These lectures were compiled for the course “Computer Mathematics for the Engineer: Efficient Computation and Symbolic Manipulation” (3/5 cr. ECTS) aimed at students on the advanced or graduate levels at Åbo Akademi University. The course is arranged by the Laboratory of Thermal and Flow Engineering at Åbo Akademi University and supported by the Finnish national graduate programme “Graduate School in Chemical Engineering”.

The material included in these notes is based on the literature cited below and should not be regarded as a complete user’s manual for the computer-aided mathematical tools presented in the course, but merely as an introduction to how these can be used for improving and assisting day-to-day mathematical chores of the modern engineer. We begin with a minimal introduction of the software packages adopted for the course and continue with details from various engineering topics. To encourage the reader to engage in a “hands-on” learning process, we tend to give only the necessary input for different themes and leave it to the reader to try the various cases for him/herself. Comments on the expected output is provided in most cases, however.

Åbo, October 2014,

[Signature]
Bibliography


Contents

1 An Overview of wxMaxima 11

1.1 Introduction ............................................. 11
1.2 Examples ................................................. 12
1.3 Expressions ............................................... 13
  1.3.1 Utility Functions ................................. 18
  1.3.2 Expression Construction ...................... 19
1.4 Evaluation ................................................. 20
  1.4.1 Evaluation of Atomic and Non-Atomic Expressions ... 20
  1.4.2 Modifications of Evaluations .................. 20
  1.4.3 Defaults and Argument Evaluation .............. 21
  1.4.4 Callable Functions ............................... 22
  1.4.5 Automatically Quoted Arguments ............... 22
  1.4.6 Preventing Evaluation ............................ 24
  1.4.7 Extra Evaluation ................................. 24
  1.4.8 Arguments ........................................ 26
  1.4.9 Arguments ........................................ 27
1.5 Simplification ............................................ 27
  1.5.1 Examples of Maxima Simplification .......... 28
  1.5.2 Exceptions to Maxima Simplification ......... 29
1.6 Special Purpose Functions ............................... 29
  1.6.1 Applying arguments to functions ............ 29
  1.6.2 The mapping function ......................... 30
  1.6.3 The lambda-function ............................ 31
1.7 Built-in object types .................................... 31
  1.7.1 Lists ........................................ 31
  1.7.2 Matrices ........................................ 32
  1.7.3 Sets ........................................ 33
1.8 Maxima Programming ..................................... 34
  1.8.1 Branching ...................................... 36
  1.8.2 Iteration ...................................... 36
1.9 Some Useful Flags ...................................... 40
1.10 Summary ................................................ 42
## 2 Symbolic Calculation

2.1 Initialization ................................................. 43
2.2 Saving and Loading ........................................... 45
2.3 Functions and Equations ...................................... 47
  2.3.1 Functions .............................................. 47
  2.3.2 Equations - A Practical Application ..................... 49
2.4 Systems of Equations ......................................... 52
2.5 Evaluating Limits ............................................. 53
2.6 Differential Calculus .......................................... 54
  2.6.1 Computation of Solutions with the Laplace Transformation 57
  2.6.2 Differentiation Using the Chain Rule .................... 61
2.7 Integral Calculus ............................................... 65
  2.7.1 Integration Examples .................................... 66
  2.7.2 Integration by Parts .................................... 72
  2.7.3 Change of Variable ..................................... 73
2.8 Matrix Computations ........................................... 74
2.9 Visualization ................................................... 75
  2.9.1 Publication-Quality Graphics for \LaTeX{} .................. 76
2.10 Numerical Data ............................................... 76

## 3 An Overview of Octave

3.1 Introduction ................................................... 77
  3.1.1 Help! ................................................... 78
  3.1.2 Input Conventions ...................................... 78
  3.1.3 Variables and Standard Operations ....................... 78
3.2 Vector and Matrix Operations ................................ 79
  3.2.1 Vectors ............................................... 79
  3.2.2 Matrices .............................................. 79
  3.2.3 Basic Matrix Arithmetic ................................ 80
  3.2.4 Element-wise Operations ................................ 80
  3.2.5 Indexing and Slicing ................................... 81
  3.2.6 Solving Linear Systems of Equations ..................... 81
  3.2.7 Inverses, Decompositions and Eigenvalues ................ 82
  3.2.8 Testing for Zero Elements ............................... 82
3.3 Control Structures ............................................ 82
  3.3.1 Functions .............................................. 82
  3.3.2 Global Variables ....................................... 83
  3.3.3 Loops .................................................. 83
  3.3.4 Branching .............................................. 84
  3.3.5 Functions of Functions .................................. 84
  3.3.6 Efficiency Considerations ............................... 85
3.4 Input and Output ............................................... 85
3.5 Graphics ....................................................... 86
  3.5.1 2D Graphics ............................................ 86
  3.5.2 3D Graphics ............................................ 86
  3.5.3 Commands for 2D and 3D Graphics ......................... 87
## 4 Numerical Computation

4.1 Preliminaries .............................................. 89
  4.1.1 Running Octave ........................................ 89
  4.1.2 Editing What You Have Typed .......................... 89
  4.1.3 Startup Files ......................................... 90
  4.1.4 Quitting Octave ....................................... 90
  4.1.5 Help and Documentation ............................... 91

4.2 Functions and Scripts .................................... 91
  4.2.1 Defining Functions ................................. 92
  4.2.2 Multiple Return Values .............................. 94
  4.2.3 Variable-length Argument Lists ..................... 95
  4.2.4 Variable-length Return Lists ....................... 96
  4.2.5 Returning From a Function .......................... 97
  4.2.6 Function Files ....................................... 97
  4.2.7 Subfunctions ......................................... 98
  4.2.8 Private Functions ................................... 98
  4.2.9 Script Files ......................................... 99
  4.2.10 Function Handles, Inline Functions, and Anonymous Functions ................................................. 100

4.3 Equations and Optimization ............................... 101
  4.3.1 Nonlinear Equations .................................. 101
  4.3.2 Optimization ......................................... 103

4.4 Boundary/Initial Value Problems ......................... 105
  4.4.1 Integrating Differential Equations ................. 105

4.5 Partial Differential Equations ........................... 106

4.6 Integration ................................................ 106
  4.6.1 Functions of One Variable .......................... 106
  4.6.2 Functions of Multiple Variables .................... 108

4.7 Interpolation .............................................. 109
  4.7.1 One-Dimensional Interpolation ....................... 109
  4.7.2 $n$-Dimensional Interpolation ......................... 111

5 Exercises .................................................. 113

5.1 wxMaxima .................................................. 113
  5.1.1 Inverse Functions .................................... 113
  5.1.2 Limits ................................................ 114
  5.1.3 Differentiation ........................................ 114
  5.1.4 Integration ............................................ 115
  5.1.5 Integral and Differential Equations .................. 116
  5.1.6 Linear Algebra ....................................... 117

5.2 Octave .................................................... 117
  5.2.1 Linear Algebra ....................................... 117
  5.2.2 Timing ............................................... 118
  5.2.3 Stability Functions of BDF-integrators ............ 119
### 5.2.4 3D Plotting ........................................ 120
### 5.2.5 Hilbert Matrix ..................................... 121
### 5.2.6 Least Square Fit of a Straight Line ............... 121
### 5.2.7 Numerical Differentiation .......................... 122
### 5.2.8 Trapezoidal Rule ................................ 122
### 5.2.9 Optimization ...................................... 123
### 5.2.10 Sequences ......................................... 123

#### 6 Course problems 125

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Single Nonlinear Equation</td>
<td>125</td>
</tr>
<tr>
<td>6.2 Another Single Nonlinear Equation</td>
<td>126</td>
</tr>
<tr>
<td>6.3 A Set of Linear Equations</td>
<td>127</td>
</tr>
<tr>
<td>6.4 Nonlinear Regression</td>
<td>129</td>
</tr>
<tr>
<td>6.5 A Set of Nonlinear Equations</td>
<td>130</td>
</tr>
<tr>
<td>6.6 A Set of Ordinary Differential Equations</td>
<td>131</td>
</tr>
<tr>
<td>6.7 Another Set of Ordinary Differential Equations</td>
<td>132</td>
</tr>
<tr>
<td>6.8 A Second-Order Ordinary Differential Equation</td>
<td>134</td>
</tr>
<tr>
<td>6.9 Ordinary Differential Equations and Algebraic Equations</td>
<td>134</td>
</tr>
<tr>
<td>6.10 A Control Loop</td>
<td>135</td>
</tr>
<tr>
<td>6.11 Stiff Differential Equations</td>
<td>137</td>
</tr>
<tr>
<td>6.12 Partial Differential Equations</td>
<td>138</td>
</tr>
<tr>
<td>6.13 Integral Equations</td>
<td>139</td>
</tr>
<tr>
<td>6.14 Special Topics</td>
<td>141</td>
</tr>
<tr>
<td>6.14.1 Perturbation Theory</td>
<td>141</td>
</tr>
<tr>
<td>6.14.2 Calculus of Variations</td>
<td>142</td>
</tr>
<tr>
<td>6.14.3 JWKB-approximation</td>
<td>143</td>
</tr>
<tr>
<td>6.14.4 Monte Carlo Methods</td>
<td>145</td>
</tr>
<tr>
<td>6.14.5 Coding Theory</td>
<td>146</td>
</tr>
</tbody>
</table>
Background

The use of advanced mathematical methods has become increasingly significant for many modern engineering disciplines. As many of these methods nowadays are computer-aided, their efficient use requires apart from underlying theoretical knowledge also familiarity with relevant software packages. The aim of these notes is to combine these two into an engineering introduction to computer-aided mathematics, with emphasis on symbolic manipulation.

We will use the freeware applications “wxMaxima” for symbolic calculation and “Octave” for numerical calculation. Our main focus for both of these is based on command-driven use, although the former of these also features interactive menus and initialization windows. The reason for this is that menu-driven use of the software is to large extent self-instructive and our preference to learn how to program batch computations later on. Therefore, we choose to leave the details for using integrated menus to the interested student and proceed to a survey of the relevant commands involved in the themes treated in this course. In our attempt to awake interest for further learning, we will start with a “minimal” introduction to Maxima and later to Octave.

Both Maxima and Octave invoke the public-domain graphics package Gnu-plot for visualizing computation results. The graphs produced this way cannot be as readily customized as by using Gnuplot in a standalone fashion, however. For example, if graphs are intended for use in a \LaTeX-document, correct fonts and formulas can be produced by generating a “.tex”-figure and including this file in the \LaTeX-document. Brief instructions on how to do this is given below.
Chapter 1

An Overview of wxMaxima

1.1 Introduction

wxMaxima is a system, consisting of the core code package Maxima that does the actual computations and an interactive user interface based on so called x-widgets. The main working purpose is to handle expressions, such as $x + y$, $\sin(a + b\pi)$, and $uv - vu$. It is up to the user whether an expression is meaningful or not, however, manipulations can be done without assigning numerical values to variables and parameters.

Maxima belongs to a category of mathematics software known as computer algebra systems – or CAS for short. This implies that Maxima is a tool primarily for manipulation of mathematical objects as symbols rather than as numerical entities. The package Octave, which will be introduced later during the course, is more suitable for doing numerical computation.

Running wxMaxima opens an interactive window with several different menus and shortcuts across the headboard. These, which are designed for utilities as well as for different types of calculations, will be discussed in detail later. Commands can be entered both through submenus invoked by these and by using the command prompt. In earlier versions of wxMaxima, the command prompt is visible throughout, however, in later versions various prompts are chosen from “Cell” in the “Edit”-menu with shortcuts:

- F5 New Input Cell
- F6 New Text Cell
- F7 New Subsection Cell
- F8 New Section Cell
- F9 New Title Cell

For these, hitting “Enter” in the earlier version corresponds to choosing “Evaluate All Cells” from the same submenu or the shortcut Ctrl-R. For instance, the result
CHAPTER 1. AN OVERVIEW OF WXMAXIMA

Figure 1.1: The wxMaxima graphical interface.

displayed in Figure 1.1 can be rendered in \LaTeX by selecting “Display TeX-form” from the “Maxima”-menu to give:

\[ \sqrt{\pi} \left( \left( \sqrt{2} i + \sqrt{2} \right) \sin 1 + \left( \sqrt{2} i - \sqrt{2} \right) \cos 1 \right) \text{erf} \left( \frac{\left( \sqrt{2} i + \sqrt{2} \right) \pi}{4} \right) + \left( \left( \sqrt{2} i - \sqrt{2} \right) \sin 1 + \left( \sqrt{2} i + \sqrt{2} \right) \cos 1 \right) \text{erf} \left( \frac{\left( \sqrt{2} i - \sqrt{2} \right) \pi}{4} \right) \]

1.2 Examples

In the following, we will take a look at some brief examples on the use of wxMaxima.

Example 1.2.1. The command “stardisp” controls the way wxMaxima displays the operation of multiplication. Setting

(\%i1) stardisp: true;

will show all multiplications with an asterisk. The value “false” shows products with a blank space, which can distracting in more complicated expressions.

Example 1.2.2. The sequence

(\%i2) V: 4/3 * \%pi * r^3;
1.3. EXPRESSIONS

assigns the value \(\frac{4\pi r^3}{3}\) to the name \(V\). Names in Maxima are also termed identifiers. The value of the name \(r\) (for radius) is “itself”, in place of a specific numerical value, at this point – that is, \(r\) is \(r\). Note that the number \(\pi\) is entered using the Maxima special character \(%\pi\). Now, we assign the value \(r = 10\) to the radius

\[
{\%i3}\ r:10;
\]

As seen, the way to assign a specific value to a name is by using a colon character (:) and not the symbol “equal to”. We can think of this as assigning “equality by definition”. So, we defined the value of \(V\) to be equal to the expression above and the radius equal to 10, demonstrating that names may be assigned different kinds of objects. It should be noted that \(V\) does not change after assigning a numerical value to the radius \(r\) - it is still an expression. If we want to evaluate this expression with the value assigned to \(r\), we use double single quotes according to:

\[
{\%i5}\ ''V;
\]

This results in \(r = 10\) being substituted into the expression for \(V\) and the numerical constants, except \(\pi\), being simplified (also try what happens if the assignment \(r = 10\) is done before assigning the expression to \(V\)!). All irrational numbers having similar special symbols, i.e. starting with \(%\) are treated like this in Maxima. In a situation where we want a numerical result, however, an approximation has to be used for \(\pi\). This is done as follows:

\[
{\%i6}\ ''V, numer;
\]

producing a floating-point number approximation for the volume \(V\).

1.3 Expressions

Essentially everything in Maxima is expressions, which can be divided into three types or classes:

1. Mathematical expressions
2. Object expressions
3. Programming constructs

A Maxima expression is built up from “atoms” and operators with corresponding arguments. When an expression is evaluated, the result is its value which can differ from the expression itself. This was illustrated above. For example, in the expression \(1 + 2\), we have the operator + (the binary operator of addition). The two arguments to this operator are literal numerals in the form of the two “atoms” 1 and 2. The expression \(\sin(x)\) consists of an operator with one argument, the single letter symbol \(x\) – an atom in the form of a literal name.
The value of \( x \) may be itself if no other value has been assigned – \( x \) is \( x \), or \( x \) is, e.g., 10 (from the input \( x:10 \)).

An atom can be a name (a character symbol like \( V \), \( x \) and \( r \) above), a string enclosed in quotation marks (e.g., "volume" and "radius"), or a numeral such as an integer or decimal number like 1, -5, 3.14159, etc.

Expressions in Maxima not consisting of atoms are termed “non-atomic” and can be represented as \( \text{op}(a_1, \ldots, a_n) \), where \( \text{op} \) is an operator and \( (a_1, \ldots, a_n) \) are its \( n \) arguments. This format is also used internally by Maxima, including the basic arithmetic operations such as addition, subtraction, multiplication and division which are input conventionally using the symbols \(+\) \( \ast\) \(-\) \(/\). These are “seen” by Maxima in the form \( \text{op}(a,b) \) – for example \( 1 + 2 \) is represents \( \text{op}(1,2) \), where "\text{op}" is the appropriate operation for addition. Similar operators are used for function evaluation, e.g., in connection with trigonometric and special functions.

More complex mathematical expressions may involve combinations of names, literal numerals, and operators. Literal numerals include constants with special Maxima names such as \( \%\pi \) and \( \%e \). As listed above, in Maxima objects are also expressions.

**Example 1.3.1.** A list \([a_1, \ldots, a_n]\) of \( n \) objects is an object expression. These can be created with the operation \( \langle \text{op} \rangle \text{list} = \) and arguments. In the case of a matrix object, which also is an expression, we have ( \( \text{op} = \text{matrix} \) ):

\[
\text{matrix}(\text{list}((a_1,1), (a_1,2), \ldots, (a_1,n)), \ldots, \text{list}((a_m,1), (a_m,2), \ldots, (a_m,n)))
\]

These arguments are non-atomic, they are list object expressions each of which may or may not be composed of atoms. The value of these expressions is the list object or the resulting matrix object.

**Observation 1.3.2.** While all expressions have a value, the value of an expression may be lost unless the expression is assigned a name using the the \( : \) operator. Merely entering an expression as Maxima input will result in the display of the value of the expression (unless suppressed), and then the object and its value passes out of existence. Sometimes this is okay, for example when Maxima is being used as a symbolic calculator. However, Maxima has a built-in feature that mitigates this issue significantly.

The special Maxima name \( \% \) is assigned the value of the last entered expression and may be used like any name assigned a value. Of course, the value of \( \% \) changes with each new expression entered as input. The values of each entered expression are saved in the special names prefixed as \( \%o \) with a sequential numeral for each line of output. These \( \%o \) names (e.g., \( \%o1, \%o2, \ldots, \%on \)) may be used in Maxima expressions as names with the values of the displayed output.

The downside to the frequent use of this feature is that these generic names generated by Maxima have little if any literal significance. One can easily recognize that \( V \) means volume and \( r \) means radius, but what does \( \%o*(\%o7-%o5)/\%o3 \) signify?
Programming constructs are also expressions. A code block is represented internally to Maxima as an expression with an operator and arguments. Hence, a conditional statement (if $a$ then $b$ elseif $c$ then $d$) is represented as if $(a, b, c, d)$ and a loop (for $a$ in $L$ do $S$) as $\text{mdoin}(a, L, S)$.

Maxima has two functions to help evaluate what kind an expression is. The function $\text{atom}(expr)$ takes an expression as an argument and indicates whether this is simply atomic. The function $\text{op}(expr)$ returns the overall operation of an expression if it is not simply atomic. Expressions of this class have values which vary according to the type of programming construct. For example, the value of a loop expression may be “done” while the value of a block expression is the value of the last argument listed.

**Example 1.3.3. Maxima expressions**

1. **Atoms are symbols, strings, and numbers.** Below, we have a various of these grouped into a list:

   $$\text{(%i9) } [a, \text{ foo, foo_bar, "Hello, world!"}, 42, 17.29];$$

2. **Mathematical expressions**

   $$\text{(%i10) } [a + b + c, a * b * c, \text{foo = bar, a*b < c*d}];$$

   These may simply be combinations of atoms and operators or may also be relations between expressions. For example $\text{foo = bar}$ is considered to be a mathematical expression. This expression does not say that $\text{foo}$ is equal to $\text{bar}$, instead it is merely a logical relation between (in this case two atoms) the names $\text{foo}$ and $\text{bar}$. The relation between the two names holds only when their values are the same, resulting in the value “true” and otherwise “false”.

3. **Object Expressions – Lists and matrices** The elements of a list or matrix can be any kind of expression, even another list or a vector/matrix.

   $$\text{(%i11) } L: [a, b, c, \%pi, \%e, 1729, 1/(a*d - b*c)];$$

   This input is a list object expression assigned to the name $L$. The elements of this list are all atoms except the last which is a mathematical expression. Now, look at the input.

   $$\text{(%i12) } L2: [a, b, [c, \%pi, [\%e, 1729], 1/(a*d - b*c]));$$
This is a different list object expression assigned to the name \( L_2 \). The last element is itself a list object consisting of two atoms, another list object and a mathematical expression. These objects result in index names being created when assigned to a name. The values are accessed by appending brackets to the object name:

\[
(\%i13) \ L[7];
\]

which returns the seventh element of the list \( L \). Similarly,

\[
(\%i14) \ L2[3];
\]

returns the third element of the list \( L_2 \), which is in itself a list. The assignment

\[
(\%i15) \ M:\text{matrix} ([\%pi, 17], [29, \%e]);
\]

defines the matrix \( M = \begin{bmatrix} \pi & 17 \\ 29 & \%e \end{bmatrix} \). It is also possible to assign matrices with elements consisting of submatrices:

\[
(\%i16) \ M2:\text{matrix} ([[\%pi, 17], a*d - b*c],
\begin{bmatrix} 1 & a \\ b & 7 \end{bmatrix}, \%e);
\]

The submatrix \begin{bmatrix} 1 & a \\ b & 7 \end{bmatrix} on the second row, first column can be referenced with the command:

\[
(\%i18) \ M2[2][1];
\]

**Example 1.3.4. Programming constructs**

These are expressions such as \( x : y \) which means assign \( y \) to \( x \); the value of the assignment expression is the value of \( y \). There is probably no better example of the difference between an expression and its value than this object. This expression is always non-atomic, but its value is atomic if the value of \( y \) is atomic.

The block operation groups several expressions, and evaluates them one after another; the value of the block operation expression is the value of the last expression of the block. Try to guess the value of the expression below before entering it as input.

\[
(\%i19) \ ("a block", a:42) - ("another block", b:17);
\]
We observe that this expression is the difference between the values of two block expressions. The string atoms that are the first argument of each block do nothing. They are unassigned atoms and they pass out of existence after the value of the statement is displayed. Their purpose is merely to serve as arguments to demonstrate a block expression. This input does two things in one statement. First, it assigns 42 to the name a and 17 to the name b, and then it performs the difference operation using the value of these assignment statements as arguments. However, an assignment is not itself an operation. Entering \texttt{op(a:42)} results in an error message. This is because Maxima “sees” \texttt{a:42} as an atom. The input \texttt{atom(a:42)} and the value is true. \texttt{(a:42) - (b:17)} is merely a mathematical expression with two atoms as arguments. The value of this expression is 25. If you input \texttt{op((a:42) - (b:17))} the result is +. This peculiarity is due to the fact that Maxima is true to mathematics in the sense that subtraction is defined in terms of the sum of a and the additive inverse of b – \texttt{a - b} means by definition \texttt{a + (-b)}.

The result of \texttt{(%i20) [a, b];}

illustrates that although the string atoms of the block expressions were lost, and the block expressions themselves are gone, the names a and b have values that persist. This is a list expression object formed using the atoms a and b as arguments with values 42 and 17 respectively.

Consider \texttt{(%i21) block ([a], a: 42, a^2 - 1600) + block ([b], b: 5, %pi^b);}

This is a Maxima block programming construct object expression formed using two block operations ( \texttt{op = block} ) with the arguments as shown. Notice that the block operation returned the value of \texttt{a^2 - 1600 + block ([b], b: 5, %pi^b)} ,

where the last term in this argument was also a block operation. It returned a value of \texttt{%pi^5} – this was added to the 164 from the result of \texttt{42^2 - 1600 = 164} to yield the final output of \texttt{\pi^5 + 164}. Moreover \texttt{(%i22) (if a > 1 then %pi else %e) + (if b < 0 then 1/2 else 1/7);}

shows two programming object expressions as arguments to the addition operation. The characteristic operation of this expression is addition – \texttt{op = +}. The arguments are the two if operations that result in the two values \texttt{\pi} and \texttt{\frac{1}{2}} that are the addendi of the total expression. Note that the arguments to the if operations (\texttt{op is if}) may be relational mathematical expressions – \texttt{(a > 1)} and \texttt{(b < 0)}. These are evaluated as arguments in this case. What is interesting to note is that because the overall operator of this expression is \texttt{+}, this is a mathematical expression in the end. The values of the \texttt{op = if} expressions are atomic – \texttt{\pi} and \texttt{\frac{1}{2}}. The expressions themselves are non-atomic.
1.3.1 Utility Functions

These functions demonstrate how Maxima works and we have already seen examples of their use. The function \( \text{op}(\text{expr}) \) returns the overall (or characteristic) operator of a expression while \( \text{args}(\text{expr}) \) returns the arguments, and \( \text{atom}(\text{expr}) \) tells whether an expression is an atom.

Example 1.3.5. Let's try the following:

\[
\begin{align*}
\%(i23) & \quad \text{op}(p + q); \\
\text{The result of this is } & \text{“+”}. \text{ When an expression contains more than one operator, the outermost of these is obtained:} \\
\%(i24) & \quad \text{op}(p + q > p*q); \\
\text{Similarly when we have nested functions; the outer function is picked up:} \\
\%(i25) & \quad \text{op(sin}(p + q)); \\
\text{We can also pick up the operator of a function defined in the same expression, e.g.,} \\
\%(i26) & \quad \text{op(foo}(p, q) := p - q); \\
\text{gives the assignment operator } & \text{“:=”}. \\
\text{The corresponding utility for picking up function arguments is the function} \\
\text{args, try} \\
\%(i27) & \quad \text{args}(p + q); \\
\text{giving } & [q, p]. \text{ A composite expression like} \\
\%(i28) & \quad \text{args}(p + q > p*q); \\
\text{yields } & [q+p, p*q]. \text{ A curiosity is revealed when we try} \\
\%(i29) & \quad \text{args}(\text{foo}(p, q)); \\
\text{for the function defined above. It turns out that Maxima interprets subtraction} \\
\text{as addition of the corresponding number with opposite sign. In this case the} \\
\text{result is } & [p, -q]. \text{ As already noted, the function} \text{atom} \text{ returns “true” for an} \\
\text{expression being purely atomic and false otherwise, e.g.,} \\
\%(i30) & \quad \text{atom}(p); \\
\%(i31) & \quad \text{atom}(p + q); \\
\%(i32) & \quad \text{atom(sin}(p + q));
\end{align*}
\]
1.3. EXPRESSIONS

1.3.2 Expression Construction

The single quote tells Maxima to “construct” an expression but suppresses “evaluation” of the expression. It is important to note the difference between the assignment operator :., evaluation and the single quote. This is illustrated in the following

**Example 1.3.6.** Consider the results of the following inputs.

(\%i33) x:2+3;

(\%i34) x;

(\%i35) ’x;

(\%i36) op(x);

(\%i37) op(’x);

(\%i38) x;atom(x);

(\%i39) ’x;atom(’x);

The results of the above show that the value of an expression and its constructed value may be different even if both are atoms.

**Example 1.3.7.** Consider the assignment

(\%i40) x:if p > q then p else q;

In this case, x itself is the assigned expression, while the single quote results in just “x”. Compare the results of the following:

(\%i41) op(x);

(\%i42) op(’x);

So, there is a difference between value and constructed value also for non-atomic expressions.

Observe that ’x is atomic but ’(if p > q then p else q) is not. The former does not return an operator as already seen and the latter returns the operator “if”. The expression using the if programming construct is constructed as an argument to op, but it is not evaluated – it remains a non-atomic expression. Thus the characteristic operation of this expression is op = if. In a situation where p and q are atoms with certain values, the value of the expression (if p > q then p else q) is different from its constructed value:

(\%i43) p:3;q:%pi;(if p > q then p else q);’(if p > q then p else q);

Now, we consider a block programming construct
In the first case, \([a], a: 42, a^2 - 1600\) are returned, in the second case the declaration of the construct and in the last case the value \(42^2 - 1600 = 164\). The expression using the block programming construct is constructed as an argument to \texttt{args}, but it is not evaluated – it remains a non-atomic expression. Thus the arguments of this expression are the elements of the list returned as output. The constructed expression has a value of itself. The value of the expression is 164.

For our previous conditional programming construct, we try

\begin{verbatim}
(%i47) args('(if p > q then p else q));
\end{verbatim}

yielding the list \([p>q, p, true, q]\).

\section*{1.4 Evaluation}

In Maxima, the value of a symbol (name or identifier) is an expression associated with the symbol. Every symbol has a value and if not otherwise assigned a value, a symbol evaluates to itself – e.g., \(x\) evaluates to \(x\).

\subsection*{1.4.1 Evaluation of Atomic and Non-Atomic Expressions}

Numbers and strings are atoms and always evaluate to themselves. Names are atoms and evaluate to themselves or to an assigned value – an expression which may be atomic or non-atomic.

Non-atomic expressions are evaluated approximately as follows:

1. Each argument of the operator of the expression is evaluated.

2. If an operator is associated with a callable function, the function is called, and the return value of the function is the value of the expression.

\subsection*{1.4.2 Modifications of Evaluations}

Expressions can be modified in ways that cause less or more evaluation. Less evaluation is obtained by using certain functions that do not evaluate some of their arguments. A single quote character (\(\text{'\textbackslash'}\)) as a prefix prevents evaluation, but not construction of an expression – see the section above about expression construction.

More evaluation is caused by using two single quotes (\(\text{'\textbackslash'}\)), resulting in an extra evaluation at the time the expression is parsed as input. This means that modification occurs only at input time for expressions prefixed with \(\text{'\textbackslash'}\). When applied to a general expression (e.g., \(\text{'\textbackslash'}\text{expr}\)), the value of \texttt{expr} is substituted for \texttt{expr} in the input. The function \texttt{ev(a)} causes an extra evaluation of \(a\) every time
1.4. EVALUATION

\texttt{ev(a)} is evaluated. \texttt{apply(f, [a1, a2, \ldots, an])} causes the evaluation of the arguments \((a1, \ldots, an)\) even if \(f\) ordinarily “quotes” them. The function \texttt{define(f (a1, \ldots, an), expr)} constructs a function definition like \(f(a1, \ldots, an) := expr\), but evaluates the function body (\texttt{expr}) while := does not.

The following sections explain and provide some examples of evaluation.

1.4.3 Defaults and Argument Evaluation

Symbols evaluate to themselves if not otherwise assigned a value. If assigned a value names evaluate to those values. Consider a block expression assigning values to the names \(a\), \(b\) and \(e\). The return value is the value of the last expression in the block – \(e = 5\) in this case. These are atoms.

\[(\%i48) \text{block(a: 1, b: 2, e: 5);}\]

Arguments of operators are ordinarily evaluated – unless evaluation is prevented in some way. Look at the assignment

\[(\%i49) \text{block(x: %pi, y: %e);}\]

\[(\%i50) x;y;\]

We have a block expression assigning values to the names \(x\) and \(y\). As previously, the return value is the value of the last expression in the block – \(%e\) in this case. The arguments were evaluated. Hence the return value \(%pi\) for \(x\) and not \(x\). In the case of a function, such as

\[(\%i51) \sin(x + y);\]

we see that the function is evaluated according to the trigonometric identity

\(\sin(\pi + e) = \sin(\pi + e) = -\sin e.\)

An expression containing a relation, e.g.,

\[(\%i52) x > y;\]

evaluates using values assigned to the names \(x\) and \(y\). The return value is the expression itself with the the assigned values of the names \(x = %pi\) and \(y = %e\). To find out if \(\pi > e\) is true we can use

\[(\%i53) \text{is(x>y);}\]

For comparison, try

\[(\%i54) \text{is(x=y);}\]

To illustrate the difference between default evaluation and calculation of numerical value for a function, consider the factorial function

\[(\%i55) x!;\]

\[(\%i56) x!, \text{numer;}\]

In the latter case, a numerical evaluation based on the well-known relation between the gamma- and factorial functions takes place:

\[n! = \Gamma(n + 1) = \int_0^\infty e^{-t} t^n \, dt.\]
1.4.4 Callable Functions

All operators consisting of callable functions result in calls to those functions. If this is prevented, another expression with the same operator is obtained. Consider the function definition

\[
\text{(\%i57) foo(p, q) := p - q;}
\]

This is a callable function, i.e., if we assign the value

\[
\text{(\%i58) p: \%phi;}
\]
\[
\text{(\%i59) q: \%pi;}
\]

Here, \%pi is \pi and \%phi is a reserved name for the constant ratio \( \left( \frac{1 + \sqrt{5}}{2} \approx 1.618034 \right) \) of the golden mean. Now, try the calls

\[
\text{(\%i60) foo(p, q);}
\]
\[
\text{(\%i61) foo(p, q), numer;}
\]

We observe that the value of the call to \texttt{foo} is the value of the evaluated arguments (p and q) applied to the expression used to define the function \texttt{foo}. For comparison, we call

\[
\text{(\%i62) bar(p, q);}
\]

The function \texttt{bar} is not callable with arguments p and q – it has not been assigned any expression, merely created as a name. The value of \texttt{bar} is hence itself. In spite of this, the arguments (p and q) are evaluated however as displayed.

