.5878 1.5909 1.5937 1.5962
```

Experiment with  $n$  (i.e.  $n$ ) and see how large a value we must choose for  $n$  before the value  $\left|S_n - \frac{\pi^2}{6}\right| < 0.01$ . ■

In the Exercise above, we compute an array  $S$  which holds  $S_1, S_2$ , etc. See the Help browser for how Matlab function `cumsum` works.

## 5.4 The basic repetition statement

In order to repeat Matlab statements a specific number of times, the construct

```
for var = array
    statements
end
```

is used.<sup>1</sup> Here, `for` and `end` are keywords that should be used in verbatim<sup>2</sup>, and `array` is an array. The columns of `array` are then assigned one at a time into `var` starting with the first column, and the statements are executed. Often, `array` is a row array, hence the “columns” are scalars. It should be remembered that e.g. `1:N` creates a row array `[1, 2, ..., N]`, thus the statement `variable = 1:N` is quite common.

Let us look at an example: suppose we want to print the numbers  $i$  and  $i^2$  to the screen for  $i \in \{1, \dots, 10\}$ : clearly we do not intend the meaning  $i = \sqrt{-1}$ . Since `i` by default denotes  $\sqrt{-1}$ , we use `I` as the *variable* name instead of `i`. The *statement* is `[I, I^2]`, which ensures that  $i$  and  $i^2$  are displayed on the same line:

```
>> format compact
>> for I=1:10, [I, I^2], end
ans =
     1     1
ans =
     2     4
ans =
     3     9
ans =
     4    16
ans =
     5    25
ans =
     6    36
```

<sup>1</sup>Alternatively, the statements can be written in a single line as `for variable = array, statement 1, statement 2, ..., statement n, end`.

<sup>2</sup>Verbatim = word for word, exactly as spoken or written.

```

ans =
    7    49
ans =
    8    64
ans =
    9    81
ans =
   10   100

```

Here, we have typed the command on a single line.

We can also type the commands on separate lines ( $i$  and  $\sqrt{i}$ , for a variation):

```

>> for I=1:10
[I, sqrt(I)]
end
ans =
    1    1
ans =
 2.0000    1.4142
ans =
 3.0000    1.7321
ans =
    4    2
ans =
 5.0000    2.2361
ans =
 6.0000    2.4495
ans =
 7.0000    2.6458
ans =
 8.0000    2.8284
ans =
    9    3
ans =
10.0000    3.1623

```

In practice, we often use (much) more complex statements, and it is not practical to type the commands on the command line of Matlab. One reason is that it is easy to lose track of which commands we have carried out and which we have not carried out. Another reason is that it is easy to make mistakes: If we make mistakes in any line of the command sequence, we have to redo everything that depends on the erroneous command.

## 5.5 Scripts and the Matlab editor

A script is a file containing a sequence of Matlab statements. In order to run those statements, we simply type the name of the script file on Matlab's command line. Matlab's editor is convenient for typing and editing the statements in a script file, and we can start the editor by issuing the command `>> edit` on Matlab's command line. Figure 5.1 illustrates the opened Matlab Editor, containing three lines of Matlab commands.

Before running the script, it is necessary to save the file (**File/Save**). The script file must be stored either in a file in the *Current Directory* — see fig. 1.1 p. 4, or in Matlab's so-called *Path* — see **help path**. To run the script, either click on the *Save and run icon* of the Matlab Editor (see fig. 5.1), or type the file name in the Matlab Command Window. The file in fig. 5.1 has been saved with the file name `RepetitionDemo1`:

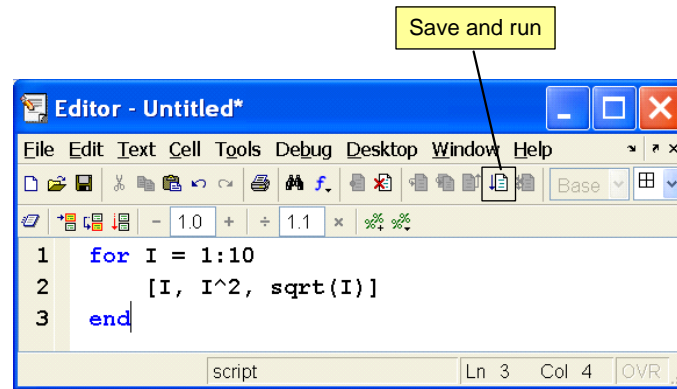


Figure 5.1: Matlab Editor window, containing a sequence of Matlab commands. Note: if the file has been saved, the “Save and run” icon changes to the “Run” icon.

```

>> RepetitionDemo1
ans =
     1     1     1
ans =
 2.0000    4.0000    1.4142
ans =
 3.0000    9.0000    1.7321
ans =
     4    16     2
ans =
 5.0000   25.0000    2.2361
ans =
 6.0000   36.0000    2.4495
ans =
 7.0000   49.0000    2.6458
ans =
 8.0000   64.0000    2.8284
ans =
     9    81     3
ans =
10.0000 100.0000    3.1623
  
```

It is considered good practice to *comment* script files in Matlab so that it is easier to understand what is taking place, when looking over the script at a later date. Characters appearing after the symbol % on a (command) line, are considered as comments — see fig. 5.2 which produces a multiplication table.

The result of running the script (named `RepetitionDemo2`) is shown below:

```

>> RepetitionDemo2
>> A
A =
     1     2     3     4     5     6     7     8     9    10
     2     4     6     8    10    12    14    16    18    20
     3     6     9    12    15    18    21    24    27    30
     4     8    12    16    20    24    28    32    36    40
     5    10    15    20    25    30    35    40    45    50
     6    12    18    24    30    36    42    48    54    60
     7    14    21    28    35    42    49    56    63    70
     8    16    24    32    40    48    56    64    72    80
     9    18    27    36    45    54    63    72    81    90
    10    20    30    40    50    60    70    80    90   100
  
```

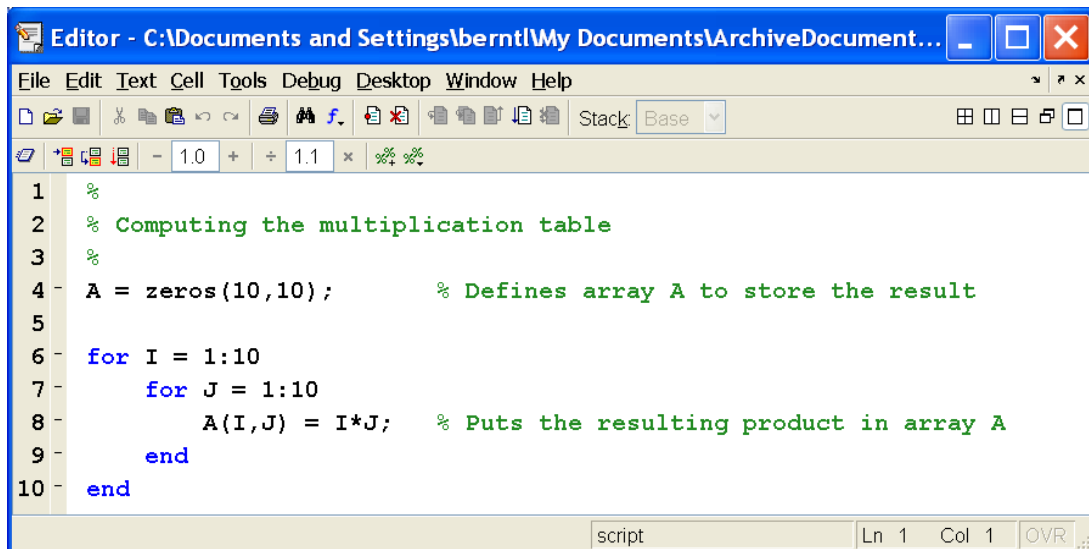


Figure 5.2: Matlab editor showing script file with comments and spaces.

In order to find the product  $7 \times 9$ , find the intersection of the row with 7 in the first column, and the column with 9 in the first row. The result is 63.

## 5.6 Examples: Repetition and Sums

Suppose we want to use the `for` statement to find the sum  $S_n = \sum_{k=1}^{n-1} \frac{1}{k^2} = S_{n-1} + \frac{1}{n^2}$  and the Fibonacci numbers  $f_k$ . First, we study how to find  $S_n$ . The following Matlab script will find  $S_n$ , fig. 5.3.

The result is:

```

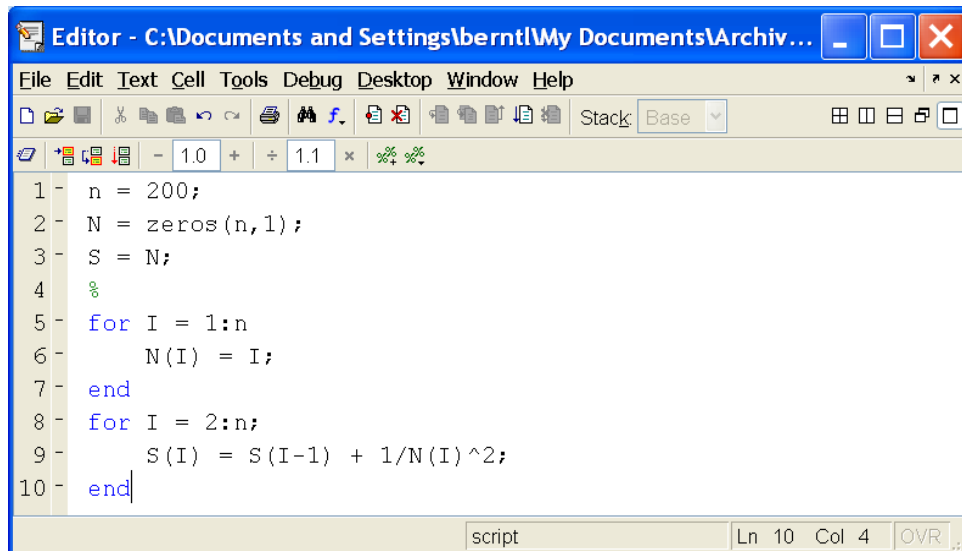
>> SumInvSquare
>> S
S =
    0
    0.25
    0.36111
    0.42361
    0.46361
    0.49139
    0.5118
    0.52742
    0.53977
    0.54977
    ...
    0.63985
    0.63987
    0.6399
    0.63992
    0.63995

```

Similarly, we find the Fibonacci numbers as follows, fig. 5.4.

The result is:

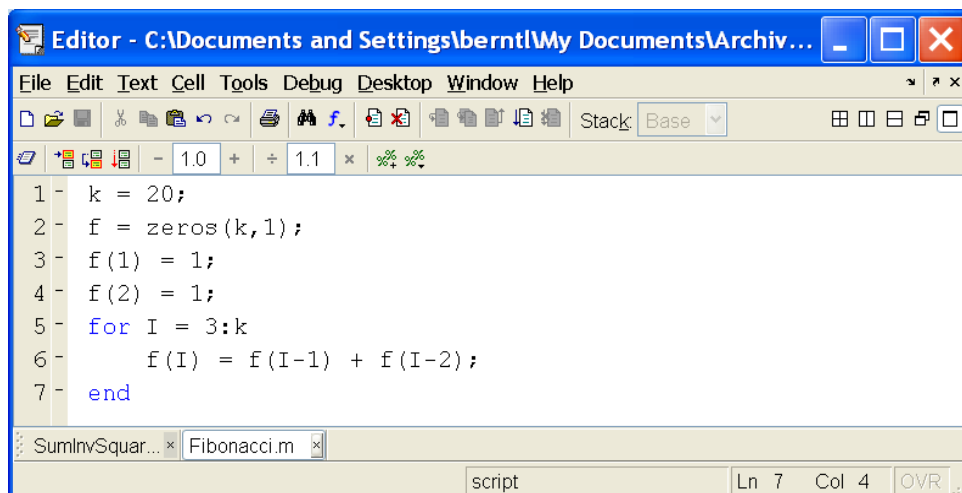




```

Editor - C:\Documents and Settings\bernt\My Documents\Archiv...
File Edit Text Cell Tools Debug Desktop Window Help
Stack: Base
- 1.0 + 1.1 x % %
1 - n = 200;
2 - N = zeros(n,1);
3 - S = N;
4 - %
5 - for I = 1:n
6 -     N(I) = I;
7 - end
8 - for I = 2:n;
9 -     S(I) = S(I-1) + 1/N(I)^2;
10 - end
script Ln 10 Col 4 OVR

```

Figure 5.3: Script SumInvSquare for computing  $S_n$ .


```

Editor - C:\Documents and Settings\bernt\My Documents\Archiv...
File Edit Text Cell Tools Debug Desktop Window Help
Stack: Base
- 1.0 + 1.1 x % %
1 - k = 20;
2 - f = zeros(k,1);
3 - f(1) = 1;
4 - f(2) = 1;
5 - for I = 3:k
6 -     f(I) = f(I-1) + f(I-2);
7 - end
SumInvSquar... x Fibonacci.m x
script Ln 7 Col 4 OVR

```

Figure 5.4: Script Fibonacci for computing  $f_k$ .

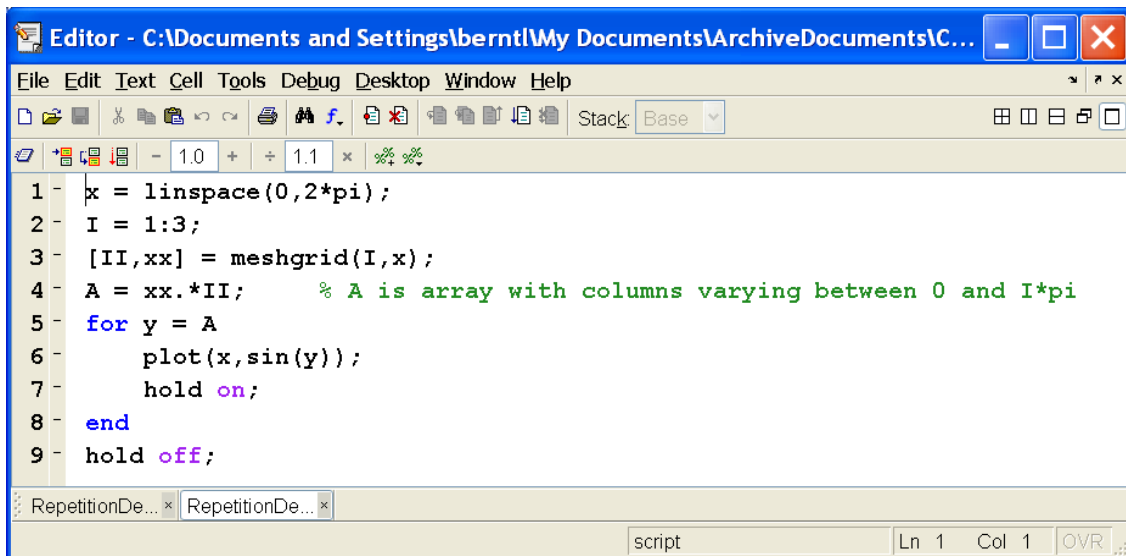


Figure 5.5: Matlab Editor with script where the repetition `array` has more than one row.

```
>> Fibonacci
>> f
f =
     1
     1
     2
     3
     5
     8
    13
    21
    34
    55
    89
   144
   233
   377
   610
   987
  1597
  2584
  4181
  6765
```

## 5.7 Example: Basic repetition over array\*

This example uses some techniques introduced in Appendix D. Let us also study the case where `variable` = `array`, and `array` has more than one row. We consider the script of fig. 5.5.

The result of issuing the command

```
>> RepetitionDemo3
```

is fig. 5.6.

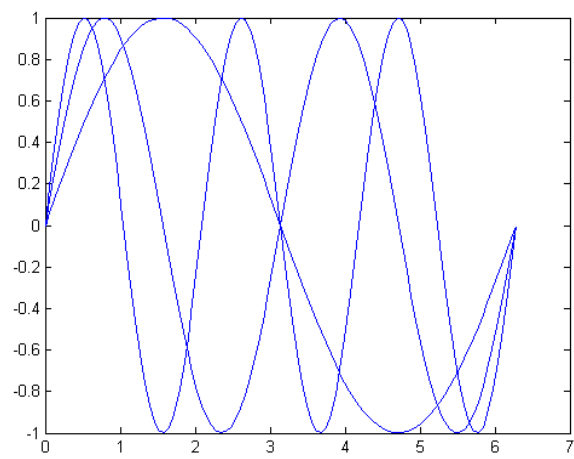


Figure 5.6: The result of running script RepetitionDemo3.



# Chapter 6

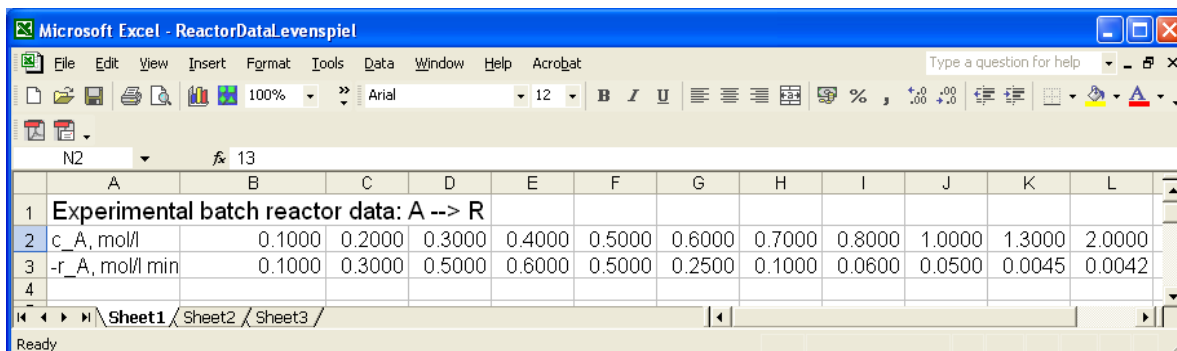
## Problems

**Problem 6.1** We consider a *batch reactor*: a component  $A$  is converted to component  $R$  in a closed vessel through a chemical reaction. The concentration of component  $A$ ,  $c_A$  (mol/l) can be described by the ordinary differential equation (ODE):

$$\frac{dc_A}{dt} = r_A,$$

where  $r_A$  is the so-called rate of production of component  $A$  (mol/(l · min)).  $r_A$  is a function of  $c_A$  as given in fig. 6.1, taken from (Levenspiel 1972), p. 117.

1. Use the *Import Wizard* of Matlab to import the data from file `ReactorDataLevenspiel.xls`. (**Hint:** in Matlab, use command `File/Import Data...`).
2. Extract the values related to  $c_A$  and  $r_A$  from the `data` variable, and put them into variable names of your choice. Make sure that the chosen names are legal variable names in Matlab, and that the chosen variable names not already are used as function names. (**Note:** the Excel spreadsheet gives  $-r_A$ , and not  $r_A$ . Since  $-r_A$  is positive,  $r_A$  is negative, which means that component  $A$  is *consumed*.)  
Plot the experimental data to make sure that the data are correct. If you suspect that some data are outliers, then remove these faulty data. Replot the experimental data when possible outliers are removed.
3. Save the data in a `.mat` file. Clear the Matlab workspace, and check that the workspace is empty. Load the data from the file where you saved them.
4. Experiment with fitting models to the data for  $(c_A, r_A)$ . Use the following model types:
  - Polynomial least squares models of various orders. (**Hint:** `polyfit`, `polyval`.)
  - Spline interpolation models. (**Hint:** `spline`, `ppval`.)



	A	B	C	D	E	F	G	H	I	J	K	L
1	Experimental batch reactor data: A --> R											
2	c_A, mol/l	0.1000	0.2000	0.3000	0.4000	0.5000	0.6000	0.7000	0.8000	1.0000	1.3000	2.0000
3	-r_A, mol/l min	0.1000	0.3000	0.5000	0.6000	0.5000	0.2500	0.1000	0.0600	0.0500	0.0045	0.0042
4												

Figure 6.1: Experimental batch reactor data for reaction with stoichiometry  $A \rightarrow R$ . Taken from (Levenspiel 1972), p. 117.

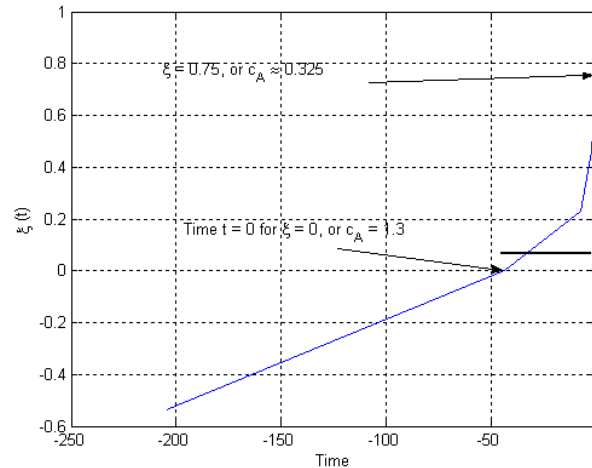


Figure 6.2: Extent of reaction  $\xi(t)$  for reactor data based on trapezoidal integration.

- Hermitian cubic polynomials. (**Hint:** `pchip`, `ppval`.)

What model types are acceptable? (**Hint:** Is it acceptable that  $r_A$  becomes negative?)

5. \* The *extent*  $\xi_A$  of reaction is defined as follows:

$$\xi_A(t) \triangleq \frac{c_A(0) - c_A(t)}{c_A(0)}.$$

Find  $\xi_A(t)$ , and plot  $\xi_A(t)$  as a function of  $t$  when  $c_A(0) = 1.3$  mol/l. At  $t = 0$ ,  $\xi(0) = 0$ . At what time does  $\xi_A(t)$  rise above 0.75?

(**Hint:** The principle is based on finding  $c_A(t)$  from the solution of the ODE

$$\frac{dc_A}{dt} = r_A \iff \frac{dc_A}{r_A(c_A)} = dt \iff \int_{c_A(0)}^{c_A(t)} \frac{dc_A}{r_A(c_A)} = \int_0^t dt = t.$$

This leads to  $F(c_A(t), c_A(0)) = t$ , which may be inverted to give  $c_A(t) = G(t, c_A(0))$ . Then we can find  $\xi_A(t)$  from its definition. A possible procedure is as follows:

- Use array operations to compute the array containing  $1/r_A$ .
  - Use Matlab function `cumtrapz(x,y)`, which computes an array with a numeric approximation of  $\left(\int_{x_1}^{x_2} y(x) dx, \int_{x_1}^{x_3} y(x) dx, \dots, \int_{x_1}^{x_N} y(x) dx\right)$  where  $x_i$  is a discrete value of  $x$ . (**Hint:** In the current problem,  $x$  is  $c_A$  and  $y$  is  $1/r_A$ ).
  - Plot  $c_A(t)$ . (**Hint:** Since  $\int_{c_{A,1}}^{c_{A,i}} \frac{dc_A}{r_A(c_A)} = F(c_A(t_i), c_{A,1}) = \int_0^t dt = t_i$ , each pair  $(F(c_A(t_i), c_{A,1}), c_{A,i}) \equiv (t_i, c_{A,i})$ . We can use this fact to plot  $(t, c_A(t))$ . Verify that the result is similar to in fig. 6.2.)
  - By zooming in at the points of fig. 6.2 where  $\xi_A = 0$  (i.e.  $c_A = 1.3$ , which indicates time  $t = 0$ ) and the time where  $\xi_A = 0.75$  (the sought extent of reaction), find the time for achieving 75% extent of reaction.
6. \* The calculations in the previous question are rather coarse due to relatively few data points. We can achieve smoother (but not necessarily more accurate) calculations as follows:
- Use array operations to compute the array containing  $1/r_A$ .
  - Use your preferred interpolation model for  $(c_A, 1/r_A)$ .
  - Compute  $1/r_A$  for an extended set of data points in the interval  $(\min(c_A), \max(c_A))$  (e.g. using the `linspace` function of Matlab) with the help of the interpolation model.

- Use Matlab function `cumtrapz(x,y)` to compute the relevant array containing  $\int_{c_{A,1}}^{c_{A,i}} \frac{dc_A}{r_A(c_A)} = F(c_A(t_i), c_{A,1}) = \int_0^{t_i} dt = t_i$ , and plot the smoothed pair  $(t_i, c_{A,i})$ , as well as  $(t_i, \xi_{A,i})$ .
- Find the time necessary to achieve 75% extent of reaction. Is the answer different from the result in the previous question?





## **Part II**

# **Exploiting the power of Matlab**



# Chapter 7

## Revisiting Part I

(If you do not recall the following topics, you should re-check Part I or use Matlab's help facilities.)

**The Matlab environment** See Chapter 1.

- Command window
- Help window
- Workspace window
- Array editor

**Basic operations**

- The array data type (scalars, row and column arrays, 2 dimensional arrays), see Section 2.2.
- Assignment, Section 2.3.
- Variables, see Section 2.5
- Special variables: `pi`, `i`, `j`, etc., Table 2.1 p. 9.
- How to create arrays: `zeros`, `ones`, `eye`, `diag`, colon operator, `rand`, `randn`, `linspace`, `logspace`, `size`, `length`. See Sections 2.6.2 – 2.6.5, and 2.10.
- Selecting parts of arrays: `A(2:3,5:8)`, `A(1,3:end)`; building/concatenating arrays. See Section 2.11.
- Simple writing and reading of data from file: `save`, `load`, import wizard, see Section 2.7.
- Housekeeping of data: `clear`, `who`, `whos`, `what`, commands, and files. See Section 2.12.
- Basic binary array operations: `+`, `-`, `.*`, `./`. See Table 2.6 p. 34.
- Basic unary array functions: `sin`, `cos`, `tan`, `exp`, `log`, `log2`, `sqrt`, `abs`, `ceil`, `floor`, transpose, etc. See Tables 2.7 p. 36, 2.8 p. 37, 2.9 p. 37, 2.10 p. 39, and 2.11 p. 40.
- Arrays vs. vectors and matrices, see Section A.

**Plotting** See Table 3.1 p. 43. In particular, note:

- Basic plotting: `plot`, `semilogx`, `semilogy`, `loglog`, `linestyle`, `marker`, `color`
- Tailoring your plot: `axis`, `title`, `xlabel`, `ylabel`, `legend`, `hold`, `subplot`
- `figure`, `close`

**Simple data analysis\*** See Table 4.1 p. 57. In particular, you should check out rudimentary commands such as `max`, `min`, `mean`, `sum`, as well as model fitting techniques such as `polyfit`, `polyval`. See also how you can use the `find` command to spot outliers and missing data (`NaN`).

**Introduction to automation of tasks**

- Basic string operations. See Tab. E.1 p. 174, and Tab. E.2 p. 176 for a more detailed treatment.
- `for`-loops, see Section 5.4.
- Scripts, see Section 5.5.

# Chapter 8

## More program flow control

### 8.1 Introduction

In Part I of these lecture notes, we started to look into Matlab programming by developing Matlab scripts using the Matlab editor, and automating repetitive tasks using the `for` loop. In this chapter, we will start by looking into program flow control by the `if` statement. To do this, we must first discuss the concepts of logical variables and relations, as well as logical expressions. Then we are ready for discussing the `if` statement. Thereafter, we will discuss the `switch` statement, and finally we will discuss an alternative repetition statement: the `while` loop.

### 8.2 Logical variables in Matlab

Writing scripts and functions, we often need to choose between alternative actions depending on the result of one or more tests. To do this we need relational and logical operators and logical (Boolean) variables.

Traditionally, Matlab only supported *floating point* numbers, but later versions of Matlab support various other data types similar to other programming languages, such as logical (Boolean) variables. Logical variables can only have two possible outcomes, `true` or `false`. Since the early Matlab versions did not support logical variables, such variables were “emulated” by interpreting the floating point number 0 to be `false`, and any floating point number  $\neq 0$  to be `true`.

### 8.3 Relations: functions and operators

In programming languages, relations are statements which are either `true` or `false`, i.e. statements where the result is a logical quantity. Table 8.1 lists Matlab functions and operators for relational statements.

Matlab’s parser translates the *operator expressions* into the function form. So if we write `x > y`, Matlab sees the function `gt(x,y)`. The output of a relational expression is a *logical variable*, having the value `true` or `false`.

**Examples** Consider the following simple examples:

Table 8.1: Matlab relational functions and operators. In the examples, it is assumed that  $x = 3, y = 4$ .

Math operator	Matlab function	Matlab operator	Example
<	lt	<	<code>x &lt; y --&gt; true</code>
≤	le	<=	<code>x&lt;=y -&gt; true</code>
=	eq	==	<code>x==y -&gt; false</code>
≠	ne	~=	<code>x~=y -&gt; false</code>
>	gt	>	<code>x&gt;y -&gt; false</code>
≥	ge	>=	<code>x&gt;=y -&gt; false</code>

