Introduction to Fortran

Dr. Nikos Daskalakis

February 10, 2020

Contents

Preface

This document is prepared as support document/help for the Fortran course of the PEP masters program. It is by no means a complete guide or a textbook. It serves the needs of the specific course and will be updated accordingly in the course of the course.

Instructions on how to connect to the computer where you will do all the coding and the experiments, as well as a very basic introduction to the Linux terminal and the vi editor will be provided during the first lecture, along with your account informations

IMPORTANT NOTE: Before getting to the class prepare your computer for the lesson! Follow the instructions for the VPN connection to the university VPN server from here.

If you are using a windows computer you will have to have some kind of terminal in order to be able to connect to the lesson server. If you have Windows Subsystem for Linux (WSL), or cygwin installed, then you are already covered. If you do not know what that is, or you don't know if you have any terminal installed, please download and install the free version of MobaXterm program from here

1 Introduction

`Fortran`, derived from Formula Translation, is a general-purpose, **compiled** programming language that is especially suited to numeric computation and scientific computing.

1.1 Brief history of Fortran

Fortran is a programming language with a really long history. It was first developed in 1957 by a team of programmers in IBM. Its great success was based on the fact that it was easier to code in Fortran than assembly (the "native" language of the computer), with minimal compromise on the computation speed. In 1966 Fortran got its first coding standards, making it this way machine independent (the same code could run on different computers, where the versions up to this point were deeply machine dependent, meaning that the code produced for one computer could not be transferred to another computer).

Fast forward to 1990, Fortran 90 becomes available. This is the version that we will base this course upon. Fortran 90 was fully compatible with previous versions, but added a lot of improvements to the language. Since then there have been several updates to the language (Fortran 95 in 1995, Fortran 2003, Fortran 2008 and the latest one, Fortran 2018, in November 28, 2018), but these updates are not important for this course.

1.2 Fortran in Science and Engineering

Fortran is the primary language for some of the most intensive super-computing tasks, like weather forecasting, climate modelling, atmospheric process modelling, fluid dynamics, data analysis, computational economics etc. Even languages like python rely on Fortran for the computationally heavy processes.

Apart from historical reasons, scientist still code in Fortran because of its speed (it is essentially impossible to write slow code) and its easy to understand structure and syntax.

2 Material for the course

The chapters that follow are meant to cover as much as possible for what you will need for this course. This book cannot replace the lectures and probably will not cover all the material that will be taught in the class. It is supposed to serve as a support document for what will be needed in the class and for the exercises.

2.1 Where do I start to create a program?

2.1.1 Background

The first step is to understand a few basic things about computers and programming languages. Computers speak their own language, known as machine language. Each different type of CPU "speaks" its own, unique machine language. Machine languages consist of numbers only. The closest humans reach to this, is the assembly languages. These are very similar to machine, but allow the programmer to substitute names for numbers, making it much easier to program in. Fortran belongs in a category of languages called *High Level Languages (HLL)*. These languages allow the programmer to write code that is independent of the type of the computer that the code will run on. The main advantage of HLLs is that they are easier to read, write and maintain over low level languages.

Most of the languages you ever heard of, and most probably all the programming languages that you are ever going to use, are HLLs. In order for the computer to understand what is written in a program written in these languages, you first have to translate the program to machine language so that the computer can understand it. And here comes the first big distinction of programming languages, based on the method used to translate the code to machine language. There are two categories: *Compiled* languages and *interpreted* languages.

Interpreted languages, like Python, IDL, Matlab, R, are high level languages that use *interpreters* to run. In reality what an interpreter does, is translate the high-level instructions to an intermediate form, which then executes.

Compiled languages, like Fortran, C, C++, use a compiler to translate the high-level instructions (source code) directly to machine language. What the compiler does, is to first transform the source code into object code and the produce and executable program by combining the information in the object code using a linker.