1.4.5 Automatically Quoted Arguments

Some functions “quote” their arguments automatically. Such built-in functions are \texttt{save()}, := and \texttt{kill()}, which means that these by default behave like a single quote being passed as a prefix to their argument. Hence, if \texttt{foo} is called as \texttt{foo('p,q)} the result is \( p - \%pi \) and not \( \%phi - \%pi \).

Try, e.g.,

\[
\text{(\%i63) \{foo('p,q),foo(p, 'q),foo('p, 'q)\};}
\]
\[
\text{(\%i64) block(a: 1, b: %pi, c: x + y);}
\]
\[
\text{(\%i65) [a, b, c];}
\]

We note that in the last two lines of input the arguments of the expressions were evaluated automatically. The function

\[
\text{save(filename, name1, name2, name3, ...)}
\]
\[
\text{(\%i66) save("tmp.save", a, b, c);}
\]
stores data in the form of Lisp expressions – the current values of name1, name2, ...
in filename. The arguments are the names of variables, functions, or other objects and the value returned for save is filename. This function evaluates filename and quotes all other arguments – here names a, b, and c. The file temp.save has the following as contents:

;;; -*- Mode: LISP; package:maxima; syntax:common-lisp; -*-
(in-package :maxima)
(DSKSETQ $A 1)
(ADD2LNC '$A $VALUES)
(DSKSETQ $B '$%PI)
(ADD2LNC '$B $VALUES)
(DSKSETQ $C '((MPLUS SIMP) $%E $%PI))
(ADD2LNC '$C $VALUES)
]

In this file there is a single quote prefixed to the arguments and the arguments are not evaluated. For a and b this is no huge issue since these are atoms to begin with so they are the same whether or not evaluated. However, c is shown in the file as having a value of $C '((MPLUS SIMP) $%E $%PI)) which is the constructed argument expression in the form of an op = + with arguments (%e, %pi). Effectively, the above call is equivalent to save("tmp.save", 'a, 'b, 'c) only here the quoting of arguments is automatic.

Consider the function and call

(%i67) f(a) := a^b;

(%i68) f(7);

The function f is defined using the := operation. The argument a and name b are not evaluated in the value for f(a). However, the call f(7) results in the evaluation of argument a and name b. Note that the value of a was set to 1 above (a:1) but it is evaluated in the function call as an argument to f and the value in this scope for a is 7. Moreover, the value of b was set to %pi above (b:%pi) but it is not evaluated. Remember, the assignment := does not evaluate the function body as an argument to a function definition statement – a and b are both part of the function body here.

All names defined by the user are displayed with the command

(%i69) values;

The kill function removes the values from the arguments, it does not evaluate the arguments – the return value of kill is “done”.

(%i70) kill(a, b, c, x, expr, L, L2);

The names in the argument are “killed” by this statement and their values thus lost. The command values displays a list of all bound user names. The list comprises symbols assigned expressions using :, ::, or :=. In wxmaxima, :: is the same as :, except that both sides of the assignment are evaluated, c.f. wxmaxima help.
values;

Using values again shows that the killed names are no longer listed and their values removed. The list expression \([a, b, c]\) references these names again, but these have no value assigned and are not saved as bound names.

1.4.6 Preventing Evaluation

A single quote prevents evaluation even if it would ordinarily happen. Consider the function definition and evaluation

\[
\text{(%i72) } \text{foo}(x, y) := y - x;
\]

\[
\text{(%i73) block}(a: \%e, b: 17);
\]

\[
\text{(%i74) foo}(a, b);
\]

The \textit{foo} function returns the value of the evaluated arguments when called. The call

\[
\text{(%i75) foo('a, 'b)};
\]

however, returns the function value with arguments unevaluated but instead with their names. Furthermore, calling

\[
\text{(%i76) 'foo}(a, b);
\]

results in \textit{foo}(\%e, 17). Here, the function itself is single quoted and hence not evaluated, although the arguments (a and b) are. The returned value is the function with the evaluated arguments. If we, instead of single quoting just the function name, quote the entire function call:

\[
\text{(%i77) '}(\text{foo}(a, b));
\]

we will get a result similar to what is obtained from the call \('foo('a, 'b);\).

1.4.7 Extra Evaluation

Two single quotes ('' or quote-quote) causes an extra evaluation at the time the expression is parsed. Differentiating the function \(\sin x\) with respect to \(x\) is done by

\[
\text{(%i78) diff}(\sin (x), x);
\]

The function call \textit{diff}(expr, x) returns the derivative of \textit{expr} with respect to the variable \(x\). The result is \(\cos x\), as expected. Let us define our own function that does this differentiation:

\[
\text{(%i79) foo}(x) := (\text{diff}(\sin (x), x));
\]

This function returns the derivative from the built-in function \textit{diff}. Recall that the := operation does not evaluate the function body, i.e., \textit{expr} in \textit{f(x)}:=\textit{expr} where \textit{expr} is some expression that defines the function, presumably involving the argument name \(x\). Evaluation in this case can be accomplished as:
Our function foo(x) is now redefined with \texttt{diff(sin(x),x)} prefixed by two single quotes which returns the evaluated derivative (or differential) of \texttt{expr} with respect to the variable \texttt{x}. The result of the evaluation of \texttt{diff(sin(x),x)} is \texttt{cos(x)}.

Using \texttt{ev(expr)} causes an extra evaluation every time it is itself evaluated. Contrast this with the effect of quote-quote('') above:

The function \texttt{block} here assigns \texttt{yy} to the name \texttt{xx} and then \texttt{zz} to the name \texttt{yy}.

\begin{verbatim}
(%i81) block(xx: yy, yy: zz);
(%i82) [xx, yy];
\end{verbatim}

From the second input line, we see that both are evaluated to their assigned values. Now, we define a function that uses the double quote to force evaluation:

\begin{verbatim}
(%i83) foo(x) := '"'x;
\end{verbatim}

The two single quotes prefixed to \texttt{x} cause an extra evaluation of \texttt{x} when the function \texttt{foo(x)} is called. Since \texttt{x} has no value then, the function definition \texttt{foo (x) := x} is returned:

\begin{verbatim}
(%i84) foo(x);
\end{verbatim}

Now, keeping in mind that \texttt{xx} was assigned the value \texttt{yy} which in turn was assigned the value \texttt{zz}, try the call

\begin{verbatim}
(%i85) foo(xx);
\end{verbatim}

This produced the value \texttt{yy} of the first assignment. Remember that arguments are evaluated when a function is called with arguments. We now define a function using the \texttt{ev} operator

\begin{verbatim}
(%i86) bar(x) := ev (x);
\end{verbatim}

This function definition for \texttt{bar(x)} uses the \texttt{ev(expr)} function instead of two single quotes. This means that \texttt{ev(expr)} is retained as part of the function definition for \texttt{bar(x)} as expected. When \texttt{bar(expr)} is called the argument \texttt{expr} will be evaluated as an argument and then by \texttt{ev(expr)}:

\begin{verbatim}
(%i87) bar(xx);
\end{verbatim}

Here, the expression \texttt{xx} is evaluated as \texttt{xx:yy} as the argument to \texttt{bar(expr)}. Then, because the function is defined using \texttt{ev(expr)} this is equivalent to \texttt{ev(yy)}, which returns the value of \texttt{yy} being \texttt{zz}. This is what we mean by saying that \texttt{ev(expr)} is evaluated each time it is called.
1.4.8 Arguments

The function \( \text{apply}(f, [a_1, a_2, \ldots, a]) \) causes the evaluation of arguments to the function \( f \) in the list even if they are ordinarily “quoted” by \( f \). In general, some functions deal with certain arguments as though they are prefixed by the single quote character \( ' \). Arguments are evaluated when a function is called with arguments, although some functions deviate from this by “single quoting” some arguments. For example, if the function \( f(x) \) is called with \text{exprx} as an argument \text{exprx} is evaluated.

Example 1.4.1. Consider, e.g.,

\[
\text{(%i88)} \text{block(a: aa, b: bb, c: cc)}; \\
\text{(%i89)} \text{block(aa: 11, bb: 22, cc: 33)}; \\
\text{(%i90)} [a, b, c, aa, bb, cc];
\]

The two block expressions above assign \text{aa} to \text{a}, \text{bb} to \text{b}, and \text{cc} to \text{c}, then \text{11} to \text{aa}, \text{22} to \text{bb} and \text{33} to \text{cc}. The list expression above displays these values.

Now, try evaluating arguments to the \text{kill}-function:

\[
\text{(%i91)} \text{apply(kill, [a, b, c])};
\]

The input \text{kill(a, b, c)} would not ordinarily result in the evaluation of arguments, the function would only be applied to \text{a}, \text{b} and \text{c}. However, with the above use of \text{apply()} the arguments to \text{kill} will be evaluated. In this case the result is the same as from calling \text{kill(aa, bb, cc)}, as seen from the list:

\[
\text{(%i92)} [a, b, c, aa, bb, cc];
\]

The result of using \text{apply} with the is that the names \text{a}, \text{b} and \text{c} retain their values – \text{aa}, \text{bb} and \text{cc}, respectively. However, the names \text{aa}, \text{bb} and \text{cc} now have the default values of themselves respectively as their assigned values were killed. On the other hand, when \text{kill} is called with the arguments \text{a}, \text{b} and \text{c} directly:

\[
\text{(%i93)} \text{kill(a, b, c)};
\]

their values are killed, as seen from:

\[
\text{(%i94)} [a, b, c, aa, bb, cc];
\]

Now, all the names assigned values above have lost these and assume default values.

To summarize, we may think of the \text{apply} function (\text{apply}(f, [x])) as “apply to the function \( f \) the evaluated argument \( x \)".
1.4.9 Arguments

Using `define(f, [a1, a2, ..., an], expr)` evaluates the expression (expr) that is body of the function definition. This is an alternative way to using the form of `f(a1, a2, ..., an):= expr` to define a function. In this case expr is not evaluated. The `define()` function defeats this behavior.

**Example 1.4.2.** Let us do the definite integral

\[ \int_0^\pi \sin(ax) \, dx = \frac{1 - \cos(a\pi)}{a} \]

This is accomplished by the call

(%i95) integrate(sin(a*x), x, 0, %pi);

The final form of the result can be obtained after simplification, using the call

(%i96) ratsimp(integrate (sin(a*x), x, 0, %pi));

Now, use this expression to define a function:

(%i97) foo(x) := integrate(sin(a*x), x, 0, %pi);

Hence, the function foo(x) is defined as the same symbolic definite integral, but we see that the expression defining foo is not evaluated. In contrast, using `define()` yields:

(%i98) define(foo(x), integrate(sin(a*x), x, 0, %pi));

In this case, the expression defining foo is evaluated. A shorter route to the simplification we already looked at is

(%i99) ratsimp(define(foo(x), integrate(sin(a*x), x, 0, %pi)));

Let us next assign a value to the parameter a:

(%i100) block(a:3, define(foo(x), integrate(sin(a*x), x, 0, %pi)));

So, as well as the integral the value of the final result is evaluated when `define()` is applied.

Consequently, we may think of the function `(define(f, [x], exprx) or define(f(x), exprx))` as “define a function named f with the argument x as the evaluated expression exprx”.

1.5 Simplification

After evaluating an expression, Maxima attempts to find an equivalent expression which is “simpler”. Maxima applies several rules for what is meant by “simplification”. For example, 1 + 1 simplifies to 2, \(x + x\) simplifies to \(2x\), and
\( \sin \pi \) simplifies to 0. However, many well-known identities are not applied automatically. Such are, e.g., double-angle formulas for trigonometric functions, or rearrangements of ratios such as \( \frac{\sin \theta}{\cos \theta} = \tan \theta \). These identities can be applied by use of Maxima-functions and user-defined simplification rules can be specified with the function \texttt{tellsimpafter}.

Remark 1.5.1. Simplification is a matter of convention to some degree. Most people agree on what “simpler” means in most expression contexts. However, on some points what is simpler is a matter of personal preference or convention. Why are fractions expressed in lowest terms simpler? One answer is because someone said so. Another answer could be because the numerals of the numerator and denominator are the smallest and so forth.

### 1.5.1 Examples of Maxima Simplification

The following are some examples of simplification:

(\%i101) \[
[1 + 1, x + x, x * x, \\
\sin(\pi), (a + b)^2, (a/b + c/b), \\
\sin(2*x), a * \exp(b * \%i)]
\]

The expressions in the above list of elements are simplified according to the Maxima rules of simplification.

A quote single quote character as a prefix prevents evaluation but not simplification. When the global flag \texttt{simp} is false, simplification is prevented but not evaluation.

(\%i102) \[
'[1 + 1, x + x, x * x, \sin(\pi), \\
(a + b)^2, (a/b + c/b), \sin(2*x), \\
a * \exp(b * \%i)]
\]

Remark 1.5.2. It is important to make clear the difference between evaluation and simplification. Recall that evaluation is a process that results in an expression in terms of the values of the ingredients (atoms and non-atoms) of an expression. Simplification is an equivalent form of an expression that is “simpler” by some criteria.

With \texttt{simp} set to “false”, we can try the following

(\%i103) \[
'[1 + 1, x + x, x * x, \sin(\pi), \\
(a + b)^2, (a/b + c/b), \sin(2*x), \\
a * \exp(b * \%i)]
\]

(\%i104) \texttt{block([x: 1], x + x)};

and we observe that the expressions in the list and block statement are not simplified. The automatic simplification can be switched on again, e.g., with the command

(\%i105) \texttt{kill(all)$simp:true$}

which clears values from names and sets the simplification flag to true.
1.6 Special Purpose Functions

1.5.2 Exceptions to Maxima Simplification

As mentioned, some identities are not applied automatically. In these cases user input is required in order to simplify certain expressions. Some functions which apply identities that may result in simpler expressions are `expand()`, `ratsimp()`, `trigexpand()` and `demoivre()`. Consider, e.g., the quadratic expression

(a + b)^2;

To expand this expression, we must invoke the function

expand((a + b)^2);

Neither is the sum

a/b + c/b;

simplified automatically. Simplification of rational expressions is done by invoking the function

ratsimp(a/b + c/b);

Expandable trigonometric functions, like the sine for the double angle

sin(2*x);

can be handled with the function

trigexpand(sin(2*x));

Dealing with complex exponential functions is customarily done with the use of Euler’s formula:

\[ e^{i\phi} = \cos \phi + i\sin \phi \ . \]

In Maxima, the exponential

a * exp(b * %i);

is expanded according to Euler’s formula by calling the function

demoivre(a * exp (b * %i));

1.6 Special Purpose Functions

1.6.1 Applying arguments to functions

As have seen above in the examples of this function, the function

apply(f, [x1, x2, ... , xn])

constructs and evaluates an expression in the form of \( f(\text{arg}_1, \text{arg}_2, \ldots , \text{arg}_n) \) where \( \text{arg}_1 = x_1, \text{arg}_2 = x_2, \ldots , \text{arg}_n = x_n \). The arguments of the expression are always evaluated. Next, we construct \( \sin(\pi x) \) by applying \( \pi x \) to the sine function, after clearing all values from names:
(\%i114) \texttt{kill(all)}$

(\%i115) \texttt{apply(sin, [x * \%pi])};

Now, create the list

(\%i116) \texttt{L1:[a, b, c, x, y, z]};

Addition of the elements in this list can now be done by applying addition to
the list, according to

(\%i117) \texttt{apply("+", L1)};

Similarly, we can find the smallest element of a list with

(\%i118) \texttt{L2:[1, 5, -10.2, 4, 3]};
(\%i119) \texttt{apply(min, L2)};

If we have a user-defined function

(\%i120) \texttt{F(x) := x / 1729};

we can assign \texttt{F} as a value to the name and display the function definition

(\%i121) \texttt{fname:F};

(\%i122) \texttt{dispfun(F)};

The value of \texttt{fname} is now \texttt{F}. However, the call \texttt{dispfun(fname)} would produce
an error message since \texttt{fname} is not the name of the user-defined function. The
correct way to use \texttt{dispfun} in this case is

(\%i123) \texttt{apply(dispfun, [fname])};

1.6.2 The mapping function

Using the \texttt{map()} function constructs and evaluates an expression for each item in
a list of arguments. The arguments of the expression are always evaluated and
the result is a list object, consider a yet undefined function \texttt{foo} and addition:

(\%i123) \texttt{kill(all)}$

(\%i124) \texttt{map(foo, [x, y, z])};

(\%i125) \texttt{map("+", [1, 2, 3], [a, b, c])};

For the latter, note that the length of the lists must be the same or an error will
result since "+" is a binary operation. Mapping the \texttt{atom()-function} (checks
whether the argument is an atom or not) like

(\%i126) \texttt{map(atom, [a, b, c, a + b, a + b + c])};

yields "true" for the first three arguments an "false" for the other two, since those
are expressions, not atoms.
1.6.3 The lambda-function

Using `lambda()` defines and returns a “lambda” expression, i.e., an “unnamed” or “anonymous” function. The lambda expression can be used in some contexts like an ordinarily named function. However, lambda does not evaluate the function body. Try the function call

```lisp
(%i127) f: lambda ([x, y], (x + y)*(x - y));
```

Here, the name `f` is assigned the lambda expression. No function is defined and no evaluation takes place. However, the call `f(a, b)` returns the value of the lambda expression as though it were a function even though there is no function named `f`. Instead `f` has the value of the lambda expression, as seen from:

```lisp
(%i128) f;
```

We can apply the expression named by `f` to the elements of the argument list

```lisp
(%i129) apply(f, [p, q]);
```

We can apply the lambda expression named by `f` to pairwise elements, e.g., by mapping it to two equally long lists, according to:

```lisp
(%i130) map(f, [1, 2, 3], [a, b, c]);
```

We get a list of three expressions, one for each pair – `a` and `1`, `b` and `2` and `c` and `3`.

As we have seen, a lambda function is a means to have a name behave as a function without being defined as a function.

1.7 Built-in object types

Objects are represented as expressions, which are made up of an operator and its arguments. The most important built-in object types are lists, matrices, and sets.

1.7.1 Lists

1. A literal list is indicated like this: `[a, b, c].`

2. If `L` is a list, `L[i]` is element `i` of the list.

3. `map(f, L)` applies `f` to each element of `L`.

4. `apply("+", L)` is the sum of the elements of `L`.

5. For `x in L do expr` evaluates `expr` for each element of `L`.

6. `length(L)` is the number of elements in `L`. 
Example 1.7.1.  \((%i131)\) \(L:\{a, \text{foo, foo\_bar, \"Hello, world!\", 42, 17.29}\};\)
\((%i132)\) \(L[1]; L[3];\)
\((%i133)\) \(\text{map(\text{display},L)};\)
\((%i134)\) \(\text{apply("+",L)};\)
\((%i135)\) \(\text{apply("+",[1,2,3,4])};\)
\((%i136)\) \(\text{for e in L do display(e)};\)
\((%i137)\) \(\text{length(L)};\)

1.7.2 Matrices

1. A matrix is defined as:

\[
\text{matrix}(L1, \ldots, Ln)
\]

where \(L_1, \ldots, L_n\) are lists which represent the rows of the matrix.

2. If \(M\) is a matrix, \(M[i, j]\) or \(M[i][j]\) is element \((i,j)\).

3. The operator \(\cdot\) represents non-commutative multiplication. \(M \cdot L, L \cdot M\) and \(M \cdot N\) are non-commutative products, where \(L\) is a list and \(M\) and \(N\) are matrices.

4. \text{transpose}(M) is the transpose of \(M\).

5. \text{eigenvalues}(M) returns the eigenvalues of \(M\).

6. \text{eigenvectors}(M) returns the eigenvectors of \(M\).

7. \text{length}(M) returns the number of rows of \(M\).

8. \text{length(transpose(M))} returns the number of columns of \(M\).

Example 1.7.2. Consider a matrix and various operations of referencing its elements:

\((%i138)\) \(M:\text{matrix}([a, b, [c, \%pi, \{\%e, 1729, 1/(a*d - b*c)\}]];\)
\((%i139)\) \(M[1,1]; M[1,3];\)
\((%i140)\) \(M[1,3][3];\)
1.7. BUILT-IN OBJECT TYPES

The third element of the matrix \( M \) is a list. The above expression has a value of the third element of that list – a set object. Let us assign a matrix, by specifying row vectors, and look at two ways of referencing elements and how to take the transpose:

\[
\begin{align*}
(\%i141) & \quad M: matrix([1,2],[3,4]); \\
(\%i142) & \quad M[1,1]; M[2][2]; \\
(\%i143) & \quad transpose(M);
\end{align*}
\]

Assign a larger, 3 \( \times \) 3-matrix, and look at how to extract the number of rows of a matrix. In this case we determine the number of columns by taking the transpose:

\[
\begin{align*}
(\%i144) & \quad M: matrix([1,2,3,4],[5,6,7,8]); \\
(\%i145) & \quad length(M); length(transpose(M));
\end{align*}
\]

Consider a 2 \( \times \) 2-matrix and compute its eigenvalues and eigenvectors:

\[
\begin{align*}
(\%i146) & \quad M: matrix([4,1],[3,2]); \\
(\%i147) & \quad eigenvalues(M); \\
(\%i148) & \quad X: eigenvectors(M);
\end{align*}
\]

We see that the eigenvalues of the matrix \( M = \begin{bmatrix} 4 & 1 \\ 3 & 2 \end{bmatrix} \) are \( \lambda_1 = 1 \) and \( \lambda_2 = 5 \). The eigenvectors are \( x_1 = \begin{bmatrix} 1 \\ -3 \end{bmatrix} \) and \( x_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \). We also note that the function \( \text{eigenvectors()} \) returns the eigenvalues as well in an output list with the eigenvectors.

1.7.3 Sets

1. Maxima understands explicitly-defined finite sets. Sets are not the same types of objects as are lists. Thus an explicit conversion is needed to change one into the other.

2. A set is specified like this: \{a,b,c, ... \} where the set elements are (a,b,c, ... ). An alternative way to specify a set is to use the set function as in \( \text{set(a,b,c, ... )} \).

3. \( \text{union(A, B)} \) is the union of the sets \( A \) and \( B \).

4. \( \text{intersection(A, B)} \) is the intersection of the sets \( A \) and \( B \).

5. \( \text{cardinality(A)} \) is the number of elements in the set \( A \).
Example 1.7.3. Consider the set

\(\texttt{(\%i149)} \{9,8,7,6,5,4,3,2,1\}; S:\{f,e,d,c,b,a\}; f;\)

Oops! The name \(f\) was assigned a lambda expression previously. Thus the name \(f\) evaluates to this value when \(S\) is assigned the set object expression. Let us redefine some sets and make some set calculations

\(\texttt{(\%i150)} \text{kill(all)}\)$
\(\texttt{S:set(f,e,d,c,b,a)}\); $f;

\(\texttt{(\%i151)} S;\)

\(\texttt{(\%i152)} (A:\{1,3,5,7,9\}, B:\{1,2,4,6,8,9\}, \text{union}(A,B));\)

\(\texttt{(\%i153)} \text{intersection}(A,B);\)

\(\texttt{(\%i154)} \text{cardinality(union}(A,B)); \text{cardinality}(S);\)

1.8 Maxima Programming

In Maxima, there is one and only one name space, which contains all Maxima symbols. Parallel name spaces are not possible. All variables are global unless are explicitly declared as local variables. Functions, lambda expressions, and blocks can have local variables. The value of a variable is whatever was assigned most recently, either by explicit assignment or by assignment of a value to a local variable in a block, function, or lambda expression. This policy is known as dynamic scope. If a variable is a local variable in a function, lambda expression, or block, its value is local but its other properties (as established by declare) are global. The function \texttt{local()} makes a variable local with respect to all properties. By default a function definition is global, even if it appears inside a function, lambda expression, or block. \texttt{local(\textit{f}), f(\textit{x}) := \ldots} creates a local function definition. \texttt{trace(\textit{foo})} causes Maxima to print an message when the function \textit{foo} is entered and exited.

We now consider some examples of Maxima programming.

Example 1.8.1. All variables are global unless they appear in a declaration of local variables. Functions, lambda expressions, and blocks can have local variables.

\(\texttt{(\%i155)} (x: 42, y: 1729, z: \text{foo}\text{*bar})\)

\(\texttt{(\%i156)} f(x, y) := x*y*z;\)

The name \(z\) does not appear in as a argument of the function \(f\) and it was assigned the mathematical expression \texttt{foo*bar}. Then, with respect to \(f\) the name \(z\) is global.
When $f$ is evaluated with $x$ $aa=$ and $y$ $bb=$, these names are local to $f$ and the name $z$ is not. Note that Maxima rearranges the order of the names that are the factors of the value of $f(aa, bb)$ alphabetically as shown above. The factors that are the value of the global name $z$ (foo and bar) appear in that order together with the argument values for $f$, i.e., $x$ and $y$ which are $aa$ and $bb$, respectively. The global names $x$ and $y$ with values 42 and 1729 are not relevant to $f$.

Let us try an anonymous lambda expression, to which two names are applied:

$$\text{(\%i159) lambda([x, z], (x - z)/y);}$$

$$\text{(\%i160) apply(\%, [uu, vv]);}$$

The anonymous lambda is assigned (as a feature of Maxima) as a value to the name '%. This name is used to reference the lambda expression in the next expression as an argument to the apply() function. The name $y$ was not referenced as part of an argument to the lambda expression. Therefore it is global with respect to the value of the name '%. The names $x$ and $z$ were referenced as part of the arguments to the lambda expression and so these names are local with respect to the value of the name '%. The global value of $y$ is evaluated in the above expression as 1729. The global values of $x$ and $z$ (42 and foo*bar) are irrelevant to the expression.

Try the block operation

$$\text{(\%i161) block([y, z], y: 65536, [x, y, z]);}$$

Because the name $z$ is referenced in the first argument of the block operation that is a list, it becomes local to the block. The last argument which is returned is the value of the list – $[x, y, z]$. The name $x$ is global and has a value of 42. The name $y$ is local to the block having been assigned a value of 65536. The name $z$ is local to the block for the reason above but was assigned no value in the block. Hence the value of $z$ is itself and that is what is returned.

Example 1.8.2. The value of a variable is whatever was assigned most recently, either by explicit assignment or by assignment of a value to a local variable.

$$\text{(\%i162) foo(y) := x - y;}$$

$$\text{(\%i163) x:1729;}$$

When $\text{foo(y)}$ was assigned the expression $x-y$ above the value of $x$ was itself. Here $x$ is now explicitly assigned the atom 1729. If we call $\text{foo}$ with the argument $y = \%pi$

$$\text{(\%i164) foo(\%pi);}$$
the result is our function expression with the explicitly assigned atom 1729 as
the value of x. Recall that callable functions are evaluated when referenced.

Define the function dependent on x through foo as:

(%i165) bar(x) := foo (%e);

(%i166) bar(42);

Now the function bar(x) is defined using the value of the expression foo(%e),
which is x-%e. Recall that the expression of a function defined with := is not
evaluated by default. This explains why bar is x-%e and not 42-%e. The name
x is assigned the value 42 as a name local to bar when it is called as bar(42).
This is the most recent assignment of a value to x. Therefore when f(%e) is
evaluated in the context of bar the value of x is 42 and not 1729.

1.8.1 Branching

The if statement is used for conditional execution. The syntax is:

if <condition> then <expr_1> else <expr_2>

The result of an if statement is expr_1 if condition is true and expr_2 otherwise. expr_1 and expr_2 can be any Maxima expressions (including nested if statements), and condition is an expression which evaluates to true or false
and is composed of relational and logical operators which are as follows:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>less than</td>
<td>&lt;</td>
<td>relational infix</td>
</tr>
<tr>
<td>less than or equal to</td>
<td>&lt;=</td>
<td>relational infix</td>
</tr>
<tr>
<td>equality (syntactic)</td>
<td>=</td>
<td>relational infix</td>
</tr>
<tr>
<td>negation of equality</td>
<td>#</td>
<td>relational infix</td>
</tr>
<tr>
<td>equality (value)</td>
<td>equal</td>
<td>relational infix</td>
</tr>
<tr>
<td>negation of equality</td>
<td>notequal</td>
<td>relational infix</td>
</tr>
<tr>
<td>greater than</td>
<td>&gt;=</td>
<td>relational infix</td>
</tr>
<tr>
<td>and</td>
<td>and</td>
<td>logical infix</td>
</tr>
<tr>
<td>or</td>
<td>or</td>
<td>logical infix</td>
</tr>
<tr>
<td>not</td>
<td>not</td>
<td>logical prefix</td>
</tr>
</tbody>
</table>

1.8.2 Iteration

The do statement is used for performing iteration. Due to its great generality
the do statement will be described in two parts. First, the usual form will be
given which is analogous to that used in several other programming languages (Fortran, Algol, PL/I, etc.); then we move on to the other features.

There are three variants of this form that differ only in their terminating conditions. They are:

\[
\begin{align*}
&\text{for variable: initial\_value step increment thru limit do body} \\
&\text{for variable: initial\_value step increment while condition do body} \\
&\text{for variable: initial\_value step increment unless condition do body}
\end{align*}
\]

(Alternatively, the step may be given after the termination condition or limit.)

initial\_value, increment, limit, and body can be any expressions. If the increment is 1 then "step 1" may be omitted.

The execution of the do statement proceeds by first assigning the initial\_value to the variable (henceforth called the control-variable). Then:

1. If the control-variable has exceeded the limit of a thru specification, or if the condition of the unless is true, or if the condition of the while is false then the do terminates.

2. The body is evaluated.

3. The increment is added to the control-variable.

The process from (1) to (3) is performed repeatedly until the termination condition is satisfied. One may also give several termination conditions in which case the do terminates when any of them is satisfied.

In general the thru test is satisfied when the control-variable is greater than the limit if the increment was non-negative, or when the control-variable is less than the limit if the increment was negative. The increment and limit may be non-numeric expressions as long as this inequality can be determined.

However, unless the increment is syntactically negative (e.g. is a negative number) at the time the do statement is input, Maxima assumes it will be positive when the do is executed. If it is not positive, then the do may not terminate properly.

Note that the limit, increment, and termination condition are evaluated each time through the loop. Thus if any of these involve much computation, and yield a result that does not change during all the executions of the body, then it is more efficient to set a variable to their value prior to the do and use this variable in the do form.

The value normally returned by a do statement is the atom done. However, the function return may be used inside the body to exit the do prematurely and give it any desired value. Note however that a return within a do that occurs in a block will exit only the do and not the block. Note also that the go function may not be used to exit from a do into a surrounding block.

The control-variable is always local to the do and thus any variable may be used without affecting the value of a variable with the same name outside of the do. The control-variable is unbound after the do terminates.

Example 1.8.3. Various loops.
for a:-3 thru 26 step 7 do display(a)$
    a = -3
    a = 4
    a = 11
    a = 18
    a = 25

s: 0$
for i: 1 while i <= 10 do s: s+i;
done
s; 55

Note that the condition while i < -10 is equivalent to unless i > 10 and also thru 10.

series: 1$
term: exp (sin (x))$
for p: 1 unless p > 7 do
    (term: diff (term, x)/p,
     series: series + subst (x=0, term)*x^p)$
series;
    \[
    \frac{7}{90} \cdot x^1 - \frac{6}{240} \cdot x^2 - \frac{5}{15} \cdot x^3 + \frac{4}{8} \cdot x^4 + \frac{2}{2} \cdot x^5
    \]
(%o4) \begin{array}{cccccc}
90 & 240 & 15 & 8 & 2 & \end{array}

which gives 8 terms of the Taylor series for e^{\sin(x)}.

poly: 0$
for i: 1 thru 5 do
    for j: i step -1 thru 1 do
        poly: poly + i*x^j$
poly;
    5 \cdot x^5 + 4 \cdot x^4 + 3 \cdot x^3 + 2 \cdot x^2
(%o3) \begin{array}{cccc}
5 & 4 & 3 & 2 & \end{array}
guess: -3.0$
guess:
for i: 1 thru 10 do
    (guess: subst (guess, x, 0.5*(x + 10/x)),
     if abs (guess^2 - 10) < 0.00005 then return (guess));
    3.162280701754386

This iteration computes the negative square root of 10 using Newton-Raphson iteration a maximum of 10 times. Had the convergence criterion not been met the value returned would have been done.

