University of Twente
The Netherlands

msm

Programming in Engineering

Stefan Luding, Holger Steeb & Katja Bertoldi

Multi Scale Mechanics – MSM, TS, CTW
University of Twente
Stefan Luding, Holger Steeb, Katja Bertoldi

Preface

The manuscript contains a short introduction into the interpreter language MATLAB and the object-oriented compiler language C++ extended by different small and larger exercises.
It consists of two disjoint parts compiled as two chapters: The C++ and the MATLAB chapter. Therefore, students who are interested in self-study, could begin either with the C++ or the MATLAB part. Within both chapters, you will find small examples. These examples can be used for self-evaluation. At the end of the C++ and the MATLAB chapter, you will find various larger exercises similar to the exercises of the final assignment.

History

Version 1.0 The manuscript is compiled for a 10-days summer course *Programming in Engineering*, 7 July - 18 July 2008.

Version 1.1 Update of the manuscript for a self-study purposes, 3 September 2008.

Enschede, 3 September 2008

Stefan Luding
Holger Steeb
Katia Bertoldi

Contents

1

Introduction to C/C++

The goal of this short course in C/C++ is not to become a perfect C/C++ programming guru. For this we refer to special course and a broad range of literature – see below.

The goal of this introduction to C/C++ is how to perform basic tasks, operations, and programming within the framework of the programming language C/C++. Goal of this course is to make you understand simple C/C++ programs, allow to write own code, and to get used to debugging of code.

First of all you need a C++ compiler. On UNIX, LINUX and MAC systems this should be installed already. On MS-windows you have to follow the instructions in section 1.14.

1.1 A short history of C/C++

C was developed in the 70's by K. Thompson, B. Kernighan, and D. Ritchie, and was introduced together with the operating system UNIX. Today, LINUX is the free derivate of UNIX, and like its ancestor is based on C, i.e., the operating system is – in big parts – written in C. The original goal was a standardized language for the structured programming. However, the so-called ANSI-standard was only introduced as late as 1988 as ANSI-C.

C++ can be seen as an extension of C, developed, propagated and introduced between 1983 und 1985 by B. Stroustrup. Besides the goal to develop a higher, object oriented language, down-ward compatibility with C was desired. Therefore, C is to be seen as a fraction/part of C++. The ANSI Standard for C++ was only fixed in June 1998. In the

framework of this course, we will use C++ syntax whenever this is simpler as C, e.g. for I/O (input/output).

Some reasons for the use of C/C++ are

- it is free! (e.g. g++ compiler or DevC++ environment)
- it is system-independent – applicable also on super-computers
- it is rather low level – close to the operating system
- it is extendable, re-usable, and robust, ...

1.2 Programming-style

This section briefly discusses some issues related to a good style of programming. Good style means that programs are simple and easy to read. It involves some rules like intendations that make the program also optically structured and, finally, it ivnvolves comments that describe in words what the program is doing.

Note that a good style does not only help other people reading and understanding your program, it also will help you to understand your own program – after you have not looked at it for a while. Think not of the few 100 lines we will write in this course, think of several 10,000 lines of code that a programmer can write during a thesis for example.

In "real life" people do NOT spend most of the time with programming itself, but with debugging (error-search), updates, changes, optimization and re-using of existing codes and code-fragments. For these tasks, a clean and good style are not only helpful, but absolutely necessary.

This is – by the way – true for all programming languages.

1.2.1 Comments

There are two things a program should do: First, it tells the computer what to do and, second, it also tells the (human) reader of the program what it does.

A working program without comments is a time-bomb. Even the simplest ideas flowing into the program will be forgotten after some time ... and without comments, a program can not be re-used. Without comments, a program cannot be understood or changed so that either one has to program anew, or one has to spend a lot of time to understand an existing program without comments – both is a waste of time.

In C++ there are two types of comments.

The first, traditional comments begin with `/*` und end with `*/`. They can enclose singel characters or many lines and are most convenient for larger comments, or for making old code invisible to the compiler.

The alternative comment begins with two slash `//` and ends automatically at the end of a line. It can start anywhere in the line and is thus convenient for commenting on single commands.

As a suggestion, each program should contain at the beginning:

- ***Header***
  Name of program and what is it supposed to do.
- ***Author***
  Author and (e-mail) contact address
- ***Date***
  Date of changes
- ***How to use the program***
  Here the mode of usage is described
- ***Files (I/O)***
  Iinput data file format and output data-files
- ***Restrictions***
  Are there, e.g., restrictions to the input-data-files?
- ***Revisions***
  Dates and type of changes performed
- ***Error-management***
  What happens when a program detects an error
- ***Other remarks***
  ...

1.3 Clear style

A program should be read like a thesis or a scientific paper. It should contain an introduction, like the header-comments above, and it should be clearly separated in chapters, paragraphs, etc. with clear boundaries. Comments can be used as a way to optically separate paragraphs within the program.

1.4 Language Elements

The first language element are *comments*. They do not affect the function of the program or the computer – they enhance the understanding of the program when read by another (human) user. A comment is everything between '`/*`' and '`*/`' or after '`//`' steht.

The second essential element are *data*, i.e., counters, physical constants, variables, etc. It is necessary to first tell the computer which data are required for the program to work. This could be done anywhere in the program, however, it is neither efficient nor clean or clear programming. It is recommended to define data (only) at the beginning of a function/program. Exceptions to this rule will be discussed below.

Operators and *Functions* are required to manipulate the data. Examples for operators are the basic computations `+`, `-`, `*`, or `/`. However, there are many more as summarized in the following table and discussed in more detail later in this script.

expression	short-form for:
`x += y`	`x = x+y`
`x -= y`	`x = x-y`
`x *= y`	`x = x*y`
`x /= y`	`x = x/y`
`x %= y`	`x = x%y`

Table 1.1: Short expressions and the long form they replace

Examples for functions are the trigonometric functions `sin` and `cos`, but functions can also be user-defined groups or sequences of operations that are repeated many times. Use of functions makes the program shorter and better readable.

Note that the so-called main program `main()` is also a function – it is *always* called `main`. (That means no other function can use this name.) Begin and end of a function are marked by the curly brackets '{' and '}'.

The next class of language elements are those used for the communication between the user and the computer, the so-called I/O (input/output) functions and operators. In C/C++ normed libraries exist already that contain the definitions of the I/O commands.

The simplest way of I/O is output on the screen and input via the keyboard. The next way – especially when a lot of information has to be input into the program or is returned – is via data-files (for both input and output).

Finally, there are *organisational* elements, that do not per-se belong to the programming language, but can be used for management of the program *before* compilation and execution, e.g., by the so-called pre-processor. This allows to shorten and clarify the program by just moving big, un-clear code-fragments out of the program, or by conditinally selecting alternative parts of the code.

operator	evaluation from
::	left to right
() [] -> . X++ X-- typeid XXX_cast<TYPE>()	left to right
! ~ ++X --X + - * & (TYPE) sizeof new delete	right to left
.* ->*	left to right
* / %	left to right
+ -	left to right
>> <<	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
= *= /= %= += -= <<= >>= &= |= ^=	right to left
?:	right to left
throw	–
,	left to right

Table 1.2: Priority and associativity of operators

The first program (DevC++ New Program) looks like:

```
#include <cstdlib>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    system("PAUSE");
    return EXIT_SUCCESS;
}
```

- The first lines include libraries `#include <iostream>` and `#include <cstdlib>`, get the definitions from the standard libraries for I/O and standard C-routines and functions. The former are required for input and output of text and numbers, the latter contain various helpful and necessary tools. This line must be situated at the beginning of each main-programm – outside of the `main()...` function. This libray contains operators/functions like '`cin`', '`cout`', '`<<`' or '`>>`'.

- The command `using namespace::std` imports all symbols (like `cin` or `cout`) from namespace `std`. Such namespaces in C++ are used, to separate algorithms from different libraries. A command `cout` from another namespace, e.g., a self-defined

one, cannot be used anymore. This is supposed to help against confusing names for functions: in principle, C++ allows "overloading"; event though this provides an enormous freedom, it is better to avoid it in order to avoid confusion. In general, we will use the C++ standard libraries, whereas for using C-functions, the command `using` is explicitly needed.

- The command `main(...)` is a function of type Integer `int`. Every progam contains at least one such function (there can be much more functions, but only one special "main". The terms in brackets are used to transfer parameters to the function main, however, we delay the discussion of this to later.

- `return EXIT_SUCCESS;` returns the value of 0 to the calling program, i.e., to the operating system. `EXIT_SUCCESS` is defined in the standard library as integer with value zero. Zero is the common return value if the function finished without errors.

- The brackets `{...}` enclose the main-program, i.e. the commands, the main function `main(...)` consists of. In general, program blocks are enclosed by these curly brackets, and each begin-bracket `{` must be accompanied by an end-bracket `}`.

- The command `system("PAUSE");` writes the word "Pause" to the screen and then waits for a key-stroke from the user. This is a convenient way to stop the program before it completely ends and disappears from the screen.

- Note that *every* command has to be ended by a semicolon '`;`'. This is because a new line alone is not the beginning of a new command or program element. It is better to not use several '`;`' within a single line in order to keep the program clean and readable. Since the semicolon has the function to separate two commands, it is *not* found at the end of program-blocks, that are already clearly terminated by the end-bracket of the pair `{...}`.

1.5 Data-types, Variables, Constants

1.5.1 Numbers

Computers process numbers (and also symbols) in binary representation. The binary system was developed by G.W. Leibniz when studying Chinese letters. Based on the binary representation, also octal (3 bits) or hexa-decimal (8 bits) representations are used in the computer.

Example – if you really want to know:
The number 496 can be represented in the following ways (where the subscript indicates

the type of representation, and the superscipt are mathematical powers):

$$\begin{aligned}
496_{10} &= 4\cdot 10^2 + 9\cdot 10^1 + 6\cdot 10^0 \\
&= 1\cdot 2^8 + 1\cdot 2^7 + 1\cdot 2^6 + 1\cdot 2^5 + 1\cdot 2^4 + 0\cdot 2^3 + 0\cdot 2^2 + 0\cdot 2^1 + 0\cdot 2^0 \\
&= 111110000_2 \\
&= 7\cdot 8^2 + 6\cdot 8^1 + 0\cdot 8^0 \\
&= 760_8 = \texttt{0760} \quad \text{(in C++)} \\
&= 1\cdot 16^2 + 15\cdot 16^1 + 0\cdot 16^0 \\
&= 1F0_{16} = \texttt{0x1F0} \quad \text{(in C++)}
\end{aligned} \tag{1.1}$$

The number 5 is much simpler:

$$\begin{aligned}
5_{10} &= 5\cdot 10^0 \\
&= 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 \\
&= 000000101_2 \\
&= 0\cdot 8^1 + 5\cdot 8^0 \\
&= 5_8 = \texttt{0005} \quad \text{(in C++)} \\
&= 0\cdot 16^1 + 5\cdot 16^0 \\
&= 005_{16} = \texttt{0x005} \quad \text{(in C++)}
\end{aligned} \tag{1.2}$$

Numbers x in floating-point representation (`0.496e+03`) are transformed to fixed-point notation using the formula

$$x = mb^e \tag{1.3}$$

with the "mantissa" m (here `496`), the basis b ($b = 10$ could be also $b = 2, 8, 10, 16$), and the exponent e (here `3`). The in-fact implementation of floating-point numbers is machine/processor-dependent and will not be discussed here.

Exercise 1 is related to this paragraph

1.5.2 Basic data-types and their declaration

The most basic built-in data-types in C++ are: `bool, char, int, float, double`. Historically the processors were optimized to deal with those – however, the development of processors has progressed so much that also other data-types are efficiently treated. Usually, a compiler can optimize a code such that a user does not feel anymore the difference between different types of data.

The type `bool` is a logical data-type that can be used, e.g. to deal with the result of a logical comparison using logical operators, see Sec. (1.7.3). It can only have two values

`true` or `false`. Furthermore, its freely transformable to data-type `int`, where `true` coore-sponds to 1 and `false` to 0. This transformation was introduced for compatibility reasons, to old programs where instead of `bool` just `int`'s was used. Some simple declarations are:

```
bool    test;                  // declare test
test  = (0 < 5);               // comparison with result true
                               // test is set to true
int     i(test);               // i is declared and set to 1
int     j=0;                   // j is declared and set to 1
```

Initialisation/Constuction of a variable can be performed as *type variable = type_constant* or equivalently as *type variable(type_constant)*. (*For experts:* The equal symbol '=' here is not the setting of the variable, since this was just created; technically, here is *not* the set-to Operator '=' called, but a copy-operator.)

The type `int` is that type that is recognized by the compiler (i.e. the compiler-programmer) as most appropriate for dealing with integer numbers. Its size is not fixed in the standard. However, with help of the operator `sizeof`, one can determine the size of a type `int` in units of a data-type `char`, which is also the smallest data-type (again, selected by the compiler as most appropriate for single letters).

Different computers also support further integer data-types, e.g., 16, 32, or 64 bits large. These are defined using `short` and `long` oder (not ANSI conform) `long long`. The longer form of declaration `short int` and `long int` is not required. Note that an integer variable always has an integer value – even if one assigns a real (floating-point) value to it.

Examples for `int`eger declarations are:

```
bool    test;                  // declare test
test  =(0 < 5);                // (see above)
int     i(test);               // i is declared and set to 1
int     j=0;                   // j is declared and set to 0
long    l, n, m;               // l, n, and m are declared "long"
short   s1, s2;                // s1 and s2 are declared "short"

cout << sizeof(test) << " "
     << sizeof(i) << " "
     << sizeof(l) << " "
     << sizeof(s1) << endl;    // check size of: test, i, l, s1

i = 37;
j = 38.5674;                   // ATTENTION
cout << i << " " << j << " " << i/j << endl;
                               // this leads to output: 37 38 0
```

The data-types `float` and `double` are the single- and double-precision floating point

data-types that are implemented optimally in the processor. `long` can be used to modify `double` in order to get quad-precision (if supported). sollte). The in-fact representation of floating-point numbers is hardware dependent, e.g., in 1998 at least the double-precision basic operations were hard-ware implemented. Nowadays, with 64-bit processors more and faster operations with larger numbers are possible. Type `float` should only be used if memory is a problem, since for the operations, often `float` has to be modified to `double`.

C/C++ does not specify how variables are represented in memory and the use of memory is not specified per type either. However, at least the relations:
`1 = sizeof(char) = sizeof(bool)`
$\leq$ `sizeof(short)` $\leq$ `sizeof(int)` $=$ `sizeof(unsigned)` $\leq$ `sizeof(long)`
and
`sizeof(float)` $\leq$ `sizeof(double)` $\leq$ `sizeof(long double)` are always true.

In general, on UNIX-machines, `int`-variables are 32, `short` 16, and `char` 8 bit long. Double-precision `double` requires 64 bit, and `float` only 32 bit. The values that are used are different from machine to machine, but also can be different from compiler to compiler.

Homework:
Find out the size of variables of different types on your computer.

In the following example, different data-types are declared:

```
#include <string>
using string;              // Define this at the very beginning

char       c1 = 'x';       // one character: 'x'
char      c2[] = "hello";  // text array with 5 characters
                           //   (note: length is 6 due to added '\0')
string     str("hello");   // the C++ way, arbitrary length strings
                           //   (note: length is 5 - no added '\0')
char       c3 = '\n';      // the new-line character
int        i_var;          // integer variable with name i_var
long int   il_var = 1;     // long constant
long int   jl_var = 1L;    // another long constant
short      is_var;         // short integer variable (int per default)
double     d_var = 34.2;   // real number 34.2 with double precision
```

Note that for `char` variables single characters can be enclosed by two apostrophs `'`, where as longer chains of characters must be enclosed by a double apostroph `"`. Above, the variables `c1` und `c3` both have lenght unity (1), whereas the chain `"hello"` is accessible in the "field" `c2` and has length 6, since during its declaration, automatically, a symbol `'\0'` is added. (*Try the definition* `char c2 = 'hello';` *that will lead to an error.*) More flexible is the C++ method to use `string`, which do not need the terminator `'\0'`. Note

that for strings, the standard-library `string` must be included, see above.

Declaration of variables is not limited to the beginning of a block, but can be done everywhere at first occurence of a variable – be it clear, useful, and efficient or not.

Important and useful is the declaration within a `for(;;)` loop, as discussed in more detail below:

```
double a;
cin >> a;              // input a via keyboard
int    i_trunc = a;    // throw away the fractional part
for ( int i=0; i < i_trunc; i++ )
{
   ...
}
```

In the above program, the variable `i` is outside of the `for(;;)` loop not existent. i.e., neither before nor after.

Summary
(i) Declarations can be located at arbitrary position in the program
(ii) Initialisation with `()` oder `=` form
(iii) `sizeof(type)` can be used if the size of a type must be known, e.g., for portability between different architectures.

1.5.3 Variable-attributes

Besides the attributes `short` or `long` integer variables can be also modified by `signed` and `unsigned`, that allow to use integers (positive or negative) or positive integers, respectively. The `int` can be dropped here.

1.5.4 Life-time and position in memory

`auto` If there is no further declaration, variables are `auto`matic. This means they are constructed as soon as the program reaches the line where they are declared. As soon as the program leaves the block where they are declared, the memory is freed again. (The block is, in general, the inner-most enclosing pair of brackets, or like in the case of a `for(;;)` loop, the end of this object.) *For experts: Variables of this type are usually assigned by the compiler to the so-called stack.* The value of such a variable is random! if not explicitly assigned or initialized. Thus *make sure to initialize* your variables!

`static` The keyword `static` makes a variable exist during the whole duration of the program. *For experts: Such variables are saved on the heap-memory.* If not otherwise initialized, they get the value 0 (zero). A `static` variable inside a function has thus the value 0, if the function is called for the first time, and keep this (or another assigned value) between subsequent calls to this function. In contrast, an `auto`matic variable is every time newly created and initialized. The following function counts how often it was called and returns this value.

```
int f_count_calls(){
  static int count;  // 0 at first function call
  return ++count;    // increment before use as return value
                     // (i.e. returns one on the first call)
}
```

`const` Variables that are decleared `const`, can only be assigned a value at initialization and not anymore thereafter. This is used to define constants within a namespace that – by the compiler – can be optimized such that their use costs less time than using variables. Thus, if variables can be defined as `const`ants, this might lead to a faster program execution.

```
 const int    Size = 100;
 const double Pi   = 3.1415926;   // Self-defined value of PI
                                  // When including cmath one can
                                  // use the predefined constant M_PI
                                  // see FirstSteps3.cpp
```

1.5.5 Fields

Fields are used to combine variables of the same type in a connected range of memory. *For experts: Access to variables can be faster if they are in such a connected range of memory and, on the other hand, using fields also can make a program shorter and more clear.* Access to the elements of a field is achieved by an integer index. A linear field (with one index) can be seen as a vector, while a field with two indices represents a matrix. Mathematical formulas with indices can often be represented in a program using fields.

A field is declared in C/C++ by adding brackets `[]` to the variable name. At the same time, this operator also is used to access the elements of the field, see below. In C/C++ fields always start with the index 0, so that the last valid index is the length of the field reduced by 1. Multi-dimensional fields are declared by multiple (not enfolded) pairs of brackets, see below.

Such fields receive – at declaration – a fixed size. Thus, the argument in brackets `[]`, must be either a number, a `const` variable, or a well-defined variable with a known value for the compiler (at compilation time).

```
#include <string>
using string;

// array of 20 int's
int  a[20];

// array of three strings, initialized to some values:
string  stra2[] = { "Katia", "Holger", "Stefan" };

// two dimensional array of 20x30 elements of int type
int  ia[20][30];
// initialization of field ia
for ( int i=0; i < 20; i++ )
  for ( int j=0; j < 30; j++ )
    ia[i][j] = i*j;
```

Later, we will see that fields and pointers are closely related. The use of pointers allows to define "dynamic" fields of a size that is determined during run-time, e.g., following a user-decision and -input.

1.6 Control-structures

For implementing algorithms into a certain programming language, this language has to provide certain control-structures that allow repeated or conditional execution of parts of the program.

In C (and C++) the most-used control strucures are the `if`-statement for the conditional execution, and `for`-loops for a repetition of a block of commands. Alternatively, the rejecting `while()`- and the accepting `do{...}while()`-loops exist, as well as the `switch()...case`:-statement, that jumps to one of many blocks of commands, based on the value of a certain variable. Finally, there is the command `goto`, that sometimes can be used to escape from (deeply nested) loops.

1.6.1 The if-command

The `if`-command is used to execute parts of the program only under certain conditions. This command can be (but does not have to be) complemented by an `else`-statement, that is visited/exectuted in case the condition in the `if`-command is `false`. The `else`-part can consist of arbitrarily many `else if` and one `else`. As well `if`, `else if`, and `else`-branches can be single lines (terminated by a semicolon) or blocks, enclosed in curly brackets `{}`. An `else`-block always relates to the previous, actual `if`-command, i.e., also

if its part of an `else if`. In the following example, the second `if`-command is a statement by its own, outside the `else`-part of the first `if`.

```
#include<iostream>
#include<cmath>
// ...

  // solution of quadratic equations  x^2 + p x + q = 0

  double p=2., q=3.;              // or other values
  double sol1, sol2;
  double root_arg = 0.25*p*p - q; // calculate the argument for the root

  if ( root_arg > 0. )
  {                               // begin of if-block
    sol1 = 0.5*p + sqrt(root_arg);
    sol2 = 0.5*p - sqrt(root_arg);
  }                               // end if-block
  else if ( root_arg < 0. )       // 'error' condition, 'else' part
    cout << "no real solution exists" << endl;
  else                            // refers to last if(root_arg < 0.)
    sol1 = sol2 = 0.5 * p;        // assignment from right to left

  if ( sol1 > 0. )
    cout << "graph cuts positive x-axis";
                                  // ok, else branch missing
```

1.6.2 The ternary ?: operator

For completeness, also the ternary `?:` operator is treated here. Ternary means that this operator requires three arguments and is used to conditionally evaluate parts of expressions.

```
inline int abs(int number) {
  // compute absolute value
  return number > 0 ? number : -number;
}
```

Is the term before the question-mark true, then the term before the colon is examined. Otherwise, the term after the colon will be used. The two alternatives must be of the same type. Note that here, the use of brackets is recommended – partly because the ternary operator is of low priority and also because the terms that belong together should be made clear.

```
int i1   =       3 > 4 ? 0 : 1;    //  i1   is 1
int i2   = 3 * ( 3 > 4 ? 0 : 1 );  //  i2   is 3
int i3   = 3 *   3 > 4 ? 0 : 1;    //  i3   is 0, since * binds
                                   //    stronger than >
```

This operator always can be avoided by using an `if`-command, however, it could be more clear or at least shorter in some situations. werden, kann aber in vereinzelten Situationen klarer sein. In any case it is a possibility to write un-readable code and therefore should be used sparsely.

(As final remark, the ternary operator is also embedded in the graphics program gnuplot.)

