Programming and Modelling

Module 1 & 2

Introduction to Fortran Programming

C. Thieulot

September 2016

Institute of Earth Sciences
Utrecht University

Contents

1 Introduction

The course in 'programming and modelling' starts with a four weeks on basic computer programming. The other half of the course (i.e. the remaining four weeks are given by other teachers). These four weeks (36-39) comprise two 2-week modules which focus on program development and applications using the Fortran programming language and consist of several instruction sessions and computer lab sessions, for which a total of eights half days have been scheduled. The instruction sessions introduce the programming tools and the computer lab experiments to be done by the students. The experiments of the computer labs are done in the computer workstation rooms of the Buys Ballot buildings, where assistance will be available, during the sessions scheduled for the course.

Students work together in groups of two and each group produces a Lab report that has to be submitted on the first monday after this module ends. The deadline for the first report is then Monday 19th September 3pm. The deadline for the second report is then Monday 3rd October 3pm.

Concerning the course software: the course will be taught using the Linux operating system the GNU Fortran compiler.

Useful references on Fortran programming are,

- M. Metcalf, J. Reid and M. Cohen, *FORTRAN 95/2003 explained*, Oxford University Press, 2008.
- S.J. Chapman, *Fortran 90/95 for Scientists and Engineers*, McGraw-Hill, 1998.
- `http://www.dmoz.org/Computers/Programming/Languages/Fortran`
- `http://gcc.gnu.org/wiki/GFortran`
- `http://gcc.gnu.org/onlinedocs/gfortran`

Links

- gnuplot tutorials:
  `http://www.gnuplot.info/`
- xmgrace tutorials:
  `http://plasma-gate.weizmann.ac.il/Grace/doc/UsersGuide.html`
  `http://mintaka.sdsu.edu/reu/grace.tutorial.html`
  `http://exciting-code.org/xmgrace-quickstart`
- An impressive and most useful website dedicated to programming and numerical modelling
  `http://people.sc.fsu.edu/~jburkardt/`
- A page about Makefiles and much other useful things :
  `http://www.jfranken.de/homepages/johannes/vortraege/make.en.html`

All enquiries about exams and special requests should be addressed to C. Thieulot (c.thieulot-at-uu.nl).

2 Requirements and course credit

2.1 Computer lab report (50% credit)

At the end of each module, a single computerlab report is produced per group of two students. Communication between different student teams is encouraged but (partial) copying work from other teams will lead to exclusion from the course.

Submit your report as a **pdf** file per e-mail before the start of the next course module (see deadlines hereabove). Make sure to include a title, list of authors and production date in your report. Use page numbering. Your report should not exceed the length of 10 pages. I expect 1 report per pair of students.

Structure and contents of the computerlab report:

- The report should be structured in the following way. Each assigment from the course notes should be treated in a separate numbered report section. The different sections contain: a) brief statement of the problem to be solved, b) a short description of the actions taken to solve the problem and c) a short description of your results.
- Include the unaltered source text of your (self created) fortran procedures at the end of the corresponding sections (in a reduced size typewriter font as used in the lecture notes). Use indentation and comment text in your fortran source texts to increase the readability. Source texts copied from the lecture notes or the computer network should be referenced but not be reproduced in your report.
- Integrate screen (text) output and graphics output in the main body of your report in the relevant section. Do not add these items as separate appendices. Define the different screen output texts as numbered tables and the graphics items as numbered figures and refer to these table and figure numbers in the text.
- Your report must contain a short conclusion section where you summarize the main points of what you have learned in this course module

2.2 Midterm written examination (50% credit)

At the end of the second course module a written exam, of one hour, is scheduled about the combined contents of the first two (required) course modules. In this individual exam a simple programming assignment has to be made from scratch i.e. without the use of computer equipment or reference literature.

3 Template for all exercises

```
!=============================================================!
!
! student number(s):
!
! date:
!
! exercise number:
!
! description of the program:
!
!
!
! comments to corrector:
!
!
!=============================================================!

program ....




end program

!=============================================================!
```

Example of proper indentation:

```
do i=1,10
   do j=1,100
      do k=-2:2
         ...
         array(i,j,k)=i+j-k
         ...
      end do
   end do
end do
```

4 Fortran programming exercises (Module 1)

4.1 Introduction

A tutorial overview of the main elements of the Fortran programming language is given in Appendix A. To get acquainted with the basics of applying Fortran, study the examples in the Appendix and work through the exercises in the following sections. Include a short description of each exercise, together with the program source text and input plus output in your lab report for this course module.

To get started with Fortran program development perform the exercises stated below. Use input/output redirection such that you can save your input and output files for your lab report (see Appendix A.2).

It is expected from every group that they carry out all exercises and include the code and results in the final lab report (see section 2.2).

4.2 Tutorial exercises for the 1st week

4.2.1 Hello World

To test the different steps in the procedure of program development and excution write a rudimentary fortran program `hello_world` (see Appendix A.2) that prints the text `hello world` in the terminal window.

4.2.2 A 2×2 equation solver

Write a program which prompts for the values a, b, c, d, f, g and returns the solution x, y of the following linear system:

$$\begin{cases} ax + by = f \\ cx + dy = g \end{cases}$$

[1] Test the proper working of your program in the following way: 1) for a given test vector pair $x = x_0, y = y_0$ compute the corresponding vector pair $f = f_0, g = g_0$ by matrix-vector multiplication and 2) solve the system of equations with the pair f_0, g_0 as the righthand side vector and 3) compare the computed solution vector with the pair x_0, y_0.

4.2.3 Tables of multiplication

In all Fortran program units, (program, subroutine, function) you must put an `implicit none` declaration after the first line of source code. This will enforce explicit typing of all variables used in the program unit and removes an important source of programming errors.

You will use Fortran arrays and do-loop constructs to fill arrays with tables of multiplication (see A.6.1). Necessary input will be read from `stdin` and output will be written to `stdout` (see A.2.1).

1. Write a Fortran program that can produce a single multiplication table. Use a one-dimensional array `table1`, which must be declared as a fixed length (10) array in the declaration block of your program.

   Read the table number from `stdin`, denoted by 'logical unit number' 5, using 'list directed' input (free format input), denoted by the asterisk character '*'.

   ```
   read(5,*) numtable
   ```

   Fill the table in a do-loop,

   ```
   do i=1, 10
      ...
   end do
   ```

   The table array can be written using a 'list directed output' statement,

[1] Use the Cramer formula for the inverse $\mathbf{A}^{-1}$ of a matrix $\mathbf{A}$ with,

$$\mathbf{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad \mathbf{A}^{-1} = \frac{1}{\det(\mathbf{A})} \begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$

where $\det(\mathbf{A}) = ad - bc$ is the determinant of matrix $\mathbf{A}$.

```
write(6,*) (table1(i),i=1,10)
```

Use output redirection to write the table data to a text file `table1.dat`.

2. Compute multiplication tables for $n = 1, ..., m$ in a two-dimensional array `table2`.

   Use an array declaration with fixed array lengths in both dimensions and make sure to declare sufficient array space. Fill the two-dimensional array `table2` in a program control structure *(construct)* with two *nested* do-loops, see Appendix A.6 for an example.

```
do i=1,m
  do j=1, 10
     ...
  end do
end do
```

   Check what happens if the array has been declared with insufficient arraysize. Print the 2-D table when it has been filled in a loop over table rows,

```
do i=1,m
 write(6,*) (table2(i,j),j=1,10)
end do
```

Produce an output text file containing the 2-D table data.