```

>> x = 2;
>> y = 3;
>> X = [1,2,3];
>> Y = [4,3,2];
>> x < y
ans =
    1
>> y < x
ans =
    0
>> x == y
ans =
    0

```

We can also give name to the result from a statement:

```

>> logic = x == y
logic =
    0
>> logic = x < y
logic =
    1

```

We can consider relations between arrays of *equal size*, where e.g.  $[x, y, z] \leq [u, v, w]$  is interpreted as  $[x \leq u, y \leq v, z \leq w]$ :

```

>> X < Y
ans =
    1 1 0
>> X == Y
ans =
    0 0 0
>> log = X >= Y
log =
    0 0 1

```

## 8.4 Logical expressions

In the previous section, we considered statements about the *ranking* of floating point numbers. The result was logical quantities which assessed the truth of these statements.

It is also useful to work with logical expressions, i.e. combinations of *logical variables*. Table 8.2 gives an overview of Matlab operators for combining logical variables.

**Any(a)** is true if at least one element in **a** is true. **All(a)** is true if all elements in **a** are true.

**Examples** Consider the following simple examples:

```

>> 1 & 0
ans =
    0
>> 1 & 1
ans =
    1

```

Table 8.2: Matlab logical functions and operators. In the examples, it is assumed that  $a = [1, 0, 0]$ ,  $b = [1, 1, 0]$ .

Logical operator	Matlab function	Matlab operator	Example
AND	and	&	a & b --> [1 0 0]
OR	or		a b --> [1 1 0]
Exclusive OR	xor		xor(a,b) --> [0 1 0]
NOT	not	~	~a --> [0 1 1]
Any	any		any(a) --> 1
All	all		all(a) --> 0

```
>> 1 | 0
ans =
     1
>> xor(1,0)
ans =
     1
>> any([1,0,1])
ans =
     1
>> all([1,0,1])
ans =
     0
>> all([1,1,1])
ans =
     1
```

Often, the logical expressions are combined with relations:

```
>> x = 2;
>> y = 3;
>> X = [1,2,3];
>> Y = [4,3,2];
>> (x<y) & (X<Y)
ans =
     1     1     0
>> any((x<y) & (X<Y))
ans =
     1
>> all((x<y) & (X<Y))
ans =
     0
```

Finally, we can make the logical expressions simpler by introducing logical variables:

```
>> logic1 = x<y;
>> logic2 = X<Y;
>> logic3 = logic1 & logic2
logic3 =
     1     1     0
>> logic4 = any(logic3)
logic4 =
     1
>> logic5 = all(logic3)
logic5 =
     0
```

## 8.5 Making logical choices: the if statement

### 8.5.1 Motivating example

As a motivation, consider the formulae for the root  $x \in \mathbb{C}$  of a second order polynomial  $ax^2 + bx + c = 0$ :

$$x = \begin{cases} \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, & a \neq 0 \\ -\frac{c}{b}, & a = 0, b \neq 0 \\ \emptyset, & a = 0, b = 0, c \neq 0 \\ \mathbb{C} & a = 0, b = 0, c = 0 \end{cases}.$$

We see that the answer depends on the values of coefficients  $(a, b, c)$ . In order to write a Matlab script for the solution  $x$  with  $(a, b, c)$  is known, we need to be able to test which formulae is relevant.

### 8.5.2 Structure of if statements

The standard test statement in most programming languages is the **if** statement. In Matlab, the basic syntax is as follows:

```
if expression
    statements
end
```

Here, **expression** is a logical variable/expression. If the logical variable is *true*, then the statements are executed. If not, they are not executed.

In many cases, we want to test for more than one logical variable/expression. We then use the following extended version of the **if** statement:

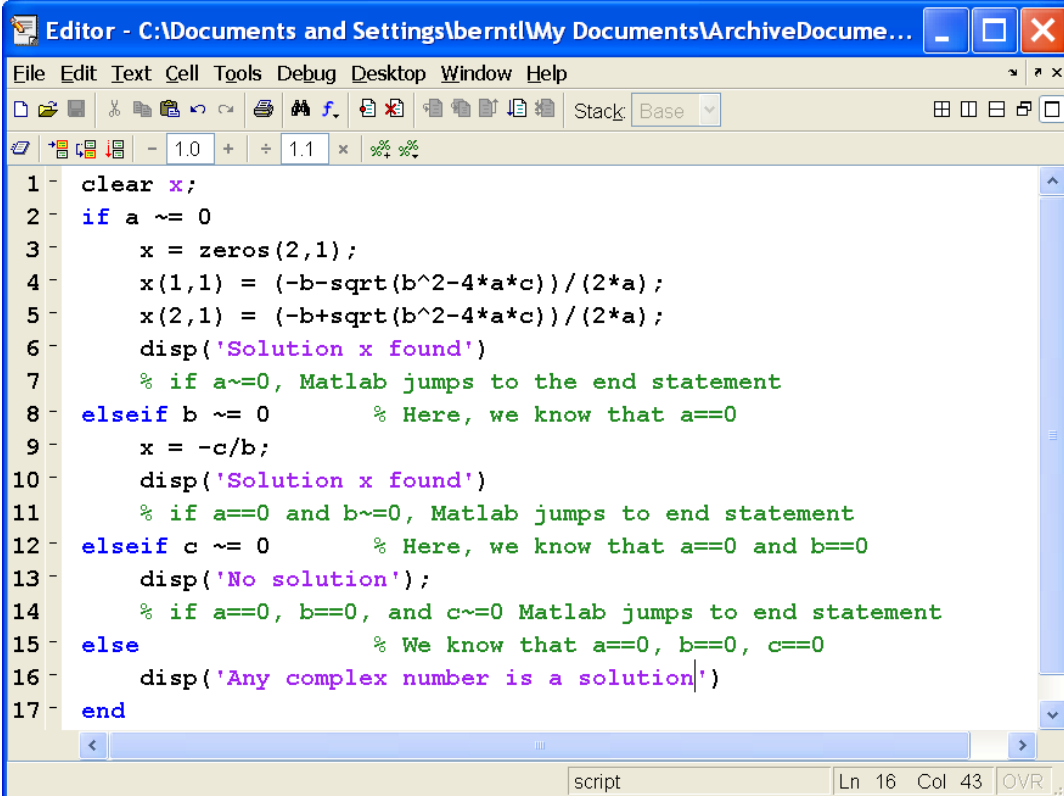
```
if expression1
    statements1
elseif expression2
    statements2
elseif expression3
    statements3
...
else
    statements
end
```

### 8.5.3 Example: use of if statement

The following script will find the root of the second order polynomial, fig. 8.1. Below are some examples of the use of this script:

```
>> a=1;b=2;c=-2;
>> rootscript
Solution x found
>> x
x =
    -2.7321
     0.7321
>> a=2;b=1;c=2;
>> rootscript
Solution x found
>> x
x =
```





```

1 - clear x;
2 - if a ~= 0
3 -     x = zeros(2,1);
4 -     x(1,1) = (-b-sqrt(b^2-4*a*c))/(2*a);
5 -     x(2,1) = (-b+sqrt(b^2-4*a*c))/(2*a);
6 -     disp('Solution x found')
7 -     % if a~=0, Matlab jumps to the end statement
8 - elseif b ~= 0      % Here, we know that a==0
9 -     x = -c/b;
10 -    disp('Solution x found')
11 -    % if a==0 and b~=0, Matlab jumps to end statement
12 - elseif c ~= 0     % Here, we know that a==0 and b==0
13 -    disp('No solution');
14 -    % if a==0, b==0, and c~=0 Matlab jumps to end statement
15 - else              % We know that a==0, b==0, c==0
16 -    disp('Any complex number is a solution|')
17 - end

```

Figure 8.1: Script for finding root of  $ax^2 + bx + c = 0$ , where  $a, b, c$  are specified in the Matlab Command Window.

```

-0.2500 - 0.9682i
-0.2500 + 0.9682i
>> a=0;b=2;c=-3;
>> rootscript
Solution x found
>> x
x =
    1.5000
>> a=0;b=0;c=2;
>> rootscript
No solution
>> a=0;b=0;c=0;
>> rootscript
Any complex number is a solution

```

## 8.6 Logical choices: the switch statement

When we need to test for a logical variable, or for a (complex) logical expression, we use the `if` statement as discussed in the previous section.

In many cases, the `if` statements are of the following type:

```

if variable == case_value1
    statements1
elseif variable == case_value2
    statements2
elseif variable == case_value3

```

```

    statements3
...
else
    statements
end

```

This particular kind of `if` statement structure can be expressed in the slightly more convenient form of a `switch` statement:

```

switch variable
  case case_value1
    statements1
  case case_value2
    statements2
  case case_value3
    statements3
...
  otherwise
    statements
end

```

## 8.7 Repetition loop: the `while` statement

### 8.7.1 Motivating example

We have already met the `for` statement:

```

for var = array
    statements
end

```

It is characteristic for the `for` statement that the variable `var` loops through the values of `array` a predetermined number of times.

Sometimes, we do not know in advance how many times we want to repeat an operation. As an example, suppose we want to find

$$S = \sum_{k=1}^{\infty} \frac{1}{k^2} = \frac{\pi^2}{6} \approx 1.6449.$$

to a certain accuracy. We thus use the approximation  $S_n = \sum_{k=1}^n \frac{1}{k^2} = S_{n-1} + \frac{1}{n^2}$ , where  $n$  is unknown but implicitly determined by the required accuracy  $|S_n - S_{n-1}| < 10^{-6}$ .

### 8.7.2 Structure of `while` statement

To find an approximation  $S_n$ , we can then use the `while` statement:

```

while expression
    statements
end

```

Here, the statements are executed as long as `expression` is true.

```

Editor - C:\Documents and Settings\berntl\My Documents\ArchiveDocume...
File Edit Text Cell Tools Debug Desktop Window Help
Stack: Base
- 1.0 + 1.1 x
1 - dS = 1;
2 - n_old = 0;
3 - Sn_old = 0;
4 - %
5 - while dS > 1e-6
6 -     n = n_old + 1;
7 -     Sn = Sn_old + 1/n^2;
8 -     dS = Sn - Sn_old;
9 -     %
10 -    n_old = n;
11 -    Sn_old = Sn;
12 - end
13 - disp(['S = S', num2str(n), ' = ', num2str(Sn)]);
rootscript.m x whileexampl... x
script Ln 2 Col 10 OVR

```

Figure 8.2: Script for finding  $S_n = \sum_{k=1}^n \frac{1}{k^2} = S_{n-1} + \frac{1}{n^2}$  to a prespecified accuracy of  $|S_n - S_{n-1}| < 10^{-6}$ .

### 8.7.3 Example: use of while statement

The following example illustrates how we can compute  $S_n$  such that the accuracy requirement is fulfilled, fig. 8.2. Running this script gives the following answer:

```
>> whileexample
S = S1000 = 1.6439
```

A possible problem with the use of the `while` statement, occurs if we use an erroneous test expression which never is fulfilled: for that case, the while loop will continue forever, and the only way to stop the script execution may be to hit the `Ctrl+C` key.<sup>1</sup>

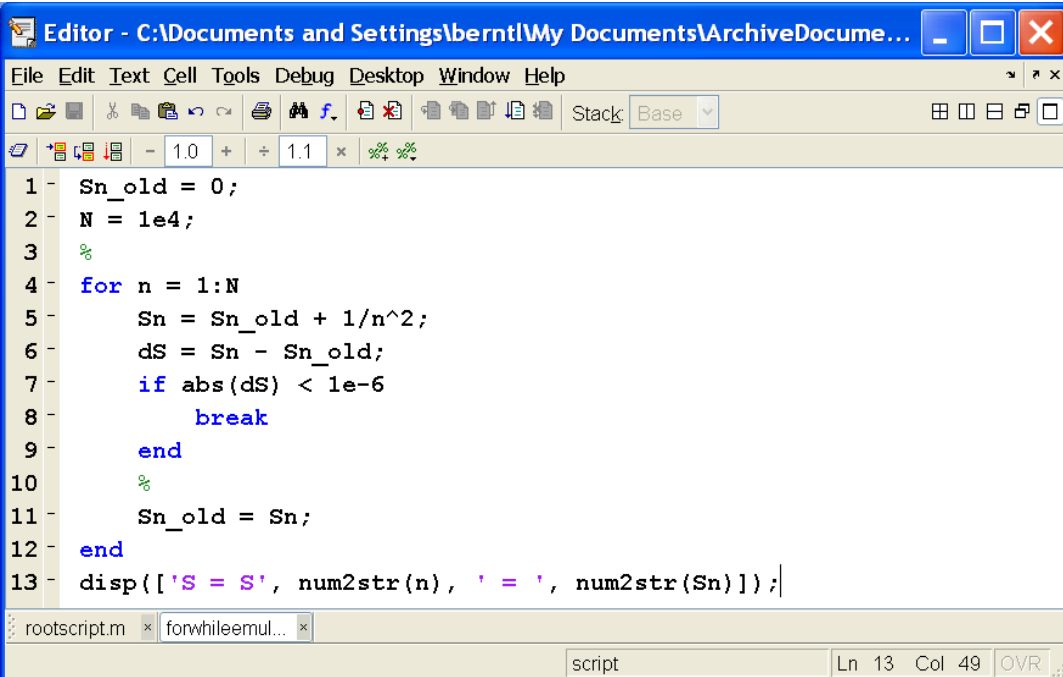
### 8.7.4 Breaking out from for loop

Finally, it should be noted that the same construct can be achieved by combining a `for` loop with an `if` statement, fig. ???. Executing this script leads to:

```
>> forwhileemulate
S = S1000 = 1.6439
```

The disadvantage of using the `for` statement here, is that we do not know in advance how large `N` should be. The advantage of using the `for` statement, is that the loop will not be repeated indefinitely even when we make a mistake in the program code.

<sup>1</sup>`Ctrl+C` is achieved by holding down the `Ctrl` key on the computer keyboard, and while your are doing this, press the `C` key. This is an emergency action only, which *usually* (but not always) works.



The image shows a screenshot of a MATLAB script editor window. The window title is "Editor - C:\Documents and Settings\bernt\My Documents\ArchiveDocume...". The menu bar includes "File", "Edit", "Text", "Cell", "Tools", "Debug", "Desktop", "Window", and "Help". The toolbar contains various icons for file operations and editing. The script content is as follows:

```
1 - Sn_old = 0;
2 - N = 1e4;
3 - %
4 - for n = 1:N
5 -     Sn = Sn_old + 1/n^2;
6 -     dS = Sn - Sn_old;
7 -     if abs(dS) < 1e-6
8 -         break
9 -     end
10 - %
11 -     Sn_old = Sn;
12 - end
13 - disp(['S = S', num2str(n), ' = ', num2str(Sn)]);
```

The status bar at the bottom indicates the current position is "Ln 13 Col 49" and the window is titled "script".

Figure 8.3: Script with for loop, where an if statement is used to break out of the loop.

## Chapter 9

# Writing your own functions

So far, we have used some of Matlab's built-in functions. In Matlab it is very easy to write your own functions if Matlab does not have one that suits your needs.

### 9.1 Example: Distance to horizon

#### 9.1.1 Basic case

If you look out to the sea, the distance  $x_h$  to the horizon depends on how high above the sea level your eyes are, fig. 9.1. Approximating the earth with a perfect sphere and neglecting refraction, the

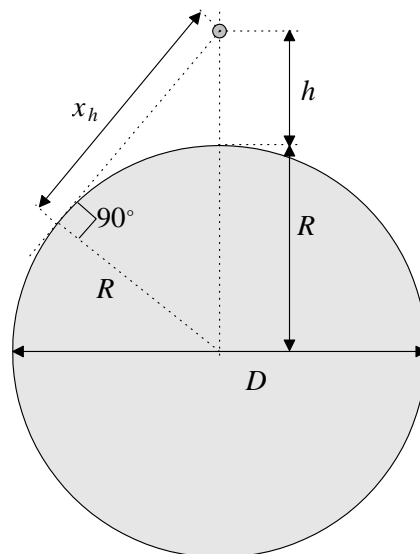


Figure 9.1: Distance  $x_h$  to visible horizon from height  $h$ .  $D$  and  $R$  are the diameter and radius, respectively, of the earth.

Pythagorean law leads to:

$$R^2 + x_h^2 = (R + h)^2 \Leftrightarrow x_h = \sqrt{h(2R + h)} = \sqrt{h(D + h)},$$

where  $h$  is the elevation of your eyes above the sea level and  $D$  is the diameter of the earth.  $x_h$  is the length of the straight line between your eyes and the horizon. Knowing that the earth's diameter  $D \approx 12000$  km, we can create a Matlab function for this. We open a new document in the editor by clicking the leftmost button on Matlab's toolbar and type:

```
function xh = horizon(h)
```

```
% Distance to the horizon
% Usage: xh = horizon(h)
d = 12e6; % Earth's diameter
xh = sqrt(h.*(h+d));
We save this file as horizon.m. Now we can test it:
>> horizon(2)
ans =
    4899
```

We have thus chosen to name the function `horizon`; we thus also save the function file and give it the name `horizon.m`.

Note that we have used element by element operations (`.*`, `./`, `.^`) rather than matrix operations (`*`, `/`, `^`). This ensures the right answer if we want to calculate the function for an array of arguments at once, as we might want to do for plotting:

```
>> hp = 1:1000;
>> plot(hp,horizon(hp))
```

The first comment lines are echoed when we write `help horizon`:

```
>> help horizon
Distance to the horizon
Usage: xh = horizon(h)
```

Maybe we want to be able to find the distance to the horizon on the moon or some other planet. We can easily modify the function so that it takes the planet diameter as a second argument:

```
function xh = horizon2(h,d)
% Distance to the horizon
% Usage: xh = horizon(h,d)
% h : Observer distance above surface
% d : Planet diameter
% xh : Distance to horizon
xh = sqrt(h.*(h+d));
```

Note that Matlab searches for the *file* `horizon2.m` when we write:

```
>> horizon2(2,6e6)
ans =
    3464.1
```

We should thus save the new function in a file `horizon2`. The function name itself (`horizon2`) is not important in itself, but it is considered good practice to let the function name be equal to the file name.

We can make a version that uses the earth's diameter as default if we give only `h` as the argument. As long as only one argument is used, the user will not note the difference from the original `horizon.m`, so we overwrite this file. Below, `nargin` knows the number of input arguments that the user has used:

```
function xh = horizon(h,d)
% Distance to the horizon
% Usage: xh = horizon(h) and xh = horizon(h,d)
% h : Observer distance from surface
% d : Planet diameter Default: 12e6 (Earth)
```

```
% xh : Distance to horizon
if nargin < 2
    d = 12e6; % Gives d a value if the user does not
end
xh = sqrt(h.*(h+d));
```

### 9.1.2 Returning more than one result

We may also return more than one value:

```
function [xh, angle] = horizon3(h,d)
% Distance to the horizon
% Usage: xh = horizon(h) and xh = horizon(h,d)
% h : Observer distance from surface
% d : Planet diameter Default: 12e6 (Earth)
% xh : Distance to horizon
% angle: angle between radius to horizon and to observer (degrees)
if nargin < 2
    d = 12e6; % Gives d a value if the user does not
end
xh = sqrt(h.*(h+d));
angle = atan(xh/(d/2))*180/pi;
```

Test:

```
>> [xh,angle]=horizon3(1e4)
xh =
    3.4655e+005
angle =
    3.3057
```

From an airplane at 10 000 m above Oslo (60° N) you can see most of the way to Trondheim (63.5° N).

We can ask for the second output value by specifying it in the calling statement (otherwise, it is discarded):

```
>> xh=horizon3(1e4)
xh =
    3.4655e+005
```

We can obtain the number of output arguments from within the function. This can save some computer time if some outputs require a lot of calculations. Below, `nargout` knows the number of output arguments that the user has asked for:

```
function [xh, angle] = horizon4(h,d)
% Distance to the horizon
% Usage: xh = horizon(h) and xh = horizon(h,d)
% h : Observer distance from surface
% d : Planet diameter Default: 12e6 (Earth)
% xh : Distance to horizon
% angle: angle between radius to horizon and to observer (degrees)
if nargin < 2
    d = 12e6; % Gives d a value if the user does not
end
```

```
xh = sqrt(h.*(h+d));
if nargin > 1
    angle = atan(xh/(d/2))*180/pi;
end
```

It is good practice to test that the arguments given by the user are valid:

```
function xh = horizon(h)
% Distance to the horizon
% Usage: xh = horizon(h) and xh = horizon(h,d)
if h < 0
    error('Argument must be positive')
end
xh = sqrt(h.*(h+d));
```

### 9.1.3 Exercises

**Exercise 9.1** The Fibonacci numbers  $f_k$  are defined by:

$$\begin{aligned} f_1 &= 1 \\ f_2 &= 1 \\ f_{k+1} &= f_k + f_{k-1}, \quad k > 2. \end{aligned}$$

Thus the first 8 Fibonacci numbers are: 1, 1, 2, 3, 5, 8, 13, 21.

- Write a function `fib(n)` that returns Fibonacci number `n`.
- Let the function return the vector of numbers up to and including  $f_n$  in an optional second return variable. Also make the function able to take a two-element array as an optional second input variable. The two elements of the vector should be used as alternative starting values. E.g.: `[fn,s]=fib(6,[8,3])` should return `fn = 39` and `s = [8,3,11,14,25,39]`. ■

**Exercise 9.2** When calculating the distance to the horizon  $x_h$  from an elevation  $h$ , the distance from the “shore” to the horizon will also be approximately  $x_h$  when  $h \ll D$ .

- Write a function that computes  $h$  when  $x_h$  is known.
- Approximately, how high above the ground do you need to be to be able to see from one end of Lake Mjøsa<sup>1</sup> to the other end? Assume that Lake Mjøsa is 120 km long.
- Approximately, how high above the ground do you need to be to be able to see from the southern shore of Norway to the northern shore of Denmark? Assume that the distance is 150 km.
- A possible approximative expression for  $h$  is

$$h \approx \frac{1}{D}x_h^2.$$

Write another function for computing  $h$  based on this approximation. Compare the expression with the exact expression for a number of distances  $x_h$ , and suggest how large  $x_h$  can be before the simple formula  $h \approx x_h^2/D$  becomes inaccurate. ■

<sup>1</sup>Lake Mjøsa is the largest inland lake in Norway.



## 9.2 Workspaces and variables

Like most programming languages, Matlab uses several *workspaces* or *contexts*. As long as you issue commands in the *command window* or use *scripts*, you work in the *base workspace*. All variables that you have created are available to you until you clear them or until the session is terminated. Inside a function, you do not have access to the variables in the base workspace. This function will never work, even if you define the variable `a` before calling the function:

```
function s = testsum(x)
s = x + a;
```

We can try:

```
>> a = 4;
>> s = testsum(3)
??? Undefined function or variable 'a'.
Error in ==> C:\HiT\Matlabkurs\testsum.m
On line 2 ==> s = x + a;
```

Inside the function, Matlab does not “see” the `a` from the base workspace. Information to the function is passed only through the function’s arguments. Likewise, variables that you create or change inside a function have no direct influence on the base workspace. Information can pass back only through the returned variables. Only the sequence of input and output variables is significant. Which variable names you use have no effect.

### Example

Consider these examples:

```
function s = vartest(x,y)
a = 2;
s = x + a*y;
```

We test the function:

```
>> a = 4;
>> x = 1; y = 2;
>> z = vartest(x,y)
z =
    5
>> a
a =
    4
>> z = vartest(y,x)
z =
    4
```

We see that the assignment `a=2`; in the function workspace does not affect the variable `a` in the base workspace. Furthermore, interchanging the sequence of `x` and `y` in the call gives two different results. In fact, we could have used some totally different variable names in the call:

```
>> arg1 = 1; arg2 = 2; z = vartest(arg1,arg2)
z =
```

```

5
>> z=vartest(1,2)
z =
5

```

Assigning the answer to the variable `s` in the function's workspace has no effect in the base workspace:

```

>> s
??? Undefined function or variable 's'.

```

Understanding this segregation between workspaces is essential to understanding how functions work. ■

### 9.3 Loops vs. matrix operations

Say we want to calculate the inner product of two vectors  $a$  and  $b$ . The inner product is defined as

$$a \cdot b = \sum_{i=1}^n a_i b_i,$$

where  $n$  is the number of elements of  $a$  and  $b$ . Say we represent both  $a$  and  $b$  as Matlab column vectors. Then we can calculate the inner product by using the formula above, but we can also exploit the fact that the inner product is the same as the matrix product between the transpose of  $a$  with  $b$ . The latter gives a much simpler expression:

```

% Inner product example
a = [3;6;-2;1];
b = [-3; 0; 4; 1];
% Loop formulation
s1 = 0;
for k = 1:length(a)
    s1 = s1+a(k)*b(k);
end
% Matrix/vector expression:
s2 = a'*b;

```

In older versions of Matlab, the matrix expression was much quicker than the loop form, but from Matlab v. 6.5, the loop runs almost as fast. You should use the formulation that results in the most easily readable code. Very often, this is the matrix form.

**Exercise 9.3** Element  $C_{ij}$  in the product  $C = A \cdot B$  is defined as  $C_{ij} = \sum_{k=1}^n A_{ik} B_{kj}$ .

- Write function `C = matmult(A,B)` that multiplies two square matrices without using Matlab's matrix operations. Test that the matrices are indeed square and of the same size, and abort using the error statement if this is not the case.

(**Hint:** The value of `C(i,j)` is the inner product of `A(i,:)` with `B(:,j)`).

- Using two random matrices (`A = rand(3); B= rand(3);`), compare the output to `A*B`. ■

## Chapter 10

# Advanced use of the Matlab editor

### 10.1 Errors and the debugger

I really hate this damn machine.  
I wish that they would sell it.  
It never does quite what I mean,  
but only what I tell it!  
— The programmer's lament

We inevitably make mistakes when we write functions and scripts in Matlab. Some are *syntax errors*: we write something that does not make sense to Matlab, no matter what the variables contain. Matlab spots these errors before it tries to execute your code. We may for instance forget the multiplication operator `*`:

```
>> a = 2b
??? a = 2b
|
Error: Missing operator, comma, or semicolon.
```

Matlab tells you what it believes is wrong and marks the spot where it had to give up.

Syntax errors are usually relatively easy to correct. However, even when there are no syntax errors in the Matlab code, there may be logical errors or algorithmic errors which occur during run-time and lead to unintended results or even program crash — these are not always easy to find. Run-time errors depend on the contents of your variables. When the running of the Matlab code terminates, Matlab reverts to the base workspace, and we no longer have access to the variables in the function workspace. This makes it difficult to find out what went wrong.

One simple remedy is to have the function print intermediate results, either by using the `disp` statement, or simply by *omitting the semicolon* at the end of selected statements. However, these are not good strategies for finding errors, because when the code has been debugged and errors fixed, we need to remove the `disp` commands and insert semicolons; this may be a major job.

A more advanced solution is to use the *debugger*. The debugger is a powerful tool to help you understand what is going on inside your functions, fig. 10.1. In order to activate the debugger, a *breakpoint* must have been defined in the Matlab file, see fig. 10.1. With one or more breakpoints in the Matlab file, the debugger can be started as follows:

- if you want to debug a script file, either click on the (Save and) Run icon in the editor, or type the name of the script file in the command window;
- if you want to debug a function file, either call the function (with appropriate arguments) from the command line, or run a script that calls the function.

The debugger allows you to stop execution inside a script or function. You can stop at a specified line, or when a warning or error is triggered. A breakpoint marks one or more lines of code where you

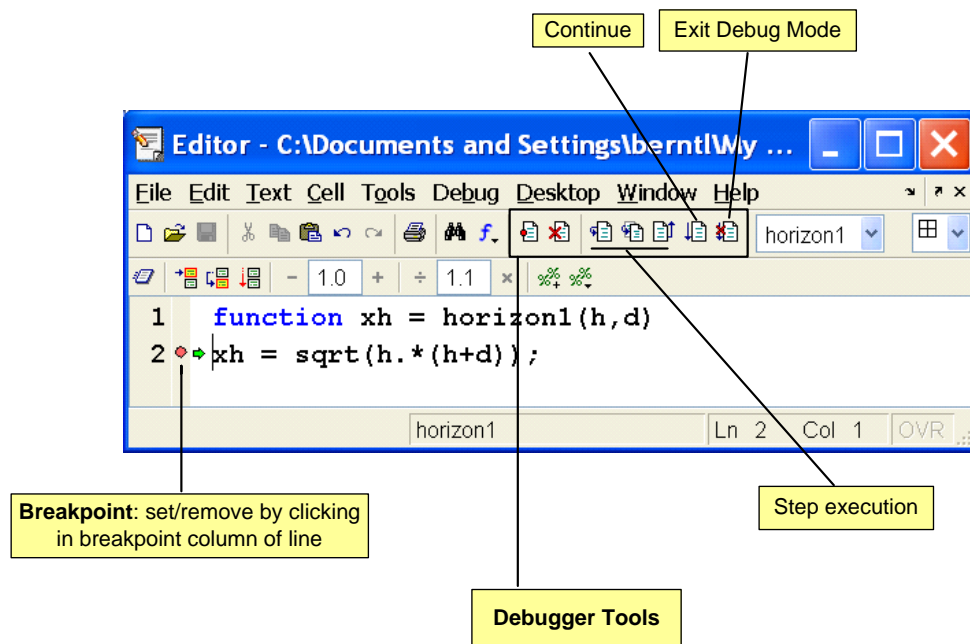


Figure 10.1: Some debugger tools.

want execution to pause. You define the breakpoints by clicking the button showing a red circle on the edge of the paper, or from the Breakpoints menu.

When the debugger halts execution, you will see a special prompt in the *command window*:

```
K>>
```

At the same time: in the *editor window*, a green arrow points to the line about to be executed, see fig. 10.1. You may now examine the contents of variables. This can be done by letting the cursor hover over a variable in the editor window — the value of the chosen variable will then appear in a yellow box over the editor window. Or you can browse the workspace browser and use the array editor. Or you can type commands in the command window:

```

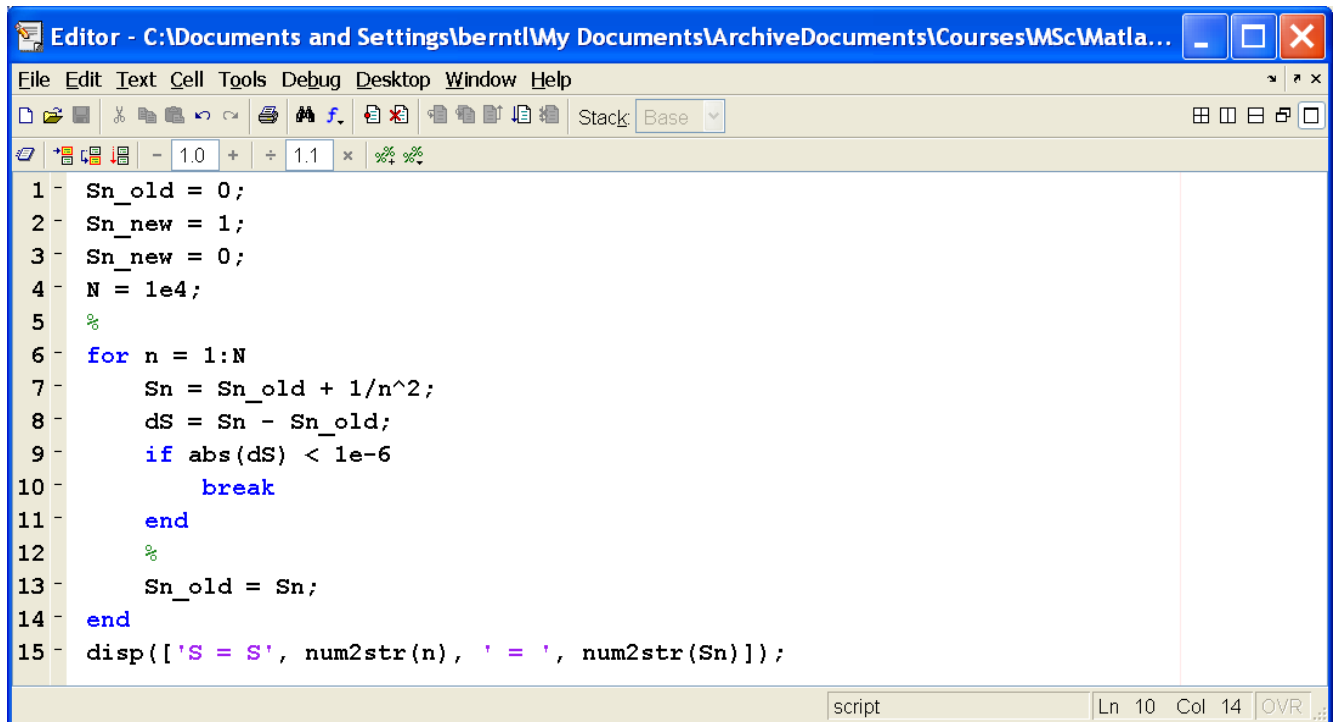
>> horizon1(1000,64e3)
K>> h
h =
    1000
K>> d
d =
    64000
K>> h*d
ans =
    64000000

```

You may *step* from line to line or into subfunctions, using one of the three Step execution icons, see fig. 10.1. All debugger buttons show yellow information balloons when you let the cursor hover over them, so finding your way around the debugger is quite intuitive.

To get out of the debugger, click the Exit Debug Mode icon (fig. 10.1), or enter the command

```
K>> dbquit
```



```

1 - Sn_old = 0;
2 - Sn_new = 1;
3 - Sn_new = 0;
4 - N = 1e4;
5 - %
6 - for n = 1:N
7 -     Sn = Sn_old + 1/n^2;
8 -     dS = Sn - Sn_old;
9 -     if abs(dS) < 1e-6
10 -         break
11 -     end
12 -     %
13 -     Sn_old = Sn;
14 - end
15 - disp(['S = S', num2str(n), ' = ', num2str(Sn)]);

```

Figure 10.2: Example of Matlab code with “lint”: statement `Sn_new = 1;` is superfluous and should have been removed.

in the command window.

Finally: the **Debug** menu in the Editor window should also be consulted. This menu contains the same operations as can also be carried out using the debugger icons.

The debugger is useful for understanding runtime errors, but even more so for finding out why your Matlab code does not return the answer that you expected. If you are serious about using Matlab as a tool, learning to use the debugger is mandatory.

## 10.2 De-linting the code

Developing complicated Matlab code in the Matlab editor will to some degree involve some trial and error: we try with some code, then change our mind and modify the code. In this development, it is very common that we forget to remove unnecessary code, or that we forget to define some variables. This problem is illustrated in the (not very realistic) example of fig. 10.2.

We can use the M-Lint tool to find such code: type **Tools/Check Code with M-Lint**, and the report in fig.10.3 is provided.

As the report in fig. 10.3 indicates, the statement on line 2 of the code in fig. 10.2 should be removed. (Clearly, Matlab can not guess that statement on line 3 is without interest.)

## 10.3 Structuring the file

From Matlab 7 of, it is possible to structure Matlab code in the editor by defining text cells. Essentially, a text cell is initiated by putting the symbol `%%` (double `%` character) in the first position of a line. The cell ends on the line preceding the next `%%` symbol. After the `%%` symbol, a space should be inserted, followed by a descriptive text. The Matlab editor marks a cell by framing it with a yellow box: when you put the cursor in a cell, the frame is shown. In order for this to work, the *Cell Mode* must have been *Enabled*, see the **Cell** menu of the Matlab editor. Figure 10.4 illustrates a simple script where cells have been defined.

A simple use of such cells is use it for browsing a long Matlab file: click on the **Show cell titles** icon, fig. 10.5

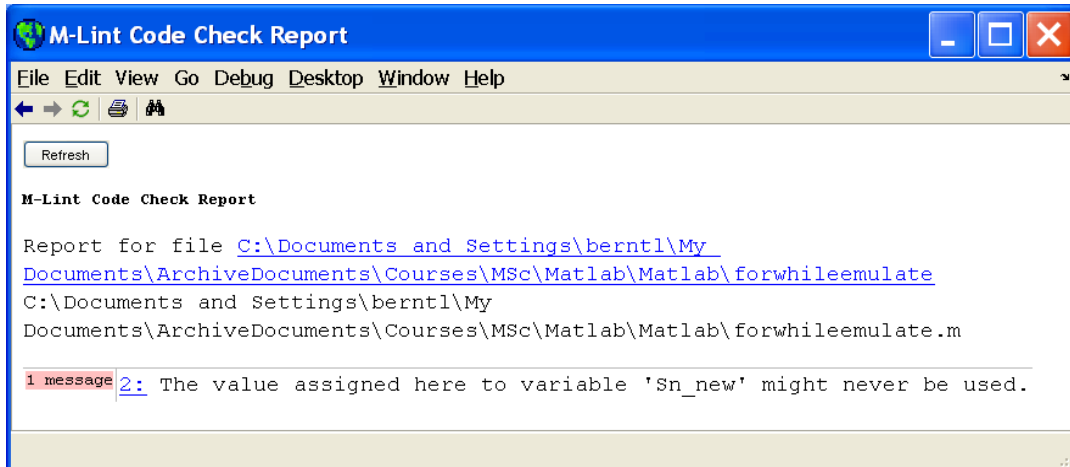


Figure 10.3: Report from checking the code of fig. 10.2 with M-Lint.

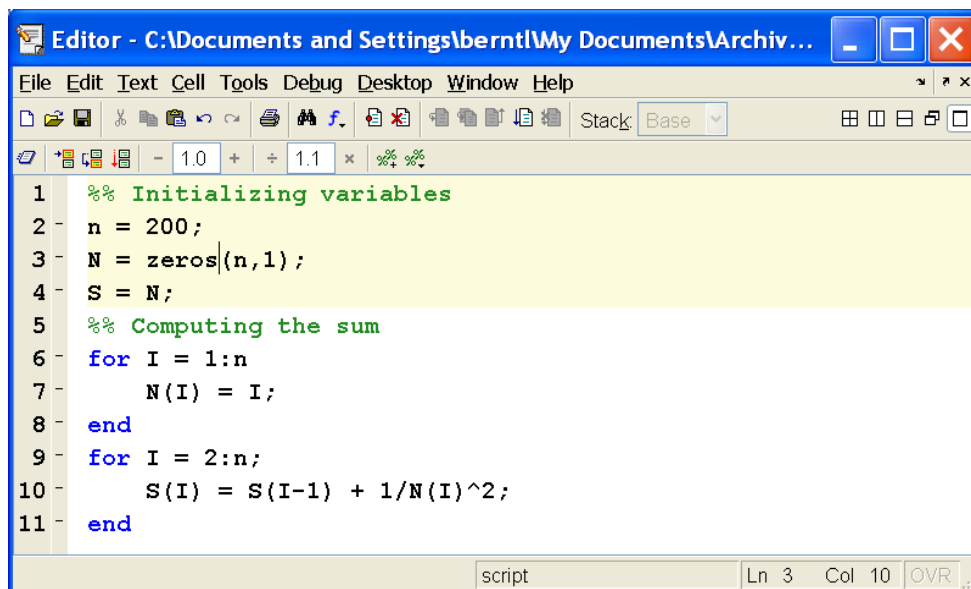


Figure 10.4: Example of text cells in the Matlab editor.

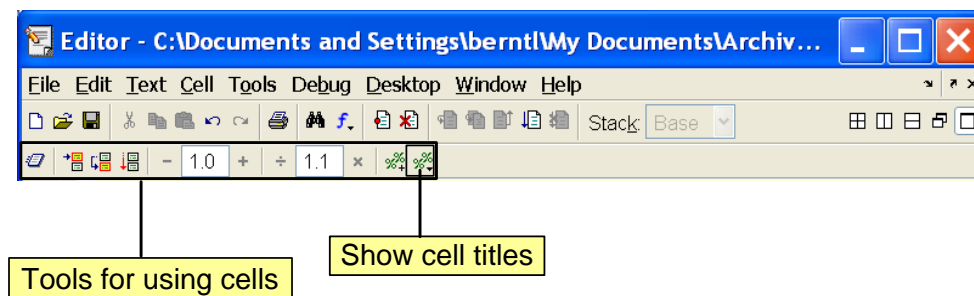


Figure 10.5: Tools for taking advantage of cells in Matlab files. The `Show cell titles` tool can be used for browsing through the cells: clicking this tool, gives a list of the cells.

and a list of the cell headlines (descriptive text following `%` symbol) pops up. By clicking on the relevant cell title, the cursor is positioned in the cell head. This allows for easy navigation of large Matlab files.

However, cells can be used for much more. It is e.g. possible to insert markup commands in the Matlab file, e.g. making text boldface, etc., and also markup text using  $\text{\TeX}$  syntax so that it can be interpreted as mathematics. The Matlab file can then be published as an HTML document, a  $\text{\LaTeX}$  document, a Word document, etc., in such a way that the published document documents the code with headlines, section heads, with computed results, etc.





# Chapter 11

## Data structures in Matlab

Historically, the matrix was the only data type in Matlab. Vectors and scalars are special cases of the general matrix. Since Version 5, some important new data structures have arrived. One is the multi-dimensional array, which just extends the matrix to more than two dimensions. More important newcomers are the structure and the cell array.

### 11.1 Structures

A *structure* is a data structure that can hold diverse data types, not necessarily numbers, and with named “data containers” called *fields*, similar to a record with fields in a database. Example:

```
>> tank1.diameter = 0.5;
>> tank1.height = 0.4;
>> tank1.type = 'cylinder';
```

This is useful if we want to collect several items of information about some object. We can now tell e.g. a function to calculate the volume: all it needs to know about `tank1` by using the *tank1 structure* as function argument:

```
function V = volume(a)
switch a.type
    case 'cylinder'
        V = pi*a.diameter^2*a.height/4;
    case 'box'
        V = a.length*a.width*a.height;
    case 'sphere'
        V = pi*a.diameter^3/6;
    otherwise
        error('unknown type')
end
```

Here we have used a new flow control structure, `switch ... case ... otherwise ... end`.

```
>> volume(tank1)
ans =
    0.025
>> tank2.type = 'sphere';
>> tank2.diameter = 2;
>> volume(tank2)
ans =
    4.1888
```

*Structures* may be used to limit the number of arguments needed by a function. This can be achieved by collecting all parameters in a parameter structure.

## 11.2 Cell arrays

A *cell array* is a data structure that can hold diverse data types, not necessarily numbers. The File Import Wizard returns text and data as cell arrays, as we saw in Part I of the course. We can collect items into a cell array by enclosing them in curly brackets:

```
>> A = rand(4);  
>> C = {A sum(A) prod(prod(A))}  
C =  
[4x4 double] [1x4 double] [8.7912e-006]
```

We access the contents of the individual cells by curly brackets:

```
>> b = C{2}  
b =  
2.1251 1.8055 2.8766 2.8019
```

The Matlab Help contains much more information about structures and cell arrays.

## Chapter 12

# Handle graphics

As you have seen, simple plotting in Matlab is quite easy. At the same time, Matlab graphics is very flexible, and you have great freedom to get the exact graphics that you need. Here, we will demonstrate how to modify some properties of graphs. You can find out more by consulting Matlab's help system.

We consider the plotting of two Taylor series for  $\cos x$ , and compare them to the exact function:

$$\cos x \approx \begin{cases} T_1 = 1 - \frac{1}{2}x^2 + \frac{1}{24}x^4 \\ T_2 = -(x - \frac{\pi}{2}) + \frac{1}{6}(x - \frac{\pi}{2})^3 \end{cases} ,$$

To make sure that we can distinguish between the three lines, we use markers for  $T_1$  and  $T_2$ :

```
x = linspace(0,pi/2,25);
f = cos(x);
T1 = 1 - x.^2/2 + x.^4/24;
T2 = -(x-pi/2)+(x-pi/2).^3/6;
figure(1)
h = plot(x,f,x,T1,'*',x,T2,'+');
```

The result is displayed in fig. 12.1.

Note the form of the plot command. The variable `h` now contains *handles* to each of the three curves. In fact, `h` is an *object* with certain *properties*, and we can get a list of all of the properties for the first curve by typing `get(h(1))`:

```
>> get(h(1))
```

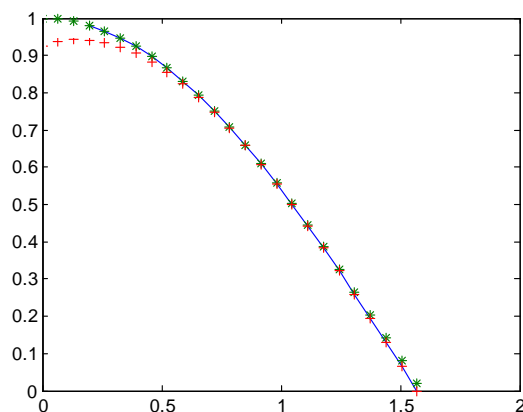


Figure 12.1: Comparison of  $\cos x$ , and two Taylor expansions of  $\cos x$ .

