E-LERU Project 3

Computational Condensed Matter Physics

Mebarek Alouani
Jörg Baschnagel
Torsten Kreer

UFR de Sciences Physiques
ULP Strasbourg

August 31, 2009

Contents

Chapter I

Introduction

I.1 Computation and Physics

Often physicists use analytical methods to solve physics problems. For that many approximations are needed to simplify the mathematical equations so that an analytical solution is possible. The simplification of the equations need to be justified, which reduces the generality of the theory. The dilemma is then to find the appropriate approximations that will allow the physicist to solve the equations and maintain the generality of the theory.

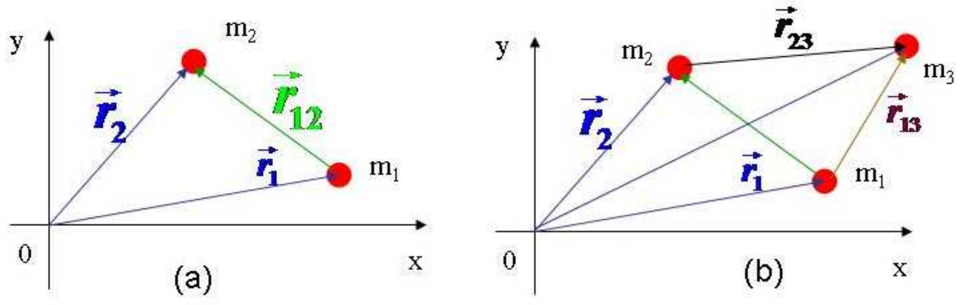


Figure I.1: Interaction of two (left plot) and three (right plot) classical particles.

First example (Newton universal attraction). Case of two interacting classical particles (see Fig. I.1):

$$\vec{F}_{1\leftarrow 2} = m_1 \frac{\mathrm{d}^2\vec{r}_1}{\mathrm{d}t^2} = K\frac{m_1 m_2}{r_{12}^2}\hat{r}_{12}$$
$$\vec{F}_{2\leftarrow 1} = m_2 \frac{\mathrm{d}^2\vec{r}_2}{\mathrm{d}t^2} = -K\frac{m_1 m_2}{r_{12}^2}\hat{r}_{12}$$

with $\hat{r} = \vec{r}/r$. Using the initial conditions, these equations can be solved analytically (see Berkeley mechanics). However, there is no analytical solution to the problem of three or more interacting particles whose trajectories are given by the following differential equations:

$$\begin{aligned}
\vec{F}_{1\leftarrow 2,3} &= m_1 \frac{\mathrm{d}^2 \vec{r}_1}{\mathrm{d}t^2} = K \frac{m_1 m_2}{r_{12}^2} \hat{r}_{12} + K \frac{m_1 m_3}{r_{13}^2} \hat{r}_{13} \\
\vec{F}_{2\leftarrow 1,3} &= m_2 \frac{\mathrm{d}^2 \vec{r}_2}{\mathrm{d}t^2} = -K \frac{m_1 m_2}{r_{12}^2} \hat{r}_{12} + K \frac{m_2 m_3}{r_{23}^2} \hat{r}_{23} \\
\vec{F}_{3\leftarrow 1,2} &= m_3 \frac{\mathrm{d}^2 \vec{r}_2}{\mathrm{d}t^2} = -K \frac{m_1 m_3}{r_{13}^2} \hat{r}_{13} - K \frac{m_2 m_3}{r_{23}^2} \hat{r}_{23}
\end{aligned}$$

One then has to either simplify the problem by considering that the influence of the third mass is negligible, and add its contribution as a perturbation, or solve the problem using computational methods. The only approximations for the latter method are these of the numerical methods or the limitation of the computer.

Second example. There is no analytical solution for the Schrödinger equation,

$$\left(-\frac{\hbar^2}{2m}\Delta + V(\vec{r})\right)\psi(\vec{r}) = E\psi(\vec{r}) \ ,$$

for a realistic potential $V(r)$ except for that of the hydrogen atom or the He+ ion. One has then to construct a numerical program that can be used for any system and boundary conditions (see the section concerning the solution of differential equations).

I.2 A brief history about computers

In the sixteen century, John Neper (1550-1617) published the logarithmic table up to 8 digits. By writing

$$\begin{aligned}
\ln(ab) &= \ln\,(a) + \ln\,(b) \\
\ln\left(a^{\frac{1}{n}}\right) &= \frac{\ln(a)}{n} \ ,
\end{aligned}$$

we can transform a multiplication of two numbers or the nth square root to an addition of logarithms or division of a logarithm by n, respectively. We have then just to look up the numbers in the table associated with the logarithms. At the beginning of the 19th century, the production of logarithmic tables and other tables such as trigonometric tables became a big industry.

In the nineteen century, Charles Babbage (1792-1871) understood that the elementary operations can be performed by a machine. He used the binary code and the punched cards to construct the analytical engine. The project was stopped because it was too expensive. Later, Herman Hollerith (1859-1929) made the first counting machine based on reading punched cards. The American bureau of census, employed more than thousand people for about seven years to enter into several machines 21000 pages of documents! In 1890, the first census was made after several weeks of computation and the united states population was found to be about 62 622 250 inhabitants. In 1896, Hollerith started the Tabulating Machine Company, which became in 1924 the International Business Machines (IBM).

The first true computer was constructed by John Vincent Atanasoff, a physicist for Iowa State University. At this stage, the electronic digital computer replaced the electro-mechanics by vacuum tubes. The ENIAC (Electronic Numerical Integrator and Computer) was constructed by John Mauchly and J. Presper Echert just after the second world war.

The ENIAC was able to make 5000 additions and 400 multiplications per second. But, it was very expensive to maintain, because it consumed 150kW and needed a very costly ventilation system for the heat dissipation. The machine weighted 30 tons and occupied 16000 m^2. In 1947, Metropolis and Frankel used the ENIAC computer to solve numerically the droplet model in order to study the nuclear fission.

In the beginning of the 1950 MANIAC-I (Mathematical analyzer, Numerator, Integrator and Computer) was constructed, and was very similar to ENIAC. The first computer based on the transistor technology was the CDC 1604 made by Control Data Corporation of Seymour Cray. The first transistor machine made by IBM in 1960 and was named IBM 7090. Later, IBM constructed and sold twelve thousand IBM 1401. The frame of the computer was blue, this is which earned IBM the Big Blue name.

Time sharing computers started at MIT in 1961. Before that there were only the BATCH queue. The submitted jobs have to wait in the queue. Even the smallest job has to wait for the big jobs in the queue to finish before it starts running. The time sharing computer allows many users to login to the machine and gives the illusion to every user that he is alone on the machine.

In 1965, ARPA (Advanced Research Project Agency) of the Pentagon financed the project MULTICS (MULTIplexed Information Computing Service) which allows more than few hundred users to connect simultaneously to a single computer and the machine can handle about a thousand users. From 1969 to 1974, Ken Thompson and Dennis M. Ritchie developed the UNIX operating system. A simplified version was used in the PDP-7 of DEC. UNIX was written in the C language invented by Ritchie; it is now the most used operating system, thanks to Linus Torvalds who created in 1991, a gnu version named LINUX. This is the operating system that we will use in this course together with the Fortran language to transform algorithms into programs that can be compiled to run in any modern computer under UNIX or Linux operating system.

I.3 Introduction to the UNIX Operating System

I.3.1 Directories

File and directory paths in UNIX use the forward slash / to separate directory names in a path.

Examples (see Fig. I.2 for illustration):

- **/** is the root directory
- **/home** is the directory home of all users
- **/home/user1** user1 is the home directory of user1 and is a subdirectory of /home

Here you will find the basic commands to move around in a Unix environment:

- **pwd** show the present working directory, or current directory
- **cd** change current directory to your HOME directory
- **cd directory1/subdirectory1** change current directory to the subdirectory subdirectory1 which is a subdirectory of directory1
- **cd ..** change current directory (subdirectory1) to the parent directory (directory1) of the current directory.

Listing directory contents:

- **ls** list a directory
- **ls -l** list a directory in long (detailed) format; see Fig. I.3 for an illustration.

To change the file permissions and attributes we can use the chmod command as follow:

- **chmod 755 file** changes the permissions of file to be rwx for the owner, and rx for the group and the world (7 = rwx = 111 binary. 5 = r-x = 101 binary)

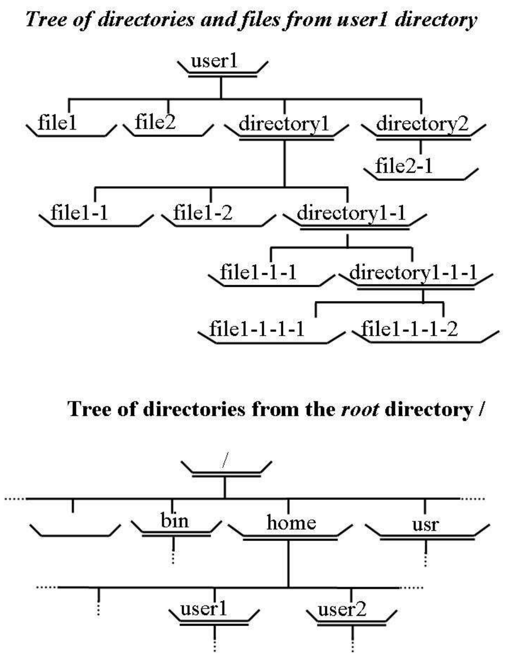


Figure I.2: Upper panel: Tree of directories and files from *user1* directory. Lower panel: Tree of directories from the *root* directory.

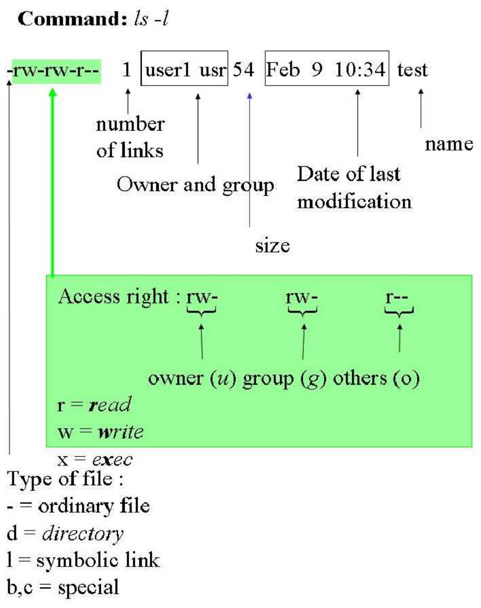


Figure I.3: Illustration of the *ls* command.

- **chgrp user file** makes file belong to the group user
- **chown user2 file** makes user2 the owner of file
- **chown -R user2 dir** makes user2 the owner of dir and everything in its directory tree

You must be the owner of the file/directory or be root before you can do any of these things.

I.3.2 Moving, Renaming, and Copying Files

- **cp file1 file2** copy a file named file1 into file2
- **mv file1 file2** move or rename a file named file1 into file2
- **rm file1 file2 ...** remove or delete a file named file1 file2 ...
- **rm -r dir1 dir2...** recursively removes a directory and its contents BE CAREFUL!
- **mkdir dir1 dir2...** makes directories named dir1, dir2 ...
- **rmdir dir1 dir2...** removes empty directories dir1, dir2, . . .

To view or edit files one can use one the following commands:

- **cat filename** dumps a file to the screen in ascii.
- **more filename** progressively dumps a file to the screen: ENTER = one line down SPACEBAR = page down q=quit
- **less filename** like more, but you can use Page-Up too. Not on all systems.
- **vi filename** edits a file using the vi editor. All UNIX systems have vi in some form.
- **emacs filename** edits a file using the emacs editor. Not all systems have emacs.
- **head filename** shows the first few lines of a file.
- **head -n filename** shows the first n lines of a file.
- **tail filename** shows the last few lines of a file.
- t**ail -n filename** shows the last n lines of a file.

I.3.3 Shells

The behavior of the command line interface will differ slightly depending on the **shell** program that is being used.

Depending on the shell used, some extra behaviors can be quite nifty.

You can find out what shell you are using by the command:

printenv SHELL

You can create a file with a list of shell commands and execute it like a program to perform a task. This is called a shell script. This is in fact the primary purpose of most shells, not the interactive command line behavior.

Environment Variables

You can teach your shell to remember things for later using environment variables. For example under the shell bash:

export rootbin=/usr/local/bin defines the variable rootbin with the value /usr/local/bin

cd $rootbin changes your present working directory to the value of rootbin

printenv rootbin will print out the value of rootbin, or /usr/local/bin

echo $rootbin will also print the value of rootbin

Interactive History

A feature of bash and tcsh (and sometimes others) you can use the up-arrow keys to access your previous commands, edit them, and re-execute them.

Filename Completion

A feature of bash and tcsh (and possibly others) you can use the TAB key to complete a partially typed filename. For example if you have a file called constantine-monks-and-willy-wonka.txt in your directory and want to edit it you can type 'vi const', hit the TAB key, and the shell will fill in the rest of the name for you (provided the completion is unique).

Bash will even complete the name of commands and environment variables. And if there are multiple completions, if you hit TAB twice bash will show you all the completions. Bash is the default user shell for most Linux systems.

Redirection

grep string filename > **newfile** redirects the output of the above grep command to a file 'newfile'

grep string filename >> **existfile** appends the output of the grep command to the end of 'existfile'

The redirection directives, > and >>can be used on the output of most commands to direct their output to a file.

Pipes

The pipe symbol || is used to direct the output of one command to the input of another.

For example:

ls -l || **more** This commands takes the output of the long format directory list command *ls -l* and pipes it through the more command (also known as a filter). In this case a very long list of files can be viewed a page at a time.

Command Substitution

You can use the output of one command as an input to another command in another way called command substitution. Command substitution is invoked when by enclosing the substituted command in backwards single quotes. For example:

cat ‘find . -name myfile.tex‘

The cat command dumps to the screen all the files named myfile.tex that exist in the current directory or in any subdirectory tree.

Grep

Searching for strings in files using the **grep** command

grep string filename prints all the lines in a file that contain the string

Find

Searching for files using the **find** command

find search_path -name filename

find . -name aaa.txt finds all the files named aaa.txt in the current directory or any subdirectory tree

find / -name vimrc finds all the files named ’vimrc’ anywhere on the system

find /usr/local/bin -name ’*.run* finds all files whose names contain the string ’.run’ which exist within the ’/usr/local/bin’ directory tree

I.3.4 Tar

Reading and writing tapes, backups, and archives using the **tar** command

The tar command stands for ’tape archive’. It is the standard way to read and write archives (collections of files and whole directory trees). Often you will find archives of stuff with names like stuff.tar, or stuff.tar.gz. This is stuff in a tar archive, and stuff in a tar archive which has been compressed using the gzip compression program respectively.

Chances are that if someone gives you a tape written on a UNIX system, it will be in tar format, and you will use tar (and your tape drive) to read it. Likewise, if you want to write a tape to give to someone else, you should probably use tar as well.

Tar examples:

- **tar xv** extracts (x) files from the default tape drive while listing (v = verbose) the file names to the screen
- **tar tv** lists the files from the default tape device without extracting them
- **tar cv file1 file2** writes files ’file1’ and ’file2’ to the default tape device
- **tar cvf archive.tar file1 file2...** creates a tar archive as a file archive.tar containing file1, file2...etc.
- **tar xvf archive.tar** extracts from the archive file

- **tar cvfz archive.tar.gz dname** creates a gzip compressed tar archive containing everything in the directory 'dname' (does not work with all versions of tar)
- **tar xvfz archive.tar.gz** extracts a gzip compressed tar archive (does not work with all versions of tar)
- **tar cvfI archive.tar.bz2 dname** creates a bz2 compressed tar archive (does not work with all versions of tar)

Compression

File compression can be made by one of the following commands: **compress**, **gzip**, **bzip2**.

The standard UNIX compression commands are compress and uncompress. Compressed files have a suffix .Z added to their name. For example:

- **compress myfile** creates a compressed file myfile.Z
- **uncompress myfile** uncompresses *myfile* from the compressed file *myfile.Z*. Note the **.Z** is not required.

Another common compression utility is gzip (and gunzip). These are the GNU compress and uncompress utilities. gzip usually gives better compression than standard compress, but may not be installed on all systems. The suffix for gzipped files is **.gz**

- **gzip myfile** creates a compressed file myfile.gz
- **gunzip myfile** extracts the original file from myfile.gz

The bzip2 utility has (in general) even better compression than gzip, but at the cost of longer times to compress and uncompress the files. It is not as common a utility as gzip, but is becoming more generally available.

- **bzip2 myfile** creates a compressed Iges file myfile.bz2
- **bunzip2 myfile.bz2** uncompresses the compressed myfile file.

I.3.5 Man

To look for help use the **man** command. Most of the commands have a manual page which give sometimes useful, often more or less detailed, sometimes cryptic and unfathomable descriptions of their usage. Some say they are called man pages because they are only for real men.

Example:

man ls	Shows the manual page for the ls command

I.3.6 Basics of the VI Editor

To open a file use

vi filename

To start typing text type one of the following letter where you want to insert text

i	inserts before current cursor position
I	inserts at beginning of current line
a	inserts (appends) after current cursor position
A	appends to end of line
r	replaces 1 character
R	replaces mode
<**ESC**> **key**	terminates insertion or overwrites mode

Deletion of text:

x	deletes single character
dd	deletes current line and put in buffer
ndd	deletes **n** lines (**n** is a number) and put them in buffer
J	attaches the next line to the end of the current line (deletes carriage return)

To undo the last typing just first press the $< ESC >$ key and then type u

u	undo last command

To cut and paste:

yy	yanks current line into buffer
nyy	yanks n lines into buffer
p	puts the contents of the buffer after the current line
P	puts the contents of the buffer before the current line

Cursor positioning:

Cntl d	pages down
Cntl u	pages up
:n	positions cursor at line n
:$	positions cursor at end of file
Cntl g	displays current line number
h,j,k,l	move the cursor Left, Down, Up, and Right respectively (your arrow keys should also work if your keyboard mappings are anywhere near sane)

String substitution:

- **:n1,n2s/string1/string2/g** Substitute string2 for string1 on lines n1 to n2. If g is included (global), all instances of string1 on each line are substituted. If g is not included, only the first instance per line is substituted.

wedge matches start of line

. matches any single character

$ matches end of line

These and other special characters (like the forward slash) can be escaped with \ i.e to match the string **/usr/STRIM100/SOFT** say \**/usr\/STRIM100\/SOFT**

Examples:

- **:1,$s/dog/cat/g** substitutes 'cat' for 'dog', every instance for the entire file - lines 1 to $ (end of file)
- **:23,25s/frog/bird/** substitutes 'bird' for 'frog' on lines 23 through 25 (only the first instance on each line is substituted)

Saving and quitting and other ex commands:

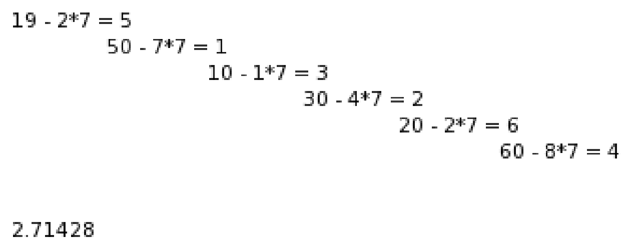
These commands are all prefixed by pressing colon (:) and then entered in the lower left corner of the window. You cannot enter a ex command when you are in an edit mode. Press <**ESC**> to exit from an editing mode.

- **:w** writes the current file
- **:w new.file** writes the file to the name 'new.file'
- **:w! existing.file** overwrites an existing file with the file currently being edited
- **:wq** writes the file and quit
- **:q** quit
- **:q!** quits with no changes
- **:e filename** opens the file 'filename' for editing
- **:set number** turns on line numbering
- **:set nonumber** turns off line numbering

I.4 Introduction to Algorithms

An algorithm is a way of reducing a more or less difficult task to a sequence of simpler steps. In computation, the simpler steps are the four operations of arithmetic. As an example, let's consider the case of dividing a number by another one, using the method of successive subtraction. To be more specific, let's take two numbers like 19 and 7. The task here is to perform the more difficult operation of division as sequence of simpler steps. Here are the necessary steps of this simple algorithm:

1. We will start subtracting 7s until less than 7 remains.



```
19 - 2*7 = 5
        50 - 7*7 = 1
                10 - 1*7 = 3
                        30 - 4*7 = 2
                                20 - 2*7 = 6
                                        60 - 8*7 = 4


2.71428
```

Figure I.4: Simple algorithm for dividing two numbers using successive subtractions.

2. At the same time we count the number of subtractions made and write the number down. In our example, we find 2 subtractions.
3. We put a point after the number of subtractions to separate the integer part from the decimal one.
4. We attach zero to the remainder (here 5 becomes 50).
5. We repeat steps 1, 2, and 4 as many times until the desired numerical precision is reached.

In Fig. I.4 we illustrate the important part of the subtraction for our example.

I.5 Programming Languages

Programming languages must be:

- totally unambiguous (unlike natural languages, for example, English),
- expressive – it must be fairly easy to program common tasks,
- practical – it must be an easy language for the compiler to translate,
- simple to use.

All programming languages have a very precise syntax (or grammar). This ensures all syntactically-correct programs have a single meaning.

High-level programming languages. Assembler code is a Low-Level Language. Fortran 90, Fortran 77, ADA, C and Java are High-Level Languages.

- a program is a series of instructions to the CPU,
- could write all programs in assembler code but this is a slow, complex and error-prone process,
- high-level languages are more expressive, more secure and quicker to use,
  the high-level program is compiled (translated) into assembler code by a compiler.

An example problem. To convert from °F (Fahrenheit) to °C (Centigrade) we can use the following formula:

$$T_{\mathrm{C}} = 5(T_{\mathrm{F}} - 32)/9 \; .$$

To convert from °C to K (Kelvin) we add 273. The program would accept a Fahrenheit temperature as input and produce the Centigrade and Kelvin equivalent as output.

An example program.

```
PROGRAM Temp_Conversion
   IMPLICIT NONE
   INTEGER :: T_F, T_C, T_K
   PRINT*, "Please type in the temp in F"
   READ*, T_F
   T_C = 5*(T_F - 32)/9.0
   PRINT*, "This is equal to", T_C, "C"
   T_K = T_C + 273
   PRINT*, "or", T_K, "K"
END PROGRAM Temp_Conversion
```

This program, called Temp.f90, can be compiled:

```
mea@smalley> f90 Temp.f90 -o Temp.x
```

and run:

```
mea@smalley> ./Temp.x
 Please type in the temp in F
45
 This is equal to 7 C
 or  280 K
```

Analysis of program. The code is delimited by **PROGRAM** ... **END PROGRAM** statements. Between these there are two distinct areas:

1. Specification Part
   - Use of **IMPLICIT NONE** instruction telling the compiler that all variables must be declared. This is very helpful because it will insure rigorous typing of the variables. It must be always used if one would like to write good programs.
   - Specifies named memory locations (variables) for use. Here we declare three memory locations **T_C**, **T_F**, and **T_K**.
   - Specifies the type of the variable. Here the three variables are of **INTEGER** type.
2. Execution Part
   - Asks for data using the **PRINT** statement. It prints strings of characters on the screen.
   - Reads in data using the **READ** command.
   - Calculates the temperature in °C and K.
   - Prints out results using again the **PRINT** statement.

A closer look at the specification part.

- **IMPLICIT NONE** – this should always be present. Means all variables must be declared.
- **INTEGER** :: **T_F**, **T_C**, **T_K** – declares three **INTEGER** (whole number) variables.

Other variable types:

- **REAL** – real numbers, e.g., 3.1459, π
- **LOGICAL** – take values .TRUE. or .FALSE.
- **CHARACTER** – contains single alphanumeric character, e.g., 'a'
- **CHARACTER(LEN=12)** – contains 12 alphanumeric characters, a string

Fortran 90 is not case sensitive. **PI** is the same as **Pi** or **pi** and **INTEGER** is the same as **integer**. As a good rule of thumb, it is best if all Fortran statements are written in capital letters and all variables are written in small letters. This makes the program easily readable.

A closer look at the execution part. This is the part of the program that does the actual 'work'.

PRINT, "Please type in the temp in F" – writes the string to the screen
READ*, T_F – reads a value from the keyboard and assigns it to the **INTEGER** variable **T_F**
T_C = 5*(T_F-32)/9.0 – the expression on the right hand side is evaluated and assigned to the INTEGER variable **T_C**

*	is the multiplication operator,
-	is the subtraction operator,
/	is the division operator (takes longer than *),
=	is the assignment operator.

PRINT*, "This is equal to", T_C, "C" – displays a string on the screen followed by the value of a variable (**T_C**) followed by a second string ("C"). By default, input is from the keyboard and output to the screen.

How to write a computer program. There are 4 main steps:

(i) specify the problem,

(ii) analyze and break down into a series of steps towards solution,

(iii) write the Fortran 90 code,

(iv) compile and run (i.e., test the program).

It may be necessary to iterate between steps iii and iv in order to remove any mistakes. The testing phase is very important.

An algorithm of a quadratic equation solver. Write a program to calculate the roots of a quadratic equation of the form:

$$ax^2 + bx + c = 0 \; . \tag{I.1}$$

The roots are given by the following formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \; . \tag{I.2}$$

The algorithm:

(i) READ values of **a**, **b** and **c**,

(ii) if **a** is zero then stop as we do not have a quadratic equation,

(iii) calculate value of discriminant $\delta = b^2 - 4ac$,

(iv) if δ is zero then there is one root: $x = -b/(2a)$,

(v) if δ is > 0 then there are two real roots: $x = \dfrac{-b - \sqrt{\delta}}{2a}$ and $x = \dfrac{-b + \sqrt{\delta}}{2a}$,

(vi) if δ is < 0 there are two complex roots: $x = \dfrac{-b - \mathrm{i}\sqrt{-\delta}}{2a}$ and $x = \dfrac{-b + \mathrm{i}\sqrt{-\delta}}{2a}$,

(vii) **PRINT** the solutions.

A Program of the quadratic equation solver

```
PROGRAM qe_solver
 IMPLICIT NONE
  INTEGER :: a, b, c, delta
  REAL    :: real_part, imag_part
  PRINT*, "Type in integer values for a, b and c"
  READ*, a, b, c
  IF (a /= 0) THEN
      ! Calculate discriminant
      delta = b*b - 4*a*c
      IF (delta == 0) THEN                  ! one root
         PRINT*, "Root is ", -b/(2.0*a)
      ELSE IF (delta > 0) THEN              ! real roots
         PRINT*, "Roots are",(-b - SQRT(REAL(delta))) /( 2.0 * a),&
            "and",      (-b + SQRT(REAL(delta))) / (2.0 * a)
       ELSE                              ! complex roots
         real_part = -b / (2.0 * a)
         ! delta < 0 so must take SQRT of -delta
         imag_part = (SQRT(REAL(-delta))/(2.0 * a))
         PRINT*, "1st Root", real_Part, "-", imag_part, "i"
         PRINT*, "2nd Root", real_Part, "+", imag_part, "i"
       END IF
  ELSE                              ! a == 0
      PRINT*, "Not a quadratic equation"
  END IF
END PROGRAM qe_solver
```

The testing phase of the quadratic equation solver. After calling the program qe_solver.f90 and compiling it, we obtain the executable name qe_solver.x by invoking the following command:

```
mea@smalley> ifort qe_solver.f90 -o qe_solver.x
```

Notice that switches or flags can be supplied to the ifort compiler: -o qe_solver.x: gives executable a different name, for example, qe_solver.x. It is nice to always use **.x** at the end so that you know that the file is an e**x**ecutable. To learn more about the Intel Fortran compiler type:
man ifort

The output from the program is as follows:

```
 mea@smalley> ./qe_solver.x
 Type in values for a, b and c
1 -3 2
 Roots are   2.0000000 and   1.0000000
mea@smalleyuxa{adamm} 36> a.out
 Type in values for a, b and c
1 -2 1
 Root is    1.0000000
mea@smalley> ./qe_solver.x
 Type in values for a, b and c
1 1 1
 1st Root  -0.5000000 +   0.8660254 i
 2nd Root  -0.5000000 -   0.8660254 i
mea@smalley> ./qe_solver.x
 Type in values for a, b and c
0 2 3
 Not a quadratic equation
```

It can be seen that the 'test data' used above exercises every line of the program. This is important in demonstrating correctness.

Points raised. The previous program introduces some new Fortran constructs,

- **comments** – anything on a line following a ! is ignored,
- **&** – means the line is continued,
- **IF construct** – different lines are executed depending on the value of the Boolean expression,
- **relational operators** – **==** ('is equal to') or > ('is greater than'),
- **nested constructs** – one control construct can be located inside another.
- **procedure call** – **SQRT**(x) returns square root of x.
- **type conversion** – in above call, x must be REAL. In the program, **delta** is **INTEGER**, **REAL(delta)** converts **delta** to be real valued.

To save CPU time we only calculate the discriminant, **delta**, once in the case where **a** is not zero.

Bugs - Compile-time errors. In the previous program, if we accidentially typed:

```
      dleta = b*b -4*a*c
```

The compiler generates a compile-time or syntax error:

```
mea@smalley> ifort qe_solver.f90 -o qe_solver.x
fortcom: Error: qe_solver.f90, line 9: This name does not have a type,
      and must have an explicit type.   [DLETA]
      dleta = b*b - 4*a*c
------^
compilation aborted for qe_solver.f90 (code 1)
```

Bugs - Run-time errors. If we had typed:

real_part = -b/(.0*a)

Then the program would compile but we would get a run-time error,

```
mea@smalley> ./qe_solver.x
 Type in integer values for a, b and c
1 1 1
 1st Root -Infinity      -  0.8660254     i
 2nd Root -Infinity      +  0.8660254     i
```

It is also possible to write a program that gives the wrong results!

I.5.1 Introduction to Fortran

This course of Fortran 90 is adapted from the course given at the University of Liverpool by A. C. Marshall. Licence was granted by Marshall to teach or modify the content of his course for non commercial use.

History

- FORTRAN stands for FORmula TRANslation invented at IBM by Backus in the late fifties. It is the oldest of the established "high-level" languages.
- The first compiler was made in 1957 and was spread to many computer vendors.
- The first official standard came in 1972 and was called 'FORTRAN 66'.
- It was updated in 1980 to FORTRAN 77.
- It was updated further in 1991 to Fortran 90.
- The next upgrade was in 1996 - removed obsolescent features, corrected mistakes and added limited new facilities such as ELEMENTAL and PURE user-defined procedures and the FORALL statement.
- Fortran is now an ISO/IEC and ANSI standard and keeps improving with time.

Fortran new features. Fortran90 now supports many modern languages' features such as:

- free source form;
- array syntax and many more (array) intrinsics;
- dynamic storage and pointers;
- portable data types (KIND s);
- derived data types and operators;
- recursion;
- MODULE s;

- procedure interfaces;
- enhanced control structures;
- user defined generic procedures;
- enhanced I/O.

Advantages of the new features. Fortran 90 is:

- more natural;
- greater flexibility;
- enhanced safety;
- parallel execution;
- separate compilation;
- greater portability;

but because of of all this it becomes

- larger;
- more complex.

Language obsolescence. Fortran 90 has carried forward the whole of Fortran 77 and also a number of features from existing Fortran compilers. This has been done to protect the investment in the millions of lines of code that have been written in Fortran since it was first developed. Inevitably, as modern features have been added, many of the older features have become redundant and programmers, especially those using Fortran 90 for the first time, need to be aware of the possible pit-falls in using them. Fortran 90 has a number of features marked as obsolescent, this means,

- they are already redundant in Fortran 77;
- better methods of programming already existed in the Fortran 77 standard;
- programmers should stop using them;
- the standards committee's intention is that many of these features will be removed from the next revision of the language, Fortran 95.

Obsolescent features. The following features are labelled as obsolescent and will be removed from the next revision of Fortran, Fortran 95,

- The arithmetic IF statement: It is a three way branch statement of the form,
  IF(<expression >) < label1 >,< label2 >,< label3 >
  Here < **expression** > is any expression producing a result of type INTEGER, REAL or DOUBLE PRECISION, and the three labels are statement labels of executable statements. If the value of the expression is negative, execution transfers to the statement labelled < **label1** >. If the expression is zero, transfer is to the statement labelled < **label2** >, and a positive result causes transfer to< **label3** >. The same label can be repeated. This relic of the original Fortran has been redundant since the early 1960s when the logical IF and computed GOTO were introduced and it should be replaced by an equivalent CASE or IF construct.