Instead of always adding a quantity to the control-variable one may sometimes wish to change it in some other way for each iteration. In this case one may use next expression instead of step increment. This will cause the
control-variable to be set to the result of evaluating expression each time through the loop.

(%i6) for count: 2 next 3*count thru 20 do display (count)$
   count = 2
   count = 6
   count = 18

As an alternative to

for variable: value ...do...

the syntax

for variable from value ...do...

may be used. This permits the from value to be placed after the step or next value or after the termination condition. If from value is omitted, then 1 is used as the initial value.

Sometimes one may be interested in performing an iteration where the control-variable is never actually used. It is thus permissible to give only the termination conditions omitting the initialization and updating information as in the following setting to compute the square-root of 5 using a poor initial guess.

(%i1) x: 1000$
(%i2) thru 20 do x: 0.5*(x + 5.0/x)$
(%i3) x;
(%o3) 2.23606797749979
(%i4) sqrt(5), numer;
(%o4) 2.23606797749979

If it is desired one may even omit the termination conditions entirely and just give do body which will continue to evaluate the body indefinitely. In this case the function return should be used to terminate execution of the do.

(%i1) newton (f, x):= ([y, df, dfx], df: diff (f ('x), 'x),
   do (y: ev(df), x: x - f(x)/y,
   if abs (f (x)) < 5e-6 then return (x))$$(%i2) sqr (x) := x^2 - 5.0$
(%i3) newton (sqr, 1000);  
(%o3) 2.236068027062195

(Note that return, when executed, causes the current value of x to be returned as the value of the do. The block is exited and this value of the do is returned as the value of the block because the do is the last statement in the block.)

One other form of the do is available in Maxima. The syntax is:

for variable in list end_tests do body
The elements of list are any expressions which will successively be assigned to the variable on each iteration of the body. The optional termination tests end_tests can be used to terminate execution of the do; otherwise it will terminate when the list is exhausted or when a return is executed in the body. (In fact, list may be any non-atomic expression, and successive parts are taken.)

Example 1.8.4. Alternative form of do-loop

(%i1) for f in [log, rho, atan] do ldisp(f(1))$
(%t1) 0
(%t2) rho(1)
(%t3) %pi
(%t4) ev(%t3,numer);
(%o4) 0.78539816

1.9 Some Useful Flags

We have already seen use of the flag simp. There are several other system variables that work in a similar way.

Example 1.9.1. For example, when exptdispflag is true, Maxima displays expressions with negative exponents using quotients, e.g., \( x^{-1} \) as \( \frac{1}{x} \). This is done when simp is turned on, as seen from the calls below

(%i167) kill(all)$
exptdispflag:true;

(%i168) simp:false$
x^-1;
a*x^-2*y*z^-1;
simp:true$
x^-1;
a*x^-2*y*z^-1;
exptdispflag:false$

(%i169) 1/x;

Example 1.9.2. The function display (arg_1, arg_2, ...) displays identities for which the left side is each one of the arguments unevaluated, and whose right side is the value of the expression centered on the line. This function is useful in blocks and for statements in order to have intermediate results displayed, consider:

(%i170) display(sqrt((x*y^2*((x*y^3))^(1/4))^(1/3)))
Note that an equality is displayed with the left hand side (lhs) being the unevaluated expression of the argument the right hand side (rhs) being the evaluated expression with applicable simplification. The displayed output is an expression itself (an equality relation expression) and that this expression is lost after it is displayed since the value of the display function is “done” and not this equation expression. To see this we use the `rhs()` function on the output variable which should return the right-hand side of a relation expression:

```maxima
(%i171) rhs(%%);
```

The result is 0. This makes sense since the rhs of the expression `done` is 0. A way to get around this is to use the `ldisplay()` function:

```
ldisplay (arg1, arg2, ... , argn)
```

This function displays the argument expressions to the console as printed output. Each expression is printed as an equation of the form lhs = rhs in which lhs is one of the arguments of `ldisplay` and rhs is its value. The `ldisplay()` function assigns an intermediate expression label to each equation and returns the list of labels. The labels hold the value of the equation expressions.

```maxima
(%i172) ldisplay(sqrt((x*y^2*((x*y^3))^(1/4))^(1/3)))

(%i173) %t172;

(%i174) rhs(%t172);
```

The expression resulting from the `ldisplay()` function is not “done”, rather it is a labeled name – `%t172` in this case. This name holds the value of the equation expression that was the result of the `ldisplay()` function and works like any other name – it has an expression as a value. The `rhs()` function results in an expression that is the right hand side of the equation relation.

There are many useful Maxima flags (option variables) that may be used to alter the behavior of Maxima and its functions in order to get desired results. Refer to the Maxima Manual for a complete description of these flags and functions. This how-to has demonstrated the use of a few of these. The last one now follows:

**Example 1.9.3. %edispflag**

When `%edispflag` is true, Maxima displays the Napier number `%e` to a negative exponent as a quotient.

```maxima
(%i175) %e^-2;
%edispflag:true$

%e^-2;
```
1.10 Summary

The above sections cover most of the material of the Minimal Maxima document. We have postponed many more specific “how-to” examples to separate treatment later on.

The above examples and explanation do not exhaust all the features applicable to Maxima basics. Enough has been demonstrated to be of use to the most common purposes. The reader is directed to the Maxima documentation related to set objects and related topics for more information and examples.
Chapter 2

Symbolic Calculation

We will continue here with wxMaxima and take a closer look at some useful applications for various mathematical tasks an engineer would encounter in research and routine computation. Focus will be on use of Maxima with the wx graphical user interface and we are not attempting to include all the fine points that would be available with a proficient command-line use of the program. Our goal is getting started with wxMaxima through the GUI. Since most corresponding command-line functions are shown simultaneously when the menus are used, a “heavy-user” of the package is bound to learn these gradually along the line.

2.1 Initialization

wxMaxima can be configured through the “edit” menu of the main GUI by choosing “configure” and the desired options from the submenus. For example, ticking the option for evaluating input cells at pressing enter is a handy adjustment to the defaults. This menu also makes it possible to customize the appearance of wxMaxima, such as fonts, input/output formats and numbering/indexing.

To get help with a name used by Maxima, just type \texttt{? name}. For example, the command

\texttt{? integrate}

launches a Maxima help dialog for the function \texttt{integrate}. Sometimes the help finder does not match your entry exactly, then there will be a list of matches at the left to choose from.

Maxima uses some nonintuitive syntax for assigning a value to a label. Here are the rules:

Rule 2.1.1. Assigning values.

1. “:” used to assign a value to a variable: \texttt{a : 3}; \texttt{a : b}; etc.
2. “=” used to compare values for equality: \(a = 3\); etc., moreover “<” (less than), “>” (greater than) and “#” (not equal).

3. “:=” used to define a function: \(f(x) := x^2\);

4. “\texttt{kill(name);}” used to erase a previous assignment of a variable or function: \(\texttt{kill(a); \texttt{kill(f);}}\)

5. “\texttt{kill(all);}” used to erase all assignments.

The order of precedence in computations is the customary: (1) powers (\(^\) and roots (\(\sqrt{\) ), (2) multiplication (\(\ast\) ) and division (\(\div\) ) and (3) addition (\(\plus\) ) and subtraction (\(\minus\) ). To override this order, it is necessary to use parentheses.

**Example 2.1.2. Reserved numbers and display settings.**

\%pi; (this will appear as a Greek letter only if a Greek font has been installed, see "Installation")
\texttt{float(\%pi);} (meaning "give me the numerical value")
3.141592653589793
\texttt{fpprec:1000;} (increase the "floating point precision" or "fpprec")
1000
\texttt{float(\%pi);} (again, "give me the numerical value")
3.141592653589793 (no change in the number of displayed places. Why?)
2.2 SAVING AND LOADING

bfloat(%pi); ("bfloat" means "big float")
3.1415926535897932384626433832[943 digits]
30019276611195909216420196b0 (better, but not what I was hoping for)
set_display(ascii); (this is a special wxMaxima setting that forces all
the digits to be printed out)
set_display(ascii);
bfloat(%pi); (try again)
(a list of 1000 digits of pi)
set_display(xml); (return to the default way of displaying things)
set_display(xml);

The “fpprec” and “set_display(ascii)” settings only affect floating-point
numbers, integers are always printed out fully. The character “%” always refers
to the nearest prior result. To list all the digits of a floating-point value, we must
use “set_display(ascii)” and if we want wxMaxima to render equations and
functions in a pretty way, we can use “set_display(xml)”. This will display
greek letters etc., however, it requires installation of greek fonts.

2.2 Saving and Loading

Assume we are in a work session and have produced some variable definitions
and functions worth keeping. Maxima saves this work using the command:

stringout("my_work.txt",values,functions);

This will save all defined variables and functions in a plain-text file, suitable for
editing and re-use in a later Maxima session. In the above example, the file will
be saved under our home directory, which if we are running Linux, typically
is “/home/(username)/”. Unfortunately, because of a programming error, un-
der Windows the file will be saved under the wxMaxima installation directory,
which is typically “C:/Program Files/Maxima-(version)/wxMaxima/”. One
can only hope this error will be corrected in a future version of wxMaxima
so the default directory for file operations will be the user’s home directory,
“C:/Documents and Settings/(username)/”.

Assuming we have made entries into Maxima in a particular order, and
those entries and their order is important, and we would like to preserve what
we typed:

stringout("my_input.txt",input);

This saves an unedited list of everything you have typed since beginning your
Maxima session. To be useful, it might be edited, or parts of it extracted into
another file. But this form creates a record of all your inputs for later reference.
In case we do not recall all details and simply would like to save everything, we
can enter:

stringout("everything.txt",all);
Other options for this command are “values” and “functions”.

The two most useful loading commands are “\texttt{batch()}” and “\texttt{batchload()}”. These funny-sounding names go back to the early days of Fortran, when a “batch” was a stack of 80-column program and data cards, and computers were gigantic boxes kept in airtight rooms manned by people in white lab coats. This naming style tells us how long Maxima, and its predecessor Macsyma, have been around. There are any number of ways to create files suitable for Maxima loading. One obvious way is to save a file as shown above, using \texttt{\texttt{stringout('filename',values,functions)};"}. Such a file is quite suitable for reloading into Maxima, and it will restore all your definitions. Another way is to write Maxima code into a file, then load it. In this approach, one might extract useful sections from saved data files, and create new entries to accomplish a particular goal. An easy way to go about this is to have a text editor and wxMaxima running at once. One would type something into the text editor, save the file in its current form, then turn to wxMaxima and issue one of these two commands:

\begin{verbatim}
batch("everything.txt");
\end{verbatim}

(to see each instruction as it is executed)

\begin{verbatim}
batchload("everything.txt");
\end{verbatim}

(to load and execute instructions silently)

The second option above is quieter, but it will still print out results and create graphs. Once a program is working, bug-free and has become a useful tool, the second loading option is preferred.

\textbf{Example 2.2.1.} Now for an example of loading and running a Maxima program file. Here is a listing of a very simple routine that prints a multiplication matrix:

\begin{verbatim}
for y:1 thru 12 do
  (s:"",
   for x:1 thru 12 do
   s:sconcat(s,printf(false,"~4d",x*y)),
   print(s));
\end{verbatim}

To get this into Maxima, use your browser to save the file on your computer, and provide the path and filename to Maxima using the “\texttt{batchload()}” command: \texttt{\texttt{batchload(\"/path/to/example/file.txt\")\;"}}. If you have done this correctly, the following display will appear:

\begin{verbatim}
1 2 3 4 5 6 7 8 9 10 11 12
2 4 6 8 10 12 14 16 18 20 22 24
3 6 9 12 15 18 21 24 27 30 33 36
4 8 12 16 20 24 28 32 36 40 44 48
5 10 15 20 25 30 35 40 45 50 55 60
6 12 18 24 30 36 42 48 54 60 66 72
7 14 21 28 35 42 49 56 63 70 77 84
8 16 24 32 40 48 56 64 72 80 88 96
\end{verbatim}
2.3. FUNCTIONS AND EQUATIONS

Windows note: Because of a programming error in wxMaxima at the time of writing, you will have to use a full path to tell wxMaxima where your browser put the file. Most likely it will be located here:

"C:/Documents and Settings/(username)/My Documents/maxima_mult_matrix.txt"

There are some file saving and loading features in wxMaxima that create and read files with the suffix ".wxm", these represent an easy way to save and load an entire session.

2.3 Functions and Equations

2.3.1 Functions

In the simplest cases, one can create a useful function with a few keystrokes:

\[
\text{average}(a) := \frac{\text{sum}(a[i], i, 1, \text{length}(a))}{\text{length}(a)}
\]

As explained above, this definition will be saved along with all other user-defined content with the command “\text{stringout}'filename',\text{values},\text{functions}')”. The reason for creating functions is to encapsulate and name useful operations and algorithms, just as the reason for creating variables is to encapsulate and name useful values. Look at another example, there is no \(\text{log10}(x)\) function in Maxima (or most other software), because there is no pressing need for this to be explicitly defined. But there almost always is a \(\text{log}(x)\) function that produces a natural logarithm for its input argument. Thus, we can define the function:

\[
\text{log10}(x) := \frac{\text{log}(x)}{\text{log}(10.0)};
\]

A test:

\[
\text{log10}(1e6);
\]

5.999999999999999

Not perfect (why?). It has to do with conversion between binary and decimal number bases.

Not all function writing is as straightforward as the above example. It sometimes happens that one would like to capture the result of a prior calculation and make it into a new function. This is a bit more complex. Take at look at:

\[
a * x^2 + b * x + c = 0;
\]

\[
a x^2 + b x + c = 0
\]

\[
\text{solve}(% , x);
\]

9 18 27 36 45 63 72 81 90 99 108
10 20 30 40 50 60 70 80 90 100 110 120
11 22 33 44 55 66 77 88 99 110 121 132
12 24 36 48 60 72 84 96 108 120 132 144
At this point we need to introduce a new practice. To avoid a lot of confusion as these examples get more complex, we save intermediate results under distinctive variable names:

\[
eq:a \times 2+b \times x+c=0;
\]
\[
a x^2 + b x + c = 0
\]
\[
sol:solve(eq,x);
\]

This variable assignment will turn out to be very important below. Now we want to capture the quadratic solution created above and put it into a function that will accept any arguments for a, b and c, and produce numerical results for numerical arguments. Omitting the details of how Maxima organizes and stores its results, it should suffice to say that we have saved the two roots of the quadratic equation in a list named “sol”, and the list can be accessed by way of an index. Like this:

\[
sol[1];
\]
\[
sol[2];
\]

Now, we can access the two roots independently. But for use in a function definition, we need to get rid of the “x =” part of the expression. Try:

\[
rhs(sol[1]);
\]
\[
rhs(sol[2]);
\]

Build a function out of these pieces:

\[
quad(a,b,c) := [ \text{rhs(sol[1])}, \text{rhs(sol[2])} ];
\]

Okay, we seem to have created the function. Let us test it:

\[
quad(1,2,3);
\]

This is wrong. We should have seen each appearance of a, b and c replaced with the numerical values provided in the function call. The reason for this failure is subtle, but included here because it is a common problem. Here is the solution:

\[
quad(a,b,c) := \text{ev([ rhs(sol[1])}, \text{rhs(sol[2]) ]};
\]

This change works because the right-hand expression is evaluated (“ev()”) before being assigned to the function definition, and the evaluation replaces references to list contents with the contents themselves. Test this new form:

\[
quad(1,2,3);
\]

Correct! (note the presence of “i”, which signifies that the result is complex). Now we can enter any desired arguments to our new “quad()” function and get the two roots of the quadratic equation for those values. To reduce the output to simple numeric quantities, we need only express it this way:

\[
float(quad(1,5,3));
\]
2.3. FUNCTIONS AND EQUATIONS

Just to demonstrate how powerful this approach can be, make some small editorial changes in the above example:

\[
eq: a \cdot x^3 + b \cdot x^2 + c \cdot x + d = 0;
\]

\[
sol: \text{solve}(eq, x);
\]

\[
cubic(a, b, c, d) := \text{ev}([\text{rhs(sol[1])}, \text{rhs(sol[2])}, \text{rhs(sol[3])}]);
\]

Remember about these examples that declaring a function in the default way:

\[
f(x) := x^2;
\]

is only guaranteed to work in the simplest cases. Remember that the function “ev()” (meaning evaluate) can be used to solve some of the more difficult assignment problems.

2.3.2 Equations - A Practical Application

We proceed to show how a single equation can be used to construct a set of related equations. On this basis a set of functions can be created to solve an entire class of problems.

Example 2.3.1. Future Value

In finance and banking, there is an equation that can be used to compute the future value of an annuity. Not surprisingly it is called the “future value” equation. It has several variations. One variation arises from the fact that a payment into an annuity might be made at the beginning of the payment period, or the end. Two forms of the future value equation exist to take this distinction into account:

**Payment at beginning:**

\[
fv = \frac{(ir + 1) \cdot pmt - (ir + 1)^{np} \cdot (ir \cdot pv + ir \cdot pmt + pmt)}{ir},
\]

**Payment at end:**

\[
fv = \frac{pmt - (ir + 1)^{np} \cdot (ir \cdot pv + pmt)}{ir},
\]

where

\[
pv = \text{Present value of the annuity, the account balance at the beginning of the annuity term.}
\]

\[
fv = \text{Future value of the annuity, the account balance at the end of the annuity term.}
\]

\[
np = \text{Number of payments into the annuity during its term.}
\]

\[
pmt = \text{Payment amount made during each period of the annuity.}
\]

\[
ir = \text{Interest rate per period.}
\]
There are a few important things to be aware of when dealing with financial equations. One is that amounts withdrawn from an annuity have a positive sign, amounts deposited into an annuity have a negative sign. Adverse interest rates have a negative sign. Another important point is that the interest rate is per period, not per year or per term, so a 12% annual interest rate means 1% per period (assuming monthly payments), and the correct numerical value for 1% is 0.01, not 1. Each of the above variables can be positive or negative numbers. A negative present value implies a deposit made at the beginning of the annuity’s term. A negative payment amount implies a net inflow of cash during each period. A negative interest rate implies an interest deduction from the account’s balance each period. Here is an example annuity computation:

\[ pv = 0 \]
\[ np = 120 \]
\[ pmt = -100 \]
\[ ir = 0.01 \]

These values describe an annuity such as an investment, with a favorable interest rate of 1% per period, a deposit of 100 currency units per period, and 120 payment periods. For these values, and assuming payment at the end of each period, the future value will be 23,003.87 currency units (rounded to the nearest 1/100). For the case of payment at the beginning of the period, the future value will be 23,233.91 currency units. The reader can check these results using a spreadsheet program, most of which have an \text{FV()}\ function available. Here are typical spreadsheet entries for this problem:

<table>
<thead>
<tr>
<th>Spreadsheet Entry</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>=FV(0.01;120;-100;0;0)</td>
<td>23,003.87</td>
<td>Payment at end.</td>
</tr>
<tr>
<td>=FV(0.01;120;-100;0;1)</td>
<td>23,233.91</td>
<td>Payment at beginning.</td>
</tr>
</tbody>
</table>

Now, we want to create functions for each of the variables in the future value equation, so we can solve for any value in an annuity. And it would be nice to be able to deal with the issue of payment at beginning or end without having to write two sets of equations. The first step in our project is to create a meta-equation that accommodates the issue of payment at beginning or end. The new equation deals with this issue by including a new variable “pb” that makes the equation switch between modes. Here is the meta-equation:

\[ eq:fv=((pb*ir+1)*pmt-(ir+1)^np*(pb*ir*pmt)+pmt+ir*pv))/ir; \]

Now for a sanity check. Can we derive the two forms of the future value equation from this meta-equation, simply by setting the value of of the variable \text{pb}? Let’s see:

\[ eq; \] (let’s check to see if the equation was read correctly)
\[ pb:0; \] (set \text{pb} = 0, meaning payment at end)
\[ ev(eq); \] (use "evaluate" to interpret the equation)
\[ pb:1; \] (set \text{pb} = 1, meaning payment at beginning)
\[ ev(eq); \] (use "evaluate" to interpret the equation)
Perfect result. Note that a variable that assumes values of zero or one can be used as a switch, in effect adding or removing terms from an equation. The function "ev()" simplifies equations that are submitted to it, and in this case, if the pb variable is set to zero, "ev()" eliminates any terms that are multiplied by it. If the pb variable is set to one, the "ev()" function retains the associated terms and eliminates only the pb variable itself. The meta-equation in essence recreates the two canonical forms of the future value equation, and applying the "ev()" function proves this. But it’s important to note that, even without this term elimination, the original meta-equation will still deliver correct results if the pb variable is set to zero or one to convey the user’s intentions. This idea seems to be successful at setting the value of the pb variable in the meta-equation appears to recreate the two original forms of the future value equation. This will greatly reduce the number of required equation forms and the overall complexity of the project.

Now we can create all the forms the future value equation can take, based on the variables within it. First, some keyboard entries to make sure this approach will work:

```
kill(pb); (remove any assignment to pb)
eq; (is our equation still present?)
sol:solve(eq,pv); (solve for variable pv, present value)
rhs(sol[1]); (isolate the body of the solution for use in a function declaration)
```

It seems we can create versions of the meta-equation to provide results for each of the variables in the original equation. Now write a function that automates the process of solving for specific variables and isolating the desired expression:

```
solve_for_var(eq,v) := ev(rhs(solve(eq,v)[1]),fullratsimp);
```

The reader will appreciate that I have skipped over a few details in the design of this function. The “solve_for_var()” function solves for a particular variable, isolates the required expression, prepares it for assignment to a function description, and simplifies it (“fullratsimp”) if possible. Such a function has the advantage that its actions are encapsulated, they don’t have to be entered repeatedly, and further, if a change is made to the algorithm, the change only needs to be entered in one location to have an effect on all the derivations. Having written a function to create equations for each of the variables, we can now simply and quickly produce all the desired functions:

```
fpv(fv,np,pmt,ir,pb) := solve_for_var(eq,pv);
ffv(pv,np,pmt,ir,pb) := solve_for_var(eq,fv);
fnp(fv,pv,pmt,ir,pb) := solve_for_var(eq,np);
fpmt(fv,pv,np,ir,pb) := solve_for_var(eq,pmt);
fir(fv,pv,np,pmt,pb) := solve_for_var(eq,ir);
```

NOTE: The interest rate case cannot be solved in closed form, and the function declaration is only present for completeness ... Even though there is a derivation...
for “ir”, the interest rate, it turns out this case is not soluble in closed form, as Isaac Newton discovered in his second career as Chancellor of the Exchequer, during which time he applied his substantial mathematical gifts to the mundane problems of finance. I only include the derivation to reveal something about symbolic math packages. On occasion they will very happily produce meaningless results. And if the resulting function is tested, it will ... well, how shall I say this? ... go completely nuts. Try it:

fir(23003.87,0,-100,120,0);

This produces a huge display of garbage results. The reason for the crazy behavior? Simple “solve()” didn’t really solve this one. The result created by “solve()” has “ir” terms on both sides of the equals sign. An error message would have been more convenient, but no symbolic math packages are perfect, and Maxima is, after all, free. In normal practice, computing interest rates involves a root finding approach using another form of the equation.

2.4 Systems of Equations

Systems of (non)linear equations can be solved with the function “algsys([eq1,eq2,...,eqn],[var1,var2,...,varn])”. Consider the system

Example 2.4.1.
\[
\begin{align*}
  x^2 - x + y &= 0, \\
  2x - y &= 0,
\end{align*}
\]

having the solution \((x, y) = (-1, -2)\). Here is the corresponding input to wxMaxima

eq1: x^2 - x + y = 0;

\[\text{eq2: } 2\times - y = 0; \]

algsys([eq1,eq2], [x, y]);

The function “eliminate([eq1,eq2,...,eqn],[variables to be eliminated])” can be used to eliminate variables in an algebraic equation system. Consider eliminating \(x\) from the above system:

Example 2.4.2.
\[\text{Example 2.4.2.} \]

eq1: x^2 - x + y = 0;

\[\text{eq2: } 2\times - y = 0; \]

eliminate([eq1,eq2], [x]);

The result from this is simply \([y*(y+2)]\), which should be interpreted as the equation \(y(y + 2) = 0\).

Linear systems can be solved with the function “linsolve([eq1,eq2,...,eqn],[var1,var2,...,varn])”. Consider the following three-dimensional system
2.5. Evaluating Limits

Example 2.4.3.

\[
\begin{align*}
2x + y - z &= 0, \\
x - y - z &= -2, \\
5x - 3y + 2z &= 5,
\end{align*}
\]

having the solution \((x, y, z) = \frac{1}{19} \cdot (12, 13, 37)\). The corresponding input to wxMaxima would be

\[
eq 1: 2x + y - z = 0; \\
eq 2: x - y - z = -2; \\
eq 3: 5x - 3y + 2z = 5; \\
linsolve([eq1, eq2, eq3], [x, y, z]);
\]

2.5 Evaluating Limits

Maxima is very good at evaluating limits. For example, to calculate

\[
L = \lim_{x \to 0} \frac{1 - \cos x}{x}
\]

we can use the menu “Calculus” in the GUI or simply invoke the command:

\[
\text{limit}((1-\cos(x))/x, x, 0);
\]

A one-sided limit, e.g.,

\[
L = \lim_{x \to 0^+} \frac{1 - \cos x}{x}
\]

can be evaluated with

\[
\text{limit}((1-\cos(x))/x, x, 0, \text{plus});
\]

Similarly, the other one-sided limit can be evaluated with the option “minus”. The names “+inf” and “-inf” for \(-\infty\) and \(+\infty\), respectively. For a limit that does not exist, Maxima returns “ind” or “infinity”. For a limit that does exist and is infinite, Maxima returns “+inf” or “-inf”. Finally, we look at a case with a limit of an expression containing absolute values.

Example 2.5.1. The limit

\[
\lim_{x \to 0} \frac{x^2 + |x|}{x^2 - |x|}
\]

can be evaluated according to

\[
\text{limit}((x^2+\text{abs}(x))/(x^2-\text{abs}(x))), x, 0);
\]
2.6 Differential Calculus

wxMaxima can be used for differentiating expressions and factoring the derivatives. For example, consider the function

\[ f(x) = (x^2 + 3x + 2)^2 (x + 1)^2 \]

To find \( f'(x) \) and \( f''(x) \) and simplify the resulting expressions, we can choose from the GUI menus “Calculus/Differentiate” and “Simplify/Factor Expression” twice. In the dialog box, we can choose how many times to differentiate, however, since we want both derivatives, we choose to differentiate once repeatedly. The corresponding command lines are:

```
f(x) := ( x^2 + 3*x + 2 )^2 * ( x + 1 )^2;
diff(f(x),x,1);
factor(%);
fp(x) := ''%;
diff(fp(x),x,1);
factor(%);
fpp(x) := ''%;
```

As mentioned earlier, “\%” refers to the previous result and the double quote in the assignments for “fp(x)” and so on makes Maxima execute the differentiation and evaluate the expression. Numerical values for \( f'(x) \) and \( f''(x) \) can now be obtained by referencing, e.g., “fp(3/2)” for a rational form and “float( fp(3/2) )” for floating-point value.

We now continue with a look at differential equations. A differential equation is an equation expressing the relationship between a function and one or more of its derivatives. When the equation is supplemented with an initial condition, we have an initial value problem. When we have one or more boundary conditions, we are dealing with a boundary value problem.

**Example 2.6.1.** A differential equation with initial condition.

\[ y(t) + rcy'(t) = b , \]
\[ y(0) = a . \]

In the first equality, we see that an unknown function \( y(t) \) is added to its derivative \( y'(t) \), which is scaled by the two constant parameters \( r \) and \( c \). In the second equality, an initial value is assigned to the function \( y(t) \). The meaning of this statement is that, when \( t = 0 \), \( y(t) = a \). Unlike numerical equations, differential equations describe dynamic processes, i.e., things are changing. The derivative term \( y'(t) \) describes the rate of change in \( y(t) \). The variable \( t \) represents time (although this is just an interpretation). At time zero, the function \( y(t) \) equals \( a \), therefore at that moment the derivative term \( y'(t) \) is equal to \( \frac{a}{r} \). Notice that \( y'(t) \), which represents the rate of change in \( y(t) \), has its largest value at time zero. Because of how the equation is written, we see that the value of \( y'(t) \) (the
rate of change) becomes proportionally smaller as \( y(t) \) becomes larger. Eventually, for some very large value of \( t \), the rate of change represented by \( y'(t) \) becomes arbitrarily small, as \( y(t) \) approaches the value of \( b \), but never quite gets there. Put very simply, this equation describes a system in which the rate of change in the value of \( y(t) \) depends on the remaining difference between \( y(t) \) and \( b \), and as that difference decreases, so does the rate of change. As it happens, this equation is used to describe many natural processes, among which are:

Electronic circuits consisting of resistors and capacitors (hence the terms \( r \) and \( c \)), where the voltage on a capacitor changes in a way that depends on the current flowing through a resistor, and the value of the resistor’s current depends on the voltage on the capacitor. Heat flow between a source of heat energy and a cooler object being heated by it (like a pot on a stove). In such a system, the temperature of the heated body changes at a rate that depends on the remaining difference in temperature between the two bodies. The rate of gas flow between two pressure vessels with a constricted passage between them. In this system also, the rate of gas flow depends on the remaining pressure difference, and the pressure difference declines over time. This is by no means a comprehensive list of applications. But the statements for a differential equation are only the beginning, and not all differential equations have analytical solutions (solutions expressible in closed form, consisting of normal mathematical operations). Others require numerical methods and are only solvable in an approximate sense.

Let’s see if Maxima can find a solution for this equation. Here are the steps:

\[
\text{let's see if Maxima can find a solution for this equation. Here are the steps:}
\]

\[
\text{Note the way the equation and initial condition are defined in the first two lines.}
\]

\[
\text{A higher derivative } \frac{\text{d}^n y(t)}{\text{d}t^n} \text{ can be defined with the command } \text{'diff}(y(t), t, n), \text{ where the prime is included to force evaluation of the derivative when the equation is solved in the third line. From the fourth line we now have a function that embodies the solution to our differential equation. We can use it to solve real-world problems. Here’s an example from electronics:}
\]

**Example 2.6.2. R-C circuit diagram**

In this experiment, we have an electronic circuit consisting of a resistor and a capacitor. At time zero, we close a switch that connects our circuit to a battery, we then use an oscilloscope to measure the voltage on the capacitor over time (cf. the diagram 2.2). Here are the Maxima instructions to set up and graph the response of the described circuit:

\[
\text{r:10000;}
\text{c:100e-6;}
\text{a:0;}
\text{b:12;}
\text{wxplot2d(f(t,r,c,a,b),[t,0,5],}
\]

\[
\text{Note the way the equation and initial condition are defined in the first two lines.}
\]

\[
\text{A higher derivative } \frac{\text{d}^n y(t)}{\text{d}t^n} \text{ can be defined with the command } \text{'diff}(y(t), t, n), \text{ where the prime is included to force evaluation of the derivative when the equation is solved in the third line. From the fourth line we now have a function that embodies the solution to our differential equation. We can use it to solve real-world problems. Here’s an example from electronics:}
\]

**Example 2.6.2. R-C circuit diagram**

In this experiment, we have an electronic circuit consisting of a resistor and a capacitor. At time zero, we close a switch that connects our circuit to a battery, we then use an oscilloscope to measure the voltage on the capacitor over time (cf. the diagram 2.2). Here are the Maxima instructions to set up and graph the response of the described circuit:

\[
\text{r:10000;}
\text{c:100e-6;}
\text{a:0;}
\text{b:12;}
\text{wxplot2d(f(t,r,c,a,b),[t,0,5],}
\]
We proceed to study a little more complex differential equation belonging to the same general class.

