Numerical Techniques Practicum 1

Getting started with LINUX and FORTRAN:

Solving the oscillation equation.

Daan Degrauwe

October 4, 2022

Most NWP and climate models are Fortran codes working on Linux machines. For this reason, the exercises and projects for the Numerical Techniques course will be given in such an environment.

Since not all students are familiar with Linux and/or Fortran, this first practicum is conceived as a step-by-step exercise, where both Linux basics and Fortran basics are dealt with.

The goal is to write a Fortran program that calculates a numerical solution for the oscillation equation, as given by

$$\frac{d\psi}{dt} = i\kappa\psi$$

with initial condition $\psi_{t=0} = 1$. The exact solution is given by $\psi(t) = \exp(i\kappa t)$.

For this exercise, it is assumed that you work on the HPC (High-Performance Computer) infrastructure of UGENT, and that you connect to it through the `login.hpc.ugent.be` web page.

The first thing to do (preferably before the first practical session) is therefore to ask for an HPC account on `https://www.ugent.be/hpc/en/access/faq/access`. Afterwards, you should be able to login via `login.hpc.ugent.be`. This should give you an interface like

Different tools from this web platform will be used from this web platform: the "Files" application to access files on the HPC, the "Cluster Desktop" application to enter a Linux desktop, and the "Jupyter Notebook" application for visualization of results.

What is a bit peculiar about using HPC infrastructure is that everything you do is done inside a 'job'. This means that you first have to submit/launch your job, and once it starts you can begin your work.

1 Accessing files

Under the "Files" application, hitting "Home Directory" should give you a graphical interface to your personal space on the HPC infrastructure. It lists all existing files and allows to download them, to upload new files, to view and edit (text) files, etc. However, in this practicum we'll work inside a Linux desktop to create/view/edit files. The Files application is mainly useful to exchange files with your local machine.

2 Linux desktop

Launching a desktop job is done under "Interactive Apps"-"Cluster Desktop". Choose an appropriate time for your job (typically a few hours), and set the clustername to "slaking". Then hit the "Launch" button at the bottom. After a few minutes, you should see something like the following under "My Interactive Sessions":

This means the job has started, and you can begin to work in it. Hit the "Launch Cluster Desktop" button to enter the desktop environment, which should look as follows:

The most important applications in this desktop environment are the Linux terminal (red circle) and the File browser (blue circle).

2.1 The Linux terminal

Opening a Linux terminal gives you access to what is called the *command line.* In Linux, almost everything you do is done by typing commands on the command line. For instance, try to execute the following command:

```
$  date
```

where the $ denotes that the command should be typed in the command line. The $ should not be typed itself! Hitting ENTER after this command prints the current date on the screen.

Note: Linux is case-sensitive. This means that typing DATE or Date will not work.

The Linux commands that are needed for this lesson will be explained during the course of the exercise. Appendix **??** gives an overview of some of the most common Linux commands. At this point, you get three tricks for free:

- Linux stores a history of commands you entered. To repeat a previous command, hit the UP arrow.

- Hitting the TAB key autocompletes. For example, if you type dat, and then hit TAB, it will autocomplete to date. This also works for files and directories. If multiple options exist for autocompletion, you can hit TAB twice, to see the options. For instance, if you type da, and then hit TAB twice, you should see that date and dash are possible commands.

- You can copy/paste text in the PuTTY window by selecting the text with the left mouse button, and pasting it with the right mouse button.

When working on a Linux command line, it is important to note that one works in a specific directory. Normally, this directory is visible before the $ symbol. You can also use the pwd command to print the working directory.

2.2 Data transfer between Athena and helios

As explained before, Athena and helios are actually two different machines. Luckily, it is possible to mount the same drive (the so-called "H-drive") on the two systems. This makes transfering data between both machines much easier.

To mount the H-drive on helios, execute the following command:

```
$  newns  -i
```

After providing your password, you can access the H-drive with

```
$  cd /files/${USER}/home/
```

Note: these two commands should be executed everytime you log in on helios.

2.3 Creating and modifying a text file

Creating and modifying text files is very important for the rest of this exercise (and the remainder of the course). Make sure to go through this section meticulously.