An interpreter can immediately execute high-level programs, where with compiled languages one has to always successfully compile the program first. The advantage of an interpreter is the fact that it does not need to go through the compiler first to run the code. The advantage of the compiled programs is that they generally run faster than interpreted programs for the same set of calculations.

2.1.2 Steps to follow for writing code

For creating a program in any language there are some basic steps that you have to follow:

1. Understand the problem:

   You cannot start coding without knowing what you need to calculate.

2. Develop the algorithm:

   Next step is to figure out all the steps needed for solving your problem. A very nice tool that helps with that are flow charts. In flowcharts you write down the sequence of computational steps that will combine your input data to the solution of the problem, hence the output data.

3. Translate the algorithm to code

   This is the step where you actually write down the code

4. Syntax error checking

   When the code is ready you have to check for syntax errors for the executable to be created. For Fortran this is done using a compiler.

In this class we will ber writing in Fortran 90/95. The compiler we are going to use is called `gfortran`. When you prepare a program you will have to give the file a name with the suffix `.f90`, for the compiler to know what standards to use for syntax error checking and executable creation.

If you created a program in a file called `program.f90` you can compile it using:

`gfortran program.f90`

If no errors come up, then the executable will be created. The executable will be named `a.out`. If you want to give a different name to the executable then you have to compile it with the output flag:

`gfortran -o name.exe program.f90`

In this case the executable will be named `name.exe`

For running the program write in the terminal:

`./name.exe`

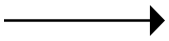
2.2 Flow charts and their use in coding (basis of all proper coding)

A flow chart is a type of diagram that represents an algorithm. The flowchart shows all the steps the algorithm takes as boxes of various shapes, with each shape having a different meaning. The order of the steps is denoted by connecting the boxes with arrows.

The basic "boxes" used in a flowchart are:

- Terminal:Rounded rectangles used for indicating flowchart's starting and ending points.

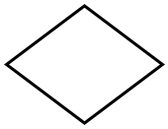
- Flow Lines: Connecting the steps of the flowchart.



- Input/Output: Parallelograms designate input or output operations.

- Process: The rectangles depict a process such as a mathematical computation or a variable assignment.

- Decision: The diamond represents the true/false statements.



A flowchart is independent of the language used to code and it is bound to the problem. This is why in a flowchart we do not use code. Instead we use what is called pseudocode. Basically in each step of a flowchart we describe the algorithm in *human readable* language.

2.3 Basic terms

In order to follow this book (and the course) you must know what a few terms mean.

A variable in a program is a container for data that the program **can change**. Variable are declared at the very beginning of a Fortran program, defining what kind of data they can store. The different types of variables can be:

- character: A character string
- integer: An exact representation of a positive or negative integer number.
- real: Floating point numbers (decimal numbers).

- complex: An ordered pair of numbers corresponding respectively to the real and imaginary parts of a complex number, both of which numbers are individually of type REAL.
- logical: A variable that can take the values true or false.

A parameter is a container for data that the program **cannot change**. Parameters can the same kinds as variables and remain constant throughout the program.

Keywords are words that the compiler understands and performs a task based on them.

NOTE: Fortran does not reserve words. Any word can be used in any context. Nevertheless it is not advisable to use keywords as variable names, in order to avoid confusion.

2.4 Structure of a Fortran program

Fortran code follows a strict structure. All Fortran programs should start with the following template:

```
program name
    implicit none
    [specification part]
    [execution part]
    [subprogram part]
end program
```

Even though not all of the above are mandatory, it is advisable to use this template in your code. All Fortran programs should start with the line:

```
program program_name
```

The keyword `program` denotes the start of a program and it is always followed by the name of the program (one word).

The second line reads `implicit none`. This tells the compiler to not allow non-declared variables. Even if it makes coding extremely more strict, it is one of the most helpful keywords in coding since it helps avoid confusion and errors in the code. This keyword is considered to be mandatory in all programs for this course.