4.2.4 computepi

One can show that the number π can be computed as follows:

$$\pi = 4 \sum_{n=0}^{\infty} (-1)^n \frac{1}{2n+1}$$

In practice ∞ is a mathematical abstraction which computers cannot represent nor process. This is why we shall use

$$\pi = 4 \sum_{n=0}^{m} (-1)^n \frac{1}{2n+1} \qquad \text{with} \qquad m >> 1$$

Write a Fortran program that evaluates the series expression and prints the computed (approximate) value of π. Compare the outcome of your calculation with the analytic value $\pi = 4 \cdot \arctan(1)$ that can be computed within machine precision using the Fortran intrinsic function `atan`. [2]

To investigate the effect of different number representations (single or double precision real variables - see Apppendix A.4) do the following tests and comment on the differences in the outcome for different parameters and declarations of the real variables:

- declare the variable `pi_r` as a (single precision) real
- compute and print `pi_r` by means of the above formula with m=10^3, 10^6, 10^9.
- declare now the variable `pi_dp` as a double precision real
- compute and print `pi_dp` by means of the above formula with m=10^3, 10^6, 10^9.

[2] `http://gcc.gnu.org/onlinedocs/gfortran/ATAN.html`

4.2.5 Golden Ratio

Write a program which computes the Golden ratio ϕ through the Fibonacci numbers:

$$u_0 = 1 \qquad u_1 = 1 \qquad \ldots \qquad u_{n+1} = u_n + u_{n-1}$$

where the ratio u_{n+1}/u_n converges towards ϕ.

Compare your outcome with the formula $\phi = \frac{1+\sqrt{5}}{2}$ and determine how many Fibonacci numbers are necessary to get a correspondence in 5 decimals. [3]

4.2.6 Computing and writing function tables to file

Compute a sine and cosine table for a single period and write the tables of $x_i, \sin(x_i)$ and $x_i, \cos(x_i), i = 1, \ldots, n$ to separate two-column output files. Inspect the data in these files by plotting them in a single frame using the graphics program `gnuplot`. Produce a single graph showing both curves. The graph should include axis labelling and a legend to distinguish both curves. Export your completed plot to a graphics output file in a suitable format. Include this plot in your lab report.

[3] `http://en.wikipedia.org/wiki/Golden_ratio`

4.3 Tutorial exercises for the 2nd week

4.3.1 Simple ballistics

Consider a point mass launched *vertically* in a gravity field with given uniform gravitational acceleration g and initial velocity v_0 (Fig.1a). Write a Fortran program that will read the values of g and v_0 from the standard input file. The program computes trajectory data of the point mass in separate tables, stored as 1-D Fortran arrays, containing the vertical coordinate and its corresponding vertical velocity. The table entries correspond to discrete time values, $t_i = (i-1)\cdot\Delta t,\ i = 1,\ldots,n$, for a fixed time step Δt. The tables are filled in a Fortran do-loop over n relevant time values between launch and impact. The time window covered in the computation should start with the launch and end with the final impact when the point mass returns at the point of departure.

Use sufficient table points to be able to produce smooth plot curves of the vertical coordinate and the velocity as functions of time in separate frames and label the axes of both graphs properly (use gnuplot or xmgrace (appendix C)).

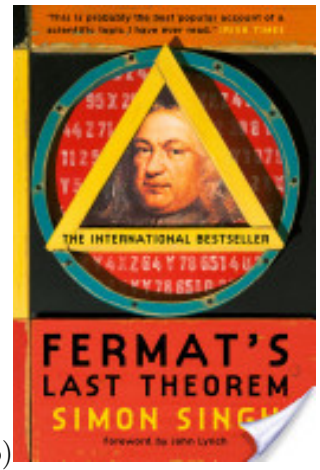
a)  b) 

Fig.1: a) canon from 1558; b) an extraordinary book I highly recommend

4.3.2 Fermat's conjecture

In number theory, Fermat's Last Theorem (sometimes called Fermat's conjecture, 1637) states that no three positive integers a, b, and c can satisfy the equation $a^n + b^n = c^n$ for any integer value of n greater than two.

The first successful proof was released in 1994 by Andrew Wiles, and formally published in 1995, after 358 years of effort by mathematicians (Fig.1b). Verify the conjecture for $1 \le a, b, c \le 100$ for $1 \le n \le 10$.

You may wish to start by using brute force. In a second time, you may wish to have a look and refine your algorithm.

4.3.3 Prime numbers (1)

Write a program which prompts the user for a number less than 999, and tests whether this number is a prime number. Hint: in Fortran the expression `mod(n,m)` gives the remainder when `n` is divided by `m`; it is meant to be applied to integers. Examples are `mod(8,3)=2`, `mod(27,4)=3`, `mod(11,2)=1`, `mod(20,5)=0`.

4.3.4 Prime numbers (2)

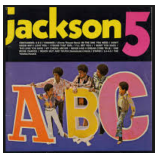
A Mersenne (1588-1648) number is a number of the form $M_n = 2^n - 1$. Are all M_n prime numbers ? Give the first four Mersenne numbers which are prime numbers.

4.3.5 Extrema

Declare an array `tab` of length 50 and fill it so that $tab(i) = (i - 25)^3 + i^2 + i$. By means of a do loop find its minimum, maximum and average value.

4.3.6 Easy as a,b,c

Write a program which prompts the user for three numbers a, b, c and which returns the roots of the polynomial equation $ax^2 + bx + c = 0$.



4.3.7 P-norms

The Euclidean norm of a vector $\mathbf{a} = (a_1, a_2, a_3)$ in three dimensions is given by

$$|\mathbf{a}|_2 = \sqrt{a_1^2 + a_2^2 + a_3^2} \qquad \text{or} \qquad |\mathbf{a}|_2 = (a_1^2 + a_2^2 + a_3^2)^{1/2}$$

One can generalise this by defining the p-norm of an $n-$dimensional object $\mathbf{u}$ as follows:

$$|\mathbf{u}|_p = (|u_1|^p + |u_2|^p + |u_3|^p + |u_4|^p + ...|u_n|^p)^{1/p} = \left(\sum_{i=1}^{n} |u_i|^p \right)^{1/p}$$

- create three arrays of size 567
- fill these arrays with three different methods of your choice (be creative :)
- write a program which computes the 1-norm, 2-norm, 3-norm and 100-norm of these arrays

4.3.8 Magic square

Create a two-dimensional array and fill it so that it contains the following numbers:

14	9	23	16
19	20	10	13
8	15	17	22
21	18	12	11

Verify that it is a magic square.

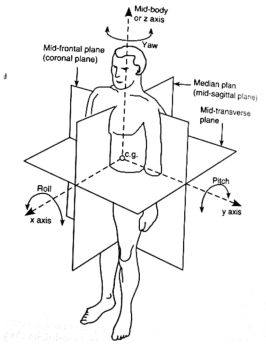
5 Fortran programming exercises (Module 2)

5.1 myabsvalue function

Write a function which returns -1 if its argument is negative and +1 if its argument is positive. Think. Write a small program which tests this function.

5.2 centergrav function

Write a function which accepts as arguments the coordinates of 5 points and returns the coordinates of the center of gravity of these points. Write a small program which tests this function.

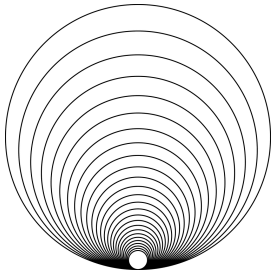


5.3 swap subroutine

Write a subroutine which takes two real numbers as arguments and returns them swapped. Write a small program which tests this subroutine.