1.6.3 (do) while-loops

Loops that are constructed using `while` are executed until the condition in the condition part of the loop is not true anymore. Is the condition already false at first occurence of the `while`-loop, the loop will not be executed at all.

```
char c=0;                       // 'loop initialization'
while( c != 'y' && c != 'n' )   // first time c==0, so we enter the loop
cin >> c;                       // can also be a block in { }
```

The symbol/character variable has to be initialised such that the loop is entered. In such cases, the `do ... while(...)` loop might be more intuitive.

```
char c;
do {
  cin >> c;
} while( c != 'y' && c != 'n' );
```

1.6.4 for-loops

When working with fields, one often needs loops that access all elements of a field, one by one. One can program this with `while`:

```
const int Size = 20;
// ..
int a[Size];
```

```
// ..
int i=0;                // loop initialization
while ( i < Size )      // loop condition, rejecting if false
{
  a[i] = i;
  i++;                  // loop control, performed after each pass
}
```

A shorter alternative used the `for(;;)` loop. The following is almost equivalent to the above, but shorter and more clear.

```
int a[20];
// ..
for( int i=0; i < 20; i++ )
  a[i] = i;            // work on elements 0 .. 19
```

There are two advanteages: First, the loop-beginning, -ending, and stepping is nicely combined in one line and, second, the variable `i` is only defined within the loop and not used (and not needed) outside.

1.6.5 break and continue

In order to exit a single, not-nested `for` or `while` loop, one can use the `break` command. The loop is terminated and the program continues directly after the loop. Thus `break` can be used to terminate loops under certain conditions, earlier than the loop implies, or it can be used to exit "endless" loops.

```
// copy input to output
while( true )
{
  char c;
  cin >> c;
  if ( cin.fail() ) break;   // end of input or something wrong
  cout << c;
}
```

The command `continue` terminates the processing of the present block and continues, dependent on the type of the loop, at the corresponding control-command (`for(;;)`), or (`(do{}) while()`).

```
#include <cmath>              // for sqrt() function
```

```
  double a[20];
  // ...
  double sum_sqrt=0.;
  for( int i=0; i < 20; i++ )
  {
    if ( a[i] < 0. ) continue;   // work on next element
    sum_sqrt += sqrt( a[i] );
  }
```

Note again that `continue` and `break` only exit the innermost loop-block and cannot be used to exit nested loops.

1.6.6 goto-command, labels

In order to leave deeply nested loops, the `goto` command can be used.

```
void f()
{
  int a[20][30];

  for( int i=0; i<20; i++ )
    for( int j=0; j <30; j++ )
       if ( a[i][j] == 42. )       // found the answer
         goto label_end_of_loop;   // multi level break

label_end_of_loop:                 // label, we jump here from inside the loop
}
```

The label with name `label_end_of_loop` marks the jump-goal of the `goto`. Within a function the name can be freely chosen – terminated by a colon `:`. The problem with `goto` is, that it is not visible *from where* the jump to this label comes. Furthermore, using `goto` often requires further use – and thus further unclear structure of the program.

It is allowed to leave local blocks using `goto`, but one cannot enter a block inside an `if`-block. Neither is it possible to jump out of a function.

Since `goto`'s always can be avoided, it is better to do so – in some special cases a single `goto` could be helpful for better structuring the program.

1.6.7 switch-command

A command related to the `goto` is the `switch`-command. Dependent on the value of an integer variable, several `case`-blocks are visited. This construct is often used when there is a larger number of possible alternatives to be chosen from.

In principle, a `switch` can always be replaced by `if...else if`-constructs, but often the `switch` is more clear and helps understanding and structuring the program.

```
char c;
cin >> c;
switch( c )
{                              // begin case block
case 'a': cout << "hello world!";
          break;
case 'b': cout << "HELLO ";   // fallthrough intended, comment missing break
case 'c': cout << "WORLD!";
          break;
default:  cout << "unknown input";
          break;               // not necessary, but good style
}                              // end case block
```

The above program-segment returns in small letters `"hello world!"` if the letter `'a'` is typed. In case `c` the word `WORLD!` is printed, but in case `b` the full sequence `HELLO WORLD!`. This strange behavior comes from the missing `break` in the block `b`. This behavior, namely that a execution is performed until a `break`, is called ("fallthrough"). Since this is often not desired, using this option required a comment whenever a `break` is missing. The `default`-label ist optional and is visited if none of the other labels is valid in the `case`-list.

1.7 Functions and Operators

Longer programs usually contain parts or blocks that are repeated a certain (large) number of times. These parts can then be defined elsewehere (as a function) and within the program are replaced by a short-cut or place-holder, i.e., the name of the function. It is wise to give the `function` a name that already implies its function.

A function (also called procedure or subroutine in other languages), has input and output variables/parameters. In the simplest case the transfer of input-parameters is done as argument, and the output as a return-value.

C++ enforces that the types of input and output are specified. In the following example, the function `power` computes from a double `base` value and an integer number `exponent`

a new number in double precision that is returned to the calling function/program. The name already implies the algorithm used here.

```
// raise 'base' to the power 'exponent', with exponent being integer,
// base being double; accept also negative exponents and check ...
double power(double base, int exponent){
   double result  = 1.;    // will multiply this by base 'exponent' times
   bool   neg_exp = false;
   if (exponent < 0 )      // for negative exponents we use a single
   {
                           // division at the end of the function
      neg_exp = true;
      exponent = -exponent;
   }
   for (int i=0; i < exponent; i++ )
      result *= base;      // successive multiplication

   if ( neg_exp )          // have to take the inverse
      result = 1./result;

   return result;          // return the result to the calling program
}
```

Note that C++ already has a function `pow` that is described by
`double pow(double base, double exponent)`, only different in the type of the exponent.

The names of the arguments are free to be chosen. Inside the function block these names are to be used. Note that like for loops, there is no semicolon after the function.

The keyword `return` terminates the function at the specified position, moves the value of the variable specified after it on the stack, and returns to the calling program. In the case above, the expression consists only of a single variable `result`. Round brackets are not required here. A subroutine can contain several return keywords at different places in the function block. This can be a useful tool to deal with problems/errors.

Finally, one has to define or declare a function before it can be used in the program. This declaration is essentially the first line of the function (with semicolon). This so-called prototype tells the compiler the name of the function and the types of it and its arguments.

```
double power(double base, int exponent);
```

At first definition, the command block is replaced by the single semicolon. Note that the type is required, but the name of the arguments in the function is not needed here. However, a clear program specifies variables that tell something about their purpose.

Declarations can (in principle) be repeated arbitrarily often in a program – they are a promise to the compiler that the function will be defined later. Unlike variables, functions are always global – it is not allowed to define them inside a block with limited existence.

A program that reads a number from the keyboard and then uses the function `power` to finally write the result could look like this.

```
#include <iostream>
using namespace std;

double power(double base, int exponent);  // declaration of 'power'

int main()                                // definition of main
{
  double in;
  cin >> in;                              // read value from keyboard

  cout << "-3rd to 3rd power of " << in << ": ";
  for( int i= -3; i <= 3; i++ ){
    cout << power(in, i) <<  " ";         // call power and use return value
  }
  cout <<  "\n";

  return 0;                               // return to operating system
}

double power(double base, int exponent)   // definition of power
{
  double result = 1.;
  // ... program text as above

  return result;
}
```

The transfer of arguments to a function takes place "by value", i.e., before the function is called, the arguments are copied and the function uses these copies. This way, the variables that contain the values that are transferred to the function can not be changed. If this is desired, e.g., because one return value is not enough and many variables are to be changed in a function, then one has to use pointers as disucssed later in section 1.8.

1.7.1 Functions without arguments or return value – void

The keyword `void` can be used like a type, but is not really a type. It is useful to mark functions that don't have a return value. Examples are functions that just write something to the screen or to a file and functions that modify the input-variables directly.

```
void error(string message){
  cout << "error occured: " << message << "\n";
}
```

Also functions without arguments sometimes can be useful. In C++ this is implied by an empty list of arguments like for the pseudo-randomnumber generator as declared in the C library `stdlib.h`:

```
  void srand(unsigned int seed);
  int  rand(void);
```

It is used by first selecting a sequence of random numbers by calling (once only) the initialization function `srand(.)` with a positive number. Then, for each call of `rand()` a new, integer number is returned.

1.7.2 Libraries

As seen above, declarations/prototypes are sufficient for the compiler to translate a program. Whereever called in the program, the function-call is prepared during compilation. The function itself can be then defined and "linked" to the program later. Function-definitions are found in so-called object-files (*.o) or libraries.

The classical example for a library is the `cmath` file that was incuded at the beginning and involves functions like the `sqrt()`. The declarations are found in `cmath` itself, and the translated/compiled definitions are found in `libm.a`. A program-block that uses the `sqrt()` function can look like this.

```
  double  d;
  cin >> d;
  if ( d < 0. )
    cout << "cannot compute the square root of negative numbers\n";
  else
  {                          // compute sqrt(d) and print value
    cout << "sqrt(" << d << ") = " << sqrt(d) << "\n";
  }
}
```

The compiler must contain the linker-option "-lm" and must know the correct path where it can find the library. As a remark, under LINUX, the compilation would look like this:

```
g++ myprog.cc -o myprog -lm
```

In DevC++ the corresponding command can be found under "Compilation".

The short `-l` means "library" and `m` is the abbreviation for the file `libm.a`, that one gets by adding `lib`, `m` and the ending `.a`.

1.7.3 Operators

Operators allow us to make a program readable. Sometimes, unfortunately, they also make a program un-readable, cryptic and unclear. A term like `3 * z + 5` is easily understandable, whereas the term `plus(mult(3,z),5)` that is expressed using functions is much less clear.

There exist uniray, binary, and ternary operators, see sec. 1.6.2). Binary operators like the multiplication- and addition-operators, have two arguments, while unary operators, like the adress-operator `&` require only one.

Arithmetic operators

The arithmetic operators are `+, -, *, /` and `%`. The Operator `%` (modulo) forms the integer rest of the division of the left with the right operand (both integer). If both are positve, the result is positive and strictly smaller than the right operand. For one or two negative operands, the result depends on implementation.

Combining the above operators with the '`=`' allows to repeat the left operand, as for example:

```
int i=3, j=7;      // initialize
i    = 3 + j % i;  // i = 3 + (j mod i),   i is set to 4
i   += 3 * 4 + j;  // i = i + (3 * 4 + j), i is set to 23
i   /= 5;          // integer division, i is now 4
i   /= 5;          // integer division, i is now 0 !
```

The type that is returned by an operator expression depends on the type of the operand. If both operands are type `T`, also the result is type `T`. For different operand-types the "better" type wins; one operand is first transformed to the "better" type, then the operation is carried out. C/C++ has the following "type promotions" durch:

```
bool -> char -> short -> int -> long
             -> float -> double -> long double
signed -> unsigned  (!)
```

These rule of type-transformation are also valid for the following operators.

The unary operators `++` and `--` exist in postfix and prefix-form, and they lead to an increment or decrement by one, respectively. In the postfix form, the value of the operand *before* the operation is returned, whereas in the pre-fix form, the value *after* is returned.

```
int i=5, j;
j = i++;        // j is 5 now (i before increment) i is incremented to 6

double a[20];
i=j;                                // i=j=5
cout << a[++j] << a[j++];   // prints a[6] twice,    j is now 7
cout << a[i++] << a[++i];   // prints a[5] and a[7], i is now 7

// j++ = i;     // ERROR: cannot assign to a temporary
// i = (j++)++; // ERROR: cannot apply ++ to a temporary
```

Comparison- und logical operators

For operations between two integral datatypes, there exist bitwise, logical, and shift operators. (C++ also provides keywords for these). These operators are summarized in table 1.3, and they also can be combined with the `=`.

`|`	`bitor`	bitwise OR
`&`	`bitand`	bitwise AND
`^`	`xor`	bitwise XOR
`<<`		left-shift, n-times
`>>`		right-shift, n-times
`~`	`compl`	complement, bitwise NOT, unary
`|=`	`or_eq`	bitwise OR, to-left assignment
`&=`	`and_eq`	bitwise AND, with assignment
`^=`	`xor_eq`	bitwise XOR, with assignment
`<<=`		left-shift, n-times with assignment
`>>=`		right-shift, n-times with assignment

Table 1.3: Bit-manipulation-operators

The logical operators are `&&` (or `and`) and `||` (`or`) and the negation `!`. The operators `&&` and `||` have the special property that the evaluation of this expression is terminated

already if the result is clear (so-called sequencing). This property is also shared by the comma-Operator `,` – that sometimes is used to construct complicated `for`-loops. Its value is the same as the value of the right expression.

```
double    a[20];
unsigned ind[5];

// safe, even if some ind[i] >= 20, since the last expression
//  will anyway not be evaluated in that case
for (int i=0; i < 5 && ind[i] < 20 && a[ind[i]] >= 0; i++ )
   sqrt(a[ind[5]]);

// sequence operator used to combine two expressions
int i, j;
for ( i=0, j=2; i < 18; i++, j++ )
   a[i] = a[j];

 // bizar but legal use of ,
 i = 5*i, 3;    // i is set to 3, 5*i is computed, but discarded
```

The (arithmetic) comparison-operators are `==`, `!=`, `<`, `<=`, `>` and `>=`. The value of a logical operation and the result of a comparison in C++ are of type `bool`, i.e. either `true` or `false`.

Further operators

The only ternary operator, `?:`, which is used to select alternative expressions of the same type, was already discussed (see sec. 1.6.2).

```
int i;
cout << (i > 0) ? i : 0;     // never prints a negative number
```

The operator `()` leads to a function-call, `[]` is the index-operator for fields and pointers, a point `'.'` allows to select an element from a structure and `->` is equivalent when a pointer to a structure is given.

```
struct Person {
  string name;
  int age;
};

void f(){
```

```
  Person    p;
  Person *p_p = &p;
  p.age    = 25;
  p_p->age = 25;  // equivalent
}
```

The unary operators * and & are used to find the value of the memory location where a pointeri points at and to find an address of an arbitrary variable (see sec. 1.8).

Operator-expressions are (like in C) treated according to their priority and associativity, see table 1.4.

operator	evaluation from
::	left to right
() [] -> . X++ X-- typeid XXX_cast<TYPE>()	left to right
! ~ ++X --X + - * & (TYPE) sizeof new delete	right to left
.* ->*	left to right
* / %	left to right
+ -	left to right
>> <<	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
= *= /= %= += -= <<= >>= &= |= ^=	right to left
?:	right to left
throw	–
,	left to right

Table 1.4: Priority and associativity of operators

1.7.4 inline functions

In general, several things happen when a function is either called or closed. Some of this can be avoided or optmized by use of so-called `inline`-functions. However, we will not go into detail here.

1.7.5 Overloading of functions

In contrast to C and some other languages, C++ not only the name of the function, but also the types of the arguments are relevant. Therefore it is possible to define a function for each type `int`, `double` or `struct Date {...}`, all with the same name. The name does not have to contain type information thus. In C++, the compiler selects the right function with the appropriate type by itself. One can think of an internal function-name used by the compiler that also contains the argument-types.

```
void swap(int &a, int &b)   // internal name e.g. swap_int_int
{
  int tmp = a;
  a = b; b = tmp;
}

void swap(double &a, double &b)    // swap_double_double
{
  double tmp = a;
  a = b; b = tmp;
}

void f()
{
  double a, b;
  int    i, j;
  // ...
  swap(a, b);  // double version
  swap(i, j);  // int    version
}
```

1.8 Pointers, pointer-field duality, and references

1.8.1 Pointers

From the viewpoint to the computer, every memory-cell in the (virtual) memory is uniquely marked by a certain number. C/C++ allows to use these **adresses** in memory-space for programming purposes. Since it is allowed to self-define names for these address-pointers, the program can also become much more clear and readable. The declarations:

```
int    i=5;
double a=12.34;
char   *c1="hello";   // declaration of an array of char
```

```
int    j=0;
```

create the memory-entries in Fig. 1.1 (left column). Only the memory-entries (the values of the variables) occupy space in the computer memory. The names of variables do not occupy space even of one would use instead of `i` the more meaningful name `counter`.

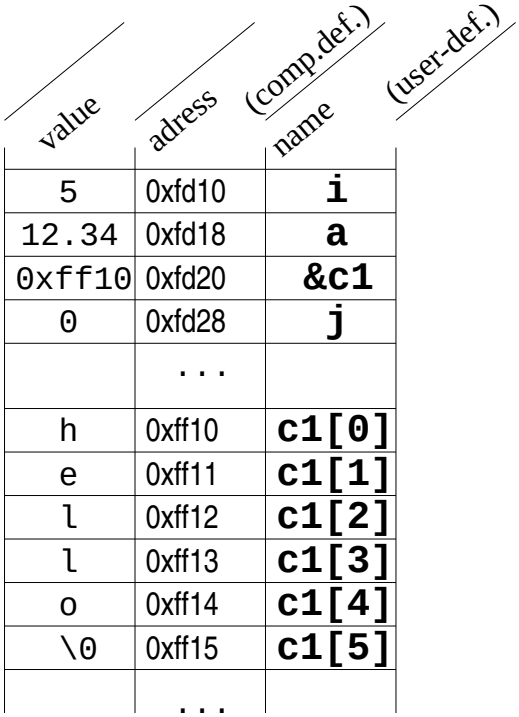


Figure 1.1: (Shematic) memory occupations (left), computer-internal memory adress (mid), and (user-defined) name of variables (right).

In general, there is no guarantee that sequentially defined varibales also occupy memory sequentially (except for fields). The variable `c1` of type `char` contains a pointer to another range of memory where the values of the field are stored. The definition of `*c1` reserves some range in memory that is large enough to store the word `"hello"`. The elements `'h'`, `'e'`, `'l'`, `'l'`, and `'o'` can also be accessed using the form `c[0]`, `c[1]`, `c[2]`, `c[3]`, and `c[4]`. Note again that numbering in C/C++ always starts with `0`.

If one has to insert or delete a range of a sorted list (quickly and efficiently), then this is not possible using a linear field like the above `*c1`. A *linear field* is special in so far that all data are stored in subsequent positions in memory. As well inserting as removing entries from this fiel requires copying the full range "above" the modified element. (In average that is half the field-length to be moved). This is acceptable for short fields but not for long ones. A possible solution to this is a *linked list* which contains not only the values of the values of the elements but also the information where the next element can be found. The actual location does not matter, i.e., only the name of the memory-cells have to be modified. For example, in order to remove an element x from a linked list, only

the name entry of the previous element has to be changed such that it does not point to x anymore, but to the following element. This information is available in the x-element and thus just has to be copied. Deleting several subsequent entries requires the same effort and entering a (part of a) linked list from somewhere else is also rather efficient. However, due to this additional options, a linked list also usually requires more memory that a normal field.

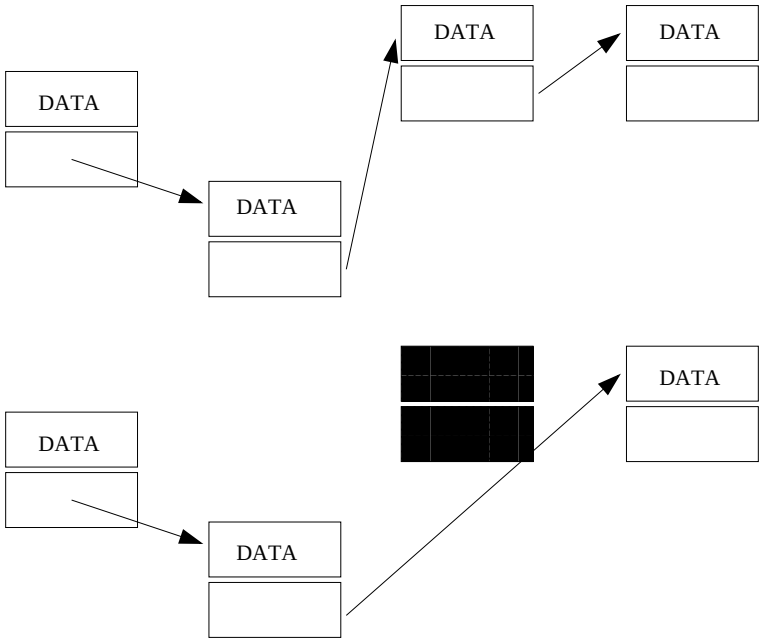


Figure 1.2: Use of pointers in a linked list. (Top) the list before an element is removed, (bottom) the list after, where the black element denotes the removed one that is not part of the list anymore..

In order to allow a computer independent dealing with such objects like linked lists, C/C++ provides also the type pointer. A pointer can point to an arbitrary data-type – which actually couls be a pointer itseld. In order to declare a pointer, a (star) `*` is used. In order to get the adress of a variable, the `&`-operator is used. The declarations in the program-fragment:

```
// Declaration part
  int     i = 5;        // initialisation
  int  *p_i = &i;       // p_i is a pointer to an int,
                        //   here the variable i
  int **pp_i = (&p_i);  // pointer to pointer to int

// How to use * and & legally in the program
       i = 0;            // set i to 0
    *p_i = 1;            // set i to 1, p_i remains unchanged
  **pp_i = 2;            // set i to 2, p_i remains unchanged
   int j = 7;            // declare another integer j
```

```
*pp_i = &j;          // change p_i to point on j, *p_i != i
  p_i = &i;          // let p_i point to i again
```

lead to memory occupation as indicated in Fig. 1.3.

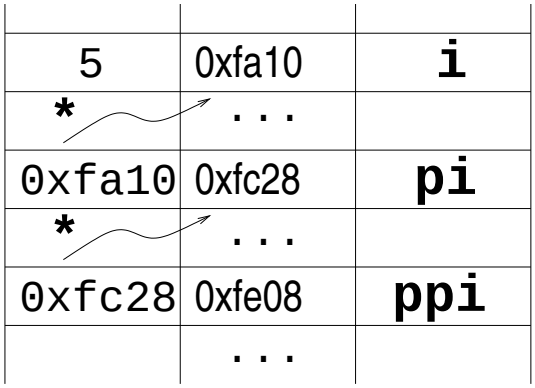


Figure 1.3: Memory occupation as in Fig. 1.1.

In order to change the value of a variable a pointer points to, or in order to use this value in an operation, also the * operator is used.

```
*p_i = 27;               // i is 27 now
*p_i = 2 * (*p_i) + 5;   // equivalent to:  i = 2 * i + 5;
```

Constant pointers are declared by using a `const` right of the pointersymbol *. They must be initialized already at declaration, since they are fixed thereafter and cannot be changed anymore.

```
const int *cp_i = &i;                  // ok, i cannot be changed
                                       //     through use of cp_i
                                       //     cp_i can be changed
const char newline = '\n';
const char * const p_nl = &newline;    // const pointer to const char
                                       // note: p_nl = '0'; -> error!
```

The * operator has thus two meanings: In a variable declaration, it marks a pointer, whereas in front of a variable (of type pointer) it leads to access on the memory (value) of the corresping memory space.