2.3.1 File creation on helios

To create a file on the H-drive, execute the following commands:

```
$   mkdir /files/${USER}/home/numtech
$   cd /files/${USER}/home/numtech
$   touch test.txt
```

Some explanation:

- `mkdir` creates a new directory

- with `cd`, you go to this directory

- `touch` creates a (empty) file

To check if the file was indeed created, run the following command:

```
$   ls
```

which gives a listing of the files in the current directory.

2.3.2 Modification on Athena

Now, we will modify this file using a text editor on Athena.

1. open the Notepad++ editor in Athena

2. inside Notepad++, open the file `H:\numtech\test.txt` (which is still empty for now).

3. put some text in the file (e.g. "Hello World!").

4. **Important**: make sure to set the end-of-line convention correctly for this file: Edit→EOL Conversion→Unix/OSX Format. It's best to do this always for files you will use in Linux.

5. Save the file

2.3.3 Review on `helios`

To check if everything went well, run the following command on `helios`:

```
$   cat test.txt
```

which should show you the text you entered in Notepad++.

3 My first Fortran program

3.1 The Fortran code

First, create a directory for this program on the H-drive, and create an empty file `osceq.F90`:

```
$   mkdir /files/${USER}/home/numtech/osceq
$   cd /files/${USER}/home/numtech/osceq/
$   touch osceq.F90
```

Open this file in Notepad++, and put the following Fortran code inside:

```
PROGRAM OSCEQ
 ! a program to solve the oscillation equation

 WRITE (*,*) "Welcome to the Oscillation Equation Solver."

 END PROGRAM OSCEQ
```

Some explanations:

- `PROGRAM` and `END PROGRAM` statements indicate the start and end of our program

- Text after a `!` denotes a comment. Use comments abundantly to make notes on the code.

- The `WRITE` statement is used to print text on the screen, or to write text to a file (see later). The argument `(*,*)` means that writing should be done to screen, in default formatting.

- Fortran is not case sensitive. This means it doesn't matter whether you write `PROGRAM` or `program` or `PrOgRaM`.

3.2   Compiling

Before being able to execute Fortran code, it must be *compiled*. This is the process of transforming a code that's readable to humans (Fortran) into a code that's readable to computers (executable files). For this, we use a *compiler*.

Compilation is quite easy (as long as your code doesn't contain any mistakes). On `helios`, we will use the `gfortran` compiler. To convert the Fortran code into an executable program, type the following in the command line:

```
$   gfortran osceq.F90 -o osceq
```

This command tells `gfortran` the following: take the file `osceq.F90`, convert it to an executable program, and put the result in the file `osceq`.

If all goes well, you should now have two files in the directory: `osceq.F90` and `osceq`. You can check this with the Linux `ls` command. If you don't see the file `osceq`, the compilation failed, most probably due to a coding mistake in the Fortran file. If you encounter errors during the compilation, try to understand what went wrong. Always look at the first error (later errors may be consequence of the first error). Ask for help if necessary: don't lose too much time on practicalities. You can use comments to indicate where you made a mistake in your program, in order to avoid such mistakes in the future.

Remark: make sure to recompile the program after every modification of the Fortran file!

3.3   Running

To execute the program, type the following on the command line:

```
$   ./osceq
```

This will print some text to screen.

3.4   Automating tasks with Linux scripts

You may have noticed that recompiling and running often go together. It is therefore useful to create a Linux script that automates these two tasks. First create a file `run.sh`:

```
$   touch run.sh
$   chmod +x run.sh
```

Where the `chmod` command marks the file as an executable script.

Next, open this file in Notepad++, and put the following text in it:

```
# Script to compile and run the oscillation equation program

# remove executable file
rm osceq

# Compile
echo "Compiling"
gfortran -o osceq osceq.F90

# Run
echo "Running"
./osceq
```

Some explanations:

- Comments in Linux scripts are indicated by a `#`.

- The `rm` command removes the existing executable.

- The `echo` command prints text to the screen.

Also, remember to set the end-of-line convention to Unix/OSX, as explained under **??**.

Running the script is done with

```
$   ./run.sh
```

4 Defining and using variables in Fortran

Let us now introduce some variables in our Fortran program. However, before making further modifications, it is good practice to take a copy of the current directory:

```
$   cd ..
$   cp -r osceq/ osceq_v1
$   cd osceq/
```

With the first command, we move one directory up (so to `/files/${USER}/home/numtech/`). The `cp` command takes a copy; to copy directories, the additional argument `-r` is required.