**Example 2.6.3. Wave-driven equation**

Consider an inhomogeneous differential equation driven by a continuous waveform. This could also be an R-C circuit, or any natural system in which there is a path of resistance to the flow of something cyclical. Here are the command lines:

```plaintext
eq: y(t) := -r*c*'diff(y(t),t) + m*sin(%omega*t);
sol: desolve(eq,y(t));
```

There is a problem with this solution - the group of terms at the right includes the familiar $e^{-\frac{t}{rc}}$ expression that appears in equations with defined initial values. But, because this equation describes a continuous process with no beginning and no end, we need to set the conditions at time zero in such a way that all times will be treated equally (and the right-hand subexpression will be eliminated). After mulling this over, we decide the correct way to characterize the initial conditions would be to submit the right-hand expression as a negative value at time zero, which has the effect of preventing the assignment of a special time-zero value. Here is that entry and the outcome:

```plaintext
init_val: -(c*m*r*(%e^-(t/r*c))*%omega)/(c^2*r^2*%omega^2+1);
atvalue(y(t),t=0,init_val);
sol: desolve(eq,y(t));
```

```plaintext
f(t,r,c,%omega,m) := ev(rhs(sol),fullratsimp,factor);
/* simplify, factor and declare a function */
```
2.6. DIFFERENTIAL CALCULUS

We now have a working embodiment of this equation, and it turns out this form has as many real-world applications as the earlier example. Here is another electronic example using an R-C circuit, but this time with a sinewave generator driving our circuit:

```plaintext
m:1;
f:440;
r:1000;
c:0.05e-6;
%omega:2*%pi*f;
xplot2d([sin(%omega*t),f(t,r,c,%omega,m)],[t,0,.005],
[gnuplot_preamble, "unset key;
set title 'f = 440 Hz, r = 1000, c = 100 uf'"]);
```

2.6.1 Computation of Solutions with the Laplace Transformation

The Laplace transformation is a powerful tool to solve a vast class of ordinary differential equations. The principle is simple: The differential equation is mapped onto a linear algebraic equation, which is solved. The solution of the algebraic equation is then mapped back onto the solution of the given differential equation. The mathematical theory of the Laplace transformation is not trivial, but for practical purposes it is often sufficient to simply apply the transformation.

The Laplace transformation maps a transcendental function onto a rational expression in a new variable:

```plaintext
laplace(exp(-b*t)*cos(w*t),t,s);
```

\[
\frac{s + b}{s^2 + 2 b s + b^2 + w^2}
\]

\[\text{assume}(w > 0);\]

\[\text{[w > 0]}\]
Without that assumption, Maxima would ask later whether \( w \) is zero or nonzero. The inverse Laplace transformation maps a rational expression onto a transcendental function:

\[
\text{ilt}(\%, s, t); = -b \ t \ \%e \cos(t \ w)
\]

The knowledge of these functions is almost sufficient to solve an ordinary differential equation:

\[
\text{ode}: \quad '\text{diff}(y(t), t, 2) + 5*'\text{diff}(y(t), t) + 4*y(t) = t;
\]

Recall the importance of writing a quoted \texttt{diff} to represent the derivative as well as writing the function \( y \) with its formal argument: \( y(t) \). It is not sufficient to write \( '\text{diff}(y, t, 2) \)

\[
\frac{d^2}{dt^2} (y(t)) + 5 \frac{d}{dt} (y(t)) + 4 \ y(t) = t
\]

Next, we specify the initial conditions. We do that with the function \texttt{atvalue}:

\[
\text{atvalue}(y(t), t=0, 0); = 0
\]

\[
\text{atvalue}(\text{diff}(y(t), t), t=0, 0); = 0
\]

Now we can compute the Laplace transform of the equation:

\[
\text{lap_ode} : \text{laplace(ode, t, s)};
\]

\[
\frac{2}{s} \text{laplace}(y(t), t, s) + 5 \ s \text{laplace}(y(t), t, s) + \frac{1}{4} \text{laplace}(y(t), t, s) = \frac{1}{2}
\]

This is a linear equation in the unknown \( \text{laplace}(y(t), t, s) \). We solve it with \texttt{solve}:

\[
\text{sol} : \text{solve}(\%, \ '\text{laplace}(y(t), t, s));
\]

Note that you have to write the unknown with a quote. Without the quote, Maxima would try to evaluate the expression \( \text{laplace}(y(t), t, s) \).
The answer is a list with one equation, the solution of the linear equation. Now, we can apply the inverse Laplace transformation. To apply apply a transformation to the elements of a list, we have to use map:

\[
\text{map}( \lambda [\text{eq}], \text{ilt(eq, s, t)), sol);}
\]

\[
- t \quad - 4 t
\]

\[
\%e \quad \%e \quad t \quad 5
\]

\[
[y(t) = \frac{\%e}{3} - \frac{\%e}{48} + \frac{t}{4} - \frac{5}{16}]
\]

This is again a list with one equation. Note that the inverse Laplace transformation was applied to both sides of the equation: On the left hand side, the application of the inverse Laplace transformation to the Laplace transform of \( y(t) \) yields \( y(t) \). On the right hand side, the application of the inverse Laplace transformation yields the solution of the given differential equation. It is of course possible to solve this ordinary differential equation with the command \texttt{ode2}:

Next, we consider a system of two ordinary differential equations. First, we define the equations:

\[
\text{assume(s>0);}
\]

\[
[s > 0]
\]

\[
diff\_eq1: \text{’diff(f(x), x, 2) + ’diff(g(x), x) + 3*f(x) = 15*exp(-x);}
\]

\[
diff\_eq2: \text{’diff(g(x), x, 2) - 4*’diff(f(x), x) + 3*g(x) = 15*sin(2*x);}
\]

Next, we define the initial values. For this example, we have initial values for both functions and their derivatives at point \( x = 0 \).

\[
\text{atvalue (f(x), x=0, 35);}\]

\[
\quad 35
\]

\[
\text{atvalue (’diff(f(x), x), x=0, -48);}\]

\[
\quad -48
\]

\[
\text{atvalue (g(x), x=0, 27);}\]

\[
\quad 27
\]

\[
\text{atvalue (’diff(g(x), x), x=0, -55);}\]

\[
\quad -55
\]

Now we can compute the Laplace transforms:

\[
\text{lap\_eq1:laplace(diff\_eq1,x,s);}\]

\[
\text{lap\_eq2:laplace(diff\_eq2,x,s);}\]

These are two linear equations in the unknowns \( \text{laplace(f(x), x, s)} \) and \( \text{laplace(g(x), x, s)} \). We solve them with \texttt{solve}:

\[
\text{linsolve([lap\_eq1,lap\_eq2],[’laplace(f(x), x, s),’laplace(g(x), x, s))]);}
\]

The denominators of the solutions are polynomials in \( s \) of degree 7. The computation of the inverse Laplace transform requires that the denominators are decomposed into linear and quadratic factors. The function \texttt{ilt} does not always factor denominators, so we do it:
factored: map(lambda([eq], factor(eq)), %);

Now we can apply the inverse Laplace transform to both equations:

sol: map(lambda([eq], ilt(eq, s, x)), factored);

We obtain a solution for the given system of ordinary differential equations:

\[
\begin{align*}
  f(x) &= -15 \sin(3x) + 2 \cos(2x) + 30 \cos(x) + 3 e^{-x}, \\
  g(x) &= 30 \cos(3x) + \sin(2x) - 60 \sin(x) - 3 e^{-x}
\end{align*}
\]

In principle we are done, but there is still one interesting technicality: How can we prove that the solution satisfies the given system of equations? The proof requires four additional steps. First, we write the two differential equations as a system:

\[
\begin{align*}
  \text{ode\_system:} & \quad \left[ \begin{align*}
    (f(x)) &= 15 \cos(3x) + 2 \sin(2x) + 30 \cos(x) + 3 e^{-x}, \\
    (g(x)) &= 30 \cos(3x) + \sin(2x) - 60 \sin(x) - 3 e^{-x}
  \end{align*} \right],
  \\
  \text{ode\_system, sol;}
\end{align*}
\]

Next we substitute the solution into that system:

\[
\begin{align*}
  \text{ode\_system, sol;}
  \begin{align*}
    (f(x)) &= 15 \cos(3x) + 2 \sin(2x) + 30 \cos(x) + 3 e^{-x}, \\
    (g(x)) &= 30 \cos(3x) + \sin(2x) - 60 \sin(x) - 3 e^{-x}
  \end{align*}
\end{align*}
\]
We obtain a list of two equations that contain symbolic derivatives. To enforce the computation of the derivatives, we have to evaluate the equations:

\[
\text{map (lambda( [eq], ev(eq, diff)), %);} \\
\text{[45 sin(3 x) + 3 (- 15 sin(3 x) + 2 cos(2 x) + 30 cos(x) + 3 e )} \\
\text{- 6 cos(2 x) - 90 cos(x) + 6 e = 15 e ,} \\
\text{3 (30 cos(3 x) + sin(2 x) - 60 sin(x) - 3 e ) - 270 cos(3 x)} \\
\text{- 4 (- 45 cos(3 x) - 4 sin(2 x) - 30 sin(x) - 3 e ) - 4 sin(2 x)} \\
\text{+ 60 sin(x) - 3 e = 15 sin(2 x)]}
\]

This looks better, but simplification is obviously needed:

\[\text{trigsimp(%)};\]
\[\text{[15 e = 15 e , 15 sin(2 x) = 15 sin(2 x)]}\]

These two identities complete the proof. It should be noted that the equations of this example can also be solved with \texttt{desolve}.

### 2.6.2 Differentiation Using the Chain Rule

A common application of the chain rule for differentiation is \textit{implicit differentiation}, which Maxima can handle, e.g., using the function \texttt{depends}. Consider finding \(\frac{dy}{dx}\) from the implicit expression \(x^2 + y^2 = 5^2\) as follows:
(\%i1)  eqn : x^2 + y^2 = 25;
    2  2
(\%o1)  y + x = 25
(\%i2)  depends(y,x);
(\%o2) [y(x)]
(\%i3)  deriv_of_eqn:diff(eqn,x);
    dy
(\%o3)  2 y -- + 2 x = 0
    dx
(\%i4)  solve(deriv_of_eqn,'diff(y,x));
    dy  x
(\%o4)  [-- = - -]
    dx  y

Working with derivatives of unknown functions can be cumbersome in Maxima. If we want, for example, the first order Taylor polynomial of \( f(x + x^2) \) about \( x = 1 \), we get

(\%i1)  taylor(f(x + x^2),x,1,1);
(\%o1)  f(2)+(at('diff(f(x^2+x),x,1),x=1))*(x-1)+...

To “simplify” the Taylor polynomial, we must assign a gradient to \( f \)

(\%i2)  gradef(f(x),df(x))$
(\%i3)  taylor(f(x+x^2),x,1,1);
(\%o3)  f(2)+3*df(2)*(x-1)+...

This method works well for simple problems, but it is tedious for functions of several variables or high order derivatives. The positional derivative package \texttt{pdiff} gives an alternative to using \texttt{gradef} when working with derivatives of unknown functions.

To use the positional derivative package, you must load it from a Maxima prompt. Assuming \texttt{pdiff.lisp} is in a directory that Maxima can find, this is done with the command

(\%i4)  kill(all)$
(\%i1)  load("pdiff.lisp")$

Use the full pathname if Maxima cannot find the file. Note that the \texttt{kill(all)} is needed because the \texttt{gradef} definition will conflict with the proper functioning of the \texttt{diff} commands. Loading \texttt{pdiff.lisp} sets the option variable \texttt{use_pdiff} to true; when \texttt{use_diff} is true, Maxima will indicate derivatives of unknown functions positionally. To illustrate, the first three derivatives of \( f \) are

(\%i2)  [diff(f(x),x),
     diff(f(x),x,2),
     diff(f(x),x,3)];
(\%o2)  [f([(1)](x),f([(2)](x),f([(3)](x))]}
The subscript indicates the order of the derivative; since \( f \) is a function of one variable, the subscript has only one index. When a function has more than one variable, the subscript has an index for each variable.

By an unknown function, we mean a function not bound to a formula and that has a derivative unknown to Maxima.

```maxima
(%i3) [diff(f(x,y),x,0,y,1), diff(f(y,x),x,0,y,1)];
(%o3) [f[(0,1)](x,y),f[(1,0)](y,x)]
```

Setting `use_pdiff` to false (either locally or globally) inhibits derivatives from being computed positionally.

```maxima
(%i4) diff(f(x,x^2),x), use_pdiff : false;
(%o4) 'diff(f(x,x^2),x,1)
```

```maxima
(%i5) diff(f(x,x^2),x), use_pdiff : true;
(%o5) f[(1,0)](x,x^2)+2*x*f[(0,1)](x,x^2)
```

Taylor polynomials of unknown functions can be found without using `gradef`. An example:

```maxima
(%i6) taylor(f(x+x^2),x,1,2);
(%o6) f(2)+3*f[(1)](2)*(x-1)+((2*f[(1)](2)+9*f[(2)](2))*(x-1)^2)/2+...
```

Additionally, we can verify that \( y = f(x-ct) + g(x+ct) \) is a solution to a wave equation without using `gradef`

```maxima
(%i7) y : f(x-c*t) + g(x+c*t)$
(%i8) ratsimp(diff(y,t,2) - c^2 * diff(y,x,2));
(%o8) 0
```

Expressions involving positional derivatives can be differentiated

```maxima
(%i10) e : diff(f(x,y),x);
(%o10) f[(1,0)](x,y)
(%i11) diff(e,y);
(%o11) f[(1,1)](x,y)
```

The chain rule is applied when necessary

```maxima
(%i12) [diff(f(x^2),x), diff(f(g(x)),x)];
(%o12) [2*x*f[(1)](x^2),g[(1)](x)*f[(1)](g(x))]
```

The positional derivative package does not alter the way known functions are differentiated

```maxima
(%i13) diff(exp(-x^2),x);
(%o13) -2*x*exp(-x^2)
```

To convert positional derivatives to standard Maxima derivatives, use `convert_to_diff`
(\%i14) \hspace{1em} e : \{\text{diff}(f(x),x), \text{diff}(f(x,y),x,1,y,1)\};
(\%o14) \hspace{1em} \{f[(1)](x), f[(1,1)](x,y)\}
(\%i15) \hspace{1em} e : \text{convert_to_diff}(e);
(\%o15) \hspace{1em} [\text{'diff}(f(x),x,1), \text{'diff}(f(x,y),x,1,y,1)]

To convert back to a positional derivative, use \text{ev} with \text{diff} as an argument

(\%i16) \hspace{1em} \text{ev}(e, \text{diff});
(\%o16) \hspace{1em} \{f[(1)](x), f[(1,1)](x,y)\}

Conversion to standard derivatives sometimes requires the introduction of a dummy variable. An example:

(\%i17) \hspace{1em} e : \text{diff}(f(x,y),x,1,y,1);
(\%o17) \hspace{1em} f[(1,1)](x,y)
(\%i18) \hspace{1em} e : \text{subst}(p(s), y, e);
(\%o18) \hspace{1em} f[(1,1)](x,p(s))
(\%i19) \hspace{1em} e : \text{convert_to_diff}(e);
(\%o19) \hspace{1em} \text{at}(\text{'diff}(f(x,y),x,1,y,1), [y=p(s)])

Dummy variables have the form $c_i$, where $i = 0,1,2 \ldots$ and $c$ is the value of the option variable \text{dummy_char}. The default value for \text{dummy_char} is $%x$. If a user variable conflicts with a dummy variable, the conversion process can give an incorrect value. To convert the previous expression back to a positional derivative, use \text{ev} with \text{diff} and \text{at} as arguments

(\%i20) \hspace{1em} \text{ev}(e, \text{diff}, \text{at})
(\%o20) \hspace{1em} f[(1,1)](x,p(s))

Maxima correctly evaluates expressions involving positional derivatives if a formula is later given to the function. (Thus converting an unknown function into a known one.) Here is an example; let

(\%i21) \hspace{1em} e : \text{diff}(f(x^2),x);
(\%o21) \hspace{1em} 2*x*f[(1)](x^2)

Now, give $f$ a formula

(\%i22) \hspace{1em} f(x) := x^5$
\hspace{1em} and evaluate $e$

(\%i23) \hspace{1em} \text{ev}(e);
(\%o23) \hspace{1em} 10*x^9

This result is the same as

(\%i24) \hspace{1em} \text{diff}(f(x^2),x);

In this calculation, Maxima first evaluates $f(x)$ to $x^{10}$ and then does the derivative. Additionally, we can substitute a value for $x$ before evaluating
2.7. INTEGRAL CALCULUS

We can duplicate this with

(%i26) subst(2,x,diff(f(x^2),x));
(%o26) 5120
(%i27) remfunction(f);
(%o27) [f]

We can also evaluate a positional derivative using a local function definition

(%i28) e : diff(g(x),x);
(%o28) g[(1)](x)
(%i29) e, g(x) := sqrt(x);
(%o29) 1/(2*sqrt(x))
(%i30) e, g = sqrt;
(%o30) 1/(2*sqrt(x))
(%i31) e, g = h;
(%o31) h[(1)](x)
(%i32) e, g = lambda([t],t^2);
(%o32) 2*x

2.7 Integral Calculus

Although the process of finding an integral can be viewed as the inverse of the process of finding a derivative, in practice finding an integral is more difficult. It turns out that the integral of fairly simple looking functions cannot be expressed in terms of closed-form functions, and this has been one motivation for the introduction of definitions for a variety of “special functions”, as well as efficient methods for numerical integration.

In the following, we always assume we are dealing with a real function of a real variable and that the function is single-valued and continuous over the intervals of interest. The Maxima manual entry for integrate includes (we have made some changes and additions):

Function: integrate(expr, var)
Function: integrate(expr, var, a, b)
Attempts to symbolically compute the integral of expr with respect to var.
integrate(expr, var)
is an indefinite integral, while
integrate(expr, var, a, b)
is a definite integral, with limits of integration a and b. The integral is returned if integrate succeeds. Otherwise the return value is the noun form of the integral (the quoted operator integrate) or an expression containing one or more noun forms. The noun form of integrate is displayed with an integral sign if display2d is set to true (which is the default). In some circumstances it is useful to construct a noun form by hand, by quoting integrate with a single
quote, e.g., \texttt{integrate(expr, var)}. For example, the integral may depend on some parameters which are not yet computed. The noun may be applied to its arguments by \texttt{ev(iexp, nouns)} where \texttt{iexp} is the noun form of interest.

To integrate a Maxima function \( f(x) \), insert \( f(x) \) in the \texttt{expr} slot. \texttt{integrate} does not respect implicit dependencies established by the \texttt{depends} function. \texttt{integrate} may need to know some property of the parameters in the integrand. \texttt{integrate} will first consult the \texttt{assume} database, and, if the variable of interest is not there, \texttt{integrate} will ask the user. Depending on the question, suitable responses are yes; or no.; or pos.; zero.; or neg.; Thus, the user can use the \texttt{assume} function to avoid all or some questions.

\textbf{2.7.1 Integration Examples}

\textbf{Example 2.7.1.} \textit{Our first example is the indefinite integral } \int \sin^3 x \, dx:\

\begin{verbatim}
integrate((sin(x))^3, x);
diff(%,x);
trigsimp(%);
\end{verbatim}

Notice that the indefinite integral returned by \texttt{integrate} does not include the arbitrary constant of integration which can always be added. If the returned integral is correct (up to an arbitrary constant), then the first derivative of the returned indefinite integral should be the original integrand, although we may have to simplify the result manually by using the command \texttt{trigsimp}.

\textbf{Example 2.7.2.} \textit{Our second example is another indefinite integral } \int \frac{x}{\sqrt{b^2 - x^2}} \, dx:\

\begin{verbatim}
integrate(x/ sqrt(b^2 - x^2), x);
diff(%,x);
\end{verbatim}

The definite integral can be related to the “area under a curve” and is the more accessible concept, while the integral is simply a function whose first derivative is the original integrand.

\textbf{Example 2.7.3.} \textit{Consider the definite integral } \int_{0}^{\pi} e^x \cos^2 x \, dx:\

\begin{verbatim}
i3 : integrate(cos(x)^2 * exp(x), x, 0, %pi);
\end{verbatim}

Instead of using \texttt{integrate} for a definite integral, you can try \texttt{ldefint} (think Limit definite integral), which may provide an alternative form of the answer (if successful).

Function: \texttt{ldefint(expr, x, a, b)}

This function attempts to compute the definite integral of \texttt{expr} by using limit to evaluate the indefinite integral of \texttt{expr} with respect to \( x \) at the upper limit \( b \) and at the lower limit \( a \). If it fails to compute the definite integral, \texttt{ldefint} returns an expression containing limits as noun forms. \texttt{ldefint} is not called from \texttt{integrate}, so executing \texttt{ldefint(expr, x, a, b)} may yield
a different result than \( \text{integrate}(\text{expr}, x, a, b) \). \ldefint \) always uses the same method to evaluate the definite integral, while \( \text{integrate} \) may employ various heuristics and may recognize some special cases.

**Example 2.7.4. Use of \( \ldefint \), as well as the direct use of \( \text{defint} \) (which bypasses \( \text{integrate} \)):

\[
\ldefint(\cos(x)^2 * \exp(x), x, 0, \%pi); \\
\text{defint}(\cos(x)^2 * \exp(x), x, 0, \%pi);
\]

**Example 2.7.5. A definite integral over an infinite range \( \int_{-\infty}^{\infty} x^2 e^{-x^2} \, dx \)

\[
\text{integrate}(x^2 * \exp(-x^2), x, \infty, \infty);
\]

To check this integral, we first ask for the indefinite integral and then check it by differentiation.

\[
i1 : \text{integrate}(x^2*\exp(-x^2),x) ; \\
diff(i1,x);
\]

Thus the indefinite integral is correct. The second term, heavily damped by the factor \( e^{-x^2} \) at \( \pm \infty \), does not contribute to the definite integral. The first term is proportional to the (Gauss) error function, \( \text{erf}(x) \), in which \( x \) is real. For (in general) complex \( w = u + iv \),

\[
\text{erf}(w) = \frac{2}{\sqrt{\pi}} \int_0^w e^{-z^2} \, dz ,
\]

in which we can integrate over any path connecting \( 0 \) and \( w \) in the complex \( z = x + iy \) plane, since the integrand is an entire function of \( z \) (no singularities in the finite \( z \) plane). We can plot this function by invoking the Plot/Plot 2d GUI-box and get the graph in Figure 2.4. Maxima’s limit function confirms the indications of the plot:

\[
[\text{limit}({\text{erf}(x)},x,\infty), \text{limit}({\text{erf}(x)},x,\text{minf})]; \\
[1, -1]
\]

These limits can be used in the result for the indefinite integral above.

**Example 2.7.6. Use of \( \text{assume} \).

We would like to compute the definite integral for parameter \( a \):

\[
\int_0^\infty x^a (x+1)^{-\frac{5}{2}} \, dx , \quad a > 1 .
\]

We first invoke the assumption and do the whole computation according to:

\[
(\%i1) \ (\text{assume}(a>1), \text{facts}()); \\
(\%o1) \ [a > 1] \\
(\%i2) \ \text{integrate}(x^a/(x+1)^{(5/2)}, x, 0, \text{inf}); \\
2 \ a + 2
\]
CHAPTER 2. SYMBOLIC CALCULATION

Figure 2.4: The Gauss error function.

Is \( \frac{5}{3} \) an integer?
\( \text{no;} \)

Is \( 2a - 3 \) positive, negative, or zero?
\( \text{neg;} \)

\[
\beta(-a, a + 1) \quad \frac{3}{2}
\]

The combination of \( a > 1 \) and \( 2a - 3 < 0 \) means that we are effectively assuming \( 1 < a < \frac{3}{2} \). These assumptions about \( a \) imply that \( \frac{1}{2} < \frac{2a + 3}{2} < 1 \). To be consistent, we must hence answer no to the first question.

Let us tell Maxima to forget about the assume assignment and see what the difference is.

\[
\frac{a + 1}{x^a/(x+1)^{5/2}} , x , \infty
\]

Is \( a + 1 \) positive, negative, or zero?
\( \text{pos;} \)

Is \( a \) an integer?
\( \text{no;} \)

Is \( \frac{7}{2} \) an integer?
\( \text{no;} \)
Is $2 - 3$ positive, negative, or zero?

\[ \text{neg;} \]

\[ \beta(-a, a + 1) \]

\[ \text{[is(a>1), facts()];} \]

Thus we see that omitting the initial \texttt{assume(a>1)} statement causes \texttt{integrate} to ask four questions instead of two. We also see that answering questions posed by the \texttt{integrate} dialogue script does not result in population of the facts list.

The Maxima beta function has the manual entry \texttt{beta (a, b)} and is defined in terms of the gamma function as

\[ B(r, s) = \frac{\Gamma(r)\Gamma(s)}{\Gamma(r + s)} , \]

where the gamma function can be defined for complex \( z \) and \( \text{Re}(z) > 0 \) by the integral along the real \( t \)-axis

\[ \Gamma(z) = \int_{0}^{\infty} t^{z-1}e^{-t} \, dt . \]

and for \( \text{Im}(z) = 0 \) and \( \text{Re}(z) = n \) and \( n \) an integer greater than zero we have

\[ \Gamma(n + 1) = n! \]

Next, we take a look at changing the integration variable.

\textbf{Example 2.7.7.} We can make a change of variable to enable the return of an indefinite integral. The task is to evaluate the indefinite integral

\[ \int\frac{\sin(r^2) \, dr(x)}{q(r)} \, dx \]

by telling Maxima that the \( \sin(r^2) \) in the numerator is related to \( q(r) \) in the denominator by the derivative:

\[ \frac{dq(u)}{du} = \sin(u^2) \]

Then, we would manually rewrite the integrand (using the chain rule) as

\[ \sin(r^2) \frac{dr(x)}{q(r)} = \frac{1}{q(r)} \frac{dq(r)}{dr} \frac{dr(x)}{dx} = \frac{1}{q(r)} \frac{dq}{dx} = \frac{d \ln q}{dx} . \]

and hence obtain the indefinite integral \( \ln(q(r(x))) \).

In Maxima, we assign the derivative knowledge using \texttt{gradef}:
CHAPTER 2. SYMBOLIC CALCULATION

(%i1) gradeff(q(u),sin(u^2));
(%i2) integrand : sin(r(x)^2) * 'diff(r(x),x) / q(r(x));
     d   2
    (- (r(x))) sin(r(x))
   dx

(%o2)--------------------------------------------------------
       q(r(x))

(%i3) integrate(integrand,x);
(%o3) log(q(r(x)))

(%i4) diff(%),x;
     d   2
    (- (r(x))) sin(r(x))
   dx

(%o4)--------------------------------------------------------
       q(r(x))

Note that integrate pays no attention to depends assignments, so the briefer type of notation which depends allows with differentiation cannot be used with integrate:

(%i5) depends(r,x,q,r);
(%o5) [r(x), q(r)]

(%i6) integrand : sin(r^2) * 'diff(r,x) / q;
     dr 2
    -- sin(r)
   dx

(%o6)--------------------------------------------------------
       q

(%i7) integrate(integrand,x);

                                          
               2 [ dr
            sin(r ) I -- dx ] dx
                                          /

(%o7)--------------------------------------------------------
       q

which is fatally flawed, since Maxima pulled both sin(r(x)^2) and \( \frac{1}{q(r(x))} \) outside the integral. Of course, the above depends assignment will still allow Maxima to rewrite the derivative of q with respect to x using the chain rule:

(%i8) diff(q,x);

(%o8) d q dr
    -- --
   dr dx

Example 2.7.8. Integration of Rational Algebraic Functions.
2.7. INTEGRAL CALCULUS

A rational algebraic function can be written as a quotient of two polynomials. Consider the following function of \( x \).

\[
e_1 : x^2 + 3x - 2/(3x)+110/(3*(x-3)) + 12;
\]

We can obviously find the lowest common denominator and write this as the ratio of two polynomials, using either function \texttt{rat} or \texttt{ratsimp}.

\[
e_{11} : \text{ratsimp}(e1);
\]

\[
\frac{4}{2} \quad x + 3 \quad x + 2
\]

\[
\frac{2}{x - 3 \quad x}
\]

Because the polynomial in the numerator is of higher degree than the polynomial in the denominator, this is called an improper rational fraction. Any improper rational fraction can be reduced by division to a mixed form, consisting of a sum of some polynomial and a sum of proper fractions. We can recover the “partial fraction” representation in terms of proper rational fractions (numerator degree less than denominator degree) by using \texttt{partfrac(expr, var)}.

\[
e_{12} : \text{partfrac}(e_{11},x);
\]

With this function of \( x \) expressed in partial fraction form, you are able to write down the indefinite integral immediately (e.g., by inspection, without using Maxima). But, of course, we want to practice using Maxima!

\[
\text{integrate}(e_{11},x);
\]

\[
\text{integrate}(e_{12},x);
\]

Maxima has to do less work if you have already provided the partial fraction form as the integrand; otherwise, Maxima internally seeks a partial fraction form in order to do the integral.

Remark 2.7.1. There is a global parameter \texttt{logabs} whose default value is false and which affects what is returned with an indefinite integral containing logarithms.

\[
(\%i11) \text{logabs};
(\%o11) \text{false}
\]

\[
(\%i12) \text{integrate}(1/x,x);
(\%o12) \text{log(x)}
\]

\[
(\%i13) \text{diff}(\%o12,x);
(\%o13) 1
\]

\[
(\%i14) \text{logabs:}\text{true}\$
(\%i15) \text{integrate}(1/x,x);
\]

\[
(\%o15) \text{log(x)}
\]
When we override the default and set \texttt{logabs} to true, the argument of the \texttt{log} function is wrapped with an absolute value.

\subsection*{2.7.2 Integration by Parts}

Suppose \( f(x) \) and \( g(x) \) are two continuously differentiable functions in the interval of interest. Then the integration by parts rule states that given an interval with endpoints \((a, b)\), (and of course assuming the derivatives exist) one has

\[
\int_a^b f(x)g'(x) \, dx = \left[ f(x)g(x) \right]_a^b - \int_a^b f'(x)g(x) \, dx.
\]

This result follows from the product rule of differentiation and can also be used for indefinite integrals. In practice, when integration by parts is applied, we are confronted with an integral whose integrand can be viewed as the product of two factors, which we will call \( f(x) \) and \( h(x) \):

\[
\int f(x)h(x) \, dx
\]

and we wish to use integration by parts to get an integral involving the derivative of the first factor, \( f'(x) \), which will hopefully result in a simpler integral. We then identify \( h(x) = g'(x) \) and solve this equation for \( g(x) \) (by integrating; this is also a choice based on the ease of integrating the second factor \( h(x) \) in the given integral). Having \( g(x) \) in hand we can then write out the result using the indefinite integral integration by parts rule above. We can formalize this process for an indefinite integral with the Maxima code:

\[
iparts(f,h,var):= \text{block}([g],
    \text{\texttt{g}}: \text{\texttt{integrate}}(h,var), \text{\texttt{f*g - \texttt{integrate}}(\text{\texttt{g*diff}}(f,var),var)});
\]

Let’s practice with the integral \( \int x^2 \sin(x) \, dx \), in which \( f(x) = x^2 \) and \( h(x) = \sin(x) \), so we need to be able to integrate \( \sin(x) \) and want to transfer a derivative on to \( x^2 \), which will reduce the first factor to \( 2x \). Notice that it is usually easier to work with “Maxima expressions” rather than with “Maxima functions” in a problem like this.
iparts(x^2,sin(x),x);
ev(%,nouns);
expand(%);

collectterms(%,cos(x));

If we were not using Maxima, but doing everything by hand, we would use two integrations by parts (in succession) to remove the factor \(x^2\) entirely, reducing the original problem to simply knowing the integrals of \(\sin(x)\) and \(\cos(x)\).