5.4 circle subroutine

Write a subroutine `area_and_circumference` which takes as argument the diameter D of a circle and respectively returns its area and circumference. Write a small program which tests this subroutine.



5.5 Using allocatable arrays

To practice the use of dynamic (allocatable) arrays do the following two excercises. Information including examples of allocatable arrays can be found in Appendix A.

1. Write a Fortran program which stores the integer numbers $1, 2, \ldots, n$ in a one-dimensional integer array of length n. Declare the array using the `allocatable` attribute. Read the number n from standard input and after allocating the array use the array to do the following experiment. In a do-loop over n cycles with index $i = 1, 2, \ldots, n$, store the loop index in the array position indicated by the loop index. In a second do-loop perform a summation of the array elements. Finally print the computed sum value and check the result by comparing it

with the outcome of the intrinsic function `sum` [4] and with the analytic result, $n(n+1)/2$ to validate your program code.

Repeat this experiment for increasing values of n until you get a system error message concerning an array that can not be allocated. This way you can find out, by trial and error, about the system dependent array size limit n_{max}.

For large integer values $n > n_{max}$ you will find that the analytical value for the array sum difffers from the value obtained by summing the array elements. This can be explained by the fact that the (default) internal machine representation of integers in the binary system is based on four *bytes* of eight binary digits (*bits*) each. With one bit used to indicate the sign ($\pm$), this leaves 31 bits or $2^{31} - 1$ for the maximum integer that can be represented. [5] Verify this integer overflow interpretation by redefining the summation variables as real type variables.

2. To investigate the system dependent maximum array size more systematically do the following experiment. Write a program that contains a single endless do-loop. Increment the variable `msize` from an initial zero value by one million in each loop cycle. Allocate an integer array `m_array` of size `msize`. After allocating the array check the allocated array size with the Fortran intrinsic `size` function, as in the example program of appendix **??**, and print the determined array size.

   Next fill the array in an inner loop, as in the previous exercise, with the array index. Deallocate the array at the end of each cycle of the endless loop.

   Eventually the program will crash with a system error message stating that the array could not be allocated. Include the last few lines of program output and the system error message, together with your program source text in your report.

   If the filling of the array slows the program too much to your taste, modify the program such that only a fixed number of evenly spaced array elements is filled, say `1, 1 + msize/10, 1 + 2*msize/10, ...` .

5.6 Two-dimensional tables of functions

In the first module of the programming course writing and reading tables of functions $f(x)$ of a single variable played an important role. Here we extend this practice to functions of two variables $f(x, y)$ resulting in 2-D tables that are written to file in a suitable format for plotting with the graphics program `gnuplot` (see Appendix D).

Write a Fortran program for the computation and file output of a 2-D table of grid point values of a given function of two variables $f(x, y)$. The 2-D rectangular grid is defined by the following specifications:
The grid point coordinates (x_i, y_j) are defined as, $x_i = x_1 + (i-1)\Delta x,\ i = 1, \ldots, n_x$ and $y_j = y_1 + (j-1)\Delta y,\ j = 1, \ldots, n_y$. Where $\Delta x = (x_{n_x} - x_1)/(n_x - 1)$ and $\Delta y = (y_{n_y} - y_1)/(n_y - 1)$. This defines a so called *equidistant* grid of nodal points.

The program should be designed as follows:

1. Read the grid specifications in the main program from the *standard input device (stdin)* as `xmin,xmax,nx` and `ymin,ymax,ny` respectively. The values of the grid spacing Δx and Δy are computed by the program from the input according to the spec's.

[4] The Fortran intrinsic function `sum` applied to a array returns the sum of the array elements.
[5] The integer byte length can be increased from four to eight by the gfortran compiler flag `-fdefault-integer-8`

2. To distinguish between different implemented functions in the program we apply an integer switch variable `ifunctype`. The value of `ifunctype` is read from *stdin*.

3. Grid values of the function $f(x, y)$ are computed in a properly dimensioned `(nx,ny)` 2-D allocatable real array `datarray`.

4. The function values $f(x, y)$ are computed in two nested do-loops over columns and rows of the rectangular grid. For each grid point the corresponding function value is computed in a call of a Fortran function `func` with the following header,

```
real function func(ifunctype,x,y)
implicit none
integer ifunctype
real x,y
```

   Depending on the value of `ifunctype` different implemented functions are evaluated. This function should contain an `error trap` where an error messages is printed and a `stop` statement is executed when a `not-implemented` value of `ifunctype` is input of the function.

5. After filling the function table array with grid point values write the table to a file in a format suitable for plotting with `gnuplot` (see Appendix D). Writing the table array `datarray` should be done in a subroutine call with the following header,

```
subroutine writdat2d(xmin,xmax,nx,ymin,ymax,ny,datarray,file_out)
implicit none
integer nx,ny
real xmin,xmax,ymin,ymax
real datarray(ny,nx)
character(LEN=*) file_out
```

Apply your program to your favorite function of two variables [6] and produce a contour plot and a color plot with `gnuplot`.

[6] An example suitable for testing different grid resolutions is $f(x, y) = \sin(k_n x)\cos(k_m y)$, where the wavenumbers are defined as $k_n = n/(2\pi)$, $k_m = m/(2\pi)$, corresponding to wavelengths $\lambda_n = 2\pi/k_n = 1/n$, $\lambda_m = 2\pi/k_m = 1/m$.

Appendix

A A Fortran Tutorial

A.1 Introduction

Fortran is the predominant programming language in the field of scientific and numerical computing. In order to use a Fortran program, the program source text, possibly extended with external procedures must be translated in to machine language (compiled) using a compiler program. On Linux/unix systems an open software (GNU) Fortran 77 compiler `g77` is available. Recently the GNU Fortran 95/2003 compiler (`gfortran`)has also become available. This compiler has been installed on the institute network and will be used in this course.

A.2 Installing and executing a Fortran program

To create a simple Fortran program, the Fortran source text of the program must be written in a text file, say `prog.f90`, using a text editor, similar as with the preparation of a Scilab script. The file name extension `.f90` is a (compiler) requirement for the use of the Fortran 90 extended source text format (see below). When the complete program text is available the program can be installed using the Fortran compiler, `gfortran`, by typing on the command line,

```
gfortran prog.f90
```

If no syntax errors are found by the compiler, this command will produce an *executable file* with the default name `a.out`, or when using the `-o` option as in,

`gfortran -o` *exec-file* `prog.f90`

the executable will be written to the file *exec-file*. The program can then be executed by typing on the command line the name of the executable file, either `./a.out` or *./exec-file*. More information on the many possible compiler options is available through the online Linux manual page which can be accessed by typing `man gfortran` from the command line.

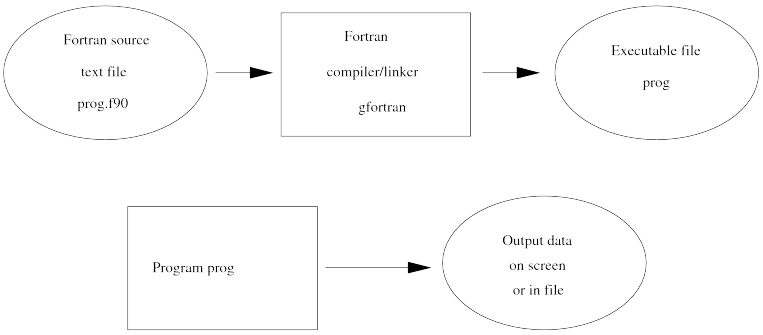


Figure 1: *From Fortran source text file to resulting program output (text/graphics). An executable program file* `prog` *is made from (a) Fortran source text file(s) with a compiler/linker program* `gfortran`.