The parts in brackets are the programmers defined parts. In the specification part the programmer has to declare all the variables and parameters of his program. The execution part is where everything that the programmer wants the program to do should be coded.

The last line of the program reads:

```
end program
```

This is the only mandatory statement in a Fortran code. The compiler will not translate the program if there is no end line.

```
end
```

The code snippet above is a fully correct, full program written in Fortran language.

2.4.1 Variable declaration

As explained before, a variable can be considered as a box capable of containing a value of a certain type. A variable has a name and a type. There are a few rules one has to follow while choosing the variable name:

- It must be smaller than 31 characters long.
- The first character of a variable **has** to be a letter.
- The remaining characters can be *letters, numbers or underscores.*
- A variable name is *case **in**sensitive.* Name, name, nAmE, NAME and naME are all identical for the Fortran compiler.

- A variable name can be a Fortran keyword, but this tactic is non-advisable.
- It is a good strategy to use meaningful names for your variables.

Variable declaration happens right after the `implicit none` statement in the specification part of the program. The type statement of a variable has the following for:

```
type-specifier :: list
```

where type specifier is one of the different types of Fortran variables (`real`, `integer`, `character`, `logical`, `complex`), and list is a list of variable names separated by commas:

```
integer :: i,j
real    :: conc, mass
```

The snippet above declares two integer variables (`i` and `j`) and two real variables (`conc` and `mass`). Note the white space between `real` and `::`. Fortran recognizes consecutive spaces as one.

In the type-specifier part of a variable declaration the programmer can also give specifications of the variable. If for example you want to declare a parameter integer, with the value of `5`:

```
integer, parameter :: i=5
```

Note the `parameter` declaration right next to the `integer`. This means that `i` is not allowed to change its value throughout the program. It will always remain equal to 5.

One important specification is the length of `character` variables. If no length is declared, then the variable can hold strings of length equal to one. To change that you should declare the length of the string that the variable should be able to hold:

```
character(len=10) :: name
character(10)     :: name2
```

Here `name` and `name2` can hold strings of maximum 10 characters long.

Another type of variables that are commonly used in programming are arrays. Arrays are variables that can hold more than one value at a time. In Fortran, arrays declaration is happens in the `specification part` and looks like:

```
real, dimension(5) :: x
```

In the snippet above we declare an one dimension array of reals, of size 5. This means that it can hold 5 values at the same time. If you also want to assign starting values to this array, then:

```
real, dimension(5) :: x=(/1.,2.,3.,4.,5./)
real, dimension(5) :: y
data y(/1.,2.,3.,4.,5./)
```

In the snippet above we declare two arrays of reals, each holding the values from 1 to 5. Note the dot after each number in the declaration of the values. In Fortran, whenever something involves reals you should always have the decimal point. If not there is a chance (depending on the context) that it will be treated as an integer, impacting the results of the code.

A good tactic is to initialize a variable before use in the code. When a variable is declared it has a random numeric value (usually a very small number). For avoiding errors you should always give an initial value to your variable. This can be done either in the code, or during variable declaration:

```
program initialize
implicit none
real :: x=0.
real :: y,z
z = 10.; y = 2.
```

```
x = z + y
end program
```

In the program above we declare 3 real variable, initializing one in the declaration line and the other two in the main program. Note that in the line `z = 10.; y = 2` we have two different statements, separated with a semicolon `(;)`.

2.4.2 Operators

Operators are characters that represent actions. The operators allowed in Fortran are summarized in the following table:

Type	Operator	Associativity
	`**`	right to left
Arithmetic	* or /	left to right
	+ or -	left to right
Relational	<,<=,>,=>,==,/=	none
	`.NOT.`	right to left
	`.AND.`	left to right
Logical	`.OR.`	left to right
	`.EQV.`	left to right
	`.NEQV.`	left to right

It is important to know the priority of operators. The table above is constructed from highest to lowest priority. For example the exponential operator `**` has the highest priority, and is evaluated left to right. This means that `A**B**C` is equal to `A**(B**C)`. When doing math it is advisable to use parentheses to help you avoid confusion.