2.7.3 Change of Variable

Many integrals can be evaluated most easily by making a change of variable of integration. A simple example is:

\[
\int 2x (x^2 + 1)^3 \, dx = \int (x^2 + 1)^3 \, d(x^2 + 1) = \int u^3 \, du = \frac{u^4}{4} = \frac{(x^2 + 1)^4}{4}.
\]

There is a function in Maxima, called \texttt{changevar} which will help you change variables in a one-dimensional integral (either indefinite or definite).

Function: \texttt{changevar(integral-expression, g(x,u), u, x)}

Makes the change of variable in a given noun \texttt{integrate} expression such that the old variable of integration is \(x\), the new variable of integration is \(u\), and \(x \) and \(u\) are related by the equation \(g(x,u) = 0\).

Example 2.7.9. Here we use this Maxima function on the simple indefinite integral we have just done “by hand”:

\[
\text{expr} : \text{'integrate}(2*x*(x^2+1)^3,x);
\text{changevar}(\text{expr},x^2+1-u,u,x);
\text{ev}(\%, \text{nouns});
\text{ratsubst}(x^2+1,u,\%);
\text{subst}(x^2+1,u,\%o3);
\text{subst}(u=(x^2+1),\%o3);
\]

The original indefinite integral is a function of \(x\), which we obtain by replacing \(u\) by its equivalent as a function of \(x\). We have shown three ways to make this replacement to get a function of \(x\), using \texttt{subst} and \texttt{ratsubst}.

Remark 2.7.2. It should be noted that \texttt{changevar} uses the solution of \(g(x,u) = 0\) for \(x\). Hence, if there are multiple such solutions, e.g., as in the case \(g(x,u) = x^2u - 1\), Maxima may use the wrong one. Maxima does not issue any warning
of this choice and the result of picking the wrong solution is in many cases a sign error in the integral.

**Example 2.7.10.** Consider the integral

\[ \int_1^\infty \ln \left| 1 - \frac{1}{w^2} \right| \, dw , \]

which should be negative, since the curve of the integrand is negative on the entire integration interval. Maxima can do this integral both using the substitution \( u = \frac{1}{w^2} \) and directly. The endpoint contributions are evaluated as limits. We see that, for the variable prior to the substitution, we have \( w = \pm \sqrt{u} \) and choosing the negative root gives a positive final result - which is clearly wrong.

```maxima
kill(all);
logabs : true$
intg : 'integrate(log((1-w^(-2))),w);
changevar(intg, w = 1/sqrt(u), u, w);
ev(% , nouns);
logcontract(%);
define(F(u),%);
Fupper : limit(F(u), u, 0, plus);
Flower : limit(F(u), u, 1, minus);
Fupper - Flower;
float(%);
kill(all);
plot2d([log((1-w^(-2)))], [w,1,5], [y,-2,1],
[plot_format, gnuplot])$
intg : 'integrate(log(abs(1-w^(-2))),w);
ev(% , nouns);
logcontract(%);
define(F(w),%);
Fupper : limit(F(w), w, inf, minus);
Flower : limit(F(w), w, 1, plus);
Fupper - Flower;
float(%);
```

### 2.8 Matrix Computations

Linear algebra and matrix computation are not the most powerful applications for Maxima, however, with Maxima, we can compute the determinant, inverse and eigenvalues and eigenvectors of matrices which have symbolic elements (i.e., elements which involve algebraic variables). We will take a brief look at the most common operations.

A matrix is entered in wxMaxima through the “Algebra/Enter Matrix or Generate Matrix” GUI-menu box. Consider
Example 2.8.1. A simple $3 \times 3$-matrix with symbolic elements and some basic matrix operations.

\[
A : \text{matrix}(
\begin{bmatrix}
0,1,a \\
1,0,1 \\
1,1,0
\end{bmatrix})
\);
\]

We take the transpose, determinant and inverse of the matrix. When inverting, we may also keep the determinant as a factor by specifying the option `detout`.

\[
\text{At} : \text{transpose}(A);
\text{Adet} : \text{determinant}(A);
\text{Ainv} : \text{invert}(A);
\text{Ainv2} : \text{invert}(A), \text{detout};
\]

In Maxima, matrices are multiplied with the dot-operator (`*` produces the element-wise product), i.e., the product of $A$ and its inverse is calculated according to:

\[
\text{Ainv} \cdot A;
\text{ratsimp}(\%);
\text{A} \cdot \text{Ainv};
\text{expand}(\%);
\text{factor}(\%);
\]

In order to find the eigenvalues and eigenvectors of $A$, we use the function `eigenvectors(A);`, which yields a rather lengthy result. In the resulting output, the first triple (or number of rows of the matrix) gives the eigenvalues of $A$ and the next gives their respective multiplicities (here each is unrepeated). The next three triples give the corresponding eigenvectors of $A$. In order to extract from this expression one of these eigenvectors, we may use the `part` function:

\[
\text{part}(\%, 2, 1, 1);
\]

2.9 Visualization

Visualization in wxMaxima can be done in various ways; inline with the GUI or using the “openmath” and “gnuplot” packages. These offer an enormous amount of options as to graphics appearance and features and their detailed description falls out of scope of this treatment. In most cases, detailed descriptions and examples can be googled from the internet. There are interactive menus for both two-dimensional and three-dimensional graphs behind the GUI-box “Plot”. Take a look at the examples:

Example 2.9.1. Plot the curve $y = f(x) = xe^{-x^2}$:
plot2d([x * exp(-x^2)], [x,-5,5], [plot_format, gnuplot]);

Similarly, we can plot the surface \( z = g(x,y) = e^{-\frac{1}{3}(x^2+y^2)} \cos(xy) \) by invoking the command:

plot3d(exp((-x^2-y^2)*0.3)*cos(x*y), [x,-3,3], [y,-3,3], [plot_format,gnuplot]);

2.9.1 Publication-Quality Graphics for \( \LaTeX \)

For users of the \( \LaTeX \) typesetting system, it is possible to generate format-specific graphs having the correct fonts and labels. Hence, publication-quality results can be obtained immediately. Consider the following:

Example 2.9.2. Generate a \( \LaTeX \)-figure for the curve \( y = f(x) = \sin x, x \in [-\pi, \pi] \) with axis labels \( \alpha \) and \( \sin(\alpha) \):

plot2d([sin(x)], [x,-%pi,%pi], [y,-1,1], [gnuplot_term, latex], [gnuplot_out_file, " .. path.. graph.tex"], [xlabel, "$\alpha$"], [ylabel, "$\sin(\alpha)$"]);

This will create a “.tex”-formatted figure in the file “graph.tex”, which can be included in a \( \LaTeX \)-document. Check with \( \LaTeX \)-manuals for information on the required commands.

2.10 Numerical Data

Numerical data can be entered through assigning parameters and/or through conversion of existing results. The GUI menu “Numeric” contains interactive boxes for the most common conversion needs and precision settings. When dealing with numerically sensitive tasks or topics from numerical analysis, care should be taken so as to ensure that loss of accuracy can be avoided in computations. Maxima is not a numerical mathematics software and many of its algorithms are not optimized for numerical precision. We proceed with a quick glance at the most important numerical functions:

The variable \( \texttt{fpprec} \) is the number of significant digits (default equal to 16) for arithmetic on \texttt{bigfloat} numbers. \( \texttt{fpprec} \) does not affect computations on ordinary floating point numbers.

\texttt{bigfloat(expr)}: Converts all numbers and functions of numbers in \( \texttt{expr} \) to \texttt{bigfloat} numbers. The number of significant digits in the resulting bigfloats is specified by the global variable \( \texttt{fpprec} \). When \( \texttt{float2bf} \) is false a warning message is printed when a floating point number is converted into a bigfloat number (since this may lead to loss of precision).

\texttt{float(expr)}: Converts integers, rational numbers and bigfloats in \( \texttt{expr} \) to floating point numbers. It is also an \texttt{evflag}, \texttt{float} causes non-integral rational numbers and bigfloat numbers to be converted to floating point.
Chapter 3

An Overview of Octave

Figure 3.1: The surface $z = g(x, y) = e^{-\frac{1}{3}(x^2+y^2)} \cos(xy)$ visualized with Octave/GNUplot.

3.1 Introduction

Octave is an interactive programming language specifically suited for vectorizable numerical calculations. It provides a high level interface to many standard libraries of numerical mathematics, e.g. LAPACK or BLAS. The syntax of Octave resembles that of Matlab. An Octave program usually runs unmodified on Matlab. Matlab, however, being commercial software, has a larger function set, and so the reverse does not always work, especially when the program makes use of specialized add-on toolboxes for Matlab.
3.1.1 Help!

The function \texttt{help} lists the names of all built-in functions and internal variables, whereas \texttt{help name} further explains the variable or function “name”.

\textbf{Example 3.1.1.} Consider a help call for the function that computes eigenvalues for a matrix:

octave:1> help eig

3.1.2 Input Conventions

All commands can be typed in at the prompt or read from a script. Scripts are plain text files with file suffix \texttt{.m}. They are imported by calling the file name without the suffix and behave as if their content was typed in line by line. ; separates several commands within a line. A command terminated by ; does not display its result on-screen. A comma , separates two commands without suppressing on-screen output. ... at the end of the line denotes that an expression continues into the next line. Comments are preceded by \%. Octave is case sensitive.

3.1.3 Variables and Standard Operations

\texttt{varname = expression} assigns the result of \texttt{expression} to \texttt{varname}. Octave has all the usual mathematical functions +, -, \*, /, \^, \texttt{sin}, \texttt{cos}, \texttt{exp}, \texttt{acos}, \texttt{abs} etc. The operators of elementary logic are: \texttt{<} smaller, \texttt{<=} smaller or equal, \texttt{\&} and, \texttt{>} greater, \texttt{>=} greater or equal, \texttt{=} equal, \texttt{\&\&} not equal and \texttt{\!} not.

When debugging user-defined objects, the following commands are useful: \texttt{whos} shows all user-defined variables and functions. \texttt{clear name} clears \texttt{name} from memory; if no name is given, all variables and functions will be cleared. \texttt{type name} displays information about the variable or function name on the screen.

\textbf{Example 3.1.2.} Various types of input and how to run a script file.

octave:1> x12 = 1/8, long_name = 'A String'
x12 = 0.12500
long_name = A String
octave:2> sqrt(-1)-i
ans = 0
octave:3> x = sqrt(2); sin(x)/x
ans = 0.69846

And here is a script \texttt{doless}, saved in a file named \texttt{doless.m}:

\begin{verbatim}
one = 1;
two = 2;
three = one + two;
\end{verbatim}
3.2. VECTOR AND MATRIX OPERATIONS

Calling the script:

```
octave:1> doless
octave:2> whos
*** local user variables:
  prot  type    rows  cols  name
    ====  =====  ====  ====  ====
   wd real scalar  1    1    three
   wd real scalar  1    1     one
   wd real scalar  1    1     two
```

3.2 Vector and Matrix Operations

Matrices and vectors are the most important building blocks for programming in Octave.

3.2.1 Vectors

**Example 3.2.1.** Row and column vectors: \( u = (1, 2, 3) \), \( v = (1, 2, 3)^T \):

\[
\begin{align*}
u & = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix} \\
v & = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}
\end{align*}
\]

Vectors with constant increment can be generated with the command “\( \text{start}(:\text{increment}):\text{end} \).”

```
octave:1> x = 3:6
x =
   3  4  5  6
octave:2> y = 0:.15:.7
y =
   0.00000  0.15000  0.30000  0.45000  0.60000
octave:3> z = pi:-pi/4:0
z =
   3.14159  2.35619  1.57080  0.78540  0.00000
```

3.2.2 Matrices

A matrix \( A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \) is generated like in the following example.

**Example 3.2.2.** \( A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \)

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

Matrices can also assembled from submatrices:
There are functions to create frequently used $m \times n$ matrices. If $m = n$, only one argument is necessary. The function \texttt{eye(m,n)} produces a matrix with ones on the main diagonal and zeros elsewhere. When $m = n$, the identity matrix is generated. An $m \times n$-matrix with only zeros is generated with the function \texttt{zeros(m,n)}, whereas \texttt{ones(m,n)} generates an $m \times n$ matrix with all entries equal to 1. The function \texttt{rand(m,n)} generates a random matrix whose entries are uniformly distributed in the interval $(0,1)$.

### 3.2.3 Basic Matrix Arithmetic

$+$, $-$, and $*$ denote matrix addition, subtraction, and multiplication. $A'$ transposes and conjugates $A$ and $A.'$ transposes $A$.

**Example 3.2.3.** Matrix operations

\begin{verbatim}
octave:1> A = [1 2; 3 4]; B = 2*ones(2,2);
octave:2> A+B, A-B, A*B
ans =
 3 4
 5 6
ans =
-1 0
 1 2
ans =
 6 6
 14 14
\end{verbatim}

### 3.2.4 Element-wise Operations

While $*$ refers to the usual matrix multiplication, $.*$ denotes element-wise multiplication. Similarly, $./$ and $.^*$ denote the element-wise division and power operators.

**Example 3.2.4.** Element-wise matrix operations

\begin{verbatim}
octave:1> A = [1 2; 3 4]; A.^2  % Element-wise power
ans =
 1 4
 9 16
\end{verbatim}
3.2. VECTOR AND MATRIX OPERATIONS

octave:2> A^2  % Proper matrix power: A^2 = A*A
ans =
    7 10
   15 22

3.2.5 Indexing and Slicing

\( v(k) \) is the \( k \)-th element of the row or column vector \( v \). \( A(k,1) \) is the matrix element \( a_{kl} \). \( v(m:n) \) is the “slice” of the vector \( v \) from its \( m \)-th to its \( n \)-th entry. \( A(k,:) \) is the \( k \)-th row of the matrix \( A \). \( A(:,l) \) is the \( l \)-th column of the matrix \( A \). There are also functions for querying dimensions of vectors and matrices; \( \text{length}(v) \) returns the number of elements of the vector \( v \). The call \([\text{Rows},\text{Columns}] = \text{size}(A)\) returns the number of rows and columns of the matrix \( A \). Matrices can also be reshaped; \( \text{reshape}(A,m,n) \) transforms \( A \) into an \( m \times n \) -matrix. \( \text{diag}(A) \) creates a column vector containing the diagonal elements \( a_{jj} \) of the matrix \( A \). \( \text{diag}(v) \) generates a matrix with the elements from the vector \( v \) on the diagonal, whereas \( \text{diag}(v,k) \) generates a matrix with the elements from the vector \( v \) on the \( k \)-th diagonal. Moreover,

\[
A(k,:) = rv \text{ assigns to the } k \text{-th row of } A \text{ the row vector } rv.
\]

\[
A(k,:) = [] \text{ deletes the } k \text{-th row of } A.
\]

\[
A(:,j) = cv \text{ assigns to the } j \text{-th column of } A \text{ the column vector } cv.
\]

\[
A(:,j) = [] \text{ deletes the } j \text{-th column of } A.
\]

Example 3.2.5. octave:1> A = [1 2 3; 4 5 6]; v = [7; 8];
octave:2> A(2,3) = v(2)
A =
    1  2  3
    4  5  6

octave:3> A(:,2) = v
A =
    1  7  3
    4  8  8

octave:4> A(1,1:2) = v'
A =
    7  8  3
    4  8  8

3.2.6 Solving Linear Systems of Equations

The command \( A\backslash b \) solves the equation \( Ax = b \).
3.2.7 Inverses, Decompositions and Eigenvalues

\( B = \text{inv}(A) \) computes the inverse of \( A \).

\[ [L,U,P] = \text{lu}(A) \] computes the LU-decomposition \( LU = PA \).

\[ [Q,R] = \text{qr}(A) \] computes the QR-decomposition \( QR = A \).

\( R = \text{chol}(A) \) computes the Cholesky-decomposition of \( A \).

\( S = \text{svd}(A) \) computes the singular values of \( A \).

\( H = \text{hess}(A) \) brings \( A \) into Hessenberg form.

\( E = \text{eig}(A) \) computes the eigenvalues of \( A \).

\[ [V,D] = \text{eig}(A) \] computes a diagonal matrix \( D \), containing the eigenvalues of \( A \) and a matrix \( V \) containing the corresponding eigenvectors such that \( AV =VD \).

\( \text{norm}(X,p) \) calculates the \( p \)-norm of vector \( X \). If \( X \) is a matrix, \( p \) can only take the values 1, 2 or \( \text{inf} (= \infty) \). The default is \( p=2 \).

\( \text{cond}(A) \) computes the condition number of \( A \) with respect to the 2-norm.

A lot of these commands support further options. They can be listed by typing \( \text{help funcname} \).

3.2.8 Testing for Zero Elements

\[ [i,j,v] = \text{find}(A) \] finds the indices of all nonzero elements of \( A \). The resulting vectors satisfy \( A_{(i,j)} = v \neq 0 \). \( \text{any}(v) \) returns 1 if the vector \( v \) contains nonzero elements. \( \text{any}(A) \) applies any to each of the columns of the matrix \( A \).

3.3 Control Structures

3.3.1 Functions

Functions are called with arguments, whereas scripts consist of a list of consecutive commands. Both of these are stored in a text file with suffix \( \text{.m} \). All variables within a function are local in contrast to those of a script, which are global to the workspace.

Example 3.3.1. A function \( f \), saved in the file named \( f.m \).

function \( y = f(x) \)
    \( y = \cos(x/2)+x; \)
end

Remark 3.3.1. In Octave, several functions can be defined in a single script file. Matlab on the other hand, strictly enforces one function per \( \text{.m} \) file, where the name of the function must match the name of the file. If compatibility with Matlab is important, this restriction should also be applied to programs written in Octave.

Example 3.3.2. A case with two function values. Consider the function \( \text{dolittle} \), saved under the file named \( \text{dolittle.m} \).
3.3. CONTROL STRUCTURES

function [out1,out2] = dolittle (x)
    out1 = x^2;
    out2 = out1*x;
end

Let us call the function:

octave:1> [x1,x2]=dolittle(2)
x1 = 4
x2 = 8

octave:2> whos
*** currently compiled functions:
prot type rows cols name
==== ==== ==== ==== ====
wd user function - - dolittle

*** local user variables:
prot type rows cols name
==== ==== ==== ==== ====
wd real scalar 1 1 x1
wd real scalar 1 1 x2

Obviously, the variables out1 and out2 are local to dolittle. Previously defined variables out1 or out2 would not have been affected by the function call.

3.3.2 Global Variables

The declaration for a global variable is global name. Consider a function foo in the file named foo.m:

    global N  % makes N a global variable; may be set in main file

function out = foo(arg1,arg2)
    global N  % makes local N refer to the global N
    <Computation>
end

If you change N within the function, it changes in the value of N everywhere.

3.3.3 Loops

The syntax of for- and while-loops is obvious in the following:

Example 3.3.3. Loop syntax

    for n = 1:10
        [x(n),y(n)]=dolittle(n);
    end

    while t<T
For-loop backward:
for n = 10:-1:1
...

### 3.3.4 Branching

**Example 3.3.4.** Conditional branching works as follows.

```octave
if x==0
    error('x is 0!');
else
    y = 1/x;
end
```

```octave
switch pnorm
    case 1
        sum(abs(v))
    case inf
        max(abs(v))
    otherwise
        sqrt(v'*v)
end
```

### 3.3.5 Functions of Functions

The function `eval(string)` evaluates `string` as Octave code. On the other hand, `feval(funcname,arg1,...)` is equivalent to calling `funcname` with arguments `arg1,...`.

**Example 3.3.5.** Approximate an integral by the midpoint rule:

\[
\int_a^b f(x) \, dx \approx \frac{b - a}{N} \sum_{j=0}^{N-1} f \left( a + \left( j + \frac{1}{2} \right) \frac{b - a}{N} \right).
\]

We define two functions, `gauss.m` and `mpr.m`:

```octave
function y = gauss(x)
    y = exp(-x.^2/2);
end
```

```octave
function S = mpr(fun,a,b,N)
    h = (b-a)/N;
    S = h*sum(feval(fun,[a+h/2:h:b]));
end
```
3.4. INPUT AND OUTPUT

Now the function gauss can be integrated by calling:

octave:1> mpr('gauss',0,5,500)

3.3.6 Efficiency Considerations

It should be noted that loops and function calls, especially through feval, have a very high computational overhead. Consequently, excessive use of these results in slowly executable code. Therefore, if possible, all operations should be vectorized.

Example 3.3.6. We are programming the midpoint rule from the previous section with a for-loop (the file name is mpr_long.m):

```octave
def function S = mpr_long(fun,a,b,N)
    h = (b-a)/N; S = 0;
    for k = 0:(N-1),
        S = S + feval(fun,a+h*(k+1/2));
    end
    S = h*S;
end
```

We verify that mpr and mpr_long yield the same result, and compare the execution times.

```
octave:1> t = cputime;
> Int1=mpr('gauss',0,5,500); t1=cputime-t;
octave:2> t = cputime;
> Int2=mpr_long('gauss',0,5,500); t2=cputime-t;
octave:3> Int1-Int2, t2/t1
ans = 0
ans = 45.250
```

3.4 Input and Output

**Save data** var1, var2 ... saves the values of variables var1 etc. into the file data. **Load data** reads the file data, restoring the values of the previously saved variables. fprintf(string[,var1,...]) resembles C syntax for formatting output, see man fprintf under Unix. format [long|short] enlarges or reduces the number of decimals displayed. Calling format without argument restores the default. The command pause suspends evaluation until a key is pressed.

**Example 3.4.1. Display formatting.**

```
octave:1> for k = .1:.2:.5,
    > fprintf('1/%g = %10.2e\n',k,1/k);
end
1/0.1 = 1.00e+01
1/0.3 = 3.33e+00
1/0.5 = 2.00e+00
```
3.5 Graphics

3.5.1 2D Graphics

plot(x,y[,fmt]) plots a line through the points \((x_j, y_j)\). With the string fmt you can select line style and color; see help plot for details. Furthermore, semilogx(x,y[,fmt]) works like plot with a logarithmic scale for the x-axis and semilogy(x,y[,fmt]) works like plot with a logarithmic scale for the y-axis. A plot with logarithmic scales on both axes is produced with the command loglog(x,y[,fmt]).

Method 3.5.1. Procedure for plotting a function \(y = f(x)\):

1. Generate a vector with the x-coordinates of the points to be plotted.
   \[x = x_{\text{min}} : \text{step}_x : x_{\text{max}};\]
2. Generate a vector containing the corresponding y-values by letting \(f\) act on the vector \(x\) in an element-wise fashion:
   \[y = f(x);\]
3. Important: Since \(f\) operates element-wise, care must be taken to use the operators .+, .-, .^ etc., instead of the usual +, - and ^!
4. Finally, the plot command is called.

plot(x,y)

To generate a coordinate grid, invoke:

plot(x,y)
grid

Example 3.5.2. Plotting a simple function with a coordinate grid.

octave:1> x = -10:.1:10;
octave:2> y = sin(x).*exp(-abs(x));
octave:3> plot(x,y)
octave:4> grid

3.5.2 3D Graphics

The command \([xx,yy] = \text{meshgrid}(x,y)\) generates the grid data for a 3D plot, i.e., two matrices \(xx\) whose rows are copies of \(x\) and \(yy\) whose columns are copies of \(y\). Applying \(\text{mesh}(x,y,z)\) after this plots the surface \(z = f(x,y)\) in three dimensions.

Example 3.5.3. A plot of the surface \(z = \sin \left(x^2 - y^2\right)\).

octave:1> x = -2:0.1:2;
octave:2> [xx,yy] = meshgrid(x,x);
octave:3> z = sin(xx.^2-yy.^2);
octave:4> mesh(x,x,z);
3.5. GRAPHICS

3.5.3 Commands for 2D and 3D Graphics

The command title(string) writes string as title for the graphics and xlabel(string) labels the x-axis with string. Similarly for other axes with the commands ylabel(string) and zlabel(string). To set axis limits for the plot, the command axis(v) can be used. The vector v has the form

\[ v = (x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}}, [z_{\text{min}}, z_{\text{max}}]) \]

hold [on|off] controls whether the next graphics output should or not clear the previous graphics, whereas clg clears the graphics window.

3.5.4 Publication-Quality Graphics for \LaTeX

For users of the \LaTeX typesetting system, it is possible to generate format-specific graphs having the correct fonts and labels. Hence, publication-quality results can be obtained immediately. Consider the following:

**Example 3.5.4.** Generate a \LaTeX-figure for the contour-plot \( z = g(x, y) = e^{-x^2-y^2} \), \((x, y) \in [0.1, 0.6] \times [0.1, 0.6] \) with axis labels x and y and title \( e^{-x^2-y^2} \):

octave:1> x_values = [0.10 : 0.005 : 0.60];
octave:2> y_values = [0.10 : 0.005 : 0.60];
octave:3> size(x_values)

\[ \text{ans} = \]

\[
\begin{array}{c}
\text{ans} = \\
\end{array}
\]

Figure 3.2: The graph \( y = f(x) = e^{-|x| \sin x} \).
Figure 3.3: The surface $z = f(x, y) = \sin(x^2 - y^2)$.

```octave
octave:4> arg = -ones(101,1)*(x_values.^2)-(y_values'.^2)*ones(1,101);
octave:5> data = exp(arg);
octave:6> contourf(x_values,y_values,data);
octave:7> axis square; colorbar;
octave:8> xlabel('$x$'); ylabel('$y$'); title('$\mathrm{e}^{-x^2-y^2}$');
octave:9> print('-dtex', '/home/aton/tfredman/Desktop/temp/contplot.tex');
```

This will create a “.tex”-formatted figure in the file “contplot.tex”, which can be included in a \LaTeX-document. Check with \LaTeX-manuals for information on the required commands.
Chapter 4

Numerical Computation

4.1 Preliminaries

GNU Octave is a high-level language, primarily intended for numerical computations. It provides a convenient interactive command line interface for solving linear and nonlinear problems numerically, and for performing other numerical experiments. It may also be used as a batch-oriented language for data processing.

GNU Octave is freely redistributable software. You may redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation.

4.1.1 Running Octave

On most systems, Octave is started with the shell command “octave”. Octave displays an initial message and then a prompt indicating it is ready to accept input. You can begin typing Octave commands immediately afterwards.

If you get into trouble, you can usually interrupt Octave by typing Control-C (written C-c for short). C-c gets its name from the fact that you type it by holding down CTRL and then pressing C. Doing this will normally return you to Octave's prompt.

To exit Octave, type quit or exit at the Octave prompt.

4.1.2 Editing What You Have Typed

At the Octave prompt, you can recall, edit, and reissue previous commands using Emacs- or vi-style editing commands. The default keybindings use Emacs-style commands. For example, to recall the previous command, press Control-p (written C-p for short). Doing this will normally bring back the previous line of input. C-n will bring up the next line of input, C-b will move the cursor backward on the line, C-f will move the cursor forward on the line, etc.
4.1.3 Startup Files

When Octave starts, it looks for commands to execute from the files in the following list. These files may contain any valid Octave commands, including function definitions.

```
octave-home /share/octave/site/m/startup/octaverc
```

where octave-home is the directory in which Octave is installed. This file is provided so that changes to the default Octave environment can be made globally for all users at your site for all versions of Octave you have installed. Care should be taken when making changes to this file since all users of Octave at your site will be affected. The default file may be overridden by the environment variable `OCTAVE_SITE_INITFILE`.

```
octave-home /share/octave/version /m/startup/octaverc
```

where version is the version number of Octave. This file is provided so that changes to the default Octave environment can be made globally for all users of a particular version of Octave. Care should be taken when making changes to this file since all users of Octave at your site will be affected. The default file may be overridden by the environment variable as above.

```
~/.octaverc
```

This file is used to make personal changes to the default Octave environment.

```
.octaverc
```

This file can be used to make changes to the default Octave environment for a particular project. Octave searches for this file in the current directory after it reads “`~/.octaverc`”. Any use of the `cd` command in the file will affect the directory where Octave searches for it. If you start Octave in your home directory, commands from the file “`~/octaverc`” will only be executed once.

A message will be displayed as each of the startup files is read if you invoke Octave with the `--verbose` option but without the `--silent` option.

4.1.4 Quitting Octave

Quitting is done with either of the commands:

```
exit (status )
quit (status )
```

Exit the current Octave session. If the optional integer value `status` is supplied, pass that value to the operating system as the Octave’s exit status. The default value is `zero`. The commands

```
atexit (fcn )
atexit (fcn, flag )
```
register a function to be called when Octave exits. For example,

```octave
function last_words ()
disp ("Bye bye");
endfunction
atexit ("last_words");
```

will print the message “Bye bye” when Octave exits. The additional argument `flag` will register or unregister `fcn` from the list of functions to be called when Octave exits. If `flag` is `true`, the function is registered, and if `flag` is `false`, it is unregistered. For example, after registering the function `last_words` above,

```octave
atexit ("last_words", false);
```

will remove the function from the list and Octave will not call `last_words` when it exits. Note that `atexit` only removes the first occurrence of a function from the list, so if a function was placed in the list multiple times with `atexit`, it must also be removed from the list multiple times.

### 4.1.5 Help and Documentation

Octave has an extensive help facility. The same documentation that is available in printed form is also available from the Octave prompt, because both forms of the documentation are created from the same input file.

In order to get good help you first need to know the name of the command that you want to use. This name of the function may not always be obvious, but a good place to start is to just type `help`. This will show you all the operators, reserved words, functions, built-in variables, and function files. An alternative is to search the documentation using the `lookfor` function. Once you know the name of the function you wish to use, you can get more help on the function by simply including the name as an argument to `help`. For example, `help plot` will display the help text for the `plot` function. Octave sends output that is too long to fit on one screen through a pager like `less` or `more`. Type a `RET` to advance one line, a `SPC` to advance one page, and `Q` to exit the pager.

The part of Octave’s help facility that allows you to read the complete text of the printed manual from within Octave normally uses a separate program called Info. When you invoke `info` you will be put into a menu driven program that contains the entire Octave manual.

### 4.2 Functions and Scripts

Complicated Octave programs can often be simplified by defining functions. Functions can be defined directly on the command line during interactive Octave sessions, or in external files, and can be called just like built-in functions.
4.2.1 Defining Functions

In its simplest form, the definition of a function named name looks like this:

```octave
function name
  body
endfunction
```

A valid function name is like a valid variable name: a sequence of letters, digits and underscores, not starting with a digit. Functions share the same pool of names as variables. The function body consists of Octave statements. It is the most important part of the definition, because it says what the function should actually do. For example, here is a function that, when executed, will ring the bell on your terminal (assuming that it is possible to do so):

```octave
function wakeup
  printf ("\a");
endfunction
```

The `printf` statement simply tells Octave to print the string “\a”. The special character “\a” stands for the alert character (ASCII 7). Once this function is defined, you can ask Octave to evaluate it by typing the name of the function. Normally, you will want to pass some information to the functions you define. The syntax for passing parameters to a function in Octave is

```octave
function name (arg-list )
  body
endfunction
```

where `arg-list` is a comma-separated list of the function’s arguments. When the function is called, the argument names are used to hold the argument values given in the call. The list of arguments may be empty, in which case this form is equivalent to the one shown above. To print a message along with ringing the bell, you might modify the `wakeup` to look like this:

```octave
function wakeup (message)
  printf ("\a%s\n", message);
endfunction
```

Calling this function using a statement like this

```octave
wakeup ("Rise and shine!");
```

will cause Octave to ring your terminal’s bell and print the message “Rise and shine!”, followed by a newline character (the “\n” in the first argument to the `printf` statement). In most cases, you will also want to get some information back from the functions you define. Here is the syntax for writing a function that returns a single value:

```octave
function ret-var = name (arg-list )
  body
endfunction
```
The symbol `ret-var` is the name of the variable that will hold the value to be returned by the function. This variable must be defined before the end of the function body in order for the function to return a value. Variables used in the body of a function are local to the function. Variables named in `arg-list` and `ret-var` are also local to the function. For example, here is a function that computes the average of the elements of a vector:

```octave
def function retval = avg (v)
    retval = sum (v) / length (v);
endfunction
```

If we had written `avg` like this instead,

```octave
def function retval = avg (v)
    if (isvector (v))
        retval = sum (v) / length (v);
    endif
endfunction
```

and then called the function with a matrix instead of a vector as the argument, Octave would have printed an error message like this:

`error: value on right hand side of assignment is undefined`

because the body of the if statement was never executed, and `retval` was never defined. To prevent obscure errors like this, it is a good idea to always make sure that the return variables will always have values, and to produce meaningful error messages when problems are encountered. For example, `avg` could have been written like this:

```octave
def function retval = avg (v)
    retval = 0;
    if (isvector (v))
        retval = sum (v) / length (v);
    else
        error ("avg: expecting vector argument");
    endif
endfunction
```

There is still one additional problem with this function. What if it is called without an argument? Without additional error checking, Octave will probably print an error message that won’t really help you track down the source of the error. To allow you to catch errors like this, Octave provides each function with an automatic variable called `nargin`. Each time a function is called, `nargin` is automatically initialized to the number of arguments that have actually been passed to the function. For example, we might rewrite the `avg` function like this:

```octave
def function retval = avg (v)
    if (isvector (v))
        retval = sum (v) / length (v);
    else
        error ("avg: expecting vector argument");
    endif
endfunction
```
function retval = avg (v)  
    retval = 0;  
    if (nargin != 1)  
        usage ("avg (vector)");  
    endif  
    if (isvector (v))  
        retval = sum (v) / length (v);  
    else  
        error ("avg: expecting vector argument");  
    endif  
endfunction

Although Octave does not automatically report an error if you call a function with more arguments than expected, doing so probably indicates that something is wrong. Octave also does not automatically report an error if a function is called with too few arguments, but any attempt to use a variable that has not been given a value will result in an error. To avoid such problems and to provide useful messages, we check for both possibilities and issue our own error message.