As a first example of installing and running a rudimentary Fortran program consider the following three line 'main program',

```
program hello_world
print *, 'hello world'
end program
```

This program is compiled and installed in a default executable file `a.out` in the working directory by typing the command,

```
gfortran  hello_world.f90
```

Typing `./a.out` from the command line will then produce the output line

```
hello world
```

A.2.1 Input/output redirection

Programs often read input from ascii (text) input files and write output results to output text files. In case your program uses a single input and/or output file, a convenient way to implement this is through the use of the standard input (`stdin`) and output (`stdout`) device. When reading from `stdin` your program will expect input from the keyboard, when writing to `stdout` your program writes output to the screen. This is convenient while developing a small program, but in case of extended program input and output, it is more convenient to read input from a text file which has been prepared in advance and print results to another text file which can be inspected using a text editor, instead of printing on the screen. A program which has been prepaired to use `stdin` and `stdout` can be used without modification to read from a named input file, `prog.in` and write output to the named file `prog.out` using so called input/ouput redirection by typing
`prog < prog.in > prog.out`

A.3 Fortran source text formats

Fortran text is not case sensitive, upper and lower case text are not distinguished by the Fortran compiler. Fortran source texts come in two alternative formats, known as fixed and free source format, distinguished by the two filename extensions `.f` and `.f90` respectively. The fixed format is required by the older Fortran 77 standard. Here source text lines are truncated after position 72 and

the first 6 positions are reserved. Position 1 through 5 can be used for 5 digit numerical statement label fields and postion 6 for a continuation character (see below).

Comment lines in fixed format files are specified by a `c` or `*` in position 1. In free format source, text following the `!` character up to the end of the text line is interpreted as comment text.

Fortran source lines can be continued over several text lines. In the fixed source format this is done by having a blank (space) in position 6 of the initial statement line and a non-blank (and non-zero) character in position 6 of the continuation lines. In free source format a source line ended by the & character is continued on the next text line.

A.4 Program variables

All variables used in Fortran programming units must be declared with an explicit type declaration. This is enforced by including the statement `implicit none` as the second line of code of every program unit. The main intrinsic data types of the Fortran language are `integer`, `real`, `logical` and `character`. Integer and real variabeles can have the value of integer and (rounded) real numbers respectively. [7]

Logical variables take the values true or false, denoted as `.true.` and `.false.` in Fortran assignment statements. Character variables are used to hold text strings of characters. Besides intrinsic data types Fortran 90 and later versions also support so called derived types which can be defined by the user.

Program units are split in two blocks

1. A declaration block containing all the necessary declaration statements.

2. An execution block containing all the executable statements, i.e. all statements that result in program actions.

The following example shows the declaration of several real variables and a number of assignment statements where values are assigned to program variables. Arithmetic expressions are used in the assignments containing, the arithmetic operators `+,-,*,/` for summation, subtraction, multiplication and division and the `**` for exponentiation. Also used is the Fortran intrinsic function for the square root `sqrt`. Finally values of variables are written on the screen in a `print` statement.

```
program quad_polynom_roots
implicit none

real :: a,b,c,x1,x2

a = 1
b = -6
c = 6
x1 = ( -b - sqrt( b**2 -4*a*c ) ) / (2*a)
x2 = ( -b + sqrt( b**2 -4*a*c ) ) / (2*a)

print *, 'x1=',x1,' x2=',x2

end
```

[7] Real numbers are represented with finite precision using binary number representation. The precision depends on the number of *binary digits* (bits) used in the representation. The default is 32 bit (4 bytes) representation. When higher precision is required to prevent loss of accuracy real type variables should be declared with double (8 byte) precision using `real*8` as type declarator.

A.4.1 Variables of type `character`

Character type variables are used in Fortran programs to manipulate text strings. This is particularly applied in the definition of filenames (see A.8.1).

Character variables must be declared with a length parameter as in the following example.

```
program exm_char
implicit none
character(len=5) text_string
text_string(1:2) = 'aa'
text_string(3:4) = 'bb'
text_string(5:5) = 'c'
print *, text_string
end program exm_char
```

This produces the output `aabbc`. In this example a substring mechanism is used to define subsets of the character string. **??** contains an example of character variables in producing simple 'print-plots' of numerical data.

A.5 Indexed variables

Fortran contains a rich syntax for dealing with indexed variables or arrays. We only mention some simple applications of 1-D and 2-D arrays here, illustrated in the following program example.

Array variables must be declared as such and this can be done in several alternative ways. We use here only one type of array declaration that can be used in case the array index starts at the default value one. The array attribute of the variables is implied here by specification of the bracketed upper value of the array indices in the respective dimensions.

```
program mat_vec
implicit none

real :: vec(2)
real :: res(2)
real :: mat(2,2)

vec(1) = 1.0
vec(2) = 0.5
print *, 'vec', vec

mat(1,1) =  1.1
mat(1,2) =  1.2
mat(2,1) =  2.1
mat(2,2) =  2.2
print *, 'mat', mat

res(1) = mat(1,1)*vec(1) + mat(1,2)*vec(2)
res(2) = mat(2,1)*vec(1) + mat(2,2)*vec(2)
print *, 'res', res

end
```

This program produces the following screen output,

```
vec 1.000000 0.5000000
mat 1.100000 2.100000 1.200000 2.200000
res 1.700000 3.200000
```

A.6 Program Control Structures (*constructs*)

A.6.1 Program loop statements

To work with indexed variables in so called program loops, the Fortran do-end do construct is used. The syntax of this construct can be specified by,

```
do i = i_begin, i_end, i_step
   ... executable statements ...
end do
```

The statements between do and end do will be executed repeatedly for subsequent values of the index variable, `i = i_begin, i_begin + i_step, i_begin + 2*i_step, ..., i_end`. Note that the index increment may be negative. The loop statement block is not executed if the index exceeds `i_end` in the first loop cycle.

```
program mat_vec_do_loop
implicit none

real vec(2)
real res(2)
real mat(2,2)

integer i,j,n

n=2

vec = 0.0   ! initialise array using vector syntax
do i=1, n            ! fill
   vec(i) = 1.0/i    ! the vector
end do               ! array
print *, 'vec=',vec  ! write the vector array on the screen

mat = 0.0            ! initialise the (2X2) matrix
do i=1,n                          ! fill the matrix
  do j=1,n                        ! in nested loops over
    mat(i,j) = 1.0/(1+ abs(i-j))  ! columns (inner loop j)
  end do                          ! and rows (outer loop i)
end do
print *, 'mat=', mat              ! write the matrix

res = 0.0
do i=1,n                              ! matrix-vector multiplication
  do j=1,n                            ! in nested loops over
    res(i) = res(i) + mat(i,j)*vec(j) ! columns (inner loop j)
  end do                              ! and rows (outer loop i)
end do                                !
print *, 'res=', res                  ! write the result vector

end program mat_vec_do_loop
```

This program produces the following screen output,

```
vec= 1.000000 0.5000000
mat= 1.000000 0.5000000 0.5000000 1.000000
res= 1.250000 1.000000
```

A special case of a Fortran do-loop is used when the number of loop cycles is not known in advance. This is implemented as an infinite loop which exits when an explicit stop criterium is met.

```
do
   ...  executable statements ...
   if (  logical expression ) exit
   ...  executable statements ...
end do
```

The loop iteration is stopped when *logical expression* results in the value `.true.` and program control continues at the first exectutable statement following `end do`. This construct is used in controlling the convergence of iterative computations such as evaluation of series expansions, or iterative computation of the roots (zero points) of a non linear function.