Analogously, also the & operator has two meanings: During variable declaration, it denotes a reference, whereas in front of a variable, it is the so-called `adress of` operator, i.e. it provides the adress of the variable.

Note that it is possible to do arithmetics/mathematics with pointers. Above, the line `p_i = p_i + 5` would let the pointer point to 5 `int` memory entries further. For this,

the compiler needs to know how big the corresponding data-type is. Also the difference of pointers can be formed but that makes only sense inside a field.

Navigation with in a field can be done in the following way:

```
int p[30];           // declare a field of int
*(p_i + 25)  = 15;   // the value of the 25th entry (past the one
                     // pointed to by p_i) is set to 15
p_i[25]      = 15;   // equivalent ...
```

The language guarantees that 0 is never the adress of a real data-element. A pointer with value 0 can thus be used to indicate an error or, e.g., the end of a list.

Pointer to void

The type pointer to void is generic. It is guaranteed that any pointer can be transferred into a pointer to `void`, without loss of information. In C the `void *` was often used to transfer information to functions/routines that can deal with arbitrary pointers. (e.g. sort-function like `quicksort`, or the return value of `malloc`). In C++ it is recommended to use `template`-functions that guarantee type-consistency and type-safety.

1.8.2 pointer-field duality

Between pointers and fields exists a very close relation, see above the use of `[]` for the "indirect" de-referencgin of pointers. A field `a[..]` is internally transformed to a pointer on the first element of the field. Access to the field is then dealt with via the pointer-arithmetics mentioned above. This is also the reasone why C/C++-Felder always start with 0 (i.e. the localisation of field entries can be done with less operations).

```
int a[20];
int *p = a;          // ok, a is name of the field and can
                     //     be used as address to the field
int *p = &(a[0]);    // ... equivalent

p[5] = 4;            // access to a through pointer arithmetic,
                     //     a[5] is set to the value of 4
a[5] = 4;            // ... equivalent

int b[20][30];       // b is now an array of 20 arrays of
                     //     30 elements of type int
                     //     int *p = b;  error! wrong type, since
                     //     b is of type pointer to 30 int here
```

```
int *r = b[0];      // ok, r points to the first int
                    //     in the first (of 20) array(s) of 30 ints
int (*q)[30] = b;   // ok, q points to the first set of 30 int's,
                    //     q and b can now be used synonymous
q[12][15]  = 26;    // ok, set element (13,16) to the value 26
(*(q+12))[15] = 26; // ... equivalent
*(*(q+12)+15) = 26; // ... equivalent
b[12][15]  = 26;    // ... equivalent
```

1.8.3 Transfer of field-variables to functions and dynamic memory management

In order to transfer a field to a function, it is sufficient to just transfer the pointer, see sec. 1.7. For a one-dimensional field this is just the pointer to the first element, while for two-d fields it is a pointer to a field with a compatible, fixed number of elements. Only the first dimension of the field can be left out in this case, all others have to be specified at compile-time. This makes the C-type fields rather in-practical to be used when dynamic memory management is desired.

```
void f( int b[][30] )
void g( int (*q)[30] )

int  b[20][30];
int  c[10][30];
int  d[15][31];

f(b); g(b); f(c); g(c); // ok
f(d); g(d);             // errors, since second dimension not ==30
```

1.9 I/O (Input and Output)

The I/O in C++ is completely renewed as compard to C. This was mostly necessary because of the complex format in C with non-constant number of variables.

The two most important objects are I/O-channels like `istream` and `ostream`. Both are declared in the header `iostream`. In the previous chapters, the standard I/O channels '`cin`' and '`cout`' with the operators '`<<`' oder '`>>`' were already used frequently. In the following alternatives and extensions are explained.

In which sequence is I/O processed?

```
#include <iostream>
//   The programmer types this
     cout << "Salary: "   << Person.mSalary;

//   ... but the compiler reads this:
   ( cout << "Salary: " ) << Person.mSalary;
```

In order to write the string `Salary:` to the screen, the operator `<<` is called from the `ostream cout` and returns another `ostream` with which the next `<<` is called. This time, `Person.mSalary` is the second argument. Analogously after an input with `cin`, another `istream` is returned so that one can input again.

1.9.1 Elementfunctions of iostreams

Instead of the form as used up to now, I/O can also be performed in the form of functions – for compatibility. However, the function-form has much wider range of applications, see Tab. 1.5 gezeigt sind.

```
char c1='x';
cout << c1 << '\n';    // Syntax used up to now
cout.put( c1 );        // Equivalent form using functions
cout.put( '\n' );
```

`ostream::put (char c);` Example: `cout.put( c1 );`	writes character c to ostream
`char c='x';` `cout << c;`	For comparison ... also writes the char 'c'
`istream::get (char c);` Example: `cin.get( c1 );`	reads character c from istream
`char c;` `cin >> c;`	For comparison ... Reads character c, ignores SPACE, TAB, EOL

Table 1.5: Prototypes of some element-functions for I/O alternative to the use of `cout` und `cin`.

There are many related element-functions like this that allow a much more controlled and well-behaved input and output of data.

1.9.2 Formatting

Typically it is necessary to change to format of the output, e.g., in order to obtain a better readable table, or in order to prepare the output for the transfer to another software, or to write the full precision on the screen, or ... see Tab. 1.6 for a summary.

Manipulator	Description
`int width=12;` `cout << setw(width);`	sets the minimal number of digits for the following output only.
`int prec=5;` `cout << setprecision(prec);`	Changes the number of digits behind the comma, or the max. number of digits
`char c='*';` `cout << setfill(c);`	Defines a filling-symbol as, e.g., `*`.

Table 1.6: Examples of formatting functions for the output

Dabei ist zu beachten, daß die "Anderung, die durch solche Funktionen herbeigef"uhrt wird, evtl. nur f"ur die n"achste Ausgabe g"ultig ist. Um diese Funktionen verwenden zu k"onnen, muß man zuerst `#include<iomanip>` aufrufen.

```
#include<iomanip>
   double x=1234.5;
   cout << x << '\n';   // leads to: 1234.5
   cout << setfill ('*') << setw (10) << setprecision (5);
   cout << x << '\n';   // leads to: ****1234.5
   cout << x << '\n';   // back to previous setting: 1234.5
```

Furthermore there exist switches (flags) for the formatted output that need no parameters, as summarized in Tab.1.7:

Flag	Group	function
left	adjustfield	left-adjust
right	adjustfield	right-adjust
internal	adjustfield	+/- left
dec	basefield	decimal numbers
hex	basefield	hexadecimal numbers
oct	basefield	octal numbers
showbase		shows the dec-basis of hex- and oct-numbers
showpos		prints '+' before number if positive
uppercase		E, X, A-F instead of: e, x, a-f
fixed	floatfield	Fixed-point notation
scientific	floatfield	scientific notation

Table 1.7: Flags for the formatted output of data

In order to activate flags one can use alternatively

```
   cout.setf (ios:: 'flag', ios:: 'group')
```

```
cout << setiosflags (ios:: 'flag', ios:: 'group')
```

where the possibilities for `'flag'` und `'group'` are summarized above in Tab. 1.7, however, the second parameter is not always needed. In order to change the setting, or set it to default, for a single flag or for groups of flags, one can use:

```
cout.unsetf (ios:: 'flag')
cout << resetiosflags (ios:: 'flag')

cout.unsetf (ios:: 'group')
cout << resetiosflags (ios:: 'group')
```

1.9.3 Files

For reading and writing on files, one can use the channels `fstream` and `ofstream`. These are derived from the std-streams `istream` and `ostream`.

```
# include <fstream>
...
   double  z=1.234;
   int     m=5;
   ofstream outs ( "filename", ios::out );
                        // open the file with name
                        // 'filename' for writing
                        // ios::out can be skipped
   outs << x << '\n';   // the use of the self-defined output-stream
   outs << j << '\n';   // 'outs' is analogous to the use of 'cout'
   outs.close();        // close the out-fstream 'outs'

   ifstream ins  ( "filename", ios::in );
                        // open the file with name
                        // 'filename' to read from
                        // ios::in can be skipped
   ins >> z;            // the use of the self-defined input-stream
   ins >> m;            // 'ins' is analogous to the use of 'cin'
   ins.close();         // close the in-fstream 'ins'
```

At least now, when working with files, we need information whether we have reached the end of the file (EOF) or if the opening of the file was successful. For this, the so-called status-informations can be used, which return – dependent on the status of a file – either `true` oder `false` zur"uckgeben:

```
ins.good()   // true, if no error occured in stream 'ins'
```

```
ins.eof()    // true, if the end of file/stream is reached
ins.bad()    // true, if an hardware error occured
ins.fail()   // true, if a logical error or a
             //          hardware error occured

ins.clear()  // set ins.good() to true
ins.clear( ios::failbit | ins.rdstate() )
             // set fail() manually to true
```

How to read from a file to its end?

The following examples show which problems can occur doing this.

```
int n;

while (!cin.eof()){
   cin >> n;
   // .... this reads too far
}

while((cin >> n).good){
   // ... this reads not far enough
}

while(cin >> n){
  // ... this is the way it works
}
```

1.10 Dynamical Memory Management

In order to request memory, C++ provides the operators `new` and `new []` for single variables or for fields, respectively. In order to free the memory, `delete` and `delete []` have to be used.

```
int    *p_i  = new int;
int        n = 40;
int    *pa_i = new int[n];
// ...
delete   p_i;
delete[] pa_i;
```

The request for memory requires the type after `new`, and a pointer to this new element is returned. For fields, `new[]` is used and the number of memory-elements within the

brackets is requested. The request returns a pointer to the first element in the field.

```
int    *pa_i = new int[n];

pa_i[0]=5;
pa_i[1]=4;
```

The operator `new` returns an "exception" `bad_alloc`, it the request for memory does not succeed and then writes "core" and terminates..

1.11 Organisation implementation and header-files

Big programs and program-packages should not be kept in a single file since this can lead to long compilation times and non-clear dealing with and navigation in the file. Furthermore it causes big trouble if more persons work on (a single file) at the same time.

Therefore, one typically splits programs into several files (projects), that can be compiled independently from each other. There are compilable (Implementation-) C++-files with the extension `.cc` or `.cpp`. Files that only contain declarations are called header-files and are used to make functions and data from one file known to another. These have the extension `.h`. In general (except for `main.cc`) an implementation-file comes together with a header-file.

header-files should contain:

1. so-called include-guards, that hinder the multiple inclusion of header-files (required by the standard)

   ```
   #ifndef HEADER_H
   #define HEADER_H
     // declarations
   #endif
   ```

2. Declarations of functions that are used amongst several modules. For declarations of local functions, another header-file can be used.

3. Definition of `inline` functions

4. Declaration of classes and their elementfunctionen

5. `extern` declarations of global variables, and declaration of `static` varbiables

6. Declaration and definition of `template`-functions and -classes

implementation-files should contain:

1. Implementation of elementfunctions of not-template-classes
2. Implementation of regular functions
3. Declaration von file-scope, class-scope and global static variables

1.12 Literature

The many new terms used in the last pages indicates that C++ has much more possibilities, rules and options to be explored. Classes, templates, over-loading, scopes, etc. can not be discussed in this short course. However, here is a list of further literature:

1. Brian W. Kernighan and Dennis M. Ritchie, *The C-Programming Language*, 2nd edition, Prentice Hall, 1988.
2. B. Stroustrup, *The C++ Programming Language*, 3rd edition, Addison Wesley, 1997.
3. Steve Ouallinie, *Practical C++ Programming*, O'Reilly, 1995.
4. C. S. Horstmann, *Mastering C++*, John Wiley & Sons, New York, 1996.
5. S. B. Lippman, *C++ Einführung und Leitfaden*, 3. Auflage, Addison-Wesley, Bonn, 1998.

1.13 Exercises C/C++

Exercise 1

Describe with a few words what the following program is doing. Do not copy this file into your computer - solve this by reading it and write a few sentences (in 15 minutes).

```
#include<iostream>
#include<fstream>
#include<cmath>
using namespace std;

int main(int argc, char *argv[])
{
// Define field x(t) with length 1000
   double x[1000], t;  // output variables
// initial conditions
   double A, delta;     // A=amplitude, delta=phase-angle
   double mass, ksprng; // mass=mass, ksprng=spring-constant
   double t_max, dt;    // t_max=max-time, dt=time-intervall
// Request input of the parameters
   cout << "amplitude     ";  cin >> A;
   cout << "phase-angle   ";  cin >> delta;
   cout << "mass          ";  cin >> mass;
   cout << "spring-const. ";  cin >> ksprng;
   cout << "max-time      ";  cin >> t_max;
   cout << "time-interval ";  cin >> dt;
   double omega=sqrt(ksprng/mass);
// Loop from t=0 to t=t_max
   t=0.0;
   for( int i=0; i<=t_max/dt; i++ )
   {
       x[i]=A*sin(omega*t+delta);  // Compute function
       t=t+dt;                     // Step to next time
   }

   ofstream outfile("plot.data");  // Output to file
   t=0.0;
   for( int i=0; i<=t_max/dt; i++ )
   {
      outfile << t << " " << x[i] << "\n";
      t=t+dt;
   }
}
```

Exercise 2

Extend the basic C++ program by the command lines `int inum=0; cin >> inum;` where the second requests input of an integer number from the keyboard.

Write a C++ program that prints the binary representation of the integer number `inum` to the screen. You can check if the program works by, e.g., using the number 496 from the lecture.

Exercise 3

What is the greatest value of n that can be used in the sum

$$1^2 + 2^2 + 3^2 + \ldots + n^2$$

and gives a total value of the sum less than 100?
(a) Write a short C++ program that answers this question – like in the MATLAB exercise 1, cf. section 2.7.
Then answer the question also for an exponentn of 10 and a total value of the sum of (b) less than 10^6 and (c) less than 10^{12} (*Advanced*).

Exercise 3

Declare an array with 10 001 entries. Compute the solution vector $f(x_i)$, with $x_i = 0, \ldots, 10\,000$ (as fast as possible!)

$$f(x_i) = \frac{x_i}{\sin(x_i) + 2}$$

In the same program, form the alternating sum

$$S_n = f(x_0) + f(x_1) - f(x_2) + f(x_3) - \ldots$$

up to the last term and print the result to the screen. Finally, compare the result with the result obtained in the MATLAB exercise 3, section 2.7.

Exercise 4

Write a root-finding function in C/C++ and apply it to the same problem as in the MATLAB exercise in section 2.7. (*Advanced*)

Exercise 5

In the following program (part of which we have discussed today) there are hidden 10 syntax and typo-errors. See how many you can find in 15 minutes.

```
#include <iosteam>
#include <cmath>
#include <cstdlib>
using namespce std;

int main(int argc, char *argv[])
{
   int i;
   int n=5;
   int *p;
   int a[10];

   for( i=0; i<=10; i++ }
      a[i]=10;
      p[i]=10;

   cout << a << " \n";
   cout << *a << " \n";
   cout << *(a) << " \n";
   cout << a[1] << " \n"
   cout << *(a+2) << " \n";
   cout << size of( int ) << " " << sizeoff( int* ) << "\n";
   cout << sizeof( double ) << " " << sizeoff( double* ) << "\n";
   cout << " n: " << n << " " << &n << "\n";
   i=n+1;
   cout << " i: " << i << " " << &i << "\n";
   p=&n;
   cout << " p: " << p << " " << *p << "\n";
   *p+=2
   cout << " p: " << p << " " << *p << "\n";
   p++;
   cout << " p: " << p << " " << &p << "\n";

   return 0;
}
```

Exercise 6

This is an example of pointers:

(a) Use the following program segment and play around using the pointers.

```
#include <cstdlib>
#include <iostream>
#include <cmath>
using namespace std;

int main(int argc, char *argv[])
{
   char c='x';
   int a[20];
   int *p = a;          // ok, a is name of the field and can
                        //     be used as address to the field
// int *p = &(a[0]);    // ... equivalent (error->redeclaration)

   p[5] = 4;            // access to a through pointer arithmetic,
                        // a[5] is set to the value of 4
   a[5] = 4;            // ... equivalent
                        //
   int b[20][30];       // b is now an array of 20 arrays of
                        //     30 elements of type int
                        //     int *p = b;  error! wrong type, since
                        //     b is of type pointer to 30 int here
   int *r = b[0];       // ok, r points to the first int
                        //     in the first (of 20) array(s) of 30 ints
   int (*q)[30] = b;    // ok, q points to the first set of 30 int's,
                        //     q and b can now be used synonymous
   q[12][15]  = 26;     // ok, set element (13,16) to the value 26
   (*(q+12))[15] = 26; // ... equivalent
   *(*(q+12)+15) = 26; // ... equivalent
   b[12][15]  = 26;     // ... equivalent

   cout.put(c);
   int    *p_i  = new int;
//   Person *p_p  = new Person("B.Stroustrup",45);
   int        n = 40;
   int    *pa_i = new int[n];
// ...
   delete   p_i;
//   delete   p_p;
   delete[] pa_i;
```

```
//
//
   pa_i[0]=5;
   pa_i[1]=4;

   for(int i=0; i<10; i++) cout << pa_i[i] << ' ';

   return 0;
}
```

More advanced exercise:

Define an `int` field that contains the numbers:
`{ 1, 0, -10, 12, -2, 5, 5, 1, -4, -2, -4, -8, 3, 25, 51, -4, -5 }`

Then sort the entries of this linear field.

(b) Read the field from a file.
(c) Use a linked-list to store the field.
(d) Use this linked list (with pointers) to sort the field.
(e) Write the sorted list to an output file.

1.14 Installation of a C++ compiler/GUI

There are many commercial C++ compilers for various operating systems available. While many Unix-type operating systems like modern Linux distributions have C++ compilers and debuggers embedded (g++ and gcc), this is not the case for a MS-Windows systems like Microsoft Windows XP or Microsoft Vista. Nevertheless, you can use the following open source GUI-based C++ compiler (GNU General Public License (GPL)) on a MS-Windows system.

- **Download.**
  Go to: `http://sourceforge.net/projects/dev-cpp/` and click on:
  `Download Dev-C++`. Download the Binaries package `devcpp-4.9.9.2_setup.exe`.

- **Installation.**
  Double click on the setup file. On the `Choose components` window,
  select the `Full` type of installation.

- **First time configuration.**
  Accept all the default configurations.

  A very usefull tool is to have line numbers displayed.
  Tools → Editor options → Display → Line numbers

  In case the linker does not find some functions, you have to set the following command line options.
  Tools → Compiler options
  and insert `-lm -l .` into both fields and activate the input.

- **Ready to start.**
  First create a new C++ source file and save it under the name e.g. `test.cpp`.

```
1 #include <cstdlib>
2 #include <iostream>
3 using namespace std;
4 int main(int argc, char *argv[])
5 {
6     system("PAUSE");
7     return EXIT_SUCCESS;
8 }
```

  Alternatively, you can create a new project:
  File → New → Project → Console Application.
  At this point the project is created with a main file in it. You need to save the main file under the name e.g. `test.cpp`. Then you have the leave the project and open the source file (projects are much too complicated/overshooting for our purposes - we recommend to avoid them). Then you can compile it and run it!
  Furthermore, you can try to modify the file and compile and run it.

```
1     cout <<"hello!"<< endl;
2     system("PAUSE");
3     return EXIT_SUCCESS;
```

2

MATLAB introduction

MATLAB ("**MAT**rix **LAB**oratory") is a commercial program for interactive numerical analysis. You can use it in research and teaching actions, and for many day-to-day tasks, including data reduction, curve fitting, and plotting of technical data.

Before starting with the programming course and the introduction to MATLAB , we would like to rise the question: *Why use MATLAB for numerical analysis*? Note that we do not discuss here, why MATLAB is *better* than any other programming language (like e. g. C or C++ discussed in the first chapter of the course). Some answers to the latter question will be given in the next chapter.

MATLAB is a usefull for numerical analysis for the following reasons:

- It integrates plotting and analysis.
- It dynamically manages memory for matrices and vectors.
- It transparently performs operations on complex and real valued numerical quantities.
- It incorporates very high quality software libraries developed by numerical analysts and software developers throughout the world.
- The source code for nearly all of the high level algorithms used by MATLAB is available.
- It is interactive. Substantial analysis can be performed by entering one command at a time and thereby obtaining results.

- It offers high level abstractions for linear algebra operations. Manipulation of vectors and matrices, including solving linear systems, is supported by intuitive extension of the basic addition, subtraction, multiplication, and division operators.

- It is widely used as a teaching and research tool.

MATLAB References

There are several ways for getting help with MATLAB. A good way to start is to type:

```
1 >> helpdesk
```

The helpdesk command opens an HTML document, which contains the MATLAB reference book. This is quite a large document! Useful sections for beginners include Getting Started and MATLAB Functions. You might want to browse them before beginning your first session. Its helpful to keep this browser window open while you are working with MATLAB , so that you can refer to it easily.
There are other ways of getting help with MATLAB that do not make you search through the whole reference book. If you already know the name of a MATLAB function, for example, `quit`, and you want to learn about its use, enter:

```
1 >> help quit
```

You will see a description of the command `quit`. During your first experiences with MATLAB you should from time to time take a:

```
1 >> demo
```

session which will show you various features of MATLAB . Another very useful feature is the lookfor command. It looks for all the commands related to a given topic. Try:

```
1 >> lookfor 'help'
```

It will list all of the commands we just discussed.
Besides the inherent help of MATLAB , you will find various online courses and references in the internet and, last but not least, textbooks. Here, we just mention a view:
Online sources

- MATLAB home page (from manufacturers)
  `http://www.mathworks.com`

- MATLAB online course (TU Eindhoven)
  `http://www.imc.tue.nl`
- MATLAB online Reference Documentation
  `http://www.math.ufl.edu/help/matlab/ReferenceTOC.html`

Books and others references

- MATLAB An Introduction with Applications, Amos Gilat, 2008.
- MATLAB Programming for Engineers, Stephen Chapman, 2007.
- Essential MATLAB for Engineers and Scientists, Brian Hahn and Dan Valentine, 2007.

Non-commercial alternatives to MATLAB

The following list (some) alternatives to MATLAB , including freeware clones and similar commercial packages.

- SCILAB
  Scilab is a scientific software package for numerical computations providing a powerful open computing environment for engineering and scientific applications. Open source software, developed by INRIA. Available for Linux and Unix (HP-UX), Wintel PCs (2000, XP, Vista) and Mac OS X.
  `http://www.scilab.org`
- OCTAVE
  GNU Octave is a high-level language, primarily intended for numerical computations. It provides a convenient command line interface for solving linear and nonlinear problems numerically, and for performing other numerical experiments using a language that is mostly compatible with Matlab. It may also be used as a batch-oriented language. GNU Octave is also freely redistributable software. You may redistribute it and/or modify it under the terms of the GNU General Public License (GPL). Available for Linux and Unix (Sun Solaris), Wintel PCs and Mac OS X.
  `http://www.octave.org`

2.1 MATLAB introduction

The MATLAB *command window* is the main window where you type commands directly to the MATLAB interpreter. The MATLAB *editor window* is a simple text editor where you can load, edit and save complete MATLAB programs. The *editor window* also has

a menu command (debug/run) which allows you to submit the program to the *command window*. The MATLAB *help window* gives you access to a great deal of useful information about the MATLAB language and MATLAB computing environment. It also has a number of example programs and tutorials.

