Numerical Techniques Practicum 1

Basics of LINUX and FORTRAN:

Solving the oscillation equation (again).

Daan Degrauwe

November 16, 2023

Most NWP and climate models are Fortran codes working on Linux machines. For this reason, this practicum is aimed at teaching the basics of working in such an environment. 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.

1 The Linux command line

1.1 Opening the Linux command line on the UGENT HPC

Most things in Linux are done by executing commands from what is called the *command line*. The command line is accessed by opening a terminal. Let's try this out:

1. Login on `https://login.hpc.ugent.be`
2. Hit the "Open in terminal" button. This should open a terminal in a new browser tab.

Now, you can type a Linux command, for instance

```
$   date
```

Note that you shouldn't type the '`$`'; this is just used in this document to indicate when something should be executed on the Linux command line. 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 A gives an overview of some of the most common Linux commands. At this point, you get the following tricks for free:

- Selecting text in the terminal automatically places it on the clipboard. Pasting is done with `CTRL-V`
- 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 `cat`, and then hit `TAB` twice, you should see that `cat`, `catman` and `catchsegv` are possible commands.

1.2 Creating, navigating and listing directories

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. ~ denotes the home directory. You can also use the `pwd` command to print the working directory.

On the command line, execute the following commands one-by-one:

```
$   mkdir -p ${HOME}/numtech/practicum_linux_fortran
$   cd ${HOME}/numtech
$   ls
$   cd  practicum_linux_fortran
```

Some explanation:

- `mkdir` creates a new directory, the `-p` option lets you create nested directories.
- with `cd`, you navigate to this directory
- with `ls`, you list the contents of a directory

Note: you can also perform these actions from the `https://login.hpc.ugent.be` "Files" menu.

1.3 Creating, modifying and viewing files

To create an (empty) file, execute the following command:

```
$   touch test.txt
```

To check if the file was indeed created, you can run the `ls` command.

Modifying files from the command line is possible (e.g. using the Linux program `vi` or `emacs`), but this is not so intuitive. So a different approach is used here:

1. From `https://login.hpc.ugent.be`, open the "Files" menu.
2. Browse to the directory where you created the file.
3. Hit the three-dot icon next to the file, and select "Edit" from the drop-down menu. This should open a new tab where you can edit the file.
4. Put some text, and hit the "Save" button.

Next, you can check the contents of a file with the following command in the command line:

```
$   cat test.txt
```

which should show you the text you entere. Finally, remove the test file with

```
$   rm test.txt
```

2 My first Fortran program

2.1 The Fortran code

First, create a directory for this program, and create an empty file `osceq.F90`:

```
$   mkdir ${HOME}/numtech/practicum_linux_fortran/osceq
$   cd ${HOME}/numtech/practicum_linux_fortran/osceq/
$   touch osceq.F90
```

Open this file in an editor, 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`.

2.2 Compiling

Before being able to execute Fortran code, it must be *compiled*. This is the process of transforming a code that is (more or less) 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). 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 Fortran-code from 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). 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 save your Fortran file and to recompile the program after every modification of the Fortran code!

2.3 Running

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

```
$   ./osceq
```

This will print some text to screen.

2.4 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 an editor tab, 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.

Running the script is done with

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

3 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 `${HOME}/numtech/practicum_linux_fortran`). 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 an editor tab, change 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
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

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.

4 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 = 0,NT-1
```

```
  ! 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.

5 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 = 0,NT-1

  ! 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`≥ 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.

6 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
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 = 0,NT-1

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

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

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
```

7 Subroutines

To organize the code, subroutines can be used, like functions in python. A subroutine is a piece of code that can be called from somewhere else in the code. For example, let us define separate subroutines for different time schemes. Create a backup of your directory, and then modify the code in `osceq.F90` to the following:

```
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
COMPLEX, ALLOCATABLE :: PSI(:), PHI_FWD(:), PHI_TRPZ(:) ! 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_FWD(0:NT),PHI_TRPZ(0:NT))

! set initial conditions
IT=0
PSI(IT)=PSI0
PHI_FWD(IT)=PSI0
PHI_TRPZ(IT)=PSI0

! start loop over timesteps
DO IT = 0,NT-1

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

 ! numerical solution: forward scheme
 CALL OSCEQ_FORWARD(PHI_FWD(IT),KAPPA,DT,PHI_FWD(IT+1))

 ! numerical solution: forward scheme
 CALL OSCEQ_TRAPEZIUM(PHI_TRPZ(IT),KAPPA,DT,PHI_TRPZ(IT+1))

ENDDO

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

! release memory
DEALLOCATE(PSI,PHI_FWD,PHI_TRPZ)

END PROGRAM OSCEQ


SUBROUTINE OSCEQ_FORWARD(PHI_CURRENT,KAPPA,DT,PHI_NEXT)
! subroutine to take one timestep with the forward scheme

IMPLICIT NONE
```

```
! arguments
COMPLEX, INTENT(IN)  :: PHI_CURRENT    ! solution at current timestep
REAL,    INTENT(IN)  :: KAPPA          ! frequency
REAL,    INTENT(IN)  :: DT             ! time step
COMPLEX, INTENT(OUT) :: PHI_NEXT       ! solution at next timestep

! local variables
COMPLEX :: II=COMPLEX(0.,1.)