A.6.2 Conditional Statements and logical expressions

To control the program flow in computer programs conditional statements are used. The `if ( ... ) exit` in the above infinite do-loop is an example of this. Conditional statements switch the program control flow depending on the value of a logical expression as in the example below. The main construct involving a conditional statement is the `if then else end if` construct.

```
program exm_ifthenelse
implicit none
integer i, nmax
logical even
character(len=5) label
nmax = 3

do i=-nmax,nmax

   even = mod(i,2) == 0    ! intrinsic modulo function
   if ( even ) then
     label = 'even'
   else
     label = 'odd'
   end if

   if (i <=-2) then
     print *, i, ' is ',label, ' and less or equal minus two'
   else if (i > -2 .and. i < 0) then
     print *, i, ' is ',label, ' and negative'
   else if (i == 0) then
     print *, i, ' is ',label, ' and zero'
   else
     print *, i, ' is ',label, ' and positive'
   end if

end do

end program
```

Note the use of the `==`, `<=`, `>` and `<` relational operators in the `if then` statements. This program illustrates also the use of logical expressions and logical variables. The program produces the following screen output,

```
-3  is odd   and less or equal minus two
-2  is even  and less or equal minus two
-1  is odd   and negative
 0  is even  and zero
 1  is odd   and positive
 2  is even  and positive
 3  is odd   and positive
```

A.7 Program Input/Output

In general it is advisable, in setting up a modelling experiment, to enter the model parameters in a text file from which they can be read by a modelling program. This way a flexible program set-up can be used which will be applicable, not just for a single modelrun, but for a range of models. Reading input data from files and echoing input data to a program log-file is essential for documenting modelling experiments. Writing modelling results to output data files is necessary for postprocessing and reporting.

To read data from input files and write results to output files several input/output (I/O) functions are available. We show here the `open` and `close` statemens for connecting respectively disconnecting data files to your Fortran program and the `read`, `write` and `print` statements.

```
      program exm_io
      implicit none
      integer unitin, unitlog, unitout
      integer i,ndata_out
      character(len=80) fname_data_out

      unitin = 11
      open(unitin,FILE='exm_io.in')     ! open the input file
      read(unitin,*) ndata_out          ! read # output data
      read(unitin,*) fname_data_out     ! read filename output data

      unitlog = 12
      open(unitlog,FILE='exm_io.log')   ! open the log file
      write(unitlog,*) 'ndata_out     =', ndata_out
      write(unitlog,*) 'fname_data_out=', fname_data_out

      unitout = 13
      open(unitout,FILE=fname_data_out) ! open the output file

      do i=1,ndata_out                  ! write output data
        write(unitout,*) i, i**2
      end do

      close(unitin)                     ! close all open files
      close(unitlog)
      close(unitout)

      end program
```

In this example different files are connected to the program simultaneously that can be accessed through a unique *logical unitnumber*. Note how file names are specified either as literal character constants or as character variables in the open statements.

A.8 Procedures, subroutines and functions

Two types of program procedures exist in Fortran, subroutines and functions. Subroutines must be executed through an explicit 'procedure call' statement, where a list of parameters is passed to the procedure as in the example below.

```
      program exm_subroutine
      implicit none
      real xtable(10),ytable(10)
      integer i,n
      real a,b,dx,xmin,xmax

      n=3

      xmin = 0.0;               ! set range of x table
      xmax = 2.0;               !
      dx   = (xmax-xmin)/(n-1); !

      a = 2.0;                  ! set parameters passed
      b = 1.0;                  ! to the subroutine

      do i=1,n
        xtable(i) = xmin + (i-1)*dx
        call subr(a,b,xtable(i),ytable(i))
        print *, 'i=',i,' x=',xtable(i),' y=',ytable(i)
      end do

      end program
!-----------------------------------------------------------------
      subroutine subr(a,b,x,y)
      implicit none
      real a,b,x,y

      y = a * x + b
```

```
      end subroutine
```

Note how individual elements of the table x, y arrays are passed as so-called actual parameters from the main program to the subroutine. The corresponding parameter in the subroutime source text is referred to as a formal parameter. The names of the actual and formal parameters may be different as in this example.

A number of often used program procedures is available in Fortran as intrinsic functions. Examples are procedures for the evalution of math functions like the square root, sine and cosine (`sqrt, sin, cos`) etc. In the following example a similar procedure as in the foregoing example is implemented as an external (non-intrinsic) function. The function is declared as a variable of the same name as the function and the external procedure nature is specified as a so called attribute. A list of attributes is terminated by a double semicolon.

```
      program exm_function
      implicit none
      real xtable(10),ytable(10)
      integer i,n
      real a,b,dx,xmin,xmax
      real, external :: func    ! declare func as external function

      n=3                       ! def. table length <= 10
      xmin = 0.0;               ! set range of x table
      xmax = 2.0;               !
      dx   = (xmax-xmin)/(n-1); !
      a = 2.0;                  ! set function parameters
      b = 1.0;                  !

      do i=1,n
        xtable(i) = xmin + (i-1)*dx
        ytable(i) = func(a,b,xtable(i))
        print *, 'i=',i,' x=',xtable(i),' y=',ytable(i)
      end do

      end program exm_function
!-----------------------------------------------------------------------
      real function func(a,b,x)
      implicit none
      real x,a,b

      func = a * x + b

      end function func
```

These programs produce the following screen output,

```
 i= 1  x= 0.0000000  y= 1.000000
 i= 2  x= 1.000000  y= 3.000000
 i= 3  x= 2.000000  y= 5.000000
```

Note the difference between the use of the subroutine and the function in returning the computed result from the procedure to the 'calling program unit'. In the subroutine example the result is passed as a procedure parameter. In case of a function the result must be returned by assigning the result value to a function variable with the same variable name as the function procedure. Note also that in every program unit, main program, subroutine or function, all the occuring variables must be declared locally in an explicit type declaration.

A.8.1 A subroutine for writing table data to a 2-column text file

The following subroutine for writing table data to a two-column text file is given in

```
      subroutine writdat(ndata,xdata,fdata,filename)
      implicit none
```

```
integer ndata
real, dimension(ndata) :: xdata, fdata
character(LEN=*) filename
!* locals
integer i

open(UNIT=9,FILE=filename)
do i=1,ndata
   write(9,*) xdata(i),fdata(i)
end do
close (9)

return
end subroutine writdat
```

B Supplementary Fortran programming exercises

B.1 Introduction

B.1.1 Computation of arithmetic, geometric and harmonic mean values

One can define the arithmetic, geomteric and harmonic averages of n values as follows

$$A = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad G = \sqrt[n]{\prod_{i=1}^{n} x_i} \qquad H = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$

- create an array `tab` of size `m`=33
- fill it with random numbers between 0 and 1
- compute A, G, and H with do loops
- compute A, G, and H in one line of code by means of intrinsic functions
- multiply the array `tab` by 10^{23}. Compute A,G,H again. Do the results conform with your expectations ? How could we compute G differently ?