- ASSIGN statement: Used to assign a statement label to an INTEGER variable (the label cannot be used as an integer though it is generally used in a GOTO or FORMAT statement.
  ASSIGN $<$ **label** $>$ **TO** $<$ **integer-variable** $>$

- ASSIGN ed GOTO statements: Historically this was used to simulate a procedure call before Fortran had procedures – its use should be replaced by either an IF statement or by a procedure call.

- ASSIGN ed FORMAT statements: The ASSIGN statement can be used to assign a label to an integer which is subsequently referred to in an input/output statement. The same functionality can be obtained by using CHARACTER strings to hold FORMAT specifications. The FORMAT specification can either be this string or a pointer to this string.

- Hollerith format strings: Used to represent strings in a format statement like this:
  WRITE(,100)
  100 FORMAT(17H TITLE OF PROGRAM)
  The use of Hollerith strings is out-of-date as strings can now be delimited by single or double quotes:
  WRITE(,100)
  100 FORMAT('TITLE OF PROGRAM')

- the PAUSE statement:PAUSE was used to suspend execution until a key was pressed on the keyboard.
  PAUSE $<$ **stop code** $>$
  The $<$ **stop code** $>$ is written out at the PAUSE new code should use a PRINT statement for the $<$ stop code $>$ and a READ statement which waits for input to signify that execution should recommence.

- REAL and DOUBLE PRECISION DO-loop control expressions and index variables: In Fortran 77 REAL and DOUBLE PRECISION variables can be used in DO-loop control expressions and index variables. This is unsafe because a loop with real valued DO-loop control expressions could easily iterate a different number of times on different machines – a loop with control expression 1.0,2.0,1.0 may loop once or twice because real valued numbers are only stored approximately, 1.0 + 1.0 could equal, say, 1.99, or 2.01. The first evaluation would execute 2 times whereas the second would only give 1 execution. The solution to this problem is to use INTEGER variables and construct REAL or DOUBLE PRECISION variables within the loop. The following loop is obsolescent:

```
DO x = 1.0,2.0,1.0
 PRINT*, INT(x)
END DO
PRINT*, x
```

- shared DO-loop termination: A number of DO loops can currently be terminated on the same (possibly executable) statement – this causes all sorts of confusion, when programs are changed so that the loops do not logically end on a single statement any more.

```
    IF (N < 1) GOTO 100
    DO 100 K=1,N
    DO 100 J=1,N
    DO 100 I=1,N
    ...
100 A(I,J,K)=A(I,J,K)/2.0
```

  The simple solution is to use END DO instead.

- branching to an ENDIF from outside the IF block: Fortran 77 allowed branching to an END IF from outside its block, this feature is deemed obsolete so, instead, control should be transferred to the next statement instead or, alternatively, a CONTINUE statement could be inserted.

Undesirable features.

- fixed source form layout - use free form;
- implicit declaration of variables - use IMPLICIT NONE;
- COMMON blocks - use MODULE;
- assumed size arrays - use assumed shape;
- EQUIVALENCE statements;
- ENTRY statements;
- the computed GOTO statement - use IF statement.

Fortran 90 Programming

Example Fortran 90 program:

```
MODULE Triangle_Operations
 IMPLICIT NONE
 CONTAINS
 FUNCTION area(x,y,z)
  REAL :: area       ! function type
  REAL, INTENT( IN ) :: x, y, z
  REAL :: theta, height
  theta=ACOS((x**2+y**2-z**2)/(2.0*x*y))
  height=x*SIN(theta); area=0.5*y*height
 END FUNCTION Area
END MODULE Triangle_Operations

PROGRAM Triangle
  USE Triangle_Operations
  IMPLICIT NONE
  REAL :: a, b, c, area
  PRINT*, 'Welcome, please enter the&
         &lengths of the 3 sides.'
  READ*, a, b, c
  PRINT*,'Triangle''s area: ',area(a,b,c)
END PROGRAM Triangle
```

Coding style. It is recommended that the following coding convention is adopted:

- Fortran 90 keywords, intrinsic functions and user defined type names and operators should be in upper case and user entities should be in lower case but may start with a capital letter.
- indentation should be 2 or 3 spaces and should be applied to the bodies of program units, control blocks, INTERFACE blocks, etc.
- the names of program units are always included in their END statements,
- argument keywords are always used for optional arguments,
- always use IMPLICIT NONE.

Please note: In order that a program fits onto a page these rules are sometimes relaxed here. Adopting a specific and documented coding style will make the code easier to read, and will allow other people to be instantly familiar with the style of code.

Source form. The most basic rules in any programming language are those which govern the source form. These rules determine exactly how statements in a program are entered into the computer and how they are displayed. The source form rules are analogous to those fundamentals of natural language which define the alphabet used to express the words in the language, the punctuation symbols to be used, how sentences are to be written (left to right, up and down, right to left, etc.). These are the rules we are going to describe here.
Free source form:

- 132 characters per line;
- '!' comment initiator;
- '&' line continuation character;
- ';' statement separator;
- significant blanks.

Example:

```
     PRINT*, "This line is continued &
             &On the next line"; END ! of program
```

Character set. The following characters are valid in a Fortran 90 program:

alphanumeric:
a-z, A-Z, 0-9, and _ (the underscore)

symbolic:

Symbol	Description	Symbol	Description	Symbol	Description	Symbol	Description
	Space	=	equal	+	plus	-	minus
	asterisk	/	slash	(	left parent	)	right parent
,	comma	.	period	'	sigle quote	"	double quote
:	colon	;	semicolon	!	shriek	&	ampercsand
%	percent	<	less than	>	greater than	$	dollar

The $ and ? do not have any special meaning in the language but may be used to improve the readability of the fortran program.

Comments. It is good practise to use lots of comments, for example,

```
PROGRAM new_socks
!
! Program to evaluate marriage potential
!
    LOGICAL :: SmellySocks  ! Have we smelly socks?
    INTEGER :: i, j         ! Loop variables
```

everything after the ! is a comment; the ! in a character context does not begin a comment, for example, PRINT*, "No chance of ever marrying!!!"

Names. In Fortran 90 variable names (and procedure names etc.) must start with a letter

```
REAL :: a1 ! valid name
REAL :: 1a ! not valid name
```

may use only letters, digits and the underscore

```
CHARACTER :: atoz ! valid name
CHARACTER :: a-z  ! not valid name
CHARACTER :: a_z  ! OK
```

underscore should be used to separate words in long names

```
CHARACTER(LEN=8) :: user_name ! valid name
CHARACTER(LEN=8) :: username  ! different name
```

Statement Ordering. Fortran has some quite strict rules about the order of statements. Basically in any program or procedure the following rules must be used:

1. The program heading statement must come first, (PROGRAM, FUNCTION or SUBROUTINE). A PROGRAM statement is optional but its use is recommended.
2. All the specification statements must precede the first executable statement. Even though DATA statements may be placed with executable text it is far clearer if they lie in the declaration area. It is also a good idea to group FORMAT statements together for clarity.
3. The executable statements must follow in the order required by the logic of the program.
4. The program or procedure must terminate with an END statement, like **END SUBROUTINE mysub**, where **mysub** is the name of the SUBROUTINE.

Within the set of specification statements there is relatively little ordering required, however, in general if one entity is used in the specification of another, it is normally required that it has been previously defined. In other words, named constants (PARAMETER s) must be declared before they can be used as part of the declaration of other objects.

Intrinsic types. Fortran 90 has three broad classes of object type,

1. character;
2. boolean;
3. numeric.

These give rise to five simple intrinsic types, known a default types: CHARACTER for strings of one or more characters; LOGICAL for objects which have the values true or false; REAL (and DOUBLE PRECISION) for approximate, possibly fractional numbers; INTEGER for exact whole numbers; and COMPLEX for representing numbers of the form: $a + ib$. For example,

```
    CHARACTER         :: sex  ! letter
    CHARACTER(LEN=12) :: name ! string
    LOGICAL           :: wed  ! married?
    REAL              :: height
    DOUBLE PRECISION  :: pi   ! 3.14...
    INTEGER           :: age  ! whole No.
    COMPLEX           :: val  ! x + iy
```

Each type has (1) a name (2) a set of valid values (3) a means to denote values (4) a set of operators. Note, most programming languages have the same broad classes of objects. The three broad classes cannot be intermixed without some sort of type coercion being performed. REAL and DOUBLE PRECISION objects are approximate. DOUBLE PRECISION should not now be used. In Fortran 77 an object of this type had greater precision than REAL, in Fortran 90 the precision of a REAL object may be specified making the DOUBLE PRECISION data type redundant. All numeric types have finite range. A default type is not parameterized by a kind value.

Literal constants. A literal constant is an entity with a fixed value:

```
    +12345     ! INTEGER
    2.         ! REAL
    1.0        ! REAL
    -6.6E-06   ! REAL
    -6.6D-06   ! DOUBLE PRECISION
    .FALSE.    ! LOGICAL
    'Mau''dib' ! CHARACTER
    "Mau'dib"  ! CHARACTER
```

Note, there are only two **LOGICAL** values. **Integers** are represented by a sequence of digits with a + or - sign, + signs are optional. **REAL** constants contain a decimal point or an exponentiation symbol, **INTEGER** constants do not. **Character** literals are delimited by the double or single quote symbols, " and '. Two occurrences of the delimiter inside a string produce one occurrence on output; for example 'Mau"dib' but not "Mau"dib" because of the differing delimiters; there is only a finite range of values that numeric literals can take. Constants may also include a kind specifier.

Numeric and logical declarations. Variables of a given type should be declared in type declaration statements at the start of a program unit. A simplified syntax follows,

```
< type > [,< attribute-list >] :: < variable-list > [ =< value > ]
```

The :: is actually optional, however, it does no harm to use it, moreover, if < attribute-list > or =< value > are present then the :: is obligatory. The following are all valid declarations,

```
    REAL :: x
    INTEGER :: i,j
    LOGICAL, POINTER :: ptr
    REAL, DIMENSION(10,10) :: y, z(10)
    DOUBLE PRECISION, DIMENSION(0:9,0:9) :: w
```

The DIMENSION attribute declares a 10×10 this can be overridden as with z which is declared as a 1D array with 10 elements. < **attribute-list** > represents a list of attributes such as **PARAMETER**, **SAVE**,

INTENT, **POINTER**, **TARGET**, **DIMENSION**, (for arrays) or visibility attributes. An object may be given more than one attribute per declaration but some cannot be mixed (such as **PARAMETER** and **POINTER**).

Character declarations. Character variables are declared in a similar way to numeric types. CHARACTER variables can refer to one character; refer to a string of characters which is achieved by adding a length specifier to the object declaration. The following are all valid declarations:

```
CHARACTER(LEN=10)  :: name ! name with 10 characters
CHARACTER          :: sex  ! name with 1 character
CHARACTER(LEN=32)  :: str  ! name with 32 characters
CHARACTER(LEN=10), DIMENSION(10,10) :: Harray ! array containing 10x10 elements of
                                                                      10 chars each
CHARACTER(LEN=32), POINTER :: Pstr ! pointer to a character memory of 32 character
```

Constants (Parameters). Symbolic constants, oddly known as parameters in Fortran, can easily be set up either in an attributed declaration or parameter statement:

```
REAL, PARAMETER :: pi = 3.14159
CHARACTER(LEN=*), PARAMETER :: &
                  son = 'bart', dad = "Homer"
```

CHARACTER constants can assume their length from the associated literal (LEN=*). Parameters should be used: if it is known that a variable will only take one value; for legibility where a 'magic value' occurs in a program such as π ; for maintainability when a 'constant' value could feasibly be changed in the future.

Initialization. Variables can be given initial values: can use initialization expressions, may only contain PARAMETER s or literals.

```
REAL              :: x, y =1.0D5
INTEGER           :: i = 5, j = 100
CHARACTER(LEN=5) :: light = 'Amber'
CHARACTER(LEN=9) :: gumboot = 'Wellie'
LOGICAL  :: on = .TRUE., off = .FALSE.
REAL, PARAMETER :: pi = 3.141592
REAL, PARAMETER :: radius = 3.5
REAL :: circum = 2 * pi * radius
```

gumboot will be padded, to the right, with blanks. In general, intrinsic functions cannot be used in initialization expressions, the following can be: **REPEAT**, **RESHAPE**, **SELECTED_INT_KIND**, **SELECTED_REAL_KIND**, **TRANSFER**, **TRIM**, **LBOUND**, **UBOUND**, **SHAPE**, **SIZE**, **KIND**, **LEN**, **BIT_SIZE** and numeric inquiry intrinsics, for example, **HUGE**, **TINY**, **EPSILON**.

Expressions and Assignment

Assignment. Expressions are often used in conjunction with the assignment operator, =, to give values to objects. This operator is defined between all intrinsic numeric types. The two operands of = (the LHS and RHS) do not have to be the same type. The operator is defined between two objects of the same user-defined type and may be explicitly overloaded so that assignment is meaningful in situations other than those above. Examples:

```
a = b
c = SIN(.7)*12.7
name = initials//surname
bool = (a.EQ.b.OR.c.NE.d)
```

Intrinsic numeric operations. The following operators are valid for numeric expressions:

- ** exponentiation, a dyadic ("takes two operands") operator, for example, 10**2, (evaluated right to left);
- * and / multiply and divide, dyadic operators, for example, 10*7/4;
- + and - plus and minus or add and subtract, monadic ("takes one operand") and dyadic operators, for example, -4 and 7+8-3;

All the above operators can be applied to numeric literals, constants, scalar and array objects with the only restriction being that the RHS of the exponentiation operator must be scalar. Example:

```
a = b - c
f = -3*6/5
```

Note that operators have a predefined precedence, which defines the order that they are evaluated in. The LHS is an object and the RHS is an expression.

Relational operators. The following relational operators deliver a logical result:

- .GT. or > – greater than
- .GE. or >= – greater than or equal to
- .LE. or <= –less than or equal to
- .LT. or < – less than
- .NE. or /= – not equal to
- .EQ. or == – equal to

For example: i .GT. 12 or i > 12 is an expression delivering a .TRUE. or .FALSE. result.

Both sets of symbols may be used in a single statement.

Relational operators:

- compare the values of two operands
- deliver a logical result
- can be applied to numeric operands (restrictions on COMPLEX which can only use .EQ. and .NE.)
- can be applied to default CHARACTER objects – both objects are made to be the same length by padding the shorter with blanks; operators refer to ASCII order (see Appendix 32)
- cannot be applied to LOGICAL objects, for example, (bool .NE. .TRUE.) is not a valid expression but, (.NOT.bool) is
- are used (in scalar) form in IF statements (see Section 11.1) and elementally in the WHERE statement (see Section 14.18)

Consider:

```
bool = i.GT.j
IF (i.EQ.j) c = D
IF (i == j) c = D
```

The example demonstrates,

- simple logical assignment using a relational operator
- IF statements using both forms of relational operators

When using real-valued expressions (which are approximate) .EQ. and .NE. have no real meaning.

```
REAL :: Tol = 0.0001
IF (ABS(a-b) .LT. Tol) same = .TRUE.
```

Intrinsic logical operations. A LOGICAL or boolean expression returns a .TRUE. / .FALSE. result. The following are valid with LOGICAL operands,

- .NOT. – .TRUE. if operand is .FALSE.
- .AND. – .TRUE. if both operands are .TRUE.
- .OR. – .TRUE. if at least one operand is .TRUE.
- .EQV. – .TRUE. if both operands are the same
- .NEQV. – .TRUE. if both operands are different

For example, if T is .TRUE. and F is .FALSE.

- .NOT. T is .FALSE., .NOT. F is .TRUE.
- T .AND. F is .FALSE., T .AND. T is .TRUE.
- T .OR. F is .TRUE., F .OR. F is .FALSE.
- T .EQV. F is .FALSE., F .EQV. F is .TRUE.
- T .NEQV. F is .TRUE., F .NEQV. F is .FALSE.

Intrinsic character operations. Consider:

```
CHARACTER(LEN=*), PARAMETER :: str1 = "abcdef"
CHARACTER(LEN=*), PARAMETER :: str2 = "xyz"
```

substrings can be taken,
str1 is ‘abcdef’
str1(1:1) is ‘a’ (not str1(1) – illegal)
str1(2:4) is ‘bcd’
The concatenation operator, //, is used to join two strings.
PRINT*, str1//str2
PRINT*, str1(4:5)//str2(1:2)
would produce
abcdefxyz
dexy

PRINT statement. This is the simplest form of directing unformatted data to the standard output channel. For example:

```
PROGRAM Outie
 CHARACTER(LEN=*), PARAMETER :: long_name = &
           "Llanfairphwyll...gogogoch"
 REAL :: x, y, z
 LOGICAL :: lacigol
 x = 1; y = 2; z = 3
 lacigol = (y .eq. x)
 PRINT*, long_name
 PRINT*, "Spock says ""illogical&
        &Captain"" "
 PRINT*, "X = ",x," Y = ",y," Z = ",z
 PRINT*, "Logical val: ",lacigol
END PROGRAM Outie
```

produces on the screen:

```
 Llanfairphwyll...gogogoch
 Spock says "illogical Captain"
 X =    1.000  Y =    2.000  Z =    3.000
 Logical val:  F
```

As can be seen from the above example, the PRINT statement takes a comma separated list of things to print, the list can be any printable object including user-defined types (as long as they don't contain pointers). The * indicates the output is in free (default) format. Fortran 90 supports a great wealth of output (and input) formatting which is not all described here! There are a couple of points to raise:

- LOGICAL variables can be printed, lacigol = (y .eq. x) generates an F signifying .FALSE.
- Strings can be split across lines:

  ```
  PRINT*, "Spock says ""illogical&
          &Captain"" "
  ```

If a **CHARACTER** string crosses a line indentation can still be used if an & is appended to the end of the first line and the position from where the string is wanted to begin on the second - see the Spock line in the example; the & s act like a single space.

- The double " in the string, the first one escapes the second. Strings may be delimited by the double or single quote symbols, " and ', but these may not be mixed in order to delimit a string. The following would produce the same output as the statement in the program:

  ```
  PRINT*, 'Spock says "illogical&
          &Captain" '
  ```

  In this case the " delimiter does not have to be escaped.
- Notice how the output has many more spaces than the PRINT statement indicates. This is because output is unformatted. The default formatting is likely to be different between compilers.
- Each PRINT statement begins a new line, non-advancing **I/O** is available but we have to specify it in a **FORMAT** statement.

READ statement. This is the simplest form of reading unformatted data from the standard input channel, for example, if the type declarations are the same as for the PRINT. Example:

```
READ*, long_name
READ*, x, y, z
READ*, lacigol
```

would read the following input from the keyboard

```
Llanphairphwyll...gogogoch
0.4 5. 1.0e12
T
```

Note,

- each READ statement reads from a newline
- the READ statement can transfer any object of intrinsic type from the standard input

The * format specifier in the READ statement is comparable to the functionality of PRINT, in other words, unformatted data is read. (Actually this is not strictly true. Formatted data can be read but the format cannot be specified!) As long as each entity to be read in is blank separated, then the READ statement simply works through its 'argument' list. Each READ statement begins a new line so if there are less arguments to the read statement than there are entries on a line. The extra items will be ignored.

More about operations

Operator precedence.

Operator	Precedence	Example
user defined monadic	Highest	.INVERSE.A
**	-	5**3
* or /	-	5*3 or 12.0/2.4
monadic + or -	-	-2.00
dyadic + or -	-	5+3 or aa-2.00
//	-	string1//string2
> or => or < etc	-	aa > b
.NOT.	-	.NOT. bool
.AND.	-	A .AND. B
.OR.	-	A .OR. B
.EQV. or .NEQV.	-	A .EQV. B
user defined dyadic	-	y .dot. z

Note:

- in an expression with no parentheses, the highest precedence operator is combined with its operands first
- in contexts of equal precedence left to right evaluation is performed except for **

Precedence example. The precedence is worked out as follows:

I. in an expression find the operator(s) with the tightest binding
II. if there are more than one occurrence of this operator then the separate instances are evaluated left to right
III. place the first executed subexpression in brackets to signify this
IV. continue with the second and subsequent subexpressions
V. move to next most tightly binding operator and follow the same procedure

It is easy to make mistakes by forgetting the implications of precedence. The following expression,
x = a+b/5.0-c**d+1*e
is equivalent to
x = ((a+(b/5.0))-(c**d))+1*e

The following procedure has been followed to parenthesize the expression:

- The tightest binding operator is **. This means c**d is the first executed subexpression so should be surrounded by brackets.
- / and * are the second most tightly binding operators and expressions involving them will be evaluated next, put b/5.0 and 1*e in brackets.
- + and - have the next highest precedence. Since they are of equal precedence, occurrences are evaluated from the left.

- At last the assignment is made.

Likewise, the following expression,
.NOT.A.OR.B.EQV.C.AND.D.OR..NOT.E
is equivalent to
((.NOT.A).OR.B).EQV.((C.AND.D).OR.(.NOT.E))

Here:

- the tightest binding operator is **.NOT.** followed by **.AND.** followed by **.OR.**.
- the two subexpressions containing the monadic **.NOT.** are effectively evaluated first, as there are two of these the leftmost, **.NOT.A** is done first followed by **.NOT.E**.
- the subexpression **C.AND.D** is evaluated next followed by **.OR.** (left to right)
- finally the **.EQV.** is executed

Parentheses can be added to any expression to modify the order of evaluation. For a greater readability it is best to often use as many parentheses as needed.
Notice that the CPU cost of an exponentiation is higher than that of a multiplication or division and the latter is higher than that of addition or subtraction. So, whenever possible, try not to use exponentiation. Example: Instead of a**2 use a*a, and instead of 2*a use a +a.

Precision errors. Since real numbers are stored approximately, every operation yields a slight loss of accuracy. After many operations such 'round-off' errors build up and catastrophic accuracy loss can arise when values that are almost equal are subtracted. Leading digits are cancelled and rounding errors become visible. Consider:

```
x = 0.123456; y = 0.123446
PRINT*, "x = ",x," y = ",y
PRINT*, "x-y = ",x-y," should be 0.100d-4"
```

This may produce:

```
x = 0.123457  y = 0.123445
x-y = 0.130d-4 should be 0.100d-4
```

which is 30% in error. A whole branch of numerical analysis is dedicated to minimizing this class of errors in algorithms. Be careful!

Control flow

All structured programming languages need constructs which provide a facility for conditional execution. The simplest method of achieving this functionality is by using a combination of IF and GOTO which is exactly what Fortran 66 supported. Fortran has progressed since then and now includes a comprehensive basket of useful control constructs. Fortran 90 supports:

- conditional execution statements and constructs, (IF statements, and IF ... THEN ... ELSEIF ... ELSE ... END IF); These are the basic conditional execution units. They are useful if a section of code needs to be executed depending on a series of logical conditions being satisfied. If the first condition in the IF statement is evaluated to be true then the code between the IF and the next ELSE, ELSEIF or ENDIF is executed. If the predicate is false then the second branch of the construct is entered. This branch could be either null, an ELSE or an ELSEIF corresponding to no action, the default action or another evaluation of a different predicate with execution being dependent upon the result of the current logical expression. Each IF statement has at least one branch and at most one ELSE branch and may contain any number of ELSEIF branches. Very complex control structures can be built up using multiple nested IF constructs. Before using an IF statement, a programmer should be convinced that a SELECT CASE block or a WHERE assignment block would not be more appropriate.

- loops, (DO ... END DO); This is the basic form of iteration mechanism. This structure allows the body of the loop to be executed a number of times. The number of iterations can be a constant, a variable (expression) or can be dependent on a particular condition being satisfied. DO loops can be nested.
- multi-way choice construct, (SELECT CASE); A particular branch is selected depending upon the value of the case expression. Due to the nature of this construct it very often works out (computationally) cheaper than an IF block with equivalent functionality. This is because in a SELECT CASE block a single control expression is evaluated once and then its (single) result is compared with each branch. With an IF .. ELSEIF block a different control expression must be evaluated at each branch. Even if all control expressions in an IF construct were the same and were simply compared with different values, the general case would dictate that the SELECT CASE block is more efficient.

and less importantly,

- unconditional jump statements, (GOTO); direct jump to a labelled line. This is a very powerful statement, it is very useful and very open to abuse. Unstructured jumps can make a program virtually impossible to follow, the GOTO must be used with care. It is particularly useful for handling exceptions, that is to say, when emergency action is needed to be taken owing to the discovery of an unexpected error.
- I/O exception branching, (ERR=, END=, EOR=); This is a slightly oddball feature of Fortran in the sense that there is currently no other form of exception handling in the language. (The feature originated from Fortran 77.) It is possible to add qualifiers to I/O statements to specify a jump in control should there be an unexpected I/O data error, end of record or should the end of a file be encountered.
- It is always good practice to use at least the ERR= qualifier in I/O statements.

IF statement. Example:

```
IF (bool\_val) A = 3
```

The basic syntax is: IF(< logical-expression >)< exec-stmt >

If < logical-expression > evaluates to .TRUE. then execute < exec-stmt > otherwise do not.

For example:

```
IF (x .GT. y) Maxi = x
```

means 'if x is greater than y then set Maxi to be equal to the value of x'.

More examples:

```
    IF (a*b+c <= 47) Boolie = .TRUE.
    IF (i .NE. 0 .AND. j .NE. 0) k = 1/(i*j)
    IF (i /= 0 .AND. j /= 0) k = 1/(i*j) ! same
```

IF ... THEN ... ELSE construct. The block-IF is a more flexible version of the single line IF. A simple example:

```
    IF (i .EQ. 0) THEN
        PRINT*, "I is Zero"
    ELSE
        PRINT*, "I is NOT Zero"
    ENDIF
```

Note how the indentation helps. There can also be one or more ELSEIF branches:

```
    IF (i ==   0) THEN
     PRINT*, "I is Zero"
    ELSE IF (i > 0) THEN
     PRINT*, "I is greater than Zero"
    ELSE
     PRINT*, "I must be less than Zero"
    ENDIF
```

Both, ELSE and ELSEIF, are optional.

IF ... THEN .... ELSEIF construct. The IF construct has the following syntax:

IF(< logical-expression >)THEN

< then-block >

[ELSEIF(< logical-expression >)THEN

< elseif-block >

[ELSE

< else-block >]

END IF

The first branch to have a true < logical-expression > is the one that is executed. If none are found then the < else-block >, if present, is executed. For example:

```
IF (x > 3) THEN
   CALL SUB1
ELSEIF (x == 3) THEN
   A = B * C - D
ELSEIF (x == 2) THEN
    A = B * B
ELSE
    IF (y /= 0) A = B
ENDIF
```

IF blocks may also be nested.

Conditional exit loops. A loop comprises a block of statements that are executed cyclically. When the end of the loop is reached, the block is repeated from the start of the loop. Loops are differentiated by the way they are terminated. Obviously it would not be reasonable to continue cycling a loop forever. There must be some mechanism for a program to exit from a loop and carry on with the instructions following the End-of-loop. The block of statements making up the loop is delimited by DO and END DO statements. This block is executed as many times as is required. Each time through the loop, the condition is evaluated and the first time it is true the EXIT is performed and processing continues from the statement following the next END DO. Consider:

```
i = 0
DO
  i = i + 1
  IF (i > 100) EXIT
  PRINT*, "I is", i
END DO
```

! if i>100 control jumps here
PRINT*, "Loop finished. I now equals", i
this will generate
I is 1
I is 2
I is 3
....
I is 100
Loop finished. I now equals 101

This type of conditional-exit loop is useful for dealing with situations when we want input data to control the number of times the loop is executed.
The statements between the DO and its corresponding END DO must constitute a proper block. The statements may be labelled but no transfer of control to such a statement is permitted from outside the loop-block. The loop-block may contain other block constructs, for example, DO, IF or CASE, but they must be contained completely; that is they must be properly nested.
An EXIT statement which is not within a loop is an error.

Conditional cycle loops. Situations often arise in practice when, for some exceptional reason, it is desirable to terminate a particular pass through a loop and continue immediately with the next repetition or cycle; this can be achieved in Fortran 90 by arranging that a CYCLE statement is executed at an appropriate point in the loop. For example:

```
    i = 0
    DO
      i = i + 1
      IF (i >= 50 .AND. i <= 59) CYCLE
      IF (i > 100) EXIT
      PRINT*, "I is", i
    END DO
```

PRINT*, ”Loop finished. I now equals”, i

This will generate:

```
I is 1
I is 2
....
I is 49
I is 60
....
I is 100
Loop finished. I now equals 101
```

Here CYCLE forces control to the innermost DO statement (the one that contains the CYCLE statement) and the loop begins a new iteration.
In the example, the statement
IF (i >= 50 .AND. i <= 59) CYCLE
if executed, will transfer control to the DO statement. The loop must still contain an EXIT statement in order that it can terminate.
A CYCLE statement which is not within a loop is an error.

Named and nested loops. Sometimes it is necessary to jump out of more than the innermost DO loop. To allow this, loops can be given names and then the EXIT statement can be made to refer to a particular loop. An analogous situation also exists for CYCLE:

```
0|    outa: DO
1|     inna: DO
2|      ...
3|      IF (a.GT.b) EXIT outa  ! jump to line 9
4|      IF (a.EQ.b) CYCLE outa ! jump to line 0
5|      IF (c.GT.d) EXIT inna  ! jump to line 8
6|      IF (c.EQ.a) CYCLE      ! jump to line 1
7|     END DO inna
8|    END DO outa
9|    ...
```

The (optional) name following the EXIT or CYCLE highlights which loop the statement refers to. For example:

IF (a.EQ.b) CYCLE outa

causes a jump to the first DO loop named outa (line 0).
Likewise,

IF (c.GT.d) EXIT inna

jumps to line 9.

If the name is missing then the directive is applied, as usual, to the next outermost loop, so:

IF (c.EQ.a) CYCLE

causes control to jump to line 1.

The scope of a loop name is the same as that of any construct name.

DO ... WHILE loops. If a condition is to be tested at the top of a loop a DO ... WHILE loop could be used:

```
DO WHILE (a .EQ. b)
 ...
END DO
```

The loop only executes if the logical expression evaluates to .TRUE. Clearly, here, the values of a or b must be modified within the loop otherwise it will never terminate. The above loop is functionally equivalent to:

```
DO; IF (a .NE. b) EXIT
 ...
END DO
```

EXIT and CYCLE can still be used in a DO ... WHILE loop, just as there could be multiple EXIT and CYCLE statements in a regular loop.

Indexed DO loops. Loops can be written which cycle a fixed number of times. For example:

```
DO i = 1, 100, 1
 ...
END DO
```

is a DO loop that will execute 100 times; it is exactly equivalent to

```
DO i = 1, 100
 ...
END DO
```

The syntax is as follows:

```
DO < DO-var >=< expr1 >,< expr2 > [ ,< expr3 > ]

                < exec-stmts >

END DO
```

The loop can be named and the < exec-stmts > could contain EXIT or CYCLE statements, however, a WHILE clause cannot be used but this can be simulated with an EXIT statement if desired.
The number of iterations, which is evaluated before execution of the loop begins, is calculated as

MAX(INT((< expr2 >-< expr1 >+< expr3 >)/< expr3 >),0)

in other words the loop runs from < expr1 > to < expr2 > in steps of < expr3 >. If this gives a zero or negative count then the loop is not executed. (It seems to be a common misconception that Fortran loops always have to be executed once – this came from FORTRAN 66 and is now totally incorrect. Zero executed loops are useful for programming degenerate cases.)
If < expr3 > is absent it is assumed to be 1.
The iteration count is worked out as follows:

- < expr1 > is calculated,
- < expr2 > is calculated,
- < expr3 >, if present, is calculated,
- the DO variable is assigned the value of < expr1 >,
- the iteration count is established (using the formula given above).

The execution cycle is performed as follows (adapted from the standard):

- the iteration count is tested and if it is zero then the loop terminates
- if it is non zero the loop is executed
- (conceptually) at the END DO the iteration count is decreased by one and the DO variable is incremented by < expr3 > (note how the DO variable can be greater than < expr2 >.)
- control passes to the top of the loop again and the cycle begins again

More complex examples may involve expressions and loops running from high to low:

```
DO i1 = 24, k*j, -1
 DO i2 = k, k*j, j/k
  ...
 END DO
END DO
```

An indexed loop could be achieved using an induction variable and EXIT statement, however, the indexed DO loop is better suited as there is less scope for error.
The DO variable cannot be assigned to within the loop.