```

        Color: [0 0 1]
        EraseMode: 'normal'
        LineStyle: '-'
        LineWidth: 2
        Marker: 'none'
        MarkerSize: 6
        MarkerEdgeColor: 'auto'
        MarkerFaceColor: 'none'
        XData: [1x25 double]
        YData: [1x25 double]
        ZData: [1x0 double]
        BeingDeleted: 'off'
        ButtonDownFcn: []
        Children: [0x1 double]
        Clipping: 'on'
        CreateFcn: []
        DeleteFcn: []
        BusyAction: 'queue'
        HandleVisibility: 'on'
        HitTest: 'on'
        Interruptible: 'on'
        Selected: 'off'
        SelectionHighlight: 'on'
        Tag: ''
        Type: 'line'
        UIContextMenu: []
        UserData: []
        Visible: 'on'
        Parent: 150.0048
        DisplayName: ''
        XDataMode: 'manual'
        XDataSource: ''
        YDataSource: ''
        ZDataSource: ''

```

We can now use the handles to change the properties of the curves. For instance, we may increase the line width of the first curve by;

```
>> set(h(1),'linewidth',2)
```

(note: Matlab is not case sensitive when it comes to property names).

The properties can also be modified interactively, as we have seen in Section 3.2.2 p. 44. Using handles enables us to specify the desired look from our scripts and functions. Once we get the script right, we get the right look every time. Furthermore, using handles makes it possible to automate the look in repetition loops.

As an example, we may even modify the data to be plotted in the graph. This enables the creation of *animations*. Example:

```

% animation example
x = linspace(0,1,100);
y = 0*sin(x*pi);
figure(2);
h=plot(x,y);
set(h,'erasemode','xor') % Redraw only curve, not full figure
axis([0 1 -1 1])
for i = 1:1000

```

```
pause(0.01); % Wait 0.01 seconds to slow things down
y = sin(i/100*pi)*sin(x*pi);
set(h,'ydata',y) % Enter new y-data
drawnow % Update graph
end
```

For more information on graphics and graphics properties, see the interactive help. (Matlab help or Full product help, from the help menu.)

**Exercise 12.1** Make a plot of  $\sin x$  and  $\cos x$  on  $[0, \pi]$ .

1. Write a script that changes the line style of the first curve to `'-.'` and the second to `':'`, and the colors of both to black (`'k'`). You can get handles to all line objects of the current axes by the command:

```
h = findobj(gca,'Type','line')
```

2. Repeat the exercise, this time using the Figure Properties Editor. This is accessible from the Edit menu on the figure window. ■

This kind of modification may be useful if you want to copy your paper using black and white copiers. The Matlab Help System contains much more information on how to tailor your graphs to suit your needs.



# Chapter 13

## Problems

In the following problems, use the debugger and Matlab's help facilities to sort out the troubles you are sure to encounter:

**Problem 13.1** An integer  $k$  is called a proper divisor of  $n$  if  $0 < k < n$ , and if the fraction  $n/k$  is an integer. Write a function `propdiv(n)` that finds all proper divisors of the positive integer  $n$ . A simple algorithm for this is to test every integer  $k \leq n/2$  to find those that give no remainder when  $n$  is divided by  $k$ . Use the standard Matlab function `rem` to do this. ■

**Problem 13.2** Write a function `perfect(n)` that finds the first perfect number  $> n$ . A perfect number is the sum of all its proper divisors. The first three perfect numbers are 1, 6 ( $= 1 + 2 + 3$ ), and 28 ( $= 1 + 2 + 4 + 7 + 14$ ). ■

The following problems relate to graphics in Matlab.

**Problem 13.3** Let `t = linspace(0,2*pi)`. Let  $x_1 = \sin(t)$ ,  $y_1 = \cos(t)$ ,  $y_2 = \sin(2*t)$ ,  $y_3 = \cos(3*t)$ , and  $y_4 = \sin(4*t)$ . Create a figure containing a 2 by 2 subplot array. In subplot 1, plot  $y_1$  vs.  $x_1$ . In subplot 2, plot  $y_2$  vs.  $x_1$ , and so on. Use the interactive plotting tools to change line colors, add markers, and stretch or move the subplot axes. You may also change background colors and make other modifications as you like. ■

**Problem 13.4** Use the `generate m-file` menu choice in the figure's `file` menu to save the figure set-ups as an m-file function, which you may call `createfigure.m`. Use `createfigure` to make a new figure identical to the first, only now the  $x$ -axis data for each figure is  $x_2 = \cos(t)$ . ■

**Problem 13.5** Modify `createfigure` to add `xlabels` and `ylabels` to each subplot. The `ylabel` for subplot 1 may be "`cos(t)`" and so on. ■

**Problem 13.6** In this problem, do *not* use the interactive plotting tools. Make a new figure, this time showing  $y_1$  through  $y_4$  vs.  $x_1$  in the same axes. Let the plot command return handles to each graph, and use the handles to add markers and modify colors and line types of each graph. ■

To solve the next problem, it is necessary to study Appendix D.

**Problem 13.7** Write the Matlab commands: `x=0:0.25:2; y=0:0.5:5; [X,Y] = meshgrid(x,y)`. Examine `X` and `Y` to see the effect of the `meshgrid` function. Let  $Z = X.^2 + \sin(Y)$ . Plot  $Z$  as a function of  $X$  and  $Y$  using the following commands: `surf`, `mesh`, `contour`, `contourf`. Write labels on the axes. Rotate and move the graphs using the various interactive tools, including those on the camera toolbar. ■





## **Part III**

# **Becoming a Matlab Master**



# Chapter 14

## The once and future Matlab Master

### 14.1 Revisiting Part II

#### Numerical methods with Matlab

- Repetition of functions from Part I, and some new functions.

#### Writing your own functions

- Introduction through example: distance to horizon
  - Basic use of functions
  - Returning more than one result
- Workspaces and variables: local value of variables within functions
- Relational and logical expressions
- Program flow control
- Loops vs. matrix operations
- Errors and the debugger

#### Data structures in Matlab

- Structures
- Cell arrays

#### Handle graphics

- Figures and handles
- Modification of figures through handles and setting properties
- Animation

### 14.2 Overview of new material

- Functions and function handles
- Calling functions through their handles
- Anonymous functions
- Function functions

- Writing your own Newton solver
- Generalizing the newton solver
- ODE solvers
  - Writing your own Euler solver
  - Generalizing the Euler solver
  - Built-in Matlab solvers

# Chapter 15

## Functions and function handles

### 15.1 Function handles

A *function handle* is a Matlab *value* which can be used for calling a function *indirectly*. Basically, the function handle is an address to the function.

A function handle can be created as follows:

```
>> mysin = @sin
mysin =
    @sin
```

Here, `sin` is a function name — the built-in Matlab function to compute the sine of the argument. By preceding the function name by the handle indicator `@`, `@sin` is the handle for the sine function, and points to the built-in sine function. Finally: with the statement `mysin = @sin`, we *declare* variable `mysin` to be the function handle `@sin`.

From Matlab 7 on, we can use the variable name `mysin` containing the function handle of the `sin` function, just as if it was the `sin` function:

```
>> x = linspace(0,2*pi);
>> plot(x, mysin(x))
```

In older versions of Matlab, other and more arcane techniques must be used.<sup>1</sup>

The function handle is a mapping to the memory address of the function. We can treat function handles as any variable, e.g. copying the handle:

```
>> myothersin = mysin;
```

Next, we can use `myothersin` just as we used `mysin`, and

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

will give the same result as `plot(x, mysin(x))`. Function handles can e.g. be stored in cell arrays:

```
>> mysin = @sin;
>> mycos = @cos;
>> mytrig = {mysin, mycos}
```

---

<sup>1</sup>In older versions of Matlab, we would have to write the `plot` statement below as `plot(x, feval(mysin, x))`.

```
mytrig =
    [sin]    [cos]
```

Then we can use `mytrig` to plot the sine function:

```
>> x = linspace(0,2*pi);
>> plot(x, mytrig{1}(x))
```

A very important feature of the function handle, is that we can transfer the handle as an input argument to a function, and then use the function handle from within the new function — because of this, we talk about function handles as “function functions”.

## 15.2 Function functions

We have already seen examples of many built-in Matlab functions. With some of the built-in Matlab functions, the user has to supply an additional function. As an example, Matlab has built-in functions for performing numerical integration, e.g. the `quad` function<sup>2</sup>. But the user has to supply the function that he/she wants to integrate. In the simplest case, the arguments of `quad` are: (i) the function handle for the function we want to integrate, (ii) lower integration boundary, and (iii) upper integration boundary.

Suppose we want to find

$$\int_0^{\pi} \sin x dx$$

by numeric integration. This can be done as follows:

```
>> quad(@sin,0,pi)

ans =

    2.0000
```

The point here is that Matlab function `quad` is a general function that can integrate any function numerically, and we, the user must supply the function handle for the function which we want to integrate — `@sin` in this case. Because we supply the function handle `@sin`, Matlab function `quad` can internally make calls to the specified function `sin`.

Similarly, we can compute

$$\int_0^{\frac{\pi}{4}} \cos x dx$$

as follows:

```
>> quad(@cos,0,pi/4)

ans =

    0.7071
```

Some Matlab functions that require a user specified function are shown in Table 15.1.

We have previously seen how we can define our own functions in `m`-files, where the function name and the file name coincide. We can then transfer the reference to such functions using the function handle technique, e.g. to the built-in functions in Table 15.1. It is also possible to define functions in a simpler way, without writing specific `m`-files for them. Such functions are denoted *anonymous functions*.

<sup>2</sup>Numerical integration is often called *quadrature*.

Table 15.1: Some built-in Matlab functions that require a function reference (function handle) as input argument.

Function	Comment
quad, quadl	Numerical integration, quadrature
fzero	Find a zero of a function of one variable
fminbnd	Find a minimum of a function of one variable
fminsearch	Find a minimum of a function of several variables
fplot	Plot a function
ode23, ode45	Solve an ODE system
ode15s	Solve an ODE or DAE system

### 15.3 Anonymous functions

Anonymous functions give you a quick way to define functions without writing m-files. In mathematics and in computer algebra systems such as Maple, Mathematica, MuPAD, etc., we can define functions as follows:

$$f : x \rightarrow y.$$

Here, the  $f$  before the colon  $:$  is the name of the function,  $x$  is the input argument, and  $y$  is the mapping of  $x$ . As an example, we can define a function named  $g$  as

$$g : x \rightarrow \sin^2 x.$$

Figure 15.1 illustrates how such a function definition can be carried out in a Computer Algebra System such as MuPAD.

In Matlab<sup>3</sup>, we can define such a function using the following syntax:

```
>> g = @(x) (sin(x)).^2;
```

Here, we notice that we have prepared  $g(x)$  for accepting array inputs  $x$ . Then we can compute  $g(x)$  for selected values of  $x$  as follows:

```
>> x = linspace(0,2*pi,10);
>> g(x)
ans =
  Columns 1 through 6
         0    0.4132    0.9698    0.7500    0.1170    0.1170
  Columns 7 through 10
    0.7500    0.9698    0.4132    0.0000
```

Similarly, we can plot  $g(x)$  as follows:

```
>> plot(x,g(x))
```

Let us also consider another example:

$$\text{sinc} : x \rightarrow \frac{\sin x}{x}.$$

Using Matlab, we find:

```
>> sinc = @(x) sin(x)./x;
```

---

<sup>3</sup>Matlab 7 and later versions.

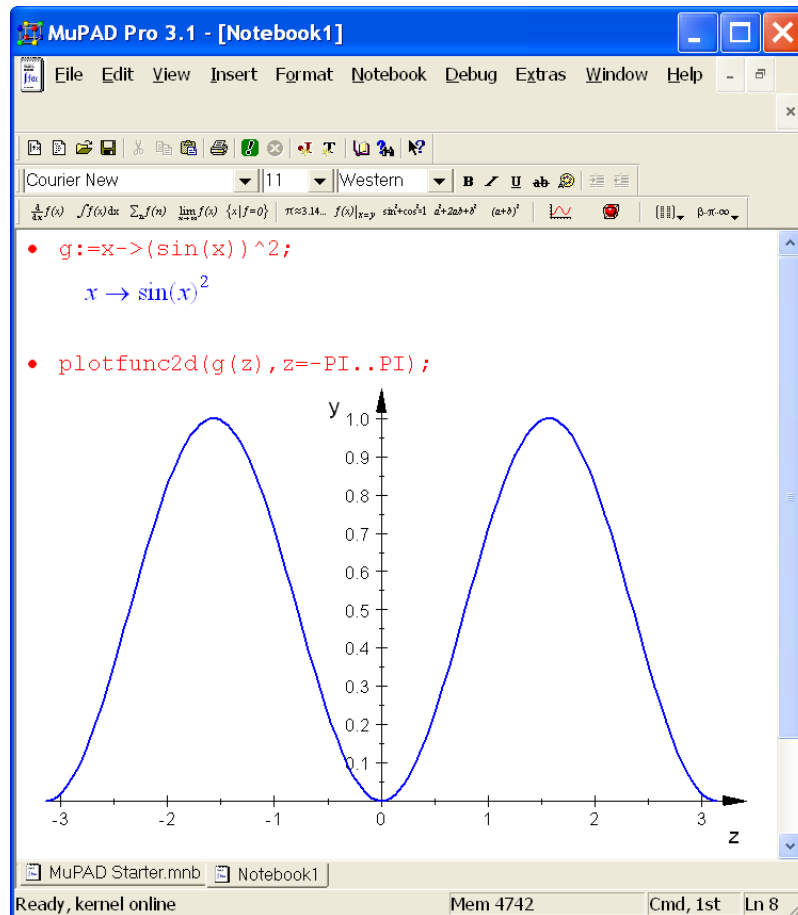


Figure 15.1: Definition of function  $g : x \rightarrow \sin^2 x$  in Computer Algebra System (CAS) MuPAD. Note in particular that the mapping  $g$  is independent of what we choose to name the *argument*.



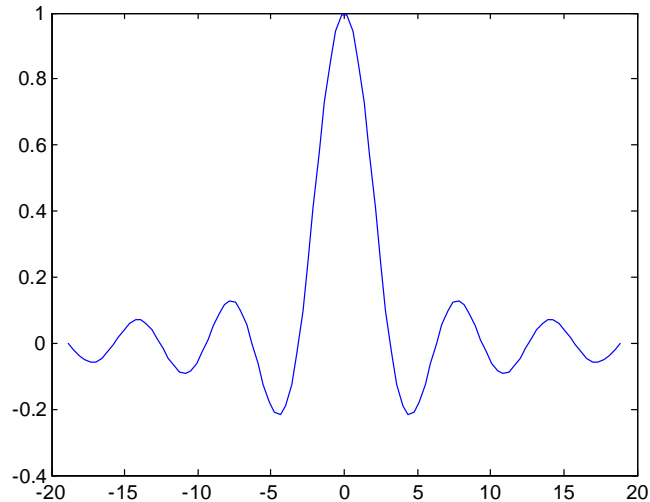


Figure 15.2: Plot of  $\text{sinc } x = \frac{\sin x}{x}$ ,  $x \in [-6\pi, 6\pi]$ .

Here, we have prepared our `sinc` function to accept array input argument `x`. We can now use the `sinc` function handle as follows:

```
>> x = linspace(-6*pi,6*pi);
>> plot(x, sinc(x))
```

to find the well known plot of the  $\text{sinc } x$  function, fig. 15.2.

We can also use function handles of anonymous functions as input arguments to the functions in Table 15.1. As an example, we could consider the problem of computing

$$\int_0^{6\pi} \text{sinc } x dx.$$

In the simplest case, the arguments of `quad` are: (i) a function handle for the function we want to integrate, (ii) lower integration boundary, and (iii) upper integration boundary. We have previously done this for the  $\sin x$  function, and in principle, we could carry out a similar integration with the anonymous function handle `sinc` in the interval  $x \in [0, 6\pi]$ :

```
>> quad(sinc, 0, 6*pi)
Warning: Divide by zero.
> In @(x) sin(x)./x
   In quad at 62
ans =
    1.5180
```

We see that Matlab has a problem since  $\sin x/x$  is not handled very well at  $x = 0$ . This illustrates a problem with anonymous functions. The general syntax is `@(args) expr`, where `args` is a list of comma-separated input arguments, and `expr` is a one line Matlab expression which depends on the input arguments. In the case of the  $\text{sinc } x$  function, we would really like to have more than one Matlab expression in the definition, e.g. `@(x) if x == 0, 1; else, sin(x)/x; end`; this is, however, more than one expression, hence is not a valid expression for the anonymous function.

One more example of anonymous functions: we have in previous chapters searched for the root of  $\sin x = 0.5$ . We can find this root by combining the idea of anonymous functions with function `fzero` in Table 15.1, specifying 1 as the initial guess of the root  $x$ :

```
>> func = @(x) sin(x) - 0.5;
>> fzero(func,1)
ans =
    0.5236
```

As indicated, anonymous functions can be functions of more than one argument:

```
>> sin2 = @(x,y) sin(x)*sin(y);
>> sin2(1,2)
ans =
    0.7651
```

## 15.4 Function functions: motivating example

### 15.4.1 Implementation of Newton method

We want solve the equations  $\sin x = 0.5$ , using Newton's method. We use the following code and store it as a script in file `newton0.m`:

```
x0 = 1;
xi = x0;
fi = sin(xi)-0.5;
Ji = cos(xi);
while abs(fi) > 1e-10
    xip1 = xi-fi/Ji;
    disp(['xi = ', num2str(xi)]);
    xi = xip1;
    fi = sin(xi)-0.5;
    Ji = cos(xi);
end
disp(['sin(xi) - 0.5 = ', num2str(sin(xi)-0.5)]);
```

Here is a brief description of how the Newton method works. We want to solve the equations

$$f(x) = 0,$$

where in the example case above,

$$f(x) = \sin x - 0.5.$$

Here are the steps in the Newton method:

**Algorithm 1** *Newton method:*

1. Let  $x^*$  denote the correct solution such that  $f(x^*) = 0$ ;  $x^*$  is currently unknown. We want to find an approximation  $x_\infty$  of  $x^*$  through iteration.
2. First, we have to guess an initial value  $x_0$ . The closer  $x_0$  is to the correct solution  $x^*$ , the faster we will find our approximate estimate  $x_\infty$  of  $x^*$ . Set counter  $i = 0$ .
3. Next, we find a first order Taylor series approximation of  $f(x)$  in the point  $x_i$ :

$$f(x) \approx f(x_i) + \left. \frac{df(x)}{dx} \right|_{x=x_i} (x - x_i).$$

4. We now let the next iterate  $x_{i+1}$  be defined by  $f(x_{i+1}) \approx 0$ :

$$\begin{aligned} 0 &= f(x_i) + \left. \frac{df(x)}{dx} \right|_{x=x_i} (x_{i+1} - x_i) \\ &\Downarrow \\ x_{i+1} &= x_i - \frac{1}{\left. \frac{df(x)}{dx} \right|_{x=x_i}} f(x_i). \end{aligned}$$

5. Hopefully, by iterating infinitely many times,  $x_\infty \approx x^*$ . Since we do not have infinitely much time, we need to terminate the approximation after a finite number of approximation. We thus need tests to see whether we are close to  $x_\infty$ . One possible tests is:

$$|f(x_{i+1})| \leq \varepsilon_f,$$

and if this test is satisfied, we set

$$x^* \approx x_{i+1},$$

and terminate the iteration. But other tests are possible, e.g.:

$$|f(x_{i+1}) - f(x_i)| \leq \varepsilon_{\Delta f},$$

which will indicate that we do not improve our approach to the correct solution in the ordinate direction. When satisfied, we set

$$x^* \approx x_{i+1},$$

and terminate the iteration. Or we could require that

$$|x_{i+1} - x_i| \leq \varepsilon_{\Delta x},$$

which will indicate that we do not improve our approach to the correct solution in the abscissa direction. When satisfied, we set

$$x^* \approx x_{i+1},$$

and terminate the iteration.

In reality, we may stop the iteration when one of the test above are satisfied, or we may require that each of them are satisfied. In practice, it may also be necessary to introduce a maximal number of iterations  $i_{\max}$ , in case the Newton method doesn't work. Other refinements may also be necessary.

6. If the iteration is not terminated, we increase the iteration counter by 1:  $i = i + 1$ , and return to step 3.

Let us illustrate the method by plotting  $f(x) = \sin x - 0.5$  together with the linear approximation

$$f(x) \approx f(1) + \left. \frac{df(x)}{dx} \right|_{x=1} (x - 1) = \sin 1 - 0.5 + (\cos 1)(x - 1),$$

see fig. 15.3.

Here is the result of running the Matlab code above:

```
>> newton0
xi = 1
xi = 0.368
xi = 0.51831
xi = 0.52359
sin(xi) - 0.5 = -1.596e-011
```

We see how the iterates approach the correct solution  $x^* = \arcsin(0.5) = 0.523599$ .

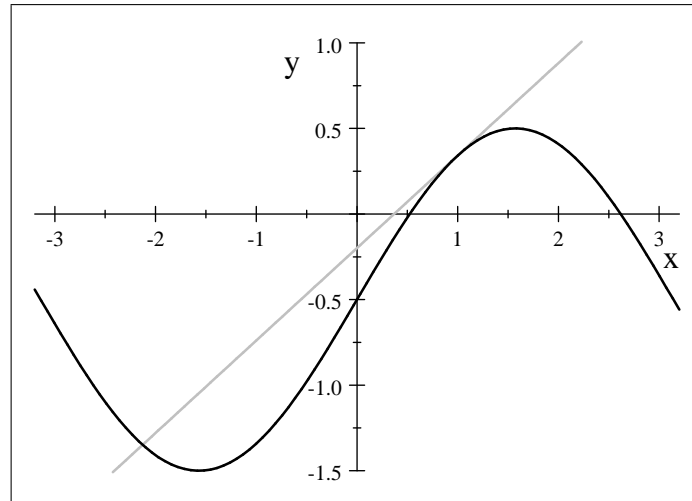


Figure 15.3: Plot of function  $f(x) = \sin x - 0.5$  (black), and linear approximation at  $x = 1$  :  $f(x) \approx \sin 1 - 0.5 + (x - 1) \cos 1$  (gray). We want to find  $x$  :  $f(x) = 0$ .

## 15.4.2 Generalizing the Newton method

Natural we observe that the Newton method is not restricted to the problem  $\sin x = 0.5$ . Furthermore, we observe that the Newton method is the same for any problem  $f(x) = 0$ , and that the only thing that is special is the form of the function  $f(x)$  and the form of the Jacobian  $J(x) = df(x)/dx$ . It is thus natural to consider separating the Newton algorithm itself from the form of the functions  $f(x)$  and  $J(x)$ .

We could consider the following strategy:

1. Define the function  $f(x)$  in a Matlab function `myfunction`, with input argument  $x$ , and output argument  $f(x)$ .
2. Define the Jacobian  $J(x) = df(x)/dx$  in Matlab function `myjacobian`, with input argument  $x$  and output argument  $J(x)$ .
3. Write a Newton solver function `mynewtonsolver`, with input argument  $x_0$ , and output argument  $x^*$  (or an approximation of  $x^*$ ).

Here is how our code works:

```
>> x0 = 1;
>> mynewtonsolver(x0)
ans =
    0.5236
```

In order to get this result, we have written the function `mynewtonsolver` for a *general* Newton solver:

```
function xsolution = mynewtonsolver(x0)
%
xi = x0;
fi = myfunction(xi);
Ji = myjacobian(xi);
while abs(fi) > 1e-10
    xip1 = xi-fi/Ji;
    xi = xip1;
    fi = myfunction(xi);
    Ji = myjacobian(xi);
```

```
end
xsolution = xi;
```

In addition, we have written the *special* function `myfunction` for defining  $f(x) = \sin x - 0.5$ :

```
function fx = myfunction(x)
%
fx = sin(x) - 0.5;
```

and the *special* function `myjacobian` for defining  $J(x) = df(x)/dx = \cos x$ .

```
function Jx = myjacobian(x)
%
Jx = cos(x);
```

### 15.4.3 Some final thoughts

We have now seen how we can separate a general algorithm (the Newton method) from a special case (finding the root of  $f(x) = \sin x - 0.5$ ) by writing a general function for the Newton method, and then implement the specific functions  $f(x)$  and  $J(x)$  in separate Matlab functions.

Still, our implementation is *not satisfactory*:

- With the current implementation, we can define one and only one problem  $f(x) = 0$  per directory on our computer! This is so since we are only allowed to have a single file with the name `myfunction`, and a single file named `myjacobian`.
- Clearly, this is too limiting. We need to be able to define many problems for each directory.
- The solution is to change the code of the Newton solver so that the user can define the file names where  $f(x)$  and  $J(x)$  are stored.

This modification of our Newton solver is the topic of the next section.

## 15.5 Function handles as function arguments

To make our Newton solver more general, i.e., such that the user can choose the name of the functions where  $f(x)$  and  $J(x) = df(x)/dx$ , we need to transfer information about the function names to the `mynewtonsolver`. The best way to transfer the names, is as input arguments to the Newton solver.

In Matlab, we do not transfer the names of the functions, but rather so-called *function handles*, which are pointers to the memory address of the functions. In order to specify that we mean the function *handles*, we precede the function names by the *function handle indicator* `@`. Here is a typical call structure to our new Newton solver, which we name `mynewtsolver`:

```
>> x0 = 1;
>> mynewtsolver(@mysinfunc, @mysinjacobian, x0)
```

Here, we notice:

1. The new function (which we have yet to code/write), is named `mynewtsolver`, and is quite similar to `mynewtonsolver`.
2. Compared to our previous solver `mynewtonsolver`, the new solver requires two additional input arguments: the function handles to the function where  $f(x)$  is defined (in this case function: `mysinfunc`) and where  $J(x) = df(x)/dx$  is defined (in this case function: `mysinjacobian`). We have to write both of these new functions.

3. In addition, we need to specify an initial guess  $x_0$ , i.e.  $x_0$ .
4. Notice that the order of the input arguments is arbitrary — we can choose the order as we want. However, when we have chosen the function order, we must stick to it!

Below is an example of how we can run the new code:

```
>> x0 = 1;
>> mynewtsolver(@mysinfunc, @mysinjacobian, x0)
ans =
    0.5236
```

Here is what the new function `mynewtsolver` looks like:

```
function xsolution = mynewtsolver(myfunc, myjac, x0)
%
% Prepared for Matlab 7
%
xi = x0;
fi = myfunc(xi);
Ji = myjac(xi);
while abs(fi) > 1e-10\qqquad \qqquad % Here, we could introduce more stop criteria
    xip1 = xi-fi/Ji;
    xi = xip1;
    fi = myfunc(xi);
    Ji = myjac(xi);
end
xsolution = xi;
```

The functions `mysinfunc` and `mysinjacobian` are defined just as before (but with other names):

```
function fx = mysinfunc(x)
%
fx = sin(x) - 0.5;
```

and:

```
function Jx = mysinjacobian(x)
%
Jx = cos(x);
```

*What have we gained* from this change? The important thing is that we can now use the Newton solver with new function names *without* changing the code within the Newton solver. This is very important: never write functions that the user is supposed to modify!

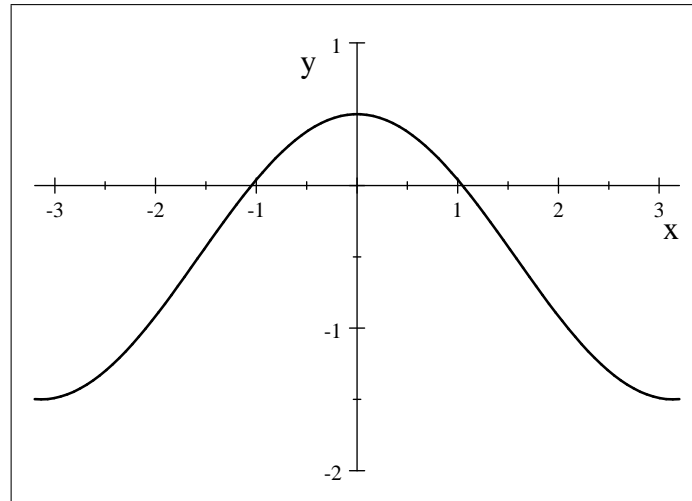
Suppose we want to use the code to solve another function, e.g.

$$f(x) = \cos x - 0.5.$$

This function is displayed in fig. 15.4. The Jacobian of this function is

$$J(x) = -\sin x,$$

and let us use  $x_0 = 0.5$  as an initial guess. We have to write two new functions, where we define  $f(x)$  and  $J(x)$ . Function `mycosfunc` is:

Figure 15.4: Function  $f(x) = \cos x - 0.5$ .