B.1.2 Computation of the p-norm of a vector

The Euclidean norm of a vector $\mathbf{a} = (a_1, a_2, a_3)$ in three dimensions is given by

$$|\mathbf{a}|_2 = \sqrt{a_1^2 + a_2^2 + a_3^2} \qquad \text{or} \qquad |\mathbf{a}|_2 = (a_1^2 + a_2^2 + a_3^2)^{1/2}$$

One can generalise this by defining the p-norm of an $n-$dimensional object $\mathbf{u}$ as follows:

$$|\mathbf{u}|_p = (|u_1|^p + |u_2|^p + |u_3|^p + |u_4|^p + ...|u_n|^p)^{1/p} = \left(\sum_{i=1}^{n} |u_i|^p\right)^{1/p}$$

- write a subroutine `pnorm` which computes the p-norm of an array `uuu` of size `n`
- write a simple program which
  1. creates an array of size `m=123`
  2. fills the array with random numbers between -π and π,
  3. computes and displays the $1, 2, 5, 10, 100$-norm of this array

B.1.3 The bubble-sort procedure for sorting numbers

The bubble-sort algorithm is a simple method for sorting a list of numbers, $a_i, i = 1, \ldots, n$, in ascending order that works according to the following stepwise specification,

1. Compare a_1 and a_2 and swap the numbers if $a_2 < a_1$
2. Compare a_2 and a_3. If $a_3 < a_2$ swap a_3, a_2 and repeat item 1
3. Repeat item 2 and for all subsequent numbers in the list: comparing it succesively with its predecessor in the list and swapping until a smaller number is encountered.

Write a subroutine with the following header that will apply the bubble-sort algorithm to a list of integers

```
subroutine bubblesort(ndata,numbers)
implicit none
integer ndata, numbers(*)
  ...
end subroutine bubblesort
```

B.1.4 A function to determine the length of a 2-column data file

Subroutine `writdat` A.8.1 is used frequently in this course for writing 2-column data tables to a text file. In order to read the data from such 2-column files in your programs it is necessary to determine first the length of the file. This way it can be verified that arrays of sufficient length are used to store the file data in memory.

To this end write a fortran function with the following header to determine the number of data on the file.

```
integer function scandat(filename)
implicit none
character(LEN=*) filename
    ...
end function scandat
```

The number of data is returned through the function name.

Hints:

- Use an *endless-do* construct as in A.6 to read the data lines on the file.
- Apply an extended version of the `read` statement, specifying the `END` clause, reading the character variable `textstring`

```
    do
       ...
       read(UNIT=logical_unit,FMT='(a)',END=100) textstring
       ...
    end do
100 continue
```

  This causes the program to continue at the `100 continue` line when the *end of file* condition occurs.

B.1.5 A function to read data from a 2-column data file

Subroutine `writdat` A.8.1 can be used for writing 2-column data tables to a text file. The number of data on an existing file can be determined by scanning the file with function `scandat` defined above in B.1.4.

Once the number of data on a file is known one can read the 2-column data from the file in two 1-D arrays of sufficient size. This can be done in a similar way as in writing the data to file in subroutine `writdat` listed in A.8.1.

To this end write a fortran subroutine `readdat` with the following header to read the 2-column data from the input data file.

```
subroutime readdat(ndata,xdata,ydata,filename)
implicit none
integer ndata
character(LEN=*) filename
    ...
end subroutime readdat
```

Hints:

- Test your implementation of `readdat` by applying it in a main program where the call to `readdat` is preceded by a call to `scandat`.
- Build an array length check in your program that produces an error message and a program `STOP` when the number of file data exceeds the declared array length.

B.1.6 Printplot of a 2-column data table

In section **??** printplots of 2-column data sets have been introduced. A subroutine for initializing a simple plotframe in a Fortran character array is listed there.

Once the plot frame has been initialized a 2-column data set can be plotted. To this end the `ndata` data points must be mapped in the 2-D rectangular character buffer `screenbuf` of `nrow` lines and `ncol` columns using *substring* operations in a similar way as in the example in **??** in **??**.

Write a subroutine `printplot` with the following header for mapping the 2-column data in the character array `screenbuf`

```
subroutine printplot(xdata,ydata,ndata,     &
                     xmin,xmax,ymin,ymax,   &
                     nrow,ncol,screenbuf)
implicit none
real, dimension(*):: xdata, ydata
real xmin,xmax,ymin,ymax
integer ndata,nrow,ncol
character(LEN=ncol) screenbuf(nrow)
  ...
end subroutine printplot
```

Hints:

- Parameters `xmin,xmax,ymin,ymax` define the represented data window. This allows to zoom in on the data.
- Mapping data points x_i, y_i, $i = 1, \ldots$ `ndata` within the data window $[$`xmin`$,$ `xmax`$] \times [$`ymin`$,$ `ymax`$]$ is done by the following transformation,

```
irow = 1 + nint((nrow-1)*(ydata(i)-ymin)/(ymax-ymin))
jcol = 1 + nint((ncol-1)*(xdata(i)-xmin)/(xmax-xmin))
screenbuf(nrow-irow+1)(jcol:jcol) = symdata
```

- Test your implementation of `printplot` by applying it in a main program where the call to `printplot` is preceded by a call to `fill_frame` (**??**) and followed by a call to a printout of the screenbuffer by a call to the following routine `print_screenbuf`

```
subroutine print_screenbuf(nrow,ncol,screenbuf)
implicit none
integer nrow, ncol
character(LEN=ncol) screenbuf(nrow)

integer i

! print the screen buffer array
do i =1, nrow
   write(6,'(a)') screenbuf(i)(1:len_trim(screenbuf(i)))
end do

return
end subroutine print_screenbuf
```

B.1.7 Removal of the mean value of a time series

Problem (Exam) Imagine you want make a computer program that can perform simple data processing on measurement data that are read from an input file. The measurement data are represented by a time series $s_i = s(t_i),\ i = 1, \ldots, N$ that correspond to signal values from a digital measurement device, say a seismograph.

A single time series is stored as a 2-column text file, containing real number data, where the first column represents the measurement times $t_i, i = 1, \ldots, N$ and the second column contains the measurement data values $s_i, i = 1, \ldots, N$.

Suppose you want to process the measurement time series by removing the meanvalue from the data. To this end you want to transform the input time series data s_i into a output time series $s_i^{'} = s_i - \mu$, where $\mu = \frac{1}{N}\sum_{i=1}^{N} s_i$ is the mean value of the input data.

After procesing the data you want to write the resulting time series to an output file with an identical file structure as used for the input data.

Assignment Write a fortran program for the data processing task described in the introduction. The program must be created according to the following specifications:

- The program applies two 1-D allocatable arrays for the measurement time values and the measurement data values respectively.
- The program reads the number of avalailable data, N, from the *standard input device* (unit 6).
- The program reads the input time series data from an input file named `timeseries_in.dat` and writes the processed time series to an output file name `timeseries_out.dat`
- The output file is written using a fortran subroutine `writdat` with the following routine header,

```
subroutine writdat(ndata,xdata,ydata,filename)
integer ndata
real xdata(*),ydata(*)
character(LEN=*) filename
```

  N.B. you don't have to write this subroutine.

- Input data are processed in a do-loop over the avaible data points and the output data values are overwriting the input data, thereby using a single array for both input- and output data.