4.2.2 Multiple Return Values

Unlike many other computer languages, Octave allows you to define functions that return more than one value. The syntax for defining functions that return multiple values is

    function [ret-list] = name (arg-list)  
        body  
    endfunction

where name, arg-list, and body have the same meaning as before, and ret-list is a comma-separated list of variable names that will hold the values returned from the function. The list of return values must have at least one element. If ret-list has only one element, this form of the function statement is equivalent to the form described in the previous section.

Here is an example of a function that returns two values, the maximum element of a vector and the index of its first occurrence in the vector.

    function [max, idx] = vmax (v)  
        idx = 1;  
        max = v (idx);  
        for i = 2:length (v)  
            if (v (i) > max)  
                max = v (i);  
                idx = i;  
            endif  
        endfor  
    endfunction
In this particular case, the two values could have been returned as elements of a single array, but that is not always possible or convenient. The values to be returned may not have compatible dimensions, and it is often desirable to give the individual return values distinct names.

In addition to setting \texttt{nargin} each time a function is called, Octave also automatically initializes \texttt{nargout} to the number of values that are expected to be returned. This allows you to write functions that behave differently depending on the number of values that the user of the function has requested. The implicit assignment to the built-in variable \texttt{ans} does not figure in the count of output arguments, so the value of \texttt{nargout} may be zero.

The \texttt{svd} and \texttt{lu} (singular value decomposition and \textit{LU}-factorization for matrices) functions are examples of built-in functions that behave differently depending on the value of \texttt{nargout}.

It is possible to write functions that only set some return values. For example, calling the function

\begin{verbatim}
function [x, y, z] = f ()
x = 1;
z = 2;
endfunction
\end{verbatim}

as

\begin{verbatim}
[a, b, c] = f ()
\end{verbatim}

produces:

\begin{verbatim}
a = 1
b = []
c = 2
\end{verbatim}

along with a warning.

\subsection{Variable-length Argument Lists}

Sometimes the number of input arguments is not known when the function is defined. As an example think of a function that returns the smallest of all its input arguments. For example,

\begin{verbatim}
a = smallest (1, 2, 3);
b = smallest (1, 2, 3, 4);
\end{verbatim}

In this example both \texttt{a} and \texttt{b} would be 1. One way to write the smallest function is

\begin{verbatim}
function val = smallest (arg1, arg2, arg3, arg4, arg5)
body
endfunction
\end{verbatim}
and then use the value of `nargin` to determine which of the input arguments should be considered. The problem with this approach is that it can only handle a limited number of input arguments. If the special parameter name `varargin` appears at the end of a function parameter list it indicates that the function takes a variable number of input arguments. Using `varargin` the function looks like this

```matlab
function val = smallest (varargin)
    body
endfunction
```

In the function body the input arguments can be accessed through the variable `varargin`. This variable is a cell array containing all the input arguments. The smallest function can now be defined like this

```matlab
function val = smallest (varargin)
    val = min ([varargin{:}]);
endfunction
```

This implementation handles any number of input arguments, but it’s also a very simple solution to the problem. A slightly more complex example of `varargin` is a function `print_arguments` that prints all input arguments. Such a function can be defined like this

```matlab
function print_arguments (varargin)
    for i = 1:length (varargin)
        printf ("Input argument %d: ", i);
        disp (varargin{i});
    endfor
endfunction
```

### 4.2.4 Variable-length Return Lists

It is possible to return a variable number of output arguments from a function using a syntax similar to the one used with the special `varargin` parameter name. To let a function return a variable number of output arguments the special output parameter name `varargout` is used. As with `varargin`, `varargout` is a cell array that will contain the requested output arguments.

As an example the following function sets the first output argument to 1, the second to 2, and so on.

```matlab
function varargout = one_to_n ()
    for i = 1:nargout
        varargout{i} = i;
    endfor
endfunction
```

When called this function returns values like this.
[a, b, c] = one_to_n()
    a = 1
    b = 2
    c = 3

If varargin (varargout) does not appear as the last element of the input (output) parameter list, then it is not special, and is handled the same as any other parameter name.

4.2.5 Returning From a Function

The body of a user-defined function can contain a return statement. This statement returns control to the rest of the Octave program. It looks like this: return. Unlike the return statement in C, Octave’s return statement cannot be used to return a value from a function. Instead, you must assign values to the list of return variables that are part of the function statement. The return statement simply makes it easier to exit a function from a deeply nested loop or conditional statement. Here is an example of a function that checks to see if any elements of a vector are nonzero.

```octave
function retval = any_nonzero (v)
    retval = 0;
    for i = 1:length (v)
        if (v (i) != 0)
            retval = 1;
            return;
        endif
    endfor
    printf("no nonzero elements found\n");
endfunction
```

Note that this function could not have been written using the break statement to exit the loop once a nonzero value is found without adding extra logic to avoid printing the message if the vector does contain a nonzero element.

4.2.6 Function Files

Except for simple one-shot programs, it is not practical to have to define all the functions you need each time you need them. Instead, you will normally want to save them in a file so that you can easily edit them, and save them for use at a later time.

Octave does not require you to load function definitions from files before using them. You simply need to put the function definitions in a place where Octave can find them.

When Octave encounters an identifier that is undefined, it first looks for variables or functions that are already compiled and currently listed in its symbol table. If it fails to find a definition there, it searches a list of directories (the
path) for files ending in “.m” that have the same base name as the undefined identifier. Once Octave finds a file with a name that matches, the contents of the file are read. If it defines a single function, it is compiled and executed.

When Octave defines a function from a function file, it saves the full name of the file it read and the time stamp on the file. If the time stamp on the file changes, Octave may reload the file. When Octave is running interactively, time stamp checking normally happens at most once each time Octave prints the prompt. Searching for new function definitions also occurs if the current working directory changes.

Checking the time stamp allows you to edit the definition of a function while Octave is running, and automatically use the new function definition without having to restart your Octave session.

4.2.7 Subfunctions

A function file may contain secondary functions called subfunctions. These secondary func- tions are only visible to the other functions in the same function file. For example, a file “f.m” containing

```octave
function f ()
    printf (“in f, calling g\n”);
    g ()
endfunction
function g ()
    printf (“in g, calling h\n”);
    h ()
endfunction
function h ()
    printf (“in h\n”)
endfunction
```

defines a main function f and two subfunctions. The subfunctions g and h may only be called from the main function f or from the other subfunctions, but not from outside the file “f.m”.

4.2.8 Private Functions

In many cases one function needs to access one or more helper functions. If the helper function is limited to the scope of a single function, then subfunctions as discussed above might be used. However, if a single helper function is used by more than one function, then this is no longer possible. In this case the helper functions might be placed in a subdirectory, called “private”, of the directory in which the functions needing access to this helper function are found.

As a simple example, consider a function `func1`, that calls a helper function `func2` to do much of the work. For example

```octave
function y = func1 (x)
```

4.2. FUNCTIONS AND SCRIPTS

\[ y = \text{func2}(x); \]
\text{endfunction}

Then if the path to \texttt{func1} is \texttt{<directory>/func1.m}, and if \texttt{func2} is found in the directory \texttt{<directory>/private/func2.m}, then \texttt{func2} is only available for use of the functions, like \texttt{func1}, that are found in \texttt{<directory>}.

4.2.9 Script Files

A script file is a file containing (almost) any sequence of Octave commands. It is read and evaluated just as if you had typed each command at the Octave prompt, and provides a convenient way to perform a sequence of commands that do not logically belong inside a function.

Unlike a function file, a script file must not begin with the keyword function. If it does, Octave will assume that it is a function file, and that it defines a single function that should be evaluated as soon as it is defined.

A script file also differs from a function file in that the variables named in a script file are not local variables, but are in the same scope as the other variables that are visible on the command line.

Even though a script file may not begin with the function keyword, it is possible to define more than one function in a single script file and load (but not execute) all of them at once. To do this, the first token in the file (ignoring comments and other white space) must be something other than function. If you have no other statements to evaluate, you can use a statement that has no effect, like this:

```octave
# Prevent Octave from thinking that this
# is a function file:
1;
# Define function one:
function one ()
    ...
endfunction
```

To have Octave read and compile these functions into an internal form, you need to make sure that the file is in the load path of Octave (accessible through the path function), then simply type the base name of the file that contains the commands. (Octave uses the same rules to search for script files as it does to search for function files.)

If the first token in a file (ignoring comments) is function, Octave will compile the function and try to execute it, printing a message warning about any non-whitespace characters that appear after the function definition.

Note that Octave does not try to look up the definition of any identifier until it needs to evaluate it. This means that Octave will compile the following statements if they appear in a script file, or are typed at the command line,

```octave
# not a function file:
1;
```
function foo ()
    do_something ();
endfunction
function do_something ()
    do_something_else ();
endfunction

even though the function do_something is not defined before it is referenced in the function foo. This is not an error because Octave does not need to resolve all symbols that are referenced by a function until the function is actually evaluated.

Since Octave doesn’t look for definitions until they are needed, the following code will always print “bar = 3” whether it is typed directly on the command line, read from a script file, or is part of a function body, even if there is a function or script file called “bar.m” in Octave’s path.

    eval ("bar = 3");
    bar

Code like this appearing within a function body could fool Octave if definitions were resolved as the function was being compiled. It would be virtually impossible to make Octave clever enough to evaluate this code in a consistent fashion.

4.2.10 Function Handles, Inline Functions, and Anonymous Functions

It can be very convenient store a function in a variable so that it can be passed to a different function. For example, a function that performs numerical minimization needs access to the function that should be minimized.

Function Handles

A function handle is a pointer to another function and is defined with the syntax @function-name. For example, f = @sin; Creates a function handle called f that refers to the function sin.

    Function handles are used to call other functions indirectly, or to pass a function as an argument to another function like quad or fsolve. For example

    f = @sin;
    quad (f, 0, pi)

You may use feval to call a function using function handle, or simply write the name of the function handle followed by an argument list. If there are no arguments, you must use an empty argument list “()”. For example
Anonymous Functions

Anonymous functions are defined using the syntax

\[ \text{@(argument-list ) expression} \]

Any variables that are not found in the argument list are inherited from the enclosing scope. Anonymous functions are useful for creating simple unnamed functions from expressions or for wrapping calls to other functions to adapt them for use by functions like `quad`. For example,

\[ f = \text{@(x) } x.^2; \]
\[ \text{quad } (f, 0, 10) \]

creates a simple unnamed function from the expression \( x.^2 \) and passes it to `quad`,

\[ \text{quad } (\text{@(x) sin } (x), 0, \pi) \]

wraps another function, and

\[ a = 1; \]
\[ b = 2; \]
\[ \text{quad } (\text{@(x) betainc } (x, a, b), 0, 0.4) \]

adapts a function with several parameters to the form required by `quad`. In this example, the values of \( a \) and \( b \) that are passed to `betainc` are inherited from the current environment.

Inline Functions

An inline function is created from a string containing the function body using the `inline` function. The following code defines the function \( f(x) = x^2 + 2 \).

\[ f = \text{inline}("x^2 + 2"); \]

After this it is possible to evaluate \( f(x) \) at any \( x \) by writing \( f(x) \).

4.3 Equations and Optimization

4.3.1 Nonlinear Equations

Octave can solve sets of nonlinear equations of the form

\[ f(x) = 0 \]
using the function \texttt{fsolve}, which is based on the Minpack subroutine \texttt{hybrd}.
This is an iterative technique so a starting point will have to be provided. This also has the consequence that convergence is not guaranteed even if a solution exists.

\textbf{Example 4.3.1.} Consider the set of equations

\[-2x^2 + 3xy + 4\sin(y) - 6 = 0,\]
\[3x^2 - 2xy^2 + 3\cos(y) + 4 = 0.\]

We first need to write a function to compute the value of the given function. Here:

function \texttt{y = f (x)}
\[y(1) = -2*x(1)^2 + 3*x(1)*x(2) + 4*\sin(x(2)) - 6;\]
\[y(2) = 3*x(1)^2 - 2*x(1)*x(2)^2 + 3*\cos(x(1)) + 4;\]
endfunction

Then, call \texttt{fsolve} with a specified initial condition to find the roots of the system of equations. For example, given the function \texttt{f} defined above,

\[\texttt{[x, fval, info] = fsolve (@f, [1; 2])}\]

results in the solution

\[x =\]
\[0.57983\]
\[2.54621\]

\[fval =\]
\[-5.7184e-10\]
\[5.5460e-10\]

\[\texttt{info} = 1\]

A value of \texttt{info = 1} indicates that the solution has converged. The function \texttt{perror} may be used to print English messages corresponding to the numeric error codes. For example,

\[\texttt{perror ("fsolve", 1)}\]

\texttt{solution converged to requested tolerance}\n
When no Jacobian is supplied (as in the example above) it is approximated numerically. This requires more function evaluations, and hence is less efficient.

In the example above we could compute the Jacobian analytically as

\[
\begin{bmatrix}
\frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} \\
\frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2}
\end{bmatrix} =
\begin{bmatrix}
3x_2 - 4x_1 & 4\cos(x_2) + 3x_1 \\
-2x_2^2 - 3\sin(x_1) + 6x_1 & -4x_1x_2
\end{bmatrix}
\]

The Jacobian can then be used with the following script and call to \texttt{fsolve}:
4.3. EQUATIONS AND OPTIMIZATION

# fsolvetest.m

function [y,J] = f(x)
    y = zeros(size(x));
    y(1) = -2*x(1)^2 + 3*x(1)*x(2) + 4*sin(x(2)) - 6;
    y(2) = 3*x(1)^2 - 2*x(1)*x(2)^2 + 3*cos(x(1)) + 4;
    if (nargout == 2)
        J(1,1) = 3*x(2) - 4*x(1);
        J(1,2) = 4*cos(x(2)) + 3*x(1);
        J(2,1) = -2*x(2)^2 - 3*sin(x(1)) + 6*x(1);
        J(2,2) = -4*x(1)*x(2);
    end
endfunction

[x, fval, info] = fsolve (@f, [0; 0],optimset("Jacobian","on"))

which gives the same solution as before.

4.3.2 Optimization

Octave comes with support for solving various kinds of optimization problems. Specifically Octave can solve problems in Linear Programming, Quadratic Programming, Nonlinear Programming, and Linear Least Squares Minimization.

Linear Programming

Octave can solve Linear Programming problems using the glpk function. That is, Octave can solve
\[
\min_x c^T x ,
\]
subject to the linear constraints \(Ax = b, x \geq 0\). The glpk function also supports variations of this problem.

Quadratic Programming

Octave can also solve Quadratic Programming problems through the function qp, this is
\[
\min_x \frac{1}{2} x^T H x + x^T q ,
\]
subject to \(Ax = b, lb \leq x \leq ub, A_l b \leq A \leq A_u b\).

Nonlinear Programming

Octave can also perform general nonlinear minimization using the successive quadratic programming solver sqp. The problem formulation is
\[
\min_x \phi(x) ,
\]
subject to

\[ g(x) = 0, \quad h(x) \geq 0, \quad lb \leq x \leq ub, \]

where \( g(x) \) is the gradient and \( h(x) \) is the hessian of \( \phi \), respectively.

**Example 4.3.2.** Consider the following:

```octave
function r = g(x)
    r = [ sumsq(x)-10;
         x(2)*x(3)-5*x(4)*x(5);
         x(1)^3+x(2)^3+1 ];
endfunction

function obj = phi(x)
    obj = exp(prod(x)) - 0.5*(x(1)^3+x(2)^3+1)^2;
endfunction

x0 = [-1.8; 1.7; 1.9; -0.8; -0.8];
[x, obj, info, iter, nf, lambda] = sqp(x0, @phi, @g, []);
```

\[ x = -1.71714 \]
\[ 1.59571 \]
\[ 1.82725 \]
\[ -0.76364 \]
\[ -0.76364 \]
\[ obj = 0.053950 \]
\[ info = 101 \]
\[ iter = 8 \]
\[ nf = 10 \]
\[ lambda = -0.0401627 \]
\[ 0.0379578 \]
\[ -0.0052227 \]

**Linear Least Squares Programming**

Octave also supports linear least squares minimization. That is, Octave can find the parameter matrix \( b \) such that the model \( y = xb \) fits data \((x, y)\) as well as possible, assuming zero-mean Gaussian noise. If the noise is assumed to be isotropic the problem can be solved using the "\" or "/" operators, or the `ols` function. In the general case where the noise is assumed to be anisotropic the `gls` is needed.
4.4 Boundary/Initial Value Problems

4.4.1 Integrating Differential Equations

Octave has built-in functions for solving nonlinear differential equations of the form

\[ \frac{dx}{dt} = f(x, t), \quad x(t_0) = x_0. \]

For Octave to integrate equations of this form, you must first provide a definition of the function \( f(x, t) \). This is straightforward, and may be accomplished by entering the function body directly on the command line. For example, the following commands define the right-hand side function for an interesting pair of nonlinear differential equations. Note that while you are entering a function, Octave responds with a different prompt, to indicate that it is waiting for you to complete your input.

**Example 4.4.1. A system of ordinary differential equations.**

```octave
octave:1> function xdot = f(x, t)
> r = 0.25;
> k = 1.4;
> a = 1.5;
> b = 0.16;
> c = 0.9;
> d = 0.8;
> xdot(1) = r*x(1)*(1 - x(1)/k) - a*x(1)*x(2)/(1 + b*x(1));
> xdot(2) = c*a*x(1)*x(2)/(1 + b*x(1)) - d*x(2);
> endfunction
```

Given the initial condition

```octave
octave:2> x0 = [1; 2];
```

and the set of output times as a column vector (note that the first output time corresponds to the initial condition given above)

```octave
octave:3> t = linspace (0, 50, 200)';
```

it is easy to integrate the set of differential equations:

```octave
octave:4> x = lsode ("f", x0, t);
```

The function `lsode` uses the Livermore Solver for Ordinary Differential Equations.\(^1\)

To display the solution graphically, use the command

---

\(^1\) Described in A. C. Hindmarsh, ODEPACK, a Systematized Collection of ODE Solvers, Scientific Computing, R. S. Stepleman et al. (Eds.), North-Holland, Amsterdam, 1983, pages 55-64.
octave:1> plot (t, x)

If you are using a graphical user interface, Octave will automatically create a separate window to display the plot.

To save a plot once it has been displayed on the screen, use the print command. For example,

print -deps foo.eps

will create a file called “foo.eps” that contains a rendering of the current plot in Encapsulated PostScript format. The command help print

explains more options for the print command and provides a list of additional output file formats.

4.5 Partial Differential Equations

To date (October 2011) there does not seem to be a stable implementation of a program package for partial differential equation in Octave. Some attempts to port the routine pdepe from Matlab to Octave have been made, however, these appear to be problematic since pdepe contains calls to routines in Matlab not available in Octave. Hence, these have to be substituted with suitable Octave alternatives. Perhaps the best approach in Octave for PDE is simply to rewrite these by hand as systems of ODE’s using some established method (lines, finite difference, finite element) and solve the resulting ODE-system with code available in Octave.

4.6 Integration

Octave comes with several built-in functions for computing the integral of a function numerically. These functions all solve one-dimensional integration problems.

4.6.1 Functions of One Variable

Octave supports three different algorithms for computing the integral

\[ \int_a^b f(x) \, dx \]

of a function \( f(x) \) over the interval for \( x \) from \( a \) to \( b \). These are:

- quad Numerical integration based on Gaussian quadrature.
- quad1 Numerical integration using an adaptive Lobatto rule.
- quadgk Numerical integration using an adaptive Gauss-Konrod rule.
- quadav Numerical integration using an adaptive vectorized Simpson’s rule.
- trapz Numerical integration using the trapezoidal method.
Besides these functions Octave also allows you to perform cumulative numerical integration using the trapezoidal method through the \texttt{cumtrapz} function.

**Example 4.6.1.** We consider the integral

\[ \int_a^b f(x) \, dx = \int_0^3 x\sqrt{|1-x|}\sin \frac{1}{x} \, dx. \]

This is a fairly difficult integration (plot the function over the range of integration to see why). The first step is to define the function:

```octave
def function y = f (x)
    y = x .* sin (1 ./ x) .* sqrt (abs (1 - x));
endfunction
```

Note the use of the “dot” forms of the operators. This is not necessary for the call to \texttt{quad}, but it makes it much easier to generate a set of points for plotting (because it makes it possible to call the function with a vector argument to produce a vector result). Then we simply call \texttt{quad}:

```octave
[v, ier, nfun, err] = quad ("f", 0, 3)
```

Although \texttt{quad} returns a nonzero value for \texttt{ier}, the result is reasonably accurate (to see why, examine what happens to the result if you move the lower bound to 0.1, then 0.01, then 0.001, etc.).

Another method for integration is implemented in the routine

```octave
[r, amat, bmat, q ] = colloc (n, "left", "right")
```

colloct computes derivative and integral weight matrices for orthogonal collocation using the subroutines given in J. Villadsen and M. L. Michelsen, *Solution of Differential Equation Models by Polynomial Approximation*.

Here is an example of using \texttt{colloc} to generate weight matrices for solving a boundary value problem for the second order differential equation

\[ u'' - \alpha u' = 0 \quad u(0) = 0 \quad u(1) = 1. \]

First, we can generate the weight matrices for \( n \) points (including the endpoints of the interval), and incorporate the boundary conditions in the right hand side (for a specific value of \( \alpha \)).

```octave
n = 7;
alpha = 0.1;
[r, a, b] = colloc (n-2, "left", "right");
at = a(2:n-1,2:n-1);
bt = b(2:n-1,2:n-1);
rhs = alpha * b(2:n-1,n) - a(2:n-1,n);
```
Then the solution at the roots $r$ is

$$ u = \begin{bmatrix} 0; (at - alpha \ast bt) \backslash rhs; 1 \\ 0.00; 0.004; 0.01 0.00; 0.12; 0.62; 1.00 \end{bmatrix} $$

### 4.6.2 Functions of Multiple Variables

Octave does not have built-in functions for computing the integral of functions of multiple variables directly. It is, however, possible to compute the integral of a function of multiple variables using the functions for one-dimensional integrals.

To illustrate how the integration can be performed, we will integrate the function

$$ f(x, y) = \sqrt{xy} \sin(\pi xy) $$

for $x$ and $y$ between 0 and 1.

The first approach creates a function that integrates $f$ with respect to $x$, and then integrates that function with respect to $y$. Since \texttt{quad} is written in Fortran it cannot be called recursively. This means that \texttt{quad} cannot integrate a function that calls \texttt{quad}, and hence cannot be used to perform the double integration. It is however possible with \texttt{quadl}, which is what the following code does.

```matlab
function I = g(y)
    I = ones(1, length(y));
    for i = 1:length(y)
        f = @(x) sin(pi.*x.*y(i)).*sqrt(x.*y(i));
        I(i) = quadl(f, 0, 1);
    endfor
endfunction
I = quadl("g", 0, 1)
0.30022
```

The above process can be simplified with the \texttt{dblquad} and \texttt{triplequad} functions for integrals over two and three variables. For example

$$ I = \texttt{dblquad} (\emptyset(x, y) \sin(pi.*x.*y).*sqrt(x.*y), 0, 1, 0, 1) $$

0.30022

The above mentioned approach works but is fairly slow, and that problem increases exponentially with the dimensionality the problem. Another possible solution is to use Orthogonal Collocation as described in the previous section. The integral of a function $f(x, y)$ for $x$ and $y$ between 0 and 1 can be approximated using $n$ points by

$$ \int_0^1 \int_0^1 f(x, y) \, dx \, dy \approx \sum_{i=1}^{n} \sum_{j=1}^{n} q_i q_j f(r_i, r_j), $$

where $(r_i)_{i=1}^{n}$ and $(q_i)_{i=1}^{n}$ are returned in vectors from \texttt{colloc(n)}. The generalization to more than two variables is straightforward. The following code computes the studied integral using $n = 7$ points.
4.7. INTERPOLATION

\[ f = @(x, y) \sin(\pi x \cdot y') \cdot \sqrt{x \cdot y'}; \]
\[ n = 7; \]
\[ [t, A, B, q] = \text{colloc}(n); \]
\[ I = q' \cdot f(t, t) \cdot q; \]
\[ 0.30022 \]

It should be noted that the number of points determines the quality of the approximation. If the integration needs to be performed between \( a \) and \( b \) instead of 0 and 1, a change of variables is needed.

### 4.7 Interpolation

#### 4.7.1 One-Dimensional Interpolation

Octave supports several methods for one-dimensional interpolation, as well as polynomial interpolation and interpolation on scattered data.

The routine \texttt{interp1} can be invoked as below:

\[ yi = \text{interp1} (x, y, xi) \]
\[ yi = \text{interp1} (y, xi) \]
\[ yi = \text{interp1} (... , method) \]
\[ yi = \text{interp1} (... , \text{extrap}) \]
\[ pp = \text{interp1} (... , "pp") \]

Interpolation is done to determine the value of \( yi \) at the points, \( xi \). If not specified, \( x \) is taken to be the indices of \( y \). If \( y \) is a matrix or an N-dimensional array, the interpolation is performed on each column of \( y \).

Method is one of:

- “nearest”: Return the nearest neighbor.
- “linear”: Linear interpolation from nearest neighbors
- “pchip”: Piecewise cubic Hermite interpolating polynomial
- “cubic”: Cubic interpolation (same as pchip)
- “spline”: Cubic spline interpolation?smooth first and second derivatives throughout the curve

Appending “*” to the start of the above method forces \texttt{interp1} to assume that \( x \) is uniformly spaced, and only \( x(1) \) and \( x(2) \) are referenced. This is a faster scheme. The default method is “linear”.

If \texttt{extrap} is the string “\texttt{extrap}”, then values are extrapolated beyond the endpoints. If \texttt{extrap} is a number, values beyond the endpoints are replaced by that number. If \texttt{extrap} is missing, “NA” will be assumed.

If the string argument “pp” is specified, then \( xi \) should not be supplied and \texttt{interp1} returns the piecewise polynomial that can later be used with \texttt{ppval} to evaluate the interpolation. There is an equivalence, such that:

\[ \text{ppval} (\text{interp1} (x, y, method, "pp"), xi) == \text{interp1} (x, y, xi, method, "extrap") \]
Duplicate points in $x$ specify a discontinuous interpolant. There may be at most two consecutive points with the same value. If $x$ is increasing, the default discontinuous interpolant is right-continuous. If $x$ is decreasing, the default discontinuous interpolant is left-continuous. The continuity condition of the interpolant may be specified by using the options, "-left" or "-right", to select a left-continuous or right-continuous interpolant, respectively. Discontinuous interpolation is only allowed for "nearest" and "linear" methods; in all other cases, the $x$-values must be unique.

An example of the use of `interp1` is

```matlab
xf = [0:0.05:10];
yf = sin (2*pi*xf/5);
 xp = [0:10];
 yp = sin (2*pi*xp/5);
 lin = interp1 (xp, yp, xf);
spl = interp1 (xp, yp, xf, "spline");
cub = interp1 (xp, yp, xf, "cubic");
near = interp1 (xp, yp, xf, "nearest");
plot (xf, yf, "r", xf, lin, "g", xf, spl, "b",
     xf, cub, "c", xf, near, "m", xp, yp, "r");
legend ("original", "linear", "spline", "cubic", "nearest");
```

See also: `interpft`. There are some important differences between the various interpolation methods. The "spline" method enforces that both the first and second derivatives of the interpolated values have a continuous derivative, whereas the other methods do not. This means that the results of the "spline" method are generally smoother. If the function to be interpolated is in fact smooth, then "spline" will give excellent results. However, if the function to be evaluated is in some manner discontinuous, then "pchip" interpolation might give better results. This can be demonstrated by the code

```matlab
t = -2:2;
dt = 1;
ti = -2:0.025:2;
dti = 0.025;
y = sign (t);
ys = interp1 (t,y,ti,"spline");
yp = interp1 (t,y,ti,"pchip");
d dys = diff (diff (ys)./dti)./ dti;
ddy p = diff (diff (yp)./dti)./ dti;
figure (1);
plot (ti,ys,"r-", ti,yp,"g-");
legend ("spline", "pchip", 4);
figure (2);
plot (ti,dys,"r+", ti,ddy p,"g");
legend ("spline", "pchip");
```

4.7. INTERPOLATION

4.7.2 $n$-Dimensional Interpolation

The function calls

\[ vi = \text{interpn} (x_1, x_2, ..., v, y_1, y_2, ...) \]
\[ vi = \text{interpn} (v, y_1, y_2, ...) \]
\[ vi = \text{interpn} (v, m) \]
\[ vi = \text{interpn} (v) \]
\[ vi = \text{interpn} (... \), method \]
\[ vi = \text{interpn} (... \), method, extrapval \]

perform $n$-dimensional interpolation, where $n$ is at least two. Each element of the $n$-dimensional array $v$ represents a value at a location given by the parameters $x_1, x_2, ..., x_n$. The parameters $x_1, x_2, ..., x_n$ are either $n$-dimensional arrays of the same size as the array $v$ in the “ndgrid” format or vectors. The parameters $y_1$, etc. have a similar format to $x_1$, etc., and they represent the points at which the array $v_i$ is interpolated.

If $x_1, ..., x_n$ are omitted, they are assumed to be $x_1 \ 1: \ \text{size} (v, 1)=, \ etc.$ If $m$ is specified, then the interpolation adds a point half way between each of the interpolation points. This process is performed $m$ times. If only $v$ is specified, then $m$ is assumed to be 1. The various methods are similar to those above for one-dimensional interpolation.

A significant difference between $\text{interpn}$ and the other two multi-dimensional interpolation functions $\text{interp2}$ and $\text{interp3}$ is the fashion in which the dimensions are treated. For $\text{interp2}$ and $\text{interp3}$, the $y$-axis is considered to be the columns of the matrix, whereas the $x$-axis corresponds to the rows of the array. As Octave indexes arrays in column major order, the first dimension of any array is the columns, and so $\text{interpn}$ effectively reverses the $x$- and $y$-dimensions. Consider the example,

\begin{verbatim}
 x = y = z = -1:1;
 f = @(x,y,z) x.^2 - y - z.^2;
 [xx, yy, zz] = meshgrid (x, y, z);
 v = f (xx,yy,zz);
 xi = yi = zi = -1:0.1:1;
 [xxi, yyi, zzi] = meshgrid (xi, yi, zi);
 vi = interp3 (x, y, z, v, xxi, yyi, zzi, "spline");
 [xxi, yyi, zzi] = ndgrid (xi, yi, zi);
 vi2 = interpn (x, y, z, v, xxi, yyi, zzi, "spline");
 mesh (zi, yi, squeeze (vi2(1,:,:)));
\end{verbatim}

where $v_i$ and $v_i2$ are identical. The reversal of the dimensions is treated in the $\text{meshgrid}$ and $\text{ndgrid}$ functions, respectively.
Chapter 5

Exercises

5.1  wxMaxima

5.1.1  Inverse Functions

Exercise 5.1.1. Consider the function \( f(x) = \frac{x+2}{3x-1} \), defined for \( x \in \mathbb{R} \setminus \{ \frac{1}{3} \} \). Study the monotonicity of the function on its set of definition. Compute the inverse function \( f^{-1}(x) \). Study the composition \( f(f(f(f(f(\cdots(x))))))) \), i.e., the function composed \( n \) times with itself.