2.4.3 Comments

Comments are an essential part of every program. Comments are text that the compiler ignores, but are used to give to the programmer useful information about the code.

Comments in Fortran start with the exclamation mark (`!`). They can be full lines, or parts of lines. No code written after an exclamation mark is executed.

```
! The program that follows is a dummy program
program dummy
implicit none
character(10) :: name ! This is an inline comment
! This is a full line comment
name = 'MyName' ! variable MyName will hold my name
end
```

Any code should be highly commented (unless the code is doing something extremely trivial) so that any user can follow what is happening in the program.

2.5 Input/output (I/O) of a program

For input and output in the program we will use the following statements:

- `read(unit#, format, options)`
- `write(unit#, format, options)`
- `print*,`
- `open(UNIT=n,FILE='filename',options)`
- `close(UNIT=n)`

The `read` statement is used to read either from a file or from the keyboard. The `write` statement is used to write either in a file or on the screen. The `print` statement is used for printing on the screen only. The `open` and `close` statements are used to open and close files on the disk.

In the following chapters we will try to explain these statements.

2.5.1 UNIT NUMBER

The UNIT is a number which has an association with a particular device. The device can be the screen or a file. The UNIT is an integer number

Standard Fortran reserves two UNIT numbers for I/O to user. They are:

- `UNIT = 5` for INPUT from the keyboard with the READ statement
- `UNIT = 6` for OUTPUT to the screen with the WRITE statement

Most versions of Fortran will also let you use the asterisk (*) for I/O to the TERMINAL. The asterisk can be used with both the `read` and `write` statements, thus there is no need to remember whether 5 or 6 is for input or output.

When I/O is to a file you must associate a unit number (which you choose) with the FILENAME. Use any unit number other than 5 and 6. On some computers, some unit numbers are reserved for use by the computer operating system.

The association of the unit number and filename occurs when the `open` statement is executed.

```
open(unit=n, file='filename', options...)
```

This statement associates UNIT n with the file mentioned. All subsequent reads or writes using unit n will be to or from this file.

When doing I/O to a file, each read statement inputs data from a NEW LINE and each write statement outputs data on a NEW LINE. Most files are SEQUENTIALLY organized.

When you are finished with the file, you may close it with: close(10) or close(unit=10) (This is optional.)

2.5.2 FORMAT IDENTIFIER

The FORMAT identifier as used in a `write` or `read` statement generally has one of two forms;

- An asterisk (*), indicates LIST-DIRECTED or "free format".
- A LABEL designates a FORMAT statement which specifies the format to use.

For simple I/O, the `*` is used as we have already discussed. When you want to FORMAT your I/O, you will generally use the second method. The FORMAT statement is a list of FORMAT DESCRIPTORS separated by commas which describe what each FIELD of the output should look like. Example:

```
      WRITE(*,10) 'USING', L2, 'AREA =', AREA
 10   FORMAT(1X, A5, 2X, I3, 4X, A6, 2X, F6.2)
```

In a format statement, there is a one-to-one correspondence with the items in the I/O list, with the exception of certain positional descriptors; X, T, /.

The FORMAT statement is defined only once (for each label referenced) in the program but may be used by any number of I/O statements. FORMAT DESCRIPTORS

- The elements in the I/O list must agree in TYPE with the FORMAT DESCRIPTOR being used.
- There are a dozen or so descriptors, The most commonly used are:

```
Descriptor          Use

rIw                 Integer data
rFw.d               Real data in decimal notation
```

```
rEw.d               Real data in scientific notation
Aw                  Character data
'x...x'             Character strings

nX                  Horizontal spacing (skip spaces)
/                   Vertical spacing (skip lines)
Tc                  Tab

Where:
     w  = positive integer specifying FIELD WIDTH
               r  = positive integer REPEAT COUNT
               d  = non-negative integer specifying number of
          digits to display to right of decimal.
               x  = any character
               n  = positive integer specifying number of columns
               c  = positive integer representing column number
```