Examples of loop counts. There now follow a few examples of different loops.

Upper bound not exact:

```
loopy: DO i = 1, 30, 2
 ... ! 15 iterations
END DO loopy
```

According to the rules (given earlier) the fact that the upper bound is not exact is not relevant. The iteration count is INT(29/2) = 14, so it will take the values 1,3,..,27,29 and finally 31 although the loop is not executed when it holds this value, this is its final value.

Negative stride:

```
DO j = 30, 1, -2
 ... ! 15 iterations
END DO
```

This is similar to above except the loop runs the other way (high to low). j will begin with the value 30 and will end up being equal to 0.

A zero-trip loop:

```
DO k = 30, 1, 2
 ... ! 0 iterations
 ... ! loop skipped
END DO
```

This is a false example in the sense that the loop bounds are literals and there would be no point in coding a loop of this fashion as it would never ever be executed! The execution is as follows: First, k is set to 30 and then the iteration count would be evaluated and set to 0. This would mean that the loop is skipped, the only consequence of its existence being that k holds the value 30.

Missing stride – assume it is 1:

```
DO l = 1,30
 ... ! i = 1,2,3,...,30
 ... ! 30 iterations
END DO
```

As the stride is missing it must take its default value which is 1. This loop runs from 1 to 30 so the implied stride means that the loop is executed 30 times. Missing stride:

```
DO l = 30,1
 ... ! zero-trip
END DO
```

As the stride is missing it must take its default value which is 1. This loop runs from high to low (30 to 1) so the implied stride means that the loop is not executed. The final value of l will be 30.

Scope of DO variables. Fortran 90 is not block structured; all DO variables are visible after the loop and have a specific value. The index variable is recalculated at the top of the loop and then compared with

$<$ expr2 $>$, if the loop has finished, execution jumps to the statement after the corresponding END DO. The loop is executed three times and i is assigned to 4 times, the index variable will retain the value that it had just been assigned. For example:

```
DO i = 4, 45, 17
 PRINT*, "I in loop = ",i
END DO
PRINT*, "I after loop = ",i
```

will produce:

```
I in loop = 4
I in loop = 21
I in loop = 38
I after loop = 55
```

Elsewhere in the program, the index variable may be used freely but in the loop it can only be referenced and must not have its value changed.

SELECT CASE construct I. Simple example:

An IF .. ENDIF construct could have been used but a SELECT CASE is neater and more efficient. Another example,

The SELECT CASE construct is similar to an IF construct. It is a useful control construct if one of several paths through an algorithm must be chosen based on the value of a particular expression.

```
SELECT CASE (i)
  CASE (3,5,7)
    PRINT*,"i is prime"
  CASE (10:)
    PRINT*,"i is > 10"
  CASE DEFAULT
    PRINT*, "i is not prime and is < 10"
END SELECT
```

The first branch is executed if i is equal to 3, 5 or 7, the second if i is greater than or equal to 10 and the third branch if neither of the previous has already been executed.

A slightly more complex example with the corresponding IF structure given as a comment:

```
    SELECT CASE (num)
       CASE (6,9,99,66)
!      IF(num==6.OR. .. .OR.num==66) THEN
         PRINT*, "Woof woof"
       CASE (10:65,67:98)
!      ELSEIF((num >= 10 .AND. num <= 65) .OR. ...
        PRINT*, "Bow wow"
       CASE DEFAULT
```

```
!       ELSE
          PRINT*, "Meeeoow"
    END SELECT
!    ENDIF
```

Important points are:

- the < case-expr > in this case is num
- the first < case-selector >, (6,9,99,66), means "if num is equal to either 6, 9, 66 or 99 then"
- the second < case-selector >, (10:65,67:98), means "if num is between 10 and 65 (inclusive) or 67 and 98 (inclusive) then"
- (100:) specifies the range of greater than or equal to one hundred
- if a case branch has been executed then when the next < case-selector > is encountered control jumps to the END SELECT statement
- if a particular case expression is not satisfied then the next one is tested

(An IF .. ENDIF construct could be used but a SELECT CASE is neater and more efficient.) SELECT CASE is more efficient than ELSEIF because there is only one expression that controls the branching. The expression needs to be evaluated once and then control is transferred to whichever branch corresponds to the expressions value. An IF ... ELSEIF ... has the potential to have a different expression to evaluate at each branch point making it less efficient.
Consider the SELECT CASE construct:

```
      SELECT CASE (I)
       CASE(1);   Print*, "I==1"
       CASE(2:9); Print*, "I$>$=2 and I$<$=9"
       CASE(10);  Print*, "I$>$=10"
       CASE DEFAULT; Print*, "I$<$=0"
      END SELECT CASE
```

We can see that if I is equal to 1 only I==1 is printed and then the SELECT CASE is terminated. IF I is between 2 and 9 then I>=2 and I<=9 is printed and the SELECT CASE is terminated etc.

Mixing Objects of Different Types

Mixed numeric type expressions. When an (sub)expression is evaluated, the actual calculation in the CPU must be between operands of the same type. This means if the expression is of mixed type, the compiler must automatically convert (promote or coerce) one type to another. Default types have an implied ordering:

INTEGER – lowest
REAL
DOUBLE PRECISION
COMPLEX – highest

Thus, if an INTEGER is mixed with a REAL the INTEGER is promoted to a REAL and then the calculation is performed; the resultant expression is of type REAL. For example:

1. INTEGER * REAL gives a REAL, (3*2.0 is 6.0)
2. REAL * INTEGER gives a REAL, (3.0*2 is 6.0)
3. DOUBLE PRECISION * REAL gives DOUBLE PRECISION
4. COMPLEX * $<$ anytype $>$ gives COMPLEX
5. DOUBLE PRECISION * REAL * INTEGER gives DOUBLE PRECISION

Consider the expression: int*real*dp*c
The types are coerced as follows:

1. int to REAL
2. int*real to DOUBLE PRECISION
3. (int*real)*dp to COMPLEX.

The above expression is therefore COMPLEX valued. Note that numeric and non-numeric types cannot be mixed using intrinsic operators, nor can LOGICAL and CHARACTER.
In general one must think hard and long about mixed mode arithmetic!

Mixed type assignment. When the RHS expression of a mixed type assignment statement has been evaluated it has a specific type. This type must then be converted to fit in with the LHS. This conversion could be either a promotion or a relegation. For example:

- INTEGER = REAL (or DOUBLE PRECISION) The RHS needs relegating to be an INTEGER value. The RHS is evaluated and then the value is truncated (all the decimal places lopped off) then assigned to the LHS.
- REAL (or DOUBLE PRECISION) = INTEGER The INTEGER needs promoting to become a REAL. The RHS expression is simply stored (approximately) in the LHS.

For example, as real values are stored approximately:

```
REAL :: a = 1.1, b = 0.1
INTEGER :: i, j, k
i = 3.9       ! i will be 3
j = -0.9      ! j will be 0
k = a - b     ! k will be 1 or 0
```

Notes:
Since i is INTEGER, the value 3.9 must be truncated, integers are always formed by truncating towards zero. j (an INTEGER,) would be truncated to 0.
The result of a - b would be close to 1.0 (it could be 1.0000001 or it could be 0.999999999), so, because of truncation, k could contain either 0 or 1.
Care must be taken when mixing types!

Integer division. If one integer divides another in a subexpression then the type of that subexpression is INTEGER. Confusion often arises about integer division; in short, division of two integers produces an integer result by truncation (towards zero). Consider:

```
REAL :: a, b, c, d, e
a = 1999/1000
b = -1999/1000
c = (1999+1)/1000
d = 1999.0/1000
e = 1999/1000.0
```

- a is (about) 1.000. The integer expression 1999/1000 is evaluated and then truncated towards zero to produce an integral value, 1. Its says in the Fortran 90 standard, [1], P84 section 7.2.1.1, "The result of such an operation [integer division] is the integer closest to the mathematical quotient and between zero and the mathematical quotient inclusively."
- b is (about) -1.000 for the same reasons as above.
- c is (about) 2.000 due to the parentheses 2000/1000 is calculated.
- d and e are (about) 1.999 because both RHS's are evaluated to be real numbers. In 1999.0/1000 and 1999/1000.0 the integers are promoted to real numbers before the division.

Intrinsic Procedures

Intrinsic procedures. Some tasks in a language are performed frequently. Fortran 90 has efficient implementations of such common tasks built-in to the language. These procedures are called intrinsic procedures. Fortran 90 has 113 intrinsic procedures in a number of different classes:

- elemental such as:
  - mathematical, such as, trigonometric functions and logarithms, for example, SIN or LOG
  - numeric, for example, SUM or CEILING
  - character, for example, INDEX and TRIM
  - bit manipulation, for example, IAND and IOR (There is no BIT data type but intrinsics exist for manipulating integers as if they were bit variables.)

  Elemental procedures apply to scalar objects as well as arrays – when an array argument is supplied the same function is applied to each element of the array at (conceptually) the same time.
- inquiry, for example, ALLOCATED and SIZE; These report on the status of a program. We can inquire about:
  - the status of dynamic objects
  - array bounds, shape and size
  - kind parameters of an object and available kind representations (useful for portable code)
  - the numerical model; used to represent types and kinds
  - argument presence (for use with OPTIONAL dummy arguments)
- transformational, for example, REAL and TRANSPOSE. The functionality includes:
  - repeat (for characters – repeats strings)

  - mathematical reduction procedures, i.e., given an array return an object of less rank
  - array manipulation – shift operations, RESHAPE, PACK
  - type coercion, TRANSFER copies bit-for-bit to an object of a different type (Stops people doing dirty tricks like changing the type of an object across a procedure boundary which was a popular Fortran 77 'trick'.)
  - PRODUCT and DOT_PRODUCT (arrays)

- miscellaneous (non-elemental SUBROUTINE s) including timing routines, for example, SYSTEM_CLOCK and DATE_AND_TIME

The procedures vary in what arguments are permitted. Some procedures can be applied to scalars and arrays, some to only scalars and some to only arrays. All intrinsics which take REAL valued arguments also accept DOUBLE PRECISION arguments.

Type conversion functions. In Fortran 90 it is easy to explicitly transform the type of a constant or variable by using the in-built intrinsic functions.

- REAL(i) converts the integer i to the corresponding real approximation, the argument to REAL can be INTEGER, DOUBLE PRECISION or COMPLEX.
- INT(x) converts real x to the integer equivalent following the truncation rules given before. The argument to INT can be REAL, DOUBLE PRECISION or COMPLEX. Other functions may form integers from non-integer values:
  - CEILING(x) – smallest integer greater or equal to x,
  - FLOOR(x) – largest integer less or equal to x,
  - NINT(x) – nearest integer to x.
- DBLE(a) converts a to DOUBLE PRECISION, the argument to DBLE can be INTEGER, REAL or COMPLEX.
- CMPLX(x) or CMPLX(x,y) – converts x to a complex value, $x + iy$.
- IACHAR(c) returns the position of the CHARACTER variable c in the ASCII collating sequence, the argument must be a single CHARACTER.
- ACHAR(i) returns the i^{th} character in the ASCII collating sequence (see 32), the argument ACHAR must be a single INTEGER.

For example:

```
PRINT*, REAL(1), INT(1.7), INT(-0.9999)
PRINT*, IACHAR('C'), ACHAR(67)
```

would give:

```
1.000000 1 0
67 C
```

Mathematical intrinsic functions.

- ASIN, ACOS – arcsin and arccos. The argument to each must be real and ≤ 1 , for example, ASIN(0.84147098) has value 1.0 (radians).
- ATAN – arctan. The argument must be real valued, for example, ATAN(1.0) is $\pi/4$, ATAN(1.5574077) has value 1.0.
- ATAN2 – arctan; the principle value of the nonzero complex number (X,Y), for example, ATAN2(1.5574077,1.0) has value 1.0. The two arguments (Y, X) (note order) must be real valued, if Y is zero then X cannot be. These numbers represent the complex value (X,Y).
- TAN, COS, SIN – tangent, cosine and sine. Their arguments must be real or complex and are measured in radians, for example, COS(1.0) is 0.5403.
- TANH, COSH, SINH – hyperbolic trigonometric functions. The actual arguments must be REAL valued, for example, COSH(1.0) is 1.54308.
- EXP, LOG, LOG10, SQRT – exponential , natural logarithm, logarithm base 10 and square root. The arguments must must be real or complex (with certain constraints), for example, EXP(1.0) is 2.7182. Note that SQRT(9) is an invalid expression because the argument to SQRT cannot be INTEGER.

All angles are expressed in radians.

Numeric intrinsic functions. As all are elemental they can accept array arguments, the result is the same shape as the argument(s) and is the same as if the intrinsic had been called separately for each array element of the argument.

- ABS – absolute value. The argument can be INTEGER, REAL or COMPLEX, the result is of the same type as the argument except for complex where the result is real valued, for example, ABS(-1) is 1, ABS(-.2) is 0.2 and ABS(CMPLX(-3.0,4.0)) is 5.0.
- AINT – truncates to a whole number. The argument and result are real valued, for example, AINT(1.7) is 1.0 and AINT(-1.7) is -1.0.
- ANINT – nearest whole number. The argument and result are real valued, for example, AINT(1.7) is 2.0 and AINT(-1.7) is -2.0.
- CEILING, FLOOR – smallest INTEGER greater than (or equal to), or biggest INTEGER less than (or equal to) the argument. The argument must be REAL, for example, CEILING(1.7) is 2 and CEILING(-1.7) is 1.
- CMPLX – converts to complex value. The argument must be two real numbers, for example, CMPLX (3.6,4.5) is a complex number.
- DBLE – coerce to DOUBLE PRECISION data type. Arguments must be REAL, INTEGER or COMPLEX. The result is the actual argument converted to a DOUBLE PRECISION number.
- DIM – positive difference. Arguments must be REAL or INTEGER. If X bigger than Y then DIM(X,Y) = X-Y, if Y>X and result of X-Y is negative then DIM(X,Y) is zero, for example, DIM(2,7) is 0 and DIM(7,2) is 5.
- INT truncates to an INTEGER (as in integer division) Actual argument must be numeric, for example INT(8.6) is 8 and INTCMPLX(2.6,4.0) is 2.
- MAX and MIN – maximum and minimum functions. These must have at least two arguments which must be INTEGER or REAL. MAX(1.0,2.0) is 2.0.

- MOD – remainder function. Arguments must be REAL or INTEGER. MOD(a,p) is the remainder when evaluating a/p, for example, MOD(9,5) is 4, MOD (-9.0,5.0) is -4.0.
- MODULO – modulo function. Arguments must be REAL or INTEGER. MODULO(a,b) is $amodb$, for example, MOD(9,5) is 4, MOD(-9.0,5.0) is 1.0.
- REAL – coverts to REAL value. For example, REAL(5) is 5.0
- SIGN – transfers the sign of the second argument to the first. The arguments are real or integer and the result is of the same type and is equal to ABS(a)*(b/ ABS(b)), for example, SIGN(6,-7) is -6, SIGN(-6,7) is 6.

Character intrinsic functions.

- ACHAR(i) – i^{th} character in ASCII collating sequence. The argument must be between 0 and 127; this function is the inverse of IACHAR, for example ACHAR(100) is 'd'. Compare to CHAR.
- ADJUSTL(str) – adjust a string left. The argument must be a string and the result is the same string with leading blanks removed and inserted as trailing blanks.
- ADJUSTR(str) – adjust a string right. The argument must be a string and the result is the same string with trailing blanks removed and inserted as leading blanks.
- CHAR(i) – i^{th} character in the compilers collating sequence Takes a single character as an argument. The result is similar to ACHAR but uses the compilers collating sequence (this will often be the same as ACHAR.)
- IACHAR(ch) – position of a character in ASCII collating sequence. Takes a single character as an argument which must be an ASCII character, and returns its position of a character in ASCII collating sequence, for example, IACHAR('d') is 100.
- ICHAR(ch) – position of a character in the compilers collating sequence. Takes a single character as an argument (which must be valid) and returns its position of a character in the compilers collating sequence, for example, IACHAR('d') is 100. (The result is often the same as IACHAR.)
- INDEX(str,substr) – starting position of a substring in a string. Takes two arguments, both must be of type CHARACTER and of the same kind, the result is the first occurrence of substr in str, for example, INDEX('gibberish','eris') is 5.
- LEN(str), LEN_TRIM – length of string Both take one string argument the first function returns the length of the string including the trailing blanks and the second discounts the blanks, for example, LEN("Whoosh!! ") is 10, LEN_TRIM ("Whoosh!! ") is 8.
- LGE(str1,str2), LGT, LLE, LLT – lexical positional operators. These functions accept two strings of the same kind, the result is comparable to that of relational operators in the sense that a LOGICAL value is returned governed by the lexical position of the string in ASCII order. This means there is a difference between the case of a letter, for example, LGT('Tin','Tin') returns .FALSE., LGE ('Tin','Tin') and LGE('tin','Tin') return .TRUE.
- REPEAT(str,i) – concatenate string i times. The first argument is a string and the second the number of times it is to be repeated, for example REPEAT('Boutrous ',2) is 'Boutrous Boutrous '.
- SCAN(str,set) – scans string for characters in a set.
- TRIM(str) – remove trailing blanks.
- VERIFY(str,set) – verify that a set of characters contains all the letters in a string. The two arguments, set and string, are characters and of the same kind. Given a set of characters (stored in a string) the result is the first position in the string which is a character that is NOT in the set, for example, VERIFY('ABBA','A') is 2 and VERIFY('ABBA','BA') is 0.

Arrays

Every array has a type (REAL, INTEGER, etc.), so each element holds a value of that type.

Array terminology. Examples of declarations:

```
REAL, DIMENSION(15)      :: X
REAL, DIMENSION(1:5,1:3) :: Y, Z
```

The above are explicit-shape arrays. If the lower bound is not explicitly stated it is taken to be 1.

Terminology:

- rank – the number of dimensions up to and including 7 dimensions. X has rank 1, Y and Z have rank 2.
- bounds – upper and lower limits of indices. An unspecified bound is 1. X has lower bound 1 and upper bound 15, Y and Z have lower bounds of 1 and 1 with upper bounds 5 and 3.
- extent – number of elements in dimension (which can be zero). X has extent 15, Y and Z have extents 5 and 3.
- size – either the total number of elements or, if particular dimension is specified, the number of elements in that dimension. All arrays have size 15.
- shape – rank and extents. X has shape (/15/), Y and Z have shape (/5,3/).
- conformable – two arrays are conformable if they have the same shape – for operations between two arrays the shapes (of the sections) must (generally) conform (just like in mathematics). Y and Z have the same shape so they conform.
- There is no storage association for Fortran 90 arrays.

Explicit-shape arrays can have symbolic bounds so long as they are initialization expressions – evaluatable at compile time.

Declarations. Literals and constants can be used in array declarations:

```
REAL, DIMENSION(100)       :: R
REAL, DIMENSION(1:10,1:10) :: S
REAL                       :: T(10,10)
REAL, DIMENSION(-10:-1)    :: X
INTEGER, PARAMETER            :: lda = 5
REAL, DIMENSION(0:lda-1)      :: Y
REAL, DIMENSION(1+lda*lda,10) :: Z
```

- default lower bound is 1
- bounds can begin and end anywhere, like the array X
- there is a shorthand form of declaration, see T
- arrays can be zero-sized (if lda = 0)

Array conformance. If an object or sub-object is used directly in an expression then it must conform with all other objects in that expression. (Note that a scalar conforms to any array with the same value for every element.) For two array references to conform both objects must be of the same shape.

A and B have the same size (15 elements) but have different shapes, so they cannot be directly equated. To force conformance the array must be used as an argument to a transformational intrinsic to change its shape, for example:

```
B = RESHAPE(A,(/5,3/))  ! is, see later
A = PACK(B,.TRUE.)      ! is, see later
A = RESHAPE(B,(/10,8/)) ! is, see later
B = PACK(A,.TRUE.)      ! is, see later
```

Arrays can have their shapes changed by using transformational intrinsics including MERGE, PACK, SPREAD, UNPACK and RESHAPE.

Array element ordering. Fortran 90 does not have any storage association meaning that, unlike FORTRAN 77, the standard does not specify how arrays are to be organized in memory. This makes passing arrays to a procedure written in a different language very difficult. The lack of implicit storage association makes it easier to write portable programs and allows compiler writers more freedom to implement local optimizations. For example, in distributed memory computers an array may be stored over 100 processors with each processor owning only a small section of the whole array – the standard will allow this. There are certain situations where an ordering is needed, for example, during input or output. Under these circumstances Fortran 90 does define ordering which can be used in such contexts. It is defined in the same manner as the Fortran 77 storage association but it does not imply anything about how array elements are stored. The array element ordering is again of column major form:
C(1,1),C(2,1),..,C(5,1),C(1,2),C(2,2),..,C(5,3)

This ordering is used in array constructors, I/O statements, certain intrinsics (TRANSFER, RESHAPE, PACK, UNPACK and MERGE) and any other contexts where an ordering is needed.

Array syntax. Can reference:

- whole arrays
  - A = 0.0 sets whole array A to zero
  - B = C + D adds C and D, then assigns result to B
- elements
  - A(1) = 0.0 sets one element to zero
  - B(0,0) = A(3) + C(5,1) sets an element of B to the sum of two other elements
- array sections
  - A(2:4) = 0.0 sets A(2), A(3) and A(4) to zero
  - B(-1:0,1:2) = C(1:2,2:3) + 1.0 adds one to the subsection of C and assigns to the subsection of B

Whole array expressions. A whole (or section of an) array can be treated like a single variable in which all intrinsic operators that apply to intrinsic types have their meaning extended to apply to conformable arrays. For example: B = C * D - B**2
As long as B, C and D conform the above assignment is valid. (Recall that the RHS of the ** operator must be scalar.) Note that in the above example, C*D is not matrix multiplication; MATMUL(C,D) should be used if this is the desired operation. The above assignment is equivalent to:

```
!PARALLEL
    B(-4,0) = C(1,1)*D(0,0)-B(-4,0)**2 ! in ||
    B(-3,0) = C(2,1)*D(1,0)-B(-3,0)**2 ! in ||
     ...
    B(-4,1) = C(1,2)*D(0,1)-B(-4,1)**2 ! in ||
     ...
    B(0,2)  = C(5,3)*D(4,2)-B(0,2)**2  ! in ||
!END PARALLEL
```

With array assignment there is no implied order of the individual assignments. They are conceptually performed in parallel. In addition to the above operators, the subset of intrinsic functions termed elemental can also be applied. For example: B = SIN(C)+COS(D)
The functions are also applied element by element. Thus, the above is equivalent to the parallel execution of:

```
!PARALLEL
    B(-4,0) = SIN(C(1,1))+COS(D(0,0))
     ...
    B(0,2)  = SIN(C(5,3))+COS(D(4,2))
!END PARALLEL
```

Many of Fortran 90's intrinsics are elemental including all numeric, mathematical, bit, character and logical intrinsics. Again it must be stressed that conceptually there is no order implied in the array statement form – each individual assignment can be thought of as being executed in parallel between corresponding elements of the arrays. This is different from the DO-loop.

Array sections. The general form of a subscript-triplet specifier is:

```
[< bound1 >]:[< bound2 >][:< stride >]
```

The section starts at < bound1 > and ends at or before < bound2 >. < stride > is the increment by which the locations are selected. < bound1 >, < bound2 > and < stride > must all be scalar integer expressions. Thus:

```
   A(:)         ! the whole array
   A(3:9)       ! A(m) to A(n) in steps of 1
   A(3:9:1)     ! as above
   A(m:n)       ! A(m) to A(n)
   A(m:n:k)     ! A(m) to A(n) in steps of k
   A(8:3:-1)    ! A(8) to A(3) in steps of -1
   A(8:3)       ! A(8) to A(3) step 1 => zero size
   A(m:)        ! from A(m) to default UPB
   A(:n)        ! from default LWB to A(n)
   A(::2)       ! from default LWB to UPB step 2
   A(m:m)      ! 1 element section
   A(m)        ! scalar element - not a section
```

are all valid.
If the upper bound ($<$ bound2 $>$) is not a combination of the lower bound plus multiples of the stride then the actual upper bound is different from that stated. This is the same principle that is applied to DO-loops. Another similarity with the DO-loops is that when the stride is not specified it is assumed to have a value of 1. In the above example, this means that A(3:8) is the same as A(3:8:1) but A(8:3) is a zero sized section and A(8:3:-1) is a section that runs backwards. Zero strides are not allowed and, in any case, are pretty meaningless!
Other bound specifiers can be absent too, if $<$ bound1 $>$ or $<$ bound2 $>$ is absent then the lower or upper bound of the dimension (as declared) is implied. If both are missing then the whole dimension is assumed.

Printing arrays. The conceptual ordering of array elements is useful for defining the order in which array elements are output. If A is a 2D array then:

PRINT*, A

would produce output in Array Element Order:
A(1,1), A(2,1), A(3,1), ..., A(1,2), A(2,2), ...

Sections of arrays can also be output. For example:
PRINT*, A(::2,::2)

would produce:
A(1,1), A(3,1), A(5,1), ..., A(1,3), A(3,3), A(5,3), ...

An array of more than one dimension is not formatted neatly. If it is desired that the array be printed out row-by-row (or indeed column by column) then this must be programmed explicitly. This order could be changed by using intrinsic functions such as RESHAPE, TRANSPOSE or CSHIFT.

Array I/O example. The PRINT statements in the following program:

```
   PROGRAM print_A
    IMPLICIT NONE
     INTEGER, DIMENSION(3,3) :: A = RESHAPE((/1,2,3,4,5,6,7,8,9/))
      PRINT*, 'Array element   =',a(3,2)
      PRINT*, 'Array section   =',a(:,1)
      PRINT*, 'Sub-array       =',a(:2,:2)
      PRINT*, 'Whole Array     =',a
      PRINT*, 'Array Transp''d =',TRANSPOSE(a)
   END PROGARM print_A
```

produce on the screen:

```
    Array element    = 6
    Array section    = 1 2 3
```

```
Sub-array         = 1 2 4 5
Whole Array       = 1 2 3 4 5 6 7 8 9
Array Transposed = 1 4 7 2 5 8 3 6 9
```

Array bound violations. Attempting to reference beyond the end of an array is one of the commonest errors in Fortran programming. Array subscript references must lie within the bounds declared for an array. Compilers rarely check for this by default although many provide a switch for this purpose (for example, -CB on the EPC compiler, -C on the NAg compiler). The following example demonstrates a possible bound violation (if M ≥ 1):

```
REAL, DIMENSION(50) :: A
 ...
DO I=1,M
  A(I) = A(I-1) + 1.0 ! refs A(0)
END DO
```

A bound violation may or may not cause the program to crash. In the example above, what will happen is that the reference A(0) will access the bit pattern that is resident in the memory location just before the start of the array. If the contents of this location can be interpreted as a REAL number then this value will be used (whatever it may be), otherwise the program will crash. It is quite likely that the value of this location will be different on each run of the program. Array bound errors are often very difficult to spot! Therefore, it is recommended that during development work array bound checking should always be switched on (along with the generate debug information switch, -g, if available) – this will alert the user to inconsistencies, however, bound checking will usually slow down the code noticeably.

Array inquiry intrinsics. These intrinsics allow the user to quiz arrays about their attributes and status and are most often applied to dummy arguments in procedures. Consider the declaration:

```
REAL, DIMENSION(-10:10,23,14:28) :: A
```

The following inquiry intrinsics are available:

- LBOUND(SOURCE[,DIM]) Returns a one dimensional array containing the lower bounds of an array or, if a dimension is specified, a scalar containing the lower bound in that dimension. For example:
  - LBOUND(A) is (/-10,1,14/) (array)
  - LBOUND(A,1) is -10 (scalar)
- UBOUND(SOURCE[,DIM]) Returns a one dimensional array containing the upper bounds of an array or, if a dimension is specified, a scalar containing the upper bound in that dimension. For example:
  - UBOUND(A) is (/10,23,28/)
  - UBOUND(A,1) is 10
- SHAPE(SOURCE) Returns a one dimensional array containing the shape of an object. For example:
  - SHAPE(A) is (/21,23,15/) (array)
  - SHAPE((/4/)) is (/1/) (array)
- SIZE(SOURCE[,DIM]) Returns a scalar containing the total number of array elements either in the whole array or in an optionally specified dimension. For example:

  - SIZE(A,1) is 21
  - SIZE(A) is 7245
  - SIZE(4) is an error as the argument must not be scalar

- ALLOCATED(SOURCE) Returns a scalar LOGICAL result indicating whether an array is allocated or not. For example:

```
    PROGRAM Main
     IMPLICIT NONE
      INTEGER, ALLOCATABLE, DIMENSION(:) :: Vec
       PRINT*, ALLOCATED(Vec)
       ALLOCATE(Vec(10))
       PRINT*, ALLOCATED(Vec)
       DEALLOCATE(Vec)
       PRINT*, ALLOCATED(Vec)
    END PROGRAM Main
```

  will produce: .FALSE., .TRUE. and .FALSE. (in that order)

Array constructors. Array constructors are used to give arrays or sections of arrays specific values. An array constructor is a comma separated list of scalar expressions delimited by (/ and /). The results of the expressions are placed into the array in array element order with any type conversions being performed in the same manner as for regular assignment. The constructor must be of the correct length for the array, in other words, the section and the constructor must conform. For example:

```
PROGRAM MAIN
  IMPLICIT NONE
  INTEGER, DIMENSION(1:10)  :: ints
  CHARACTER(len=5), DIMENSION(1:3) :: colours
  REAL, DIMENSION(1:4)      :: heights
   heights = (/5.10, 5.6, 4.0, 3.6/)
   colours = (/'RED  ','GREEN','BLUE '/)
   ! note padding so strings are 5 chars
   ints    = (/ 100, (i, i=1,8), 100 /)
    ...
END PROGRAM MAIN
```

The array and its constructor must conform. Notice that all strings in the constructor for colours are 5 characters long. This is because the string within the constructor must be the correct length for the variable.
(i, i=1,8) is an implied-DO specifier and may be used in constructors to specify a sequence of constructor values. There may be any number of separate implied DOs which may be mixed with other specification methods. In the above example the vector ints will contain the values (/ 100, 1, 2, 3, 4, 5, 6, 7, 8, 100 /). Note the format of the implied DO: a DO-loop index specification surrounded by parentheses. There is a restriction that only one dimensional constructors are permitted, for higher rank arrays the RESHAPE intrinsic must be used to modify the shape of the result of the RHS so that it conforms to the LHS:

```
INTEGER, DIMENSION(1:3,1:4) :: board
board = RESHAPE((/11,21,31,12,22,32,13,23,33,14,24,34/), (/3,4/))
```

The values are specified as a one dimensional constructor and then the shape is modified to be a 3×4 array which conforms with the declared shape of board.

Array constructors in initialization statements. Named array constants of any rank can be created using the RESHAPE function as long as all components can be evaluated at compile time (just like in initialization expressions):

```
    INTEGER, DIMENSION(3), PARAMETER :: Unit_vec = (/1,1,1/)
    CHARACTER(LEN=*), DIMENSION(3), PARAMETER :: &
          lights = (/'RED  ','BLUE ','GREEN'/)
    REAL, DIMENSION(3,3), PARAMETER :: &
          unit_matrix = RESHAPE((/1,0,0,0,1,0,0,0,1/), (/3,3/))
```

Note how the string length of the PARAMETER lights can be assumed from the length of the constructor values. The strings in the constructor must all be the same length. Previously defined constants (PARAMETER s) may also be used to initialize variables. Consider:

```
    INTEGER, DIMENSION(3,3) :: &
          unit_matrix_T = RESHAPE(unit_matrix, (/3,3/), ORDER=(/2,1/))
```

This assigns the transpose of unit_matrix to unit_matrix_T.