C Some useful Linux commands

ls	gives a list of the files and subdirectories that are located in this directory
ls -l	print also access attributes, time-stamp and size
ls q*	list all files and directories that begin with the letter 'q'
ls *q	or end with q
ls q*p	or begin with 'q' and end with 'p'
ls *.3*	or contain '.3'
pwd	gives the pathname (string) of the current directory (your location in the directory tree)
.	means the current directory
..	the parent directory above the current directory
ls ../..	gives a list of files in the directory two levels up in the directory tree
cd *dirname*	change to directory *dirname*
cd ..	go one level up in the directory tree
cd	go to home-directory (this is your home base and usually the top of the directory tree of your file space)
prog < *inputfile*	redirection: instead of typing input for *prog* yourself, *prog* uses input from *inputfile* (very useful for large programs or programs that are executed many times!)
program < *inputfile* > *outputfile*	the same goes for output data (into file instead of screen)
rm *file*	remove *file*
rm p*	remove all files that begin with 'p'
cp *a b*	copy file *a* to file *b*
cp *path/file1* .	copy file1 in directory *path* to current directory
cp *path1/file1 ../path2*	copy *path1/file1* to *../path2'*
cp -a ~*home/examples* .	copy all contents from the tree ~*home/examples* to the current directory -a means archive (preserves timestamps)
mv *../file* .	move: works just like cp, except that the original is thrown away
mv *name1 name2*	move can also be used for renaming files
mkdir *dirname*	make a new directory *dirname*
rmdir *dirname*	remove directory *dirname* if the directory is not empty try 'rm -rf *dirname*' make first sure you don't throw anything important away, data cannot be revived in Linux!

CTRL C CTRL L *prog* &	kill the job that is executing clean up the screen execute a program and continue working in Linux (so that you don't have to wait for your program to be finished, this also works with redirection)
man	online command manual. For instance, try 'man grep' if you want to know more about matching strings in a file. Or 'man diff' if you want to know how you can check the differences between two files.
make rehash which *prog*	when a 'Makefile' is available, make automatically compiles your program. An executable 'example' is made using the source code in 'example.f90' Refreshes your system search tables. Apply this after first time installation of a new program to prevent 'command not found' when executing your program Prints pathname of the exectutable *prog*
lpr -Plw lpq -Plw lprm -Plw *jobnumber*	print on printer 'lw' (third floor) which printing jobs are is the cue for printer 'lw' kill the job *jobnumber* on 'lw' (in case you are busy printing something you don't want to)

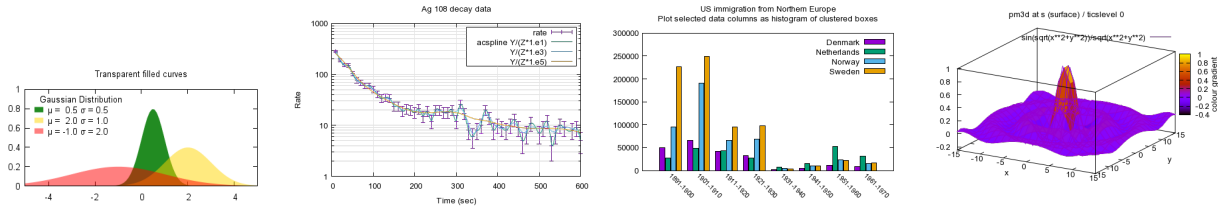
In the end it will save you a lot of time if you organise your files in a directory tree. This means that below your home directory you make several subdirectories. In those subdirectories you make subsubdirectories, etc. Then order your files in the directories, which have logical names such as: ∼home/exercise2/part3/data/.

There are many more combinations in Linux. Try 'h', 'ls -l', 'ls -la', 'l', ls -la' or try your own combinations.

Want to know more about Linux/UNIX? There is a good manual available at www.cs.uu.nl/∼piet/docs/unix.pdf (in Dutch).

D Data visualisation with GNUPLOT

Gnuplot is a portable command-line driven graphing utility for Linux, MS Windows, OSX, and many other platforms. The source code is copyrighted but freely distributed (i.e., you don't have to pay for it). It was originally created to allow scientists and students to visualize mathematical functions and data interactively, but has grown to support many non-interactive uses such as web scripting. It is also used as a plotting engine by third-party applications like Octave. Gnuplot has been supported and under active development since 1986. The official website is at this address: `http://gnuplot.info/` Here are a few examples of what can be done with gnuplot:



Gnuplot is distributed with a large set of scripts that demonstrate various features. There are all available and clearly documented at this address: `http://gnuplot.sourceforge.net/demo_4.6/`

D.1 Basic 2D plots

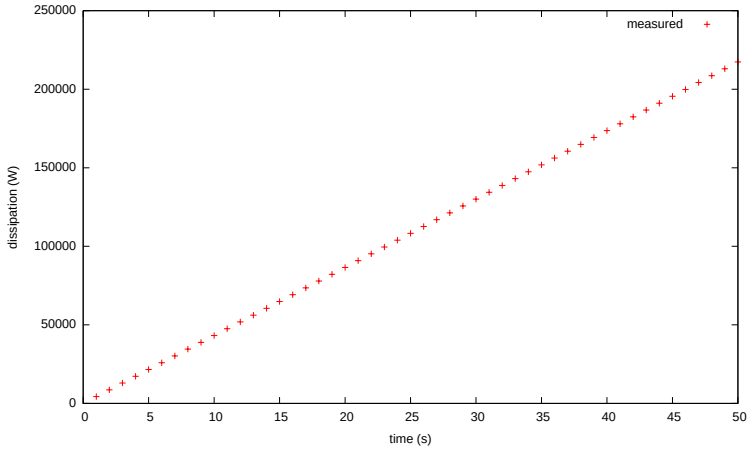
In what follows, we assume that a dataset of values has been obtained and stored in the file *datas.dat*. This file contains two columns of values. Let us create the following gnuplot script: *script1*.

```
set term postscript eps color      → output is set to be an encapsulated color postscript
set xlabel 'time (s)'              → set the label of x-axis
set ylabel 'dissipation (W)'       → set the label of y-axis
set output 'plot1.eps'             → set the name of graphics file
plot 'datas.dat' title 'measured'  → plot the data
```

We then run gnuplot on this script as follows:

```
> gnuplot script1
```

The following plot is then generated in a graphics file *plot1.eps*: [8]



[8] The filename extension *eps* stands for *extended postscript*. Such files can be visualized by (for example) the viewer program *ghostview*, usually available on Linux systems as the command `gv`.

To include graphics files in reports and manuscripts it is advisable to convert the postscript format into *portable network graphics (png)*. This can be done by the Linux `convert` utility. This is illustrated in the following example where an output *pixel density* of 300 *dots per inch (dpi)* is used:

```
> convert -density 300 plot1.eps plot1.png
```

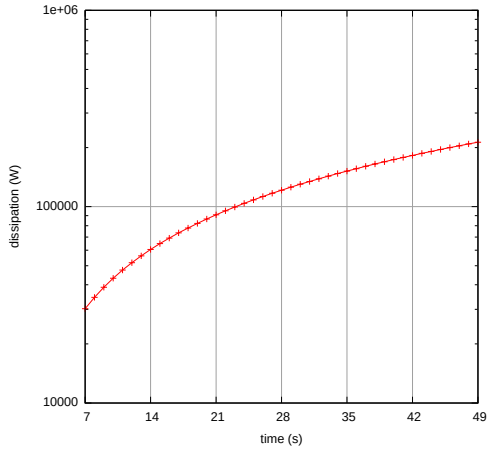
(Some) Options:

- `set grid` : display background grid
- `set log x` and `set log y` : use logarithmic scales on the x- and y-axis
- `set size square` : obtain a square plot
- `plot[-1.:4][0:5]` : provides the range of the plot in the x and y directions
- `plot 'datas.dat' with lines` : plots data with a line
- `plot 'datas.dat' with linespoint` : plots data with a line and points