Now we have stored a backup, we can proceed with modifying the source code. In Notepad++, change the file `osceq.F90` as follows:

```fortran
PROGRAM OSCEQ
! a program to solve the oscillation equation

IMPLICIT NONE        ! safety to make sure all variables are declared

! declarations
REAL    :: KAPPA    ! parameter kappa in the oscillation equation (frequency)
REAL    :: DT       ! timestep
INTEGER :: NT       ! number of timesteps
COMPLEX :: PSI0     ! initial condition

WRITE (*,*) "Welcome to the Oscillation Equation Solver."

! initialize parameters
KAPPA = 0.5
DT    = 1.0
NT    = 20
PSI0  = COMPLEX(1.0,0.0)       ! complex number 1.0 + 0.0 * i

WRITE (*,*) 'Total integration time = ',NT*DT
WRITE (*,*) 'Courant number = ',KAPPA*DT

END PROGRAM OSCEQ
```

Compile and run this program by running the `run.sh` script.

The program contains the following new ingredients:

- The statement `IMPLICIT NONE` is a safety. It's good practice to put it in all your Fortran programs. It means that all variables should be declared explicitly.

- The statements `REAL`, `INTEGER` and `COMPLEX` are used to *declare* variables. Declaration is where you tell the compiler what kind a certain variable is. Besides REAL, INTEGER and COMPLEX, Fortran also knows CHARACTER and LOGICAL types.

- The variables are *assigned* their values with the `=` operator.

- The variables can then be used in calculations as shown in the `WRITE` statements

5

It is important to be aware of the types of variables. For instance, the numbers 2 and 5 are of type INTEGER. This means that 5/2 will take value 2, because dividing two variables of type INTEGER results in another INTEGER! To correctly perform divisions, make sure that one of the numbers is of type REAL. For example, 5.0/2 will give you the desired 2.5.

It should be mentioned that the organization of a Fortran program is quite strict: *all* declarations should come before the executable statements. If you want to introduce a new variable, it should be declared at the beginning of the program, even if you only use it somewhere at the end.

5 Loops

Now we get to the actual purpose of our program: the time integration. We will do this with the forward scheme, for which

$$\phi^{n+1} = (1 + i\kappa\Delta t)\phi^n$$

The time integration is a repetitive task: given the solution at the current timestep (ϕ^n), the solution at the next timestep (ϕ^{n+1}) is calculated. A repetitive task is implemented in Fortran with a DO-loop.

First, store a copy of your program with

```
$   cd ..
$   cp -r osceq/ osceq_v2
$   cd osceq/
```

Then, modify the file osceq.F90 as follows:

```
PROGRAM OSCEQ
! a program to solve the oscillation equation

IMPLICIT NONE        ! safety to make sure all variables are declared

! declarations
REAL    :: KAPPA    ! parameter kappa in the oscillation equation (frequency)
REAL    :: DT       ! timestep
INTEGER :: NT       ! number of timesteps
INTEGER :: IT       ! current timestep
COMPLEX :: PSI0     ! initial condition
COMPLEX :: PSI, PHI ! exact and numerical time-solution
COMPLEX :: II       ! imaginary unit

! initialize parameters
KAPPA = 0.5
DT    = 1.0
NT    = 20
PSI0  = COMPLEX(1.,0.)
II    = COMPLEX(0.,1.)

! set initial conditions
IT=0
PSI=PSI0
PHI=PSI0
! show values at zero'th timestep
WRITE (*,*) 't = ',IT*DT,'; PSI = ',REAL(PSI),'; PHI = ',REAL(PHI)

! start loop over timesteps
DO IT = 1,NT

   ! exact solution
   PSI = EXP(II*KAPPA*DT)*PSI

   ! numerical solution: forward scheme
   PHI = (1+II*KAPPA*DT)*PHI

   ! show values at current timestep
   WRITE (*,*) 't = ',IT*DT,'; PSI = ',REAL(PSI),'; PHI = ',REAL(PHI)
```

```
     ENDDO

     END PROGRAM OSCEQ
```

New ingredients are:

- The functions `REAL()` and `EXP()`, which respectively take the real part of a complex number, and the exponential function

- The statements `DO IT=1,NT` and `ENDDO` denoting the start and end of a *loop*. The enumerator `IT` takes the initial value of 1, and augments every timestep by 1, until it gets larger than `NT`.