NOTES ON DESCRIPTORS:

- Values for I, F, and E descriptors will be right justified.
- You must leave enough space for negative sign in the field width for I, E, F, and for decimal point for E, F and for exponent for E.
- When using the E descriptor, the number will be NORMALIZED. This means the number is shifted over so that the first digit is in the tenths position. A leading zero and decimal are always present and a minus sign, if needed. The exponent always requires 4 spaces; E, the sign and 2 digits.
- If you overflow the space allotted to a descriptor, it will print out asterisks (at RUN TIME) in the field space.
- Character data (A) is also right-justified but if it overflows the allotted space it is left-most truncated.
- The first character of each line of output is used to control the vertical spacing. Use 1X to give normal spacing.

2.5.3 I/O with minimal examples

There are two main ways for giving input to a program. The first is the user typing the input in real time, using an input device (keyboard) while the program is running, and the second is giving the input in input-files for the program to open and read.

Equivalently there can be two ways of output of results from the program. One is results printed on and output device (screen, printer) and the second is results written in an output file.

In the following example we ask the user to give input in real time and then print it on the screen:

```
program read_input
implicit none
real :: x
character(20) :: name

read*, name,x

print*, name, x
end
```

When running this program the execution pauses at the `read` statement waiting for the user to give input. When input is given then the execution of the program continues.

Here we used the `read` statement for providing input. There are two ways of calling `read`. The first is as shown in the example above, `read*,` and then the names of the variables the program expects to read. The second would be `read(*,*)` and then the expressions. The first one is specific for reading in free format from the keyboard.

Doing the same thing but reading the input from a file and writing to another file would be:

```
program io
implicit none
real :: x
character(20) :: name
! open the input and output files
open(10,'input')
open(20,'output)
! read the contents of the input file and then close it
read(10,100) name,x
close(10)
! write to the output file and then close it
write(20,101) x,name
close(20)
! declare the format specifications we want for each file
100 format(20x,x,f8.4)
101 format(f8.4,x,20x)
end
```

2.6 Loops in coding and their use

In computer programming, a loop is a sequence of instructions that is continually repeated until a certain condition is reached.

In Fortran there are two main types of loops. The running index loops and the conditional loops

2.6.1 Running index loops

The running index loops execute a sequence of instructions for a specified number of times. The general syntax rules are:

```
[loop_name:] do variable = startValue, stopValue [,stepValue]
    one or more statements
end do [loop_name]
```

Everything that is in brackets in the above is optional. Lets try and explain the syntax of this through an example. We want to create a program that prints on the screen every second number between 10 and 0. The program would look like this:

```
program count
implicit none

integer :: i

counter: do i=10,0,-2
    write(*,*) i
end do counter

end
```

This program will print on the screen the numbers 10,8,6,4,2,0, one in each row, and then it will terminate. Here, `counter` is the name of the loop, `i` is the variable, `10` is the startValue, `0` is the stopValue, `-2` is the stepValue. If no stepValue is declared, then the step is 1. The variable `i` is also the index of the loop.

2.6.2 Conditional loops

There are two main types of conditional loops: the while (or until) loop, and the infinite loop.

The while loop runs while a statement is true, and exits the loop when it is not. The until loop runs until a statement is met, and then it exits. The infinite loop runs forever.

```
program p_while
implicit none

integer :: i

i = 10
do while (i>=0)
    write(*,*) i
    i = i - 2
end do
end
```

The program above does the exact same thing as before, counting from 10 to 0 and printing every second number, but constructed in a while loop.