2.1.1 A first program

This program demonstrates the text output function displaying the message `Hello world!`.

```
% Do it the good old fashioned way via the command window
disp('Hello World!');
% Do it the new GUI way with a message box
msgbox('Hello World!','Hello World!');
```

2.1.2 Variables and Assignment

This section describes the fundamental operations involved in the creation and use of MATLAB variables. Detailed discussions of scalars, vectors, matrices, and strings are provided in separate sections.

MATLAB variables are created when they appear on the left of an equal sign. The generic statement

```
    >> variable = expression
```

creates the "variable" and assigns to it the value of the expression on the right hand side. You do not need to define or declare a variable before it is used. Matrices do not need to explicitly dimensioned, and MATLAB allows you to increase the size of a matrix as you work.

Advanced MATLAB users will point out that the speed of MATLAB functions can be increased by pre-allocating memory for matrices and vectors. This and other performance considerations are discussed elsewhere.

Variable names must begin with an alphanumeric letter. Following that, any number of letters, digits and underscores can be added, but only the first 19 characters are retained. MATLAB variable names are case sensitive so x and X are different variables.

All numerical variables in MATLAB are matrices, a mathematical data type corresponding to a two-dimensional array of numbers. Before performing any calculations with a numeric variable, MATLAB prods and pokes into its contents. MATLAB is very careful to follow

the rigorous rules of linear algebra, and it only allows legal operations between matrices, vectors and scalars.

MATLAB keeps track of the dimension of each variable. Thus, the statement

```
>> x = 2
```

creates a scalar variable x. In MATLAB , a scalar is a variable with one row and one column.

A vector is a matrix with only one row or only one column. The distinction between row and column vectors is crucial. When working with MATLAB you will need to understand how to properly perform linear algebra using scalars, vectors and matrices. MATLAB enforces rules on the use of each of these variables

MATLAB variables may also be strings, which are matrices of individual characters. There is no typographical difference in appearance between numerical variables and string variables. For example, the following statements assign a numerical value to x and a string value to y.

```
>> x = 5.2
>> y = 'Bob'
```

The type of variable (numerical or string) is determined when the variable is created.

Numerical matrix elements may be either real

```
>> x = 6
```

or complex

```
>> x= 6 + i * 3
```

The type of element is initially determined at the time the variable is created. However, matrix elements may change from real to complex if, during an assignment, the right hand side expression evaluates to a complex number. In this sense, matrix elements with real values are simply complex numbers with zero imaginary parts.

Example 1 Multiply two complex numbers $a = 5 + 4\,i$ and $b = 3 - 7\,i$ and separate the result into the real and the imaginary part.

Example 2 How to print a line of strings looking like that:

```
radius = 1 area = 3.1416
```

Scalars

In MATLAB a scalar is a variable with one row and one column. Scalars are the simple variables that we use and manipulate in simple algebraic equations.

Creating scalars. To create a scalar you simply introduce it on the left hand side of an equal sign.

```
1    >> x = 1;
2    >> y = 2;
3    >> z = x + y;
```

Scalar operations. MATLAB supports the standard scalar operations using an obvious notation. The following statements demonstrate scalar addition, subtraction, multiplication and division.

```
1    >> u = 5;
2    >> v = 3;
3    >> w = u + v
4    >> x = u - v
5    >> y = u * v
6    >> z = u/v
```

Vectors

In MATLAB a vector is a matrix with either one row or one column. The distinction between row vectors and column vectors is essential. Many programming errors are caused by using a row vector where a column vector is required, and vice versa.

MATLAB vectors are used in many situations, e.g., creating x-y plots. In these contexts a vector is just a convenient data structure. MATLAB still enforces the rules of linear algebra so paying attention to the details of vector creation and manipulation is always important.

Creating vectors. There are numerous ways to actually create a vector, each one having advantages in particular situations.

- **placing a sequence of numbers within square braces**

```
>> v = [3 1]

v =

     3     1
```

- **using the the built-in functions ones, zeros, linspace, and logspace:** The ones, zeros linspace, and logspace functions allow for explicit creations of vectors of a specific size and with a prescribed spacing between the elements. These functions will be demonstrated by example without providing an exhaustive reference. Refer to the MATLAB manual (or help pages) for details.

  To create a vector with one of these functions you must decide how long do you want the vector to be. You must also decide whether the vector is a row or column vector.

  The ones and zeros functions have two arguments. The first is the number of rows in the matrix you wish to create. The second is the number of columns. To create a row or a column vector set the appropriate argument of ones and zeros to one.

  To create a row vector of length 5, filled with ones use

```
    >>  x = ones(1,5)

x =

     1     1     1     1     1
```

  To create a column vector of length 5, filled with zeros use

```
   >> y = zeros(5,1)

y =

     0
     0
     0
     0
     0
```

  The linspace and logspace functions create vectors with linearly spaced or logarithmically spaced elements, respectively. Here are examples including the MATLAB output.

```
    >> x = linspace(1,5,5)

    x =

         1     2     3     4     5

    >> y = logspace(1,4,4)

```

```
y =

            10         100        1000       10000
```

The third argument of both linspace and logspace is optional. The third argument is the number of elements to use between the the range specified with the first and second arguments.

- **assigning a mathematical expressions involving vectors:** Once a vector has been created, it may be assigned to another vector. If the vector on the left of the equal sign does not exist it is created to fit the expression on the right hand side of the equal sign.

```
>> x = zeros(1,5);
>> y = x;
```

- **appending elements to a scalar:** MATLAB allocates memory for all variables on the fly. This allows you to increase the size of a vector simply by assigning a value to an element that has not been previously used.

```
>> x = linspace(21,25,5)

x =

    21    22    23    24    25

>> x(7) = -9

x =

    21    22    23    24    25     0    -9
```

This augmentation should be avoided for vectors involved in computations where speed is critical. Refer to Section 2.3.2 for a discussion of the performance issues.

- **using colon notation:**

  Colon notation can be used to create a vector as follows

```
>> x = xbegin:dx:xend

or

>> x2 = xbegin:xend
```

where xbegin and xend are the range of values covered by elements of the x vector, and dx is the (optional) increment. If dx is omitted a value of 1 (unit increment) is used. The numbers xbegin, dx, and xend need not be integers.

The statements above create row vectors. For example

```
>> x = 1:5

x =

1     2     3     4     5
```

To create a column vector, append the transpose operator to the end of the vector-creating expression

```
>> y = (1:5)'

y =

     1
     2
     3
     4
     5
```

Note that the `colon` expression needs to be enclosed in parentheses. Otherwise the transpose operator is applied to the value, 5, before the vector is created.

Using `colon` notation to create a vector requires you to specify the increment, whereas using the `linspace` command requires you to specify the total number of elements. The following commands show how to create the same vector with both approaches.

```
>> xbegin=1; xend=10;  nx=5;  dx=(xend–xbegin)/(nx–1);
>> x1 = linspace(xbegin,xend,nx);
>> x2 = xbegin:dx:xend;
```

The vectors are the same only if the increment dx corresponds to an integer number of elements. Prove this by repeating the preceding statements with `nx=6`;

Each method is useful and you will probably develop your own preferences. The first two methods will be described in this section. The last two methods are described in later sections.

Addressing vector elements

Individual elements of a vector can be addressed with a Fortran like subscript. For example

```
>> x = linspace(11,15,5);
>> x(2)

ans =
```

```
5
6          12
```

Automatic augmentation of vectors does not allow you to refer to elements that have not yet been allocated.

```
1      >> y = linspace(21,25,5)
2
3      y =
4
5          21    22    23    24    25
6
7      >> y(7)
8      ???  Index exceeds matrix dimensions.
```

When a colon expression appears in place of a vector (or matrix) index, the expression is a kind of implied do loop. The expression

```
1      istart:istop
```

refers to the range of numbers between istart and istop, inclusive. For example, the following statements create a row vector, x, and then copies the third through seventh elements of x into y.

```
1      >> x = linspace(31,40,10);
2      >> y = x(3:7)
3
4      y =
5
6          33    34    35    36    37
7
8      >> y(3)
9
10     ans =
11
12         35
```

The expression, y = x(3:7), copies the third through seventh elements of x into the first through fourth elements of y. If y did not already exist it is created by the assignment.

Example 3 Create a vector such that it is components are equally spaced when you take the logarithm of it.

Matrices

Matrices are the fundamental data type in MATLAB .

Creating matrices. There are numerous ways to create a matrix:

- **placing a sequence of numbers within square braces** For example the statment

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

A =
1  2
3  4
```

creates the 2 by 2 matrix A. Note that the semicolon operator ; separates the rows.

- **using the the built-in functions eye, diag, ones and zeros** The eye, ones and zeros functions allow for explicit creations of matrix of a specific size and with a prescribed spacing between the elements.

To create a 3x2 matrix, filled with ones use

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

x =

     1     1
     1     1
     1     1
```

To create a 2x3 matrix, filled with zeros use

```
>> y=zeros(2,3)

y =

     0     0     0
     0     0     0
```

To create a 2x3 matrix, filled with 1 on the diagonal and 0 elsewhere use

```
>> y=eye(2,3)

y =

     1     0     0
     0     1     0
```

- **appending vectors to a preexisting vector**

```
>> a=1:3

```

```
a =

     1     2     3

>> A=[a;2:4]

A =

     1     2     3
     2     3     4
```

Function `diag` creates a diagonal matrix with the diagonal entries stored in the vector d

```
>> d=2:2:6

d =

     2     4     6

>> D=diag(d)

D =

     2     0     0
     0     4     0
     0     0     6
```

Addressing matrix elements

Individual elements of a matrix can be addressed in a similar way as for the vectors. For example

```
 >> A=[1 3;4 5]

A =

     1     3
     4     5

>> A(2,2)

ans =

     5
```

To obtain the second row of the matrix A, you can use

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

```
2
3 ans =
4
5      4     5
```

To obtain the first column of the matrix A, you can use

```
1
2 >> A(:,1)
3
4 ans =
5
6      1
7      4
```

The `colon` operator : stands for all columns or all rows.

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

```
1 >> A(2,:)=[]
2
3 A =
4
5      1     3
```

To extract the main diagonal of the matrix D we use function diag again to obtain

```
1 >> A=[1 3;4 5]
2
3 A =
4
5      1     3
6      4     5
7
8 >> diag(A)
9
10 ans =
11
12     1
13     5
```

Sparse matrices In various numerical solution techniques (Finite Elements) you deal with sparse matrices, i.e. matrices containing a lot of zero elements. In order to optimize the memory usage, it is convinient to store only the non-zero entries of them. There are various storage algorithm around, one which is available in MATLAB is the so-called *coordinate format*, cf. SPARSKIT, Yousef Saad:
`http://www-users.cs.umn.edu/ saad/software/SPARSKIT/sparskit.html`

```
A = [1 2 0; 0 2 3;0 4 3];
S = sparse(A);
[i,j,s] = find(S);
```

Example 4 Create a 1000×1000 identity matrix and store it as efficient as possible. You can check the memory occupied by variables with the command `whos variable_name`.

2.1.3 Advanced data structures

Strings and formatted output

As we have seen, a string in MATLAB is enclosed in single forward quotes. In fact a string is really just a row vector of character codes. Because of this, strings can be concatenated using matrix concatenation.

```
>> str = 'Hello world';
>> str(1:5)
ans =
Hello
```

There are lots of string handling functions. See the help on `strfun`. Here are a few: `strcmp(string-1,string-2)` returns `1` if strings `string-1` and `string-2` are the same and `0` otherwise.

```
>> str = 'Hello world';
>> upper(str)
ans =
HELLO WORLD
```

```
>> str = 'Hello world';
>> strcmp(str,'Hello world')
ans =
1
```

```
>> str = 'Hello world';
>> findstr('world',str)
ans =
7
```

Cell arays

Collecting objects of different sizes is a common chore. For instance, suppose you want to tabu- late the Chebyshev polynomials 1, $x\,2\,x^2-1$, $4\,x^3-3\,x$ and so on. In MATLAB one expresses a polynomial as a vector (highest degree first) of its coefficients. The number of coefficients needed grows with the degree of the polynomial. Although you can put all the Chebyshev coefficients into a triangular array, this is an inconvenient complication. **Cell arrays** are used to gather dissimilar objects into one variable. They are indexed like reg- ular numeric arrays, but their elements can be absolutely anything. A cell array is created or referenced using curly braces { } rather than parentheses.

```
>> str = {'Goodbye', 'cruel', 'world' }
str =
'Goodbye' 'cruel' 'world'
>> str{2}
ans =
cruel
```

```
>> T = cell(1,9);
>> T(1:2) = { [1], [1 0] };
>> for n = 2:8, T{n+1} = [2*T{n} 0] - [0 0 T{n-1}]; end
>> T
T =
Columns 1 through 5
[1] [1x2 double] [1x3 double] [1x4 double] [1x5 double]
Columns 6 through 9
[1x6 double] [1x7 double] [1x8 double] [1x9 double]
>> T{4}
ans =
4 0 -3 0
```

Cell arrays can have any size and dimension, and their elements do not need to be of the same size or type. Cell arrays may even be nested. Because of their generality, cell arrays are mostly just containers; they do not support any sort of arithmetic.

Structures

Structures are much like cell arrays, but they are indexed by names rather than by numbers. Say you are keeping track of the grades of students in a class. You might start by creating a structure of a finite element as follows

```
>> element.type   = 'quad';
>> element.shape  = 'quadratic';
>> element.test   = 'quadratic';
>> element.number = 1;
>> element.nodes  = [1 2 3 4 5 6 7 8];
```

The name of the structure is `element`. Probably you have more elements than one in a finite element mesh.

```
>> element(2).type   = 'quad';
>> element(2).shape  = 'quadratic';
>> element(2).test   = 'quadratic';
>> element(2).number = 2;
>> element(2).nodes  = [9 10 11 12 13 14 15 16];
```

Now you have an array of structures. This array can have any size and dimension. However, all elements of the array must have the same fields. Struct arrays make it easy to extract data into cells:

```
>> [test{1:2}] = deal(element.number)
```

In fact, deal is a very handy function for convert between cells and structs. More information can be found in the online help.

Example 5 Assume we have a shop and we sell five different vegetables (carrots, salad, tomatoes, apple, banana) which have different prices (1.5, 0.75, 1.42, 2.50, 0.89). They were bought at different days (1 july 2008, 30 june 2008, 5 july 2008, 3 july 2008, 6 july 2008). The amount of vegetables in the shop are today (50, 25, 15, 120, 35). Store these data using a `structure` and calculate the total value of the vegetables.

2.1.4 Loops: `for`, `while`

For Loops. The `for` loop allows us to repeat certain commands. If you want to repeat some action in a predetermined way, you can use the `for` loop. All of the loop structures in MATLAB are started with a keyword such as "for", or "while" and they all end with the word "end".

The for loop is written around some set of statements, and you must tell MATLAB where to start and where to end.

For example, a simple loop will go around four times each time changing a loop variable, j:

```
>> for j=1:4,
    j
end


j =

     1
```

```
 9
10
11 j =
12
13      2
14
15
16 j =
17
18      3
19
20
21 j =
22
23      4
24
25 >>
```

When MATLAB reads the "for" statement it constructs a vector, [1:4], and j will take on each value within the vector in order. Once MATLAB reads the "end" statement, it will execute and repeat the loop. Each time the for statement will update the value of j and repeat the statements within the loop. In this example it will print out the value of j each time.

For another example, we define a vector and later change the entries. Here we step though and change each individual entry:

```
1  >> v = [1:3:10]
2
3  v =
4
5       1     4     7    10
6
7  >> for j=1:4,
8       v(j) = j;
9  end
10 >> v
11
12 v =
13
14      1     2     3     4
```

Note, that this is a simple example and is a nice demonstration to show you how a for loop works. However, DO NOT DO THIS IN PRACTICE!!!! MATLAB is an interpreted language and looping through a vector like this is the slowest possible way to change a vector. The notation used in the first statement is much faster than the loop.

A better example, is one in which we want to perform operations on the rows of a matrix. If you want to start at the second row of a matrix and subtract the previous row of the matrix and then repeat this operation on the following rows, a for loop can do this in short order:

```
>> A = [ [1 2 3]' [3 2 1]' [2 1 3]']

A =

     1     3     2
     2     2     1
     3     1     3

>> B = A;
>> for j=2:3,
    A(j,:) = A(j,:) - A(j-1,:)
end

A =

     1     3     2
     1    -1    -1
     3     1     3


A =

     1     3     2
     1    -1    -1
     2     2     4
```

Example 6 Ask the user interactively to input 5 real-value numbers and accumulate the sum of these 5 numbers and print the result on the screen.

While Loops The while loop is similar to the for loop in that it allows the repeated execution of MATLAB statements. Unlike the for loop, the number of times that the MATLAB statements in the body of the loop are executed can depend on variable values that are computed in the loop. The syntax of the while loop has the following form:

```
while expression
    % MATLAB command 1
    % MATLAB command 2
    % More commands to execute repeatedly until expression is not true
end
```

where expression is a logical expression that is either true or false. For example, consider the following while loop:

```
n = 1 while n < 3
    n = n+1
end
```

This code creates the following output:

```
n =

     1

n =

     2

n =

     3
```

Example 7 Ask the user interactively to input an unknown number of data values and calculate the average and the standard deviation.

2.1.5 Logical expressions

Logical expressions are used in if statements, switch case statements, and while loops to change the sequence of execution of commands in response to variable values. A logical expression is one that evaluates to either true or false. For example, $v > 0$ is a logical expression that will be true if the variable v is greater than zero and false otherwise.

Logical expression are typically formed using the following relational operators

Table 2.1: Relational Operators.

Symbol	Relation
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
~=	Not equal to

Note that == is not the same as =; they are treated very differently in m-file scripting environments. == compares two values, while = assigns a value to a variable.

Logical expressions can be created by combining simpler logical expressions using the following logical operators:

2.1.6 Condition: `if`

The if statement is one way to make the sequence of computations executed by in an m-file script depend on variable values. The if statement has several different forms. The

Table 2.2: Logical Operators .

Symbol	Relation
$\sim$	Not
&	And
\| | Or |

simplest form is

```
if expression
    % Commands to execute if expression is true
end
```

where expression is a logical expression that is either true or false. For example, the following if statement will print "v is negative" if the variable v is in fact negative:

```
if v < 0
    disp('v is negative')
end
```

A more complicated form of the if statement is

```
if expression
    % Commands to execute if expression is true
else
    % Commands to execute if expression is false
end
```

For example, the following if statement will print "v is negative" if the variable v is negative and "v is not negative" if v is not negative:

```
if v < 0
    disp('v is negative')
else
    disp('v is not negative')
end
```

The most general form of the if statement is

```
if expression1
    % Commands to execute if expression1 is true
elseif expression2
    % Commands to execute if expression2 is true
elseif expression3
```

```
    % Commands to execute if expression3 is true
    ...
else
    % Commands to execute if all expressions are false
end
```

The following if statement is an example of this most general statement:

```
if v < 0
    disp('v is negative')
elseif v > 0
    disp('v is positive')
else
    disp('v is zero')
end
```

Example 8 In each of the following questions, evaluate the given MATLAB code fragments for each of the cases indicated. Use MATLAB to check your answers.

```
    if z < 5              a. z = 1    w = ?
       w = 2*z            b. z = 9    w = ?
    elseif z < 10         c. z = 60   w = ?
       w = 9 - z          d. z = 200  w = ?
    elseif z < 100
       w = sqrt(z)
    else
       w = z
    end
```

Example 9 Evaluate the following function.

```
h(T) = T - 10          when 0 < T < 100
     = 0.45 T + 900    when T > 100
```

2.1.7 Condition: `switch`

`Switch` statements are used to perform one of several possible sets of operations, depending on the value of a single variable. They are intended to replace nested `if` statements depending on the same variable, which can become very cumbersome. The syntax is as follows:

```
switch variable
    case value1
        statements
    case value2
        statements
```

`switch/case` Statements	`if/elseif` Statements
Easier to read	Can be difficult to read
Can compare strings of different lengths	You need `strcmp` to compare strings of different lengths
Test for equality only	Test for equality or inequality

```
    ...
    otherwise
        statements
    end
```

The end is only necessary after the entire switch block, not after each case. If you terminate the switch statement and follow it with a `case` statement you will get an error saying the use of the `case` keyword is invalid. If this happens it is probably because you deleted a loop or an `if` statement but forgot to delete the `end` that went with it, thus leaving you with surplus `end`s. Thus MATLAB thinks you ended the switch statement before you intended to.

```
>> method='linear';
switch(method)
   case 'linear'
      disp('Method is linear')
   case 'cubic'
      disp('Method is cubic')
end
```

```
>> number=1;
switch(number)
   case {52, 48,50}
      disp('This number is even')
   case {1, 2,3}
      disp('This number is odd')
end
```

Example 10 Write a script/function that converts a Roman numeral to its decimal equivalent. You should be able to handle the following conversion table:

Roman	Decimal
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

2.2 Linear algebra

Transpose. The quote operator $'$ is used to create the conjugate transpose of a vector (matrix) while the dot quote operator $.'$ creates the transpose vector (matrix). To illustrate this let us form a complex vector $a+i*b'$ and next apply these operations to the resulting vector to obtain

```
>> a=1:3

a =

     1     2     3

>> b=4:6

b =

     4     5     6

>> c=a+i*b

c =

   1.0000 + 4.0000i   2.0000 + 5.0000i   3.0000 + 6.0000i

>> c'

ans =

   1.0000 - 4.0000i
   2.0000 - 5.0000i
   3.0000 - 6.0000i

>> c.'

ans =

   1.0000 + 4.0000i
   2.0000 + 5.0000i
   3.0000 + 6.0000i
```

Size of a matrix Command `length` returns the number of components of a vector

```
>> a=1:3

a =

     1     2     3

>> length(a)
```

```
8
9  ans =
10
11      3
```

Command size returns the dimensions of a matrix

```
1  >> A=eye(3,2)
2
3  A =
4
5       1     0
6       0     1
7       0     0
8
9  >> size(A)
10
11 ans =
12
13      3     2
```

Dot operator The dot operator . is used for the componentwise application of the operator that follows the dot operator

```
1  >> a=1:4:9
2
3  a =
4
5       1     5     9
6
7  >> a.*a
8
9  ans =
10
11      1    25    81
```

The same result is obtained by applying the power operator $\wedge$ to the vector a

```
1  >> a.^2
2
3  ans =
4
5       1    25    81
```

Componentwise division of vectors a and b can be accomplished by using the slash operator / together with the dot operator .

```
>> a./a

ans =

     1     1     1
```

Dot product The scalar product between two vectors

$$c = \sum_{i=1}^{N} a_i b_i, \tag{2.1}$$

is calculated as

```
>> a=1:4:9;
>> b=1:3;
>> dot(a,b)

ans =

    38
```

Outer product The outer product between two vectors $\mathbf{a}$ and $\mathbf{b}$

$$C_{ij} = a_i b_j \tag{2.2}$$

is calculated as

```
>> a'*b

ans =

     1     2     3
     5    10    15
     9    18    27
```

Cross product The cross product of $\mathbf{a}$ and $\mathbf{b}$

$$\mathbf{a} \times \mathbf{b} = \det \begin{bmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{bmatrix}, \tag{2.3}$$

where $\mathbf{e}_1 = [1\ \ 0\ \ 0]$, $\mathbf{e}_2 = [0\ \ 1\ \ 0]$, $\mathbf{e}_3 = [0\ \ 0\ \ 1]$, is calculated as

```
>> cross(a,b)

```

```
ans =

    -3     6    -3
```

Solution of Linear Systems of Equations

A linear system of equations is a set of n linear equations in k variables (sometimes called "unknowns"). An example of a linear system of 3 equations in 3 variables is given by

$$\begin{aligned} 2\,x_1 - 1\,x_2 + 3\,x_3 &= 1 \\ 1\,x_1 + 3\,x_2 - 2\,x_3 &= 2 \\ 3\,x_1 + 0\,x_2 + 5\,x_3 &= 3 \end{aligned} \tag{2.4}$$

Linear systems can be represented in matrix form as the matrix equation,

$$\mathbf{A}\,\mathbf{x} = \mathbf{b}, \tag{2.5}$$

so that we have

```
 >> A=[2 -1 3;1 3 -2; 3 0 5]

A =

     2    -1     3
     1     3    -2
     3     0     5

>> b=[1;2;3]

b =

     1
     2
     3
```

There are many various ways of solving linear systems of equations. We can classify them as:

- **Direct methods:** If k=n and the matrix $\mathbf{A}$ is nonsingular, then the system has a unique solution in the n variables. In this case

$$\mathbf{x} = \mathbf{A^{-1}b}, \tag{2.6}$$

  where $\mathbf{A}^{-1}$ is the inverse of the matrix $\mathbf{A}$.