When running the program, it should become clear that the forward scheme is unstable.

6   Conditions

Conditional branching in Fortran is done with the `IF` statement. For instance, we could introduce the following piece of code to check for an unstable scheme:

```
PROGRAM OSCEQ
! a program to solve the oscillation equation

... (see previous version of the program)

! start loop over timesteps
DO IT = 1,NT

  ! numerical solution: forward scheme
  PHI = (1+II*KAPPA*DT)*PHI

  ! warning if unstable
  IF ( ABS(PHI) > 10.0 ) THEN
    WRITE (*,*) 'WARNING: unstable behaviour detected'
  ENDIF

ENDDO

END PROGRAM OSCEQ
```

When running long enough ($NT \geq 21$), you should see a warning in the output.

More advanced conditions are achieved with `.AND.`, `.OR.` and `.NOT.`. If you want to execute some code when the condition is *not* fulfilled, use the `ELSE` statement.

7   Arrays

Arrays are collections of numbers. In this program, they allow to store the solution for the full time-range. Consider the following program:

```
PROGRAM OSCEQ
! a program to solve the oscillation equation

IMPLICIT NONE        ! safety to make sure all variables are declared

! declarations
REAL    :: KAPPA      ! parameter kappa in the oscillation equation (frequency)
REAL    :: DT         ! timestep
INTEGER :: NT         ! number of timesteps
INTEGER :: IT         ! current timestep
COMPLEX :: PSI0       ! initial condition
COMPLEX :: II         ! imaginary unit
```

```fortran
      COMPLEX, ALLOCATABLE :: PSI(:), PHI(:) ! arrays of exact and numerical solution

      ! initialize parameters
      KAPPA = 0.5
      DT    = 1.0
      NT    = 100
      PSI0  = COMPLEX(1.,0.)
      II    = COMPLEX(0.,1.)

      ! allocate memory for arrays
      ALLOCATE(PSI(0:NT),PHI(0:NT))

      ! set initial conditions
      IT=0
      PSI(IT)=PSI0
      PHI(IT)=PSI0

      ! start loop over timesteps
      DO IT = 1,NT

       ! exact solution
       PSI(IT) = EXP(II*KAPPA*DT)*PSI(IT-1)

       ! numerical solution: forward scheme
       PHI(IT) = (1+II*KAPPA*DT)*PHI(IT-1)

      ENDDO

      ! show values
      WRITE (*,*) 'PSI = ',REAL(PSI)
      WRITE (*,*) 'PHI = ',REAL(PHI)

      ! release memory
      DEALLOCATE(PSI,PHI)

      END PROGRAM OSCEQ
```

Working with arrays requires the following:

- During the declaration, you should add the `ALLOCATABLE` attribute, and specify the number of dimensions. In this program, one-dimensional arrays are used, hence the `(:)`. Two-dimensional arrays would be declared with `(:,:)`

- The `ALLOCATE` statement tells the compiler what the size of the array should be, and makes sure the necessary memory is reserved for this array

- A single element of an array is accessed with the `()` construct. For instance, we calculate `PHI(IT)` from `PHI(IT-1)`

- At the end, the reserved memory should be released with the `DEALLOCATE` statement.

Quite important to remember when working with arrays is that you shouldn't exceed the limits of the array. For example, using `PHI(NT+1)` in our program would lead to erroneous results or a crash of the program, because `PHI` was allocated with an upper bound of `NT`. You can tell the compiler to detect such out-of-bound errors by compiling with the switch `-fbounds-check`:

```
$   gfortran -fbounds-check osceq.F90 -o osceq
```

8 Subroutines and modules

For larger computer programs (NWP and climate models often contain millions of lines of code), it becomes necessary to organize the code in a structured way. This is achieved with *subroutines*. A subroutine is a piece of code that can be called from somewhere else in the code.

In principle, variables that are declared in the main part of the program are not known in the subroutine. Two mechanisms exist to transfer information to and from subroutines. Either by using *arguments*, or by using *global variables*. Global variables are implemented in Fortran by putting them in a *module*.

It is common practice to put all subroutines and modules in separate files.