The same for an until loop would be:

```
program p_until
implicit none

integer :: i

do until(i<0)
    write(*,*) i
    i = i - 2
end do
end
```

2.7 Logical statements

Within a program there might be cases where you have to execute block depending of the truthfulness of a statement. This can be done using `if` statements. The basic construct of an if statement is:

```
if (argument) code to be run
```

The argument has to be able to be answered with either true, or false. If the answer to the argument is false, then the `code to be run` part is not executed.

```
program print5
implicit none

integer :: i

do i=1,10
    if (i.eq.5) print*, i
```

```
end do
end
```

The program above will loop through the numbers 1 to 10, and will only print number 5. There can be more than one conditions in the same if statement. For example :

```
if (a.eq.3 .and. b.eq.5) do things

!or

if (a.eq.3 .or. a.eq.7) do things
```

In the dummy examples above, the first statement will run only if both arguments are true. The second will run if any of the arguments are true.

When we need to run different code depending on the value of a variable, we can use `block if` statements. The syntax is:

```
if (logical expression 1) then
    statements 1
else if (logical expression 2) then
    statements 2
else if (logical expresion 3) then
    statements 3
else
    statements else
end if
```

Lets try to explain this through an example:

```
program grades
implicit none

real :: grade

print*, 'Please give the percentage grade of the student:'
read*, grade

if (grade .lt. 50) then
    print*, 'The student failed... Grade: F'
else if (grade .ge. 50 .and. grade .lt. 65) then
    print*, 'Final grade: E'
else if (grade .ge. 65 .and. grade .lt. 75) then
    print*, 'Final grade: D'
else if (grade .ge. 75 .and. grade .lt. 85) then
    print*, 'Final grade: C'
else if (grade .ge. 85 .and. grade .lt. 95) then
    print*, 'Final grade: B'
else if (grade .ge. 95 .and. grade .lt. 100) then
    print*, 'Final grade: A'
else:
    print*, 'This was close to excelent! Final grade: A+'
end if

end
```

The program above gives a grade between A and F based on the percentage score of the student.

An `if...then...else if` block must always have an `else...end if`!

2.8 Intrinsic functions

Fortran provides many commonly used functions, called intrinsic functions. To use an intrinsic function you have to know:

- The name of the function
- The number of arguments the function needs
- The types of arguments
- The type of the return value of the function

For example, for the intrinsic function `log(x)`, which returns the natural logarithm of x, the argument type is `real` and the return type is `real`. This means that we have to feed the function with a `real` number and the result of the function is again a `real` number.

Some of the intrinsic functions of Fortran are:

- Mathematical functions
  - `abs(x)`: absolute value of x
  - `sqrt(x)`: square root of x
  - `sin(x)`: sine of x radian
  - `cos(x)`: cosine of x radian
  - `tan(x)`: tangent of x radian
  - `asin(x)`: arcsine of x
  - `acos(x)`:arccosine of x
  - `atan(x)`: arc tangent of x
  - `exp(x)`: exp(x)
  - `log(x)`: natural logarithm of x
  - `log10(x)`: logarithm base 10 of x
- Conversion functions:
  - `int(x)`: integer part of x
  - `nint(x)`: nearest integer of x
  - `floor(x)`: greatest integer less or equal to x
  - `ceiling(x)`: smallest integer greater or equal to x
  - `fraction(x)`: the fractional part of x
  - `real(x)`: convert x to real
- Other functions:
  - `max(x1,x2,...,xn)`: maximum of x1,x2,...,xn
  - `min(x1,x2,...,xn)`: minimum of x1,x2,...,xn
  - `mod(x,y)`: remainder `x-int(x/y)*y`

2.9 Subroutines, functions

Subroutines and functions are blocks of code that perform some operation on the input variables. Subroutines do not return a result, where functions do. We use functions and subroutines for code that is often re-used at several places.