Allocatable arrays. Fortran 90 allows arrays to be created on-the-fly. These are known as deferred-shape arrays and use dynamic heap storage (this means memory can be grabbed, used and then put back at any point in the program). This facility allows the creation of "temporary" arrays which can be created used and discarded at will. Deferred-shape arrays are:

- declared like explicit-shape arrays but without the extents and with the ALLOCATABLE attribute:

  ```
  INTEGER, DIMENSION(:), ALLOCATABLE :: ages
  REAL, DIMENSION(:,:), ALLOCATABLE :: speed
  ```

- given a size in an ALLOCATE statement which reserves an area of memory for the object:

  ```
  ALLOCATE(ages(1:10), STAT=ierr)
  IF (ierr .NE. 0) THEN
   PRINT*, "ages: Allocation request denied"
  END IF
  ALLOCATE(speed(-lwb:upb,-50:0),STAT=ierr)
  IF (ierr .NE. 0) THEN
   PRINT*, "speed: Allocation request denied"
  END IF
  ```

  In the ALLOCATE statement we could specify a list of objects to create but in general one should only specify one array statement. If there is more than one object and the allocation fails it is not immediately possible to tell which allocation was responsible. The optional STAT= field reports on the success of the storage request. If it is supplied then the keyword must be used to distinguish it from an array that needs allocating. If the result (ierr) is zero the request was successful otherwise it failed. This specifier should be used as a matter of course.

There is a certain overhead in managing dynamic or ALLOCATABLE arrays – explicit-shape arrays are cheaper and should be used if the size is known and the arrays are persistent (are used for most of the life of the program). The dynamic or heap storage is also used with pointers and, obviously, has only a finite size – there may be a time when this storage runs out. If this happens there may be an option of the compiler to specify / increase the size of heap storage.

Deallocating arrays. Heap storage can be reclaimed using the DEALLOCATE statement:

IF (ALLOCATED(ages)) DEALLOCATE(ages,STAT=ierr)

- It is an error to deallocate an array without the ALLOCATE attribute or one that has not been previously allocated space.
- There is an intrinsic function, ALLOCATED, which returns a scalar LOGICAL value reporting on the status of an array.
- The STAT= field is optional but its use is recommended.
- If a procedure containing an allocatable array which does not have the SAVE attribute is exited without the array being deallocated then this storage becomes inaccessible.

As a matter of course, the LOGICAL valued intrinsic inquiry function, ALLOCATED, should be used to check on the status of the array before attempting to deallocate because it is an error to attempt to deallocate an array that has not previously been allocated space or one which does not have the ALLOCATE attribute. Again one should only supply one array per DEALLOCATE statement and the optional STAT= field should always be used. ierr holds a value that reports on the success / failure of the DEALLOCATE request in an analogous way to the ALLOCATE statement.
Memory leakage will occur if a procedure containing an allocatable array (which does not possess the SAVE attribute) is exited without the array being deallocated, (this constraint will be relaxed in Fortran 95). The storage associated with this array becomes inaccessible for the whole life of the program.
Consider the following sorting program which can handle any number of items:

```
PROGRAM sort_any_number
 !---------------------------------------------------------!
 ! Read numbers into an array, sort into ascending order   !
 ! and display the sorted list                             !
 !---------------------------------------------------------!
 INTEGER, DIMENSION(:), ALLOCATABLE :: nums
 INTEGER :: temp, I, K, n_to_sort, ierr

  PRINT*, 'How many numbers to sort'
  READ*,  n_to_sort

  ALLOCATE( nums(1:n_to_sort), STAT=ierr)
  IF (ierr .NE. 0) THEN
   PRINT*, "nums: Allocation request denied"
   STOP ! halts execution
  END IF

  PRINT*,  'Type in ',n_to_sort, 'values one line at a time'

  DO I=1,n_to_sort
   READ*, nums(I)
  END DO

  DO I = 1, n_to_sort-1
   DO K = I+1, n_to_sort
    IF(nums(I) > nums(K)) THEN
     temp = nums(K)      ! Store in temporary location
     nums(K) = nums(I)   ! Swap the contents over
```

```
        nums(I) = temp
       END IF
      END DO
     END DO

     DO I = 1, n_to_sort
      PRINT*, 'Rank ',I,' value is ',nums(I)
     END DO

     IF (ALLOCATED(nums)) DEALLOCATE(nums, STAT=ierr)
     IF (ierr .NE. 0) THEN
      PRINT*, "nums: Deallocation request denied"
     END IF

   END PROGRAM sort_any_number
```

Where construct. There is a block form of masked assignment:

```
WHERE(A > 0.0)
 B = LOG(A)
 C = SQRT(A)
ELSEWHERE
 B = 0.0 ! C is NOT changed
ENDWHERE
```

- The mask must conform to the RHS of each assignment. A, B and C must conform.
- WHERE ... END WHERE is not a control construct and cannot currently be nested. The execution sequence is as follows: evaluate the mask, execute the WHERE block (in full), then execute the
- ELSEWHERE block. The separate assignment statements are executed sequentially but the individual elemental assignments within each statement are (conceptually) executed in parallel.

Program units

Program units. Fortran 90 has two main program units:

- Main PROGRAM: The place where execution begins and where control should eventually return before the program terminates. The main program may contain any number of procedures.
- MODULE: A program unit which can also contain procedures and declarations. It is intended to be attached to any other program unit where the entities defined within it become accessible. A module is similar to a C++ class. MODULE program units are new to Fortran 90 and are supposed to replace the unsafe Fortran 77 features such as COMMON, INCLUDE, BLOCK DATA as well as adding a much needed (limited) 'object oriented' aspect to the language. Their importance cannot be overstressed and they should be used whenever possible.

There are two classes of procedures:

- SUBROUTINE: A parameterized named sequence of code which performs a specific task and can be invoked from within other program units by the use of a CALL statement, for example,

CALL PrintReportSummary(CurrentFigures)
Here, control will pass into the SUBROUTINE named PrintReportSummary, after the SUBROUTINE has terminated control will pass back to the line following the CALL.

- FUNCTION: As a SUBROUTINE but returns a result in the function name (of any specified type and kind). This can be compared to a mathematical function, say, f(x). An example of a FUNCTION call could be:
  PRINT*, "The result is", f(x)
  Here, the value of the function f (with the argument x) is substituted at the appropriate point in the output.

Procedures are generally contained within a main program or a module. It is also possible to have 'stand alone' or EXTERNAL procedures. These will be discussed later.

Main program syntax. This is the only compulsory program unit. Every program must have one:

```
[ PROGRAM [ < main program name > ] ]

                                ...

                 < declaration of local objects >

                                ...

                 < executable stmts >

                                ...

[ CONTAINS

                 < internal procedure definitions > ]

 END [ PROGRAM [ < main program name > ] ]
```

The PROGRAM statement and < main program name > are optional, however, it is good policy to use them always. < main program name > can be any valid Fortran 90 name. The main program contains declarations and executable statements and may also contain internal procedures. These internal procedures are separated from the surrounding program unit, the host (in this case the main program), by a CONTAINS statement. Internal procedures may only be called from within the surrounding program unit and automatically have access to all the host program unit's declarations but may also override them.
Internal procedures may not contain further internal procedures, in other words, the nesting level is a maximum of 1. The diagram shows two internal procedures, Sub1 and Funkyn. However, there may be any number of internal procedures (subroutines or functions) which are wholly contained within the main program. The main program may also contain calls to external procedures. This will be discussed later. The main program must contain an END statement as its last (non-blank) line. For neatness sake this should really be suffixed by PROGRAM (so it reads: END PROGRAM) and should also have the name of the program attached too. Using as descriptive as possible END statements help to reduce confusion.

Main program example. The following example demonstrates a main program which calls an intrinsic function (FLOOR) and an internal procedure (Negative):

```
    PROGRAM Main
```

```
 IMPLICIT NONE
 REAL x
 INTRINSIC FLOOR
  READ*, x
  PRINT*, FLOOR(x)
  PRINT*, Negative(x)
CONTAINS
 REAL FUNCTION Negative(a)
  REAL, INTENT(IN) :: a
   Negative = -a
 END FUNCTION Negative
END PROGRAM Main
```

Although not totally necessary, the intrinsic procedure is declared in an INTRINSIC statement (the type is not needed – the compiler knows the types of all intrinsic functions).
The internal procedure is 'contained within' the main program and does not require declaring in the main program. The compiler is able to 'see' the procedure and therefore knows its result type, number and type of arguments.

Procedures. A procedure, such as an intrinsic function, is an abstracted block of parameterized code that performs a particular task. Procedures should generally be used if a task has to be performed two or more times, this will cut down on code duplication. Before writing a procedure the first question should be: "Do we really need to write this or does a routine already exist?" Very often a routine with the functionality already exists, for example, as an intrinsic procedure or in a library somewhere. (Fortran 90 has 113 intrinsic procedures covering a variety of functionality and the NAg fl90 Numerical Library contains over 300 mathematic procedures so there is generally a wide choice!) The NAg library deals with solving numerical problems and is ideal for engineers and scientists. fl90, the NAg Fortran 90 Mk I library, has just been released as a successor to the well respected and popular Fortran 77 library which contains at least 1140 routines.
Other libraries include: BLAS (Basic Linear Algebra Subroutines), for doing vector, matrix-vector and matrix-matrix calculations (these should always be used if possible); IMSL (Visual Numerics), akin to NAg Library; LaPACK (linear algebra package); Uniras (graphics routines), which are very comprehensive. Many of these packages will be optimized and shipped along with the compiler. Note that the HPFF defined a set of routines that should be available as part of an HPF compilation system. If the target platform is to be a parallel computer it will be worth investigating further; a number of Fortran 90 version of these procedures exist, for example, at LPAC.
As the use of Fortran 90 grows many useful (portable) library modules will be developed which contain routines that can be used by any Fortran 90 program (see World Wide Web Fortran Market).
There is also an auxiliary Fortran 90 standard known as the "Varying String" module. This is to be added to the Fortran 95 standard and will allow users to define and use objects of type VARYING_STRING where CHARACTER objects would normally be used. The standard has already been realized in a module (by Lawrie Schonfelder at Liverpool University). All the intrinsic operations and functions for character variables have be overloaded so that VARYING_STRING objects can be used in more or less the same way as other intrinsic types which contain routines that can be used by any Fortran 90 program (see World Wide Web Fortran Market).
If a procedure is to be written from scratch then the following guidelines should be followed:

- It is generally accepted that procedures should be no more than 50 lines long in order to keep the control structure simple and to keep the number of program paths at a minimum.
- Procedures should be as flexible as possible to allow for software reuse. Try to pass as many of the
- variable entities referenced in a procedure as actual arguments and do not rely on global storage or host association unless absolutely necessary.

- Try to give procedures meaningful names and initial descriptive comments.
- There is absolutely no point in reinventing the wheel – if a procedure or collection of procedures already exist as intrinsic functions or in a library module then they should be used.

Subroutines. Consider the following example:

```
PROGRAM Thingy
 IMPLICIT NONE
 .....
  CALL OutputFigures(Numbers)
 .....
CONTAINS
 SUBROUTINE OutputFigures(Numbers)
  REAL, DIMENSION(:), INTENT(IN) :: Numbers
   PRINT*, "Here are the figures", Numbers
 END SUBROUTINE OutputFigures
END PROGRAM Thingy
```

The subroutine here simply prints out its argument. Internal procedures can 'see' all variables declared in the main program (and the IMPLICIT NONE statement). If an internal procedure declares a variable which has the same name as a variable from the main program then this supersedes the variable from the outer scope for the length of the procedure. Using a procedure here allows the output format to be changed easily. To alter the format of all outputs, it is only necessary to change on line within the procedure. Internal subroutines lie between CONTAINS and END PROGRAM statements and have the following syntax:

```
SUBROUTINE < procname >[ (< dummy args >) ]

                < declaration of dummy args >

                < declaration of local objects >

                                ...

                < executable stmts >

END [ SUBROUTINE [< procname > ] ]
```

A SUBROUTINE may include calls to other procedures either from the same main program, from an attached module or from an external file. Note how, in the same way as a main program, a SUBROUTINE must terminate with an END statement. It is good practice to append SUBROUTINE and the name of the routine to this line as well.
Fortran 90 also allows recursive procedures (procedures that call themselves). In order to promote optimization a recursive procedure must be specified as such – it must have the RECURSIVE keyword at the beginning of the subroutine declaration

Functions. Consider the following example:

```
PROGRAM Thingy
 IMPLICIT NONE
 .....
```

```
   PRINT*, F(a,b)
  .....
 CONTAINS
  REAL FUNCTION F(x,y)
   REAL, INTENT(IN) :: x,y
    F = SQRT(x*x + y*y)
  END FUNCTION F
 END PROGRAM Thingy
```

FUNCTION operates on the same principle as SUBROUTINE, the only difference being that a FUNCTION returns a value. In the example, the line
PRINT, F(a,b)
will substitute the value returned by the FUNCTION for F(a,b), in other words, the value of $\sqrt{x^2+y^2}$. Just like SUBROUTINE, FUNCTION also lies between CONTAINS and END PROGRAM statements. It has the following syntax:

```
[< prefix >] FUNCTION < procname >( [< dummyargs >])

                < declaration of dummy args >

                < declaration of local objects >

                                ...

                < executable stmts, assignment of result >

END [ FUNCTION [ < procname > ] ]
```

It is also possible to declare the FUNCTION type in the DECLARATION area instead of in the HEADER:

```
FUNCTION < procname >( [< dummy args >])

                < declaration of dummy args >

                < declaration of result type >

                < declaration of local objects >

                                ...

                < executable stmts, assignment of result >

END [ FUNCTION [ < procname > ] ]
```

This would mean that the above FUNCTION could be equivalently declared as:

```
FUNCTION F(x,y)
 REAL             :: F
 REAL, INTENT(IN) :: x,y
  F = SQRT(x*x + y*y)
END FUNCTION F
```

A FUNCTION may also be recursive, see Section 17.10, and may be either scalar or array valued (including user defined types and pointers). Note that, owing to the possibility of confusion between an array reference and a FUNCTION reference, the parentheses are not optional.

Argument association. Recall, in the SUBROUTINE example we had an invocation:

CALL OutputFigures(NumberSet)

and a declaration:

SUBROUTINE OutputFigures(Numbers)

An argument in a call statement (in an invocation), for example, NumberSet, is called an actual argument since it is the true name of the variable. An argument in a procedure declaration is called a dummy argument, for example, Numbers as it is a substitute for the true name. A reference to a dummy argument is really a reference to its corresponding actual argument, for example, changing the value of a dummy argument actually changes the value of the actual argument. Dummys and actuals are said to be argument associated. Procedures may have any number of such arguments but actuals and dummys must correspond in number, type, kind and rank. (Fortran 77 programs which flouted this requirement were not standard conforming but there was no way for the compiler to check.) For the above call, Numbers is the dummy argument and NumberSet is the actual argument.

Consider:

PRINT*, F(a,b) and
REAL FUNCTION F(x,y)

Here, the actual arguments a and b are associated with the dummy arguments x and y. If the value of a dummy argument changes then so does the value of the actual argument.

Local objects. In the following procedure:

```
SUBROUTINE Madras(i,j)
 INTEGER, INTENT(IN) :: i, j
 REAL                :: a
 REAL, DIMENSION(i,j):: x
```

a, and x are known as local objects and x will probably have a different size and shape on each call. They

- are created each time a procedure is invoked,
- are destroyed when the procedure completes,
- do not retain their values between calls,
- do not exist in the programs memory between calls.

So, when a procedure is called, any local objects are bought into existence for the duration of the call. Thus, if an object is assigned to on one call, the next time the program unit is invoked a totally different instance of that object is created with no knowledge of what happened during the last procedure call meaning that all values are lost.
The space usually comes from the programs stack.

Argument intent. In order to facilitate efficient compilation and optimization hints, in the form of attributes, they can be given to the compiler as to whether a given dummy argument will:

- hold a value on procedure entry which remains unchanged on exit – INTENT(IN).
- not be used until it is assigned a value within the procedure – INTENT(OUT).
- hold a value on procedure entry which may be modified and then passed back to the calling program – INTENT(INOUT).

For example:

```
SUBROUTINE example(arg1,arg2,arg3)
  REAL, INTENT(IN) :: arg1
  INTEGER, INTENT(OUT) :: arg2
  CHARACTER, INTENT(INOUT) :: arg3
  REAL r
   r = arg1*ICHAR(arg3)
   arg2 = ANINT(r)
   arg3 = CHAR(MOD(127,arg2))
END SUBROUTINE example
```

It can be seen here that:

- arg1 is unchanged within the procedure.
- The value of arg2 is not used until it has been assigned to.
- arg3 is used and then reassigned a value.

The use of INTENT attributes is not essential but it allows good compilers to check for coding errors thereby enhancing safety. If an INTENT(IN) object is assigned a value or if an INTENT(OUT) object is not assigned a value then errors will be generated at compile time.

Host association. In Fortran 77, different routines were entirely separate from each others. They did not have access to each others variable space and could only communicate through argument lists or by global storage (COMMON). Such procedures are known as external.
Procedures in Fortran 90 may contain internal procedures which are only visible within the program unit in which they are declared, in other words, they have a local scope. Consider the following example:

```
PROGRAM CalculatePay
 IMPLICIT NONE
 REAL :: Pay, Tax, Delta
 INTEGER :: NumberCalcsDone = 0
 Pay = ...;  Tax = ... ; Delta = ...
 CALL PrintPay(Pay,Tax)
 Tax = NewTax(Tax,Delta)
  ....
CONTAINS
 SUBROUTINE PrintPay(Pay,Tax)
  REAL, INTENT(IN) :: Pay, Tax
  REAL :: TaxPaid
   TaxPaid = Pay * Tax
   PRINT*, TaxPaid
   NumberCalcsDone = NumberCalcsDone + 1
 END SUBROUTINE PrintPay
 REAL FUNCTION NewTax(Tax,Delta)
  REAL, INTENT(IN) :: Tax, Delta
   NewTax =  Tax + Delta*Tax
   NumberCalcsDone = NumberCalcsDone + 1
 END FUNCTION NewTax
END PROGRAM CalculatePay
```

PrintPay is an internal subroutine of *CalculatePay* and has access to *NumberCalcsDone*. It can be thought of as a global variable. It is said to be available to the procedures by host association. The variables *Pay* and *Tax*, on the other hand, are passed as arguments to the procedures. This means that they are available by argument association.

The decision of whether to give visibility to an object by host association or by argument association is tricky – there are no hard and fast rules. If, for example, *Pay*, *Tax* and *Delta* had not been passed to the procedures as arguments but had been made visible by host association instead, then there would be no discernible difference in the results that the program produces. Likewise, *NumberCalcsDone* could have been communicated to both procedures by argument association instead of by host association. In a sense, the method that is used will depend on personal taste or a specific 'in-house' coding-style. Here, *Pay*, *Tax* and *Delta* are not used in every single procedure in the same way as *NumberCalcsDone* which acts in a more global way!
NewTax cannot access any of the local declarations of *PrintPay* (for example, *TaxPaid*) and vice-versa. *NewTax* and *PrintPay* can be thought of a resting at the same scoping level whereas the containing program, *CalculatePay* is at an outer (higher) scoping level.
PrintPay can invoke other internal procedures which are contained by the same outer program unit (but cannot call itself as it is not recursive).
Upon return from *PrintPay* the value of *NumberCalcsDone* will have increased by one owing to the last line of the procedure.

Scope of names. Consider the following example,

```
PROGRAM Proggie         ! scope Proggie
 IMPLICIT NONE
 REAL ::  A, B, C       ! scope Proggie
 CALL sub(A)            ! scope Proggie
CONTAINS
 SUBROUTINE Sub(D)      ! scope Sub
  REAL :: D             ! D is dummy (alias for A)
  REAL :: C             ! local C (diff from Proggie's C)
   C = A**3             ! A cannot be changed
   D = D**3 + C         ! D can be changed
   B = C                ! B from Proggie gets new value
 END SUBROUTINE Sub
```

```
   SUBROUTINE AnuvvaSub  ! scope AnuvvaSub
     REAL :: C           ! another local C (unrelated)
       .....
   END SUBROUTINE AnuvvaSub
  END PROGRAM Proggie
```

This demonstrates most of the issues concerning the scope of names in procedures.
If an internal procedure declares an object with the same name as one in the host (containing) program unit then this new variable supersedes the one from the outer scope for the duration of the procedure, for example, *Sub* accesses *B* from *Proggie* but supersedes *C* by declaring its own local object called *C*. This *C* (in *Sub*) is totally unrelated to the *C* of *Proggie*.
Internal procedures may have dummy arguments, however, any object used as an actual argument to an internal procedure cannot be changed by referring to it by its original name in that procedure; it must be assigned to using its dummy name. For example, *Sub* accesses *A*, known locally as *D*, by argument association and is forbidden to assign a value to or modify the availability of *A*; it must be altered through its corresponding dummy argument. The variable *A* can still be referenced in *Sub* as if it possessed the INTENT(IN) attribute (see Section concerning the implications of the INTENT attribute).
A local variable called *A* could be declared in *Sub* which would clearly bear no relation to the *A* which is argument associated with *D*. When *Sub* is exited and control is returned to *Proggie*, the value that *C* had before the call to the subroutine is restored.
The *C* declared in *AnuvvaSub* bears no relation to the *C* from *Proggie* or the *C* from *Sub*.

SAVE attribute. The SAVE attribute can be

- applied to a specified variable. In the following example, NumInvocations is initialised only on the first call (conceptually at program start-up) and then retains its new value between calls:

  ```
  SUBROUTINE Barmy(arg1,arg2)
   INTEGER, SAVE :: NumInvocations = 0
    NumInvocations = NumInvocations + 1
  ```

- applied to the whole procedure by appearing on a line on its own. This means that all local objects are saved:

  ```
  SUBROUTINE polo(x,y)
   IMPLICIT NONE
   INTEGER :: mint, neck_jumper
   SAVE
   REAL :: stick, car
  ```

  In the above example mint, neck_jumper, stick and car all have the SAVE attribute.

Variables with the SAVE attribute are known as static objects and have static storage class. In fact, the SAVE attribute is given implicitly if an object, which is not a dummy argument or a PARAMETER, appears in an initializing declaration in a procedure. So:

INTEGER, SAVE :: NumInvocations = 0

is equivalent to

INTEGER :: NumInvocations = 0

However, the former is clearer!

Clearly, the SAVE attribute has no meaning in the main program since when it is exited, the program has finished executing.

Keyword arguments. Normal argument correspondence is performed by position. The first actual argument corresponds to the first dummy argument and so on. Fortran 90 includes a facility to allow actual arguments to be specified in any order. At the call site actual arguments may be prefixed by keywords (the dummy argument name followed by an equals sign) which are used when resolving the argument correspondence. If keywords are used then the usual positional correspondence of arguments is replaced by keyword correspondence, for example, consider the following interface description:

```
SUBROUTINE axis(x0,y0,l,min,max,i)
 REAL, INTENT(IN) :: x0,y0,l,min,max
 INTEGER, INTENT(IN) :: i
  .....
END SUBROUTINE axis
```

The SUBROUTINE can be invoked in two ways:

using positional argument invocation: CALL AXIS(0.0,0.0,100.0,0.1,1.0,10)
using keyword arguments: CALL AXIS(0.0,0.0,Max=1.0,Min=0.1, & L=100.0,I=10)

The names are from the dummy arguments.
As soon as one argument is prefixed by a keyword then all subsequent arguments (going left to right) must also be prefixed.
Note: If an EXTERNAL procedure is invoked with keyword arguments then the INTERFACE must be explicit at the call site.
In summary, keyword arguments

- allow actual arguments to be specified in any order.
- help improve the readability of the program.
- (when used in conjunction with optional arguments) make it easy to add an extra argument without the need to modify each and every invocation in the calling code.
- are the dummy argument names.

Optional arguments. Dummy arguments with the optional attribute can be used to allow default values to be substituted in place of absent actual arguments. Any argument with the OPTIONAL attribute may be omitted from an actual argument list but as soon as one argument has been dropped, all subsequent arguments (going left to right) must be keyword arguments to allow the compiler to resolve the argument correspondence. Any use of optional arguments requires the interface of the procedure to be explicit; the compiler needs to know the ordering, type, rank and names of the dummy arguments to work out the correspondence. The status of an optional argument can be found using the PRESENT intrinsic function. Many of the intrinsic procedures have optional arguments and keywords can be used for these procedures in exactly the same way as for user defined procedures.

Optional arguments example. Consider the following internal subroutine with two optional arguments:

```
    SUBROUTINE SEE(a,b)
     IMPLICIT NONE
     REAL, INTENT(IN), OPTIONAL    :: a
     INTEGER, INTENT(IN), OPTIONAL :: b
     REAL    :: ay; INTEGER :: bee
     ay = 1.0; bee = 1
     IF(PRESENT(a)) ay  = a
     IF(PRESENT(b)) bee = b
      ...
```

Both, a and b, have the OPTIONAL attribute, so the SUBROUTINE, *SEE*, can be called in the following ways:

```
    CALL SEE()
    CALL SEE(1.0,1); CALL SEE(b=1,a=1.0) ! same
    CALL SEE(1.0);   CALL SEE(a=1.0)     ! same
    CALL SEE(b=1)
```

In the above example of procedure calls, the first call uses both default values, the second and third use none, and the remaining two both have one missing argument. Within the procedure it must be possible to check whether OPTIONAL arguments are missing or not, the PRESENT intrinsic function, which delivers a scalar LOGICAL result, has been designed especially for this purpose.
If an optional argument is missing then it may not be assigned to, for example, the following is invalid:

IF (.NOT.PRESENT (up)) up = .TRUE. ! WRONG!!!

Recursive Procedures

Recursion occurs when procedures call themselves (either directly or indirectly). Any procedural call chain with a circular component exhibits recursion. Even though recursion is a neat and succinct technique to express a wide variety of problems, if used incorrectly it may incur certain efficiency overheads. In Fortran 77 recursion had to be simulated by a user defined stack and corresponding manipulation functions. In Fortran 90 it is supported as an explicit feature of the language. For matters of efficiency, recursive procedures (SUBROUTINE and FUNCTION) must be explicitly declared using the RECURSIVE keyword (see below). Declarations of recursive functions have a slightly different syntax to regular declarations. The RESULT keyword must be used with recursive functions and specifies a variable name where the result of the function can be stored. (The RESULT keyword is necessary since it is not possible to use the function name to return the result. Array valued recursive functions are allowed and sometimes a recursive function reference would be indistinguishable from an array reference. The function name implicitly has the same attributes as the result name.)
The fact that a function exhibits recursion must be declared in the header. Valid declarations are:

- INTEGER RECURSIVE FUNCTION fact(N) RESULT(N_Fact)
- RECURSIVE INTEGER FUNCTION fact(N) RESULT(N_Fact)
  (In the above the INTEGER applieds to both fact and N_Fact.)

- RECURSIVE FUNCTION fact(N) RESULT(N_Fact)
  INTEGER N_Fact

In the last case INTEGER N_Fact implicitly gives a type to fact; it is actually illegal to also specify a type for fact.
Subroutines are declared using the RECURSIVE SUBROUTINE header.

Recursive function example. The following program calculates the factorial of a number, $n!$, and uses $n! = n(n-1)!$:

```
PROGRAM Mayne
 IMPLICIT NONE
  PRINT*, fact(12) ! etc
CONTAINS
 RECURSIVE FUNCTION fact(N) RESULT(N_Fact)
  INTEGER, INTENT(IN)  :: N
  INTEGER :: N_Fact ! also defines type of fact
   IF (N > 0) THEN
     N_Fact = N * fact(N-1)
   ELSE
     N_Fact = 1
   END IF
 END function FACT
END PROGRAM Mayne
```

The INTEGER keyword in the function header specifies the type for both, fact and N_fact. The recursive function repeatedly calls itself. On each call the value of the argument is reduced by one and the function is called again. Recursion continues until the argument is zero. When this happens the recursion begins to unwind and the result is calculated. For example, $4!$ is calculated as follows:

- $4!$ is $4 \times 3!$, so calculate $3!$ then multiply by 4,
- $3!$ is $3 \times 2!$, need to calculate $2!$,
- $2!$ is $2 \times 1!$, $1!$ is $1 \times 0!$, and $0! = 1$.
- Now the code can work back up the calculation and fill in the missing values.

Input / Output

Fortran 90 has a wealth of I/O, too much to cover here. Fortran 90 allows a number of different streams (files) to be connected to a program for both reading and writing. In Fortran 90 a file is connected to a logical unit denoted by a number. Each unit can have many properties, for example:

file – the name of the file connected to the unit
action – read, write, read and write, etc.
status – old, new, replace, etc.
access method – sequential, direct.

OPEN statement. The OPEN statement is used to connect a named file to a logical unit. It is often possible to pre-connect a file before the program has begun. If this is the case then there is no need for an OPEN statement, however, there are many default I/O settings of which many are processor dependent. It is good practice not to rely on defaults but to specify exactly what is required in an explicit OPEN statement. This will make the program more portable.
The syntax is:

```
OPEN([UNIT=]< integer >, FILE=< filename >, ERR=< label >, &
          STATUS=< status >, ACCESS=< method >, ACTION=< mode >, RECL=< int-expr >)
```

where

- UNIT=< integer > specifies a numeric reference for the named file. The number must be one that is not currently in use.
- FILE=< filename > gives the filename to be associated with the logical unit. The name must match exactly with the actual file. Sometimes this means that the filename needs to contain a specific number of blanks.
- ERR=< label > specifies a numeric label where control should be transferred if there is an error opening the named file.
- STATUS=< status > specifies the status of the named file. The status may be one of the following:
  - 'OLD', – file exists.
  - 'NEW', – file does not exist.
  - 'REPLACE' – file will be overwritten.
  - 'SCRATCH' – file is temporary and will be deleted when closed.
  - 'UNKNOWN' – unknown.
- ACCESS=< method > specifies the access method:
  - 'DIRECT' – The file consists of tagged records accessed by an ID number. In this case the record length must be specified (RECL). Individual records can be specified and updated without altering the rest of the file.
  - 'SEQUENTIAL' – The file is written / read (sequentially) line by line.
- ACTION=< mode > specifies what can be done to the file. The mode may be one of the following:
  - 'READ',– open for reading.
  - 'WRITE'– open for writing.
  - 'READWRITE' – open for both, reading and writing.

There are other less important specifiers which are not covered here. There now follows an example of use:

```
    OPEN(17,FILE='output.dat',ERR=10, STATUS='REPLACE', &
            ACCESS='SEQUENTIAL',ACTION='WRITE')
```

A file output.dat is opened for writing. It is connected to logical unit number 17. The file is accessed on a line by line basis and already exists but is to be replaced. The label 10 must pertain to a valid executable statement.

```
    OPEN(14,FILE='input.dat',ERR=10, STATUS='OLD', RECL=iexp, &
            ACCESS='DIRECT',ACTION='READ')
```

Here a file is opened for input only on unit 14. The file is directly accessed and (clearly) already exists.

READ statement. Syntax (not all the specifiers can be used at the same time):

```
READ([UNIT=]< unit >, [FMT=]< format >, IOSTAT=< int-variable >, ERR=< label >, &
         END=< label >, EOR=< label >, ADVANCE=< advance-mode >, REC=< int-expr >, &
         SIZE=< num-chars >) < output-list >
```

where

- UNIT=< unit > is a valid logical unit number or * for the default (standard) output. If it is the first field then the specifier is optional.
- FMT=< format > is a string of formatting characters, a valid FORMAT statement label or a * for free format. If this is the second field then the specifier is optional.
- IOSTAT=< int-variable > specifies an integer variable to hold a return code. As usual, zero means no error.
- < label > in ERR= is a valid label to where control jumps if there is a read error.
- < label > in END= is a valid label to where control jumps if an end-of-file is encountered. This can only be present in a READ statement.
- < advance-mode > specifies whether each READ should start a new record or not. Setting the specifier to 'NO' initiates non-advancing I/O. The default is 'YES'. If non-advancing I/O is used then the file must be connected for sequential access and the format must be explicitly stated.
- EOR will jump to the specified label if an end-of-record is encountered. This can only be present in a READ statement and only if ADVANCE='NO' is also present.
- REC=< int-expr > is the record number for direct access.
- The SIZE specifier is used in conjunction with an integer variable, in this case nch. The variable will hold the number of characters read. This can only be present in a READ statement and only if ADVANCE='NO' is also present.

Consider:

READ(14,FMT='(3(F10.7,1x))',REC=iexp) a,b,c

Here, the record specified by the integer iexp is read – this record should contain 3 real numbers separated by a space with 10 column and 7 decimal places. The values will be placed in a, b and c.