Let's organize our oscillation equation program in the following pieces:

- a module CONSTANTS containing all constant variables;

- a subroutine SETUP_CONSTANTS to setup the values of these constant variables;

- a subroutine TIMELOOP to perform the time-loop;

- a subroutine WRITE_RESULT to write the results to a file;

- the main program, making calls to the other routines.

Each of these pieces is put in a separate Fortran file. The file constants.F90 contains

```
MODULE CONSTANTS
   ! a module contains global variables.
   ! These are accessible from all subroutines and from the main program

   IMPLICIT NONE

   ! universal constants
   COMPLEX :: II        ! imaginary unit

   ! oscillation equation parameters
   REAL    :: KAPPA     ! parameter kappa in the oscillation equation (frequency)
   COMPLEX :: PSI0      ! initial condition

   ! discretization variables
   REAL    :: DT        ! timestep
   INTEGER :: NT        ! number of timesteps

END MODULE CONSTANTS
```

The file setup_constants.F90 contains

```
SUBROUTINE SETUP_CONSTANTS
   ! subroutine to assign constants appropriate values

   USE CONSTANTS        ! all variables from this module are now known in this subroutine

   IMPLICIT NONE

   II   = COMPLEX(0.,1.)

   KAPPA = 0.5
   PSI0  = COMPLEX(1.,0.)

   DT    = 1.0
   NT    = 100

END SUBROUTINE SETUP_CONSTANTS
```

The file timeloop.F90 contains

```
SUBROUTINE TIMELOOP

   USE CONSTANTS

   IMPLICIT NONE

   ! declare local variables
```

```
   INTEGER                :: IT            ! current timestep
   COMPLEX, ALLOCATABLE :: PSI(:), PHI(:)  ! arrays of exact and numerical solution

   ! allocate memory for arrays
   ALLOCATE(PSI(0:NT),PHI(0:NT))

   ! set initial conditions
   IT=0
   PSI(IT)=PSI0
   PHI(IT)=PSI0

   ! start loop over timesteps
   DO IT = 1,NT

     ! exact solution
     PSI(IT) = EXP(II*KAPPA*DT)*PSI(IT-1)

     ! numerical solution: forward scheme
     PHI(IT) = (1+II*KAPPA*DT)*PHI(IT-1)

   ENDDO

   ! write the result
   CALL WRITE_RESULT(PSI,PHI)

   ! free memory
   DEALLOCATE(PSI,PHI)

END SUBROUTINE TIMELOOP
```

The file `write_result.F90` contains

```
SUBROUTINE WRITE_RESULT(PSI,PHI)

   ! write the result to a file
   USE CONSTANTS

   IMPLICIT NONE

   ! arguments
   COMPLEX, INTENT(IN) :: PSI(0:NT), PHI(0:NT)

   ! auxiliary results
   INTEGER :: IT     ! timestep

   ! open a file
   OPEN(FILE='output.dat',UNIT=20)

   ! write heading
   WRITE (20,'(A6,A16,A16)') 'time','exact','numerical'

   ! write solutions at all timesteps to the file
   DO IT=0,NT
     WRITE (20,'(F6.2,E16.8,E16.8)') IT*DT, REAL(PSI(IT)), REAL(PHI(IT))
   ENDDO

   ! close the file
   CLOSE(UNIT=20)

END SUBROUTINE WRITE_RESULT
```

And finally, the file `osceq.F90` contains

```
PROGRAM OSCEQ
   ! a program to solve the oscillation equation
```

```
    IMPLICIT NONE

    ! initialize constant parameters
    CALL SETUP_CONSTANTS()

    ! time loop
    CALL TIMELOOP()

END PROGRAM OSCEQ
```

The compilation of multiple files is done as follows:

```
$   gfortran constants.F90 setup_constants.F90 timeloop.F90 write_result.F90 osceq.F90 -o osceq
```

Remarks and new concepts:

- One can hardly say that our program became simpler by reorganizing into subroutines. However, each of the building blocks is quite simple in itself. It's much easier to find your way through the program now.

- Calling a subroutine is done with the `CALL` statement.

- Arguments have an `INTENT` attribute, which specifies whether it's an input (`IN`) or an output argument (`OUT`).

- It's very important that the argument declaration in the subroutine matches the argument that is passed in the `CALL` statement! You will get strange behaviour if this is not the case. 'Matching' means that the type (`COMPLEX`, ...) should be the same, as well as the dimension if it's an array.

- To make global variables accessible in a subroutine, the `USE` statement should be used.