A function is located at the end of the program, just before the `end` statement and after the `contains` statement.

```
program main
implicit none
real :: a,b
.
.
```

```
.
total = add(a,b)
.
.
.
contains
function add(x,y)
implicit none
real :: add,x,y
add = x + y
end function

end program
```

The function in the example above adds the two numbers given as arguments. To get the result from a function we have to assign it

Subroutines also perform operations on the input variables, but they can also change them. A subroutine can give back several results through its arguments. It is invoked with a `call` statement. For example:

```
subroutine square_cube(i,isquare,icube)
  integer, intent(in)  :: i              ! input
  integer, intent(out) :: isquare,icube ! output
  isquare = i**2
  icube   = i**3
end subroutine square_cube

program xx
  implicit none
  integer :: i,isq,icub
  i = 4
  call square_cube(i,isq,icub)
  print*,"i,i^2,i^3=",i,isq,icub
end program xx
```

The `intent` statements in the program above are used in functions and subroutines for variables that need to be passed in or out.

Appendix

A.1 Basic Linux commands

The terminal is a basic application in all Linux systems. You will be using a virtual terminal with a Linux shell for the purpose of this course. A shell is a command-line interpreter that provides a command line user interface for the operating system. There are many different shells that a user can choose from, with one of the most popular ones being the `BASH shell`.

Through the terminal the user communicates with the system through commands. The basic commands you will need to get started with the terminal are the following:

Command	Description
`cd`	[**c**hange **d**irectory] Used to navigate in the linux terminal e.g. `cd foldername` will get you in the folder `fodlername`
`cal`	[**cal**endar] shows current month's calendar
`cal` *year*	shows the calendar of the current year
`cat`	[con**cat**enate] prints on the screen the contents of a file. E.g. `cat filename` will print on the screen the contents of file `filename`
`cat -v`	also show the non printable characters in the file
`chgrp`	[**ch**ange **gr**ou**p**] changes the group of ownership of the file.
`chown`	[**ch**ange **own**er] changes the owner of the file.
`chmod`	[**ch**ange **mod**e] changes the permissions of the file.
`cmp`	[**c**o**mp**are] compares two files and shows the location of the first difference it finds.
`cp`	[**c**o**p**y] used to copy a file
`cp -r`	used to copy a folder
`mv`	[**m**o**v**e] used to move or rename a file or folder
`find`	looks in the disk for files that fulfill the criteria we provide
`grep`	[**g**lobal **r**egular **e**xpression **p**rint] looks in the files for a string
`logname`	prints on the screen the username of the current user
`ls`	[**ls**t] lists all the contents of the current folder
`ls -a`	[**ls**t **a**ll] lists all the contents of the current, including the hidden files and folders.
`ls -l`	[**ls**t **l**ong list] lists all the contents of the current folder with information about ownership, permissions, size, etc
`ls -la`	combination of the previous two
`passwd`	changes the password of the current user
`ps`	[**p**roces**s**] prints on the screen the running processes
`pwd`	[**p**rint **w**orking **d**irectory] prints on the screen the path to the current directory.
`rm`	[**r**e**m**ove] deletes a file
`rm -r`	deletes a file or folder
`tail`	shows on screen the last lines of a file
`tail -f`	shows on screen the last lines of a file, tracking the changes
`head`	prints on the screen the first lines of a file (header)
`logout`	self explanatory
`clear`	clears the screen from all previous commands and/or messages
`man`	[**man**ual] shows the manual page of a command
`mkdir`	[**m**a**k**e **dir**ectory] creates a folder

Of course this list is far from complete, but it should suffice for the introducing you to the Linux terminal.

A.2 Basic VI usage

As mentioned before, knowing how to use a text editor in the terminal can be really useful. Here I will try and give you an introduction to the vi editor.

vi is a modal editor. This means that it has several "modes". There are two primary modes: *insert mode* where you type text into the editor and it is committed to the document, and *normal mode* where you enter arguments via the keyboard that perform a variety of functions.