```
function fx = mycosfunc(x)
%
fx = cos(x) - 0.5;
```

and function mycosjacobian is:

```
function Jx = mycosjacobian(x)
%
Jx = -sin(x);
```

We solve the equation  $f(x) = \cos x - 0.5 = 0$  as follows:

```
>> x0 = 0.5;
>> mynewtsolver(@mycosfunc, @mycosjacobian, x0)
ans =
    1.0472
```

This idea of using function handles generalizes to any application that we may want to write. Here, it could be mentioned that there exist code that can automatically find the Jacobian of a vector function; such methods are denoted *automatic differentiation*.

## 15.6 Exercises

**Exercise 15.1** In the previous sections, we have solved  $\sin x = 0.5$  ( $f(x) = \sin x - 0.5 = 0$ ) with initial guess  $x_0 = 0.5$ .

- Use the functions that we have developed with initial values  $x_0 = 1.3$ ,  $x_0 = 1.4$ ,  $x_0 = 1.5$ . Comment on the results.
- Explain the result with the various initial guesses. ■

**Exercise 15.2** Redo the case  $f(x) = \sin x - 0.5 = 0$ , with  $x_0 = \pi/2$  and with  $x_0 = 0.999 \cdot \pi/2$ .

- Explain the results.

- Why may it be necessary to put restrictions on the maximal number of iterations,  $i_{\max}$ ?
- Modify the Newton solver `mynewtsolver` such that there is a limit to the number of iterations, and let your code write a message to the user if this maximum number is reached. ■

**Exercise 15.3** Use the developed code to find roots for the functions:

- $f(x) = \sqrt{x} - 1$ ,
- $f(x) = x^5 - 1.5$ ,
- $f(x) = \tan x - 0.5$ . ■

**Exercise 15.4** For some functions, it may be a hassle to find an analytic expression for the Jacobian  $J(x) = df(x)/dx$ . A possible alternative is to develop a function for computing the Jacobian numerically.

- Write a function for computing a numerical approximation for  $J(x)$  based on the central difference

$$J(x) \approx \frac{f(x + \Delta x) - f(x - \Delta x)}{\Delta x}$$

(or other approximations of the derivative). The code should automatically choose a relevant value for  $\Delta x$ , e.g. as  $\Delta x = x/100$ ,  $\Delta x = 10^{-5}$ , etc.

(**Hint:** Make sure that you avoid the possible problem that  $x$  may have the value  $x = 0$ , e.g. by setting  $\Delta x = \max(\frac{x}{100}, 10^{-5})$ .)

- Modify your Newton solver to automatically detect whether the user has specified a function for the Jacobian. If not, call the numerical approximation.

(**Hint:** Use the function `nargin` to find the number of arguments used by the user.) ■

**Exercise 15.5** Write a function `grad(f,x)` that finds the numerical gradient of a function of several variables. Assume that the function has form  $f(x)$  where  $x$  is a vector of several variables.

(**Hint:** To find the  $i$ -th component of the gradient, you must evaluate  $f$  at  $x + \Delta x$  etc. where element  $i$  of  $\Delta x$  differs from zero, while all other elements of  $\Delta x$  are zero. Test `grad` on the function  $\sin(x_1) + \cos(x_2)$ .) ■

**Exercise 15.6** Write a Newton solver that can solve the equation  $f(x) = 0$  for the general case, when both  $x$  and  $f(x)$  are vectors. It can be assumed that the number of unknowns  $x$  equals the number of equations  $f(x) = 0$ .

(**Hint:** You may assume that function `f` returns both the vector of function values, and the Jacobian, i.e. has the Matlab form `[f,J] = f(x)`. Remember that one Newton step in Matlab has the scalar case form `[f,J] = f(x); x = x - f/J;`. In the multivariable case, we need to change the scalar case line `x = x - f/J;` (`x := x - f/J`) to the multivariable version `x = x - J\f;` (`x := x - J^{-1}f`). Iterate until the norm of `x-J/f` is less than  $10^{-5}$ . Make sure that the iteration will stop eventually, even if the iteration does not converge. ■

**Exercise 15.7** Test your Newton solver in the previous exercise, on the function

$$\begin{aligned} f_1(x) &= x_1^2 + x_2^2 - 4 \\ f_2(x) &= \exp(x_1) + x_2 - 1. \end{aligned}$$

**Exercise 15.8** Write a function `jacobian(f,x)` that finds the numerical Jacobian matrix of a vector function  $f$  as a function of a vector  $x$  of variables. Thus:  $x \in \mathbb{C}^n$  and  $f(x) \in \mathbb{C}^m$ . ■

**Exercise 15.9** Modify the Newton solver so that if the user has not specified the Jacobian of the system, then the numerical Jacobian is computed using `Jacobian`. ■



**Exercise 15.10** Write a simple function `myint` for finding the integral  $I = \int_a^b f(x) dx$ , where the integral is approximated by

$$I \approx \sum_{i=0}^{N-1} f\left(a + i \cdot \frac{b-a}{N}\right) \cdot \Delta x.$$

Test function `myint` with functions  $f(x) = x^2$  and  $f(x) = \sin x$ , and compare the result with what you get using the built-in Matlab function `quad`. ■



# Chapter 16

## Simulation of dynamic systems

### 16.1 Solution of ODEs using Euler integration

We consider the damped mass-spring system in fig. 16.1. A model for the system in fig. 16.1 is:

$$\frac{dx}{dt} = v \quad (16.1)$$

$$\frac{dv}{dt} = -\frac{k}{m}(x - x_0) - \frac{\mu}{m}v - g + \frac{F}{m}. \quad (16.2)$$

The parameters, initial conditions, and input signals in Table 16.1 are to be used for this system.

Notice that we have not stated any reference position for  $x$ ! It is natural to assume that the unloaded spring hangs downwards. Since Table 16.1 specifies the unloaded spring position to be  $x_0 = 1$  m, this must mean that the spring hangs from a position  $> 1$  m (the  $x$ -axis points upwards!). Furthermore, since it is specified that  $x|_{t=0} = 1.5$  m, this means that the spring at  $t_0$  is compressed relative to its unloaded length.

In addition to the relationships above, we can express the kinetic energy  $K$ , the potential energy  $P$ , and the total energy  $E$  as follows:

$$K = \frac{1}{2}mv^2$$

$$P = mgx + \frac{k}{2}(x - x_0)^2$$

$$E = K + P.$$

$$\frac{dy}{dt} \approx \frac{y(t+h) - y(t)}{h}.$$

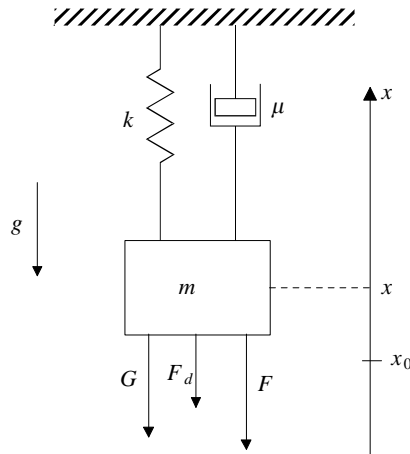


Figure 16.1: Sketch of damped mass-spring system.

Table 16.1: Numerical values for damped mass-spring system.

Numerical values		
$g = 9.81$	m/s <sup>2</sup>	gravitational acceleration
$k = 2$	J/m <sup>2</sup>	spring constant
$\mu = 0.5$	N/(m s)	damping constant
$x_0 = 1$	m	unloaded spring position
$m = 5$	kg	mass
$x _{t=0} = 1.5$	m	initial value for $x$
$v _{t=0} = 0$	m/s	initial value for $v$
$F = 0$	N	external force

We will now see how we can use the Explicit Euler Method to solve the ODE for  $x$  and  $v$ . The Explicit Euler Method works as follows:

**Algorithm 2** *In the Explicit Euler Method, we consider equations of form*

$$\frac{dy}{dt} = f(y, u, t).$$

1. Assume that  $y(t_0)$  is known, together with  $u(t_0), u(t_0 + h), u(t_0 + 2h), \dots$
2. Use the approximation

$$\frac{dy}{dt} \approx \frac{y(t_i + h) - y(t_i)}{h},$$

and rewrite the ODE to the following approximation:

$$y(t_i + h) = y(t_i) + h \cdot f(y(t_i), u(t_i), t_i).$$

3. With  $y(t_0)$  given, together with  $t_0$  and  $u(t_0)$ , compute  $y(t_0 + h)$ . Then iterate to compute  $y(t_i)$ .

The following script file `eulersim.m` will solve these equations, and compute  $x(t)$ ,  $v(t)$ , as well as  $K(t)$ ,  $P(t)$ , and  $E(t)$ .

1. Run the file by typing `>>eulersim`.
2. Plot the result using commands `>>plot(T,Y(:,1))`, `>>plot(T,Y(:,2))` and `>>plot(T,E)`.
3. Explain the figures.

Here is an example of what the file `eulersim.m` can look like:

```
% Script for simulating a damped mass-spring system
% using the Explicit Euler Method

% Defining system parameters
data;

% Specifying simulation time Tfin, step length h,
% and computing the necessary number of steps N
Tfin = 150;
h = 0.1;
N = ceil(Tfin/h); % Rounds the necessary steps up to the nearest integer

% Allocating space for the time vector T, as well as the state matrix Y
% and the energy function E
T = zeros(N,1);
Y = zeros(N,2);
E = zeros(N,2);
```

```

% Initial values for the states
Y(1,:) = [1.5, 0];

% Running the simulation
for I = 1:(N-1),
    x = Y(I,1);
    v = Y(I,2);
    fx = v;
    fv = - my*v/m - k*(x-x0)/m - g + F/m;
    Y(I+1,1) = x + h*fx;
    Y(I+1,2) = v + h*fv;
    T(I+1) = T(I) + h;
end

%Post processing to find the time response of energy E
K = 0.5*m*Y(:,2).*Y(:,2);
P = m*g*Y(:,1) + 0.5*k*(Y(:,1)-x0).^2;
E = K + P;

```

Note the command `data`. This command runs a script file (`data.m`) which holds the numerical values for parameters, etc.:

```

g = 9.81;          \quad \quad \quad % m/s^2, gravitational acceleration
k = 2;            % J/m^2, spring constant
my = 0.5;         \quad \quad \quad % N/m/s, damping constant
x0 = 1;          % m, unloaded spring position
m = 5;           % kg, masse hanging in the spring
F = 0;           % N, external force on mass

```

Also, notice the following post processing step at the end of script file `eulersim.m`:

```

Y(:,2).*Y(:,2)
(Y(:,1)-x0).^2

```

Here we have used array processing to compute the energy.

## 16.2 Generalizing our Euler solver

The script `eulersim.m` is unfortunate in the sense that we have mixed a general algorithm (the Euler method) with a specific problem (the damped spring-mass system). This unfortunate formulation is similar to a previous case, where we mixed a general method (the Newton method) with a specific problem (solving  $\sin x = 0.5$ ).

Just as in the case of developing a general Newton solver, we want to develop a general Euler solver where we separate the general method from the specific problem. Here is what we need to do:

- Separate the Euler solver from the specific model (the ODE), by specifying the ODE in separate functions.
- Allow for the user to specify the function name where the ODE is defined, with a free choice of the name of this function.

The following listing of file `odexeu.m` gives a relatively general ODE solver<sup>1</sup> based on the Explicit Euler Method:

<sup>1</sup>ODE = ordinary differential equation. In fact, the ODE solvers require that the system is in state space form, i.e. ODEs of form  $dy/dt = f(t, y)$  where  $y$  may be a vector,  $y \in \mathbb{R}^n$ .

```

function [T,Y] = odexeu(filehandle,Tspan,y0,h);
%
% Function call: [T,Y] = odexeu(filename,Tspan,y0,h)
%
% The function utilizes the Explicit Euler Method
% to solve the ordinary differential equation (ODE)  $dy/dt = f(t,y)$ 
% where y may be a vector, and f(.,.) denotes the
% vector field of the ODE
%
% Interpretation of variables:
% filehandle: the function handle of the file where the vector field is defined
% Tspan: a vector [t0,tf] specifying the initial and final time of the simulation
% y0: the vector of initial values, y(t0)
% h: the step length of the simulation
% T: the resulting vector of time instances where y is computed
% Y: a matrix of the time response of the states, one response fore each column of Y
%
% As an example, the time response of state 1 can be plotted using
% command: >>plot(T,Y(:,1))

% Prepared for Matlab 7

% Computing of the number of time steps in the Euler method:
t0 = Tspan(1);
tf = Tspan(2);
N = ceil(tf/h);\qqquad \qqquad % rounds the iteration number upwards to nearest integer
% Allocates space to store the results
n = length(y0);
T = zeros(N,1);
Y = zeros(N,n);
% Inserts initial values in T and Y
T(1) = t0;
Y(1,:) = y0';
% Computes y(t) in a for loop:
for I=1:(N-1);
    y = Y(I,:);
    t = T(I);
    fy = filehandle(t,y);
    Y(I+1,:) = (y + h*fy)';
    T(I+1) = T(I) + h;
end

```

The following script file (e.g. named `dampms.m` — damped mass spring) simulates the damped mass-spring system.

```

% Script for simulating damped mass-spring system

% Necessary data
Tfin = 150;
h = 0.1;
y0 = [1.5;0];
% Simulating system
[T,Y] = odexeu(@dampedspring,[0,Tfin],y0,h);
% Computes the energy
data;
K = 0.5*m*Y(:,2).*Y(:,2);

```

```
P = m*g*Y(:,1) + 0.5*k*(Y(:,1)-x0).^2;
E = K + P;
```

The data for the system is given in the file `data.m`, see p. 143. Also, we define the “vector field” for the differential equation in the function `dampedspring.m`:

```
function fy = dampedspring(t,y)
%
% Defining data
data;
% Defining vector field
x = y(1);
v = y(2);
fx = v;
fv = - my*v/m - k*(x-x0)/m - g + F/m;
% Setting up vector field
fy = [fx;fv];
```

In order to run the program, issue the Matlab command `>>dampms`. The result is plotted using the command `>>plot(T,E)`, etc.

**Exercise 16.1** With  $h = 0.1$  s, simulate the system in the interval  $t \in [0, 150]$  s (as indicated in the script file `dampms.m`). Plot the time response of  $x$  ( $Y(:,1)$ ) and  $v$  ( $Y(:,2)$ ) as functions of time ( $T$ ). ■

**Exercise 16.2** Plot the time response of the total energy  $E$  ( $E$ ) as a function of the time. From the relationship  $dE/dt = -\mu v^2 < 0$  (assuming  $F = 0$ ), it follows that  $E$  must decrease monotonously with the time  $t$ . Why is this theoretical fact not supported by the plot of  $E(t)$ ? ■

**Exercise 16.3** Experiment with various values on the step length  $h$ , and see if this changes the time response of  $E$ . What is the largest step length you can use, and still get a relatively good result from the simulations?

(**Hint:** Even if  $E(t)$  oscillates, we will get a relatively good picture of  $E$  by averaging the energy over a few oscillation periods. For what value of  $h$  will the energy  $E(t)$  “take off”?) ■

**Exercise 16.4** Experiment with other values on the parameters ( $g, k, \mu, x_0, m$ ) and the initial values ( $x|_{t=0} = 1.5, v|_{t=0} = 0$ ). Will the changes in the parameter values or the initial values affect the requirement on step length  $h$  such that the simulation gives a good approximation of  $E(t)$ ? ■

**Exercise 16.5** Study the undamped case  $\mu = 0$  when  $F = 0$ . In this case, the energy  $E$  should be constant according to theory,  $dE/dt = 0$ . Simulate the system with  $\mu = 0$  and step length  $h = 0.1$  (it suffices to simulate the system for 50 s, i.e. set `Tfin=50`). What happens with  $E$ ? Is the theoretically correct result that  $dE/dt = 0$ , satisfied? ■

**Exercise 16.6** In the previous exercise, we simulated the undamped case ( $\mu = 0$ ), and in this case,  $E$  should theoretically be constant. With  $h = 0.1$ , it turns out that the the energy increases, which implies that the approximation we introduced is imperfect. *One* possible way to improve the simulation is to use a better approximation for  $dy/dt$ . *Alternatively*, we can attempt to rectify the error in the approximation by introducing an artificial damping, i.e. increasing  $\mu$  until the energy on the average is constant.

What value  $\mu$  must be used such that the energy  $E$  averaged over some oscillation periods becomes constant? ■

Table 16.2: Table Caption

ODE solver	Comment
<code>ode45</code>	Fourth order (explicit) Runge-Kutta method.
<code>ode23</code>	Second order (explicit) Runge-Kutta method.
<code>ode113</code>	Variable order Adams-Bashforth-Moulton method.
<code>ode15s</code>	Variable-order multistep solver (1st - 5th order).
<code>ode23s</code>	Second order modified Rosenbrock single step method.
<code>ode23t</code>	Second order trapezoidal method.
<code>ode23tb</code>	Second order implicit Runge-Kutta method.

### 16.3 ODE solvers in Matlab

Matlab comes with a number of ODE solvers as standard. All of them are named `odeXXX`, where the addition `XXX` specifies what type of method is used in the particular solver. This should be compared to our solver `odexeu` from the previous section, where `XXX` is equal to `xeu` denoting eXplicit EUler method. A list of available solvers is shown in Table 16.2.

As a general comment, the methods are more accurate the higher their order. However, higher order methods require more computing time to find the solution. If a solution reaches a constraint (e.g. if  $y \leq y_{\max}$ , such as when a vessel becomes full), then the accuracy is lost when  $y = y_{\max}$ .

Most methods are single step methods, meaning that for an ODE of type

$$\frac{dy}{dt} = f(t, y),$$

$y_i$  is computed from the knowledge of  $y_{i-1}$ . Two methods in Table 16.2 are multistep solvers (`ode113` and `ode15s`), meaning that  $y_i$  is computed from the knowledge of  $y_{i-1}$ ,  $y_{i-2}$ , etc. Multistep solvers are usually more efficient than single-step solvers, but this is not true if the solver needs to be reset often — as an example, at start-up, only  $y(t_0)$  is known, and multistep solvers need to use a single-step method in the beginning.

Finally, the first three solvers in Table 16.2 are non-stiff solvers, while the four following methods are stiff solvers. As a general rule, stiff problems have modes with very different time constants. For such problems, non-stiff solvers are very slow at finding the solution, while stiff solvers may find the solution rapidly. As an example, for some stiff problems, solver `ode15s` may be several hundred times faster than `ode45`.

We have seen that we call our `odexeu` solver as follows:

```
[T,Y] = odexeu(@dampedspring,[0,Tfin],y0,h);
```

Here, the final input argument is the time increment `h`. In fact, it is very important to find a good choice of the time increment `h`. With too large values of `h`, the approximation may become very poor, even unstable. For too small values of `h`, the solution becomes inefficient: it takes too much time to find the solution.

All of the Matlab solvers in Table 16.2 have algorithms for automatically computing the time increment `h`. As an example: for `ode45`, it is postulated that the error in the simulation is equal to the difference between the fourth order solution and the fifth order solution, and then the time increment `h` is adjusted to achieve sufficient accuracy in the solution.

Consequently, it is not necessary to inform Matlab about the time increment — in fact, Matlab varies `h` from iteration number to iteration number. *Apart* from this modification, the input arguments to the ODE solvers in Table 16.2 are *identical* to the arguments for `odexeu`. If we want to use solver `ode45` to solve the damped mass-spring system, we need to modify the script file `dampms` into the following:

```
% Script for simulating damped mass-spring system

% Necessary data
Tfin = 150;
h = 0.1;
```



Table 16.3: Parameters and nominal operating conditions for CSTR.

Parameter/variable	Symbol	Nominal value
Volume of reactor	$V$	100 l
Volumetric feed flow	$q$	$100 \text{ l min}^{-1}$
Feed concentration of $A$	$c_{Af}$	$1 \text{ mol l}^{-1}$
Feed temperature	$T_f$	350 K
Volumetric coolant flow	$q_c$	$100 \text{ l min}^{-1}$
Coolant inlet temperature	$T_{cf}$	350 K
Densities	$\rho, \rho_c$	$1000 \text{ g l}^{-1}$
Specific heat capacities	$c_v, c_{vc}$	$1 \text{ cal g}^{-1} \text{ K}^{-1}$
Preexponential factor	$k_0$	$7.2 \times 10^{10} \text{ min}^{-1}$
Exponential factor	$E/R$	$9.98 \times 10^3 \text{ K}$
Reaction order	$m$	1
Reaction enthalpy	$-\Delta\tilde{H}_r$	$2.0 \times 10^5 \text{ cal mol}^{-1}$
Overall heat transfer factor	$hA$	$7 \times 10^5 \text{ cal min}^{-1} \text{ K}^{-1}$
Discretization time	$\Delta t$	0.01 min
Initial value, $c_A$	$c_A(t=0)$	$8.235 \times 10^{-2} \text{ mol l}^{-1}$
Initial value, $T$	$T(t=0)$	441.81 K
Nominal value, $q_c$	$q_c(t)$	$100 \text{ l min}^{-1}$