READ(*,'(A)',ADVANCE='NO',EOR=12,SIZE=nch) str

Since non-advancing output has been specified, the cursor will remain on the same line as the string is read. Under normal circumstances (default advancing output) the cursor would be positioned at the start of the next line. Note the explicit format descriptor. SIZE returns the length of the string and EOR= specifies an destination if an end-of-record is encountered.

WRITE statement. READ and WRITE can be interchanged in the following (not all the specifiers can be used at the same time):

```
WRITE([UNIT=] unit >, [FMT=]< format >, IOSTAT=< int-variable >, ERR=< label >, &

                   ADVANCE=< advance-mode >, REC=< int-expr >) < output-list >
```

where

- UNIT=< unit > is a valid logical unit number or * for the default (standard) output. If it is the first field then the specifier is optional.
- FMT=< format > is a string of formatting characters, a valid FORMAT statement label or a * for free format. If this is the second field then the specifier is optional.
- IOSTAT=< int-variable > specifies an integer variable to hold a return code. As usual, zero means no error.
- < label > in ERR= is a valid label to where control jumps if there is an WRITE error.
- ADVANCE=< advance-mode > specifies whether each WRITE should start a new record or not. Setting the specifier, < advance-mode >, to 'NO' initiates non-advancing I/O. The default is 'YES'. If non-advancing I/O is used then the file must be connected for sequential access and the format must be explicitly stated.
- REC=< int-expr > is the record number for direct access.

Consider:

WRITE(17,FMT='(I4)',IOSTAT=istat,ERR=10, END=11,EOR=12) ival

Here, '(I4)' means INTEGER with 4 digits. The labels 10, 11 and 12 are statements where control can jump on an I / O exception. ival is an INTEGER variable whose value is written out.

WRITE(*,'(A)',ADVANCE='NO') 'Input : '

Since non-advancing output has been specified, the cursor will remain on the same line as the string 'Input : '. Under normal circumstances (default advancing output) the cursor would be positioned at the start of the next line. Note the explicit format descriptor. We will later give a more detailed explanation of the format edit descriptors.

FORMAT statement / FMT= specifier. The FMT= specifier in a READ or WRITE statement can give either a line number of a FORMAT statement, an actual format string or a *.
Consider:

WRITE(17,FMT='(2X,2I4,1X,"name ",A7)')i,j,str

This writes out to the file output.dat the three variables i, j and str according to the specified format, "2 spaces, (2X), 2 integer valued objects of 4 digits with no gaps, (2I4), one space, (1X), the string 'name '

and then 7 letters of a character object, (A7)". The variables are taken from the I/O list at the end of the line. Note the double single quotes (escaped quote) around name. A single " could have been used here instead.
Consider:

READ(14,*) x,y

This reads two values from input.dat using free format and assigns the values to x and y.
Further consider:

```
    WRITE(*,FMT=10) a,b
 10 FORMAT('vals',2(F15.6,2X))
```

The WRITE statement uses the format specified in statement 10 to write to the standard output channel. The format is "2 instances of: a real valued object spanning 15 columns with an accuracy of 6, decimal places, (F15.6), followed by two spaces (2X)".
Given:

```
    WRITE(*,FMT=10) -1.05133, 333356.0
    WRITE(17,FMT='(2X,2I4,1X,''name '',A7)')11, -195, 'Philip'
 10 FORMAT('vals',2(F15.6,2X))
```

the following is written:

```
    11-195 name Philip
      -1.051330    333356.000000
```

Editor	Meaning
Iw	w integers of integer data
F$w.d$	w characters of real data (d decimal places)
E$w.d$	w characters of real data (d decimal places)
Lw	w characters of logical data
Aw	w characters of CHARACTER data
nX	skin n spacess

Edit descriptors. Fortran contains a large number of output descriptors which allow one to format the output, meaning that very complex I/O patterns can be specified. Here we only intend to give a brief summary. Any good textbook will elaborate the topic in detail:

- w determines the total number of characters (column width) that the data span. (On output, if there are insufficient columns for the data then numeric data will not (cannot) be printed, and CHARACTER data will be truncated).
- d specifies the number of decimal places that the data contain and is contained in the total number of characters. The number will be rounded (not truncated) to the desired accuracy.
- I specifies that the data is of INTEGER type.
- E writes REAL using exponent form, 1.000E+00.
- F uses 'decimal point' form, 1.000. For F and E the sign is included in the total number of characters. By default plus signs are not output.

- L specifies LOGICAL data. .TRUE. and .FALSE. are output as a single character T or F. If w is greater that one them the single character is padded to the left with blanks.
- A specifies CHARACTER data. If w is specified strings which are too long (more than w characters) will be truncated. If w is not specified any length of string is appropriate.
- X skips the specified number of characters (n blanks are output).

Descriptors or groups of descriptors can be repeated by prefixing or parenthesesing and prefixing with the number of repeats. For example, I4 can be repeated twice by specifying 2I4 and I4. 1X can be repeated twice by specifying 2(I4,1X). Many of the above edit descriptors can be demonstrated:

```
        WRITE(*,FMT='(2X,2(I4,1X),''name '',A4,F13.5,1X,E13.5)') &
              77778,3,'ABCDEFGHI',14.45,14.5666666
```

gives:

```
bb****bbbb3bnamebABCDbbbbb14.45000bbb0.14567E+02
```

where the b signifies a blank! In the above example, the first INTEGER is cannot be written as the number is too long, and the last REAL number is rounded to fit into the spaces available. The string is truncated to fit into the space available. A READ statement could use the same format editor. Type coercion is not performed so INTEGER cannot be written out as REAL.

Other I/O Statements.

- CLOSE unattaches the unit number specified in the statement. This should always be used to add an end of file mark at the closure point. It is an error to close a file that is not open.
- REWIND simply puts the file pointer back to the start.
- BACKSPACE moves the file pointer back one record, however, it often puts the file pointer back to the start, and then fast forwards.
- ENDFILE forces an end-of-file to be written into the file but the file remains open.

The above statements have other specifiers such as IOSTAT. For example:

```
    REWIND (UNIT=14)
    BACKSPACE (UNIT=17)
    ENDFILE (17)
    CLOSE (17, IOSTAT=ival)
```

Chapter II

Basic Numerical Methods

II.1 Interpolation and Approximation

II.1.1 Introduction

In general we interpolate a function $f(x)$, which is known only on a finite grid, in order to obtain values of $f(x)$ outside the points of the grid (see Fig. II.1). The values of the function on the grid are often measured or are the results of expensive numerical calculation. The interpolation assumes that the function to be interpolated is smooth so that one can use a polynomial of a lower degree to determine values of the function outside the tabulated values.

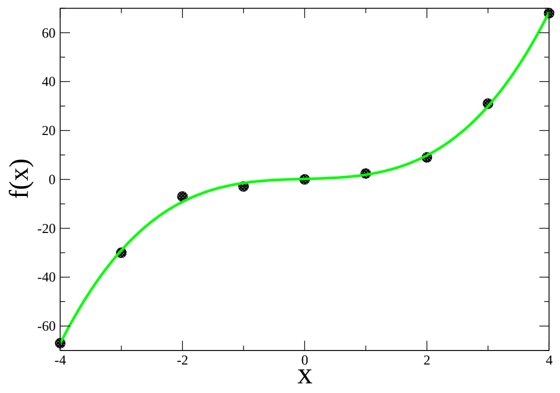


Figure II.1: Interpolation (solid line) of a function $f(x)$ known at n points (filled circles).

The function is known in a finite number of points n:

$$\left.\begin{array}{lll} \bullet & x_0 & y_0 = f(x_0) \\ \bullet & x_1 & y_1 = f(x_1) \\ \bullet & \ddots & \ddots \\ \bullet & x_n & y_1 = f(x_n) \end{array}\right\} \quad \text{We need to know } f(x) \text{ where } x \neq x_i.$$

In general, one tries to determine a physical law from the measured experimental data. Because of errors in the data, it is best to perform a least squares fit.

We would like that the polynomial $\Phi(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ passes by the known points. We can also use orthogonal functions $g_i(x)$ to perform the interpolation, so that $\Phi(x) = \sum_{i=0}^{n} a_i g_i(x_i)$. The orthogonal functions are usually trigonometric or exponential functions. Before we deal with interpolation we will first expose to the method of finite differences.

II.1.2 Finite Differences

The techniques of finite differences deal with a tabulated function in a certain number of points. To determine a value of the function at an untabulated point we need to perform an interpolation. We can also determine the derivative or the definite integral of a tabulated function. Finite differences can also be used to find the numerical solution of partial differential equations.

We will always assume that the interval between successive values is small enough to display the variation of the tabulated function. This will help us to determine the values of the function at some argument value between the tabulated values. In this case, we can obtain intermediate values to a good accuracy by using a polynomial representation of low degree of the function f.

Symbolic Linear Operators: Forward, Backward, Central Difference, Differential, and Integral Operators

There are several different notations for the single set of finite difference operators. We introduce each of these notations in terms of the so-called shift operator, which we will define first.

The shift operator E

Let $f_i = f(x_i)$, where $x_i = x_0 + ih$, be a set of n values of the function $f(x)$. The shift operator E is defined by

$$E f_i = f_{i+1} \ , \tag{II.1}$$

so that

$$E^k f_i = E^{k-1}(E f_i) = E^{k-1} f_{i+1} = \cdots = f_{i+k} \ ,$$

where k is any positive integer. The last formula can be extended to all real values of j and k, so that, for example,

$$E^{0.27} f_i = f_{i+0.27} = f(x_0 + (i + 0.27)h) \ .$$

The forward difference operator $\boldsymbol{\Delta}$

This operator Δ is defined by

$$\Delta \equiv E - 1 \ , \tag{II.2}$$

so that

$$\Delta f_i = (E - 1) f_i = f_{i+1} - f_i \ ,$$

which is the first-order forward difference at x_i. Similarly, we can determine the second-order forward difference at x_i:

$$\Delta^2 = \Delta(\Delta f_i) = \Delta(f_{i+1} - f_i) = f_{i+2} - 2f_{i+1} + f_i \ .$$

We can determine the forward difference of order k in the same manner.

The backward difference operator $\boldsymbol{\nabla}$

This operator ∇ is defined by

$$\nabla \equiv 1 - E^{-1} \ , \tag{II.3}$$

so that

$$\nabla f_i = (1 - E^{-1})f_i = f_i - f_{i-1} ,$$

which is the first-order backward difference at x_i. Similarly, we can determine the second-order forward difference at x_i:

$$\nabla^2 f_i = \nabla(\nabla f_i) = \nabla(f_i - f_{i-1}) = f_i - 2f_{i-1} + f_{i-2} .$$

We can determine the backward difference of order k in the same manner.

The central difference operator δ

This operator δ is defined by

$$\delta \equiv E^{\frac{1}{2}} - E^{-\frac{1}{2}} , \tag{II.4}$$

so that

$$\delta f_i = (E^{\frac{1}{2}} - E^{-\frac{1}{2}})f_i = f_{i+\frac{1}{2}} - f_{j+\frac{1}{2}}$$

is the first-order central difference at x_i. Similarly, we can determine the second-order central difference at x_i:

$$\delta^2 f_i = \delta(\delta f_i) = \delta(f_{i+\frac{1}{2}} - f_{j+\frac{1}{2}}) = f_{i+1} - 2f_i + f_{i-1} .$$

We can determine the central difference of order k in the same manner.

The differential operator D and integral operator I

These operators can be defined as follows:

$$Df(x) = \frac{\mathrm{d}f(x)}{\mathrm{d}x} = f'(x) \tag{II.5}$$

and

$$If(x) = \int_x^{x+h} f(x)\mathrm{d}x . \tag{II.6}$$

All these operators are distributive, commutative and associative.

Relation between symbolic linear operators

We have seen that

$$\Delta = E - 1 \quad \text{and} \quad \nabla = 1 - E^{-1} .$$

Thus,

$$\Delta = E - 1 = E\nabla . \tag{II.7}$$

We can also relate the integral operator I, Δ and D by the following relation:

$$I = \Delta D^{-1} ,$$

where D^{-1} is the inverse of the derivative, defined by

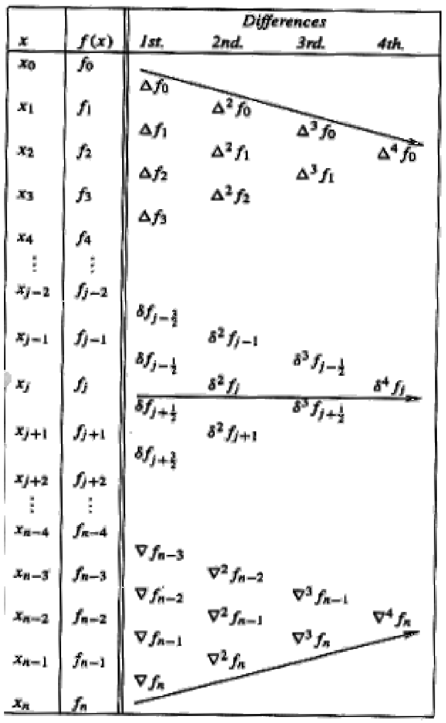
$$D^{-1}f(x) = \int f(x)\mathrm{d}x + \text{const} .$$

We can relate E to D and Δ to D using the Taylor expansion of a function,

$$f(x+h) = f(x) + \frac{h}{1!}Df(x) + \frac{h^2}{2!}D^2 f(x) + \cdots + \frac{h^n}{n!}D^n f(x) + \cdots = \mathrm{e}^{hD} f(x) .$$

It is then straightforward to relate E, D, Δ and δ by

$$E = \mathrm{e}^{hD}, \qquad \Delta = \mathrm{e}^{hD} - 1, \qquad \text{and} \qquad \nabla = 1 - \mathrm{e}^{hD} .$$



x	$f(x)$	Differences 1st.	2nd.	3rd.	4th.
x_0	f_0				
		Δf_0			
x_1	f_1		$\Delta^2 f_0$		
		Δf_1		$\Delta^3 f_0$	
x_2	f_2		$\Delta^2 f_1$		$\Delta^4 f_0$
		Δf_2		$\Delta^3 f_1$	
x_3	f_3		$\Delta^2 f_2$		
		Δf_3			
x_4	f_4				
$\vdots$	$\vdots$				
x_{j-2}	f_{j-2}				
		$\delta f_{j-\frac{3}{2}}$			
x_{j-1}	f_{j-1}		$\delta^2 f_{j-1}$		
		$\delta f_{j-\frac{1}{2}}$		$\delta^3 f_{j-\frac{1}{2}}$	
x_j	f_j		$\delta^2 f_j$		$\delta^4 f_j$
		$\delta f_{j+\frac{1}{2}}$		$\delta^3 f_{j+\frac{1}{2}}$	
x_{j+1}	f_{j+1}		$\delta^2 f_{j+1}$		
		$\delta f_{j+\frac{3}{2}}$			
x_{j+2}	f_{j+2}				
$\vdots$	$\vdots$				
x_{n-4}	f_{n-4}				
		∇f_{n-3}			
x_{n-3}	f_{n-3}		$\nabla^2 f_{n-2}$		
		∇f_{n-2}		$\nabla^3 f_{n-1}$	
x_{n-2}	f_{n-2}		$\nabla^2 f_{n-1}$		$\nabla^4 f_n$
		∇f_{n-1}		$\nabla^3 f_n$	
x_{n-1}	f_{n-1}		$\nabla^2 f_n$		
		∇f_n			
x_n	f_n				

Figure II.2: Table of difference of a function $f(x)$.

Table of difference of a function $f(x)$ [see Fig. II.2]:

$$\left.\begin{array}{lrcl} \bullet & \Delta^{r+1} f_i & = & \Delta^r f_{i+1} - \Delta^r f_i \\ \bullet & \delta^{r+1} f_i & = & \delta^r f_{i+\frac{1}{2}} - \delta^r f_{i-\frac{1}{2}} \\ \bullet & \nabla^{r+1} f_i & = & \nabla^r f_i - \nabla^r f_{i-1} \end{array}\right\} \text{ see their definitions.}$$

II.1.3 Newton Interpolation

Whenever the higher differences of a tabulated function become small, the function represented may be approximated well by a polynomial. For example, the function $f(x) = e^x$ with $0.1 \le x \le 0.4$ with a step $h = 0.05$ has small fourth-order differences and can be represented by a polynomial. Notice, however, that there are many analytical functions with singularities of severe oscillations that can not be represented by a polynomial.

Let's first consider the Newton forward interpolation formula:

The function f(x) is tabulated as a set of points $[x_i,\, f(x_i)]$, where $i \in \{0, 1, \ldots, n-1, n\}$, and $x_i = x_0 + ih$. For any argument x, we can write $f(x) = \Phi_n(x) + R(x)$, where $\Phi_n(x)$ is a polynomial passing by all the tabulated values, and $R(x)$ is a function that is equal to zero for the tabulated values [$R(x_i) = 0$].

Using the definition of the Δ operator [Eq. (II.2)] we can write

$$E^\alpha = (1+\Delta)^\alpha = 1 + \frac{\alpha}{1!}\Delta + \frac{\alpha(\alpha-1)}{2!}\Delta^2 + \cdots + \frac{\alpha(\alpha-1)\ldots(\alpha-n+1)}{n!}\Delta^n + \ldots$$

and

$$E^{\alpha} f_0 = f(x_0+\alpha h) = \left(1+\frac{\alpha}{1!}\Delta+\frac{\alpha(\alpha-1)}{2!}\Delta^2+\dots+\frac{\alpha(\alpha-1)\dots(\alpha-n+1)}{n!}\Delta^n+\dots\right) f_0$$
$$= \left(1+\frac{\alpha}{1!}\Delta+\frac{\alpha(\alpha-1)}{2!}\Delta^2+\dots+\frac{\alpha(\alpha-1)\dots(\alpha-n+1)}{n!}\Delta^n\right) f_0 + R(x)\ . \tag{II.8}$$

This formula constitutes Newton forward interpolation. If $\alpha = n$ then $E^n f_0 = f_n$ and $R(x) = 0$, but if $\alpha \neq n$ then $R(x) \neq 0$. From the Taylor expansion we can determine the error function $R(x)$ as:

$$\begin{aligned} R(x) = R(x_0+\alpha h) &= \frac{f^{n+1}(\xi)}{n+1}(x-x_0)(x-x_1)\cdots(x-x_n) \\ &= \frac{f^{n+1}(\xi)}{n+1}h^{n+1}(\alpha-1)(\alpha-2)\cdots(\alpha-n)\ , \end{aligned} \tag{II.9}$$

where ξ is a value located between x_0 and $x_0+\alpha h$. In principle we could estimate the error by maximising $R(x)$. The Newton forward interpolation formula can be used to perform linear interpolation when a tabulated function varies so slowly that first differences are approximately constant. It may be approximated closely by a straight line between adjacent tabular points.

The next simple process is quadratic interpolation. One might expect that such an approximation would give better accuracy for functions with larger variations. In this case

$$E^{\alpha} f_0 \approx f(x_0+\alpha h) = (1+\frac{\alpha}{1!}\Delta+\frac{\alpha(\alpha-1)}{2!}\Delta^2) f_0$$

is a good approximation.

Similarly, we can also derive the so-called Newton backward interpolation formula:

The function f(x) is tabulated as a set of points $[x_i,\ f(x_i)]$, where $i \in \{0, 1, \dots, n-1, n\}$, and $x_i = x_0 + ih$. For any argument x, we can write $f(x) = \Phi_n(x) + R(x)$, where $\Phi_n(x)$ is a polynomial passing by all the tabulated values, and $R(x)$ is a function that is equal to zero for the tabulated values $[R(x_i) = 0]$.

Using the definition of the ∇ operator [Eq. ((II.7))] we can write

$$E^{\alpha} = (1-\nabla)^{-\alpha} = 1+\frac{\alpha}{1!}\nabla+\frac{\alpha(\alpha+1)}{2!}\nabla^2+\dots+\frac{\alpha(\alpha+1)\dots(\alpha+n-1)}{n!}\nabla^n+\cdots$$

and

$$E^{\alpha} f_n = f(x_n+\alpha h) = \left(1+\frac{\alpha}{1!}\nabla+\frac{\alpha(\alpha+1)}{2!}\nabla^2+\dots+\frac{\alpha(\alpha+1)\dots(\alpha+n-1)}{n!}\nabla^n+\cdots\right) f_0$$
$$= \left(1+\frac{\alpha}{1!}\nabla+\frac{\alpha(\alpha+1)}{2!}\nabla^2+\dots+\frac{\alpha(\alpha+1)\dots(\alpha+n-1)}{n!}\nabla^n\right) f_0 + R(x)\ . \tag{II.10}$$

This formula constitutes Newton backward interpolation, and can be used to interpolate the function f at the end of the table. The value α will be chosen negative. It is clear that the interpolating polynomial of order n used in the Newton backward or forward formulas is unique. This is because both polynomials pass exactly through the $n+1$ tabulated values of the function f.

II.1.4 Lagrange Interpolation

In the previous subsection we have discussed Newton's forward and backward interpolation formulas and noted that higher order interpolation corresponds to higher degree polynomial approximation. In practice, however, we only require polynomial of lower degrees and the data are often not tabulated at equal intervals.

The advantage of the Newton formulas is that we can evaluate the differences at any order and find out when they become negligible. This will give the order of the polynomial to be used for the interpolation. In this subsection we consider a more general interpolation formula due to Lagrange, which does not require the function to be tabulated at equal intervals. The drawback of using directly the Lagrange's interpolation formula is that the number of operations to be performed in a numerical program is very large, and that we have to set the degree of the polynomial without any knowledge whether it can perform a good interpolation or not. We will then use an algorithm due to Aitken where the Lagrange interpolation is rewritten in order to minimize the number of operations and to check for errors in the interpolation.

Linear Interpolation

The linear interpolation is adequate when the curvature of the function $f(x)$ is small. In this case we can approximate the function between x_i and x_{i+1} by a straight line:

$$\begin{aligned} f(x) &= f_i + \frac{x-x_i}{x_{i+1}-x_i}(f_{i+1}-f_i) + \Delta f \\ &= \frac{x-x_{i+1}}{x_i-x_{i+1}} f_i + \frac{x-x_i}{x_{i+1}-x_i} f_{i+1} + \Delta f \ . \end{aligned} \tag{II.11}$$

The error Δf is of the order

$$\Delta f = \frac{A}{2}(x-x_i)(x-x_{i+1})$$

and can be estimated using a Taylor expansion of the function $f(x)$,

$$f(x) = f_i + (x-x_i)f'_i + \frac{(x-x_i)^2}{2!} f''(\xi),$$

where $\xi \in [x_i, x]$. The second derivative of $f(x)$ yields the value of A:

$$\left.\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\right|_{x=\xi} = f''(\xi) = \frac{\mathrm{d}^2 \Delta f}{\mathrm{d}x^2} = A \ .$$

We have then to find the value of x for which the error is the largest. This requires that the first derivative of Δf is zero and its second derivative is positive:

$$\max[\Delta f] \iff \left\{ \begin{array}{lll} \dfrac{\mathrm{d}\Delta f}{\mathrm{d}x} = 0 & \Longrightarrow & \dfrac{A}{2}(2x - x_i - x_{i+1}) = 0 \ \Rightarrow \ x = \dfrac{x_i + x_{i+1}}{2} \\ \dfrac{\mathrm{d}^2\Delta f}{\mathrm{d}x^2} > 0 & \Longrightarrow & A > 0 \ . \end{array} \right.$$

This produces a maximum error of

$$\max[\Delta f] = \frac{\max[f(\xi)]}{8}(x_{i+1}-x_i)^2 \ .$$

In practice, however, it is very difficult to evaluate the error because one can not easily estimate the maximum of $f(\xi)$.

Interpolation with a Polynomial of Order n

We have to find a polynomial of order n that passes through all the points of the function $f(x)$. This polynomial can be defined as follows:

$$\Phi_n(x) = \sum_{i=0}^{n} a_i \prod_{j\neq i}(x-x_j) \ .$$

We would like that $\Phi_n(x_i) = f_i$ for any x_i, where $0 \leqslant i \leqslant n$. This is satisfied only if

$$a_i = \frac{f_i}{\prod_{j\neq i}(x_i - x_j)}.$$

The Lagrange polynomial Φ_n is then given by

$$\Phi_n(x) = \sum_{i=0}^{n} \left[\prod_{j\neq i}^{n} \frac{x - x_j}{(x_i - x_j)} \right] f_i = \sum_{i=0}^{n} L_i(x) f_i,$$

where the polynomial L_i is

$$L_i(x) = \prod_{j\neq i}^{n} \frac{x - x_j}{(x_i - x_j)}.$$

It is then clear that

$$L_i(x) = \begin{cases} 0, & x = x_j \neq x_i, \\ 1, & x = x_i. \end{cases}$$

The function $f(x)$ to be interpolated is given by

$$f(x) = \phi_n(x) + R(x),$$

where $R(x)$ is the error polynomial,

$$R(x) = \prod_{j=0}^{n} (x - x_j) \frac{f^{n+1}(\xi)}{(n+1)!}.$$

We can easily check that we recover Newton interpolation formulas in the case where the function is tabulated in equal spacing grid.

Aitken method

The Lagrange interpolation can be done nicely using the Aitken method, where a series of successive linear interpolations are made:

$$f_{i,i+1,\cdots,j} = \frac{x - x_j}{x_i - x_j} f_{i,i+1,\cdots,j-1} + \frac{x - x_j}{x_i - x_j} f_{i+1,i+2,\cdots,j}.$$

The values of $f_{i,i+1,\cdots,j-1}$ and $f_{i+1,i+2,\cdots,j}$ can be obtained by adapting the same formula. We can illustrate this method by performing a four point interpolation. In this case we have three levels of iterative linear interpolations. The first level is given by:

$$f_{1,2} = \frac{x - x_2}{x_1 - x_2} f_1 + \frac{x - x_1}{x_2 - x_1} f_2,$$
$$f_{2,3} = \frac{x - x_3}{x_2 - x_3} f_2 + \frac{x - x_2}{x_3 - x_2} f_3,$$
$$f_{3,4} = \frac{x - x_4}{x_1 - x_3} f_3 + \frac{x - x_3}{x_4 - x_3} f_4.$$

The second level is given by:

$$f_{1,2,3} = \frac{x - x_3}{x_1 - x_3} f_{1,2} + \frac{x - x_2}{x_3 - x_1} f_{2,3},$$
$$f_{2,3,4} = \frac{x - x_4}{x_2 - x_4} f_{2,3} + \frac{x - x_2}{x_4 - x_2} f_{3,4},$$

and the final level of interpolation is given by:

$$f_{1,2,3,4} = \frac{x - x_4}{x_1 - x_4} f_{1,2,3} + \frac{x - x_1}{x_4 - x_1} f_{2,3,4}.$$

This method is illustrated in Fig. II.3.

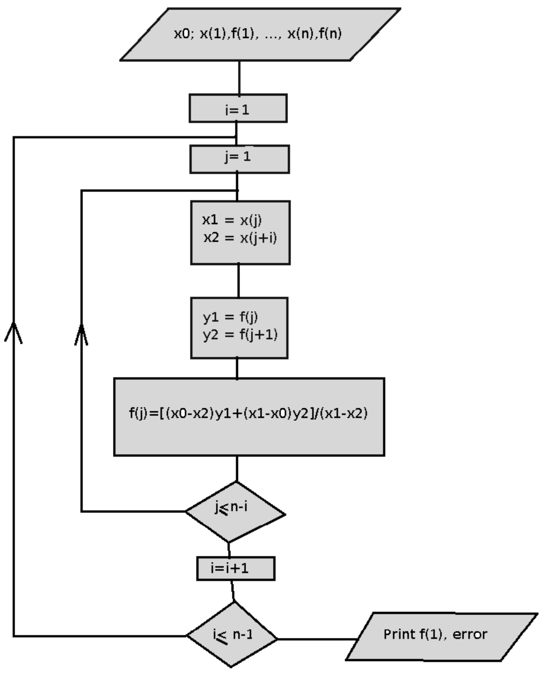


Figure II.3: Flow chart representing n-point interpolation using the Aitken algorithm, where x_0 is the point of interpolation of the function $f(x)$ tabulated for the points $x(1), x(2), ..., x(n)$ and taking the values $f(1), f(2), ..., f(n)$. The maximum error in the interpolation is given by $(|f(1) - y_1| + |f(1) - y_2|)/2$.

II.1.5 Methods of the Least-Square Fit

We would like, in general, to fit the data to find out about its behavior. We may choose a polynomial of order n to fit the data. Let's suppose that the polynomial is given by

$$P_n(x) = \sum_{k=0}^{n} a_k x^k.$$

We would like to determine the coefficients a_k so that the difference $\Delta[a_k]$ between this polynomial and the data is the smallest possible:

$$\Delta[a_k] = \int_a^b (f(x) - P_n(x))^2 \mathrm{dx},$$

where a and b are the limits of the data. In general, the data is tabulated like in the case of the interpolation (see the previous subsection). The previous equation becomes

$$\Delta[a_k] = \sum_{i=0}^{n} (f(x_i) - P_m(x_i))^2.$$

We have then to minimize this expression in order to find the coefficients a_k.

Let's first use orthogonal polynomials $u_k(x)$ so that our least squares function is given by

$$P_m(x) = \sum_{k=0}^{m} a_k u_k(x).$$

The orthogonality condition is given by

$$\int_a^b u_k(x)w(x)u_l(x)\mathrm{d}x = \langle u_k|u_l\rangle = \delta_{k,l}N_k,$$

where w is a weight function that depends on the class of the polynomial u_k, and $\delta_{k,l}$ is the Kronecker symbol,

$$\delta_{k,l} = \begin{cases} 1, & k = l, \\ 0, & k \neq l. \end{cases}$$

Let's give an example of orthogonal polynomials given by the following recursive relation:

$$u_{k+1}(x) = (x - g_k)u_k(x) - h_k u_{k-1}(x),$$

with $u_0(x) = 1$ and $h_0 = 0$. Because of the orthogonality condition of the polynomials u_k:

$$\langle u_k|u_{k+1}\rangle = 0 = \langle u_k|x|u_k\rangle - g_k\langle u_k|u_k\rangle - h_k\langle u_k|u_{k-1}\rangle.$$

This projection on the bra $\langle u_k|$ allows us to determine the coefficient g_k:

$$g_k = \frac{\langle u_k|x|u_k\rangle}{\langle u_k|u_k\rangle}.$$

A similar projection on the bra $\langle u_{k-1}|$ allows us to determine the coefficient h_k:

$$h_k = \frac{\langle u_{k-1}|x|u_k\rangle}{\langle u_{k-1}|u_{k-1}\rangle}.$$

To make the least squares fit we suppose here that $w(x) = 1$. Let's now find out the coefficients a_k that minimize $\Delta[a_k]$. The requirement is that the first partial derivatives of Δ with respect to the coefficients a_k are all zero and the second derivatives are positive:

$$\frac{\partial\Delta[a_k]}{\partial a_j} = 0 \quad \text{and} \quad \frac{\partial^2\Delta[a_k]}{\partial a_j^2} > 0\ .$$

Let's evaluate the first partial first derivatives:

$$\begin{aligned}\frac{\partial\Delta[a_k]}{\partial a_j} &= 2\sum_{i=0}^{n}\frac{\partial P_m(x_i)}{\partial a_j}[P_m(x_i) - f(x_i)] = 2\sum_{i=0}^{n}u_j(x_i)[P_m(x_i) - f(x_i)] \\ &= 2\sum_{i=0}^{n}u_j(x_i)\left[\sum_{k=0}^{m}a_k u_k(x_i) - f(x_i)\right] \\ &= 2\sum_{k=0}^{m}a_k\left[\sum_{i=0}^{n}u_j(x_i)u_k(x_i)\right] - 2\sum_{i=0}^{n}u_j(x_i)f(x_i) \\ &= 2\left(a_j\langle u_j|u_j\rangle - \langle u_j|f\rangle\right) = 0\ . \end{aligned} \tag{II.12}$$

Using this equation we can determine the coefficients a_k that minimize Δ:

$$a_k = \frac{\langle u_k|f\rangle}{\langle u_k|u_k\rangle}\ .$$

II.2 Differentiation

II.2.1 Introduction

Numerical differentiation is based on the Taylor expansion of a function $f(x)$ at the vicinity of a point x_0, where we would like to calculate the derivatives:

$$f(x) = f(x_0) + \frac{(x-x_0)}{1!}f'(x_0) + \frac{(x-x_0)^2}{2!}f''x_0) + \cdots + \frac{(x-x_0)^n}{n!}f^n(x_0) + \cdots + \frac{(x-x_0)^{n+1}}{(n+1)!}f^{n+1}(\xi),$$

where $\xi \in]x_0, x[$. The derivative of a function $f(x)$ at x_k point is given by

$$f'(x) \equiv f'_k = \lim_{\Delta x \longrightarrow 0} \frac{f(x_k + \Delta x) - f(x_k)}{\Delta x}.$$

Let's assume that the function $f(x)$ is tabulated in a equally spaced grid so that $x_k = x_0 + hk$, where $h = x_{k+1} - x_k$. The simplest (not necessarily accurate) numerical derivative is then given by

$$f'_k \simeq \frac{f_{k+1} - f_k}{h} + \mathcal{O}(h).$$

This is called the two-point derivative formula.