Let us llustrate the use of these additional options in the following gnuplot script: *script2*.

```
set term postscript eps color                  → output is set to be an encapsulated color postscript
set grid                                       → display background grid
set xlabel 'time (s)'                          → set the label of x-axis
set ylabel 'dissipation (W)'                   → set the label of y-axis
set output 'plot2.eps'                         → set the name of graphics file
set log y                                      → set logarithmic scale on y-axis
set size square                                → trigger square plot
set xtics 7                                    → set x tics spacing to 7
plot[7:49][] 'datas.dat' with linespoint notitle   → set the name of graphics file
```

The following file *plot1.eps* is then generated:



D.2 Surfaces and 3D plotting

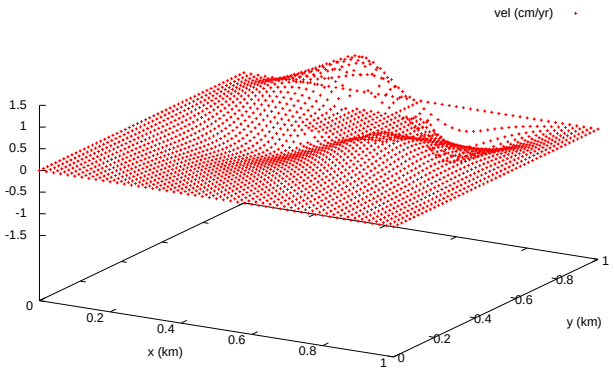
Let us now assume that the *datas3D.dat* file contains three columns of values: the x and y coordinates of points and a measured quantity at these points. gnuplot has several options for visualization of such data sets. [9]

D.2.1 Example - a surface plot of disconnected symbols

In the following example the data are plotted as a surface in a 3D box.

```
set term postscript eps color          → output is set to be an encapsulated color postscript
set xlabel 'x (km)'
set ylabel 'y (km)'
set output 'plot3.eps'
splot 'datas3D.dat' title 'vel (cm/yr)'   → splot command to generate 3D plot
```

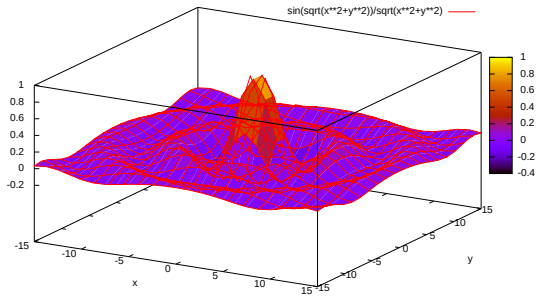
[9] Such 3-column data files are written by subroutine `writdat2d` (see Appendix **??**) for regular gridded data applied in exercises 5.6 and **??**.



D.2.2 Example - a color filled connected surface

This example illustrates the use of a connecting surface and color coding of the surface height.

```
set terminal postscript eps color
set output 'plot4.eps'
set border 4095 front linetype -1 linewidth 1.000
set samples 25, 25
set isosamples 20, 20
set xlabel "x"
set ylabel "y"
set xrange [ -15.0000  : 15.0000 ]
set yrange [ -15.0000  : 15.0000 ]
set zrange [ -0.250000 : 1.00000 ]
set pm3d implicit at s
splot sin(sqrt(x**2+y**2))/sqrt(x**2+y**2)
```

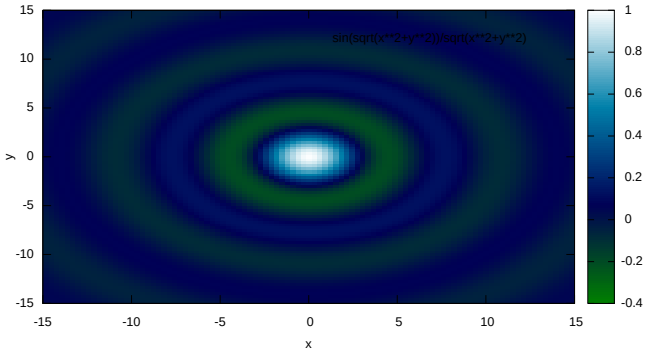


D.2.3 Example - a colored map view plot

This example illustrates the use of a regular colored 2-D map view (see `set view map`) of the 3-D data surface using an alternative color bar.

```
set terminal postscript eps color
set output 'plot5.eps'
set border 4095 front linetype -1 linewidth 1.000
set view map
set isosamples 100, 100
unset surface
set style data pm3d
set style function pm3d
set ticslevel 0
set title "gray map"
set xlabel "x"
```

```
set ylabel "y"
set xrange [ -15.0000 : 15.0000 ]
set yrange [ -15.0000 : 15.0000 ]
set zrange [ -0.250000 : 1.00000 ]
set pm3d implicit at b
set palette rgbformulae 23, 28, 3
splot sin(sqrt(x**2+y**2))/sqrt(x**2+y**2)
```



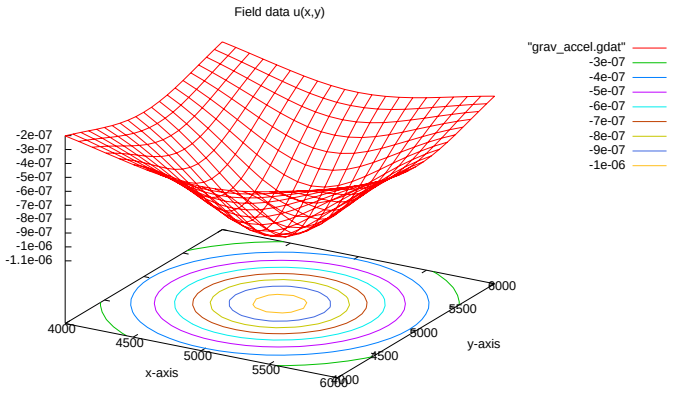
D.2.4 Example - contouring data

If one now wishes to plot isocontours of data values, Gnuplot requires the data to be saved in a text file organized in one of the two following ways:

- One set of 3 coordinates (x y z) per line, with line breaks separating the data, and an empty line to separate the rows of the matrix of data. This data format is used in data written by subroutine `writdat2d` **??**, 5.6 and **??**.

- One single value (z) per line, with line breaks separating data, and an empty line to separate the rows of the matrix of data.

```
# Gnuplot script file for plotting 3-D data
set title "Field data u(x,y)"
set xlabel "x-axis"
set ylabel "y-axis"
set contour base
set key outside
# def. range of x, y coordinates
set xrange [4e3: 6e3]
set yrange [4e3: 6e3]
# def. contour levels and surface style
set cntrparam levels auto 10
set style data lines
# outcomment these 2 lines to get x-window output
set output "plot7.eps"
set terminal postscript eps color
# plot the file between quotes
splot "grav_accel.gdat"
```



D.2.5 Example - a map view contour plot

The same data as in D.2.4 plot in map view using isocontour lines.

```
# Gnuplot script file for plotting 3-D data
set title "Field data u(x,y)"
set xlabel "x-axis"
set ylabel "y-axis"
set view 0, 0, 1, 1
set contour base
set nosurface
set xrange [4e3: 6e3]
set yrange [4e3: 6e3]
set size square
set cntrparam levels incremental -2e-6, 2e-7, 0
# outcomment these 2 lines to get x-window output
set output "grav_accel_contour.eps"
set terminal postscript eps color
set data style lines
splot "grav_accel.gdat"
```