```

y0 = [1.5;0];
% Simulating system
tic;
%[T,Y] = odexeu(@dampedspring,[0,Tfin],y0,h);
[T,Y] = ode45(@dampedspring,[0,Tfin],y0);
%[T,Y] = ode15s(@dampedspring,[0,Tfin],y0);
toc
% Computes the energy
data;
K = 0.5*m*Y(:,2).*Y(:,2);
P = m*g*Y(:,1) + 0.5*k*(Y(:,1)-x0).^2;
E = K + P;

```

Here, we have also inserted the commands `tic/toc` to measure the time it takes to solve the ODE using the various methods. For this simple case, we will find that the Euler method by far is the most efficient method. However, the other methods are more accurate, and with the built-in Matlab solvers, we do not have to find/choose the step-length  $h$ .

Finally, we should mention that we can set some parameters for the ODE solvers via the function `odeset`.

## 16.4 Exercises

**Exercise 16.7** Consider the following reactor model:

$$\frac{dc_A}{dt} = \frac{q}{V} (c_{Af} - c_A) - k_0 \exp\left(\frac{-E}{RT}\right) c_A^m$$

$$\frac{dT}{dt} = \frac{q}{V} (T_f - T) + \frac{(-\Delta\tilde{H}_r)}{\rho\hat{c}_{v,S}} \frac{k_0}{m} \exp\left(\frac{-E}{RT}\right) c_A^m + \frac{Q}{\rho V \hat{c}_{v,S}},$$

where supplied heat  $Q$  is expressed as:

$$Q = \rho_c q_c \hat{c}_{vc} \left[ 1 - \exp\left(-\frac{hA}{\rho_c q_c \hat{c}_{vc}}\right) \right] (T_{cf} - T).$$

Numerical values for parameters and nominal operating conditions are given in Table 16.3.

- Implement the reactor model and use the explicit Euler solver `odexeu` developed in this chapter.
- How much time does it take to solve the reactor model when you simulate 8 min of operation.
- Use the built-in Matlab solvers `ode45` and `ode15s`, and compare the execution time for simulating 8 min of operation with each other, and with `odexeu`.
- Change the simulation such that after 1 min of simulated time, the coolant flow:
  - Increases by 10%.
  - Decreases by 10%.

What do you observe? ■

**Part IV**  
**Closing**



# Chapter 17

## Conclusions

These lectures notes give a relatively simple introduction to Matlab, and how Matlab can be used to do engineering computations. The notes have been updated to cover Matlab 7.0. Matlab is constantly being improved: the user interface changes, new data structures are introduced, and so on. This means that it is difficult to keep lecture notes for Matlab up to date. Fortunately, there is also a large number of books on Matlab: both introductory books, specialized books (e.g. how to use Matlab to develop numerical algorithms, how to design graphical user interfaces (GUI) for your Matlab code), etc. As with computer books at large, some of the Matlab books are good, and others are less useful.

The first part of the lecture notes covers Arrays: what they are, how to name them, and how to create and manipulate them. Next, basic plotting is treated. A chapter on simple data analysis is provided, and finally the automation of tasks is introduced, with basic string handling, the basic repetition loop (the `for` loop), as well as the concept of Matlab scripts and the Matlab editor.

In the second part, more techniques for program flow control is introduced, and then the important concept of functions is introduced. Followed by this is a discussion of more advanced use of the Matlab editor such as the debugger. Finally some more advanced data structures are introduced (structures, cell arrays), and some basic information about objects are given (handle graphics).

In the third part, functions and function handles are introduced, their usefulness are illustrated through examples (a Newton solver), and finally, there is an overview of how ODE solvers work, as an introduction to the use of built-in Matlab ODE solvers.

In appendices, more details are given. In the first appendix, there is a discussion of vectors and matrices; this should be contrasted with the concept of arrays. In the second appendix, there is a technical discussion of the intricacies of indexing arrays. The third appendix gives an example of how hexadecimal numbers relate to decimal numbers; hexadecimal numbers are the native numbers of computers. Appendix four discusses three dimensional plotting, as well as the housekeeping of plots. In appendix five, a more advanced discussion of string manipulation is given. In appendix six, some numerical methods for vectors and matrices are discussed; Matlab is particular useful for handling linear algebra problems. Appendix seven discusses how anonymous functions can be used to handle extra parameters in function calls. Finally, appendix eight discusses how differential algebraic equations (DAEs) can be solved in Matlab.

To really learn to use Matlab, it is necessary to use it, explore the help browser, but also to study Matlab in more detail. Books such as Hanselman & Littlefield (2005) and Higham & Higham (2000) will provide a good start for becoming a Matlab master.



**Part V**  
**Appendices**





# Appendix A

## Arrays vs. vectors and matrices\*

### A.1 Vectors

**Definition A.1** A vector is an element of a vector space.

**Definition A.2** A vector space is a set  $\mathcal{V}$  together with an addition on  $\mathcal{V}$ :  $u + v$ ,  $u, v \in \mathcal{V}$ , and a scalar multiplication on  $\mathcal{V}$ :  $av$ ,  $a \in \mathbb{F}$  and  $v \in \mathcal{V}$ , such that the following properties hold:

1. *Commutativity*:  $u + v = v + u$  for all  $u, v \in \mathcal{V}$ .
2. *Associativity*:  $(u + v) + w = u + (v + w)$  and  $(ab)v = a(bv)$  for all  $u, v, w \in \mathcal{V}$  and all  $a, b \in \mathbb{F}$ .
3. *Additive identity*: there exists an element  $0 \in \mathcal{V}$  such that  $v + 0 = v$  for all  $v \in \mathcal{V}$ .
4. *Additive inverse*: for every  $v \in \mathcal{V}$ , there exists  $w \in \mathcal{V}$  such that  $v + w = 0$ . Normally,  $w$  is denoted  $-v$ .
5. *Multiplicative identity*: there exists an element  $1 \in \mathbb{F}$  such that  $1v = v$  for all  $v \in \mathcal{V}$ .
6. *Distributive properties*:  $a(u + v) = au + av$  and  $(a + b)u = au + bu$  for all  $a, b \in \mathbb{F}$  and all  $u, v \in \mathcal{V}$ .

What does this mean? The vector space is a set, and the operation of addition between vectors, and multiplication of vectors with a scalar must be defined. The operation of addition and multiplication must satisfy the requirements above.

**Example A.1** Let us consider the set of  $n$ -tuples of real numbers,  $\mathcal{V} = \mathbb{R}^n$ , i.e. the ordered list  $(v_1, \dots, v_n)$  where each  $v_i \in \mathbb{R}$ . In addition, let us define addition of the elements of the set,  $v + u$ , as

$$v + u = (v_1 + u_1, \dots, v_n + u_n).$$

Thus, the operation of addition is well defined. Furthermore, let us define  $\mathbb{F} = \mathbb{I} = \{1, 2, 3, \dots\}$ , i.e. the positive integers, and let us define scalar multiplication as

$$av = (a \cdot v_1, \dots, a \cdot v_n).$$

Thus, the operation of scalar multiplication is well defined. It is easy to see that Properties 1, 2, 6 are fulfilled. Furthermore, let us define  $0 \triangleq (0, \dots, 0)$ . Here it is vital to realize that the vector 0 (which is a list of  $n$  scalar zeros) is different from the scalar 0. Obviously,  $0 \in \mathbb{R}^n$ , so Property 3 is fulfilled. Furthermore, obviously Property 4 is fulfilled. Finally, scalar  $1 \in \mathbb{I} = \mathbb{F}$ , and thus Property 5 is fulfilled. ■

**Example A.2** Consider the set of all  $n$ -th order polynomials  $p^n = p_0 + p_1x + \dots + p_nx^n$ , where normally  $p_i \in \mathbb{R}$  or  $p_i \in \mathbb{C}$ . We define the addition of such polynomials as  $p^n + q^n = (p_0 + q_0) + (p_1 + q_1)x + \dots + (p_n + q_n)x^n$ , thus addition is well defined. Furthermore, we define a set  $\mathbb{F}$ , e.g.  $\mathbb{F} = \mathbb{I}$ , and scalar multiplication as  $ap^n = (ap_0) + (ap_1)x + \dots + (ap_n)x^n$ , hence scalar multiplication is well defined. It is straightforward to show that Properties 1–6 are fulfilled, hence the polynomials constitute a vector space, and e.g.  $p^5 = 1 - x^3 + 2x^5$  is a vector. ■

The most commonly used vector space, is the space where  $\mathcal{V} = \mathbb{R}^n$  (or  $\mathcal{V} = \mathbb{C}^n$ ), and where  $\mathbb{F} = \mathbb{R}$  (or  $\mathbb{F} = \mathbb{C}$ ) and where addition and scalar multiplication is defined as in Example A.1. In fact, Matlab is tailor-made for the vector space  $\mathcal{V} = \mathbb{C}^n$ ,  $\mathbb{F} = \mathbb{C}$ . Addition of vectors is performed with the operator  $+$ , and multiplication by scalar by the operator  $*$ , and vectors are represented by either row arrays or column arrays:

```
>> vec1 = [1,2,3]

vec1 =

     1     2     3

>> vec2 = [4,5,6]

vec2 =

     4     5     6

>> scalar = 3

scalar =

     3

>> vec1 + vec2

ans =

     5     7     9

>> scalar*vec1

ans =

     3     6     9
```

It should be noted that Matlab is strict in the sense that vector addition only works between two row arrays, or between two column arrays.

Note also that the operator  $+$  plays several rôles in Matlab: the plus operator can be used to add scalars. In addition, the plus operator can be used to add a vector and a scalar: in that case, the scalar is interpreted as follows: `vector + scalar = vector + scalar*ones(size(scalar))`, i.e., the scalar is first expanded into a vector, and then the vector addition is used.

In conclusion: vectors are more than simply arrays. But the most common vectors can be represented as row or column arrays with the addition of well defined vector addition and scalar multiplication. In fact, it can be shown that every vector space of dimension  $n$  in some sense is identical to the vector space given by  $\mathcal{V} = \mathbb{C}^n$  and  $\mathbb{F} = \mathbb{C}$  with addition and multiplication defined as above.

This means that Matlab is well equipped to compute with vectors. The relevant functions for performing such operations, belong to a course in linear algebra.

## A.2 Matrices

Often, the word *matrix* is used as a synonym to *array*. However, since there has been developed an algebra of matrices, and furthermore, since matrices can be considered collections of vectors or as operators on vector spaces, we will use matrix to denote something more than arrays — in these notes, and array is simply and organization/storage of data as detailed in Section 2.2.

As an example, we often define matrix addition and the multiplication of matrices with scalar — much in the same way as we do with vector spaces. Also, several definitions of matrix multiplication exist:

**Cayley product:** Named after Arthur Cayley, who often is considered the founder of matrix algebra, the Cayley (or matrix) product of matrices  $A$  and  $B$ ,  $C = A \cdot B$ , is defined as:

$$C_{i,j} = \sum_k A_{i,k} \cdot B_{k,j}.$$

It follows that the number of columns of  $A$  must equal the number of rows of  $B$ .

**Schur product:** The Schur- (or Direct- or Array-) product of matrices  $A$  and  $B$ ,  $C = A \times B$ , is defined as

$$C_{i,j} = A_{i,j} \cdot B_{i,j}.$$

It follows that  $A$  and  $B$  must have the same number of rows and columns.

**Kronecker product:** We have already met the Kronecker product of matrices  $A$  and  $B$  (see p. 162),  $C = A \otimes B$ , which is defined as

$$C = \begin{pmatrix} A_{1,1}B & A_{1,2}B & \cdots & A_{1,n}B \\ A_{2,1}B & A_{2,2}B & & A_{2,n}B \\ \vdots & & \ddots & \vdots \\ A_{m,1}B & A_{m,2}B & \cdots & A_{m,n}B \end{pmatrix}.$$

It follows that there are no restrictions on the sizes of  $A$  and  $B$ .

Another use/interpretation of matrices is that each column may represent a column vector, or that each row may represent a row vector. With these interpretations, matrices are related to vector spaces.

A third use of matrices is as a mapping/an operator on vectors, typically written as  $Av$  where  $A$  is the matrix and  $v$  the vector. Usually, the standard (Cayley) matrix product is assumed in the notation  $Av$ , and in that case, it is necessary that  $v$  is considered a column vector/a column matrix.

In Matlab, the operator  $+$  can be used to add matrices in a natural way, and furthermore, the operator  $*$  works on arrays according to the Cayley matrix product:

```
>> A = rand(2,3);
>> B = rand(3,2);
>> C = ones(2,3);
>> A+C
```

```
ans =
```

```
1.9501    1.6068    1.8913
1.2311    1.4860    1.7621
```

```
>> A*B
```

```
ans =
```

```
1.1771    1.5018
0.7405    1.0054
```

Matlab also has a large number of built-in functions for treating matrices as collections of vector spaces, e.g. `rank`, `null`, `orth`, etc., or as operators, e.g. `det`, `eig`, etc. In addition, there are a large number of built-in functions for dealing with linear equations of form

$$Ax = b.$$

This means that Matlab is well equipped to compute with matrices. The relevant functions for performing such operations, belong to a course in linear algebra.



## Appendix B

# Manipulation of Arrays and Array Functions\*

### B.1 Indexing arrays

Suppose array  $A$  is given. We often wish to pick out element  $(i, j)$  from array  $A$ . Let us denote element  $(i, j)$  of array  $A$  by  $A_{i,j}$ , which in Matlab is given as  $A(i, j)$ .

For array  $A$ , it can also be useful to be able to pick a subarray which has indices over the rectangle given by row  $i_1$  to row  $i_2$ , and column  $j_1$  to  $j_2$ . This submatrix can be denoted as  $A_{i_1:i_2, j_1:j_2}$ . In Matlab, this subarray is specified as  $A(i_1:i_2, j_1:j_2)$ :

```
>> rand('state',0)
>> A = rand(3,4)

A =

    0.95013    0.48598    0.45647    0.4447
    0.23114    0.8913    0.018504    0.61543
    0.60684    0.7621    0.82141    0.79194

>> A(1,3)

ans =

    0.45647

>> A(2:3,2:3)

ans =

    0.8913    0.018504
    0.7621    0.82141
```

Let us do some careful thinking here! We remember that the notation  $i_1 : i_2$  in fact constructs a row array consisting of the following elements:  $(i_1, i_1 + 1, \dots, i_2)$ , and similarly for  $j_1 : j_2$ . Let us therefore try to define the row arrays  $v_1$  and  $v_2$  of integers, and consider the notation  $A_{v_1, v_2}$ :

```
>> v1 = [1,3]; v2 = [4,2,2];
>> A(v1,v2)

ans =
```

```
0.4447    0.4860    0.4860
0.7919    0.7621    0.7621
```

With  $v_1$  being a row array of length  $m$ , and  $v_2$  being a row array of length  $n$ , we see that Matlab in fact works as follows:

$$A_{v_1, v_2} = \begin{pmatrix} A_{v_1(1), v_2(1)} & A_{v_1(1), v_2(2)} & \cdots & A_{v_1(1), v_2(n)} \\ A_{v_1(2), v_2(1)} & A_{v_1(2), v_2(2)} & & A_{v_1(2), v_2(n)} \\ \vdots & & \ddots & \\ A_{v_1(m), v_2(1)} & \cdots & & A_{v_1(m), v_2(n)} \end{pmatrix}.$$

What then if we only use one argument, e.g. a row array?

```
>> A(v1)

ans =

    0.95013    0.60684
```

Now it is slightly more difficult to figure out what is going on. But we see that  $A$  assumes the shape of  $v_1$ , and that

$$A(v_1) = ( A(v_1(1)) \quad A(v_1(2)) \quad \cdots \quad A(v_1(m)) ),$$

where one-dimensional addressing has been used. Similarly, if we use one argument where the argument is a two-dimensional array:

```
>> B = [1,3,2; 4,2, 2]

B =

     1     3     2
     4     2     2

>> A(B)

ans =

    0.95013    0.60684    0.23114
    0.48598    0.23114    0.23114
```

Quite often, we are interested in picking out all rows in a column, or every column in a row. We can specify every row with the symbol  $:$ , e.g.  $A_{:,v_2}$  where row array  $v_2$  suggests which columns should be included, and every column as  $A_{v_1,:}$  where row array  $v_1$  indicates which rows of  $A$  should be included.

The smallest index in Matlab is always 1. The largest index in rows and columns can be specified with the symbol **end** in Matlab, e.g.:  $A_{1:\text{end},v_2} \triangleq A_{:,v_2}$ . This notation is particularly useful if we want to partition an array into two (or more) parts, e.g. with arrays  $A_{1:i,:}$  and  $A_{i+1:\text{end},:}$ :

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

ans =

    0.9501
    0.2311
    0.6068

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

```
ans =
    0.4860    0.4565    0.4447
    0.8913    0.0185    0.6154
    0.7621    0.8214    0.7919
```

## B.2 Building superarrays

### B.2.1 Toeplitz and Hankel arrays

The structure of Toeplitz and Hankel arrays are best illustrated by examples. Let's first take a look at Toeplitz arrays:

```
>> c = [1,2,3]; r = [4,5,6,7];
>> toeplitz(c,r)
Warning: First element of input column does not match first element of input row.
        Column wins diagonal conflict.
(Type "warning off MATLAB:toeplitz:DiagonalConflict" to suppress this warning.)
> In C:\MATLAB6p5\toolbox\matlab\elmat\toeplitz.m at line 18
```

```
ans =
     1     5     6     7
     2     1     5     6
     3     2     1     5
```

```
>> toeplitz(c)
```

```
ans =
     1     2     3
     2     1     2
     3     2     1
```

```
>> toeplitz(r)
```

```
ans =
     4     5     6     7
     5     4     5     6
     6     5     4     5
     7     6     5     4
```

We see that Toeplitz arrays have constant elements along the *diagonals*. Notice also that if there is a conflict between the first element of  $c$  and the first element of  $r$  (these have to be equal!), the first element of  $c$  “wins”.

Next, we look at Hankel arrays:

```
>> hankel(c,r)
Warning: Last element of input column does not match first element of input row.
        Column wins anti-diagonal conflict.
(Type "warning off MATLAB:hankel:AntiDiagonalConflict" to suppress this warning.)
> In C:\MATLAB6p5\toolbox\matlab\elmat\hankel.m at line 27
```

```

ans =

     1     2     3     5
     2     3     5     6
     3     5     6     7

>> hankel(c)

ans =

     1     2     3
     2     3     0
     3     0     0

>> hankel(r)

ans =

     4     5     6     7
     5     6     7     0
     6     7     0     0
     7     0     0     0

```

We see that Hankel arrays have constant elements along the *anti-diagonals*. Notice that if there is a conflict between the last element of  $c$  and the first element of  $r$  (these have to be equal!), then the last element of  $c$  “wins”.

## B.2.2 Kronecker product

The Kronecker product of two arrays (or matrices)  $A$  and  $B$  is denoted  $A \otimes B$ . If  $A$  is an array of size  $m \times n$  and  $B$  is an array of size  $k \times \ell$ , then the product is of size  $m \cdot k \times n \cdot \ell$ , and the result generated as follows:

$$C = A \otimes B = \begin{pmatrix} A_{1,1}B & A_{1,2}B & \cdots & A_{1,n}B \\ A_{2,1}B & \ddots & & \vdots \\ \vdots & & \ddots & \vdots \\ A_{m,1}B & \cdots & \cdots & A_{m,n}B \end{pmatrix}.$$

Here,  $A_{i,j}B$  means the value of element  $A_{i,j}$  multiplied into every element of array  $B$ .

A common problem is to replicate array  $B$  in a certain pattern, e.g. we want to insert array  $B$  instead of a unit scalar in array  $A$ :

$$A = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}.$$

By using the Kronecker product which is achieved by `kron(A,B)`, we can produce the concatenated superarray

$$C = A \otimes B = \begin{pmatrix} B & 0 & B \\ B & B & 0 \end{pmatrix}.$$

The Matlab syntax for this operation is `kron(A,B)`:

```

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

A =

     1     0     1
     1     1     0

>> B = rand(3,2)

```



```

B =

    0.4565    0.4447
    0.0185    0.6154
    0.8214    0.7919

>> kron(A,B)

ans =

    0.4565    0.4447         0         0    0.4565    0.4447
    0.0185    0.6154         0         0    0.0185    0.6154
    0.8214    0.7919         0         0    0.8214    0.7919
    0.4565    0.4447    0.4565    0.4447         0         0
    0.0185    0.6154    0.0185    0.6154         0         0
    0.8214    0.7919    0.8214    0.7919         0         0

```

Obviously, we can use the Kronecker product to achieve the same effect as the function `repmat(A,m,n)` as follows: `kron(ones(m,n), A)`. However, the `repmat` function is more efficient than using the `kron` function.

### B.2.3 Block diagonal arrays

Function `blkdiag` is used for block diagonal concatenation of arrays:

```

>> A = rand(2,2);
>> B = randn(2,3);
>> C = blkdiag(A,B)

C =

    0.9218    0.1763         0         0         0
    0.7382    0.4057         0         0         0
         0         0   -0.4326    0.1253   -1.1465
         0         0   -1.6656    0.2877    1.1909

>> D = blkdiag(A,B,-1)

D =

    0.9218    0.1763         0         0         0         0
    0.7382    0.4057         0         0         0         0
         0         0   -0.4326    0.1253   -1.1465         0
         0         0   -1.6656    0.2877    1.1909         0
         0         0         0         0         0   -1.0000

```

Note that it is not possible to specify a diagonal off the main diagonal in the case of block diagonal arrays.



## Appendix C

# Hexadecimal numbers\*

Each byte (1 byte = 8 bits) is composed of 2 hexadecimal numbers, where the first hexadecimal number represents the first four bits, and the second hexadecimal number represents the four last bits of the byte. The 16 hexadecimal digits are  $\{0, 1, 2, \dots, 9, a, b, c, d, e, f\}$ . The first half of the first byte of  $1/3$  is given by hexadecimal number  $3_{16}$ , which in the decimal number system is  $3_{10}$  — the subscript indicates the base of the number, i.e. 16 (hexadecimal) or 10 (decimal). The binary equivalent of this number is  $0011_2 = (0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)_{10} = 3_{10}$ . The next half of the first byte is  $f_{16} = 15_{10} = 1111_2$ . In total, the first byte of the representation of  $1/3$  is thus  $00111111$ . Similarly, the first half of the second byte is given by  $d_{16} = 13_{10} = 1101_2$ . Likewise,  $5_{16} = 5_{10} = 0101_2$ . Thus, the second byte is  $11010101$ . The remaining 6 bytes are simply  $01010101$ . We can thus write the binary representation of  $1/3$  as  $00111111_1 11010101_2 01010101_3 \dots 01010101_7 01010101_8$  — this time, the subscript indicates the byte number. Let us number the bits from 1 to 64 as  $b_1 b_2 \dots b_{64}$ , such that in this number,  $b_1 = 0$ ,  $b_2 = 0$ ,  $b_3 = 1$ ,  $b_4 = 1$ , etc.

According to the IEEE standard, the representation of floating point numbers is composed of a mantissa part  $m$ , and an exponent part  $e$ , and the number is given as  $m \times 2^e$ , see (Gentle 1998). Typically,  $b_1$  signifies the sign of the mantissa (0 means positive number, 1 means negative number),  $b_2$ – $b_{12}$  (11 bits) gives the exponent ( $\tilde{e}$ ), while  $b_{13}$ – $b_{64}$  gives the mantissa (52 bits). The mantissa is scaled such that the first bit in the mantissa is always 1. Since it is always 1, it is not included, and we have to add a bit with value 1 at the beginning of the mantissa. Thus,  $b_{13}$  is really the second bit in the mantissa. Finally, there is no sign bit in the exponent  $\tilde{e}$ . Instead, a fixed number  $\hat{e}$  is subtracted from the exponent to get  $e = \tilde{e} - \hat{e}$  before we evaluate  $2^e$ .

Let us consider the representation of  $1/3$ .  $b_1 = 0$ , which means a positive sign. Next, the mantissa is given as

$$\begin{aligned}
 m &= 1b_{13}b_{14} \dots b_{64} = 10101_2 01010101_3 \dots 01010101_7 01010101_8 \\
 &= 1 \cdot 2^{-1} + b_{13} \cdot 2^{-2} + b_{14} \cdot 2^{-3} + \dots + b_{63} \cdot 2^{-52} + b_{64} \cdot 2^{-53} \\
 &= 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} \\
 &\quad + \sum_{i=3}^8 \left( 0 \cdot 2^{-6-(i-3) \cdot 8} + 1 \cdot 2^{-7-(i-3) \cdot 8} + 0 \cdot 2^{-8-(i-3) \cdot 8} + 1 \cdot 2^{-9-(i-3) \cdot 8} + 0 \cdot 2^{-10-(i-3) \cdot 8} \right. \\
 &\quad \left. + 1 \cdot 2^{-11-(i-3) \cdot 8} + 0 \cdot 2^{-12-(i-3) \cdot 8} + 1 \cdot 2^{-13-(i-3) \cdot 8} \right) \\
 &= 0.666\ 666\ 666\ 666\ 666\ 629\ 66.
 \end{aligned}$$

The exponent is given as

$$\begin{aligned}
 \tilde{e} &= b_2 b_3 \dots b_{12} = 0111111_1 1101 \\
 &= b_{12} \cdot 2^0 + b_{11} \cdot 2^1 + b_{10} \cdot 2^2 + \dots + b_3 \cdot 2^9 + b_2 \cdot 2^{10} \\
 &= 1 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + \dots + 1 \cdot 2^9 + 0 \cdot 2^{10} \\
 &= 1 \cdot 2^0 + 0 \cdot 2^1 + \sum_{i=2}^9 1 \cdot 2^i + 0 \cdot 2^{10} \\
 &= 1021.
 \end{aligned}$$

To handle the sign of the mantissa, approximately half of the maximal exponent number is subtracted from the exponent before evaluating  $2^e$ : the maximal number is  $\sum_{i=1}^{11} 2^{i-1} = 2047$ , and half of this

is  $2047/2 = 1023.5$ . In practice, the number  $\hat{e} = 1022$  is subtracted. Thus, the real exponent is  $e = 1021 - 1022 = -1$ . The number represented by the 8 bytes is thus:

$$0.666\ 666\ 666\ 666\ 666\ 629\ 66 \times 2^{-1} = 0.333\ 333\ 333\ 333\ 333\ 314\ 83.$$

We see that at the 16 first digits are correct.

## Appendix D

# Three Dimensional Plots and Plot Housekeeping\*

### D.1 Three dimensional plots

Let us consider the function

$$f(x, y) = \sin x \cos y \exp\left(-\frac{1}{10}(x^2 + y^2)\right),$$

which is displayed in fig. D.1. How can we plot this function using Matlab? We wish to plot the function in the region  $(x, y) \in [-2\pi, 2\pi] \times [-2\pi, 2\pi]$ . First we need to define the variation in  $x$  and  $y$ :

```
>> x = linspace(-2*pi,2*pi);  
>> y = linspace(-2*pi,2*pi);
```

Remember that these variables ( $\mathbf{x}$ ,  $\mathbf{y}$ ) are row vectors.

The simplest way to compute the functional value of  $z = f(x, y)$  is by using array operations. We need to compute  $z$  for all possible combinations of  $x$  and  $y$ , i.e. for  $[-2\pi, 2\pi] \times [-2\pi, 2\pi]$ . We do this by generating arrays  $\mathbf{X}$  and  $\mathbf{Y}$  as follows:

```
>> [X,Y] = meshgrid(x,y);
```

Both  $\mathbf{X}$  and  $\mathbf{Y}$  are arrays of size  $n_x \times n_y$ , where  $n_x = \text{length}(\mathbf{x})$  and  $n_y = \text{length}(\mathbf{y})$ . Array  $\mathbf{X}$  is composed of  $n_y$  columns with copies of row array  $\mathbf{x}$ , while array  $\mathbf{Y}$  is composed of  $n_x$  rows with copies of row array  $\mathbf{y}$ . The usefulness of this construction, is that if we take an arbitrary element  $\mathbf{X}(i, j)$  and  $\mathbf{Y}(i, j)$ , then these elements are the ones we need for computing  $\mathbf{Z}(i, j)$ :

```
>> Z = sin(X).*cos(Y).*exp(-(X.^2 + Y.^2)/10);
```

Now, we have computed  $z$  for a selected number of values of  $x$  and  $y$ , and the result is found in array  $\mathbf{Z}$ . There are a number of functions for plotting  $z$  as a function of  $x$  and  $y$ , depending on what we want to display. The most basic function is the `mesh` function:

```
>> mesh(X,Y,Z)
```

The result of this command is displayed in fig. D.2. Note that the default length of arrays generated using the `linspace` function, is 100. Thus, the graph in fig. D.2 is generated from  $100 \cdot 100 = 10^4$  data points, and the resulting picture file behind fig. D.2 is relatively large. Figure D.3 illustrates how we can interactively rotate the graph, in order to look for a more interesting viewpoint.

Sometimes, we want to see contour lines for the 3D plot: these are like iso-curves in a map:

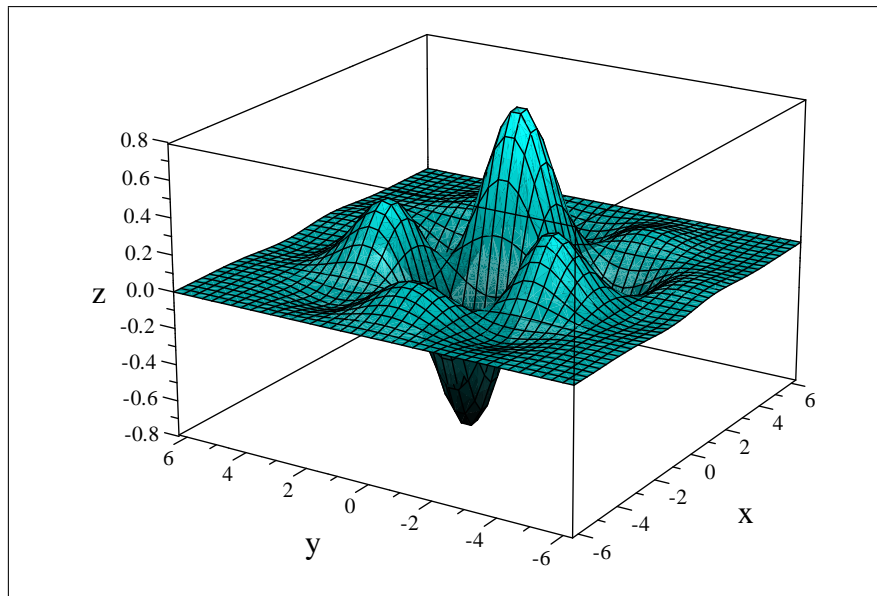


Figure D.1: Graph of function  $\sin x \cos y \exp\left(-\frac{1}{10}(x^2 + y^2)\right)$  generated using a CAS system (Computer Algebra System, here: MuPAD from within the word processor Scientific WorkPlace).

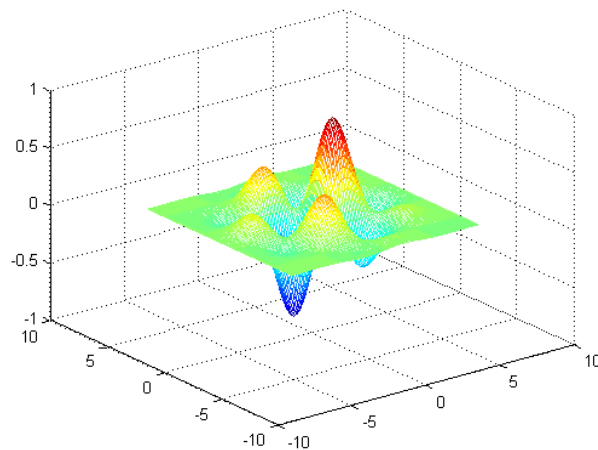


Figure D.2: Graph of function  $f(x, y) = \sin x \cos x \exp\left(-\frac{1}{10}(x^2 + y^2)\right)$  generated in Matlab.

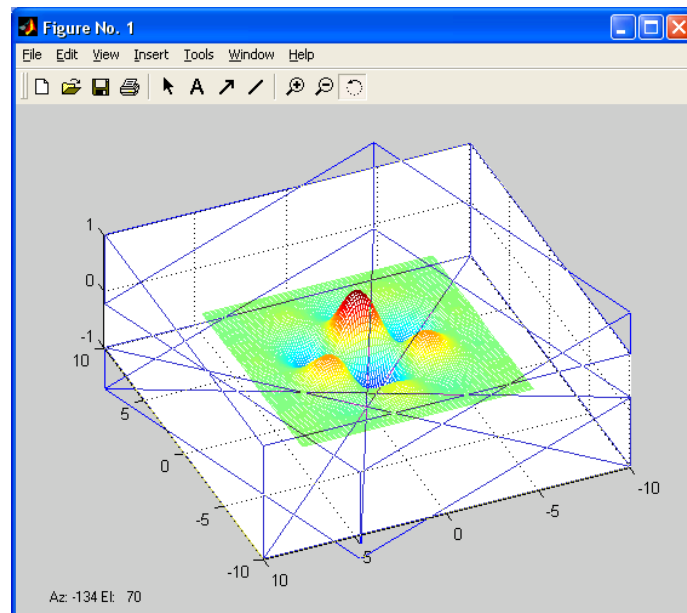


Figure D.3: In order to rotate 3D plots: click the rotation icon on the toolbar, click-and-hold in the plot, and “rotate” using the mouse.

```
>> meshc(X,Y,Z)
```

Note the small, but important difference between functions `mesh` and `meshc`. The resulting plot is shown in fig. D.4.

If we only want to see the contour lines, this is achieved using function `contour`:

```
>> contour(X,Y,Z,20)
```

draws contour lines at 20 levels, see fig. D.5.

Matlab function `surf` works similarly to function `mesh`.

**Exercise D.1** Check out the help files for the following Matlab 3D functions: `plot3`, `contour`, `contourf`, `contour3`, `mesh`, `meshc`, `meshz`, `surf`, `surfc`, `waterfall`, `bar3`, `bar3h`, `pie`, `fill3`, `comet3`, `scatter3`, `stem3`. ■

It should be noted, that in many cases, it is advantageous to plot 3D results in two dimensions, e.g.:

```
>> plot(x,Z)
```

The result of this command is shown in fig. D.6, and should be compared to fig. D.2. Perhaps the current example is not the best example, but it is often difficult to understand 3D plots.

## D.2 Housekeeping

Sometimes, it is necessary to clear the content of a figure. With the figure window selected, issue the command `clf`: the figure window is still open, but it is greyed out. In order to close a chosen figure window, issue command `close`.

It is possible to open more than one figure window at the same time. This is done by issuing the `figure` command. When this command is issued, the figure window is opened, and a so-called

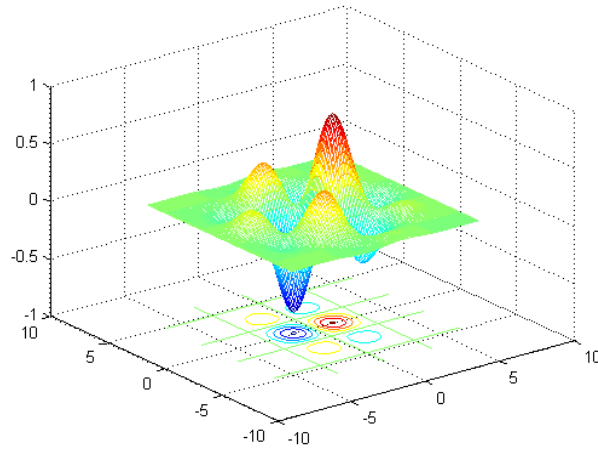


Figure D.4: Graph of function  $f(x, y) = \sin x \cos x \exp(-\frac{1}{10}(x^2 + y^2))$  generated in Matlab, with added contour lines.

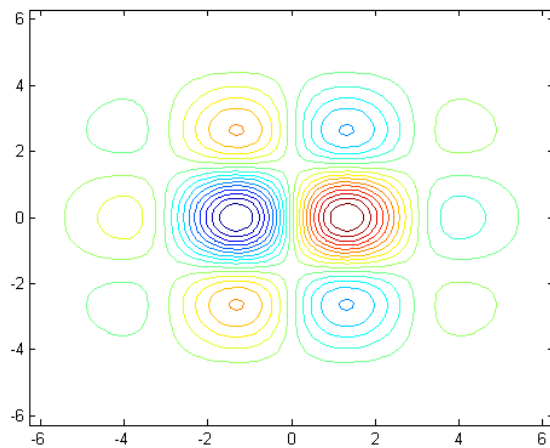


Figure D.5: Contour lines of function  $f(x, y) = \sin x \cos x \exp(-\frac{1}{10}(x^2 + y^2))$ .



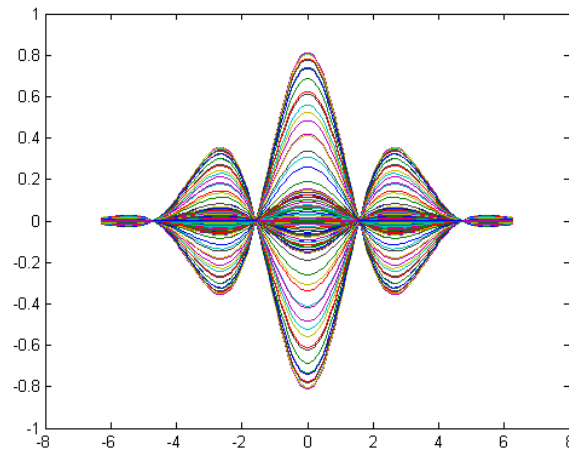


Figure D.6: Graph of function  $f(x, y) = \sin x \cos x \exp\left(-\frac{1}{10}(x^2 + y^2)\right)$ , mapped on the  $x$ - $z$  plane.

*handle* for the figure is returned. This can later on be used to address the figure using the command `figure(<handle>)`, which will make the figure window with the relevant handle the selected window. The handle for the figure can also be used to close a figure (`close(<handle>)`) — `clf` can only clear the selected/current window. The following Matlab session illustrates the use of these commands:

```
>> f1 = figure;
>> f2 = figure;
>> f3 = figure;
>> figure(f1);
>> plot(x,y);
>> figure(f3);
>> plot(x,exp(-x));
>> figure(f2);
>> plot(x,cos(x));
>> figure(f3);
>> clf;
>> close(f3)
>> close(f2)
```

It is also possible to enforce a certain number on a figure window by using the command `figure(N)` where  $N$  is an integer no. This number can then also be used in place of the handle no.

It is useful to be able to save figures from the command line. This can be done using function `saveas` with the following arguments: `saveas(H, 'FILENAME', 'FORMAT')`. Here,  $H$  is the handle of the figure, `FILENAME` is the file name with or without extension, and `FORMAT` is the file format. If the chosen format is `fig`, then it is also possible to open the figure for more manipulation in Matlab. For figures that are more or less finished, possible formats are `bmp`, `jpg`, etc.

The syntax for opening a figure, is `open <filename>`. The following example illustrates how the `saveas` function and the `open` commands may be used:

```
>> saveas(f1,'figure1','fig')
>> close(f1)
>> open figure1.fig
>> saveas(f1,'figure1','bmp')
>> saveas(f1,'figure1','jpg')
>> dir fig*
```

figure1.bmp figure1.fig figure1.jpg

## Appendix E

# Automation and String Operations\*

### E.1 Motivating example: generating sequence of plots

Based on the data in fig. 4.8 p. 68, suppose we want to produce a number of plots of the experimental data, and how well a polynomial model of order  $i$  fits the data when using the representation  $T(p)$ , e.g. as in fig. 4.11 p. 71. Let us suppose that we also want to save the figures to files which are named according to its content. The following Matlab session would do the job:

```
Import Wizard created variables in the current workspace.
>> T = data(3:end,1);
>> p = data(3:end,2);
>> pp = linspace(min(p),max(p));
>> fig1 = figure;
>> % ===== Ready to generate fig. for first order fit =====
>> plot(p,T,'kx')
>> xlabel('p [atm]')
>> ylabel('T [C^{\circ}]')
>> title('Temperature vs. pressure for saturated steam')
>> Tpar = polyfit(p,T,1);
>> hold on;
>> plot(pp,polyval(Tpar,pp),'k-')
>> saveas(fig1,'satvappfit1')
>> clf
>> % ===== Ready to generate fig. for second order fit =====
>> plot(p,T,'kx')
>> xlabel('p [atm]')
>> ylabel('T [C^{\circ}]')
>> title('Temperature vs. pressure for saturated steam')
>> Tpar = polyfit(p,T,2);
>> hold on;
>> plot(pp,polyval(Tpar,pp),'k-')
>> saveas(fig1,'satvappfit2')
>> dir satvappfit*

satvappfit1.fig  satvappfit2.fig
>> % ===== Ready to generate fig. for third order fit =====
>> % ... etc. ...
```

Clearly, it is a major job to generate figures for every model order up to, say, 60: we have 61 data points, which means that a polynomial model of order 60 implies interpolation. There must be a simpler way, i.e. it must be possible to automate the generation of these plots.

In order to understand how to automate the generation of these plots, we need to discuss strings and string operations, as well as automatic repetition of operations.

Table E.1: Some basic functions for operating on strings.

Function	Description
<code>s = 'abc'</code>	Creates a string <code>s</code> containing the character sequence <code>abc</code> .
<code>length(s)</code>	The number of characters in string <code>s</code> .
<code>blanks(n)</code>	Creates a string of <code>n</code> blank elements (spaces).
<code>s(i)</code>	If <code>s</code> is a string and <code>i</code> is a positive integer, <code>s(i)</code> is the string <code>'s(i)'</code> .
<code>char(i)</code>	If <code>i</code> is an integer representing the ASCII value of a character, <code>char(i)</code> is the resulting character string.
<code>[s1,s2]</code>	Concatenates strings <code>s1</code> and <code>s2</code> .
<code>strcat(s1,s1)</code>	Concatenates strings <code>s1</code> and <code>s2</code> , but neglects blank strings.
<code>deblank(s)</code>	Strips trailing blanks from the end of strings.
<code>class(obj)</code>	Responds with the type of the <code>obj</code> .
<code>isa(obj,'char')</code>	If <code>obj</code> is a string of characters, the result is boolean TRUE; otherwise FALSE.
<code>ischar(obj)</code>	If <code>obj</code> is a string of characters, the result is boolean TRUE; otherwise FALSE.
<code>isletter(s)</code>	Creates a row array of length <code>length(s)</code> , with value 1 for corresponding letters in <code>s</code> , and 0 otherwise.
<code>isspace(s)</code>	Creates a row array of length <code>length(s)</code> , with value 1 for corresponding spaces in <code>s</code> , and 0 otherwise.
<code>isequal(s1,s2)</code>	If strings <code>s1</code> and <code>s2</code> are equivalent, the result is boolean TRUE; otherwise FALSE.
<code>str2num(s)</code>	If <code>s</code> is a string of characters representing a floating point number, <code>str2num(s)</code> converts the string to the number.
<code>num2str(x)</code>	If <code>x</code> is a floating point number, <code>num2str(x)</code> is the character string for the number.

## E.2 Strings and string operations

It is useful to have some knowledge of strings in Matlab, since these are important for some types of task automation. A string is a sequence of characters. When defining the string, the sequence of characters must be surrounded by an apostrophe `'`:

```
>> clear
>> mystr1 = 'a';
>> mystr2 = 'abc';
>> whos
Name           Size           Bytes   Class
mystr1         1x1             2   char array
mystr2         1x3             6   char array
```

Grand total is 4 elements using 8 bytes

We see that each character in a string takes up 2 bytes of memory.

Some basic functions for operating on strings are shown in Table E.1.

Let us see how these functions work with a simple string:

```
>> mystring = 'John Donne';
>> length(mystring)
ans =
    10
>> find(mystring == ' ')
ans =
     5
```

```
>> mystr1 = mystring(1:5-1)
mystr1 =
John
>> mystr2 = mystring(5+1:end)
mystr2 =
Donne
>> [mystr1, ' ', mystr2]
ans =
John Donne
>> strcat(mystr1, ' ', mystr2)
ans =
JohnDonne
>> [mystr2, ', ', mystr1]
ans =
Donne, John
>> mystring(1)
ans =
J
>> mystring(2)
ans =
o
>> mystring(1:4)
ans =
John
>> mystring(end:-1:1)
ans =
ennoD nhoJ
```

In particular, note how we can change the sequence of characters in the string by using the command `mystring(end:-1:1)`.

Let us also see how strings and numbers can be mixed and matched:

```
>> mynum = pi
mynum =
    3.1416
>> mynumstr = num2str(mynum)
mynumstr =
    3.1416
>> class(mynum)
ans =
double
>> class(mynumstr)
ans =
char
>> isa(mynumstr, 'char')
ans =
    1
>> isa(mynum, 'char')
ans =
    0
>> ischar(mynumstr)
ans =
    1
>> isequal(mynum, mynumstr)
ans =
    0
>> isequal(mynumstr, '3.1416')
```

Table E.2: More advanced functions for operating on strings.

Function	Description
<code>lower(s)</code>	Converts any upper case character in string <code>s</code> to lower case.
<code>upper(s)</code>	Converts any lower case character in string <code>s</code> to upper case.
<code>strjust(s,type)</code>	Returns a justified string of string <code>s</code> according to <code>type = 'right', 'left',</code> or <code>'center'</code> .
<code>strrep(s,s1,s2)</code>	Replaces any occurrence of string <code>s1</code> in string <code>s</code> , with <code>s2</code> .
<code>findstr(s1,s2)</code>	Returns the starting indices of any occurrences of the shorter of the two strings <code>s1, s2</code> in the longer.
<code>strcmp(s1,s2)</code>	Returns 1 if strings <code>s1</code> and <code>s2</code> are the same, and 0 otherwise.
<code>strcmpi(s1,s2)</code>	Returns 1 if strings <code>s1</code> and <code>s2</code> are the same except for case, and 0 otherwise.
<code>strfind(s,s1)</code>	Returns the starting indices of any occurrence of string <code>s1</code> in string <code>s</code> .
<code>strncmp(s1,s2,n)</code>	Returns 1 if the first <code>n</code> characters of the strings <code>s1</code> and <code>s2</code> are the same, and 0 otherwise.
<code>strncmpi(s1,s2,n)</code>	Returns 1 if the first <code>n</code> characters of the strings <code>s1</code> and <code>s2</code> are the same except for case and 0 otherwise.
<code>eval(s)</code>	Execute string <code>s</code> with MATLAB expression.

```
ans =
     1
>> isequal(mynum,pi)
ans =
     1
>> myprod = [mynumstr, '*2']
myprod =
3.1416*2
>> str2num(myprod)
ans =
     6.2832
```

Some more advanced functions for operating on strings are shown in Table E.2.

It is also possible to operate with *regular expressions* such as `'fig*'`, see `>>help regexp`. Some of the functions also accept other arguments — see the Matlab help system for more details.

### E.3 Example: automatic generation of figures

Let us see how we can automate the task discussed in Section E.1. We assume that we manually import the data and define variables `T` and `p`, and we want to store the figures in files `satvappfigI` — saturated vapor pressure figure where `I` takes on values from 1 to 60. The script file displayed in fig. E.1 does the job of generating 60 figures.

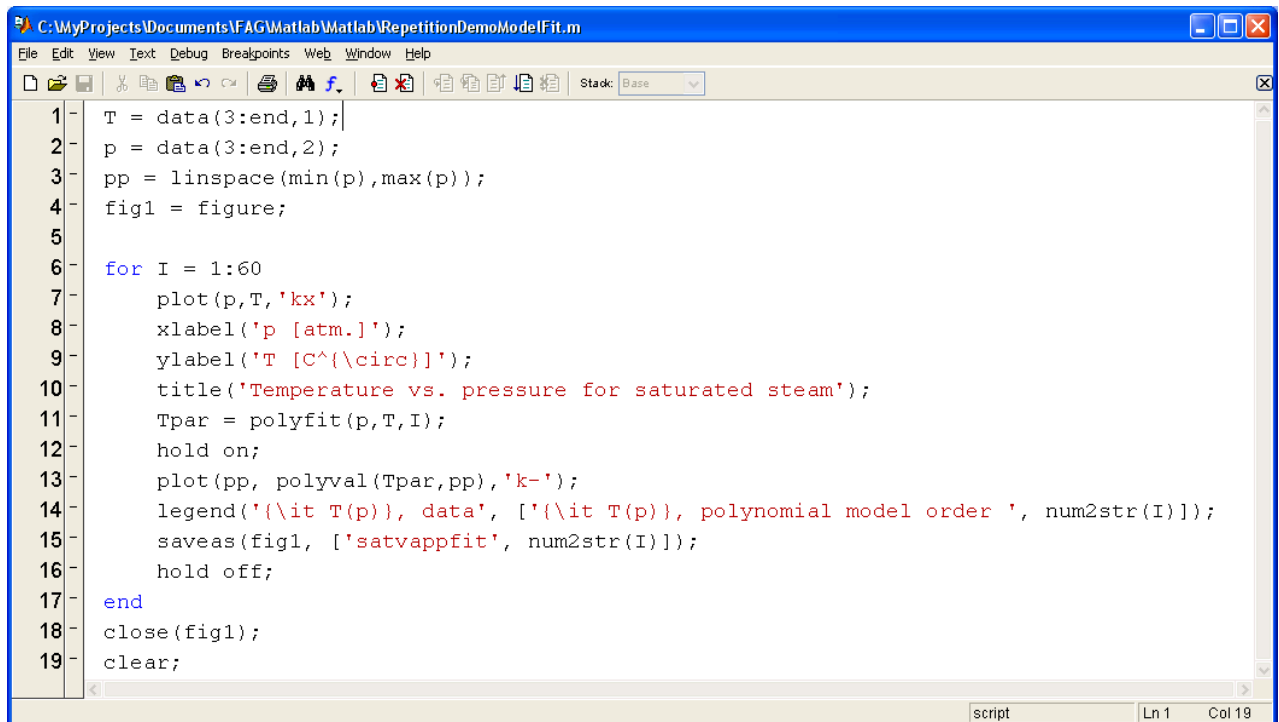
The following Matlab command runs the script:

```
Import Wizard created variables in the current workspace.
>> tic; RepetitionDemoModelFit; toc
```

Note that the Matlab command `tic` resets an internal clock, while command `toc` tells the elapsed time (in seconds) since the clock was reset.

The response to this command is:

```
Warning: Polynomial is badly conditioned. Remove repeated data points
         or try centering and scaling as described in HELP POLYFIT.
(Type "warning off MATLAB:polyfit:RepeatedPointsOrRescale" to suppress this warning.)
> In C:\MATLAB6p5\toolbox\matlab\polyfun\polyfit.m at line 75
   In C:\MyProjects\Documents\FAG\Matlab\Matlab\RepetitionDemoModelFit.m at line 11
```



```

C:\MyProjects\Documents\FAG\Matlab\Matlab\RepetitionDemoModelFit.m
File Edit View Text Debug Breakpoints Web Window Help
Stack: Base
1 T = data(3:end,1);
2 p = data(3:end,2);
3 pp = linspace(min(p),max(p));
4 fig1 = figure;
5
6 for I = 1:60
7     plot(p,T,'kx');
8     xlabel('p [atm.]');
9     ylabel('T [C^{\circ}]');
10    title('Temperature vs. pressure for saturated steam');
11    Tpar = polyfit(p,T,I);
12    hold on;
13    plot(pp, polyval(Tpar,pp),'k-');
14    legend('\it T(p), data', ['\it T(p), polynomial model order ', num2str(I)]);
15    saveas(fig1, ['satvappfit', num2str(I)]);
16    hold off;
17 end
18 close(fig1);
19 clear;
script Ln 1 Col 19

```

Figure E.1: Script file RepetitionDemoModelFit.m.

```

%
%... this warning is repeated a number of times for large values of I
%
elapsed_time =
    4.0750

```

During the execution of the script file, the following files have been created:

```
>> dir satvappfit*
```

```

satvappfit1.fig  satvappfit24.fig  satvappfit39.fig  satvappfit53.fig
satvappfit10.fig satvappfit25.fig  satvappfit4.fig   satvappfit54.fig
satvappfit11.fig satvappfit26.fig  satvappfit40.fig  satvappfit55.fig
satvappfit12.fig satvappfit27.fig  satvappfit41.fig  satvappfit56.fig
satvappfit13.fig satvappfit28.fig  satvappfit42.fig  satvappfit57.fig
satvappfit14.fig satvappfit29.fig  satvappfit43.fig  satvappfit58.fig
satvappfit15.fig satvappfit3.fig   satvappfit44.fig  satvappfit59.fig
satvappfit16.fig satvappfit30.fig  satvappfit45.fig  satvappfit6.fig
satvappfit17.fig satvappfit31.fig  satvappfit46.fig  satvappfit60.fig
satvappfit18.fig satvappfit32.fig  satvappfit47.fig  satvappfit7.fig
satvappfit19.fig satvappfit33.fig  satvappfit48.fig  satvappfit8.fig
satvappfit2.fig  satvappfit34.fig  satvappfit49.fig  satvappfit9.fig
satvappfit20.fig satvappfit35.fig  satvappfit5.fig   satvappfit10.fig
satvappfit21.fig satvappfit36.fig  satvappfit50.fig  satvappfit11.fig
satvappfit22.fig satvappfit37.fig  satvappfit51.fig  satvappfit12.fig
satvappfit23.fig satvappfit38.fig  satvappfit52.fig  satvappfit13.fig

```

We can check the fit of e.g. a 15-th order polynomial model by opening the figure:

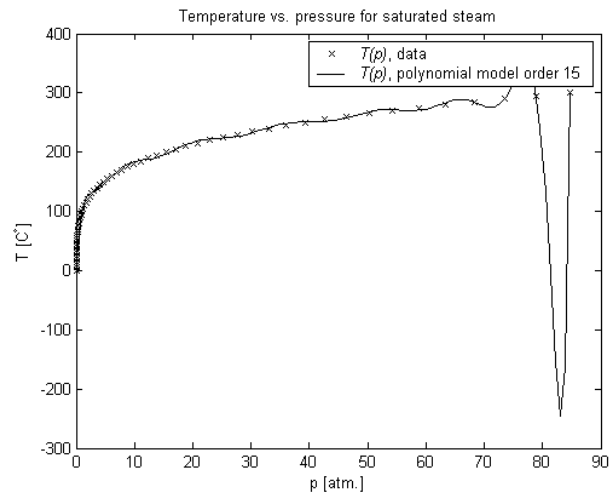


Figure E.2: The model fit of 15-th order polynomial model. Notice that although the model goes through most experimental data points, the model is very poor for interpolation at “high” pressures.

Table E.3: Some functions for number conversion.

Function	Description
<code>dec2bin(I)</code>	Converts a decimal integer to a binary string. Leading zeros are dropped.
<code>hex2dec(s)</code>	Converts hexadecimal string <code>s</code> (16 characters) to decimal integer.
<code>hex2num(s)</code>	Converts IEEE hexadecimal string to double precision number, where <code>s</code> is a 16 character string.