- Writing to a file is done in three steps:

    1. open the file with the `OPEN` statement. The filename is specified, and a *unit* number is assigned. You can freely choose this unit, as long as it is not in use for another file. You also shouldn't take 0, 1 or 6 as unit number.
    2. write stuff to the file with the `WRITE` statement. Specify the unit number of the file you want to write to, as well as the *format*. The code shows how to specify the formats for character strings and for floating point real numbers.
    3. close the file with the `CLOSE` statement.

- for the compilation, it's important to put the Fortran files containing modules (in our case `constants.F90`) *before* the Fortran files using these modules.

9   Visualizing the output with R

Fortran is a programming language that's intended to perform calculations. Visualizing results is commonly done with other programs, such as Matlab, R, gnuplot, python, etc. All of these programs are capable of reading numerical data from a file (like the one created by our program). For this course, we will use RStudio (a graphical interface to R) to postprocess (visualize and inspect) the results of our calculations. RStudio is available on Athena, and can easily be installed under Windows, Linux or OSX.

It may seem a bit awkward that we perform the calculations on a different machine and with a different programming language than what we use for the postprocessing. In fact, this situation is quite common in NWP and climate research, where the calculations are done on a dedicated High Performance Computing (HPC) Linux-machine. Such a machine is intended for heavy calculations, but not for interactive or graphical work. So the output is copied to another server or to a PC, where the scientist can postprocess the data.

To visualize the results of the oscillation equation program:

1. start RStudio from Athena

2. set the working directory to the H-drive by hitting Ctrl-Shift-h. Choose the directory where you run your Fortran code for the oscillation equation.

3. read the data from the file `output.dat` with

```
>    y=read.table('output.dat',header=TRUE)
```

4. plot the exact solution with

```
>    plot(y$time,y$exact,ylim=c(-3,3),type='b',pch=1,col=1)
```

5. add the numerical solution to the plot with

```
>    points(y$time,y$numerical,type='b',pch=2,col=2)
```

Instead of typing all these commands, it's more convenient to put them in a text file, and have RStudio execute this file. So, create a file `show_results.R` containing the following:

```
# R code to show results of the oscillation equation
# the results are supposed to be stored in 3 columns
# (time-exact-numerical) in a file 'output.dat'

# open window
windows(20,10)

# read the file
y=read.table('output.dat',header=TRUE)

# plot exact and numerical result
plot(y$time,y$exact,ylim=c(-3,3),type='b',pch=1,col=1,xlab='time',ylab='solution')
points(y$time,y$numerical,type='b',pch=2,col=2)

# add legend
legend("topleft",c("exact","numerical"),pch=1:2,col=1:2,lty=1)
```

This file can then be executed in RStudio by typing

```
>    source('show_results.R')
```

10   Exercises

1. Implement the backward and the trapezium schemes. What about their stability and phase error?

$$\phi^{n+1} = \frac{1}{1 - i\kappa\Delta t}\phi^n \qquad \text{(backward)}$$

$$\phi^{n+1} = \frac{1 + 0.5i\kappa\Delta t}{1 - 0.5i\kappa\Delta t}\phi^n \qquad \text{(trapezium)}$$

2. Implement the Matsuno scheme, and check the damping as a function of $\kappa\Delta t$. Is it accelerating or decelerating?

$$\tilde{\phi} = (1 + i\kappa\Delta t)\phi^n$$
$$\phi^{n+1} = \phi^n + i\kappa\Delta t\tilde{\phi}$$

3. Implement the Leapfrog scheme.
$$\phi^{n+1} = \phi^{n-1} + 2i\kappa\Delta t\phi^n$$

Use the forward scheme during the first timestep:

$$\phi^1 = (1 + i\kappa\Delta t)\phi^0$$

4. Try to excite the computational mode in the Leapfrog scheme (by sabotaging the first timestep).

5. Implement the Robert-Asselin filter to damp the computational mode.

$$\phi^{n+1} = \overline{\phi^{n-1}} + 2i\kappa\Delta t\phi^n$$

$$\overline{\phi^n} = \phi^n + \gamma\left(\overline{\phi^{n-1}} - 2\phi^n + \phi^{n+1}\right)$$

A   Linux commands

Getting help

`man`	show manual pages for a command (exit with `q`)
	e.g. `man date`
`help`	show help
	e.g. `help cd`
`--help`	show help info
	e.g. `date --help`