  In MATLAB the solution of a linear system of equations can be easily computed as

```
>> x=inv(A)*b

x =

    0.2857
    0.8571
    0.4286
```

Alternatively, the solution can be found using the following command

```
>> A\b

ans =

    0.2857
    0.8571
    0.4286
```

- **Iterative methods:** Because of the dimension of the systems and the sparsity properties, direct method based are not always feasible for the computation of accurate solutions and one has to use a so-called iterative method, where an approximate solution is updated in each step. The general idea is that we start with an arbitrary guess for the solution and then we use the equations to progressively improve the solution. The Generalized Minimum Residual method (with restarts)can be used within MATLAB as

```
K>> gmres(A,b)
gmres converged at iteration 3 to a
solution with relative residual 5e-016

ans =

    0.2857
    0.8571
    0.4286
```

Eigenvalues and eigenvectors

Many problems present themselves in terms of an eigenvalue problem:

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}$$

In this equation $\mathbf{A}$ is an n-by-n matrix, $\mathbf{v}$ is a non-zero n-by-1 vector and λ is a scalar (which may be either real or complex). Any value of λ for which this equation has a solution is known as an eigenvalue of the matrix $\mathbf{A}$. The vector $\mathbf{v}$ which corresponds to this value is called an eigenvector.

```
>> A=[0 1;-2 -3];
>> [v,d]=eig(A)

v =

    0.7071   -0.4472
   -0.7071    0.8944


d =

   -1     0
    0    -2
```

Solution of nonlinear equations

Generally, nonlinear algebraic problems are often exactly solvable, and if not they usually can be thoroughly understood through numeric analysis. As an example, the equation

$$x^2 + x - 1 = 0 \tag{2.7}$$

may be written as

$$f(x) = C \quad \text{where} \quad f(x) = x^2 + x \quad \text{and} \quad C = 1 \tag{2.8}$$

and is nonlinear. Though nonlinear, this simple example may be solved exactly (via the quadratic formula) and is very well understood. On the other hand, the nonlinear equation

$$x^5 - x - 1 = 0 \tag{2.9}$$

is not exactly solvable , though it may be qualitatively analyzed and is well understood, for example through making a graph and examining the roots of f(x) - C = 0.

Several iterative methods can be used to solve non linear equations, as the bisection method, the secant method and the Newton-Raphson method. These method can be easily implemented in MATLAB . In addition MATLAB provides a function `fzero` to find root of continuous function of one variable.

`fzero`. The MATLAB `fzero` finds zero of a function of one variable

```
>> x = fzero(@(x) x^5-x-1,0.1)

x =

    1.1673
```

where 0.1 is the initial guess.

Secant method. The secant method is a root-finding algorithm.

The steps to apply the secant method to find the root of an equation f(x) = 0 are

1. Use an initial guess of the root, x_i, to estimate the new value of the root x_{i+1} as
$$x_{i+1} = x_i - \frac{x_i - x_{i-1}}{f(x_i) - f(x_{i-1})} f(x_i). \tag{2.10}$$
2. Find the absolute relative approximate error, ε_a as
$$\varepsilon_a = \left(\frac{x_{i+1} - x_i}{x_{i+1}} \right). \tag{2.11}$$
3. Compare the absolute relative approximate error, ε_a with the pre-specified relative error tolerance, ε_s. If $\varepsilon_a > \varepsilon_s$, then go to step 2, else stop the algorithm. Also check if the number of iterations has exceeded the maximum number of iterations.

Example 11 Implement in MATLAB the secant method.

2.3 Specific programming in MATLAB

2.3.1 Writing and reading data – IO

In various applications, e.g. postprocessing of experimental data or analysing/visualisation of numerical results, the amount of data is large and keyboard input is rather inefficient. In these situations input and output is preferably accomplished via data files. When data is written to or read from a file it is important to use a correct data format. The data format is the key to interprete the contents of a file and must be known in order to correctly interpret the data in an input file. In general, there a two types of data files: *formatted* and *unformatted*. Formatted data files uses format strings to define exactly how and in what positions of a record the data is stored. Unformatted storage, on the other hand, only specifies the number format.

Writing to a file With the command `fprintf` you are able to write data in a formated way to files.

Example 12 Write the matrix `a` with the entries

```
100 2256
200 4564
300 3653
400 6798
500 6432
```

Specifier	Description
%c	single character
%d	Decimal notation (signed)
%e	Exponential notation (using a lowercase e as in 3.1415e+00)
%E	Exponential notation (using an uppercase E as in 3.1415E+00)
%f	Fixed-point notation
%g	The more compact of %e or %f. Insignificant zeros do not print
%G	Same as %g, but using an uppercase E
%i	Decimal notation (signed)
%o	Octal notation (unsigned)
%s	String of characters
%u	Decimal notation (unsigned)
%x	Hexadecimal notation (using lowercase letters a-f)
%X	Hexadecimal notation (using uppercase letters A-F)

Table 2.3: Conversion characters

to a file named `output.dat`.

```
>> output = fopen('output.dat','w');
>> b=a';
>> fprintf(output,'%3d %4d\n',b);
>> fclose(output);
```

Note that the command

```
fprintf('A unit circle has circumference %g.\n',2*pi)
```

displays a line on the screen.
Conversion characters specify the notation of the output This table lists the escape character sequences you use to specify non-printing characters in a format specification.

Reading from a file Suppose the data is stored in a file named `data.dat` which looks like that

```
100 2256
200 4564
300 3653
400 6798
500 6432
```

We can read the data with the following commands

```
>> input = fopen('data.dat','r');
>> a = fscanf(input,'%3d%4d');
>> fclose(input);
```

These commands

1. open a file with the name `data.dat` for reading (string `'r'`). The variable `input` is assigned a unique integer which identifies the file used.
2. read pairs of numbers (linewise) form the file `input`, one with 3 digits and one with 4 digits.
3. close the file with the identifier `input`

Note that this example produces a column vector `a` with elements, 100 2256 200 4564. . . 500 6432. This vector can be converted to a 5×2 matrix by the command

```
A = reshape(2,2,5)';
```

Formated files Some computer codes (Finite Element codes) and measurement instruments produce results in formatted data files. In order to read these results into MATLAB for further analysis the data format of the files must be known. Formatted files in ASCII format are written to and read from with the commands `fprintf` and `fscanf`. The command

Character	Description
\b	backspace
\f	form feed
\n	new line
\r	carriage return
\t	horizontal tab

Table 2.4: Escape characters

```
fprintf(output, 'format', variables)
```

writes variables an a format specified in string `'format'` to the file with identifier `output`

```
a = fscanf(input, 'format',size)
```

assigns to variable a data read from file with identifier `input` under format `'format'`.

Example 13 Study the available information and help on fscanf and fprintf commands. What is the mean- ing of the format string, `'%3dn'` ?

Example 14 Suppose a measurement system produces a record with 512 time vs. force stored on a file `'data.dat'`. Each reading is listed on a separate line according to a data format specified by the string, `'%8.6f %8.6f'`.

A set of commands reading time vs. force data from `'data.dat'` is,

1. Assign a namestring to a file identifier.

   ```
   >> fid1 = fopen('data.dat','r');
   ```

   The string `'r'` indicates that data is to be read (not written) from the file.

2. Read the data to a vector named `data` and close the file,

   ```
   >> data = fscanf(fid1, '%f %f');
   >> fclose(fid1);
   ```

3. Partition the data in separate time and force vectors,

   ```
   >> t = data(1:2:length(data));
   >> force = data(2:2:length(data));
   ```

   The force signal can be plotted in a lin-lin diagram,

   ```
   >> plot(t, press);
   ```

Unformated files Unformatted or binary data files are used when small- sized files are required. In order to interpret an unformatted data file the data precision must be specified. The precision is specified as a string, e.g., `'float32'`, controlling the number of bits read for each value and the interpretation of those bits as character, integer or floating point values. Precision `'float32'`, for instance, specifies each value in the data to be stored as a floating point number in 32 memory bits.

2.3.2 Improving the Speed of MATLAB Calculations

Large scale numerical calculations can put heavy demands on your computer.

MATLAB programs are interpreted. This would seem to make it inappropriate for large scale scientific computing. The power of MATLAB is realized with its extensive set of libraries which are compiled or are carefully coded in MATLAB to utilize "vectorization". The concept of vectorization is central to understanding how to write efficient MATLAB code.

Vectorized code takes advantage, wherever possible, of operations involving data stored as vectors. This even applies to matrices since a MATLAB matrix is stored (by columns) in contiguous locations in the computer's RAM. The speed of a numerical algorithm in MATLAB is very sensitive to whether or not vectorized operations are used.

This section presents some basic considerations to writing efficient MATLAB routines. The ideas presented below require little extra effort, yet can yield significant speed improvements. Once you understand how to use these procedures they will become natural parts of your MATLAB programming style.

Using vector operations instead of loops

Consider the following loop, translated directly from Fortran or C

```
dx = pi/30;
nx = 1 + 2*pi/dx;
for i = 1:nx
   x(i) = (i-1)*dx;
   y(i) = sin(3*x(i));
end
```

The preceding statements are perfectly legal MATLAB statements, but they are an inefficient way to create the x and y vectors. Recall that MATLAB allocates memory for variables on the fly. On the first time through the loop (i=1), MATLAB creates two row vectors x and y, each of length one. On each subsequent pass through the loop MATLAB appends new elements to x and y. Not only does this incur extra overhead in the memory allocation calls, the elements of x and y will not be stored in contiguous locations in RAM. Thus, any subsequent operations with x and y, even though these operations may be vectorized, will take a performance hit because of memory access overhead.

The preferred way to create the same two x and y vectors is with the following statements.

```
x = 0:pi/30:2*pi
y = sin(3*x);
```

The first statement creates the row vector, x, with 1 + pi/15 elements stored in contiguous locations in RAM. The second statement creates a new (matrix) variable, y, with the same number of rows and columns as x. Since x is a row vector, as determined by the preceding step, y is also a row vector. If x were, for example, a 5 by 3 matrix, then y = sin(3*x) would create a 5 by 3 matrix, y.

MATLAB is designed to perform vector and matrix operations efficiently. To take maximum advantage of the computer hardware at your disposal, therefore, you should use vectorized operations as much as possible.

Pre-allocating memory for vectors and matrices

Though MATLAB will automatically adjust the size of a matrix (or vector) it is usually a good idea to pre-allocate the matrix. Pre-allocation incurs the cost of memory allocation just once, and it guarantees that matrix elements will be stored in contiguous locations in RAM (by columns).

Consider the following sequence of statements.

```
dx = pi/30;
nx = 1 + 2*pi/dx;
nx2 = nx/2;

for i = 1:nx2
   x(i) = (i-1)*dx;
   y(i) = sin(3*x(i));
end

for i = nx2+1:nx
   x(i) = (i-1)*dx;
   y(i) = sin(5*x(i));
end
```

Since we know the size of x and y, a priori, we can pre-allocate memory for these vectors. Pre-allocation involves creating a matrix (or vector) with one vectorized statement before any of the matrix elements are referenced individually. The ones and zeros functions are typically used to pre-allocate memory.

Here is an improvement of the preceding statements with pre-allocation of memory for x and y.

```
dx = pi/30;
nx = 1 + 2*pi/dx;
nx2 = nx/2;

x = zeros(1,nx);      % pre-allocate row-vectors, x
y = zeros(1,nx);      % and y
```

```
7
8       for i = 1:nx2
9          x(i) = (i-1)*dx;
10         y(i) = sin(3*x(i));
11      end
12
13      for i = nx2+1:nx
14         x(i) = (i-1)*dx;
15         y(i) = sin(5*x(i));
16      end
```

The statements x(i) = ..., and y(i) = ... still do not take advantage of possibilities for vectorization, but at least the elements of x and y are stored contiguously in RAM. We will improve the efficiency of the loops shortly. First, however, note that we could have written the pre-allocation statements as

```
1       x = zeros(1,nx);       % pre-allocate row-vectors, x
2       y = x;                 % and y
```

The statement y = x does not mean that y will stay equal to x. It simply creates another matrix y with the same "shape" as x. Understanding that pre-allocation is important for efficiency will help you understand these apparently confusing twists of MATLAB programming logic.

We can further improve our loops by pulling the assignment of x out of the loops.

```
1       x = 0:pi/30:2*pi;      % vectorized calculation of x
2       nx = length(x);
3       nx2 = nx/2;
4
5       y = x;                 % pre-allocate memory for y
6
7       for i = 1:nx2
8          y(i) = sin(3*x(i));
9       end
10
11      for i = nx2+1:nx
12         y(i) = sin(5*x(i));
13      end
```

Finally, we observe that the calculation of y can also be vectorized. To do this we use the colon notation to refer to segments of the x and y vectors.

```
1       x = 0:pi/30:2*pi;      % vectorized calculation of x
2       nx = length(x);
3       nx2 = nx/2;
4
5       y = x;                              % pre-allocate memory for y
6
```

```
7       y(1:nx2) = sin(3*x(1:nx2));        % compute first part of y
8       y(nx2+1:nx) = sin(5*x(nx2+1:nx));  % and the second part
```

Whenever the speed of MATLAB code is important, there is no substitute for vectorization.

2.3.3 M-file programming

MATLAB provides a full programming language that enables you to write a series of MATLAB statements into a file and then execute them with a single command. You write your program in an ordinary text file (*editor window*), giving the file a name of `filename.m`. The term you use for filename becomes the new command that MATLAB associates with the program. The file extension of `.m` makes this a MATLAB m-file.
m-files can be scripts that simply execute a series of MATLAB statements, or they can be functions that also accept arguments and produce output. You create m-files using a text editor, then use them as you would any other MATLAB function or command.

Example 15 Create an m-file `exp1.m` using the *editor window*

```
1  function c = exp1(a,b);
2  c = sqrt((a.^2)+(b.^2));
```

Call the m-file from the command-line (or from another m-file)

```
1  a = 7.5;
2  b = 3.342;
3  c = exp1(a,b);
4  c =
5      8.2109
```

You might have problems with this because your path is not set correctly. The path is a collection of directories (folders) where MATLAB goes to look for programs to be run. If you saved that file first in, say, `Myfolder`, you should tell this to MATLAB by typing:

```
1  >> cd Myfolder
```

or opening the path browser utility from your MATLAB window.

2.3.4 Shell escape function

External functions, written in other programming languages like Fortran, C or C++ can be used via so called *shell escape functions* in an easy way.

Example 16 Write a vector to a file `output.dat`. Extend the file by some characters using (DOS) shell comands.

```
b=[1 2 3];
output = fopen('output.dat','w');
fprintf(output,'%4.2f\n',b)
fclose(output);
!echo 'This file was created today' >> output.dat
```

Note that shell escape functions allows to exchange data from [MATLAB] → [Fortran, C, C++, etc.] → [MATLAB] via a file interface. Obviously, this is not the most effective way to exchange data.

2.3.5 Calling C/C++ and Fortran Programs from MATLAB

Sometimes it is very useful to be able to combine MATLAB with other programming languages.

- You have a part of a source code written in e.g. C or C++ that you would like to embedd in your MATLAB code.
- MATLAB does not have the built-in functionallity you need and writing a function in MATLAB is not efficient enough.

The first situation is quite common in scientific programming. E.g. parts of large programming tools like finite element codes are quite general (e.g. assembling routines) and do not need to be rewritten all the time. Thus, many researchers have well tested and numerically efficient code that could be useful in a MATLAB program. The second situation is typically motivated by effciency reasons.
When extending MATLAB with a C/C++ or Fortran function subroutine the following steps must be taken:

1. Write the 'glue code' that interprets the communication between MATLAB and the C/C++ or Fortran subroutine.
2. Compile the 'glue code' and the function/subroutine with the MEX compiler in MATLAB .
3. Run the new function just like other functions in MATLAB .

Introduction to MEX-files

You can call your own C or Fortran subroutines from MATLAB as if they were built-in functions. MATLAB callable C and Fortran programs are referred to as MEX-files. MEX-files are dynamically linked subroutines that the MATLAB interpreter can automatically load and execute.
MEX-files have several applications:

- Large pre-existing C and Fortran programs can be called from MATLAB without having to be rewritten as M-files.
- Bottleneck computations (usually `for−loops`) that do not run fast enough in MATLAB can be recoded in C or Fortran for efficiency.

Note that MEX-files are not appropriate for all applications.

2.3.6 Calling MATLAB from C/C++

It can also be useful to do this the other way round, e.g. calling MATLAB from C/C++. This is typically in situation where you want to use MATLAB features that would be difficult or time consuming to implement in the existing code. A classic example is that you do the raw calculations in e.g C or C++ and use MATLAB to do the pre- and post-processing, e.g. visualization. We will not go into detail here, but claim that it will be much easier to understand this topic once you have understood the process of extending MATLAB .

2.3.7 Debugging and profiling

If a program doen not work as it should do, you usually debug it. Therefore, you can set breakpoints in functions (cf. degugging menue in the editor). If MATLAB reaches a breakpoint, MATLAB stops and you are able to inspect (and modify) all the active set of variables. All this can be done form the MATLAB command line. In general, debugging of a complex code (e.g. code not written by yourself) is very helpfull and could be used for your day-by-day programming excercises. More specific questions concerning debugging (breakpoints for error and warning conditions) can be found in the help on debug.
Sometimes a program spends most of its running time on just a few lines of code. These lines are then obvious candidates for optimization. You can find such lines by profiling, which keeps track of time spent on every line of every function. Profiling is also a great way to determine function dependencies (**who** calls **whom**). Get started by entering `profile viewer`, or find the profiler under the Desktop menu.

2.4 Postprocessing of data

2.4.1 2D Visualization

Basic plotting

MATLAB includes very powerful features for easy figure creation, plotting and image display. We will start by introducing the most basic features here.
Plotting commands: the most important commands for basic 2-dimensional plotting are illustrated in the following example:

Example 17 Plotting the sinus function

```
x=0:0.1:10;  % x ranges from 0 to 10 in steps of 0.1
y=sin(x);    % y contains the sin of each value of x
figure       % create an empty figure window
plot(y)      % plots all of the y values (the values on the x-axis are
             % indexes from 1 to 101)
plot(x,y)    % plots y against x (replacing the old plot on the same
             % figure, now showing the units of x on the x-axis)
close        % closes the figure window
```

In truth, since there wasn't yet a figure window, we could have skipped the `figure` command, as `plot` is smart enough to create one before showing the plot. The `plot` command has many additional possible arguments, as can be seen by typing `help plot`. There are also many other plotting commands for 2-dimensional plotting (`help graph2d`) and also for 3-D plotting (`help graph3d`). You can close a figure window by clicking on the upper right X box, using the `close` command on the figure's file menu, or by typing `close` at the command prompt.

The color and point marker can be changed on a plot by adding a third parameter (in single quotes) to the plot command. For example, to plot the above function as a red, dotted line, the m-file should be changed to:

```
    x = 0:0.1:100;
    y = 3*x;
    plot(x,y,'r:')
```

The third input consists of one to three characters which specify a color and/or a point marker type. The list of colors and point markers is as follows:

```
    y     yellow      .     point
    m     magenta     o     circle
    c     cyan        x     x-mark
```

```
4       r     red           +     plus
5       g     green         -     solid
6       b     blue          *     star
7       w     white         :     dotted
8       k     black         -.    dashdot
9                           --    dashed
```

You can plot more than one function on the same figure. Let's say you want to plot a sine wave and cosine wave on the same set of axes, using a different color and point marker for each. The following m-file could be used to do this:

```
1       x = 1:0.1:2*pi;
2       y = sin(x);
3       z = cos(x);
4       plot(x,y,'r', x,z,'gx')
```

By adding more sets of parameters to plot, you can plot as many different functions on the same figure as you want. When plotting many things on the same graph it is useful to differentiate the different functions based on color and point marker. This same effect can also be achieved using the hold on and hold off commands. The same plot shown above could be generated using the following m-file:

```
1       x = linspace(0,2*pi,50);
2       y = sin(x);
3       plot(x,y,'r')
4       z = cos(x);
5       hold on
6       plot(x,z,'gx')
7       hold off
```

Always remember that if you use the hold on command, all plots from then on will be generated on one set of axes, without erasing the previous plot, until the hold off command is issued.