II.2.2 First Derivative Formula

Three Points Derivative Formula

We can increase the accuracy of the derivative seen in the introduction by making a Taylor expansion of f_{k+1} and f_{k-1} around f_k:

$$f_{k+1} = f_k + \frac{h}{1!}f'_k + \frac{h^2}{2!}f''_k + \frac{h^3}{3!}f^3_k + \cdots$$

$$f_{k-1} = f_k - \frac{h}{1!}f'_k + \frac{h^2}{2!}f''_k - \frac{h^3}{3!}f^3_k + \cdots .$$

By subtracting the first equation from the second one we obtain

$$f_{k+1} - f_{k-1} = 2hf'_k + \frac{h^3}{3}f^3_k + \cdots .$$

This equation gives the derivative f'_k to a second order $\mathcal{O}(h^2)$:

$$f'_k \simeq \frac{f_{k+1} - f_{k-1}}{2h}. \tag{II.13}$$

Five-Point-Derivative Formula

We can increase further the accuracy of the derivative seen in the last subsection. By making a Taylor expansion of f_{k+2} to f_{k-2} around f_k and by subtraction of f_{k+1} from f_{k-1} and f_{k+2} from f_{k-2} we obtain

$$f_{k+1} - f_{k-1} = 2hf'_k + \frac{h^3}{3}f^3_k + \mathcal{O}(h^5)$$

and

$$f_{k+2} - f_{k-2} = 4hf'_k + \frac{8h^3}{3} f^3_k + \mathcal{O}(h^5).$$

By multiplying the first equation by 8 and subtracting it from the second one we obtain the derivative f'_k to a fourth order $\mathcal{O}(h^4)$:

$$f'_k \simeq \frac{1}{12h}(f_{k-2} - 8f_{k-1} + 8f_{k+1} - f_{k+2}). \tag{II.14}$$

Notice that it is not always a good idea to include more than five points in the derivative formula because of the complicated nature of the function to be derived. Often it is a better idea to reduce the step h and keep a five point-formula.

Taking a derivative at the extremities of the interval requires a special treatment. To get the first derivative at the beginning of the interval, we perform a Taylor expansion of the functions f_1 and f_2. The derivative f'_0 is then obtained by multiplying the Taylor expansion of f_2 by four and subtracting it from that of f_1. This gives:

$$f'_0 = \frac{1}{2h}(-3f_0 + 4f_1 - f_2) + \mathcal{O}(h^3).$$

A similar expansion can be done to get a derivative at the end of the interval. One obtains:

$$f'_n = \frac{1}{2h}(3f_n - 4f_{n-1} - f_{n-2}) + \mathcal{O}(h^3).$$

II.2.3 Second-Derivative Formula

Three-Point Second-Derivate Formula

To obtain the second derivative of a function $f(x)$ at a general point x_k we make a Taylor expansion of f_{k-1} and f_{k+1} up to the fourth order around f_k. To get the second derivative we add both Taylor expansions and obtain

$$f''_k = \frac{1}{h^2}(f_{k+1} - 2f_k + f_{k-1}) + \mathcal{O}(h^4).$$

Five-Point Second-Derivate Formula

The computation of the second derivative using a five point formula requires a similar Taylor expansion, but this time one has to make Taylor expansions of $f_{k\pm1}$ and of $f_{k\pm2}$ up to the sixth order around f_k. To obtain the second derivative one has to add the expansions of f_{k-1} and f_{k+1} and multiply the result by 16. Then, one has to subtract the sum of f_{k-2} and f_{k+2}. One obtains:

$$f''_k = \frac{1}{12h^2}(-f_{k-2} + 16f_{k-1} - 30f_k + 16f_{k+1} - f_{k+2}) + \mathcal{O}(h^4).$$

Note: If the grid is not regular then one has to perform first an interpolation in a regular grid and then take the derivatives as shown.

Exercise

To find out about the precision of numerical derivatives it will be helpful to take the derivatives of analytical functions, like $\sin(x)$ or $\cos(x)$, and then check against the exact derivatives. Calculate and compare the errors obtained using three and five point formulas. Do not forget to calculate the derivatives at the beginning and the end of the interval.

II.3 Integration

II.3.1 Introduction

Integration is heavily used in physics. As we stated in the introduction of this course, it is very seldom that we can solve analytically a real problem in physics without making severe approximations which are generally not justified. We rely therefore very often on numerical methods to provide solutions to physical problems. In this chapter, we will learn how to numerically integrate functions in one or several variables. In general, multidimensional integrals can be always solved by iterative schemes, i.e., by integrating over one dimension at the time. Let's assume that we would like to perform a two dimensional integral of a function $f(x,y)$ over a surface domain Σ. We can always first integrate over one variable, say the variable y, and then over x. Let's assume that the surface Σ is simple. If it is not, then we can always cut it into small simple domains and add up all the integrals to get the final answer. Let's assume again that the domain is simple convex between a and b along the x-axis and lying between the two functions $g_1(x)$ and $g_2(x)$. The two dimensional integral can then be written as

$$I = \int\int_{\Sigma} f(x,y)\mathrm{d}x\mathrm{d}y = \int_a^b \mathrm{d}x \int_{g_1(x)}^{g_2(x)} f(x,y)\mathrm{d}y.$$

If we define $G(x) = \int_{g_1(x)}^{g_2(x)} f(x,y)\mathrm{d}y$ then the integral I is given by:

$$I = \int_a^b G(x)\mathrm{d}x.$$

It is then clear that we will first calculate $G(x)$ as a one dimensional integral for all values of x between a and b and then integrate the resulting function $G(x)$ as a one dimensional integral between a and b. This method of integration of multidimensional integrals is called an iterative integral scheme because we iterate over all dimensions of the domain by performing one dimensional integrals at a time. It is therefore important to learn how to perform accurately a one-dimensional integral. This is the object of this chapter.

II.3.2 One Dimensional Numerical Integration

Let's suppose that we would like to evaluate the integral of the function $f(x)$ between a and b:

$$I = \int_a^b f(x)\mathrm{d}x.$$

Let's divide the interval $[a,b]$ into n subintervals of equal length. We define then a step $h = (b-a)/n$ and $x_k = a + kh$, where k is an integer between zero and n. Using the Riemann additivity of integrals we write

$$I = \int_a^b f(x)\mathrm{d}x = \sum_{k=0}^{n-1} \int_{x_k}^{x_{k+1}} f(x)\mathrm{d}x.$$

Linear Approximation or Trapezoidal Method

In a small interval $[x_k, x_{k+1}]$ we suppose that the function does not vary too much so that it can be approximated by a linear function. If this is not the case, we can always reduce the interval by increasing the value of n to make the linear approximation possible. Sometimes this is not a good idea because the computational time increases drastically. Usually it is not the integral that is expensive but obtaining the numerical values of the function. We may need a sophisticated and expensive method to get the function $f(x)$ in a single point x, and the function may also depend on many other parameters. This is usually the case for multidimensional functions. Everything being said, let's assume that we have a small enough interval that the linear approximation of the function $f(x)$ is possible:

$$f(x) = f(x_k) + \frac{x - x_k}{h}\,(f_{k+1} - f_k).$$

The integral of the function $f(x)$ in the interval $[x_k, x_{k+1}]$ is given by the area of the trapeze:

$$\int_{x_k}^{x_{k+1}} f(x)dx = \frac{h}{2}\,(f_{k+1} + f_k).$$

We then have to add up all the subinterval contributions to obtain

$$I = \int_a^b f(x)dx = \frac{h}{2}\sum_{k=0}^{n-1}(f_{k+1} + f_k) = h\left(\frac{1}{2}f_0 + f_1 + f_2 + \ldots + f_{n-1} + \frac{1}{2}f_n\right) .$$

Before we proceed further, it is important to estimate the error of the trapezoidal method. In computational physics errors can lead to inaccurate results or even wrong results if the errors due to numerical methods are not well controlled. To estimate the error, we expand the function f(x) up to the second order. Higher order terms are negligible if the step of integration is sufficiently small. Let's assume that the second order term of the function is called $R(x)$, and the linear term used to approximate the function in a given interval is $\phi(x)$, given by the above expression. Let's find the error for the integral between x_k and x_{k+1}:

$$f(x) = \phi(x) + R(x) .$$

It is then obvious that $R(x)$ should be given by

$$R(x) = (x - x_k)(x - x_{k+1})\frac{f''(\xi_k)}{2},$$

where $\xi_k \in [x_k, x_{k+1}]$. This expression of $R(x)$ can be obtained by using a quadratic form and setting $R(x_k) = R(x_{k+1}) = 0$ because $f(x_k) = \phi(x_k)$ and $f(x_{k+1}) = \phi(x_{k+1})$. We get

$$\int_{x_k}^{x_{k+1}} f(x)\mathrm{d}x = \int_{x_k}^{x_{k+1}} \phi(x)\mathrm{d}x + \int_{x_k}^{x_{k+1}} R(x)\mathrm{d}x.$$

The first term is the estimated integral given by the trapezoidal method while the latter term is an estimation of the error. Using the expression of $R(x)$ one can obtain by a straight integration that the error is given by

$$\Delta I_k = \int_{x_k}^{x_{k+1}} R(x)\mathrm{d}x = -\frac{f''(\xi)}{12}h^3.$$

The total error of the integral over the interval $[a, b]$ is obtained by adding all errors over the intervals $[x_k, x_{k+1}]$ for $k = 0$ to $k = n - 1$:

$$\Delta I = -\sum_{k=0}^{n-1}\frac{f''(\xi_k)}{12}h^3 = -\frac{nh^3}{12}f''(\xi),$$

where $f''(\xi)$ is the average value of f'' over the interval. To maximize the error we use the maximum value of f'' in $[a, b]$. Let's call this maximum $f''_{max}(\xi)$. The error is then given by

$$\Delta I \leq -\frac{(b-a)h^2}{12}f''(\xi),$$

where we have substituted n by $(b - a)/h$. We can see that the error is proportional to the size of the interval $[a, b]$ and decreases quadratically with interval step size.

Quadratic Approximation or the Simpson Method

Similar to the linear approximation, here we will approximate the function $f(x)$ by a second-order Lagrange polynomial ϕ_k in an interval $[x_{k-1}, x_{k+1}]$:

$$f(x) = \frac{(x-x_k)(x-x_{k+1})}{(x_{k-1}-x_k)(x_{k-1}-x_{k+1})}f_{k-1}+\frac{(x-x_{k-1})(x-x_{k+1})}{(x_k-x_{k-1})(x_k-x_{k+1})}f_k+\frac{(x-x_{k-1})(x-x_k)}{(x_{k+1}-x_{k-1})(x_{k+1}-x_k)}f_{k+1}+\mathcal{O}(h^3). \tag{II.15}$$

We have then to evaluate the integral of the Lagrange polynomial that fits the function $f(x)$ in the interval $[x_{k-1}, x_{k+1}]$:

$$I_k = \int_{x_{k-1}}^{x_{k+1}} f(x)\mathrm{d}x \approx \int_{x_{k-1}}^{x_{k+1}} \phi_k(x)\mathrm{d}x.$$

It is strait-forward to integrate the second-order Lagrange polynomial to find

$$I_k \approx \int_{x_{k-1}}^{x_{k+1}} \phi_k(x)\mathrm{d}x = \frac{h}{3}(f_{k-1}+4f_k+f_{k+1}).$$

If the division n of the interval is even the integral of the function in the interval $[a,b]$ reads

$$I = \int_a^b f(x)\mathrm{d}x \approx \frac{h}{3}\sum_{k=0}^{\frac{n}{2}-1}(f_{2k}+4f_{2k+1}+f_{2k+2}) = \frac{h}{3}(f_0+4f_1+2f_2+\ldots+2f_{n-2}+4f_{n-1}+f_n).$$

If n is odd we must add up the last subinterval. We will integrate the Lagrange polynomial written down in the interval $[x_{n-2}, x_n]$ from x_{n-1} to x_n. This produces the following result:

$$I_n = \int_{x_{n-1}}^{x_n} \phi_{n-1}(x)dx \approx \frac{h}{12}(-f_{n-2}+8f_{n-1}+5f_n).$$

To derive the error to the Simpson formula can be quite cumbersome, but we will show in the following that we can use the linear operators defined in the interpolation chapter to perform integration of high degrees and determine easily the error of integration.

Let's suppose we would like to integrate a function $f(x)$ in the interval $[x_0, x_n]$ using a polynomial of degree n. For a general value of n these integrals are known as the Newton-Cotes formulas. If the values x_0 and x_n are included in the calculation then the the formulas are known as closed Newton-Cotes formulas. However, if the first value of x is used to be $x_0 = a+\frac{h}{2}$ and the last $x_n = b-\frac{h}{2}$ the formulas are known as open Newton-Cotes formulas, and that is because they do not include the borders of the interval $[a,b]$. To show how we can get these formulas we will define the integral in terms of linear difference operators

$$I_n = \int_{x_0}^{x_n} f(x)\mathrm{d}x = (E^n-1)D^{-1}f(x_0) = F(x_n)-F(x_0),$$

where F is the primitive of the function f. We can see that the D^{-1} operator produces the primitive of the function f at the point x_0. Then the operator E^n-1 produces $F(x_n)-F(x0)$ which is equal to $F(x_0+nh)-F(x_0)$. Now the whole point is to transform $(E^n-1)D^{-1}$ in terms of the Δ operator and use the Taylor expansion to produce the result of the above integral directly. Let's first make the transformation in terms of the operator Δ. We know that $E=e^{hD}$ (see the interpolation chapter), so that $D=h^{-1}\ln(E)$ and since $E=\Delta+1$, the integral I_n becomes

$$I_k = \int_{x_0}^{x_n} f(x)\mathrm{d}x = h\frac{((1+\Delta)^n-1)}{\ln(1+\Delta)}f(x_0).$$

Now let's make a Taylor expansion of $(1+\Delta)^n$ and $\ln(1+\Delta)$

$$(1+\Delta)^n = 1 + \frac{n}{1!}\Delta + \frac{n(n-1)}{2!}\Delta^2 + \dots + \frac{n(n-1)\dots(n-j+1)}{j!}\Delta^j + \dots$$

and

$$\ln(1+\Delta) = \Delta + \frac{-1}{2!}\Delta^2 + \dots + \frac{1}{3}\Delta^3 + \frac{-1}{4}\Delta^4 \dots .$$

The integral I_n is then given by

$$I_n = \int_{x_0}^{x_n} f(x)\mathrm{d}x = h\frac{n + \frac{n(n-1)}{2!}\Delta + \dots + \frac{n(n-1)\dots(n-j+1)}{j!}\Delta^{j-1} + \dots}{1 - \frac{1}{2!}\Delta + \dots + \frac{1}{3}\Delta^2 - \frac{1}{4}\Delta^3 \dots} f_0.$$

The polynomial devision can be carried out and one obtains

$$I_n = h(1 + \frac{n}{2}\Delta + \frac{n(2n-3)}{12}\Delta^2 + \frac{n(n-2)^2}{24}\Delta^3 + \frac{n(6n^6 - 45n^2 + 110n - 90)}{720}\Delta^4 + \dots .$$

From this expression we can get all kind of approximations to the integral I_n of the function f. For example, to obtain the trapezoidal formula we set $n = 1$, and to get the Simpson formula we set $n = 2$. We will provide integration results for n up to 6. The results of the integrals are known under the name of closed Newton-Cotes formulas, because they contain the tabulated values f_0 and f_n:

$n = 2$ (Simpson formula):

$$I_2 = \frac{h}{3}(f_0 + 4f_1 + f_2) - \frac{h^5}{90}f^{(4)}(\xi)$$

$n = 3$ (Second Simpson formula):

$$I_3 = \frac{3h}{8}(f_0 + 3f_1 + 3f_2 + f_3) - \frac{3h^5}{80}f^{(4)}(\xi)$$

$n = 4$:

$$I_4 = \frac{2h}{45}(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4) - \frac{8h^7}{945}f^{(6)}(\xi)$$

$n = 5$:

$$I_5 = \frac{5h}{288}(19f_0 + 75f_1 + 50f_2 + 50f_3 + 75f_4 + 19f_5) - \frac{275h^7}{12096}f^{(6)}(\xi)$$

$n = 6$:

$$I_6 = \frac{h}{140}(41f_0 + 216f_1 + 27f_2 + 272f_3 + 27f_4 + 216f_5 + 41f_6) - \frac{275h^7}{12096}f^{(6)}(\xi).$$

Similarly, we can show that there are Newton-Cotes formulas that do not use the borders f_0 and f_n of the tabulated function f. These formulas are known as open Newton-Cotes formula. We give these formula without demonstration from $n = 2$ to $n = 6$:

$n = 2$:

$$I_2 = 2hf_1 + \frac{h^3}{3}f^{(2)}(\xi)$$

$n = 3$:

$$I_3 = \frac{3h}{2}(f_1 + f_2) + \frac{3h^3}{4}f^{(2)}(\xi)$$

$n = 4$:

$$I_4 = \frac{4h}{3}(2f_1 - f_2 + 2f_3) + \frac{14h^5}{45}f^{(4)}(\xi)$$

$n = 5$:

$$I_5 = \frac{5h}{24}(11f_1 + f_2 + f_3 + 11f_4) - \frac{95h^5}{144}f^{(4)}(\xi)$$

$n = 6$:

$$I_6 = \frac{3h}{10}(11f_1 - 14f_2 + 26f_3 - 14f_4 + 11f_5) - \frac{41h^7}{140}f^{(6)}(\xi).$$

Note that in general it is not a good idea to use higher polynomials to integrate functions because even though the polynomial passes by all the tabulated points in the interval it does not represent the function correctly. It will often oscillate around its tabulated values. Note also that the open Newton-Cotes formulas can be used if the function f has singularities. One can always integrate the function between the singularities using those formulas. To integrate smooth functions accurately we will show in the next paragraph that there are other powerful methods known as quadratures. We will present in this chapter one of the most used quadrature, known as Gauss-Legendre quadrature.

II.3.3 Gauss-Legendre Quadrature

Before providing any details about the Gauss-Legendre quadrature, let's first show that any kind of integral between a and b can transformed into an integral between -1 and $+1$. This can always be achieved by a linear change of the variable:

$$t = -1 + 2\frac{(x-a)}{(b-a)}.$$

If another more appropriate non-linear change of the variable can make the function smoother it will improve the accuracy of the method. In the following we will assume that the integration of f is between -1 and $+1$. If not then we can always make the previous change of the variable to insure that this is the case. Let's assume that we would like to compute the following integral:

$$I = \int_{-1}^{+1} f(x)\mathrm{d}x.$$

We can always express I as linear combination of the tabulated function f:

$$I = \sum_{k=0}^{N-1} w_k f(x_k).$$

Until now, this is just as the case of the Newton-Cotes formula if $x_k = x_0 + kh$. For example, for Simpson's rule, $N = 3$, and

$$x_0 = -1,\ x_1 = 0,\ x_2 = 1; w_0 = w_2 = \frac{1}{3}, w_1 = \frac{4}{3}.$$

It is clear that if the points of the interval are equally-spaced we can always choose the N weights w_k to integrate a polynomial of degree $N-1$ exactly. One has just to choose the N weights to satisfy the N following linear equations

$$I = \int_{-1}^{+1} x^p \mathrm{d}x = \sum_{k=0}^{N-1} w_k x_k^p$$

We can, however, achieve a greater precision for the same work if we give up the requirement of equally-spaced interval points. We will choose the points x_k in some optimal sense with the constraint that they lie within the $[-1,+1]$ interval. We then have $2N$ intervals at our disposal in constructing the quadrature formula (N parameters for the weights and N others for the positions of the interval points). We can therefore choose them so that we can integrate polynomials of degree $2N-1$. This choice is more powerful than using equally-spaced points. We have increased the accuracy of the method without any increase of the amount of numerical work. We can always solve a system of $2N-1$ equations to find all the weights and the non-equally-spaced interval points, however, there are more interesting methods such as, for example, the Gauss-Legendre quadrature. In this method we will use Legendre polynomials, which are orthogonal in the interval $[-1,+1]$:

$$\int_{-1}^{+1} P_i(x)P_j(x)\mathrm{d}x = \frac{2}{2i+1}\delta_{i,j}.$$

It can be easily shown that the Legendre polynomial P_i is of degree i and has i roots in the interval $[-1,+1]$. Any polynomial F of degree $2N-1$ can be written in the form

$$F(x) = Q(x)P_N(x) + R(x).$$

The exact value of the integral of F is given by

$$\int_{-1}^{+1} F(x)\mathrm{d}x = \int_{-1}^{+1} [Q(x)P_N(x) + R(x)]\mathrm{d}x = \int_{-1}^{+1} R(x)\mathrm{d}x.$$

This integration follows from the orthogonality of the polynomial P_N to all polynomial of degree $p < N$. If we take the x_k's to be the zeros of P_N the numerical integral of F reads

$$\int_{-1}^{+1} F(x)\mathrm{d}x = \sum_{k=0}^{N-1} w_k(Q(x_k)P_N(x_k) + R(x_k)) = \sum_{k=0}^{N-1} w_k R(x_k).$$

Now we have to choose the weights w_k so that a polynomial of degree $N-1$ is exact. It can be shown that the weights w_k are given by

$$w_k = \frac{2}{(1-x_k^2)P_N'^2(x_k)},$$

where P_N' is the first derivative of the Legendre polynomial P_N.
In conclusion, to integrate any function f, one has first to calculate this function for all the values of the roots of the Legendre polynomial of degree N and then choose the weights w_k as specified by the above formula.
Other Gaussian quadrature can be found by choosing different kind of orthogonal polynomials. For example, the Laguerre polynomials, L_N, are orthogonal in the interval $[0,\infty]$. With the weight function e^{-x}, we get to the Gauss-Laguerre quadrature formula:

$$I = \int_{-1}^{+1} e^{-x} f(x)\mathrm{d}x = \sum_{k=0}^{N-1} w_k f(x_k),$$

where the x_k's are the roots of L_N. The weights w_k are given by

$$w_k = \frac{(n!)^2}{L_{N+1}(x_k)L_N'(x_k)}.$$

For other Gaussian quadratures or to learn more, we encourage the reader to look up the numerical analysis literature.

II.4 Roots and Extrema of a Function

II.4.1 Introduction

Finding the roots of a function $F(x) = 0$ is a common problem in physics. We encounter problems dealing with finding roots of polynomials or non linear transcendental equations. Most often, there are no exact solutions, and one has to resort to numerical solutions. We start with an approximate trial solution, and proceed by iterations to improve it. To avoid complexity, we will deal first with one dimensional equations. We will first start with the so called iterative method, then we study the bisection, the Newton-Raphson and the iteration methods.

II.4.2 Iterative Method

If r is the root of $F(x) = 0$ and if we can rewrite this equation in the form $x = f(x)$ in such a way that $|f'(x)| < A < 1$ in an interval centered around the root r, then the sequence $x_n = f(x_{n-1})$ is such that $\lim_{n\to\infty} x_n = r$.

Proof:

For any couple $(x, y) \in I$, $|f(x) - f(y)| = |f'(\xi)||x - y| < A|x - y|$. It is then clear that $|x_n - r| = |f(x_{n-1}) - f(r)| < A|x_{n-1} - r| ... < A^n|x_1 - r|$. Since $A < 1$, it is clear then that $\lim_{n\to\infty} |x_n - r| = 0$, and therefore $\lim_{n\to\infty} x_n = r$.

The convergence is shown in an example in Fig.II.4. The choice of $f(x_{n-1})$ as the next x_n amounts to the following one of the horizontal line segments over to the line $y = x$.

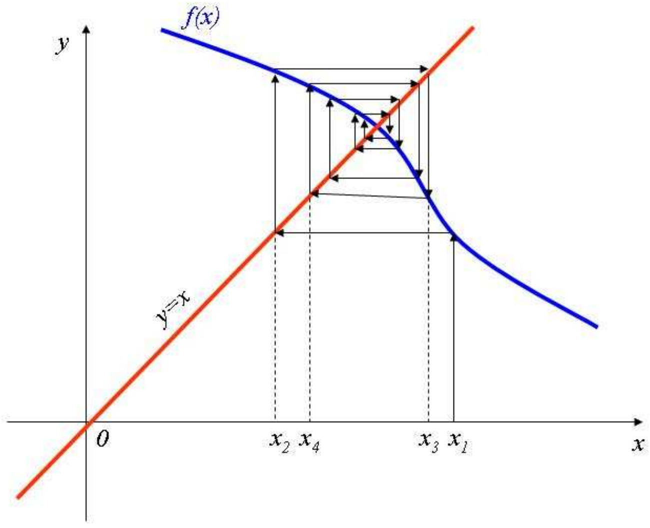


Figure II.4: Representation of the iteration method.

Example: In the year 1225 Leonardo of Pisa found that one of the roots of $F(x) = x^3 + 2x^2 + 10x - 20$ is $x \approx 1.368808107$. It seems that nobody knows how he got this approximate result.
In fact, this function can be put into the form $x = f(x)$ in many ways, for example:

$$x = \frac{20}{x^2 + 2x + 10}.$$

Write a Fortran program using the iteration method to find Leonardo's solution to a precision of 9 digits. You can start from an approximate trial solution, like $x = 1.3$. Try other starting points. You will experience that the convergence of the algorithm is very slow.
In fact, let's suppose that r is the exact solution, and x_n is an approximate one, obtained at n-th iteration. The error e_n is given by: $e_n = |r - x_n| = |f(r) - f(x_{n-1})| = |f'(\xi)||r - x_{n-1}| \approx f'(r)e_{n-1}$. As n is already large, it was appropriate to approximate $f'(\xi)$ by $f'(r)$. We can see that the error at iteration n is proportional to that of iteration $n - 1$, making the convergence of the algorithm very slow.
We can accelerate the algorithm as follows. We start from the fact that:

$$r - x_{n+1} = f'(r)(r - x_n)$$

and

$$r - x_{n+2} = f'(r)(r - x_{n+1}) .$$

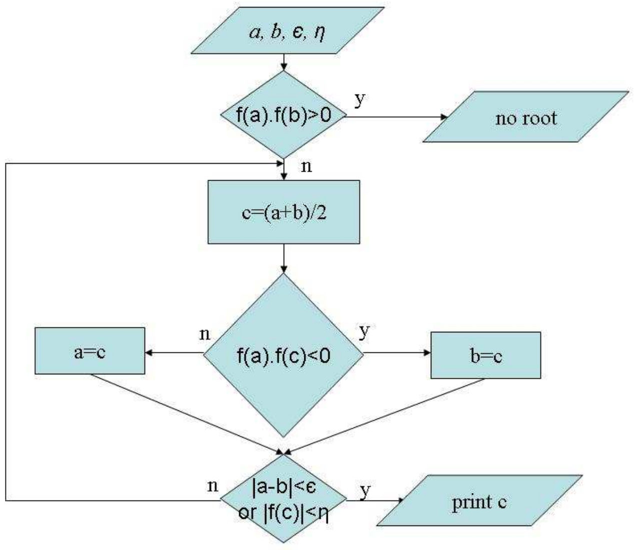


Figure II.5: Flow chart of the bisection method.

From these two equations we can determine r:

$$r = x_{n+2} - \frac{(x_{n+2} - x_{n+1})^2}{x_{n+2} - 2x_{n+1} + x_n}.$$

This is called the Aitken acceleration to the iterative algorithm.
Combine this acceleration scheme with the previous algorithm, and show that you can converge faster to the Leonardo's solution.
Hint: After few iterations using the iteration algorithm, calculate x_n, x_{n+1}, and x_{n+2} and inject them into the Aitken acceleration.

In conclusion, one has to be very careful with the iteration method, because if $f'(x) > 1$ near the root, the iterative method may diverge or may oscillate without converging.
Try to use the previous algorithm to find the square root of a number, say 2. You will find out that you get nowhere. This is because the derivative of $F(x)$ is greater than one on the left side of the root.

II.4.3 Bisection Method

This method is often referred to as the method of divide and conquer method. One has to choose an interval containing the solution of $f(x) = 0$. Let's call the interval containing the root $[a, b]$. We first define the point $c = (a + b)/2$. if $f(a)f(c) < 0$, the solution lies in the region $[a, c] = [a, (a + b)/2]$, we then replace b by c and start over again by dividing the newest interval. If $f(a)f(c) > 0$ the solution lies in the interval $[c, b] = [(a + b)/2, b]$. In this case, we replace a by c and start all over again with this newest interval until we reach a value of c such that $f(c) \approx 0$ to given numerical precision.
The algorithm is implemented in Fig. II.5. We have to define the values of ϵ (precision of the final interval), η (precision of the root), as well as the number of iterations in advance to avoid bad surprises, because the method may converge very slowly or not at all.

After n divisions of the interval by 2 the solution will be given by:

$$r = x_n \pm \frac{b - a}{2^{n+1}},$$

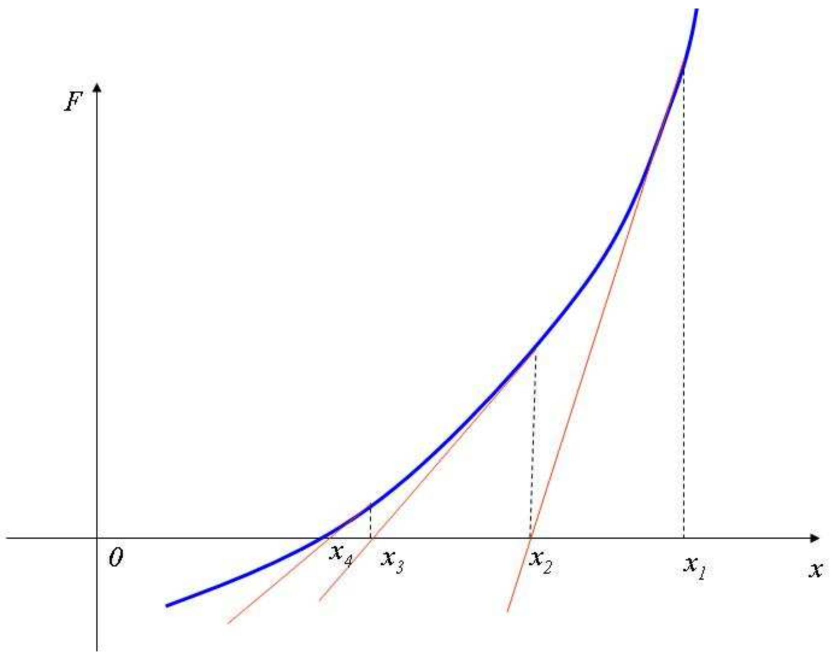


Figure II.6: Geometrical representation of the Newton-Raphson method.

where x_n is the approximate solution at the n-th iteration. The interval containing the solution has a length of $|b-a|/2^n$.

Exercise: Write a Fortran program to determine the square root of 2 with a relative error of 10^{-6}.

II.4.4 Newton-Raphson Method

This is one of the most used methods, because when it works it converges very fast to the solution. It requires the calculation of both, $f(x)$ and its first derivative $f'(x)$, at an arbitrary point x. This method is efficient even when the function is only known numerically, and one has to calculate the derivative numerically.
This method consists of extending the tangent line at a current point until it crosses zero, then setting the next guess to that zero-crossing and start extending the next tangent line at this new point (see Fig. II.6). At the n-th iteration we can show that

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}. \tag{II.16}$$

To show this, let's start with a Taylor expansion:

$$f(r) = f(x_{n-1}) + (r - x_{n-1})f'(x_{n-1}) + \frac{1}{2}(r - x_{n+1})^2 f'(\xi).$$

For small values of f close to the solution, we retain only the linear part of the expansion and we set r in the right part of the expansion to x_n, and set $f(r) = 0$. We obtain then:

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}.$$

The Newton-Raphson (NR) method can give inaccurate results or never converge if one starts far from the root. This behavior can happen if between the starting point and the root there is a local minimum or maximum of the function (see Fig. II.7).