```
>> open satvappfit15.fig
```

The result is shown in fig. E.2. Notice that although the model goes through most experimental data points, the model is very poor for interpolation at “high” pressures. For higher order models, the interpolation properties of the polynomial model is even poorer.

## E.4 Example: hex 2 floating point

Let us consider the algorithm for converting hexadecimal numbers to floating point numbers, as indicated in the footnote p. 21. In doing so, it is convenient to introduce a handful of new Matlab functions, see Table E.3.

See also functions `dec2base`, `base2dec`, `dec2hex`, and `bin2dec` using the Matlab help system.

The following script does the job:

```
% Assume that a hexadecimal number hexnum is given as a
% string of 16 hexadecimal numbers

% First we find the binary number binnum that represents hexnum:

iter = length(hexnum);
binnum = '';
for I = 1:iter,
    % We find the 4 bit equivalent of each hex number, with leading zeros:
    bytedummy = '0000';
    bytenum = dec2bin(hex2dec(hexnum(I)));
    blen = length(bytenum);
    bytedummy(5-blen:end) = bytenum;
```



```

    bytenum = bytedummy;
    % We add 4 bit strings to previous string:
    binnum = [binnum, bytenum];
end

% Next we pick out the mantissa sign, the mantissa, and the exponent:
fsign = binnum(1);
fmantissa = binnum(13:64);
fexp1 = binnum(2:12);

% We then add the assumed binary digit "1" in front of the mantissa:
fmantissa = ['1', fmantissa];

% Next, we convert the mantissa to decimal numbers:
mlen = length(fmantissa);
fman = 0;
for I = 1:mlen
    fman = fman + str2num(fmantissa(I))*2^(-I);
end

% We also need to include the sign of the mantissa:
fman = fman*sign(0.5 - str2num(fsign));

% Next, we compute the value of the temporary exponent:
elen = length(fexp1);
fexp1 = fexp1(elen:-1:1); % Reversing the bit order to simplify evaluation
fex1 = 0;
for I=1:elen,
    fex1 = fex1 + str2num(fexp1(I))*2^(I-1);
end

% Next, we subtract the offset of the exponent:
fexp = fex1 - 1022;

% Finally, we compute the floating point number:
fnum = fman*2^fexp

```

In this script, we assume that we have available the hexadecimal representation of the floating point number as 16 hexadecimal digits, where each digit is either of  $\{0, 1, 2, \dots, 9, a, b, c, d, f\}$ . This hexadecimal representation must be stored in the variable `hexnum`.

Let us see how the script (named: `hex2num_script`) works:

```

>> decnum = 1/3
decnum =
    0.33333
>> format hex
>> decnum
decnum =
    3fd5555555555555
>> hexnum = '3fd5555555555555'
hexnum =
    3fd5555555555555
>> format short g
>> hex2num_script
fnum =
    0.33333

```

We see that we found the same answer using `hex2num_script`, as we started with.

Let us try another hex string where we do not know the answer:

```
>> hexnum = '701322c182aa1135'  
hexnum =  
701322c182aa1135  
>> hex2num_script  
fnum =  
7.4272e+231
```

We can check the result by using the built-in Matlab function `hex2num`:

```
>> hex2num(hexnum)  
ans =  
7.4272e+231
```

The result appears to be identical to what was found using the script `hex2num_script`.

# Appendix F

## Numerical methods with Matlab\*

### F.1 Numerical functions

The functions in Table F.1 take arrays as arguments.

Table F.1: Some numerical functions in Matlab.

Matlab function	Description
<code>diff</code>	difference operator
<code>gradient</code>	gradient using quadratic interpolating polynomials
<code>trapz</code>	numeric integral of vector, using the trapeze method
<code>cumtrapz</code>	As <code>trapz</code> , but returning vector of cumulative values
<code>polyfit</code>	least squares fit of polynomial to data
<code>polyval</code>	evaluate polynomial
<code>interp1</code>	interpolation, various methods, 1D
<code>interp2</code>	interpolation, various methods, 2D
<code>roots</code>	roots of polynomial

We met most of these functions in Part I, and will now apply them to some more examples. We shall learn something about the inner workings of several of these functions later in the course *Numerical Mathematics with Matlab*. For now, we treat them as “black boxes” and just assume that they do their job properly.

**Exercise F.1** Generate the following vectors:

```
x = (0:0.2:10)';  
y = cos(x);
```

- Use `diff` to generate a vector `dy` containing approximate values of the derivative of  $\cos x$ .
- Plot `dy` together with  $-\sin x$  in the same plot.

NB! Note that `diff(y)` has one element less than `y`. What should we use as abscissa axis variable (`x`-axis variable) when plotting `dy`? ■

**Exercise F.2** Repeat the previous exercise using the `gradient` function to find approximate values of the derivative. ■

**Exercise F.3** Repeat Exercise F.1, but generate approximate values of the integral instead of the derivative. Plot these together with  $\sin x$ . ■

**Exercise F.4** Use `interp1` and `x` and `y` from Exercise F.1 to find the value of  $\sin(p/2)$ . ■

Table F.2: Basic binary operations among scalars  $s$  and matrices  $M$ .

Operation	Legal/Illegal	Interpretation
$s1+s2, s1-s2$	L	Addition/subtraction of scalars
$s1*s2, s1/s2$	L	Standard scalar product/division
$s+M, s-M$	I	Interpreted as $s*\text{ones}(\text{size}(M))+M$ , etc.
$s*M$	L	Standard multiplication by scalar
$s/M$	I	—
$M/s$	I	Interpretation: $M./(s*\text{ones}(\text{size}(M)))$
$M1+M2, M1-M2$	L	$M1$ and $M2$ must have identical size
$M1*M2$	L	No. of columns of $M1$ must equal no. rows of $M2$
$M1/M2$	I	—
$M^n$	L	Legal if $M$ is square matrix (including scalar)

## F.2 Arrays vs. Vectors and Matrices

In Part I, we considered mainly *arrays*, which we “defined” as tables where the content of each element is of the same type. We also considered unary functions on array elements, and basic binary operations between arrays. In the case of binary operations, we required the two arrays, e.g.  $A, B$ , to be of the same size, e.g.  $A \in \mathbb{C}^{m \times n}$  and  $B \in \mathbb{C}^{m \times n}$ . Furthermore, we defined addition, subtraction, multiplication, and division of arrays to be element-by-element operations. In Matlab, these are performed using the notation:

- array addition:  $A+B$
- array subtraction:  $A-B$
- array multiplication:  $A.*B$
- array division:  $A./B$

In Section A of Part I, we considered vector spaces and matrices. We saw that a wide-spread vectorspace is  $\mathbb{C}^n$ , which conveniently can be represented either by a row array ( $\mathbb{C}^{1 \times n}$ ) or by a column array ( $\mathbb{C}^{n \times 1}$ ); for simplicity, we will denote these as a row vector and a column vector, respectively. For vectors, we need to define vector addition and subtraction — these fit well with the concepts of array addition and subtraction ( $+$ ,  $-$ ). Furthermore, we need the operation of scalar multiplication. Since a scalar and a vector, considered as arrays, have different sizes, we can not use array multiplication as the vector multiplication operation. Instead, Matlab supports the binary operation  $*$  among a scalar and a vector. Next: additive and multiplicative identities fits well with the array representation of Matlab. Finally, association, distribution, and commutativity is built into the way Matlab interprets parentheses.

Matrices can also be represented by arrays (e.g.  $\mathbb{C}^{m \times n}$ ), where matrix addition and subtraction are identical to array addition and multiplication. It is also convenient to expand the meaning of the operator for scalar-vector multiplication ( $*$ ), to also denote the Cayley product of two matrices. In other words, the (Cayley) matrix product  $A*B$  is defined if the number of columns of matrix  $A$  is the same as the number of rows of matrix  $B$ , in other words: if  $A \in \mathbb{C}^{m \times k}$  and  $B \in \mathbb{C}^{k \times n}$ , then the product  $AB$  is well defined, and  $AB \in \mathbb{C}^{m \times n}$ .

There is no division operation between vectors or matrices, thus symbol  $/$  does not really have a meaning when vectors and matrices are involved.

Finally, it should be noticed that for binary operations between two *scalars*, addition, subtraction, multiplication, and division is represented by the symbols  $+$ ,  $-$ ,  $*$ , and  $/$ , respectively. Furthermore, Matlab has a special interpretation for some operations that are theoretically illegal. The operations are summarized in Table F.2.

Note that although matrix addition commutes (i.e.  $A+B = B+A$ ), in general, matrix multiplication does not commute (i.e.  $A \cdot B \neq B \cdot A$ ).

Also note that matrix transpose  $M^T$  is achieved as follows:  $M.'$ . If we want to simultaneously find the transpose and take the complex conjugate of the matrix elements (often written as  $M^H$  or  $M^*$ ), this is achieved as follows:  $M'.$

Table F.3: Basic Matrix operations.

Matlab function	Description
<code>rank</code>	Matrix rank, i.e. number of linearly independent rows/columns
<code>det</code>	Determinant, valid for square matrices.
<code>cond</code>	Condition number, measure of how close matrix is to singularity
<code>inv</code>	Matrix inverse, valid for square, non-singular matrices
<code>\</code>	Solve linear equations using Gauss elimination
<code>norm</code>	Matrix/vector norm, extension of Pythagorean length of vectors
<code>eig</code>	Eigenvalues and eigenvectors

## F.3 Matrix operations

### F.3.1 Basic operations

Some basic matrix functions are given in Table F.3.

**Exercise F.5** For the set of equations:

$$\begin{aligned} 4x_1 - 2x_2 + x_3 &= 15 \\ -3x_1 + x_2 + 4x_3 &= 8 \\ x_1 - x_2 + 3x_3 &= 13, \end{aligned}$$

formulate the equations as a matrix equation of form  $Ax = b$ . ■

**Exercise F.6** For matrix  $A$  of Exercise F.5:

- Find the rank of  $A$ .
- Find the determinant of  $A$ .
- Find the condition number of  $A$ .
- If possible, find the inverse of  $A$ . ■

**Exercise F.7** For vector  $b$  and matrix  $A$  in Exercise F.5:

- Find the norm of vector  $b$ .
- Find the norm of matrix  $A$ .
- Experiment with other norms for  $b$  and  $A$  than the default norm (Hint: use the help system), and compare the results. ■

**Exercise F.8** If possible:

- Find the solution  $x$  of the equation  $Ax = b$  in Exercise F.5 using the backslash operator `/`.
- Verify that you get the same result using the matrix inverse. ■

### F.3.2 Matrix factorizations/decompositions<sup>1</sup>

Matlab allows for the basic matrix factorizations/decompositions in Table F.4.

<sup>1</sup>These methods are used extensively in linear algebra.

Table F.4: Basic matrix factorizations in Matlab.

Factorization of $A$	Matlab command	Comment
$A = R^T R$	<code>R=chol(A)</code>	Cholesky factorization: $A$ is positive definite, $R$ is upper triangular
$PA = LU$	<code>[L,U,P] = lu(A)</code>	LU factorization: $A$ is square, $L$ is lower triangular, $U$ is upper triangular, $P$ is permutation matrix ( $P^T P = I$ )
$AE = QR$	<code>[Q,R,E] = qr(A)</code>	QR decomposition: $E$ is permutation matrix ( $E^T E = I$ ), $Q$ is unitary matrix ( $Q^H Q = I$ ), $R$ is upper triangular
$A = USV^H$	<code>[U,S,V] = svd(A)</code>	Singular value decomposition: $U$ and $V$ are unitary matrices ( $U^H U = I$ , $V^H V = I$ ), and $S$ is diagonal

### F.3.3 Vector space commands<sup>2</sup>

With  $A \in \mathbb{C}^{m \times n}$  and  $x \in \mathbb{C}^n$ ,  $Ax$  can be considered as the column vector  $Ax = \sum_{i=1}^n x_i a_i$ , where  $a_i \in \mathbb{C}^m$  is column  $i$  of matrix  $A$ . The subspace consisting of all possible vectors  $Ax$  (i.e. by varying  $x$ ) is known as the *column space* of matrix  $A$ ,  $\mathcal{R}(A)$ . This subspace has the same number of so-called basis vectors, as the rank of  $A$ , rank  $A$ .

- The command `orth(A)` responds with a matrix with columns which constitute a complete set of basis vectors for  $\mathcal{R}(A)$ .

The particular matrix equation  $Ax = 0$  may have many solutions. The set of possible solutions forms a subspace of  $\mathbb{C}^n$  which is known as the *nullspace* of  $A$ , denoted  $\mathcal{N}(A)$ . The nullspace can be characterized by a complete set of basis vectors. With  $A \in \mathbb{C}^{m \times n}$ , it can be shown that the number of basis vectors in  $\mathcal{N}(A)$  is  $n - \text{rank } A$ .

- The command `null(A)` responds with a matrix with columns which constitute a complete set of basis vectors for  $\mathcal{N}(A)$ .
- The command `null(A, 'r')` has the same response as `null(A)`, but with operation `null(A, 'r')`, the resulting basis vectors are rational numbers.

The concepts *column space*  $\mathcal{R}(A)$  and *nullspace*  $\mathcal{N}(A)$  are used in order to assess the solvability of equations  $Ax = b$ . In particular:

1. A solution exists for equation  $Ax = b$  if and only if  $b \in \mathcal{R}(A)$ . We have that  $b \in \mathcal{R}(A) \Leftrightarrow \text{rank}[A, b] = \text{rank } A$ .
2. If a solution exists for  $Ax = b$ , then the solution is unique if and only if  $\mathcal{N}(A) = \{0\}$  (i.e. the nullspace has zero basis vectors).

<sup>2</sup>These methods are used extensively in advanced linear algebra.

## Appendix G

# Extra parameters and anonymous functions\*

Before going into more details about how to use function functions, let us consider how anonymous functions can simplify the *interface* between our own functions and built-in Matlab functions.

Many of the built-in Matlab functions, e.g. in Table 15.1, accept extra input parameters. As an example, the `quad` function accepts the more general form:

```
quad(fh1, a, b, tol, trace, P1, P2, ...)
```

Here, the first input argument is the function handle to the function we want to integrate (`fh1`), the second input argument is the lower integration limit (`a`), the third input argument is the upper integration limit (`b`), the fourth input argument is a user specified integration tolerance (`tol`), the fifth input argument specifies whether the user wants to see the intermediate results during the iteration for a solution (`trace`), while the remaining input arguments are extra parameters to the function referenced by the function handle (`fh1`). If we do not want to use parameters `tol` and `trace`, we can replace each of these by `[]`:

```
quad(fh1, a, b, [], [], P1, P2, ...)
```

We can alternatively use extra parameters with `quad` by introducing an anonymous function as an interface:

```
fh2 = @(x) fh1(x, P1, P2, ...);  
quad(fh2, a, b)
```

Using the anonymous function as an interface function, the notation and use of function functions with extra arguments becomes simpler.

Let us illustrate this idea with an example. We shoot up a projectile with an initial velocity  $v_0$  at an angle  $\varphi$  with the horizon. The velocities in the horizontal and vertical directions are

$$\begin{aligned}v_x &= v_0 \sin \varphi \\v_y &= v_0 \cos \varphi - gt\end{aligned}$$

The speed is thus

$$v = \sqrt{v_x^2 + v_y^2} = \sqrt{v_0^2 + g^2 t^2 - 2v_0 g t \cos \varphi}$$

If the ground is in the horizon plane, the time before the projectile hits the ground is the solution for  $t > 0$  of  $\int_0^{t_{\max}} v_y(t) dt = 0$  or  $v_0 t_{\max} \cos \varphi - \frac{1}{2} g t_{\max}^2 = 0$ :

$$t_{\max} = \frac{2v_0}{g} \sin \varphi$$

We now want to find the arc length of the trajectory, defined as

$$L = \int_0^{t_{\max}} v dt$$

If we were only interested in the result for one set of specified values for  $v_0$ ,  $\varphi$ , and  $g$ , then we could hard code those values in a function and integrate it using `quad`. But it is more flexible if we write a function with those parameters as input variables:

```
function v = speed(t, v0 , phi, g)
% Speed of projectile shot up with initial velocity v0 at angle
% phi with the horizon. g: gravitational acceleration
vx = v0*cos(phi);
vy = v0*sin(phi)-g*t;
v = sqrt(vx^2+vy.^2);
```

There are two ways of integrating this function using `quad`. Without using an anonymous function as interface, we have:

```
>> v0 = 10;g = 10;phi = pi/4;
>> tmax = 2*v0*sin(phi)/g;
>> L = quad(@speed, 0, tmax, [], [], v0, phi, g)
L =
  11.4779
```

This works fine, but the user has to enter values (or empty arrays as placeholders) to ensure that the parameter arguments come at the right place in the function call. Another, perhaps more serious, problem is that the method for passing extra parameters when writing your own function functions is not so straightforward. In fact, most users find it a constant source of frustration.

In Matlab 7, we can alternatively use the anonymous function facility to define a new interface function:

```
>> f = @(t) speed(t, v0, phi, g)
f =
  @(t) speed(t,v0,phi,g)
```

Next, we can integrate `f` to find the arc length:

```
>> L = quad(f,0,tmax)
L =
  11.4779
```

We can also define the function inside the call to `quad`. In this example, we change the angle a bit, just to demonstrate that the parameters do indeed influence the result:

```
>> phi = pi/3;
>> L = quad(@(t)speed(t,v0,phi,g),0,tmax)
L =
  9.1949
>> L = quad(f,0,tmax)
L =
  11.4779
```

Note that `f` is unaffected by the value change of the parameter.



## Appendix H

# DAE solvers in Matlab\*

Two of the Matlab ODE solvers can also solve DAEs: `ode15s` and `ode23t`. Regarding the damped mass-spring system, we could alternatively have posed this as a DAE system:

$$\frac{dx}{dt} = v \quad (\text{H.1})$$

$$\frac{dE}{dt} = Fv - \mu v^2 \quad (\text{H.2})$$

$$P = mgx + \frac{k}{2}(x - x_0)^2 \quad (\text{H.3})$$

$$E = K + P \quad (\text{H.4})$$

$$K = \frac{1}{2}mv^2. \quad (\text{H.5})$$

However, this model is *not entirely equivalent* to the model in eqs. 16.1 – 16.2: with the given initial values,  $(x, v)_{t=0} = (1.5, 0)$ , we see that  $dx/dt = 0$  and  $dE/dt = 0$ , hence the model in eqs. H.1 – H.5 will predict that the system stays at rest!

Instead, we choose to consider the following DAE:

$$\begin{aligned} \frac{dx}{dt} &= v \\ \frac{dv}{dt} &= -\frac{k}{m}(x - x_0) - \frac{\mu}{m}v - g + \frac{F}{m} \\ 0 &= P - mgx - \frac{k}{2}(x - x_0)^2 \\ 0 &= K - \frac{1}{2}mv^2 \\ 0 &= E - K - P. \end{aligned}$$

This model obviously is equivalent: we simply use the same model for  $x$  and  $v$ , and add three algebraic equations in order to compute  $P$ ,  $K$  and  $E$ . These equations constitute a DAE: a differential algebraic equation with 2 ordinary differential equation, and 3 additional algebraic equations. The standard form for DAEs in Matlab is

$$M \frac{dy}{dt} = f(t, y).$$

Here, we define

$$y = (x, v, P, K, E)^T.$$

In standard form, we have:

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} \frac{dx}{dt} \\ \frac{dv}{dt} \\ \frac{dP}{dt} \\ \frac{dK}{dt} \\ \frac{dE}{dt} \end{pmatrix} = \begin{pmatrix} v \\ -\frac{k}{m}(x - x_0) - \frac{\mu}{m}v - g + \frac{F}{m} \\ P - mgx - \frac{k}{2}(x - x_0)^2 \\ K - \frac{1}{2}mv^2 \\ E - K - P \end{pmatrix}.$$

Notice that we have to specify  $F$  as a function of  $t$  or  $y$  in order to have the system in standard form. Notice also that we have a choice as to which of the algebraic equations belongs to  $dv/dt$ ,  $dK/dt$ , and  $dP/dt$ . We can make that choice more or less as we like.

Since Matlab will attempt to compute  $y$ , notice that this formulation allows for automatic computation of all quantities of interest:  $x, E, v, K, P$ , and that no post processing is necessary.

The main problem with solving DAEs is that we need *consistent conditions for  $y(0)$*  and  $dy(0)/dt$ . Here, it is simplest to specify  $x(0)$  and  $v(0)$ , and to compute  $E(0)$ ,  $K(0)$ , and  $P(0)$  from the algebraic equations. Furthermore, we need a strategy for finding  $dy(0)/dt$ . We need to require that

$$M \frac{dy(0)}{dt} = f(0, y(0)),$$

which is a simple, linear algebra equation of type  $Ax = b$ , where  $A = M$ , the unknown is  $x = dy(0)/dt$ , and  $b = f(0, y(0))$ . We can solve this equation using the pseudo inverse of  $M$ , which in Matlab is computed through the use of function `pinv`.

Here is how we call DAE solvers for the damped mass-spring system (script file `dampmsdae.m`):

```
% Script for simulating damped mass-spring system

% Necessary data
Tfin = 150;
data;
xinit = 1.5;
v0 = 0;
P0 = m*g*xinit + k*(xinit-x0)^2/2;
K0 = m*v0^2/2;
E0 = K0 + P0;

% Setting up initial values:
y0 = [xinit, v0, P0, K0, E0]';

% Setting up the M matrix in M*dy/dt = f(t,y):
M = zeros(length(y0));
M(1,1) = 1;
M(2,2) = 1;

% Computing consistent value for dy(t0)/dt:
Mdy0dt = dmsdae(0,y0);
dy0dt = pinv(M)*Mdy0dt;

% Specifying options for the ode solver:
opt = odeset('Mass',M,'MassSingular','yes', 'MstateDependence', 'none',...
    'InitialSlope', dy0dt, 'AbsTol', 1e-6, 'RelTol', 1e-3);

% Simulating system - notice that we include the options opt
tic;
[T,Y] = ode15s(@dmsdae, [0,Tfin], y0, opt);
toc
```

Next, we need to define function `dmsdae`:

```
function fy = dmsdae(t,y)
%
% Defining data
data;
% Naming variables
x = y(1);
```

```

v = y(2);
P = y(3);
K = y(4);
E = y(5);
% Defining elements of the vector field
fx = v;
fv = -k*(x-x0)/m - mu*v/m - g + F/m;
fP = P - m*g*x - k*(x-x0)^2/2;
fK = K - m*v^2/2;
fE = E - K - P;
% Setting up the vector field
fy = [fx, fv, fP, fK, fE]';

```

The DAE is solved by running the script file `dampmsdae`. Solving the DAE (variables:  $x, v, P, K, E$ ) takes approximately 3 times as long as solving the simple ODE (variables:  $x, v$ ), but then we find more variables.

To sum up:

- Describing models as DAEs reduces<sup>1</sup> the necessary amount of formula manipulation.
- We can reduce/eliminate the necessary work to find auxiliary variables<sup>2</sup>, if we describe models as DAEs.

---

<sup>1</sup>Note: in this case, we didn't really save much manipulation, but normally we can save work by formulating models as DAEs.

<sup>2</sup>In this case, we eliminated the necessary post processing to find  $P$ ,  $K$ , and  $E$ .



**Part VI**

**References**



# Bibliography

Gentle, J. E. (1998), *Numerical Linear Algebra for Applications in Statistics*, Springer-Verlag, New York.

Hanselman, D. C. & Littlefield, B. L. (2005), *Mastering MATLAB 7*, Prentice Hall, Upper Saddle River, NJ. ISBN: 0-13-143018-1.

Higham, D. J. & Higham, N. J. (2000), *Matlab Guide*, SIAM, Philadelphia.

Lamport, L. (1986), *LaTeX. A Document Preparation System*, Addison-Wesley, Reading, Massachusetts.

Levenspiel, O. (1972), *Chemical Reaction Engineering*, second edn, John Wiley & Sons, New York.

# Index

`ans`, 31

`dampedspring`, 145

`dampms`, 144

`eulersim`, 142

history list

tidligere kommandoer, 32

`load`, 14

ODE solver

Explicit Euler Method, the, 143

`save`, 14

`who`, 31

`whos`, 31