! time scheme
PHI_NEXT = (1+II*KAPPA*DT)*PHI_CURRENT

END SUBROUTINE OSCEQ_FORWARD


SUBROUTINE OSCEQ_TRAPEZIUM(PHI_CURRENT,KAPPA,DT,PHI_NEXT)
! subroutine to take one timestep with the trapezium scheme

IMPLICIT NONE

! arguments
COMPLEX, INTENT(IN)  :: PHI_CURRENT    ! solution at current timestep
REAL,    INTENT(IN)  :: KAPPA          ! frequency
REAL,    INTENT(IN)  :: DT             ! time step
COMPLEX, INTENT(OUT) :: PHI_NEXT       ! solution at next timestep

! local variables
COMPLEX :: II=COMPLEX(0.,1.)

! time scheme
PHI_NEXT = (1+0.5*II*KAPPA*DT)/(1-0.5*II*KAPPA*DT)*PHI_CURRENT

END SUBROUTINE OSCEQ_TRAPEZIUM
```

Some remarks on this code:

- In fact, it is common practice to put different subroutines in separate files. For this practicum, however, everything is kept in a single file.
- Calling a subroutine is done with the `CALL` statement.
- Arguments to subroutines 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.
- Variables that are declared in the main part of the program are not known in the subroutine (except if they are passed as an argument). For the example above, the imaginary unit `II` was redefined inside the subroutines.

8 Writing the results to a file

Writing numerical results to a file is done with the `WRITE` statement, but the file should be opened before and closed afterwards. Change the code in the Fortran file as follows:

```
PROGRAM OSCEQ

! ... (keep from before)

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

```
  WRITE (*,*) 'PHI_TRPZ = ',REAL(PHI_TRPZ)

  ! write to file
  OPEN(UNIT=8,FILE='results.dat')
  WRITE (8,'(E16.8)') REAL(PSI)
  WRITE (8,'(E16.8)') REAL(PHI_FWD)
  WRITE (8,'(E16.8)') REAL(PHI_TRPZ)
  CLOSE(UNIT=8)

  ! release memory
  DEALLOCATE(PSI,PHI_FWD,PHI_TRPZ)

  END PROGRAM OSCEQ

      ! ... (keep subroutines from before)
```

9 Importing and plotting results in python

Create a Jupyter notebook in the directory where your Fortran program runs, and put the following code in a code cell:

```
# load libraries
%matplotlib notebook
import matplotlib.pyplot as plt
import numpy as np

# read data from results.dat
results=np.loadtxt('results.dat')

# reshape into 3 x (nt+1) array
results=np.reshape(results,(3,len(results)//3))

# plot data
psi=results[0,:]
phi_fwd=results[1,:]
phi_trpz=results[2,:]

plt.figure()
plt.plot(psi,label='exact solution')
plt.plot(phi_fwd,label='forward scheme')
plt.plot(phi_trpz,label='trapezium scheme')
plt.ylim([-2,2])
plt.legend()
plt.show()
```

10 Submitting jobs to the HPC queue

So far, we have been working on the `donphan` cluster, which is okay for small computations. However, for more serious work (such as weather and climate modeling!) it is no longer possible to work interactively on a system. Instead, you need to *submit* your computation script, and it will only be *executed* once the required resources (CPU cores) are available on the system. This means you may have to wait for some time (maybe days!) before your results become available.

Guidelines on how to do this for the UGENT clusters can be found on `https://docs.hpc.ugent.be/Linux/`. In this practicum, we will just illustrate the general idea with a very simple job.

First, create (or modify) your compilation/run script to the following:

```
#!/bin/bash
#PBS -N osceq_job             ## job name
#PBS -l nodes=1:ppn=1         ## single-node job, single core
#PBS -l walltime=0:00:10      ## estimated run time
```

```
# go to right directory
cd ${HOME}/numtech/practicum_linux_fortran/osceq

# remove executable and results
rm osceq results.dat

# compile
gfortran osceq.F90 -o osceq

# run!
./osceq
```

Next, indicate that you want to use the `doduo` cluster by executing the following on the command line:

```
$   module swap cluster/doduo
```

Then, submit the job with

```
$   qsub run.sh
```

The job will now enter the queue. You can check its status with the command `qstat`. This will tell you the job ID and whether it is queued (`Q`), running (`R`), or completed (`C`). Once it is completed, you will have the output of the job (what it normally shows on the screen) in the files `osceq_job.o{JOB_ID}` and `osceq_job.e{JOB_ID}`. Review the content of these files with

```
$   cat osceq_job.o* osceq_job.e*
```

Some important notes:

- Try to put a realistic runtime in your job header. If the estimated runtime is too short, your job will be killed before it finishes. If it is too long, you may have to wait for a long time before your job starts.
- Test your scripts, computations, etc. before you submit them; especially for large and/or long experiments.
- Only request more CPU cores when you need them. Not everything goes faster on more CPUs.

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`

Job submission

`qsub`	submit a job to the queue
`qstat`	inquire job status
`qdel`	delete a job from the queue