Subplotting

More than one plot can be put on the same figure using the subplot command. The subplot command allows you to separate the figure into as many plots as desired, and put them all in one figure. To use this command, the following line of code is entered into the Matlab command window or an m-file:

```
1       subplot(m,n,p)
```

This command splits the figure into a matrix of m rows and n columns, thereby creating m*n plots on one figure. The p'th plot is selected as the currently active plot. For

instance, suppose you want to see a sine wave, cosine wave, and tangent wave plotted on the same figure, but not on the same axis. The following m-file will accomplish this:

```
x = linspace(0,2*pi,50);
y = sin(x);
z = cos(x);
w = tan(x);

subplot(2,2,1)
plot(x,y)
subplot(2,2,2)
plot(x,z)
subplot(2,2,3)
plot(x,w)
```

Changing the axis

Now that you have found different ways to plot functions, you can customize your plots to meet your needs. The most important way to do this is with the axis command. The axis command changes the axis of the plot shown, so only the part of the axis that is desirable is displayed. The axis command is used by entering the following command right after the plot command (or any command that has a plot as an output):

```
axis([xmin, xmax, ymin, ymax])
```

Adding text

Another thing that may be important for your plots is labeling. You can give your plot a title (with the title command), x-axis label (with the `xlabel` command), y-axis label (with the `ylabel` command), and put text on the actual plot. All of the above commands are issued after the actual plot command has been issued.

A title will be placed, centered, above the plot with the command: title('title string'). The x-axis label is issued with the following command: `xlabel`('x-axis string'). The y-axis label is issued with the following command: `ylabel`('y-axis string').

Furthermore, text can be put on the plot itself in one of two ways: the text command and the `gtext` command. The first command involves knowing the coordinates of where you want the text string. The command is `text(xcor,ycor,'textstring')`. To use the other command, you do not need to know the exact coordinates. The command is `gtext('textstring')`, and then you just move the cross-hair to the desired location with the mouse, and click on the position you want the text placed.

To further demonstrate labeling, take the step response plot from above. Assuming that

you have already changed the axis, copying the following lines of text after the axis command will put all the labels on the plot:

```
title('step response of something')
xlabel('time (sec)')
ylabel('position, velocity, or something like that')
gtext('unnecessary labeling')
legend('vel')
```

The text "unnecessary labeling" was placed right above the position, I clicked on.

Other commands that can be used with the plot command are:

- clf (clears the current plot, so it is blank)
- figure (opens a new figure to plot on, so the previous figure is saved)
- close (closes the current figure window)
- loglog (same as plot, except both axes are log base 10 scale)
- semilogx (same as plot, except x-axis is log base 10 scale)
- semilogy (same as plot, except y-axis is log base 10 scale)
- grid (adds grid line to your plot)

Of course this is not a complete account of plotting with Matlab, but it should give you a nice start.

Plotting - General remarks

The process of constructing a basic graph to meet your presentation graphics requirements is outlined in the following workflow. The table shows seven typical steps and some example code for each. If you are performing analysis only, you may want to view various graphs just to explore your data. In this case, steps 1 and 3 may be all you need. If you are creating presentation graphics, you may want to fine-tune your graph by positioning it on the page, setting line styles and colors, adding annotations, and making other such improvements.

1. Prepare your data

```
x = 0:0.2:12; y1 = bessel(1,x); y2 = bessel(2,x); y3 = bessel(3,x);
```

2. Select a window and position a plot region within the window

```
figure(1) subplot(2,2,1)
```

3. Call elementary plotting function

```
h = plot(x,y1,x,y2,x,y3);
```

4. Set axis limits, tick marks, and grid lines

```
axis([0 12 −0.5 1]) grid on
```

5. Annotate the graph with axis labels, legend, and text

```
xlabel('Time') ylabel('Amplitude')
legend(h,'First','Second','Third') title('Bessel Functions') [y,ix]
= min(y1); text(x(ix),y,'First Min \rightarrow',...
              'HorizontalAlignment','right')
```

6. Export graph

```
>> print −djpeg myfigure
>> print −dtiff myfigure
>> print myfigure
>> print −depsc myfigure
```

To copy the plot into Word you can select directly on the figure

```
Edit → Copy Figure
```

7. Editing plots. MATLAB formats a graph to provide readability, setting the scale of axes, including tick marks on the axes, and using color and line style to distinguish the plots in the graph. However, if you are creating presentation graphics, you may want to change this default formatting or add descriptive labels, titles, legends and other annotations to help explain your data. MATLAB supports two ways to edit the plots you create:

   - Using the mouse to select and edit objects interactively.
   - Using MATLAB functions at the command-line or in an M-file.

```
>> figure
>> hold on
>> h1=plot(x,cos(x))
>> h=plot(x,sin(x))
```

```
>> set(h1,'LineWidth',2,'LineStyle','.','Color','g')
>> set(h,'LineWidth',2,'LineStyle','^','Color','r')
>> hold off
```

Using Multiple X- and Y-Axes

```
x1=0:0.01:2*pi;
x2=0:0.01:3*pi;
y1=sin(x1);
y2=10*cos(x2);

figure
hold on
hl1 = plot(x1,y1,'Color','r');
ax1 = gca;  %get current axis
set(ax1,'XColor','r','YColor','r') ax2 =
axes('Position',get(ax1,'Position'),...
           'XAxisLocation','top',...
           'YAxisLocation','right',...
           'Color','none',...
           'XColor','k','YColor','k');
hl2 = line(x2,y2,'Color','k','Parent',ax2) set(ax1,'Xlim',[0 2*pi])
set(ax2,'Xlim',[0 2*pi])
hold off
```

Plotting - Simple graphics

Besides the MATLAB command `plot(x,y)` there are further basic commands which can be used for plotting and visualisation

Example 18 Visualise a finite element mesh consisting of three quadrilaterals and store the result, i.e. the picture, to an encapsulated postscript file `mesh.eps`.

```
% ------------------------------------------------------------------------
% prints some finite elements in a simple way
% ------------------------------------------------------------------------
figure

ele     = [1 2 3 4;4 3 5 6;6 5 7 8];
num_ele = size(ele,1);
nod     = [0 0;1 0;1 1;0 1;1 2; 0 2;1 3;0 3];
num_nod = size(nod,1);

MA=max(nod); MI=min(nod);

set(gca,'xlim',[MI(1)-2 MA(1)+2],'ylim',[MI(2)-1 MA(2)+1],'Visible','off')
% ------------------------------------------------------------------------
% 'visible','off'     ->    prevents axis lines, tick marks,
```

```
%                              and labels from being displayed
% ------------------------------------------------------------------------

for i=1:num_ele
   nodes(1:4,1:2) = nod(ele(i,1:4),1:2);       % local coordinates
   patch(nodes(:,1),nodes(:,2),[1 1 0.6]);     % print one element
end
print -depsc -tiff -r200 mesh.eps
```

Editing plots

MATLAB formats a graph to provide readability, setting the scale of axes, including tick marks on the axes, and using color and line style to distinguish the plots in the graph. However, if you are creating presentation graphics, you may want to change this default formatting or add descriptive labels, titles, legends and other annotations to help explain your data. MATLAB supports two ways to edit the plots you create:

- Using the mouse to select and edit objects interactively.
- Using MATLAB functions at the command-line or in an M-file.

2.4.2 Contour plots

A filled (2-DIM) contour plot displays isolines calculated from matrix z and fills the areas between the isolines using constant colors. The color of the filled areas depends on the current figure's colormap.

Example 19 To view a contour plot of the function $z = x\ \exp(-x^2 - y^2)$ over the range $-2 \leq x \leq 2$ and $-2 \leq y \leq 3$

```
[X,Y] = meshgrid(-2:.05:2,-2:.05:3);
Z = X.*exp(-X.^2-Y.^2);
[C,h] = contourf(X,Y,Z,25);
colormap hsv
print -depsc -tiff -r300 contourf.eps
close
[C,h] = contour(X,Y,Z,15);
print -depsc -tiff -r300 contour.eps
close
```

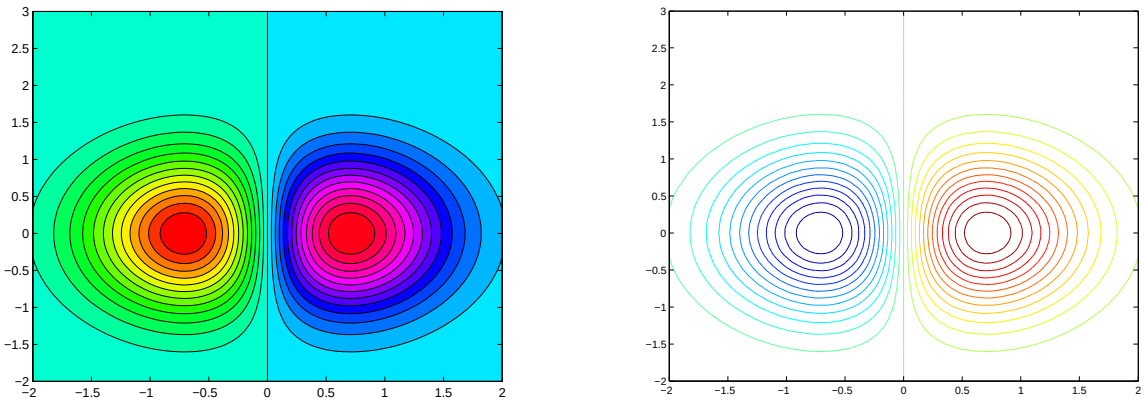


Figure 2.1: Example 19: Output of the `contour` and `contourf` example printed by the command: `print −depsc −tiff −r300 myfile`

2.4.3 Visualisation in 3-dim

Example 20 A 3-dim surface plot using the `subplot` command for two windows is shown next

```
x=(1:.1:7)';
y=x;
z=cos(x)*sin(y)';
% surface plot
subplot(2,1,1);          % subplot 1
surf(x,y,z);             % surface
subplot(2,1,2);          % subplot 1
[CS,H]=contour(x,y,z);   % contour
clabel(CS,H)             % labeling
```

2.4.4 Creating movies

The MATLAB command `addframe(avifile,frame)` appends the data in frame to the AVI file identified by the variable `avifile`, which was created by a previous call to the file `output.avi`.

Example 21 Create a movie of an evolving sinus curve in the intervall $4\,\pi$.

```
clc
clear all
mov = avifile('output.avi')
x = 0:0.1:4*pi;
for k=1:length(x)
```

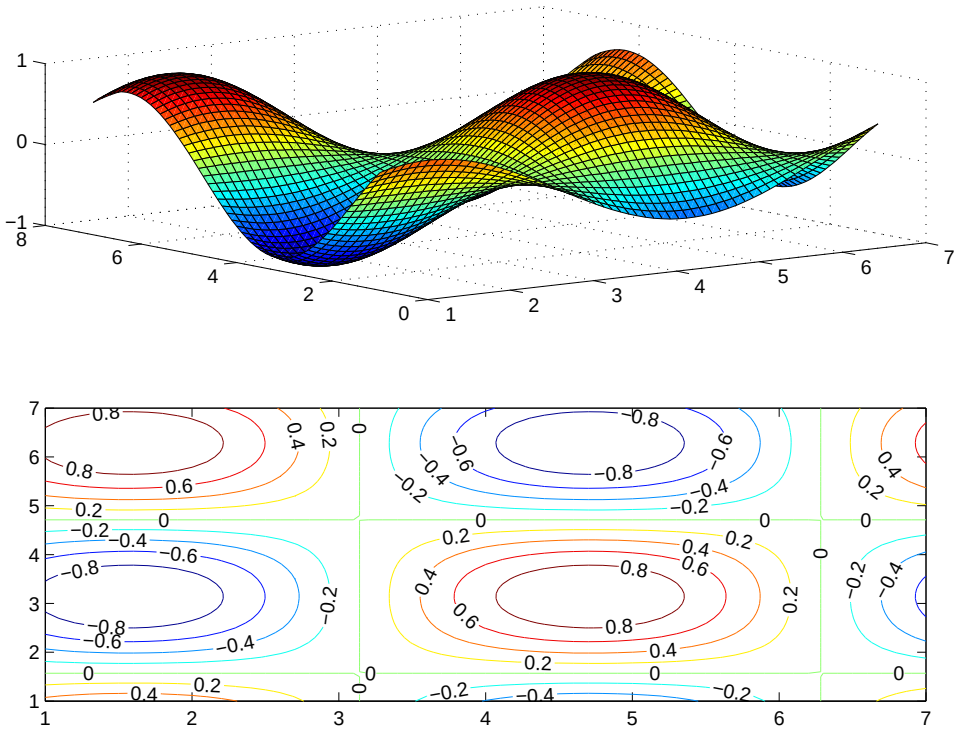


Figure 2.2: Example 20: 3-dim surface and contour plot with labels printed by the command: `print -depsc -tiff -r300 myfile`

```
    xx(1) = 0;
    yy(1) = 0;
    xx(k) = x(k);
    yy(k) = sin(x(k));
    h=plot(xx,yy);
    axis([0 4*pi -1.5 1.5])
    set(h,'color','m','linewidth',2,'marker','o')
    F=getframe;
    mov = addframe(mov,F);
end
mov = close(mov);
```

2.5 MATLAB toolboxes - The curve fitting toolbox

A common task is to make sense of data from experimental observation, i.e. mechanical tests. A prominent example in mechanical engineering is the experimental observation of a load-displacement curve in a uniaxial tension test. Typically, we have an array (typically an ASCII-file from the measurement software) containing in the first row the displacements applied to the speciment while we have in the second row the associated mechanical forces which are measured by a load cell.

`cftool` displays a window for fitting curves to (experimental) data. You can create a data set using data in your workspace and you can create graphs of fitted curves superimposed on a scatter plot of the data. The basic command to open the GUI is

```
cftool(x,y)
```

while `x` and `y` are vectors containing the data. Obviously, `x` and `y` must have the same length.

Example 22 Fit a set of given data in the vectors `x` and `y` by the curve fitting toolbox `cftool` using a quadratic polynomial.

```
% randomized quadratic function
x = (0:10)';
y = [0.1;1.1;3.6;10.1;17.2;23.9;38.1;51.0;67.5;80.4;99.0];
cftool(x,y);
```

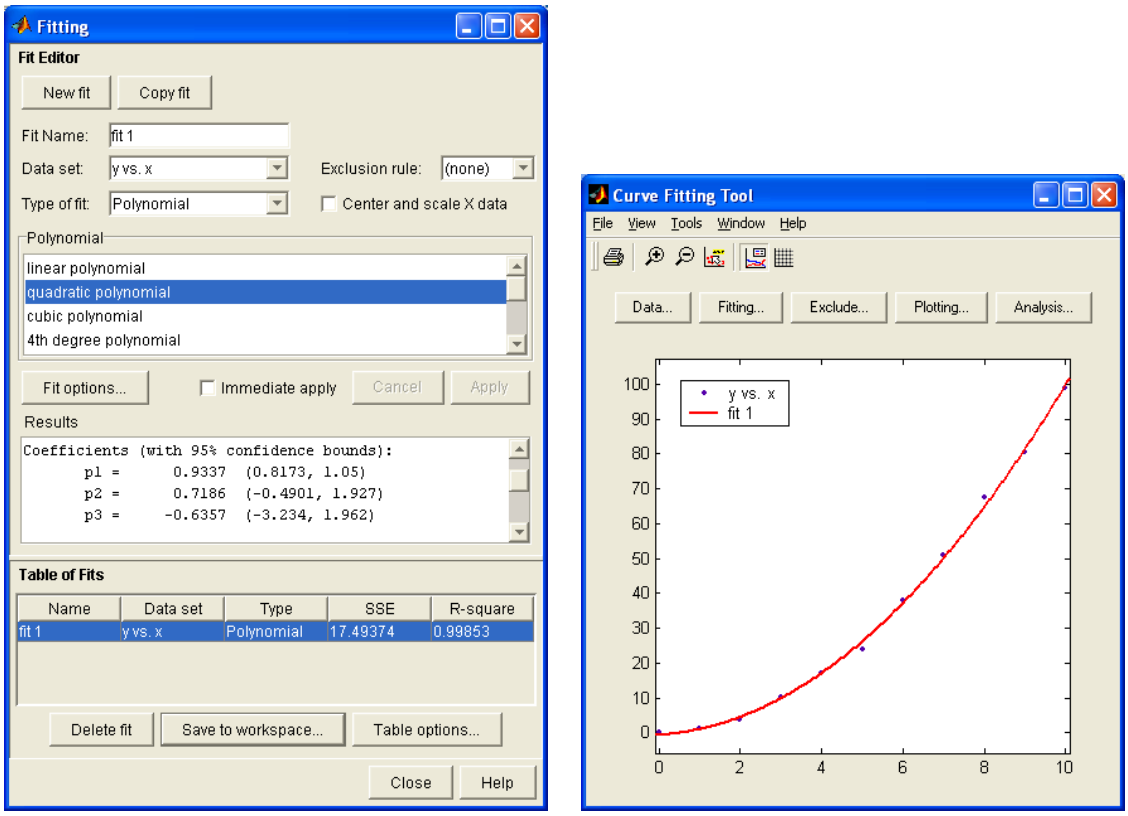


Figure 2.3: GUI of the curve fitting tool box `cftool`.

2.5.1 Curve-fitting without the GUI

In certain cases, curve fitting should be run from a shell window, e.g. for automization purposes. Thus we trace back to the same example as mentioned but include the fitting

part in a m-file.

Fitting data to a polynomial

The above dataset may be fitted to a polynomial of degree n by use of the function `polyfit`. Again, the error is minimized in a least square-sense.

```
A = polyfit(y,x,n)}
```

returns the coefficients a_i with $i = 0, \ldots, n$ of the polynomial $y(x)$ of order n defined as

$$y(x) = a_n\, x^n + a_{n-1}\, x^{n-1} + \ldots + a_2\, x^2 + a_1\, x + a_0$$

with the solution vector containing the unknown coefficients

$$A = [\, a_n,\, a_{n-1},\, \ldots, a_2,\, a_1,\, a_0\,].$$

Example 23 Find the coefficients of a second order polynomial

```
x = (0:10)';                    %
y = [0.1;1.1;3.6;10.1;17.2;23.9;38.1;51.0;67.5;80.4;99.0];
%y = x.*x + 0.4*rand(10,1);  % artifical data
plot(x,y,'om');                 % visualisation
hold on;
a = polyfit(x,y,2);             % find coefficients a_i
f2= polyval(a,x);               % create polynomial
p = [1 0 0];                    % non-randomized solution
f = polyval(p,x);               % create polynomial
plot(x,f2,'r');                 % visualisation
plot(x,f, 'b');                 % visualisation
legend('data','fit 2nd order polynomial','exact polynomial');
```

Fitting data to an arbitrary function

An alternative to the function `polyfit()` is the more general function `fit(x,y,'ltype')` or `fit(x,y,ftype)`, where the second one fits the data to the fit type object specified by `ftype`. You create a fit type object with the `fittype` function.

```
x = (0:10)';                    %
y = [0.1;1.1;3.6;10.1;17.2;23.9;38.1;51.0;67.5;80.4;99.0];
% using predefined functions (see help cflibhelp)
fit(x,y,'poly2');
% used-defined function
ftype=fittype('a+b*x+c*x*x','testproblem');
fit(x,y,ftype);
```

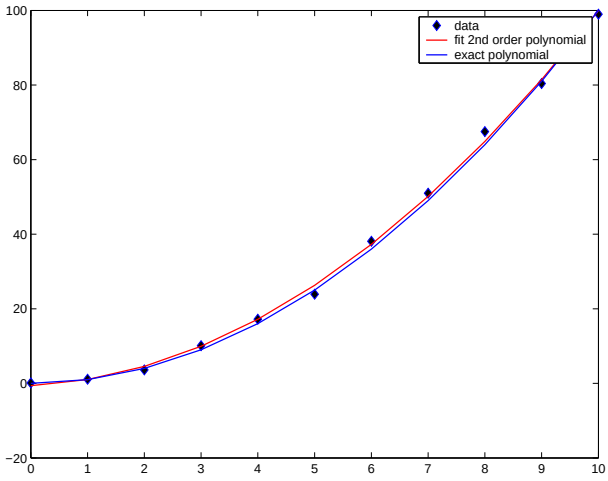


Figure 2.4: Output of the curve-fitting

When the form of the relationship is a linear combination of basis functions and the parameters are the coefficients, the resulting optimization is a *linear* least-squares problem and can be solved using the `backslash`.

Example 24 Find the coefficients of the following function

$$y(x) = a_0 + a_1 \cos(x) + a_2 \sin(x)$$

```
x = (1:100)'
y = 4 + 0.1*cos(x) - 0.2*sin(x);
y = y + 0.1*randn(100,1);
ftype=fittype('a+b*cos(x)+c*sin(x)','testproblem');
fit(x,y,ftype);
```

or

```
x = (1:100)';
y = 4 + 0.1*cos(x) - 0.2*sin(x);
y = y + 0.1*randn(100,1);
A = [ ones(100,1) cos(x) sin(x) ];
A\y
ans =
  4.0048
  0.0938
-0.1951
```

Note that the coefficients are not identical, which is caused by the different mathematical optimization techniques and the inherent tolerances.

2.6 MATLAB toolboxes - The ODE toolbox

Before going into mathematical details, we would like to point out that we have to be very carefull using the ODE toolbox as a *black box solver* for various ODEs. Note that you have to analyze carefully the *type* of ODE before you use a certain numerical method to solve it. W.r.t the numerical method, it is e.g. important to distinguish so-called *stiff* and *non-stiff* ODEs. A *stiff* ODE is characterized by a problem

- which can be solved not by or only with ineffective small increments with explicit techniques,
- which has a big change in the solution in very small regions δx (steep gradients), but has to be solved in large domains Δx with $\Delta x >> \delta x$.

Here, we discuss only functions which can be used for *non-stiff* problems. Note that in MATLAB also solvers for *stiff* problems are included `ode15s, ode23s, ode23t` and `ode23tb`. from the numerical point of view, the function `ode45` is

- an explicit embedded Runge-Kutta scheme
- a one-step method
- handy for a *first try* of various problems

2.6.1 1st order ODE

MATLAB has an extensive library of functions for solving ordinary differential equations. Note that we have only the time in this short course to show a very basic introduction. Therefore, we will focus on the main two, the built-in functions `ode23` and `ode45`, which implement versions of Runge-Kutta 2nd/3rd-order and Runge-Kutta 4th/5th-order, respectively.

Example 25 Solve the following ODE of 1st order

$$\frac{\mathrm{d}y}{\mathrm{d}x} = x\,y^2 - y, \qquad y_0 := y(x=0) = 1,$$

in the intervall $x \in [\,0, 0.5\,]$!