In order to start editing a file (new or existing) in vi you just have to write in the command line:

```
daskalakis@lamosnb2 ~$ vi filename
```

vi always starts in normal mode. This means that you cannot start typing into your file right away.

Following are some of the commands vi can accept when in normal mode:

Cursor movement commands:

Command	Description
`h`	move one character left
`j`	move one character right
`k`	move one line down
`l`	move one line up
`[Enter]`	move to the beginning of the next line
`$`	move to the last column of the current line
`0`	move to the first column of the current line
`^`	move to the first non-empty column of the current line
`w`	move to the beginning of the next word or punctuation mark
`W`	move to the column after the next empty character
`b`	move to the beginning of the previous word
`B`	move to the beginning of the previous word ignoring punctuation marks
`e`	move to the end of the next word
`E`	move to the end of the next word ignoring punctuation marks
`H`	move to the top of the screen
`M`	move to the middle of the screen
`L`	move to the bottom of the screen

Screen movement commands:

Command	Description
`G`	move to the last line of the document
`#G`	move to line number #
`z+`	move so that the line currently at the bottom of the screen goes to the top of the screen
`z`	move current line to the middle of the screen
`z-`	move current line to the bottom of the screen
`Ctrl-f`	move one screen down
`Ctrl-b`	move one screen up
`Ctrl-d`	move half a screen down
`Ctrl-u`	move half a screen up

Text insert commands:

All the following commands allow to enter *insert mode* and edit the document. When in insert mode,

at the bottom left of the screen you can see the `-- INSERT --` indication, which means you are editing the document. To return to the *normal mode* hit the `Esc` key.

Command	Description
`r`	replace the character under the cursor with the next character you hit on the keyboard
`i`	start inserting text where the cursor is
`a`	insert text starting from the next column after the cursor
`o`	insert a new line below the current line and start inserting text at the beginning of that line
`R`	replace text until the `Esc` key is pressed
`I`	insert text starting from the first non-blank column of the line. If there is no text in the line, insert at the beginning of the line
`A`	insert text starting at the end of the current line
`O`	insert a new line above the current line and insert at the beginning of that line

Commands for deleting text:

Command	Description
`x`	delete the current character
`dd`	delete the current line
`dw`	delete the current word, starting from the position of the cursor
`db`	delete from the position of the cursor until the beginning of the current word
`D`	delete from the cursor up to the end of the current line

All the above commands save whatever was last deleted in a buffer, and can be used (pasted) again in the document.

Commands for copying and pasting:

Command	Description
`yy`	copy the current line
`p`	paste after the cursor
`P`	paste before the cursor

Search commands:

Command	Description
`/`	search for text forward (case sensitive)
`?`	search for text backwards (case sensitive)
`f`	search for a character forward in the current line
`F`	search for a character backwards in the current line

Other commands:

Command	Description
`.`	redo the last command
`u`	undo
`ctrl-r`	redo (after undo)
`ZZ`	save and quit
`ZQ`	discard changes and quit

When in *normal mode*, you can also use a variety of commands after hitting colon `(:)`. Some of these commands are:

Command	Description
`:w`	save
`:x`	save and quit
`:wq`	save and quit
`:q`	quit
`:q!`	quit discarding changes
`:#`	go to line number #
`:$`	go to the end of the document

A.3 Basic PICO usage

PICO or NANO is a very simple, very basic text editor for use in the Linux terminal. It is the equivalent of notepad for Windows. It offers no syntax highlighting or high level editing, but can be used for easy and fast modification of files.

For opening a file with pico you type:

```
pico <filename>
```

The application will open and you can directly start modifying the contents of the file. At the bottom of the screen you can see the options available. For exiting the application you type `Ctrl-X`. For saving you type `Ctrl-O`. It offers a basic search function with `Ctrl-W`. For saving and exiting you type `Ctrl-OX`.

If you find yourselves struggling with vi, you can use pico for its ease of use.