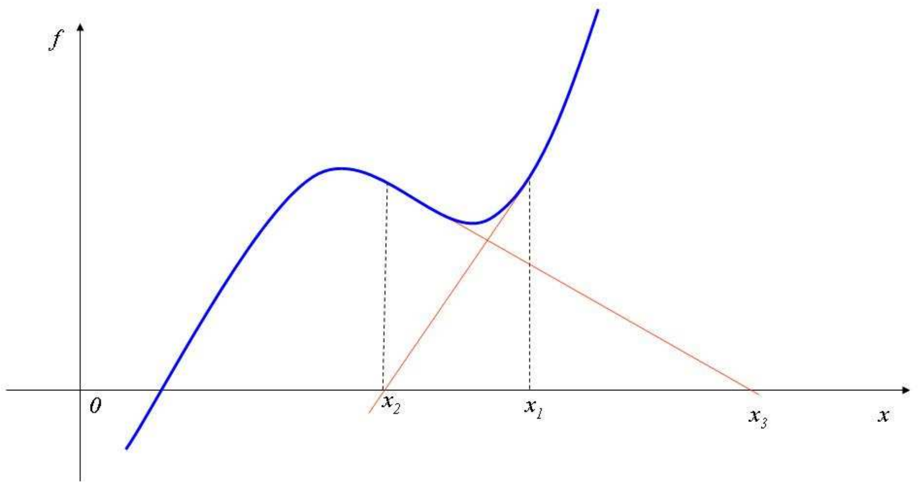


Figure II.7: Case where the Newton-Raphson method never converge.

Let's use this Newton-Raphson algorithm to find the bound states of a particle in a box in one dimension. To simplify the Schroedinger equation (SE), let's put $\hbar^2/2m = 1$. The Schrödinger equation in one dimension is given by

$$\left(-\frac{\mathrm{d}^2}{\mathrm{d}x^2} + V(x)\right)\phi(x) = E\phi(x),$$

where

$$V(x) = \begin{cases} -V_0 & 0 \le x \le a \\ 0 & |x| > a \end{cases}$$

represents the box potential. With the help of this potential the SE becomes:

$$\begin{cases} \left(\frac{\mathrm{d}^2}{\mathrm{d}x^2} + (V_0 + E)\right)\varphi(x) = 0, & \text{if } 0 \le x \le a \\ \left(\frac{\mathrm{d}^2}{\mathrm{d}x^2} + E\right)\varphi(x) = 0, & \text{if } \quad |x| > a. \end{cases}$$

The bound states for which $E < 0$ are given by:

$$\begin{cases} \varphi(x) = A\sin\left(\sqrt{V_0 - |E|}\,x\right), & \text{if } 0 \le x \le a \\ \varphi(x) = B\mathrm{e}^{-\sqrt{|E|}x}, & \text{if } 0 \le x \le a. \end{cases}$$

To make sure that the solution is differentiable at $x = a$, one has to ensure that the logarithmic derivative at $x = a$ is continuous:

$$\frac{1}{\varphi(a^-)}\frac{\mathrm{d}\varphi}{\mathrm{d}x}(a^-) = \frac{1}{\varphi(a^+)}\frac{\mathrm{d}\varphi}{\mathrm{d}x}(a^+).$$

This leads to:

$$\sqrt{V_0 - |E|}\cot\left(\sqrt{V_0 - |E|}\,a\right) = -\sqrt{|E|}.$$

To find the energy of the bound state, one has to solve this non linear equation using the Newton-Raphson algorithm.

Exercise: Write a Fortran program that can produce the energy of the bound state of quantum well of height $V_0 = 3$ eV and a size $a = 10$ Å.

We can explain the fast convergence of the NR algorithm by showing that the convergence is quadratic. In fact, let's use a Taylor expansion around the root of $f(x)$:

$$f(r) = f(x_{n-1}) + (r - x_{n-1})f'(x_{n-1}) + \frac{1}{2}(r - x_{n-1})^2 f''(\xi).$$

Because we are close to the root, we may approximate r by x_n on the right side, and can neglect the second order contribution, so that

$$0 = f(x_{n-1}) + (x_n - x_{n-1})f'(x_{n-1}).$$

If we subtract these two last equations, and knowing that $f(r) = 0$ we obtain:

$$0 = (r - x_n)f'(x_{n-1}) + \frac{1}{2}(r - x_{n-1})^2 f''(\xi).$$

If we define the error as $e_n = r - x_n$ then we obtain:

$$e_n = -\frac{1}{2}\frac{f''(r)}{f'(r)}e_{n-1}^2.$$

The error at the n-th iteration is then proportional to the square of the error at the $n-1$-th iteration. The NR method may not converge if the starting point is far away from the root. However, we can always use a slow algorithm that is full proof to approximate the root and use NR method to polish it.

Exercise: Show that the following algorithm, which gives the square root of a number A, is just a special case of the NR algorithm:

$$x_n = \frac{1}{2}\left(x_{n-1} + \frac{A}{x_{n-1}}\right).$$

Write a program that produces the square root of 2 using this algorithm. Can you generalize this algorithm to calculate the n-th root of 2?

II.4.5 Interpolation Method (Regula Falsi)

We approximate the real function $f(x)$ by a linear function:

$$y = f(a) + \frac{f(a) - f(b)}{a - b}(x - a)$$

so that $y(a) = f(a)$ and $y(b) = f(b)$. We then find the point c, where the linear function y is zero. By making a linear interpolation, we determine easily the point where this function passes zero:

$$c = a - \frac{(a - b)}{f(a) - f(b)}f(a).$$

The algorithm can continue like in the bisection method, but with a clear advantage. It approaches the root much faster. Hence, the convergence is much better than that of the bisection method, but not as good as the NR algorithm.
We can also show the same formulas by combining the NR method and by calculating the derivative as if the function is linear in the interval $[x_{n-1}, x_n]$ by writing the derivative as:

$$f'(x_n) = \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}},$$

so that

$$\begin{aligned} x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} &= x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \\ &= \frac{f(x_n)x_{n-1} - f(x_{n-1})x_n}{f(x_n) - f(x_{n-1})}. \end{aligned}$$

This method is also called the secant method.

Instead of a linear function, we could also use a second order polynomial, by choosing a point c at the middle of the interval $[a, b]$. We can use the Lagrange interpolation to find the coefficient of the polynomial that passes by the three points. One has then to find the zero of the polynomial in the $[a, b]$ interval. Let's call x_0 the point where the polynomial is zero. We have then to find the sign of $f(a)f(x_0)$ like in the bisection algorithm and check on which side is the root, starting over again with a newest second order polynomial.
In fact, given the three points: (x_0, y_0), (x_1, y_1), and (x_2, y_2), where $y = f(x)$, the Lagrange second degree polynomial is given by:

$$p(x) = \frac{x_1 - x_0}{x_2 - x_0}\left(A_2 h^2 + A_1 h + A_0\right),$$

where $h = x - x_2$ and the A_i coefficients are given by:

$$\begin{aligned}
A_2 &= \frac{(x_1 - x_0)\, y_2 + (x_0 - x_2)\, y_1 + (x_2 - x_1)\, y_0}{(x_2 - x_1)\,(x_1 - x_0)^2} \\
A_1 &= \frac{(x_1 - x_0)\,(2x_2 - x_1 - x_0)\, y_2 - (x_2 - x_1)^2\, y_1 + (x_2 - x_1)^2\, y_0}{(x_2 - x_1)\,(x_1 - x_0)^2} \\
A_0 &= \frac{x_2 - x_0}{x_1 - x_0} y_2
\end{aligned}$$

We can then solve this second order polynomial for h to find :

$$h = -\frac{2A_0}{A_1 \pm \sqrt{A_1^2 - 4A_2A_0}}.$$

We will then take the solution with the sign that makes the denominator larger in absolute value so that we take the root which is between x_0 and x_2. The next approximation of the root of $f(x)$ is then given by

$$x_3 = x_2 + h.$$

. We repeat the process again with all subscripts advanced by one. This method is known as the Muller's method, and can be used even to find the solution of complex functions. In this case, we have to make sure that all variables and constants are declared as complex numbers.

Extremes of a Single-Variable Function

We can develop numerical algorithms to determine the maximum or the minimum of a function $f(x)$ based on one of the algorithms for finding the root of a function. We know that an extreme of $f(x)$ is located at the point with

$$g(x) = \frac{df(x)}{dx} = 0,$$

which is a maximum or minimum depending whether the second derivative $g'(x) = f"(x)$ is, respectively, positive or negative. So the root search algorithms can find the point where the function $f(x)$ has a maximum or a minimum. Let's say that we are looking for a minimum of f. To make sure that we end up in a minimum and not in a maximum, in each step of updating the value of x, we need to make a decision whether $f(x_{n+1})$ is decreasing. If it is indeed decreasing, we update the value of x, i.e., $x_{n+1} = x_n + \Delta x$. If not, we reverse the update, i.e., $x_{n+1} = x_n - \Delta x$.

Exercise: Use the Newton-Raphson method to determine the bond length of a diatomic molecule, such as NaCl. We approximate the interaction of the two ions, Na^+ and Cl^-, separated by the distance r, with the following potential:

$$V(r) = -\frac{e^2}{r} + a\mathrm{e}^{-r/\rho},$$

where e is the charge of a proton, and a and ρ are effective parameters of this interaction. We use $a = 1.09 \times 10^3$ eV, and $\rho = 0.33$ Å.

II.4.6 Newton-Raphson Method for Multi-Variable Functions

The Newton-Raphson method can be generalized, in a straight way, to systems of several non-linear equations and variables. To illustrate the method, let's treat the case of two equations. We seek to find the roots of the system of two equation $f(x, y)$ and $g(x, y)$:

$$\begin{cases} f(x, y) = 0 \\ g(x, y) = 0. \end{cases}$$

Let's generalize the concept of tangent and perform a Taylor expansion of these function near the roots r, q:

$$\begin{cases} f(r, q) = 0 = f(x_n, y_n) + h_n \frac{\partial f}{\partial x} + k_n \frac{\partial f}{\partial y} \\ g(r, q) = 0 = g(x_n, y_n) + h_n \frac{\partial g}{\partial x} + k_n \frac{\partial g}{\partial y} \end{cases}$$

If we define the Jacobian matrix as:

$$J = \begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y}, \end{pmatrix}$$

the approximate roots x_{n+1}, y_{n+1} at the $n+1$ iteration are given by:

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} x_n \\ y_n \end{pmatrix} - J^{-1} \begin{pmatrix} f(x_n, y_n) \\ g(x_n, y_n). \end{pmatrix}$$

We can see that the Newton-Raphson method is more involved in multidimensional cases. We will define first the starting position x_1, y_1, which is not obvious, because we don't necessarily know an approximate solution to the problem at hand. This is why the use of the NR method in multidimensional search of roots is very complex. Once the starting position is known, one has then to calculate the partial derivatives at the starting position. We need thus to compute the inverse of the Jacobian matrix. This is not a trivial operation, because it may be singular, or nearly singular. Once this is done we can calculate the next approximation of the roots using the above equation. We can then check for convergence of the roots to a given numerical precision. If the precision is not achieved we can restart all over again, using the new approximate roots as a starting point, and keep iterating until convergence.

It is rather straightforward to generalize the above equations to more than two non-linear equations. But this is usually not a good idea, because there are many much more interesting methods, like the conjugate gradient methods which will be developed later in this course.

II.5 Fast Fourier Transform

II.5.1 Introduction

Many problems of physics can be analyzed using the so called Fourier transform methods or spectral analysis. It was indeed Fourier who first decomposed any periodic function $f(t)$ of period T into a summation of simple harmonic functions. Each of those functions is periodic and of period T, and the angular frequency is a multiple of the fundamental angular frequency $2\pi/T$. The function f can be written as

$$f(t) = \sum_{k=-\infty}^{k=+\infty} g_k \mathrm{e}^{\mathrm{i}k\omega t}. \tag{II.17}$$

This is commonly known as the Fourier series or the Fourier theorem of periodic functions. Here the g_k are the Fourier coefficients, which are given by[1]

$$g_k = \frac{1}{T}\int_0^T f(t)\,\mathrm{e}^{-\mathrm{i}k\omega t}\mathrm{d}t.$$

II.5.2 Fourier Transform

The Fourier transform generalize the Fourier series to a non periodic function. Let's suppose that the function f is known in an interval $[a,b]$. Let's assume furthermore that we know a complete set of orthogonal functions ϕ_k such that

$$\int_a^b \varphi_k^*(t)\,\varphi_{k'}(t)\,\mathrm{d}t = \delta_{k,k'}.$$

We can then always decompose the function f on the basis functions ϕ_k:

$$f(t) = \sum_{k=-\infty}^{k=+\infty} g_k \varphi_k(t).$$

If the function is square integrable the coefficient g_k are given by

$$g_k = \int_a^b \varphi_k^*(t)\,f(t)\,\mathrm{d}t.$$

In addition, if we suppose that the function f is defined in the interval $[-T/2, T/2]$ and then extend the period to infinity, we can extend the Fourier series given by Eq. II.17 to a Fourier integral

$$f(t) = \frac{1}{\sqrt{2\pi}}\int_0^T g(\omega)\,\mathrm{e}^{\mathrm{i}\omega t}\mathrm{d}\omega,$$

which is known as the Fourier integral or the Fourier transform. The Fourier coefficients g are then given by

$$g(\omega) = \frac{1}{\sqrt{2\pi}}\int_0^T f(t)\,\mathrm{e}^{-\mathrm{i}\omega t}\mathrm{d}t.$$

[1]We assume that the reader is familiar with the Fourier series and transforms, so no details or demonstrations will be given in this overview.

$f(t)$ real	$g(-\omega) = g^\star(\omega)$
$f(t)$ imaginary	$g(-\omega) = -g^\star(\omega)$
$f(t)$ odd	$g(-\omega) = -g(\omega)$ and g is odd
$f(t)$ even	$g(-\omega) = g(\omega)$ and g is even
$f(t)$ real and odd	$g(\omega)$ imaginary and odd
$f(t)$ real and even	$g(\omega)$ real and even
$f(t)$ imaginary and odd	$g(\omega)$ real and odd
$f(t)$ imaginary and even	$g(\omega)$ imaginary and odd

Table II.1: Symmetries of the function f and its Fourier transform g

Notice that the use of $\sqrt{2\pi}$ in the Fourier transform is just a convention, so that the transform and its inverse are symmetric. We could also not use the $\sqrt{2\pi}$ but then one has to renormalize the integral of g by 2π instead of $\sqrt{2\pi}$.

The two last equations are commonly known as the Fourier transform and the inverse Fourier transform. Note that we can show easily that the two equations are consistent by using the fact that

$$\int_{-\infty}^{+\infty} \mathrm{e}^{\mathrm{i}(\omega-\omega')t}\mathrm{d}t = \delta(\omega-\omega').$$

The Fourier transform can be generalized to any dimensional space. It is often used in three dimensions. The Fourier transform of function $f(\mathbf{r})$ is given by

$$f(\vec{r}) = \frac{1}{(2\pi)^{\frac{3}{2}}} \iiint g(\vec{k})\mathrm{e}^{\mathrm{i}\vec{k}\cdot\vec{r}}\mathrm{d}^3k$$

$$g(\vec{k}) = \frac{1}{(2\pi)^{\frac{3}{2}}} \iiint f(\vec{r})\mathrm{e}^{-\mathrm{i}\vec{k}\cdot\vec{r}}\mathrm{d}^3r.$$

It is always helpful to use the symmetries of the function that we would like to Fourier transform because it may greatly reduce the numerical computation. In Table II.1 we give all possible symmetries of a function f and the corresponding symmetries of its Fourier transform. For example, if f is even, then g is even. We can also define other properties of the Fourier transform concerning time scaling and time shift. Let's define the Fourier transform and its inverse as follows: $[f(t), g(\omega)]$. We show the following properties:

- Time scaling: $[f(\alpha t), g(\omega/\alpha)/\alpha]$.
- Frequency scaling: $[f(t/\alpha)/\alpha, g(\alpha\omega)]$.
- Time shifting: $[f(t-t_0), g(\omega)\mathrm{e}^{\mathrm{i}\omega t_0}]$.
- Frequency shifting: $[f(t)\mathrm{e}^{-\mathrm{i}\omega_0 t}, g(\omega-\omega_0)]$.

II.5.3 Convolution and Correlation of Two Functions

Let's assume that we know two functions f and h and their transforms g and k. The convolution of two functions f and h is written $f * h$, and is defined as:

$$f * h \equiv \int_{-\infty}^{+\infty} f(\tau)\, h(t-\tau)\, \mathrm{d}\tau.$$

It is easy to show the following theorem:

$$[f * h(t), g(\omega)h(\omega)]$$

.

The correlation of two functions is defined as:

$$Corr\,(f,h) \equiv \int\limits_{-\infty}^{+\infty} f\,(\tau + t)\,h\,(\tau)\,\mathrm{d}\tau.$$

Then it is again easy to show the following theorem:

$$Corr(f,h) = g(\omega)^{\star} h^{(}\omega)$$

II.5.4 Parseval Theorem

The Parseval theorem states that the total power signal is independent of whether it is evaluated in the time domain or the frequency domain, i.e.:

$$\int\limits_{-\infty}^{+\infty} |f\,(t)|^2\,\mathrm{d}t = \int\limits_{-\infty}^{+\infty} |f\,(\omega)|^2\,\mathrm{d}\omega.$$

II.5.5 Fast Fourier Transform

The fast Fourier transform (FFT) is a way of calculating the discrete Fourier transform (DFT) of N points using less computation. Before the article of J. W. Cooley and J. W. Tukey, in 1965, it was asumed that the discrete Fourier transform requires N^2 operations. In fact, if we define

$$W \equiv \mathrm{e}^{2\pi\mathrm{i}/N}$$

the DFT can be defined using a matrix-vector multiplication

$$(H_n) = \left(W^{nk}\right)(h_k)\,,$$

where the repeated index k runs from 0 to $N-1$. By rearranging the terms appropriately it was shown that it only requires $\mathcal{O}\left[N\,\mathrm{Log}_2\,(N)\right]$ operations. To do so, a numerical algorithm called the fast Fourier transform, or FFT for short, was devised by Cooley and Tukey. Notice that the difference between $\mathcal{O}\left[N\,\mathrm{Log}_2\,(N)\right]$ and $\mathcal{O}\left(N^2\right)$ can be very large. For $N = 10^6$, the difference can be between few seconds of CPU and few weeks on a simple Pentium Intel computer. It seems, however, that the FFT algorithm was already known to Gauss in 1805, and to others, like Lanczos.

To understand this huge reduction in the number of operations, let's suppose that the number N of DFT components is power of 2, like $N = 2^M$. However, this is by no means a restriction of the algorithm or a limitation, in fact any power of low prime numbers can be used. The DFT of length N can be written as a sum of even-numbered points and odd-numbered points:

$$H_n = \sum_{k=0}^{N-1} h_k \mathrm{e}^{\frac{2\pi\mathrm{i}kn}{N}} = \sum_{k=0}^{\frac{N}{2}-1} h_{2k}\mathrm{e}^{\frac{2\pi\mathrm{i}(2k)n}{N}} + \sum_{k=0}^{\frac{N}{2}-1} h_{2k+1}\mathrm{e}^{\frac{2\pi\mathrm{i}(2k+1)n}{N}}.$$

This expression can be rewritten as:

$$H_n = \sum_{k=0}^{\frac{N}{2}-1} h_{2k}\mathrm{e}^{\frac{2\pi\mathrm{i}kn}{N/2}} + W^n \sum_{k=0}^{\frac{N}{2}-1} h_{2k+1}\mathrm{e}^{\frac{2\pi\mathrm{i}kn}{N/2}} = H_n^e + W^n H_n^o.$$

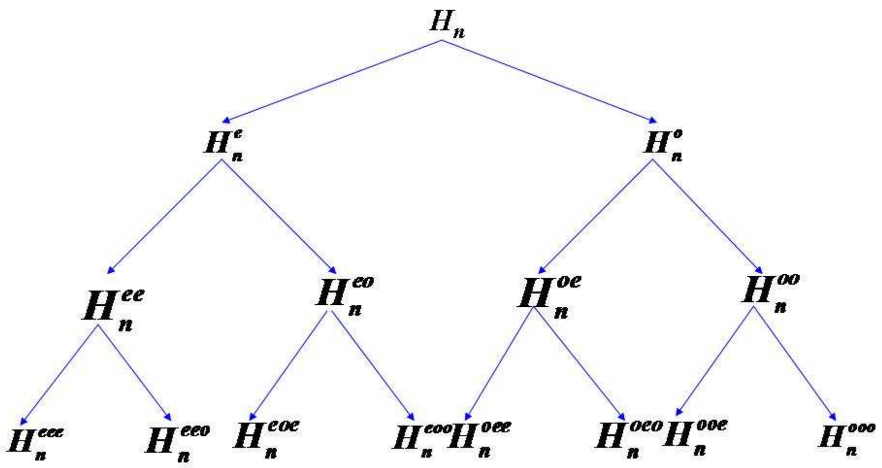


Figure II.8: Tree splitting of the DFT into even and odd-numbered sums.

From this last expression, it is easy to notice that we need only to calculate half of the H_n terms; the other half can be almost obtained for free. In fact it is easily seen that there is a periodicity with a period of $N/2$. We can then determine $H_{n+N/2}$ if we have already calculated H_n^e and H_n^o. In fact:

$$H_{n+\frac{N}{2}} = H_n^e + W^n e^{i\pi} H_n^o.$$

The nice thing about this decomposition is that we can continue it recursively, by dividing each of the H_n^e and H_n^o into even-numbered and odd-numbered sums. This recursion continues until we subdivide the data down to transforms of length 1. Like this, we obtain a tree of $\mathrm{Log}_2(N)$ sub-branches containing at the end of the lowest branches the DFT input data (see Fig. II.8). The above tree will make it possible to calculate all the H_n directly. One does not, of course, add up the whole tree for a single value of n and start over again. Such a procedure will let the algorithm scale as $\mathcal{O}(N^2)$. In our tree example, the H^{eee}, H^{eeo}, etc. are in fact the input DFT coefficients. They do not depend on the value of n because they are periodic in n with a period 1. Now, one has to figure out which value of h_k will correspond to which pattern of e's and o's in the tree. It is obvious that, in our example, H^{eee} will correspond to h_0 and H^{ooo} to h_7. To figure out the other points of h_n we will just reverse the patterns of e's and o's, and let $e = 0$, and $o = 1$, so that the binary value of the pattern will give us the value of k. To understand this issue, we have illustrated it in the example where $N = 8$ as shown in Fig. II.9. We can see that the original data above was split exactly as we have expected it using bit reversing of each binary written index of the Fourier coefficients. The FFT algorithm is now easy to implement. We have just to combine adjacent pairs to get two-point transforms, then combine adjacent pairs of pairs to obtain a four-point transforms and so on, until the first and second halves of the whole data set are combined into the final transform, without, of course, forgetting to multiply the odd sum by the appropriate $\exp\left[2\pi in/\left(N/2^M\right)\right]$, where n is the index of the calculated DFT, and, in our example, $M = 0, 1,$ or 2 is the level of the tree. Each sub-branch combination takes of $\mathcal{O}(N)$ operations, and since there are sub-branches the whole FFT scales as $\mathcal{O}\left(N\,\mathrm{Log}_2\left(N\right)\right)$.

The algorithm can be simply implemented as in the following Fortran example which is similar to the one originally written by N. M. Brenner (see Numerical recipes book by W. H. Press *et al.*)

```
subroutine fft (ar, ai, n, m)
 real (kind=8), intent(inout) :: ar(n), ai(n)
 integer(kind=4), intent(in) :: n, m
!local variables
 real (kind=8) ::  temp1, temp2, pi, q
```

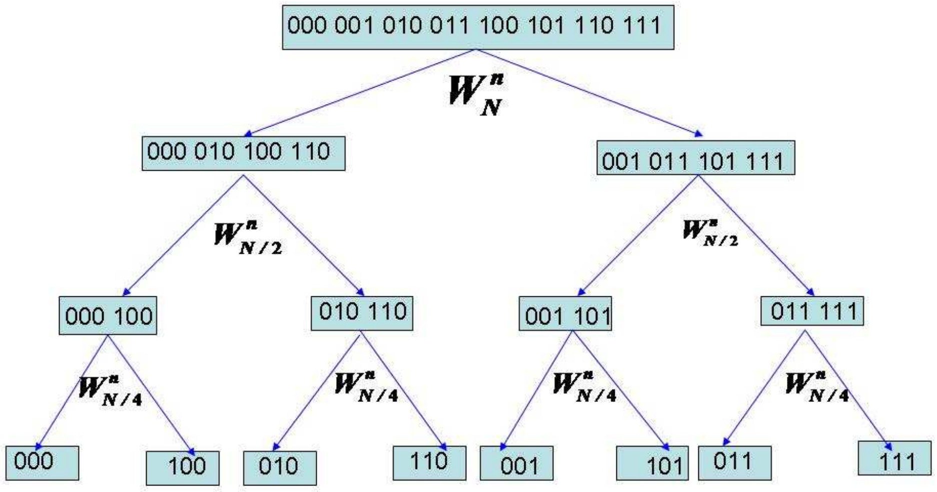


Figure II.9: Tree spletting of the DFT into even and odd-numbered sums using binary numbering of the index of the data. Here $W^n_{N/2^M} = \exp(2\pi in/N/2^M)$ ($M = 0$, 1, or 2).

```
integer(kind=4):: l, k, j, l1, l2, n1

pi = 4.0D0 * atan(1.0D0)
n1 = 2**m
IF (n1 /= n) then
  print *, 'Try again m and n not compatible'
  stop
END IF
l = 1
DO k = 1, n-1  ! Rearranging the DFT data to bit-reversed order
   IF (k < l) THEN
      temp1    =  ar(l)
      temp2    =  ai(l)
      ar(l)    = ar(k)
      ai(l)    = ai(k)
      ar(k)    = temp1
      ai(k)    = temp2
   ENDIF
   j = n/2
   DO WHILE (J <= L)
      l = l - j
      j = j/2
   END DO
   l = l + j
ENDDO
l2 = 1
DO l = 1, m
   q = 0.0D0
   l1 = l2
   l2 = 2 * l1
   DO k = 1, l1
      u = cos(q)
      v = -sin(q)
      q = q + pi/l1
```

```
       DO j = k, N, n2
          i = j + l1
          temp1 = ar(i) * u - ai(i) * v
          temp2 = ar(i) * v + ai(i) * u
          ar(i) = ar(j) - temp1
          ar(j) = ar(j) + temp1
          ai(i) = ai(j) - temp2
          ai(j) = ai(j) + temp2
       ENDDO
     ENDDO
 ENDDO
END SUBROUTINE FFT
```

Chapter III

Differential Equations

III.1 General introduction

In physics we encounter many types of differential equations. In this chapter we will study three types of them. We will start with initial value problems. These are related to a great set of time dependent ordinary differential equations where, for example, the initial values of the position and the velocity are given. In classical mechanics we have encountered many cases involving the dynamics of a system where we had to solve the Newton equations of motion. The second type of differential equations are composed problems with boundary conditions. The most well known differential equation is the Poisson equation. To solve this equation we have to know the potential and its derivative at the boundary of the system where we would like to determine the potential. The third type of differential equations involve the case of eigenvalue problems. We have to find the solution of the differential equation for a selected set of parameter values of the equation. The most known example is the Schroedinger equation:

$$\left[-\frac{\hbar^2}{2m}\nabla^2 + V(r)\right]\phi(r) = e\phi(r).$$

In this case we have usually to determine the energy e of all the bound states. This means that we have to find all the energies e where the function $\phi(r)$ is square summable. We will come back to this example at the end of this chapter where we will solve the one dimensional Schroedinger equation.

III.2 Initial Value Problems

As we said in the general introduction, this kind of problems concern the dynamics of systems, for example, finding the solution of three body motion (sun, earth and the moon) or determining the propagation of the sound in a liquid.
In general, the behavior of a dynamical system can be described by a set of first order coupled differential equations:

$$\frac{\mathrm{d}\vec{y}}{\mathrm{d}t} = \vec{g}(\vec{y}, t)$$

where $\vec{y} = (y_1, y_2, ..., y_n)$ is the dynamical variable vector, and $\vec{g}(\vec{y}, t) = [g_1(y,t), g_2(y,t), ..., g_n(y,t)]$ is the generalized velocity vector. In principle, the numerical solution of these coupled first order differential equations (if it exist) can be obtained for the initial condition $\vec{y}(t=0) = \vec{y}_0$. To illustrate this concept let's consider the equation of a pendulum with friction α and an external force $f_0\cos(\omega_0 t)$. The equation of motion is given by the following second order differential equation:

$$\frac{\mathrm{d}^2x}{\mathrm{d}t^2}+\alpha\frac{\mathrm{d}x}{\mathrm{d}t}+x=f_0\cos(\omega_0 t).$$

For simplicity we have set the eigenfrequency to one. To transform this equation into a set of coupled first order differential equations we will put $y_1=x$ and $y_2=\frac{\mathrm{d}x}{\mathrm{d}t}$. This produces the following coupled equations

$$\begin{cases}\dfrac{\mathrm{d}y_1}{\mathrm{d}t}=y_2\\ \dfrac{\mathrm{d}y_2}{\mathrm{d}t}=-\alpha y_2-y_1+f_0\cos(\omega_0 t).\end{cases}$$

In general, a n-order differential equation,

$$a_n(x,t)\frac{\mathrm{d}^nx}{\mathrm{d}t^n}+a_{n-1}(x,t)\frac{\mathrm{d}^{n-1}x}{\mathrm{d}t^{n-1}}+\cdots a_0(x,t)=0,$$

can be transformed into a set of first order coupled differential equations by setting $y_1=x$:

$$\begin{cases}\dfrac{\mathrm{d}y_1}{\mathrm{d}t}=y_2\\ \dfrac{\mathrm{d}y_2}{\mathrm{d}t}=y_3\\ \vdots=\vdots\\ a_n(y_1,t)\dfrac{\mathrm{d}y_n}{\mathrm{d}t}=-a_{n-1}(x,t)y_n\cdots-a_1(y_1,t)y_2-a_0(y_1,t).\end{cases}$$

III.3 Euler and Picard Methods

III.3.1 Euler Method

To make things simple, we will consider only the case of one dynamical variable. The generalization to many variables is easy and will be done later. In the Euler method one takes the derivative as a simple finite difference:

$$\frac{\mathrm{d}y}{\mathrm{d}t}=\frac{y_{n+1}-y_n}{t_{n+1}-t_n}=g(y_n,t_n)\equiv g_n.$$

We define $\tau=t_{n+1}-t_n$. The previous finite difference produces a simple algorithm for obtaining recursively all values of y_n

$$y_{n+1}=y_n+\tau g_n+\mathcal{O}(\tau^2)\ .$$

This algorithm is not accurate because after N steps the error becomes very large of the order of $N\mathcal{O}(\tau^2)=\mathcal{O}(\tau)$. This method is therefore not adapted to find numerical solutions of realistic physical problems. We can, in principle, improve the algorithm by integrating the first order differential equation between t_n and t_{n+m},

$$\int_{t_n}^{t_{n+m}}\frac{\mathrm{d}y}{\mathrm{d}t}\mathrm{d}t=\int_{t_n}^{t_{n+m}}g(y_n,t)\mathrm{d}t\Longrightarrow y_{n+1}=y_n+\int_{t_n}^{t_{n+m}}g(y,t)\mathrm{d}t$$

which is the exact solution of the differential equation if we could carry out the integral of the general velocity g. In general, it is not possible to integrate exactly the function g, and one has to approximate it. If we use $m=1$ and $\int_{t_n}^{t_{n+1}}g(y_n,t)\mathrm{d}t=\tau g_n$ we recover the Euler method.