For any differential equation in the form $\mathrm{d}y/\mathrm{d}x = y' = f(x, y)$, we begin by defining the function $f(x, y)$. Note that for single equations, we can define $f(x, y)$ as an inline function which simplifies the procedure. The basic usage for MATLAB 's solver `ode45` is

```
ode45(function,domain,initial condition)
```

Thus, we can formulate the problem as

```
    f  = inline('x*y^2+y');
[x,y] = ode45(f,[0.0 0.5],1);
plot(x,y)
```

MATLAB returns two column vectors, the first with values of `x` and the second with values of `y` which can be plotted with the standard command `plot(x,y)`. Approximating this solution numerically, the algorithm `ode45` has selected a certain discretization of the interval $[\,0, 0.5\,]$, and MATLAB has returned a value of `y` at each point in the partition. It is often the case that we would like to specify the partition, i.e. the numerical effort, on which MATLAB returns an approximation. For example, we might only want to approximate the solution at 5 points: $y(.1), y(.2), \ldots, y(.5)$. We can specify this by entering the vector `[0,.1,.2,.3,.4,.5]` as the domain in `ode45`. That is, we use

```
ic    = 1.0;
disc  = 0:0.1:0.5;
    f  = inline('x*y^2+y');
[x,y] = ode45(f,disc,ic);
plot(x,y)
```

Example 26 Solve the same ODE in the domain $x \in [\,0, 0.9\,]$ with a finer but an equidistant discretization of $\Delta x = 0.01$. Use a m-file `firstode.m` to define the function.

```
function yprime = firstode(x,y);
yprime = x*y^2 + y;
```

and the main program calling this m-file looks like

```
disc  = 0:0.01:0.9;
ic    = 1.0;
[x,y] = ode45(@firstode,disc,ic);
```

2.6.2 2nd order ODE

Next we would like to discuss a typical ODE arising in various problems e.g. in mechanics. This ODE is of second order and describes e.g. the behaviour of a pendelum with one degree of freedom.

$$m\,\ddot{x}(t) + c\,x(t) = f(t)$$

with the force described by

$$f(t) = 10 \qquad t > 0.$$

The (two) initial conditions are given as $x(0) = 0$ and $\dot{x}(0) = 0$. The stiffness of the spring c is given as $c = 1.5\,e3$ N/m and the mass m is given as $5\,e3$ kg. First we have to transform the ODE of 2nd order into a system of ODEs of first order depending on the position x and the velocity v

$$\begin{aligned} \dot{x} &= v, \\ \dot{v} &= \frac{1}{m}\left[f(t) - c\,x\right]. \end{aligned}$$

or in a generalized form, replacing $x \equiv y_1$ and $v \equiv y_2$

$$\begin{aligned} \dot{y}_1 &= y_2, \\ \dot{y}_2 &= \frac{1}{m}\left[f(t) - c\,y_1\right]. \end{aligned}$$

To solve this ODE, we define the function with the m-file `secode.m`

```
c = 1.5e3;
m = 5.0e3;
function dy = secode(t,y)
dy = [y(2); 1/m * (f(t) - c * y(1))]
```

and the function `F` with the m-file `F.m`

```
function f = F(t)
% f = 10;
f = 20*t*(t<=5) + 20*(10-t) * (t>5 & t<=10);
```

The problem can now be solved with

```
disc  = 0:0.01:30;
ic    = [0 0];
[t,y] = ode45('secode',disc,ic)
plot(t,y(:,1),'r')
hold on
plot(t,y(:,1),'b')
legend('position x','velocity v')
title('pendelum')
xlabel('time [t]')
ylabel('position/velocity')
```

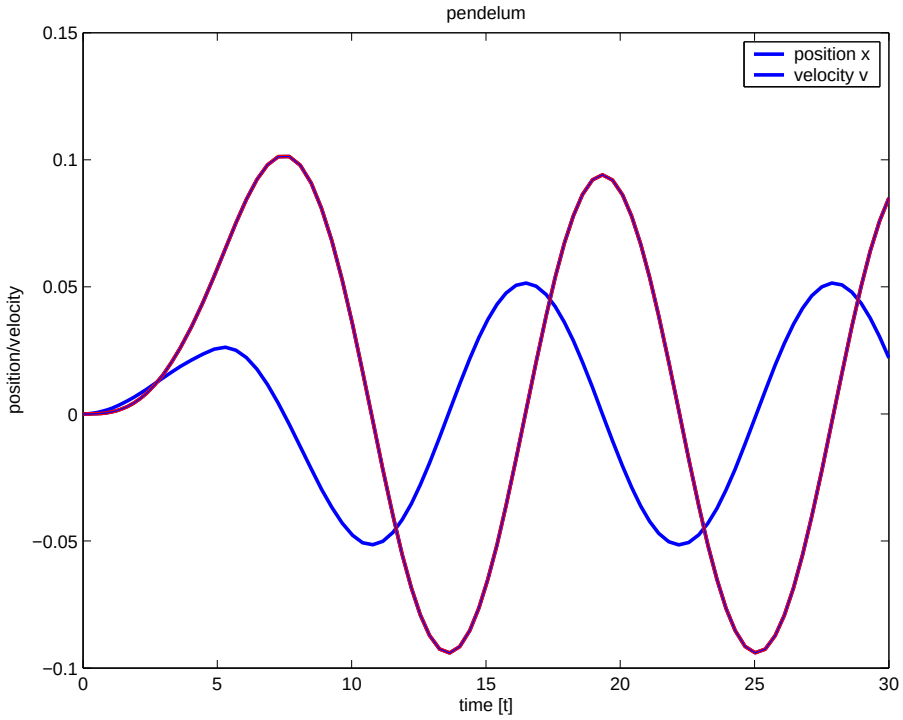


Figure 2.5: Solution, i.e. position and velocity of the mechanical pendelum

2.7 Exercises MATLAB

Exercise 1

What is the greatest value of n that can be used in the sum

$$1^2 + 2^2 + 3^2 + \ldots + n^2$$

and gives a total value of the sum less than 100?

Exercise 2

Solve the following system of linear equations

$$\begin{aligned} 2\,x_1 - 1\,x_2 + 3\,x_3 &= 1 \\ 1\,x_1 + 3\,x_2 - 2\,x_3 &= 2 \\ 3\,x_1 + 0\,x_2 + 5\,x_3 &= 3 \end{aligned}$$

using both a direct and an iterative method.

Exercise 3

Construct an array with 10 000 entries where. Compute the solution vector $f(x_i)$, with $x_i = 1, \ldots, 10\,000$ as fast as possible!

$$f(x_i) = \frac{x_i}{\sin(x_i) + 2}$$

Exercise 4

Solve the following non linear equation

$$x^5 - x - 1 = 0$$

using both the bisection method and the MATLAB function `fzero`.
Hint. The bisection method is a root-finding algorithm which repeatedly divides an interval in half and then selects the subinterval in which a root exists.
Suppose we want to solve the equation $f(x) = 0$. Given two points a and b such that $f(a)$ and $f(b)$ have opposite signs, we know by the intermediate value theorem that f must have at least one root in the interval [a, b] as long as f is continuous on this interval. The bisection method divides the interval in two by computing $c = (a + b)/2$. There are now two possibilities: either $f(a)$ and $f(c)$ have opposite signs, or $f(c)$ and $f(b)$ have opposite signs. The bisection algorithm is then applied recursively to the sub-interval where the sign change occurs.

Exercise 5

Solve the following non linear equation

$$x^5 - x - 1 = 0$$

using both the Newton-Raphson method.

Hint. The steps to apply Newton-Raphson method to find the root of an equation f(x) = 0 are

1. Evaluate $f'(x)$ symbolically
2. Use an initial guess of the root, x_i, to estimate the new value of the root x_{i+1} as

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}. \tag{2.12}$$

3. Find the absolute relative approximate error, ε_a as

$$\varepsilon_a = \left(\frac{x_{i+1} - x_i}{x_{i+1}} \right). \tag{2.13}$$

4. Compare the absolute relative approximate error, ε_a with the pre-specified relative error tolerance, ε_s. If $\varepsilon_a > \varepsilon_s$, then go to step 2, else stop the algorithm. Also check if the number of iterations has exceeded the maximum number of iterations.

Exercise 6

Given the array `A = [ 2 4 1 ; 6 7 2 ; 3 5 9]`, provide the commands needed to

1. assign the first row of `A` to a vector called `x1`
2. assign the last 2 rows of `A` to an array called `y`
3. compute the sum over the columns of `A`
4. compute the sum over the rows of `A`
5. compute the standard error of the mean of each column of `A` (the standard error of the mean is defined as the standard deviation divided by the square root of the number of elements used to compute the mean.)

Exercise 7

Given `x = [3 15 9 12 −1 0 −12 9 6 1]`, provide the command(s) that will

1. set the values of x that are positive to zero
2. set values that are multiples of 3 to 3 (rem will help here)
3. multiply the values of x that are even by 5
4. extract the values of x that are greater than 10 into a vector called y
5. set the values in x that are less than the mean to zero
6. set the values in x that are above the mean to their difference from the mean

Exercise 8

Evaluate the following function `y(x)` using only logical indexing:

```
y(x) = 2          if x  < 6
     = x − 4      if 6  ≤ x < 20
     = 36 − x     if 20 ≤ x ≤ 35
```

You can check your answer by plotting `y` vs. `x` with symbols. The curve should be a triangular shape, always above zero and with a maximum of 16. It might also be useful to try setting `x` to 1:35. Using multiple steps (or a simple m-file) is recommended for this problem.

Exercise 9

Given the vector `x = [1 8 3 9 0 1]`, create a short set of commands that will

1. add up the values of the elements (Check with `sum`)
2. computes the running sum (for element `j`, the running sum is the sum of the elements from 1 to `j`, inclusive. Check with `cumsum`)
3. computes the sine of the given x-values (should be a vector)

Exercise 10

Create an M-by-N array of random numbers (use `rand`). Move through the array, element by element, and set any value that is less than 0.2 to 0 and any value that is greater than (or equal to) 0.2 to 1.

Exercise 11

Given `x = [4 1 6]` and `y = [6 2 7]`, compute the following arrays

1. aij = xiyj
2. bij = xi/yj
3. ci = xiyi, then add up the elements of c

4. dij = xi/(2 + xi + yj)
5. eij = reciprocal of the lesser of xi and yj

Exercise 12

Write a script that will use the random-number generator rand to determine the following:

1. The number of random numbers it takes to add up to 20 (or more).
2. The number of random numbers it takes before a number between 0.8 and 0.85 occurs.
3. The number of random numbers it takes before the mean of those numbers is within 0.01 of 0.5 (the mean of this random-number generator).

It will be worthwhile to run your script several times because you are dealing with random numbers. Can you predict any of the results that are described above?

Exercise 13

Write a script that asks for a temperature (in degrees Fahrenheit) and computes the equivalent temperature in degrees Celcius. The script should keep running until no number is provided to convert.
Remark: the function `isempty` will be useful here.

Exercise 14

From a Finite Element analysis we have stored the results (nodal forces (f_x, f_y) and nodal displacements u_x, u_y) at two finite element nodes (3126) and (3127) in a complex data format. The name of the file is `input.dat`. Read the forces and dislacements for all load steps and store them in vectors (`f` and `u`). Plot all load displacement curves using the command `plot(a,b)` and write the data in new files.

Exercise 15

For a Finite Element analysis you need various input files describing the boundary value problem. The discretization, i.e. the Finite Element mesh, of a 2-dim domain under consideration is stored in two ASCII files containing the coordinates of the nodal points

(`Nodes.inp`) and the topology of the Finite Elements (`Elements.inp`). Try to open these files and store the information in two matrices in the following format:

```
10.73  15.95
11.08  16.90
11.98  18.10
12.78  18.70
...
```

and

```
689      412      130      874      875      876
456        7      240      877      878      879
409      238      129      880      881      882
...
```

Write the data in the new format in two files named `new-elements.dat` and `new-nodes.dat`.

Exercise 16

Compute the value of π using the following series

$$\pi = 4 \sum_{k=0}^{m} \frac{(-1)^k}{2k+1} \tag{2.14}$$

How many terms are needed to obtain an accuracy of $1e-3$? How accurate is the sum of 100 terms of this series?

Exercise 17

The Fibonacci numbers are computed according to the following relation:

$$F_n = F_{n-1} + F_{n-2}$$

with $F_0 = 0$ and $F_1 = 1$.

1. Compute the first 10 Fibonacci numbers.
2. For the first 50 Fibonacci numbers, compute the ratio

$$\frac{F_n}{F_{n-1}}$$

It is claimed that this ratio approaches the value of the golden mean $\frac{1+sqrt(5)}{2}$. What do your results show?

Exercise 18

We found the following algorithm on a web page for computing π:

1. Set a = 1, $b = 1/\sqrt{2}$, t = 1/4 and x = 1
2. Repeat the following commands until the difference between a and b is within some desired accuracy:

```
y = a
a = (a + b)/2
b = sqrt(b*y)
t = t - x*(y - a)^2
x = 2*x
```

3. From the resulting values of a, b and t, an estimate of π is

$$\pi_{est} = \frac{(a+b)^2}{4*t}$$

How many repeats are needed to estimate π to an accuracy of 1e-4? Compare the performance of this algorithm to the result from the former exercise.

Exercise 19

Write a script that asks for an integer (n) and then computes the following based on the value of the integer:

While the value of n is greater than 1, replace the integer with half of its value, $n/2$, if the integer is even. Otherwise, replace the integer with three times its value, plus 1, $3*n+1$.

Make provision to count the number of values in (or the length of) the sequence that results.

Example calculation: If n = 10, the sequence of integers is 5, 16, 8, 4, 2, 1 and so the length is 6.

Make a plot of the length of the sequence that occurs as a function of the integers from 2 to 30. For example, when n = 10, the length is 6 while for n = 15, the length is 17. Is

there any pattern? Try larger numbers to see if any pattern occurs. Is there any integer for which the sequence does not terminate?

Exercise 20

Write a script/function that converts a Roman numeral to its decimal equivalent. There are two distinct situations that you might design your program for:

- The "old" style where the order of the symbols does not matter. In this case, IX and XI both mean 10 + 1 or 11. You should be able to handle the following conversion table:

Roman	Decimal
I	1
V	5
X	10
L	50
C	100
D	500
M	1000

- The "new" style where the order of the symbols does matter. For example, IX is 9 (10 - 1), XC is 90 (100 - 10). The conversion table given above still holds.

The function input will be useful here. The format

```
1     >> str = input('Roman numeral: ','s')
```

will provide a way to get the Roman number into your program as a string.

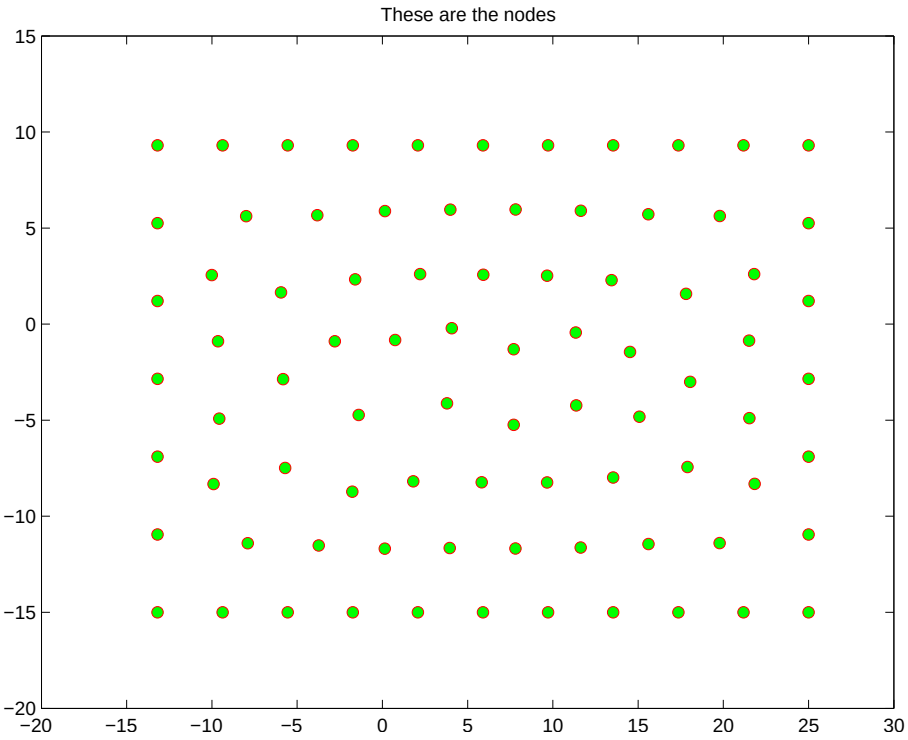
Exercise 21

Write a function that will do the inverse of the previous problem - convert a decimal number into a Roman number.

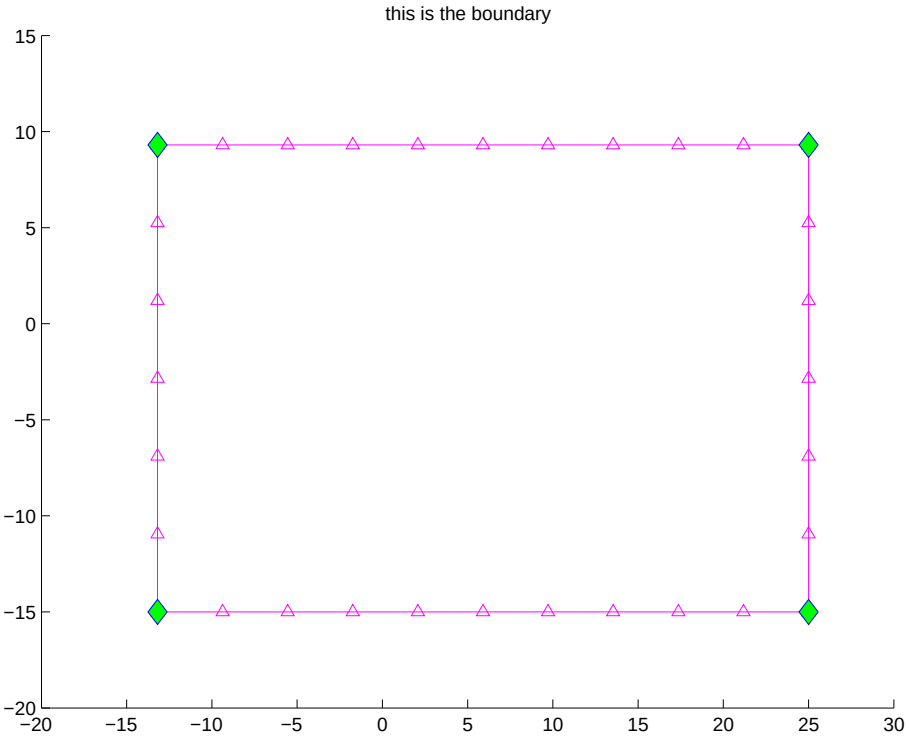
Exercise 22

- Open the file `InputCAE.inp`. Extract the coordinate of the nodes and the element connectivity.

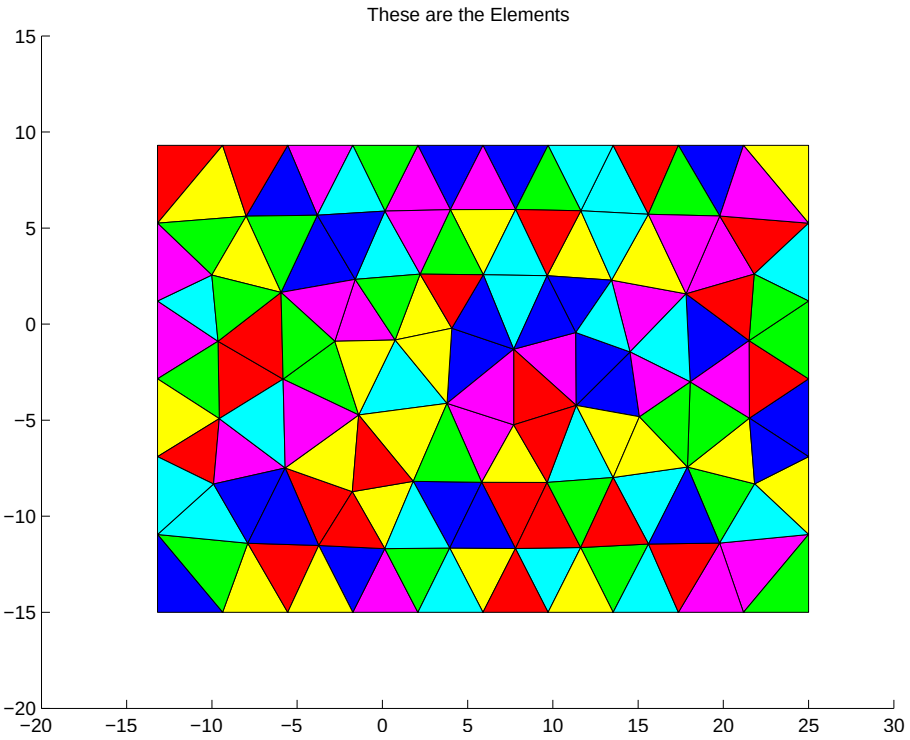
- Plot the nodes on a figure with markers



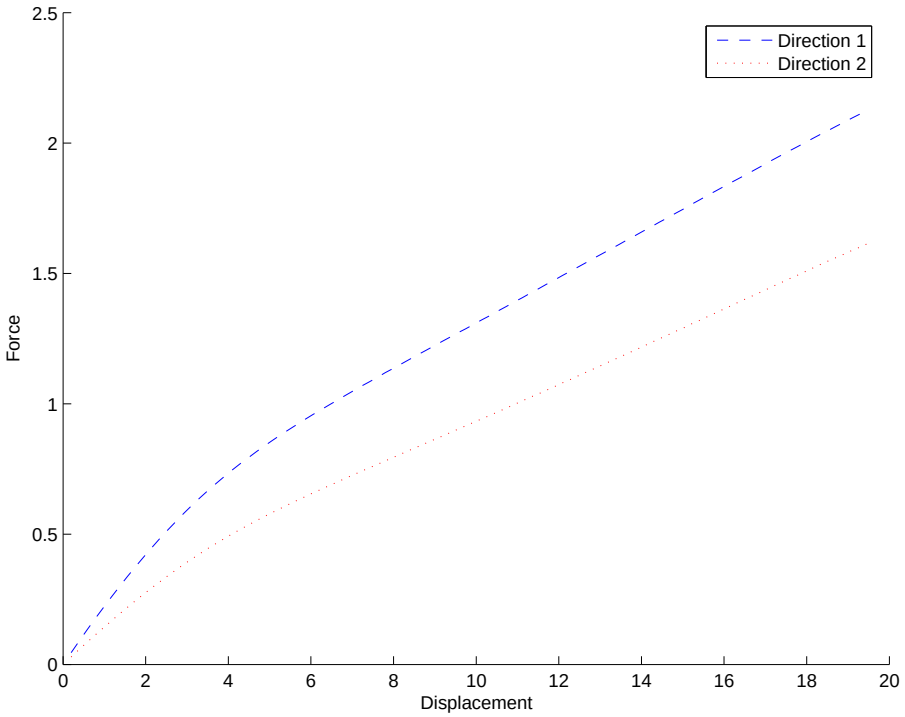
- Plot the boundaries of the mesh with a continuous line