Solution: We find \( f' \) and show that the derivative always is negative. We solve the inverse function and observe that \( f^{-1}(x) = f(x) \). Therefore the composition is equal to the identity if \( n \) is even and \( f \) otherwise.

\[
\begin{align*}
f(x) &:= (x+2)/(3x-1); \\
fp(x) &:= \text{diff}(f(x),x); \\
factor(\text{ratsimp}(\text{diff}(f(x),x))); \\
f(f(f(f(x)))) &\text{ratsimp}(); \\
f(f(f(x))) &\text{ratsimp}(); \\
\end{align*}
\]

Exercise 5.1.2. A study of the mass balance of water for a tank that is drained at the rate \( y(t) \) as a function of time \( t \) resulted in the following dimensionless relationship:

\[
t = y_0 - y + \ln \frac{1 - y_0}{1 - y},
\]

where \( y(0) = y_0 \) is the outflow of water at time \( t = 0 \). Unfortunately, this equation is implicit for \( y(t) \), however, it can be considered an inverse function \( t = f^{-1}(y) \). Discuss the existence of \( f \) and the possibility of deriving an explicit equation \( y = f(t) \).

Solution: It turns out that \( t = g(y) \) has an inverse function, as \( g' \) is positive for the physically feasible case \( 0 < y < 1 \) and negative otherwise. Hence,
CHAPTER 5. EXERCISES

If exists. However, an explicit relation \( y = f(t) \) cannot be obtained; Maxima fails to implement the solution and we get a relation with \( y \) on both sides.

\[
g(y) := y_0 - y + \log\left((1-y_0)/(1-y)\right);
gp(y) := \text{diff}(g(y),y);
\]

Exercise 5.1.3. Consider the function \( f(x) = \ln(1 + x^2) + x^3 + 7 \). Compute the inverse function \( f^{-1}(x) \) and the derivative \( (f^{-1})'(7) \).

5.1.2 Limits

Exercise 5.1.4. In the special theory of relativity, the kinetic energy of a particle having mass \( m \) and velocity \( v \) is

\[
T(v,c) = \frac{mc^2}{\sqrt{1 - (v/c)^2}} - mc^2,
\]

where \( c \) is the speed of light. On the other hand, in classical mechanics, the particle would have kinetic energy \( K(v) = \frac{1}{2}mv^2 \). Compute the following limits

a) \( \lim_{v \to 0} \frac{T(v,c)}{K(v)} \).

b) \( \lim_{c \to \infty} \frac{T(v,c)}{K(v)} \).

Solution:

\[
# \text{case (a)}
T(v,c) := (m*c^2) * (1/sqrt(1-(v/c)^2) - 1);
K(v) := (1/2) * m * v^2;
\text{limit}(T(v,c) / K(v), v, 0);
# \text{case (b)}
\text{limit}(T(v,c) / K(v), c, \text{inf});
\]

Exercise 5.1.5. Consider the function \( f(x) = \left\lvert x \right\rvert - x \), \( x \neq 0 \). Study the one-sided limits at the origin and conclude whether the limit \( \lim_{x \to 0} f(x) \) exists.

5.1.3 Differentiation

Exercise 5.1.6. An electrical current passing through a circular coil with radius \( r \) exerts a force

\[
F(x) = \frac{kx}{(x^2 + r^2)^{3/2}}
\]
on a small magnet at the distance \( x \) above the center of the coil. \( k \) is a constant of proportionality. Show that \( F \) has a maximum for \( x = \frac{r}{\sqrt{2}} \).

Solution: \( F \) is continuous and differentiable for every \( x \in \mathbb{R} \) and \( \lim_{x \to \pm \infty} F(x) = 0 \). Thus, a maximum must be located in the critical points \( x_{\pm} = \pm \frac{r}{\sqrt{2}} \). One of these is the location of a minimum and the other that of a maximum. This is seen from the observation \( F''(x_+) = -F''(x_-) \).
5.1. WXMAXIMA

\[ F(x) := \frac{k \cdot x}{(x^2 + r^2)^{5/2}}; \]
define(Fp(x), diff(F(x), x));

\[ \text{sol} : \text{solve}([Fp(x) = 0], [x]); \]
Fpp(sol);
ratsimp(ratsimp(%));

**Exercise 5.1.7.** Compute \( y' \), when
(a) \( xy^2 + \sin (y) = x^3 + 1. \)
(b) \( y = (\cos (x))^{\ln x} \)
(c) \( \frac{\sqrt{x^3 + \sin (x)}}{\sqrt{x + 2x}} \)

**Exercise 5.1.8.** Determine the value of \( t \) that gives the maximum of the integral\(^1\)
\[ \int_t^{t+3} x(5 - x) \, dx. \]

**5.1.4 Integration**

**Exercise 5.1.9.** Compute the integral
\[ \int \frac{x^5 + x^4 - 8}{x^3 - 4x} \, dx. \]

Solution:
\[ f(x) := \frac{(x^5 + x^4 - 8)}{(x^3 - 4x)}; \]
integrate(f(x), x);
ratsimp(%);

**Exercise 5.1.10.** The Mercator projection is a cylindrical map projection presented by the Flemish geographer and cartographer Gerardus Mercator in 1569. It became the standard map projection for nautical purposes because of its ability to represent lines of constant course, known as rhumb lines or loxodromes, as straight segments which conserve the angles with the meridians. Mathematically, it can be represented in the form
\[ x(\lambda) = \lambda, \]
\[ y(\phi) = \int_0^\phi \frac{dt}{\cos (t)}, \]
where \((x, y)\) are coordinates on a planar map and \((\lambda, \phi)\) are latitudes and longitudes on the face of the earth. Obtain an explicit form the projection by computing the integral.

---

Exercise 5.1.11. Compute the integral
\[ \int \arcsin \left( \sqrt{x + 1} - \sqrt{x} \right) \, dx . \]

Exercise 5.1.12. Compute the integral
\[ \int \frac{x^3 \arcsin x}{\sqrt{1 - x^2}} \, dx . \]

Exercise 5.1.13. The Riemann-Liouville generalized \( \alpha \)-derivative \((0 < \alpha < 1)\) for a function \( f(x) \) is defined as:
\[ \frac{d^\alpha}{dx^\alpha} f(x) = \frac{1}{\Gamma(1 - \alpha)} \frac{d}{dx} \int_0^x \frac{f(t)}{(x - t)^\alpha} \, dt , \]
where \( \Gamma \) is the gamma-function. Try to use this formula to compute
\[ \frac{d^\frac{1}{2}}{dx^\frac{1}{2}} \left( \frac{d^\frac{1}{2}}{dx^\frac{1}{2}} x^k \right) . \]
Discuss the result, is it what could be expected?

5.1.5 Integral and Differential Equations

Exercise 5.1.14. Consider the model for logistic growth of an animal population, the size of which is a function of time \( p(t) \) governed by the integral equation:
\[ p(t) = p(0) + k \int_0^t p(s) \left( 1 - \frac{p(s)}{L} \right) \, ds , \]
where \( k \in \mathbb{R} \) is a constant and \( L \) is the maximum size of the population which can be sustained by the amount of food available. It can be presumed that \( 0 < p(0) < L \). Compute the function \( p(t) \) and investigate the relative significance of the parameter \( k \).

Exercise 5.1.15. Compute the integral
\[ \alpha = \int_0^\frac{\pi}{4} f(t) \sin(t) \, dt \]
when it is known that the function \( f(x) \) solves the integral equation
\[ f(x) = x + \sin x \int_0^{\frac{\pi}{4}} f(t) \sin(t) \, dt = x + \alpha \sin x . \]

Exercise 5.1.16. The velocity \( v \) of a freely falling body with mass \( m = 1 \) kg is modeled by Newton's second law:
\[ m \frac{dv}{dt} = mg - kv^2 , \]
where \( t \) is time (measured in seconds), \( g = 9.81 \frac{m}{s^2} \) och \( k = 1 \frac{N\cdot s^2}{m^2} \). The term \( kv^2 \) is included in the mathematical model in order to account for airflow resistance.
5.2. OCTAVE

117

\( a) \) Eventually the velocity will reach a steady-state due to equilibrium between gravity and airflow resistance. Compute this “terminal velocity”.

\( b) \) Compute the function \( v(t) \) for the case \( v(0) = 0 \). Plot a graph.

5.1.6 Linear Algebra

Exercise 5.1.17. Consider the matrices

\[ P = \begin{pmatrix} 1 - a & a \\ b & 1 - b \end{pmatrix}, \quad S = \begin{pmatrix} -a & a \\ b & -b \end{pmatrix}. \]

\( a) \) Show that \( S^n = k^{n-1}S \), for \( n \geq 2 \) da \( k = -(a + b) \).

\( b) \) Show that \( P^n = I + \frac{(1+k)^{n-1}}{k} S \), where \( I \) is the identity matrix.

5.2 Octave

5.2.1 Linear Algebra

Exercise 5.2.1. Consider a matrix \( A \) and a vector \( b \) with

\[ A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad b = \begin{pmatrix} 36 \\ 88 \end{pmatrix}. \]

Solve the system of equations \( Ax = b \). Compute the LU- and QR-decompositions, as well as the eigenvalues and eigenvectors of \( A \). Compute the Cholesky decomposition of \( A^T A \) and verify that \( \text{cond} \left( A^T A \right) = \text{cond} \left( A \right)^2 \).

Solution:

\[ A = \text{reshape}(1:4,2,2).'; \quad b = [36; 88]; \]

\[ A\backslash b \]

\[ [L,U,P] = \text{lu}(A) \]

\[ [Q,R] = \text{qr}(A) \]

\[ [V,D] = \text{eig}(A) \]

\[ A2 = A.'*A; \]

\[ R = \text{chol}(A2) \]

\[ \text{cond}(A)^2 - \text{cond}(A2) \]

Exercise 5.2.2. The notorious ebola-virus breaks out in a population and from one week to another \( \frac{1}{10} \) of the well individuals from the previous week are reported to be infected. Half of the infected people of the previous week are cured the following week.

\( a) \) Express the mapping between vectors containing the number of well individuals \( f_i \) and ill individuals \( s_i \) from one week to another according to

\[ \begin{pmatrix} f_{n+1} \\ s_{n+1} \end{pmatrix} = A \begin{pmatrix} f_n \\ s_n \end{pmatrix}, \]

where \( A \) is a square matrix describing a Markov process. In such a matrix, the column sums will all be equal to one and the entries positive.
b) Compute the number of well people and ill people after one week, two and three weeks. Is there a “steady-state”, i.e., numbers of these that do not change?

c) Compute the eigenvalues of $A$. Is one an eigenvalue and what is the corresponding eigenvector?

**Exercise 5.2.3.** Consider the parabolic initial-boundary value problem

$$\frac{\partial^2 u(x,t)}{\partial x^2} = \frac{\partial u(x,t)}{\partial t}, \quad (x,t) \in [0,1] \times [0,\infty[$$

$$u(0,t) = 0,$$

$$u(1,t) = 0,$$

$$u(x,0) = e^{-\pi^2 \cdot 0} \sin(\pi x).$$

Set up a suitable discretization and solve the above system for the function $u(x,t)$.

Solution:

```matlab
function udot = dufun(u,t,A)
    udot = ( A * u )';
endfunction

h = 0.06;
x = 0:h:1;
N = size(x,2);
u0 = sin(pi*x);
DelA = diag(-2*ones(N,1)) + diag(ones(N-1,1),-1) + diag(ones(N-1,1),1);
DelA = DelA/(h^2);
t = linspace(0,1,20);
u = lsode(@(u,t) dufun(u,t,DelA),u0,t);
plot(x,u');
hold on;
plot(x,(sin(pi*x)')*exp(-(pi^2)*t),'+');
```

**5.2.2 Timing**

**Exercise 5.2.4.** Compute the matrix-vector product of a $100 \times 100$ random matrix and a random vector in two different ways. First, use the built-in matrix multiplication $\ast$. Next, use for-loops. Compare the results and computing times.

Solution:

```matlab
A = rand(100); b = rand(100,1);
t = cputime;
v = A*b; t1 = cputime-t;
w = zeros(100,1);
t = cputime;
```
for n = 1:100,
    for m = 1:100
        w(n) = w(n)+A(n,m)*b(m);
    end
end

t2 = cputime-t;

norm(v-w), t2/t1

Running this script yields the following output.

ans = 0
ans = 577.00

5.2.3 Stability Functions of BDF-integrators

Exercise 5.2.5. Calculate all the roots of the polynomial

$$\frac{147}{60} \zeta^6 - 6 \zeta^5 + \frac{15}{2} \zeta^4 - \frac{20}{3} \zeta^3 + \frac{15}{4} \zeta^2 - \frac{6}{5} \zeta + \frac{1}{6}.$$  

Hint: Use the command \texttt{compan}.

Plot these roots as points in the complex plane and draw a unit circle for comparison. (Hint: Use \texttt{hold, real} and \texttt{imag}).

![Figure 5.1: The roots of the polynomial for the BDF-integrator.](image)

Solution:
bdf6 = [147/60 15/2 -20/3 15/4 -6/5 1/6];
R = eig(compan(bdf6));
plot(R,'+'); hold on
plot(exp(pi*i*[0:.01:2]));
if any(find(abs(R)>1))
    fprintf('BDF6 is unstable\n');
else
    fprintf('BDF6 is stable\n');
end

5.2.4 3D Plotting

Exercise 5.2.6. Plot the graph of the function

\[ f(x, y) = \exp(-x^2 - y^2) . \]

![3D plot of the function](image)

Figure 5.2: The surface \( z = f(x, y) = e^{-x^2-y^2} \).

Solution:
x = -3:0.1:3;
[xx,yy] = meshgrid(x,x);
z = exp(-xx.^2-yy.^2);
figure, mesh(x,x,z);
title('e^{-x^2-y^2}');
print('3d-plot2.eps','-deps')
Exercise 5.2.7. Marine biologists have observed that when a shark smells blood in the water, it will swim in the direction where the concentration of blood increases most rapidly (steepest ascent). Based on certain experiments in ocean water, a model for the blood concentration (measured in ppm = “parts per million”) at some point \((x, y)\) on the water surface can be formulated according to
\[
c(x, y) = e^{-10^{-4}(x^2+2y^2)},
\]
when the point of discharge for the blood is at the origin and the coordinate positions are given in meters. Apply this model for the following tasks:

a) Obtain a graph with contours for \(c(x, y)\) and use this graph for anticipating the route of the shark from different coordinate points to the origin.

b) Suppose that the shark starts from the point \((x_0, y_0)\). Derive a differential equation for the route of the shark and solve this equation. Compare the solution to the anticipated one in a).

5.2.5 Hilbert Matrix

Exercise 5.2.8. For each \(n \times n\) Hilbert matrix \(H\), where \(n = 1, \ldots, 15\), compute the solution to the linear system \(Hx = b\), \(b = \text{ones}(n,1)\), and compare this solution to the exact one obtained through the exact inverse Hilbert matrix. Calculate the error and the condition number of the matrix and plot both in semi-logarithmic coordinates. (Hint: Use \texttt{hilb}, \texttt{invhilb}.)

Solution:

```matlab
err = zeros(15,1); co = zeros(15,1);
for k = 1:15
    H = hilb(k);
    b = ones(k,1);
    err(k) = norm(H\b-invhilb(k)*b);
    co(k) = cond(H);
end
semilogy(1:15,err,’r’,1:15,co,’x’);
```

5.2.6 Least Square Fit of a Straight Line

Exercise 5.2.9. Calculate the least square fit of a straight line to the points \((x_j, y_j)\), given as two vectors \(x\) and \(y\). Plot the points and the line.

Solution:

```matlab
function coeff = least_square (x,y)
    n = length(x);
    A = [x ones(n,1)];
    coeff = A\y;
    plot(x,y,’x’);
    hold on
```

CHAPTER 5. EXERCISES

5.2.7 Numerical Differentiation

**Exercise 5.2.10.** Consider an analytic function \( f(x) \) and its Taylor series in the complex plane according to

\[
  f(x_0 + ih) = f(x_0) + ih f'(x_0) - \frac{h^2}{2!} f''(x_0) - \frac{ih^3}{3!} f^{(3)}(x_0) + \ldots
\]

Suppose that we take the imaginary part of both sides of the above; this yields for the derivative

\[
  f'(x_0) = \text{Im} \left( \frac{f(x_0 + ih)}{h} \right) + O(h^2).
\]

Study this as an alternative approach to computing the derivative of \( f \) at \( x_0 \) as compared to the usual finite differences. As an example, take \( x_0 = \frac{\pi}{4} \) with the function

\[
  f(x) = e^x \sin^3 x + \cos^3 x.
\]

5.2.8 Trapezoidal Rule

**Exercise 5.2.11.** Write a program to integrate an arbitrary function \( f \) in one variable on an interval \([a, b]\) numerically, i.e., estimate

\[
  \int_a^b f(x) \, dx
\]

using the trapezoidal rule with subinterval division \( h = \frac{b-a}{N} \):

\[
  \int_a^b f(x) \, dx \approx h \left( \frac{f(a) + f(b)}{2} + \sum_{j=1}^{N-1} f(a + jh) \right).
\]

For a function \( f \) of choice, we will check by generating a doubly logarithmic error plot that the trapezoidal rule is of order 2. Solution:

```matlab
function S = trapez(fun,a,b,N)
    h = (b-a)/N;
    % fy = feval(fun,[a:h:b]); better:
    fy = feval(fun,linspace(a,b,N+1));
    fy(1) = fy(1)/2;
    fy(N+1) = fy(N+1)/2;
    S = h*sum(fy);
```
function y = f(x)
    y = exp(x);
end

for k=1:15;
    err(k) = abs(exp(1)-1-trapez('f',0,1,2^k));
end
loglog(1./2.^[1:15],err);
hold on;
title('Trapezoidal rule, f(x) = exp(x)');
xlabel('Increment');
ylabel('Error');
loglog(1./2.^[1:15],err,'x');

5.2.9 Optimization

Exercise 5.2.12. Compute possible extreme values for the function

\[ f(x, y) = x^2 + y^2 - 2x - 6y + 14. \]

In case of a numerical strategy, use a plot for obtaining initial values.

Exercise 5.2.13. Investigate whether the function \( f(x, y) = 2x^2 - 2y^2 - \ln (1 - x^2 - y^2) \) has a smallest value on the domain \( D_f = \{(x, y) : x^2 + y^2 < 1\} \).

Exercise 5.2.14. Study the function

\[ f(x) = \tan (\sin (x)) - \sin (\tan (x)). \]

Investigate whether the function has a maximum or a least upper bound.

5.2.10 Sequences

Exercise 5.2.15. The logistic sequence is used in biology for modeling the dynamics of populations, according to

\[ p_{n+1} = kp_n (1 - p_n) , \quad k > 0 , \]

where \( p_n \) corresponds to the size of the \( n \)th generation of some animal species. For simplicity, \( p_n \) can be taken to be the fraction of the maximum population, i.e., \( 0 \leq p_n \leq 1 \). Compute the \( N \) initial terms of the sequence \( \{p_n\}_n=0^\infty \), with starting point \( p_0 \), \( 0 < p_0 < 1 \) for the cases

(a) \( N = 30 \), \( p_0 = \frac{1}{2} \) for two values of \( k \), \( 1 < k < 3 \). Provide a graph and estimate the convergence of the sequence. Try also other values for \( p_0 \) and \( k \).
(b) Repeat (a) for $k \in [3.0, 3.4]$.

(c) Repeat (a) for $k \in [3.4, 3.5]$.

(d) Repeat (a) for $k \in [3.6, 4.0]$, provide a graph of at least 1000 terms and discuss the result. What happens if $p_0$ is changed by 0.001? This type of behavior is called chaotic and can be observed in certain insect populations.
Chapter 6

Course problems

In this final chapter, we consider a set of problems for application of the software tools treated above. Many of the problems are from various sectors of chemical engineering, however, some more general topics from mathematical physics and applied mathematics are also included. Although many of the exercises can be done without software tools, the basic idea is to demonstrate the use of wxMaxima and Octave in their solution. Therefore, these packages should be applied as extensively as possible and a self-contained report of the theory together with documentation and program listings should be included in the solution.

6.1 Single Nonlinear Equation

We use the van der Waal equation of state to calculate molar volume and compressibility factor for a gas.

The ideal gas law is known to describe the pressure-volume-temperature relationship for a gas only at relatively low (near atmospheric) pressures. At higher pressures, a more complex pressure-volume-temperature model is required. The van der Waal equation of state is given by

\[
\left( p + \frac{a}{V^2} \right) (V - b) = RT ,
\]

where

\[
a = \frac{27}{64} \left( \frac{R^2 T_c^2}{p_c} \right),
\]

and

\[
b = \frac{RT_c}{8p_c} .
\]
The variables and constants are defined as follows:

\[ p = \text{pressure in atm.} \]
\[ V = \text{molar volume in } \text{liters/mol}. \]
\[ T = \text{temperature in K.} \]
\[ R = \text{gas constant} \left( 0.08206 \text{ atm \cdot liter/mol \cdot K} \right). \]
\[ T_c = \text{critical temperature} \ (405.5 \text{ K for ammonia}). \]
\[ p_c = \text{critical pressure} \ (111.3 \text{ atm for ammonia}). \]

The reduced pressure, \( p_r \), and compressibility factor are defined as:

\[ p_r = \frac{p}{p_c}, \quad Z = \frac{pV}{RT}. \]

**Exercise 6.1.1.** Based on the above theory, do the following:

(a) Compute the molar volume and compressibility for gaseous ammonia at pressure \( p = 56 \text{ atm} \) and temperature \( T = 450 \text{ K} \) using the van der Waal equation of state.

(b) Repeat the computation for the following reduced pressures \( p_r = 1, 2, 4, 10 \) and 20.

(c) Depict the variation \( Z(p_r) \).

### 6.2 Another Single Nonlinear Equation

Here, an application from fluid dynamics will be studied. We calculate the terminal velocity for solid particles falling in a fluid under the force of gravity.

A simple force balance applied to a spherical particle reaching terminal velocity in a fluid leads to the equation

\[ v_t = \sqrt{\frac{4g (\rho_p - \rho) D_p}{3C_D \rho}}, \]

where \( v_t \) is the terminal velocity in \( \frac{\text{m}}{\text{s}} \), \( g = 9.81 \frac{\text{m}}{\text{s}^2} \) is the acceleration of gravity, \( \rho \) and \( \rho_p \) are the densities in \( \frac{\text{kg}}{\text{m}^3} \) of the fluid and of the particle, respectively, \( D_p \) is the diameter of the spherical particle in m and \( C_D \) is a dimensionless drag coefficient.

The drag coefficient on a spherical particle at terminal velocity varies with
the Reynolds number $Re$ as follows:

\[
C_D = \begin{cases} 
\frac{24}{Re} , & Re < 0.1 \\
\frac{24}{Re} (1 + 0.14 Re^{0.7}) , & 0.1 \leq Re \leq 1000 \\
0.44 , & 1000 < Re \leq 350000 \\
0.19 - \frac{80000}{Re} , & Re > 350000 
\end{cases}
\]

where the Reynolds number is defined as

\[
Re = \frac{D p v}{\eta},
\]

with the dynamic viscosity $\eta$ in Pa s or $\text{kg m}^{-1}\text{s}$. 

**Exercise 6.2.1.** Calculate the terminal velocity for

(a) coal particles with $\rho_p = 1800 \ \text{kg m}^{-3}$, $D_p = 208 \cdot 10^{-6} \ \text{m}$ falling in water at $T = 298.15 \ \text{K}$ with $\rho = 994.6 \ \text{kg m}^{-3}$ and $\eta = 8.931 \cdot 10^{-4} \ \text{kg m}^{-1}\text{s}$

(b) coal particles in water separated in a centrifugal device with an acceleration of $30g$.

### 6.3 A Set of Linear Equations

We consider material balance equations applied to a steady-state process without recycling.

The four solvents xylene, styrene, toluene and benzene are separated from a composite inflow by use of the array of distillation towers shown in the figure below, where $F$, $B$, $B_1$, $B_2$, $D$, $D_1$ and $D_2$ are the molar flow rates measured in $\text{mol min}^{-1}$.

Applying material balances for the four individual components yields the set of linear equations:

\[
\begin{align*}
\text{Xylene:} & \quad 0.07D_1 + 0.18B_1 + 0.15D_2 + 0.24B_2 = 0.15 \cdot 70 \\
\text{Styrene:} & \quad 0.04D_1 + 0.24B_1 + 0.10D_2 + 0.65B_2 = 0.25 \cdot 70 \\
\text{Toluene:} & \quad 0.54D_1 + 0.42B_1 + 0.54D_2 + 0.10B_2 = 0.40 \cdot 70 \\
\text{Benzene:} & \quad 0.35D_1 + 0.16B_1 + 0.21D_2 + 0.01B_2 = 0.20 \cdot 70
\end{align*}
\]

Moreover, we can formulate overall balances as well as component material
balances for the two distillation towers; for column \#2 we obtain:

\[
Molar \text{ flow rates: } D = D_1 + B_1
\]

\[
Xylene: \quad X_{Dx}D = 0.07D_1 + 0.18B_1
\]

\[
Styrene: \quad X_{Ds}D = 0.04D_1 + 0.24B_1
\]

\[
Toluene: \quad X_{Dt}D = 0.54D_1 + 0.42B_1
\]

\[
Benzene: \quad X_{Db}D = 0.35D_1 + 0.16B_1.
\]

Here, \(X_{Dx}\), \(X_{Ds}\), \(X_{Dt}\) and \(X_{Db}\) denote the mole fractions of xylene, styrene, toluene and benzene, respectively. Similarly, for the distillation column \#3, we have

\[
Molar \text{ flow rates: } B = D_2 + B_2
\]

\[
Xylene: \quad X_{DxB}B = 0.15D_2 + 0.24B_2
\]

\[
Styrene: \quad X_{Dx}B = 0.10D_2 + 0.65B_2
\]

\[
Toluene: \quad X_{Dt}B = 0.54D_2 + 0.10B_2
\]

\[
Benzene: \quad X_{Db}B = 0.21D_2 + 0.01B_2.
\]

**Exercise 6.3.1.** Compute the molar flow rates for

(a) streams \(B_1, B_2, D_1\) and \(D_2\).

(b) streams \(B\) and \(D\) and their composition.
6.4 Nonlinear Regression

We use polynomials, a modified Clausius-Clapeyron equation and the Antoine equation to model vapor pressure versus temperature data in thermodynamics. Consider the vapor pressure data for benzene in the table below:

<table>
<thead>
<tr>
<th>Temperature, T (°C)</th>
<th>Pressure, P (mm Hg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>-36.7</td>
<td>1</td>
</tr>
<tr>
<td>-19.6</td>
<td>5</td>
</tr>
<tr>
<td>-11.5</td>
<td>10</td>
</tr>
<tr>
<td>-2.6</td>
<td>20</td>
</tr>
<tr>
<td>+7.6</td>
<td>40</td>
</tr>
<tr>
<td>15.4</td>
<td>60</td>
</tr>
<tr>
<td>26.1</td>
<td>100</td>
</tr>
<tr>
<td>42.2</td>
<td>200</td>
</tr>
<tr>
<td>60.6</td>
<td>400</td>
</tr>
<tr>
<td>80.1</td>
<td>760</td>
</tr>
</tbody>
</table>

We now wish to model the relationship $p(T)$ based on a set of regression models and the data given in the table. Initially, we consider a polynomial

$$p(T) = \sum_{i=0}^{n} a_i T^n = a_0 + a_1 T + a_2 T^2 + \cdots + a_n T^n,$$

where $a_0, a_1, a_2, \ldots, a_n$ are the parameters (coefficients) to be determined by regression and $n$ is the degree of the polynomial. Beside the coefficients, also the choice of the degree $n$ is customarily optimized so as to give the best representation of the tabulated data by the polynomial. Our criterion for "best representation" is attainment of a minimum of the least-squares objective function:

$$\sum_{\text{data points}} (p_{\text{data}} - p(T_{\text{data}}))^2.$$

An alternative to polynomial regression is the equation of Clausius-Clapeyron, which for our data units take the form

$$\log (p(T)) = A - \frac{B}{T + 273.15}.$$ 

Note that the denominator is just the temperature on the absolute scale. Parameters $A$ and $B$ are parameters to be determined by the regression.
A similar form of mathematical model is the Antoine equation, with the modification of one added parameter in comparison with Clausius-Clapeyron’s equation:

$$\log (p(T)) = A - \frac{B}{T + C}.$$ 

In this case, however, the regression analysis is trickier, as the Antoine equation cannot be linearized. Therefore, nonlinear regression must be utilized.

**Exercise 6.4.1.** Perform regression analysis on the above tabulated data with mathematical model according to
(a) an $n$:th degree polynomial with optimal choice of $n$.
(b) the Clausius-Clapeyron’s equation.
(c) the Antoine equation.

### 6.5 A Set of Nonlinear Equations

We consider a system of nonlinear algebraic equations describing chemical equilibrium for multiple reactions. The following chemical reactions are taking place in a constant-volume, gas-phase batch reactor:

- $A + B \leftrightarrow C + D$
- $B + C \leftrightarrow X + Y$
- $A + X \leftrightarrow Z$

A system of algebraic equations describes the equilibrium of these reactions. From thermodynamics and stoichiometry we can derive the following relationships.

$$K_{C_1} = \frac{c_C c_D}{c_A c_B}, \quad K_{C_2} = \frac{c_X c_Y}{c_B c_C}, \quad K_{C_3} = \frac{c_Z}{c_A c_X}.$$ 

$$c_A = c_{A_0} - c_D - c_Z, \quad c_B = c_{B_0} - c_D - c_Y,$$
$$c_C = c_D - c_Y, \quad c_Y = c_X + c_Z.$$

Here, the variables $c_A, c_B, c_C, c_D, c_X, c_Y$ and $c_Z$ denote the equilibrium concentrations of the various species, resulting from the initial concentrations $c_{A_0}$ and $c_{B_0}$. The equilibrium constants $K_{C_1}, K_{C_2}$ and $K_{C_3}$ are known.

**Exercise 6.5.1.** Compute the equilibrium concentrations $c_A, c_B, c_C, c_D, c_X, c_Y$ and $c_Z$ from the above equations both algebraically and numerically, based on the parameter values $K_{C_1} = 1.06, K_{C_2} = 2.63, K_{C_3} = 5$ and $c_{A_0} = 1.5 = c_{B_0}$.