Navigation

`cd`	change directory
	e.g. `cd /files/${USER}/home/numtech/`
`cd ..`	go one directory up
`pwd`	show current directory
`ls`	list contents of directory
`mkdir`	create a directory
	e.g. `mkdir testdir`

File manipulation

`touch`	create empty file
	e.g. `touch test.txt`
`cp`	copy a file
	e.g. `cp test.txt test2.txt`
`cp -r`	copy a directory
	e.g. `cp -r testdir/ testdir2`
`rm`	remove a file
	e.g. `rm test2.txt`
`rm -r`	remove a directory
	e.g. `rm -r testdir2`
`mv`	move (rename) a file or directory
	e.g. `mv test.txt test2.txt`

File contents

`cat`	show file contents
	e.g. `cat test.txt`
`less`	show file contents (exit with `q`)
	e.g. `less test.txt`
`diff`	show difference between two (text) files
	e.g. `diff test.txt test2.txt`

B   More advanced interaction with Linux machines

B.1   Connecting with a graphical interface enabled: Xming

Although the command-line is the most important way to interact with a Linux system, it is sometimes desirable to have a graphical interface (e.g. to show figures). For this, it is necessary to start the program Xming *before* starting PuTTY. Xming is installed in the PC rooms, but not on Athena. It can easily be installed on your own laptop though.

After starting Xming by double-clicking it, start PuTTY as before, but make sure to enable "X11 forwarding" under Connection→SSH→X11.

To test whether the graphics work, try to execute the program `xclock` from the command line. A small clock window should appear.

Note: this does not work with Athena's PuTTY.

B.2 Transferring files: WinSCP

Mounting the H-drive as a shared directory between Athena and `helios` is by far the easiest way to transfer files between both systems. However, the procedure described in this section is more general in the sense that it works for any Windows and any Linux machine.