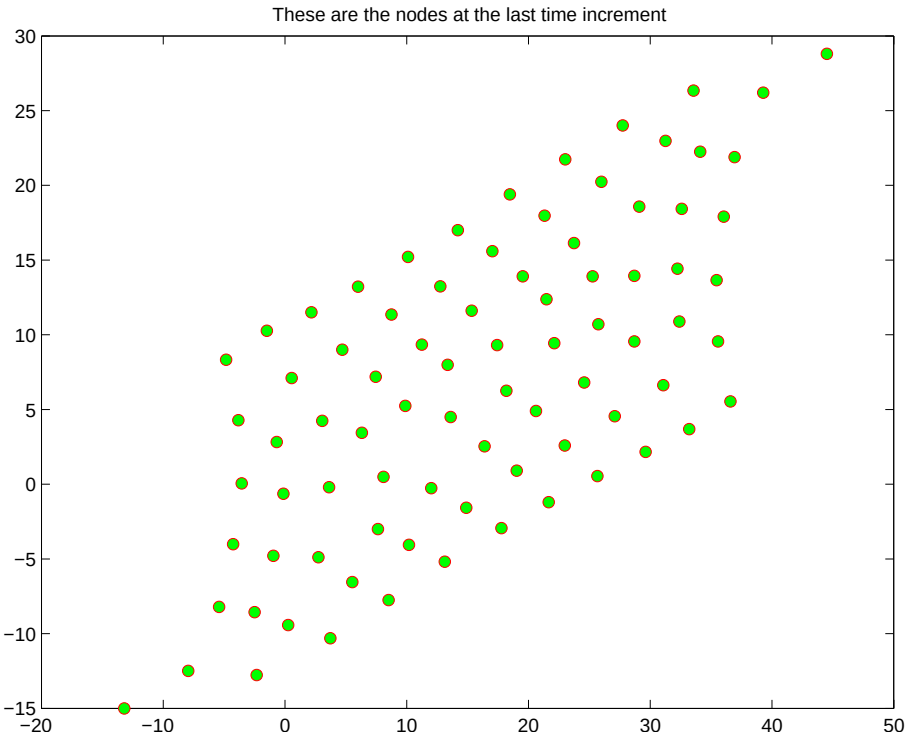
- Plot all the elements with filled two-dimensional triangle



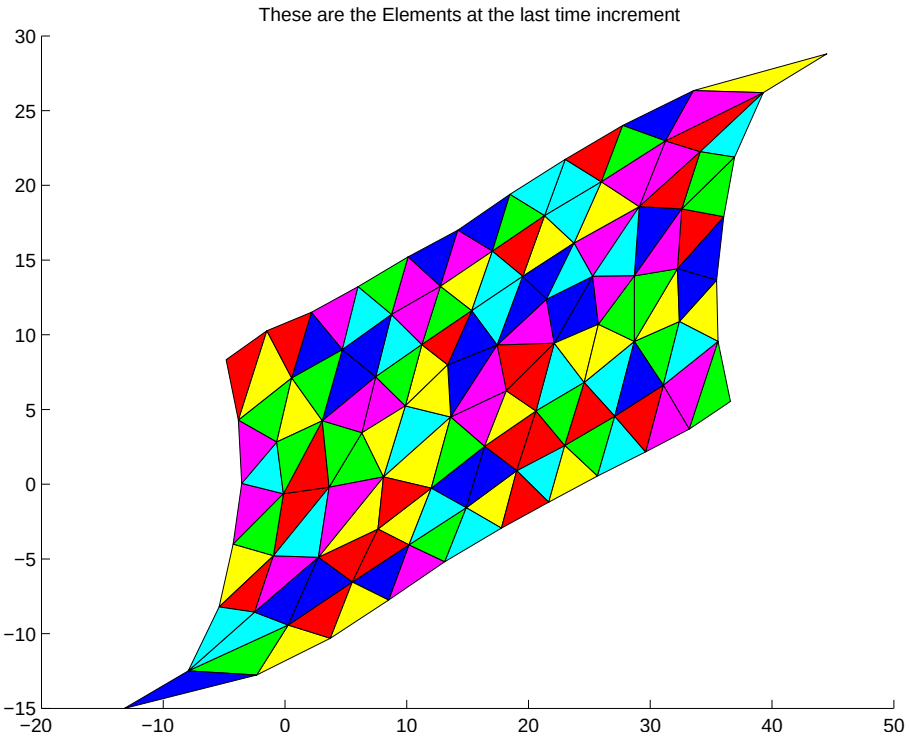
- Read the `j_InputEx4.dat`.Read the displacements at different time steps
- Plot the force at node 3 versus the displacement of node 1



- Plot the nodes at the last time step

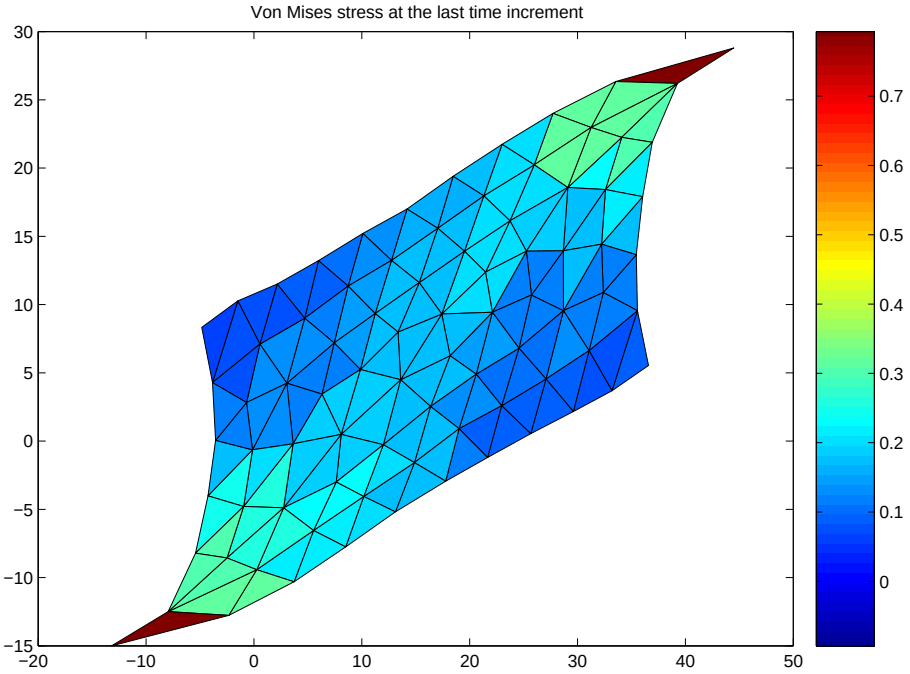


- Plot the elements at the last time step



- Make a movie to capture the evolution of the mesh with time
- Read the file `ex4.rpt`

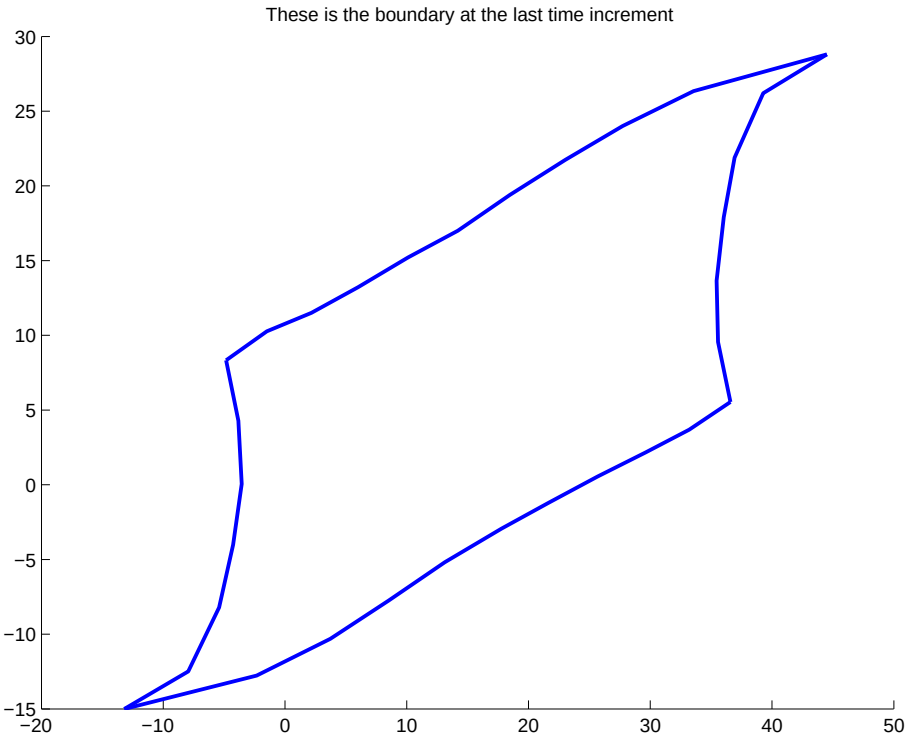
- Create a Surface plot of the vonMises stress for the last time increment. Plot the vonMises stress both on the deformed and undeformed configuration



Exercise 23

This refers to the data and exercise 22.

- Plot with a solid line the boundary of the body at the last time step.

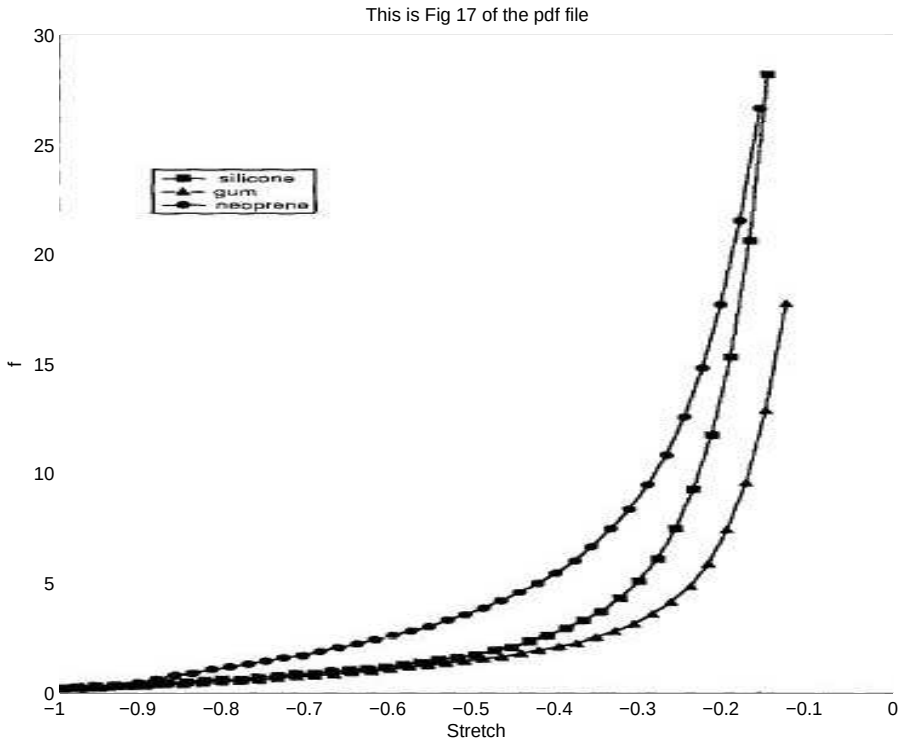


- Create a movie for the evolution of the deformation of the boundary (something as the movie `Boundary.avi` on TeleTop).

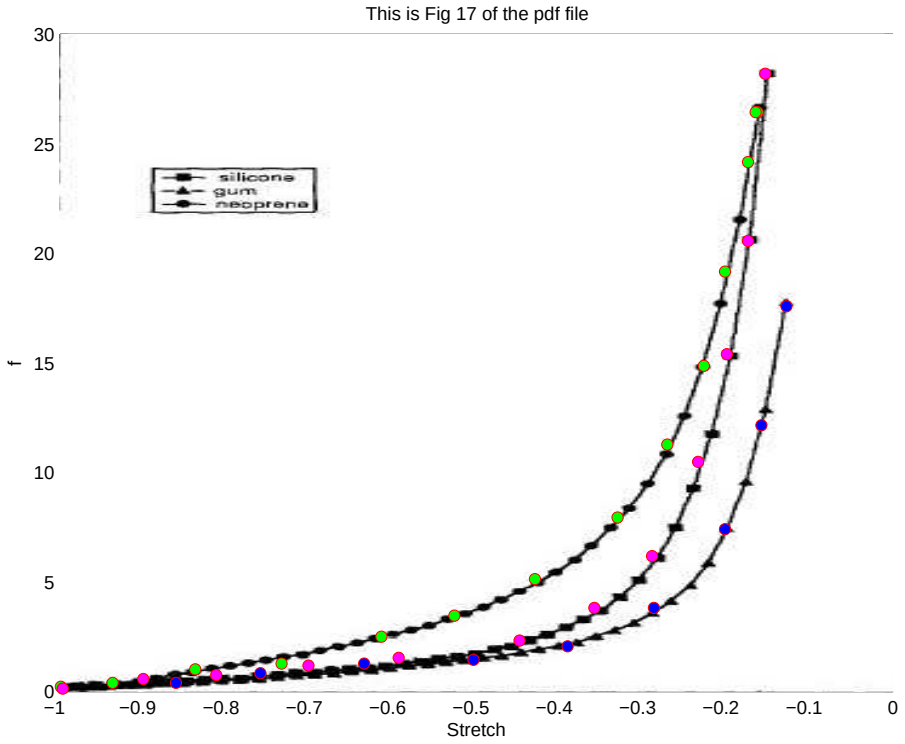
Exercise 24

You want to report in a plot data you found in a `*.pdf` document (this is typical when you want to compare data available on the literature with new data you produce). You want to do that using MATLAB . Extract the data from Fig. 17 found in the document `Arruda_Boyce.pdf`.

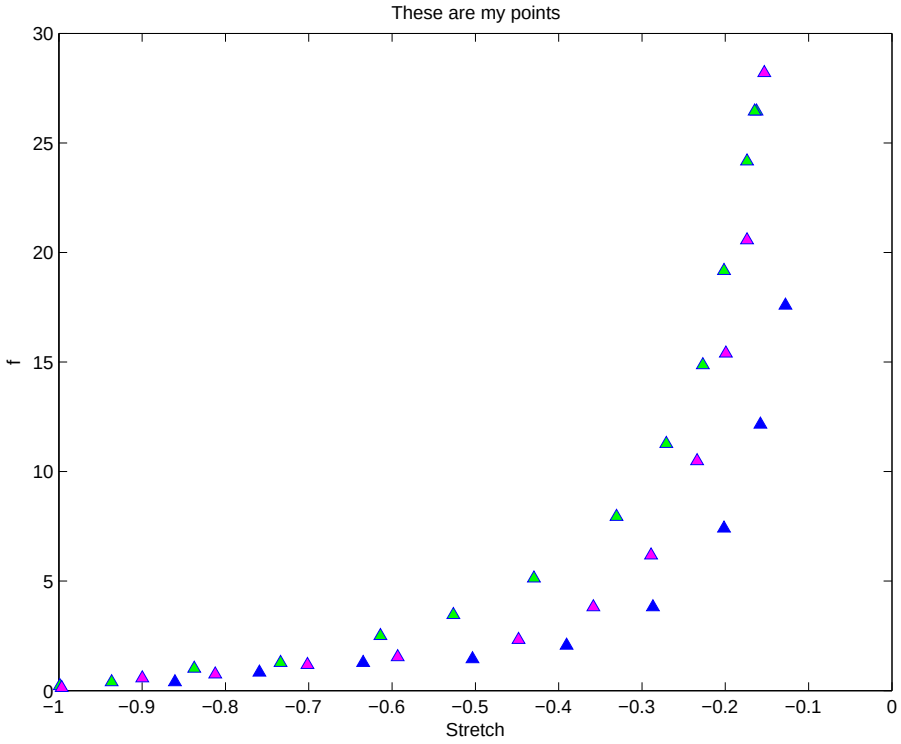
- Open the figure into a MATLAB figure. (The limits for the axis have to be preserved as they are in the pdf file)



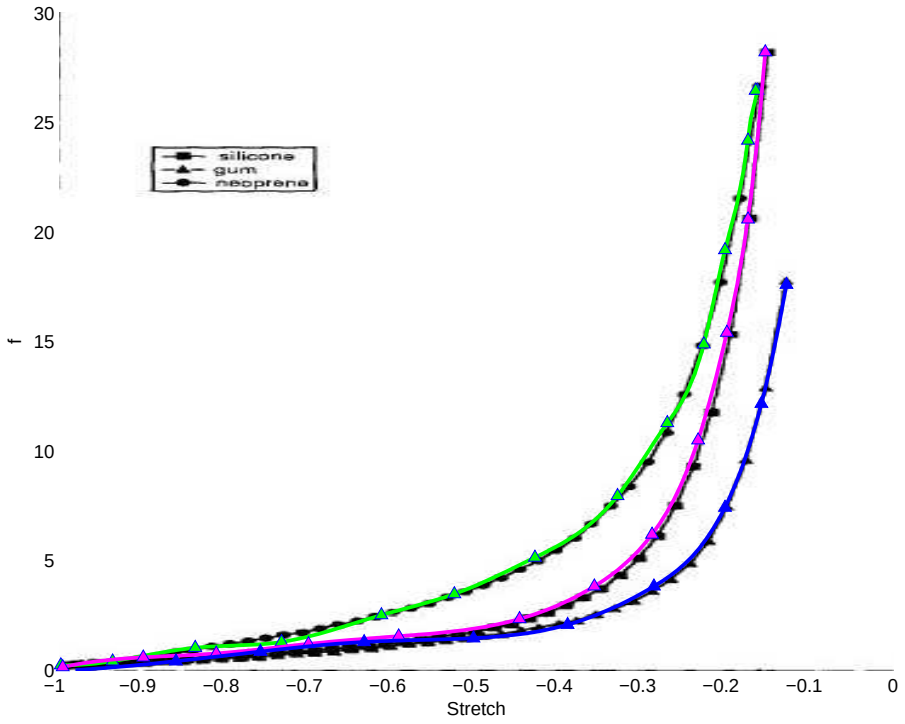
- Pick up points of each curve (separately) with the mouse and plot the points on the figure with Markers



- Write the coordinates of the points you picked up in an Excel file in 3 separated sheets (one for each curve). In each sheet you will have 2 columns with the x and y coordinates off all the points you picked up.
- Open the Excel file you created, read the data and plot all the points you picked for the 3 curves.



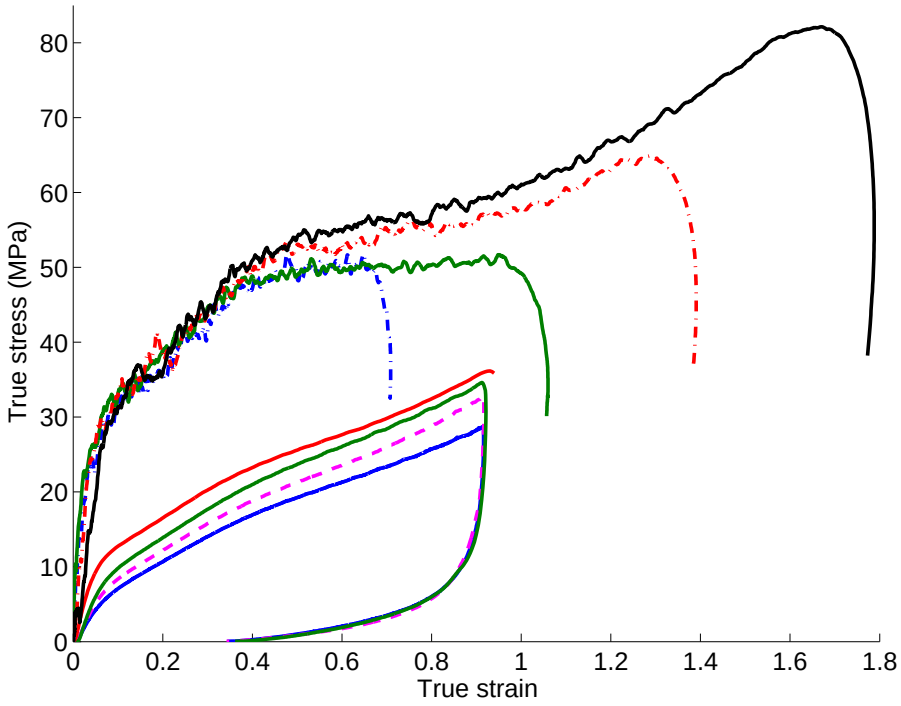
- Interpolate the points with a curve.



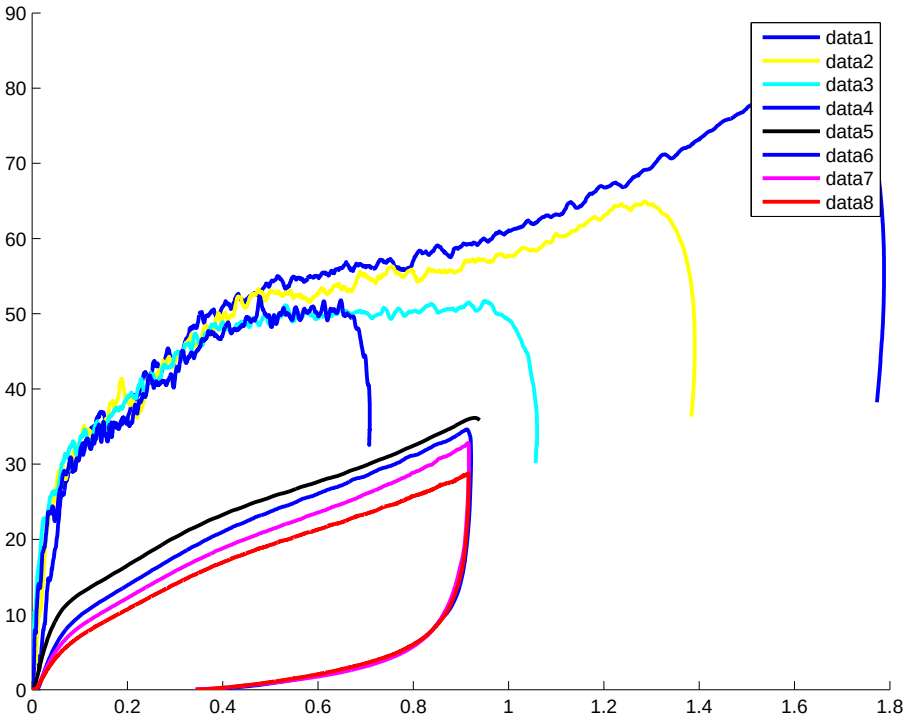
Exercise 25

You plotted some of your data using MATLAB and saved them in the figure `data.fig`. Accidentally you lost the file containing the data and now you want to extract the data directly from the figure.

- Open the figure in MATLAB



- Extract the data of all the curves. You don't want to pick up point with the mouse, but you want to extract the data directly writing a simple script to be more precise and not loose some data points.
- Plot the curves using the data you extracted



- Save the data in an Excel file. Save the data for each curve in a separate sheet.