For the numerical solution, use the starting points
(a) $c_D = c_X = c_Z = 0$
(b) $c_D = c_X = c_Z = 1$
(c) $c_D = c_X = c_Z = 10$
6.6 A Set of Ordinary Differential Equations

In this problem, a set of first order ordinary differential equations is used to model unsteady heat transfer in a series of perfectly mixed tanks. Three tanks in series are used for preheating a multicomponent oil mixture before it is fed to a distillation column, see the figure below. Each tank is initially filled with 1000 kg of oil at 20 °C. Saturated steam at 250 °C condenses within coils immersed in the tanks. Oil is fed into the first tank at a rate of 100 \( \text{kg/min} \) and it overflows into the subsequent tanks at the same flow rate. The temperature of the feed into the first tank is 20 °C. The tanks are perfectly mixed, so that the temperature inside the tanks is uniform and the steam outlet temperature agrees with this interior temperature. The heat capacity of the oil is \( C_p = 2.0 \ \text{kJ/kg} \). For a particular tank, the rate at which heat is transferred from the steam to the oil is given by

\[
Q = K (T_{\text{steam}} - T) ,
\]

where \( K = 10 \ \text{kJ/min}^{-1} \) is the effective heat transfer coefficient of the coil in the tanks, \( T \) is the temperature of the steam in °C and \( Q \) is the rate of heat transferred in \( \text{kJ/min} \). We now proceed to formulate energy balances for each of the three tanks. Note that the mass flow rate into each tank is the same, i.e., \( W = W_1 = W_2 = W_3 \). Moreover, the mass in each tank is constant, as the volume and density of the oil mixture is constant. Hence, \( M = M_1 = M_2 = M_3 \) and we see that for the first tank:

\[
M C_p \frac{dT_1}{dt} = W C_p T_0 + K (T_{\text{steam}} - T_1) - W C_p T_1 .
\]

It should be noted that we do not need mass balances for any of the tanks, since the masses do not change at steady state. For the second and third tanks, we derive similar equations:

\[
M C_p \frac{dT_2}{dt} = W C_p (T_1 - T_2) + K (T_{\text{steam}} - T_2) ,
\]

\[
M C_p \frac{dT_3}{dt} = W C_p (T_2 - T_3) + K (T_{\text{steam}} - T_3) .
\]

Exercise 6.6.1. Determine the steady-state temperature in each of the three tanks. What time will be required for \( T_3 \) to reach 99 % of its steady-state temperature after startup?
6.7 Another Set of Ordinary Differential Equations

We study a set of ordinary differential equations with known boundary conditions, with the objective of aiding the design of a gas phase catalytic reactor with pressure drop. In the packed-bed reactor depicted below, a first-order reversible gas phase reaction is taking place according to

\[ 2A \rightleftharpoons C. \]

The reactor is surrounded by a heat exchanger and it works in plug flow mode without radial gradients (differences) in temperature, pressure and concentration at any location within the packed bed made up of the solid catalyst.

The most important variables for reactor design is the conversion \( X \) and temperature \( T \) as functions of the location within the catalyst bed, specified by the catalyst weight \( W \). A material balance for reactant \( A \) then leads to the differential equation

\[ F_A \frac{dX}{dW} = -r_A, \]

where the catalytic reaction rate is given by

\[ r_A = k_r \left( c_A^2 - \frac{c_C}{K_C} \right). \]

The rate constant \( k \) is based on the Arrhenius expression for reactant \( A \)

\[ k_r = k e^{E_A/R (1/450 - 1/T)}, \]

and the equilibrium constant at each temperature can be calculated from the similar van’t Hoff-expression

\[ K_C = K_C e^{\Delta H_r/R (1/450 - 1/T)}. \]

With the help of stoichiometry, we can obtain expressions for the concentr-
tions $c_A$ and $c_C$ according to

$$
c_A = c_{A_0} \left( \frac{1 - X}{1 - 0.5X} \right) \frac{T_0}{T} y,
$$

$$
c_C = c_{A_0} \left( \frac{0.5X}{1 - 0.5X} \right) \frac{T_0}{T} y,
$$

$$
y = \frac{p}{p_0}.
$$

The pressure drop along the packed bed reactor is governed by the differential equation for partial pressure

$$
\frac{dy}{dW} = -\alpha \frac{1 - 0.5X}{2y} \frac{T}{T_0}.
$$

Applying an energy balance yields a differential equation for temperature

$$
\frac{dT}{dW} = \frac{U_a (T_a - T) + r_A \Delta H_r}{F_{A_0} C_p^A}.
$$

**Exercise 6.7.1.** Solve the coupled differential equations for the conversion $X$, reduced pressure $y$ and temperature $T$ along the reactor bed, i.e., for the interval $W \in [0, 20]$ kg. Use parameter values from the table below

<table>
<thead>
<tr>
<th>$C_p^A$</th>
<th>$R$</th>
<th>$F_{A_0}$</th>
<th>$\Delta H_r$</th>
<th>$E_A$</th>
<th>$k$</th>
<th>$\alpha$</th>
<th>$K_C$</th>
<th>$p_0$</th>
<th>$c_{A_0}$</th>
<th>$y_{A_0}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>40.0</td>
<td>8.314</td>
<td>5.0</td>
<td>-40</td>
<td>41.8</td>
<td>0.5</td>
<td>0.015</td>
<td>25000</td>
<td>10</td>
<td>0.271</td>
<td>1.0</td>
</tr>
</tbody>
</table>

(a) Plot the results in a representative graph.

(b) At approximately $W = 16$ kg, a special feature should appear in the results of (a). Investigate which parameters affect this and the reasons behind it.

(c) Plot the concentration profiles $c_A(W)$ and $c_C(W)$. 
6.8 A Second-Order Ordinary Differential Equation

We consider an application from transport phenomena and reaction kinetics, modeled by a two-point boundary value problem for a second-order ordinary differential equation. Diffusion with simultaneous first-order irreversible chemical reaction in a single phase containing only reactant $A$ and product $B$ can be described by the differential equation

$$\frac{d^2c_A}{dz^2} = \frac{k}{D_{AB}} c_A,$$

where $c_A$ is the concentration of reactant $A$ in $\text{kmol/m}^3$, $z$ is the spatial variable in m, $k$ is the homogeneous reaction rate constant in $\text{s}^{-1}$ and $D_{AB}$ is the binary diffusion coefficient in $\text{m}^2/\text{s}$. A typical geometry for this problem is a one-dimensional layer with one end exposed to a known concentration $c_{A0}$ and the other end without diffusion, i.e., no flux of material across that boundary. Hence, the boundary conditions would be

$$c_A = c_{A0}, \quad z = 0,$$
$$\frac{dc_A}{dz} = 0, \quad z = L.$$

Exercise 6.8.1. Apply the numerical data

$$c_{A0} = 0.2 \text{ kmol/m}^3,$$
$$k = 10^{-3} \text{ s}^{-1},$$
$$D_{AB} = 1.2 \cdot 10^{-9} \frac{\text{m}^2}{\text{s}},$$
$$L = 10^{-3} \text{ m}$$

and compute

(a) an analytical solution of $c_A(z)$, $z \in [0, L]$.

(b) a numerical solution of $c_A(z)$, $z \in [0, L]$.

(c) depict the the concentration profiles obtained in (a) and (b) in the same graph.

6.9 Ordinary Differential Equations and Algebraic Equations

We consider batch distillation of an ideal binary mixture of the two components designated “1” and “2”. Then, the moles of liquid remaining to be distilled $L$
as a function of the mole fraction $x_2$ of component “2” can be described by the ordinary differential equation

$$\frac{dL}{dx_2} = \frac{L}{x_2(k_2 - 1)} ,$$

where $k_2$ is the vapor-liquid equilibrium rate for component 2. For this ideal system, we have $k_i = \frac{p_i}{p_{tot}}$ where $p_i$ is the vapor pressure of component $i$ and $p_{tot}$ is the total pressure. The pressures are measured in mmHg.

A common model for vapor pressure of component $i$ is the Antoine equation

$$p_i = 10^{(A_i - \frac{B_i}{T + C_i})} ,$$

where $A_i$, $B_i$ and $C_i$ are parameters and $T$ is temperature measured in °C.

The temperature inside the distillation pot follows something called the “bubble point curve” and is governed by the implicit algebraic equation

$$k_1x_1 + k_2x_2 = 1 .$$

Exercise 6.9.1. An ideal mixture of benzene (component 1) and toluene (component 2) obeys the Antoine equation with the parameters

$A_1 = 6.90565 , \quad A_2 = 6.95464$

$B_1 = 1211.033 , \quad B_2 = 1344.8$

$C_1 = 220.79 , \quad C_2 = 219.482$

The total pressure is $p_{tot} = 912$ mmHg. Given that $L = 100$ mol for $x_2 = 40 \%$, compute $L$ for $x_2 = 80 \%$.

6.10 A Control Loop

We study dynamics of the continuous first-order process with delay depicted below. The system consists of a well-stirred tank, heater and PI-temperature controller. A stream of liquid with density $\rho$, specific heat $c_p$ and temperature $T_i$ is fed into the heated tank at a constant rate of $W$. The volume and temperature in the well-stirred tank is $V$ and $T$, respectively. The temperature of the discharge (at rate $W$ and temperature $T_o$) from the tank is controlled by heating the water in the tank at rate $q$, based on the measurement $T_m$ with a set point temperature $T_r$. The objective is to maintain $T_o = T_r$ although the inlet temperature $T_i$ may differ from the steady-state design temperature $T_{is}$.

In order to model this system, governing model and control equations must be formulated. An energy balance for the tank yields the differential equation:

$$\frac{dT}{dt} = \frac{Wc_p(T_i - T) + q}{\rho Vc_p} ,$$

with initial condition $T = T_r$ at $t = 0$, corresponding to steady-state operation at the set point temperature $T_r$. 
The thermocouple for measuring $T_m$ is described by a first-order system with delay $\tau_d$ for the discharge liquid from the tank to reach the measurement point. Thus, we have

$$T_o(t) = T(t - \tau_d).$$

The effect of this delay may be taken into account by a Padé approximation giving a first-order linear differential equation for the discharge temperature

$$\frac{dT_o}{dt} = \frac{2}{\tau_d} (T - T_o) - \frac{dT}{dt}, \quad T_o(t = 0) = T_r.$$

The above equation gives the input temperature for the thermocouple. However, its shielding and electronics are modeled by a first-order system for the dynamics of the measured temperature $T_m$:

$$\frac{dT_m}{dt} = \frac{T_o - T_m}{\tau_m},$$

where the time constant $\tau_m$ is known. The heat input to the tank, as governed by the controller is described by

$$q = q_s + K_c (T_r - T_m) + \frac{K_c}{\tau_I} \int_0^t (T_r - T_m(s)) \, ds,$$

where $K_c$ is the proportional gain of the controller, $\tau_I$ is the integration time constant or reset time. The heat input $q_s$ is that required at steady state for the design conditions:

$$q_s = W c_p (T_r - T_{is}).$$

For convenience, we can note that the integral in the controller equation above satisfies the fundamental theorem of calculus, i.e., we have another differential equation for the integral according to:

$$\frac{d}{dt} \int_0^t (T_r - T_m(s)) \, ds = T_r - T_m(t).$$

**Exercise 6.10.1.** Solve the coupled differential equations for the temperatures $T$, $T_o$ and $T_m$ in
6.11 STIFF DIFFERENTIAL EQUATIONS

(a) A situation where the system initially operates at design steady state for the temperature 80 °C when the inlet temperature $T_i$ is suddenly changed to 40 °C at $t = 10$ min. Let the control loop be open ($K_c = 0$).

(b) A situation similar to case (a) with the parameter values in the table below.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\rho V c_p$</td>
<td>4000 $\frac{kJ}{K}$</td>
</tr>
<tr>
<td>$W c_p$</td>
<td>500 $\frac{kJ}{min K}$</td>
</tr>
<tr>
<td>$T_{is}$</td>
<td>333.15 K</td>
</tr>
<tr>
<td>$T_r$</td>
<td>353.15 K</td>
</tr>
<tr>
<td>$\tau_d$</td>
<td>1 min</td>
</tr>
<tr>
<td>$\tau_m$</td>
<td>5 min</td>
</tr>
<tr>
<td>$K_c$</td>
<td>50 $\frac{kJ}{min K}$</td>
</tr>
<tr>
<td>$\tau_I$</td>
<td>2 min</td>
</tr>
</tbody>
</table>

6.11 Stiff Differential Equations

A differential equation is considered “stiff” if the solution sought is varying slowly, while other, nearby solutions vary rapidly. This means that stiffness is an issue connected with computational efficiency when one solves the differential equation numerically. If computation time would be irrelevant, we would not be concerned about stiffness. We could then use standard numerical methods for the solution and wait for a sufficient amount of steps for the computation to be executed. However, since the decrease in computational speed may be orders of magnitude in pathological cases, it is desired to apply special methods to solve this type of equations.

**Example 6.11.1.** A simple stiff ordinary differential equation can be found in a model of flame propagation. When a match is lit, the ball of flame grows very rapidly until reaching a critical size. It then remains at this size because the amount of oxygen consumed in this combustion process balances the amount of oxygen sucked into the ball through its surface. The model can be formulated dimensionlessly as

$$\frac{dy}{dt} = y^2 - y^3, \quad 0 < t < \frac{2}{\delta},$$

where the scalar variable $y(t)$ represents the radius of the flame ball. The terms $y^2$ and $y^3$ come from the surface area and volume, respectively, of this ball. A critical parameter in this case is $\delta$, describing the “small” initial radius of the flame ball.

**Exercise 6.11.1.** Solve the flame propagation problem above

(a) analytically. Apply the Lambert W-function.

(b) numerically, by use of standard “nonstiff” methods as well as special “stiff” methods. Consider differences in computation time and compare the results to that of case (a).
6.12 Partial Differential Equations

Partial differential equations (PDE) have more than one independent variable, in contrast to ordinary differential equations (ODE). Most computational methods for PDE are based on some sort of reformulation into ODE and subsequent use of efficient solution methods for these. There is an extensive theory, with continuous and vigorous development over recent years, for PDE solution properties and existence. Nevertheless, some of the most popular solution methods are classical and we will take a look at one here.

Example 6.12.1. Consider the PDE describing the (normalized) temperature in the earth at depth $x$ and time $t$, when the temperature at the surface is $h(t)$.

$$\frac{\partial^2 u(x, t)}{\partial x^2} = \frac{\partial u(x, t)}{\partial t}, \quad x \in [0, \infty], \quad t \in [0, \infty],$$

$$u(0, t) = h(t),$$

$$\lim_{x \to \infty} u(x, t) = 0,$$

$$u(x, 0) = 0.$$

This problem can be solved using numerous strategies, to which we will return shortly. First, we will take a look at how the time variation in the boundary condition is handled. Consider a similar problem for a constant boundary condition:

$$\frac{\partial^2 v(x, t)}{\partial x^2} = \frac{\partial v(x, t)}{\partial t}, \quad x \in [0, \infty], \quad t \in [0, \infty],$$

$$v(0, t) = H,$$

$$\lim_{x \to \infty} v(x, t) = 0,$$

$$v(x, 0) = 0.$$

Now, according to Duhamel’s principle we have the following convolution-like relationship between the two solutions $u(x, t)$ and $v(x, t)$:

$$u(x, t) = h * \frac{\partial v(x, \cdot)}{\partial t} = \int_0^t h(x) \frac{\partial v(x, t - \tau)}{\partial t} \, d\tau.$$

Exercise 6.12.1. Consider the initial-boundary value problem for $v(x, t)$. Solve it using the approach:

(a) Introduce the new independent variable $\eta = \frac{x}{\sqrt{4t}}$ and convert the problem to a boundary value problem for an ordinary differential equation. Solve the ODE with the new boundary conditions.

(b) Use the Laplace-transform to convert the problem to a boundary value problem for an ordinary differential equation. Solve the ODE with the new boundary conditions.

Furthermore, try to verify the relationship between $u(x, t)$ and $v(x, t)$ by using the Laplace-transform. Finally, solve the complete problem for $u(x, t)$ numerically and compare the result to the one given above by Duhamel’s principle.
6.13 Integral Equations

An equation where the unknown function/solution is part of an definite (constant integration bounds) or indefinite (variable bounds) integral is customarily termed an \textit{integral equation}. The former type is called a \textit{Fredholm} equation and the latter a \textit{Volterra} equation. If the unknown solution appears only in the integrand, the equation is designated type I and otherwise type II.

\textbf{Definition 6.13.1.} A type II \textit{Volterra} integral equation

\[ y(t) = f(t) + \int_{a}^{t} K(x, t)y(x) \, dx \]

\textbf{Definition 6.13.2.} A type I \textit{Fredholm} integral equation

\[ f(t) = \int_{a}^{b} K(x, t)y(x) \, dx \]

Whereas a differential equation describes the solution \( y(t) \) \textit{locally} through its derivatives, an integral equation describes the global variation over the entire definition set of the solution \textit{including} the boundary conditions. For a differential equation, the boundary conditions have to be specified separately whereas they are condensed into the properties of the \textit{kernel} \( K(x, t) \) in the integral equation.

Integral equations provide an alternative to modeling processes by differential equations in engineering. In many respects an integral equation description is more compact and more suited to investigations of, e.g., solution existence and uniqueness. There are also physical phenomena, such as heat radiation in energy balance formulations, which cannot be modeled by differential equations but readily with integral equations.

Frequently, in modeling and simulation, an objective is to analyze sensitivity to input disturbances. Here, integral equations come in handy, as the integral \( \int K(x, t)y(x)dx \) often has the nice property of mollifying rapid variation in \( y(x) \) whereas the effect of \( y'(x) \) is the opposite. In many cases, though, differential equations are more suited to practical computations, since closed-form solutions do not exist for integral equations nearly as frequently.

Integral equations may be solved by a great variety of methods, based on the type of the equation and its particular form of the kernel \( K(x, t) \). We will briefly consider a few simple methods for the \textit{Volterra} equation of type II

\[ y(t) = f(t) + \int_{a}^{t} K(x, t)y(x) \, dx \]

and the \textit{Fredholm} equation of type II

\[ y(t) = f(t) + \int_{a}^{b} K(x, t)y(x) \, dx \]
Differentiating the integral equation leads to an initial value problem for a differential equation. This approach works for, e.g., a type II Volterra and in a case where the kernel only depends on \(x\), i.e., \(K = K(x)\), we obtain a linear differential equation

\[
y'(t) - K(t)y(t) = f'(t), \quad y(a) = f(a),
\]

**Exercise 6.13.1.** Verify the above differential equation and obtain a formal solution together with an example for the kernel \(K(x) = 3x^2\).

In addition to differentiation, iteration can be applied for solution of Volterra equations of the type II. We start with the initial guess \(y_0(t)\) and solve for \(y_1(t), y_2(t), \ldots\), from the original integral equation according to

\[
y_{j+1}(t) = f(t) + \int_a^t K(x, t)y_j(x) \, dx
\]

It can be proved that the sequence of solution always converges for continuous kernels \(K(x, t)\) and \(f(t)\).

**Exercise 6.13.2.** Consider a Volterra equation of type II with the kernel \(K(x, t) = 3x^2\) and \(f(t) = 1, \ a = 0\). Solve this equation by iteration based on the initial guess \(y_0(t) = 1\).

The Fredholm equation of type II can also be solved in closed form, if the kernel can be separated similarly to the approach of “separating variables” in solution of partial differential equations:

\[
K(x, t) = \sum_{n=1}^N \phi_n(x)\alpha_n(t).
\]

In many cases this separation can be achieved approximately, by using linearly independent basis functions \(\phi_n(x)\). Considering this in the integral equation under the assumption that the order of summation and integration can be interchanged (the sum is finite and convergence of the integral is rapid enough), leads to the equation

\[
y(t) = f(t) + \sum_{n=1}^N \alpha_n(t) \int_a^b \phi_n(x)y(x) \, dx.
\]

Noting that the integral is a constant depending on the particular value of the summation index \(n\) allows us to introduce

\[
c_n = \int_a^b \phi_n(x)y(x) \, dx.
\]

If we now multiply the integral equation in the previous step above by \(\phi_m(t)\) and integrate, we obtain

\[
c_m = \int_a^b \phi_m(t)f(t) \, dt + \sum_{n=1}^N c_n \int_a^b \alpha_n(t)\phi_m(t) \, dt.
\]
6.14. SPECIAL TOPICS

Considering this for \( m = 1, \ldots, N \) yields a system of simultaneous linear equations, which can be expressed in matrix form as

\[(I - A) c = b,\]

where \( I \) is the identity matrix, \( c \) the vector consisting of the coefficients \( c_k \) and \( b \) the vector of the known constants \( b_m = \int_a^b \phi_m(t)f(t) \, dt \). The matrix \( A \) consists of the constants \( \int_a^b \alpha_n(t)\phi_m(t) \, dt \). Hence, the integral equation has been reduced to a linear algebra problem of an equation system with the solution

\[ c = (I - A)^{-1} b. \]

With the above solution the Fredholm equation of type II can be solved using the formula

\[ y(t) = f(t) + \sum_{n=1}^{N} c_n \alpha_n(t). \]

Exercise 6.13.3. Consider the equation

\[ y(t) = 1 + \int_0^1 3x^2(e^t^2 - 1) / (e - 1) \, y(x) \, dx. \]

Solve this equation using the above described separation strategy.

6.14 Special Topics

6.14.1 Perturbation Theory

Consider the equation satisfied by \( y(x) \):

\[ y + \frac{1}{50} \ln (1 + y) = x^2. \]

Unfortunately, this equation cannot be solved exactly, however, we can apply a perturbation method amounting to the following: We replace the factor \( \frac{1}{50} \) in front of the logarithm by a parameter, \( \epsilon \), and seek an approximate solution \( y(x, \epsilon) \) to the modified equation

\[ y + \epsilon \ln (1 + y) = x^2. \]

We may assume that this solution can be expressed as a MacLaurin-series for the parameter \( \epsilon \):

\[ y(x, \epsilon) = y(x, 0) + \epsilon y_\epsilon(x, 0) + \frac{\epsilon^2}{2} y_{\epsilon\epsilon}(x, 0) + \cdots, \]

where \( y_\epsilon(x, 0) = \frac{\partial y(x, \epsilon)}{\partial \epsilon} \bigg|_{\epsilon=0} \), \( y_{\epsilon\epsilon}(x, 0) = \frac{\partial^2 y(x, \epsilon)}{\partial \epsilon^2} \bigg|_{\epsilon=0} \) etc. These derivatives can now be computed through differentiating the modified equation and substituting \( \epsilon = 0 \). This, e.g., immediately yields \( y(x, 0) = x^2 \). Determine the five first terms of the above MacLaurin-series and put \( \epsilon = \frac{1}{50} \) to obtain an approximation of the initial equation.
6.14.2 Calculus of Variations

We will take a look at a method of solving variational problems with constraints by use of constraints. In general form, we are interested in optimizing an integral of a functional $F(x, y, y')$ (a function of our function $y(x)$, which we adjust in order to reach the optimum)

$$I[y(x)] = \int_{x_0}^{x_1} F(x, y, y') \, dx.$$ 

We thus wish to find a function $y(x)$ such that the above integral is minimized or maximized subject to the constraint

$$G(x, y, y') = 0.$$ 

Here, this is done by forming the Lagrangian function

$$L(x, y, y', \lambda) = F(x, y, y') + \lambda G(x, y, y')$$

and solving the Euler-Lagrange equation:

$$\frac{d}{dx} \left( \frac{\partial L}{\partial y'} \right) - \frac{\partial L}{\partial y} = 0$$

together with the constraint $G(x, y, y') = 0$ gives an optimum for the functional $I[y(x)]$ under the given constraint. Consider the following brief

Example 6.14.1. An agitated tank contains a mass $m$ of water at temperature $0^\circ C$. It is desired to raise the temperature of the water to $40^\circ C$ over a time $T = \sqrt{2} \, h$ by feeding hot water into the tank at a rate of $m$ per hour, at the same time draining the tank continuously at the same rate of $m$ per hour. The inlet and outlet temperatures, as functions of time, are $T_1(t)$ and $T_2(t)$, respectively. The inlet temperature is controlled by an adjustable electric heater inside the feed pipe and an optimal strategy for the heating is achieved by minimizing the functional

$$I[T_1(t), T_2(t)] = \int_0^{\sqrt{2}} \left( [T_1(t) - 0]^2 + [T_2(t) - 0]^2 \right) \, dt.$$ 

By applying an energy balance over the tank, we can derive the constraint

$$\frac{dT_2(t)}{dt} = T_2(t) - T_1(t).$$

The initial- and end-conditions for the outlet temperature are $T_2(0) = 0 \, ^\circ C$ and $T_2(\sqrt{2}) = 40 \, ^\circ C$, respectively.

We note that the general form $G(x, y, y') = 0$ of the constraint is $G = \frac{dT_2(t)}{dt} - T_2(t) + T_1(t)$ and the Lagrangian can be formulated as

$$L[T_1(t), T_2(t), \lambda(t)] = [T_1(t)]^2 + [T_2(t)]^2 + \lambda(t) \left( \frac{dT_2(t)}{dt} - T_2(t) + T_1(t) \right).$$
6.14. SPECIAL TOPICS

Based on this function, we have the Euler-Lagrange equations:

\[
\frac{d}{dt} \left( \frac{\partial L}{\partial T_1'} \right) - \frac{\partial L}{\partial T_1} = 0 ,
\]

\[
\frac{d}{dt} \left( \frac{\partial L}{\partial T_2'} \right) - \frac{\partial L}{\partial T_2} = 0 .
\]

**Exercise 6.14.1.** Combine the Euler-Lagrange equations and constraint equation in the example above to derive a linear ordinary differential equation for the outlet temperature \( T_2(t) \). Using the conditions \( T_2(0) = 0 \, ^\circ C \) and \( T_2(\sqrt{2}) = 40 \, ^\circ C \), formulate a boundary value problem and solve it. Use the solution to obtain the inlet temperature \( T_1(t) \).

6.14.3 JWKB-approximation

Many applications of mathematical physics can be modeled by the general wave equation

\[
\rho(x) \frac{\partial^2 y}{\partial t^2} = \frac{\partial}{\partial x} \left[ T(x) \frac{\partial y}{\partial x} \right] ,
\]

where \( y \) is a spatial variable, \( t \) is time and \( \rho(x) \) and \( T(x) \) are called the density and transfer coefficient, respectively. A classical approach to solving this type of partial differential equation (PDE) is the method of separation of variables, which amounts to seeking special solutions of the form

\[
y(x, t) = f(x) g(t) .
\]

Substituting this into the PDE yields the equality

\[
\frac{1}{g(t)} \frac{d^2 g(t)}{dt^2} = \frac{1}{\rho(x) f(x)} \frac{d}{dx} \left[ T(x) \frac{df(x)}{dx} \right] .
\]

Now, we note that the left-hand side of this equation is a function of time \( t \) only, whereas the right-hand side is a function of \( x \) only. As the equality nevertheless must be valid for all values of \( x \) and \( t \), it must be concluded that the expressions on both sides must be constant. Setting this constant equal to the negative number \(-\omega^2\) leads, for the right-hand side, to the equation

\[
\frac{d}{dx} \left[ T(x) \frac{df(x)}{dx} \right] + \omega^2 \rho(x) f(x) = 0 .
\]

If \( T \) and \( \rho \) are constant, we have a solution of the form

\[
f(x) = Ae^{i\omega x} ,
\]

where all the new names denote constant parameters. Keeping this in mind, for constant \( T \) and slow variation in \( \rho(x) \), we try a solution of the form

\[
f(x) = A(x)e^{iS(x)} .
\]
Substituting this into our differential equation for \( f(x) \) and cancelling the exponential gives the equation

\[
A'' + 2iS'A' + iS''A - S'^2A + \left( \frac{\omega^2}{u^2} \right) A = 0 ,
\]

where \( u^2 = \frac{T}{\rho(x)} \). Taking real and imaginary parts of the above equation leads to the two equations

\[
2S'A' + S''A = 0 ,
\]

\[
A'' + \frac{\omega^2}{u^2} A - S'^2A = 0 .
\]

If \( u(x) \) is slowly varying, the derivatives of \( S \) as well as \( A \) will be small compared with the functions themselves. In particular, in the second equation above, \( A'' \ll \frac{\omega^2}{u^2} \), which means that it can be neglected. Integrating the remaining equation yields

\[
S(x) = \int^x \frac{\omega}{u(p)} \, dp .
\]

The lower integration limit is arbitrary, but is sometimes for convenience taken to be zero. There remains to compute the factor \( A(x) \). This can be done from the first equation if we note that multiplication by \( A \) gives

\[
2AA'S' + S''A^2 = 0 \iff \frac{d}{dx} \left( A^2S' \right) = 0 \iff A^2S' = \alpha^2 ,
\]

where \( \alpha^2 \) is a positive constant. Thus, \( A(x) = \text{constant} \cdot \sqrt{u(x)} \). The real part of our solution to the differential equation is hence

\[
f(x) = \text{constant} \cdot \sqrt{u(x)} \sin \left( \int^x \frac{\omega}{u(p)} \, dp \right) .
\]

**Exercise 6.14.2.** Consider a vibrating string stretching from \( x = 0 \) to \( x = L \), with the density

\[
\rho(x) = \rho_0 + \epsilon \sin \frac{\pi x}{L} ,
\]

where \( \rho_0 \) and \( \epsilon \) are constant parameters with the property \( \epsilon \ll \rho \). Use a Taylor-series expansion for the root \( \sqrt{1 + z} \) in order to compute the frequencies

\[
S(L) = \int_0^L \frac{\omega}{u(p)} \, dp = \omega \sqrt{\frac{\rho_0}{T}} \int_0^L \sqrt{1 + \frac{\epsilon}{\rho_0} \sin \frac{\pi x}{L}} \, dp .
\]
6.14.4 Monte Carlo Methods

Monte Carlo methods can be considered statistical simulation methods based on generating sequences of random numbers. The term “Monte Carlo” was coined by Nicholas Constantine Metropolis (1915-1999) and inspired by Stanislaw Ulam (1909-1986), because of the similarity of statistical simulation to games of chance, the city of Monte Carlo being a worldwide center for gambling. As may be anticipated, the efficiency of these methods is heavily dependent on the efficiency and quality of producing machine-generated random number sequences. Hence, a substantial part of the research in the field is focused on these objectives. A popular application for the Monte Carlo method is computation of multidimensional integrals, for which standard methods of discretization can become computationally expensive. For example, in $d$ dimensions the error of the composite trapezoidal rule for numerical calculation of an integral converges as $O\left(\frac{1}{n^d}\right)$ when the number of discretization points is $n$. The corresponding error for the Monte Carlo method can be shown to be $O\left(\frac{1}{\sqrt{n}}\right)$. Thus, the inequality

$$O\left(\frac{1}{\sqrt{n}}\right) < O\left(\frac{1}{n^d}\right), \quad d > 4.$$  

This tells us that Monte Carlo integration pays off for five dimensional (and higher) integrals

$$\int_V f(w) \, dw_1 dw_2 dw_3 \cdots dw_d.$$ 


(i) Generate $n$ randomly distributed points, $x_1, x_2, \ldots, x_n$ in the “hypervolume” $V = [a_1, b_1] \times [a_2, b_2] \cdots [a_d, b_d]$. That is, these points are “$n$-tuples” of coordinates within the volume.

(ii) Determine the average

$$\bar{f} = \frac{1}{n} \sum_{i=1}^{n} f(x_i).$$

(iii) Compute an approximation to the desired integral

$$\int_V f(x) \, dV = (b_1 - a_1) (b_2 - a_2) \cdots (b_d - a_d) \cdot \bar{f}.$$ 

(iv) An error estimate is given by

$$\text{error} = (b_1 - a_1) (b_2 - a_2) \cdots (b_d - a_d) \cdot \sqrt{\frac{\bar{f}^2 - \bar{f}}{n}}.$$
Exercise 6.14.3. Compute the multidimensional integral

\[ \int_0^7 \left( \int_0^5 \left( \int_0^9 \left( \int_0^{11} \left( 6 - x^2 - y^2 - z^2 - u^2 - w^2 \right) \, dw \right) \, du \right) \, dz \right) \, dy \, dx \]

(a) by Monte Carlo approximation.

(b) analytically.

Moreover, compare the results in (a) and (b) and obtain an error estimate for the result in (a).

6.14.5 Coding Theory

Every book published has a so called ISBN-code, consisting of a string $a_1 a_2 \ldots a_{10}$ of integers with the property $0 \leq a_i \leq 9$ if $1 \leq i \leq 9$ and $0 \leq a_{10} \leq 10$. However, instead of the number 10, the greek number $\chi$ is used. The element $a_{10}$ is a checksum calculated by the formula

\[ a_{10} = \sum_{i=1}^{9} 9a_i \pmod{11}. \]

Show, e.g., by direct calculation that if two digits $a_j$ och $a_k$ are interchanged or if one digit $a_j$ is changed, the checksum will be disrupted and become erroneous.