WinSCP is a free program that allows to transfer data between a Windows machine (e.g. your laptop) and a remote server (e.g. `helios`). It is installed in the PC rooms and on Athena, and it's easy to install on your laptop as well.
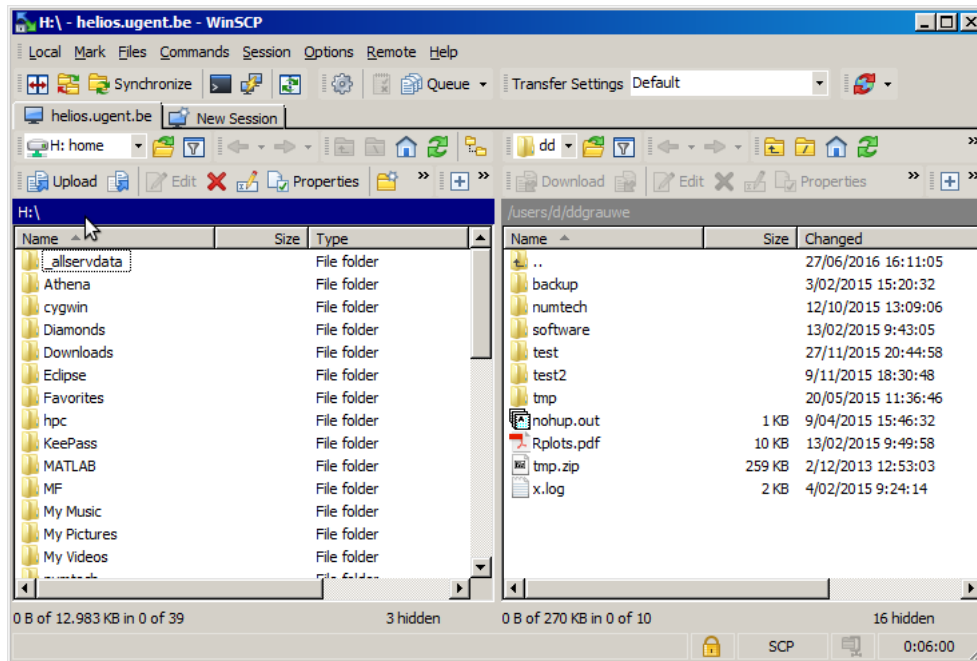
To connect to `helios`, set the File protocol to *SCP*, and the Host name to *helios.ugent.be*:
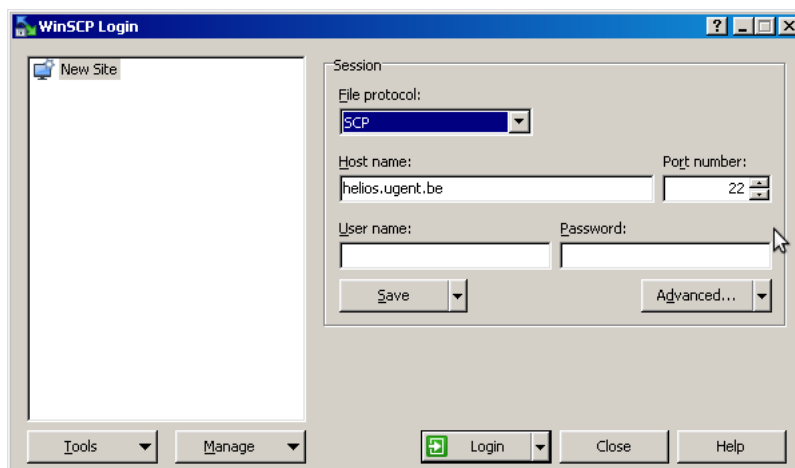
Hit the "Login" button, enter your account name and password, and you'll get something like this:
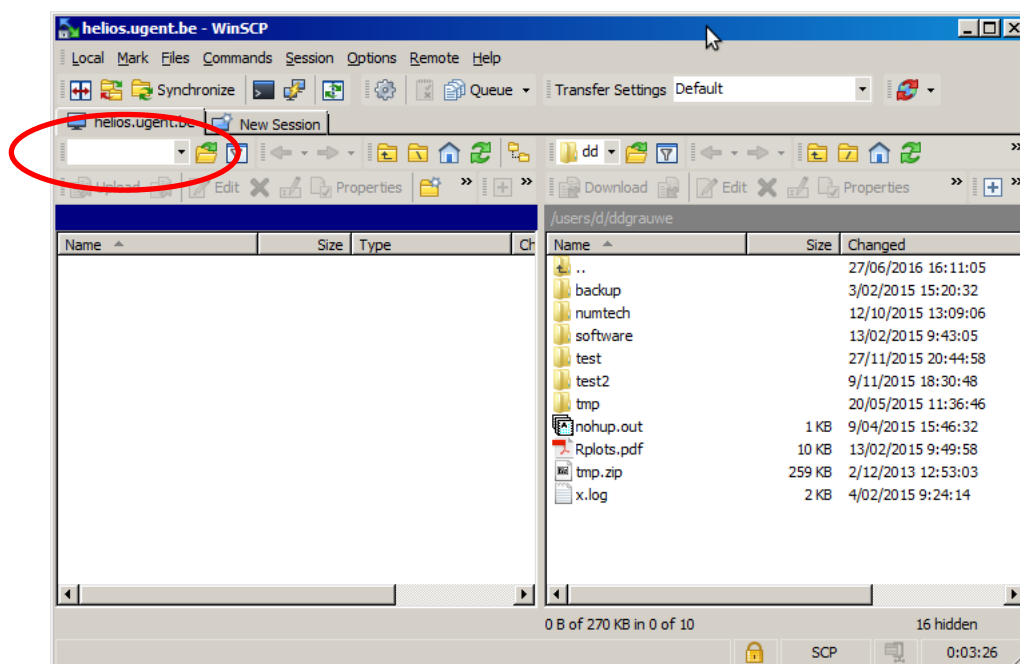
Next, select the H-drive in the circled area of the previous screenshot, to end up with

In this window, the left panel represents your local file system (i.e. your laptop, the PC of the PC room, or the H-drive when using Athena), while the right panel represents the remote system (i.e. `helios`). Transferring files can be done by dragging between the two panels.

C Overview of programs

Program	OS	Description
PuTTY	Windows/OSX	make connection from Windows machine to Linux machine
Xming	Windows	enable graphical connection between Windows and Linux machine. To be used in combination with PuTTY.
Notepad++	Windows/OSX	text editor
WinSCP	Windows	transfer files between two machines
RStudio	Windows/OSX/Linux	graphical interface to R; used for postprocessing and visualization
gfortran	OSX/Linux	Fortran compiler