III.3.2 Euler-Picard Method

If we use the trapezoidal method to integrate the general velocity given in the previous subsection we obtain the so called Euler-Picard method provided we find a way of obtaining the initial value of y_{n+1}:

$$y_{n+1} = y_n + \frac{\tau}{2}\left[g(y_n, t_n) + g(y_{n+1}, t_{n+1})\right].$$

To use this method we have to provide an initial guess for y_{n+1} to be used in the right hand side of the equation and then iterate the method to provide a self-consistent solution. This method can be very time consuming because we need to converge y_{n+1}. This will not happen if the initial guess is not close to the solution. A more interesting method for providing a more accurate guess for y_{n+1} is known as the predictor-corrector method and will be exposed in the next subsection.

III.3.3 Predictor-Corrector Method

In this method, we use a less accurate algorithm to find the initial guess for y_{n+1}. We can use, for example, the Euler method. Then we substitute this value in the right hand side of the previous equation:

$$y^0_{n+1} = y_n + \tau g_n + \mathcal{O}(\tau^2)$$

and

$$y_{n+1} = y_n + \frac{\tau}{2}\left[g(y_n, t_n) + g(y^0_{n+1}, t_{n+1})\right].$$

Here, we do not need to iterate y_{n+1}. We can also find different predictor and corrector algorithms to solve this problem by using higher polynomial integration like the formulas of Newton-Cotes. However, in general, it is not a good idea to use more than two predicted values of y because the formulas will get complicated, and the numerical error increases enormously. One is also tempted to insert the corrected value of y_{n+1} in the corrector formula and keep iterating. The gain in accuracy is not sufficient to justify the extra effort and computer time. In the next section we will show that there is a much more powerful method, called the Runge-Kutta method, which is much superior to these low level methods.

III.4 Runge-Kutta Methods

In the previous method, the starting point requires at least two values of the function f. However, for dynamical systems we usually have only the initial condition which produces only the first starting point. We will see in this section that the Runge Kutta method requires only the first starting point and is a very accurate and powerful method that can, in principle, treat any dynamical system. This method is based on the Taylor expansion of the function y and the generalized velocity g. Let's start with the expansion for y:

$$y(t+\tau) = y + \tau y' + \frac{\tau^2}{2}y'' + \frac{\tau^3}{3!}y^{(3)} + \cdots.$$

If we substitute y' by the generalized velocity g and y'' by its first derivative and so on we obtain

$$y(t+\tau) = y + \tau g + \frac{\tau^2}{2}\frac{\mathrm{d}g}{\mathrm{d}t} + \frac{\tau^3}{3!}\frac{\mathrm{d}^2 g}{\mathrm{d}t^2} + \cdots.$$

We will now write the full derivatives of g with respect to t with respect of the partial derivatives and replacing any derivative of y by the general velocity or its derivatives. For the first few derivatives we obtain:

$$\frac{\mathrm{d}g}{\mathrm{d}t} = \frac{\partial g}{\partial y}\frac{\partial y}{\partial t} + \frac{\partial g}{\partial t} = g_y g + g_t$$

and

$$\frac{\mathrm{d}^2 g}{\mathrm{d}t^2} = g_{tt} + 2gg_{ty} + g^2 g_{yy} + gg_y^2 + g_t g_y.$$

We use all these partial derivatives in the Taylor expansion of y as given by the previous formula:

$$y(t+\tau) = y + \tau g + \frac{\tau^2}{2}(g_y g + g_t) + \frac{\tau^3}{6}(g_{tt} + 2gg_{ty} + g^2 g_{yy} + gg_y^2 + g_t g_y) + \cdots . \tag{III.1}$$

Here, we will use the fact that the partial derivatives are continuous, i.e., $g_{ty} = g_{yt}$. We can also formally write the Taylor expansion as

$$y(t+\tau) = y + \alpha_1 k_1 + \alpha_2 k_2 + \alpha_3 k_3 + \cdots + \alpha_n k_n,$$

where the k_n are given by

$$\begin{aligned} k_1 =& \tau g(y,t) \\ k_2 =& \tau g(y + \nu_{21}k_1, t + \nu_{21}\tau) \\ k_3 =& \tau g(y + \nu_{31}k_1 + \nu_{32}k_2, t + (\nu_{31} + \nu_{32})\tau) \\ &\vdots \\ k_n =& \tau g\left(y + \sum_{i=1}^{n-1}\nu_{ni}k_i, t + \tau\sum_{i=1}^{n-1}\nu_{ni}\right), \end{aligned} \tag{III.2}$$

where the parameters α_i and ν_{ij} $(1 \leq i \leq n$ and $j < i)$ are to be determined by making Taylor expansions of the functions k_i into a power series of τ. We can then insert the results in Eq. (III.4) and compare term by term to the expansion of $y(t+\tau)$ given by Eq. (III.1). This comparison allows us to determine the the parameters α_i and ν_{ij}. If we truncate the expansion to the order n we obtain n equations containing $n + n(n-1)/2$ parameters α_i and ν_{ij}. We have then the freedom to arbitrarily choose some of them.

To make things clear we will develop in more detail the case for $n = 2$. Eq. (III.1) becomes

$$y(t+\tau) = y + \tau g + \frac{\tau^2}{2}(g_y g + g_t) \tag{III.3}$$

and, to the order $n = 2$, Eq. (III.4) becomes

$$y(t+\tau) = y + \alpha_1 k_1 + \alpha_2 k_2,$$

with k_1 and k_2 given by

$$\begin{cases} k_1 = & \tau g(y,t) \\ k_2 = & \tau g(y + \nu_{21}k_1, t + \nu_{21}\tau). \end{cases}$$

Now we will make a Taylor expansion of k_2 to the second order:

$$k_2 = \tau g + \nu_{21}\tau^2(gg_y + g_t).$$

We now substitute the expressions of k_1 and k_2 into Eq. (III.4) and obtain

$$y(t+\tau) = y + (\alpha_1 + \alpha_2)\tau g + \alpha_2\tau^2\nu_{21}(g_y g + g_t). \tag{III.4}$$

By comparing this final equation to that of the Taylor expansion to the second order given by Eq. (III.3) we obtain

$$\begin{cases} \alpha_1 + \alpha_2 & = 1 \\ \alpha_2\nu_{21} & = \dfrac{1}{2}. \end{cases}$$

We see that we have three parameters and two equations. We have then the freedom to choose one parameter freely and determine the two others from the set of equations. We can, for example, set $\nu_{21} = 1$ and obtain $\alpha_1 = \alpha_2 = \dfrac{1}{2}$. For a given problem the flexibility of the choice of the parameters may provide a way to increase the accuracy of the method.

The most used Runge Kutta method is when the expansion is set to the order four. Similar to what we have done for the $n = 2$ expansion one can get

$$y(t+\tau) = y + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4),$$

with k_1 to k_4 given by

$$\begin{cases} k_1 = \tau g(y,t) \\ k_2 = \tau g(y + \dfrac{k_1}{2}, t + \dfrac{\tau}{2}) \\ k_3 = \tau g(y + \dfrac{k_2}{2}, t + \dfrac{\tau}{2}) \\ k_4 = \tau g(y + k_3, t + \tau). \end{cases}$$

III.4.1 Chaotic Dynamics of a Driven Pendulum

Let us illustrate the use of the Runge Kutta method to solve the problem of a driven pendulum. We will use the case of a pendulum of length l and a mass m attached at the end. We will only consider the motion confined to a vertical plane. The forces that act upon the pendulum are the driving force F_d, the gravity force F_g, and the resistive force F_r. Let's assume that the driving force is given by $F_\mathrm{d} = F_0 \cos(\Omega t)$, where F_0 is the amplitude of the force and Ω is its angular frequency. Let us also suppose that the resistive force is proportional to the the velocity v of the pendulum and is given by $F_\mathrm{r} = -kv$, where k is a parameter. We apply Newton's second law to the pendulum and find that

$$ma = F_\mathrm{g} + F_\mathrm{d} + F_\mathrm{r},$$

where $F_\mathrm{g} = -mg \sin\theta$ is the force of gravity along the direction tangent to the trajectory of the mass m, with θ the angle between the rod and the vertical direction. The acceleration $a = l\mathrm{d}^2\theta\mathrm{d}t^2$ is along the tangential direction. By substituting the values of the force into the equation of motion of the pendulum and knowing that $v = l\mathrm{d}\theta\mathrm{d}t$ we obtain

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} + \alpha\frac{\mathrm{d}\theta}{\mathrm{d}t} + \omega_0^2 \sin\theta = f_0 \cos(\Omega t) \tag{III.5}$$

where $\omega_0^2 = g/l$, $\alpha = k/m$, and $f_0 = F_0/(ml)$. We can transform this second order ordinary differential equation into a system of first order coupled differential equations using Eq. (III.2), and we obtain

$$\begin{cases} \dfrac{\mathrm{d}y_1}{\mathrm{d}t} = y_2 \\ \dfrac{\mathrm{d}y_2}{\mathrm{d}t} = -\alpha y_2 - \alpha \sin y_1 + f_0 \cos(\Omega t), \end{cases}$$

where we have set $y_1 = \theta$ and $y_2 = \mathrm{d}\theta/\mathrm{d}t$. We will use the fourth order Runge-Kutta method to solve these set of equations. It is easy to generalize the Runge-Kutta method to a multi-variable equation

$$y_{n+1} = y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4),$$

with k_1 to k_4 given by

$$\begin{cases} k_1 = & \tau g(y,t) \\ k_2 = & \tau g(y + \dfrac{k_1}{2}, t + \dfrac{\tau}{2}) \\ k_3 = & \tau g(y + \dfrac{k_2}{2}, t + \dfrac{\tau}{2}) \\ k_4 = & \tau g(y + k_3, t + \tau) \end{cases}$$

where y_n and k_i $(1 \leq i \leq 4)$ are multidimensional vectors.

III.5 Boundary Value and Eigenvalue Problems

III.5.1 Introduction

One class of problem in physics requires the knowledge of the function at the boundary of the region where we would like to find the solution. One of such typical problems is the solution of the classical Poisson equation where one would like to determine the potential. Usually a charge distribution is used on that region and one knows the electrical potential at the boundary. In one dimension, the general form of the differential equation is given by

$$y'' = f(y, y', x),$$

where the first derivative y' and second derivative y'' are with respect to the position x. To solve this equation either y or y' is given at each boundary of the interval $[x_1, x_2]$. We can always change the x variable so that the limits are $x = 0$ and $x = 1$. In one dimension, we would have one of the four boundary conditions

$$\begin{cases} (1) & y(0) = & y_0 & \text{a}ndy(1) = & y_1 \\ (2) & y(0) = & y_0 & \text{a}ndy'(1) = & y_1' \\ (3) & y'(0) = & y_0' & \text{a}ndy(1) = & y_1 \\ (4) & y'(0) = & y_0' & \text{a}ndy'(1) = & y_1'. \end{cases}$$

We can not directly use the previous Runge-Kutta algorithm to find a numerical solution of this problem because we do not know the initial conditions for both, y and y'.

The problem of boundary conditions involving an eigenvalue ε and the general differential equation is given by

$$y'' = f(y, y', x, \varepsilon).$$

The solution of this kind of problem is even more complicated because only for some selected eigenvalues the equation yields an acceptable solution which is consistent with the known boundary conditions. In the next subsection we will explore a powerful method known as the shooting method.

III.5.2 The Shooting Method

This method is easy to understand but powerful for solving both, boundary condition and eigenvalue problems. To illustrate this method let us first use it for the boundary condition problem. We have then to transform the second order differential equation given by Eq. (III.5.1) into a set of first order differential equations like in the case of initial condition problems. Let us use $y = y_1$ and $y' = y_2$, then this boundary condition equation becomes

$$\begin{cases} \dfrac{\mathrm{d}y_1}{\mathrm{d}x} = y_2 \\ \dfrac{\mathrm{d}y_2}{\mathrm{d}x} = f(y_1, y_2, x). \end{cases}$$

To solve the problem, let us assume the first of the four boundary conditions given above, i.e., $y(0) = y_0$ and $y(1) = y_1$. One can use either of the four conditions; it will not change anything to the solution of this kind of problems. To make use of the methods developed for the initial value problem one has to find a way of using the initial conditions even though one of them is not known. The basic strategy of the shooting method is to introduce an arbitrary starting value δ for the messing initial condition and require that this value should be iteratively adjusted so as to reproduce the other boundary condition. It is clear that if one starts from any value $y'(0) = \delta$ then one ends up with a final boundary $y_\delta(1)$ which is, in general, different from the known boundary $y(1)$. To solve the problem one has to find the value of δ that produces the correct value of $y(1)$ with a given precision. In practice, one has to find the root of the following equation: $G(\delta) = y_\delta(1) - y(1) = 0$. One can, for example, use the Newton-Raphson algorithm to find the root of this equation.

III.6 Sturm-Liouville Problem

Many of the differential equations of physics can be cast in the form of a linear, second-order equation of the following type:

$$y'' + a(x)y' + b(x)y = s(x).$$

In the case that the functions $a(x)$, $b(x)$ and $s(x)$ behave well, one could use the previous shooting method to solve this problem. In this case there is no need to find the ultimate starting value for the parameter δ which produces $G(\delta) = y_\delta(1) - y(1) = 0$. This is because of the superposition principle of linear equations which states that any linear combination of the solutions is still a solution of the linear equation. One therefore needs only two starting points, δ_1 and δ_2, and to find out the coefficients of the linear combination of the general solutions that produce the correct boundary conditions. Let's suppose that the two particular solutions of the equations corresponding to δ_1 and δ_2 are y_{δ_1} and y_{δ_2}. The general solution is then

$$y(x) = \alpha y_{\delta_1}(x) + \beta y_{\delta_2}(x).$$

The α and β coefficients can be determined from the fact that $y(0) = y_0$ and $y(1) = y_1$ which produces the following Kramer system of equations:

$$\begin{cases} \alpha + \beta = 1 \\ \alpha y_{\delta_1}(1) + \beta y_{\delta_2}(2) = y_1 \end{cases}$$

which can easily solved to produce the values of α and β.

An important class of linear equations is known under the name of Sturm-Liouville problem and is defined by

$$[p(x)y']' + q(x)y = s(x).$$

In general, the function q may depend on the parameter ε which is the eigenvalue of the equation. The one dimension Schroedinger equation is a simple case of the Sturm-Liouville problem where $p(x) = 1$, $q(x) = 2m[\varepsilon - V(x)]/\hbar^2$, and $s(x) = 0$. Here, $V(x)$ is the one particle potential and ε is the energy. Bessel and Legendre equations are also other special cases. There is a particular powerful algorithm for solving the Sturm-Liouville problem known as the Numerov or the Cowling's algorithm.

III.6.1 The Numerov Algorithm

In this subsection we will illustrate the Numerov algorithm to the Sturm-Liouville problem for the case where the function $p(x) = 1$ and leave out the general case to the reader. We begin by approximating the second derivative of y by the three-point formula that we have developed in the finite difference section:

$$\frac{y_{n+1} - 2y_n + y_{n-1}}{h^2} = y_n'' + \frac{h^2}{12} y_n^{(4)} + \mathcal{O}(h^4).$$

The nice thing about this development is that we can keep the fourth order derivative $y_n^{(4)}$ by exploiting the Sturm-Liouville differential equation:

$$y_n^{(4)} = \frac{\mathrm{d}^2(-q(x)y + s(x))}{\mathrm{d}x^2} \,|_{x=x_n} = -\frac{(qy)_{n+1} - 2(qy)_n + (qy)_{n-1}}{h^2} + \frac{s_{n+1} - 2s_n + s_{n-1}}{h^2} + \mathcal{O}(h^2).$$

Putting back this last expression into the previous one and substituting y_n'' back into the Sturm-Liouville equation produces

$$\left(1 + \frac{h^2}{12}q_{n+1}\right) y_{n+1} - 2\left(1 - \frac{5h^2}{12}q_n\right) y_n + \left(1 + \frac{h^2}{12}q_{n-1}\right) y_{n+1} = \frac{h^2}{12}\left(s_{n+1} + 10s_n + s_{n-1}\right) + \mathcal{O}(h^6). \tag{III.6}$$

The local accuracy of the Numerov algorithm is $\mathcal{O}(h^6)$. However, because of the repeated use of the three-point formulas the global accuracy of the algorithm is only $\mathcal{O}(h^4)$. Solving Eq. (III.6) for either y_{n+1} or y_{n-1} then provides a recursion relation for integrating either forward or backwards in x. However, to start the algorithm one needs two starting points, for example, y_0 and y_1. One could use, for example, the Runge-Kutta method to obtain y_1 and then the Numerov algorithm to continue the recursion relation.

III.7 The One-Dimensional Schrödinger Equation

Integrating the Schrödinger equation numerically is very important in understanding quantum mechanics. In this section we will apply the methods developed in the previous sections to the one-dimensional Schrödinger equation:

$$-\frac{\hbar^2}{2m}\frac{\mathrm{d}^2\Psi(x)}{\mathrm{d}x^2} + V(x)\Psi(x) = E\Psi(x), \tag{III.7}$$

where $\hbar$ is the Planck constant, m the mass of the particle, E its eigenvalue, Ψ its eigenvector, and V is the external potential. We can rewrite the Schrödinger equation as

$$\Psi''(x) + \frac{2m}{\hbar^2}[E - V(x)]\Psi(x) = 0, \tag{III.8}$$

which is a special case of the Sturm-Liouville equation with $p(x) = 1$, $q(x) = (2m/\hbar^2)[E - V(x)]$ and $s(x) = 0$. We can then use the Numerov algorithm to solve this eigenvalue problem. Let us assume that the particle is confined by a potential well $V(x)$. The boundary condition of the wave function is that when $\mid x \mid \longrightarrow \infty$ then $\Psi(x) \longrightarrow 0$. The wave function increases exponentially if the eigenvalue is below the potential (the classically forbidden region), then oscillates in the region where the eigenvalue is larger than the potential, and decreases again exponentially in the region where the eigenvalue is below the potential. The error accumulated will become significant if one integrates from the exponentially increasing region to the oscillating region and then to the exponentially decreasing region. This is because an exponentially increasing function is also a solution to the Schroedinger equation if the eigenvalue is not a bounded state. To make the algorithm stable one has to integrate from both sides (forward and backwards) and then match the solutions in the well region. It is then clear that the matching point has to be chosen when the eigenvalue is equal to the potential. Let us call this point x_0. We have then to adjust the trial eigenvalue until the solution integrated from the right matches that obtained from the integration from the left. Since the function Ψ_l obtained by integrating from the left and the function Ψ_r obtained by integrating from the right towards x_0 satisfy a homogeneous equation, their normalizations can always be chosen such that the two functions are equal at x_0. (For instance, one can define $\Psi_l(x) = \tilde{\Psi}_l(x)\Psi_r(x_0)/\tilde{\Psi}_l(x_0)$, where $\tilde{\Psi}_l$ is the initial left solution before the equality.) The eigenvalue is then found by equality of the derivatives at x_0, i.e., the solutions match smoothly at the matching point. If we combine the matching of the functions and their derivatives at x_0 we obtain the so called logarithmic derivative

$$\frac{\Psi_l'(x_0)}{\Psi_l(x_0)} = \frac{\Psi_r'(x_0)}{\Psi_r(x_0)}.$$

Numerically, we can insure this by using, for example, a three point formula for the derivative. We can define a function $G(E)$ that is equal to the difference of the left and right logarithmic derivative:

$$G(E) = \frac{[\Psi_l(x_0+h) - \Psi_l(x_0-h)] - [\Psi_r(x_0+h) - \Psi_r(x_0-h)]}{2h\Psi_r(x_0)} = 0$$

We then have to find the correct eigenvalue E which ensures that $G(E) = 0$ to a given accuracy.

In conclusion, here are the steps required to integrate an eigenvalue Sturm-Liouville problem such as the one dimensional Schroedinger equation:

- We have to choose carefully the region of integration. It should be large enough compared to the extension of the potential. One has to make sure that the limits of the region have a negligible effect on the solution. One can always check this by choosing different regions and compare the solutions.
- We should have a physical insight to guess the lowest eigenvalue of the problem.
- x_0 has to be chosen with care. If the potential is given numerically one has to find the zero of $V(x) - E = 0$ using one of the algorithms that we developed in the chapter concerning the zeros of functions, e.g., the Newton-Raphson method.
- One has then to perform forward and backward integrations towards x_0. The function Ψ_l has to be integrated to $x_0 + h$ and Ψ_r to $x_0 - h$ such that one can calculate derivatives using three point formulas. Scale the forward or the backward solution such that $\Psi_l(x_0) = \Psi_r(x_0)$.
- Calculate $G(E) = \frac{[\Psi_l(x_0+h) - \Psi_l(x_0-h)] - [\Psi_r(x_0+h) - \Psi_r(x_0-h)]}{2h\Psi_r(x_0)}$ and carry out a root search to determine the first eigenvalue.
- Carry out the previous steps for the determination of the next eigenvalue. One can start with a slightly higher value of E and keep increasing it until the root search provides the next eigenvalue.

Chapter IV

Monte Carlo Simulations

IV.1 Introduction

We consider a system of N classical particles in the *canonical ensemble*. That is, a thermodynamic system in which the volume V, temperature T, and N are constant.

The goal of statistical mechanics is to calculate thermodynamic observables in such an ensemble. The observables are defined by statistical averages

$$\langle A(\vec{r}_1,\ldots,\vec{r}_N)\rangle = \frac{\int \mathrm{d}^3\vec{r}_1 \ldots \int \mathrm{d}^3\vec{r}_N \int \mathrm{d}^3\vec{p}_1 \ldots \int \mathrm{d}^3\vec{p}_N A(\vec{r}_1,\ldots,\vec{r}_N)\mathrm{e}^{-\beta\mathcal{H}}}{\int \mathrm{d}^3\vec{r}_1 \ldots \int \mathrm{d}^3\vec{r}_N \int \mathrm{d}^3\vec{p}_1 \ldots \int \mathrm{d}^3\vec{p}_N \mathrm{e}^{-\beta\mathcal{H}}} \tag{IV.1}$$

where

- we assume that A is only a function of $\{\vec{r}_{i=1,\ldots,N}\}$ (case which often appears),
- the hamiltonian $\mathcal{H}$ = total energy

$$\mathcal{H}(\{\vec{r}_i,\vec{p}_i\}) = \underbrace{\sum_{i=1}^{N}\frac{\vec{p}_i^2}{2m}}_{\text{kinetic energy}} + \underbrace{U(\vec{r}_1,\ldots,\vec{r}_N)}_{\text{potential energy}} , \tag{IV.2}$$

- $\beta = 1/k_\mathrm{B}T$ with k_B = Boltzmann constant.

Since A is not a function of the momenta $\{\vec{p}_i\}$, one has

$$\langle A\rangle = \frac{1}{Z_\mathrm{c}}\int \mathrm{d}^3\vec{r}_1 \ldots \int \mathrm{d}^3\vec{r}_N A(\vec{r}_1,\ldots,\vec{r}_N)\mathrm{e}^{-\beta U(\vec{r}_1,\ldots,\vec{r}_N)} \tag{IV.3}$$

with the partition function

$$Z_\mathrm{c}(T,V,N) = \int \mathrm{d}^3\vec{r}_1 \ldots \int \mathrm{d}^3\vec{r}_N \mathrm{e}^{-\beta U(\vec{r}_1,\ldots,\vec{r}_N)} . \tag{IV.4}$$

Equation (IV.3) is a d-dimensional integral. Since d is large, i.e., $d = 3N$, the challenge consists in calculating such high-dimensional integrals. The Monte Carlo simulation method provides a numerical solution to this problem.

IV.2 Calculation of High-Dimensional Integrals

The basic idea to calculate high-dimensional integrals is to replace the integral by a sum, i.e.,

$$\int_a^b \mathrm{d}x f(x) \approx \sum_{m=1}^{M} f(x_m)\Delta x_m \qquad (d=1)$$

However, the question arises of how to choose the points x_m in order to obtain a precise estimate of the integral. One can think of two possibilities for this choice.

1. possibility: Choose the x_m in a regular way. That is, choose p points along each spatial direction.

There are two problems associated with this possibility:

(a) p points in $d=1 \Rightarrow p^{3N}$ points in $d=3N$. This implies that, even for modest choices for p and N, the number of points becomes prohibitively large. The following example illustrates this problem.

Let us choose $p=10$ and $N=100$ so that $p^{3N}=10^{300}$. The numerical solution of the integral is impossible because the time t to obtain the solution scales as $t \sim p^{3N}$. If we make the reasonable assumption that a modern processor can calculate 10^9 points/s, the time t becomes $t \sim 10^{293}\,\mathrm{s} \approx 10^{285}\,\mathrm{years}$.

(b) The second problem arises from a property of the integrand in Eq. (IV.3). This integrand is large only in a limited region of configuration space Γ.

Why? Let $\vec{x} \equiv (\vec{r}_1, ..., \vec{r}_N)$ denote a point in Γ and let

$$\begin{aligned} p(E)\mathrm{d}E &\mathrel{\hat{=}} \text{probability that the configuration } \vec{x} \text{ has an energy between } E \text{ and } E+\mathrm{d}E \\ &= \frac{1}{Z_\mathrm{c}} \int_{E\le U(\vec{x})\le E+\mathrm{d}E} \mathrm{d}^{3N}\vec{x}\mathrm{e}^{-\beta U(\vec{x})} \\ &= \frac{1}{Z_\mathrm{c}}\mathrm{e}^{-\beta E} \underbrace{\int_{E\le U(\vec{x})\le E+dE} \mathrm{d}^{3N}\vec{x}}_{=g(E)\mathrm{d}E \,\hat{=}\, \text{density of states}} \\ &= \frac{1}{Z_\mathrm{c}} g(E)\mathrm{e}^{-\beta E}\mathrm{d}E\,, \end{aligned} \tag{IV.5}$$

where

$$Z_\mathrm{c}(\beta) = \int_0^\infty \mathrm{d}E g(E)\mathrm{e}^{-\beta E} \qquad [\Leftrightarrow \text{(IV.4)}] \tag{IV.6}$$

because $\int_0^\infty \mathrm{d}E p(E) = 1$.

Since $g(E)$ is a monotonously increasing function of E, whereas $\mathrm{e}^{-\beta E}$ monotonously decreases, $g(E)\mathrm{e}^{-\beta E}$ has a sharp maximum at E_max. This implies that a subensemble of configuration space points with $U(\vec{x}) \approx E_\mathrm{max}$ gives the dominant contribution to $\langle A\rangle$. A regular decomposition of Γ will choose most x_m outside this interesting region, thus leading to a statistically inaccurate estimate of $\langle A\rangle$.

2. possibility: Choose the x_m in a random way, i.e., according to some probability $P_\mathrm{sim}(\vec{x})$.

Consequence for Eq. (IV.3):

$$\langle A\rangle \approx \frac{\sum_{m=1}^{M} A(\vec{x}_m)\mathrm{e}^{-\beta U(\vec{x}_m)}/P_\mathrm{sim}(\vec{x}_m)}{\sum_{m=1}^{M} \mathrm{e}^{-\beta U(\vec{x}_m)}/P_\mathrm{sim}(\vec{x}_m)} \tag{IV.7}$$

To justify Eq. (IV.6) let us consider Eq. (IV.4) in $d=1$:

$$Z_\mathrm{c} = \int_{-\infty}^{+\infty} \mathrm{d}x\mathrm{e}^{-\beta U(x)} = \int_{-\infty}^{+\infty} \mathrm{d}x P_\mathrm{sim}(x)\frac{\mathrm{e}^{-\beta U(x)}}{P_\mathrm{sim}(x)}\,.$$

Now, let $F(x)$ be the probability to find the random variable y between $-\infty$ and x

$$F(x) = \int_{-\infty}^{x} \mathrm{d}y P_{\mathrm{sim}}(y) \tag{IV.8}$$

F has the following properties:

- $F(-\infty) = 0,\ F(\infty) = 1$,
- $\dfrac{\mathrm{d}F}{\mathrm{d}x} = P_{\mathrm{sim}}(x)$, (IV.9)
- $F(x) \,\hat{=}\,$ monotoneous function: F can be inverted, i.e. $F(x) \to x(F)$.

Insertion in Z_{c} yields

$$Z_{\mathrm{c}} = \int_{-\infty}^{+\infty} \mathrm{d}x \frac{\mathrm{d}F}{\mathrm{d}x} \frac{\mathrm{e}^{-\beta U(x)}}{P_{\mathrm{sim}}(x)} = \int_{0}^{1} \mathrm{d}F \frac{\mathrm{e}^{-\beta U(x(F))}}{P_{\mathrm{sim}}(x(F))} \approx \sum_{m=1}^{M} \Delta F_m \frac{\mathrm{e}^{-\beta U(x(F))}}{P_{\mathrm{sim}}(x(F))} \ . \tag{IV.10}$$

If we now decompose the interval $[0, 1]$ into M subintervals, we have $\Delta F_m = 1/M$ and so

$$\overset{x(F_m)=x_m}{\Longrightarrow} \qquad Z_{\mathrm{c}} \approx \frac{1}{M} \sum_{m=1}^{M} \frac{\mathrm{e}^{-\beta U(x_m)}}{P_{\mathrm{sim}}(x_m)}$$

This result may be generalized to $3N$ dimensions. With $x_m \to \vec{x}_m$ we can write

$$\begin{aligned} Z_{\mathrm{c}} \quad &\overset{(\mathrm{IV.4})}{=} \quad \int \mathrm{d}^{3N}\vec{x} \mathrm{e}^{-\beta U(\vec{x})} \\ &\approx \quad Z_M = \frac{1}{M} \sum_{m=1}^{M} \frac{\mathrm{e}^{-\beta U(\vec{x}_m)}}{P_{\mathrm{sim}}(\vec{x}_m)} \ , \end{aligned} \tag{IV.11}$$

whence Eq. (IV.7) for the mean value of an observable.

Equation (IV.7) represents the basis of the *Monte Carlo method*:

Definition: Monte Carlo method := method for calculating high-dimensional integrals, which replaces the integral by an arithmetic average over (configuration phase) points that are randomly chosen according to some probability $P_{\mathrm{sim}}(\vec{x})$.

Consequence: Because $\vec{x}_m$ is a random variable, Z_M is also a random variable with

- average

$$\begin{aligned} \langle Z_M \rangle \quad &\overset{(\mathrm{IV.11})}{=} \quad \frac{1}{M} \sum_{m=1}^{M} \left\langle \frac{\mathrm{e}^{-\beta U(\vec{x}_m)}}{P_{\mathrm{sim}}(\vec{x}_m)} \right\rangle_{P_{\mathrm{sim}}} = \left\langle \frac{\mathrm{e}^{-\beta U(\vec{x})}}{P_{\mathrm{sim}}(\vec{x})} \right\rangle_{P_{\mathrm{sim}}} \\ &= \quad \int \mathrm{d}^{3N}\vec{x}\, P_{\mathrm{sim}}(\vec{x}) \frac{\mathrm{e}^{-\beta U(\vec{x})}}{P_{\mathrm{sim}}} = \int \mathrm{d}^{3N}\vec{x}\, \mathrm{e}^{-\beta U(\vec{x})} = Z_{\mathrm{c}} \ . \end{aligned}$$

- variance

$$\begin{aligned} \sigma_M^2 \quad &= \quad \frac{1}{M^2} \sum_{m,n=1}^{M} \left\langle \left(\frac{\mathrm{e}^{-\beta U(\vec{x}_m)}}{P_{\mathrm{sim}}(\vec{x}_m)} - Z_{\mathrm{c}} \right) \left(\frac{\mathrm{e}^{-\beta U(\vec{x}_n)}}{P_{\mathrm{sim}}(\vec{x}_n)} - Z_{\mathrm{c}} \right) \right\rangle_{P_{\mathrm{sim}}} \\ &= \quad \frac{1}{M^2} \sum_{m=1}^{M} \left\langle \left(\frac{\mathrm{e}^{-\beta U(\vec{x}_m)}}{P_{\mathrm{sim}}(\vec{x}_m)} - Z_{\mathrm{c}} \right)^2 \right\rangle_{P_{\mathrm{sim}}} \qquad (\{\vec{x}_m\} \text{ and } \{\vec{x}_n\} : \text{independent ensembles}) \\ &= \quad \frac{1}{M} \underbrace{\left[\left\langle \left(\frac{\mathrm{e}^{-\beta U}}{P_{\mathrm{sim}}} \right)^2 \right\rangle_{P_{\mathrm{sim}}} - Z_{\mathrm{c}}^2 \right]}_{=\sigma^2} \ . \end{aligned} \tag{IV.12}$$

Equation (IV.12) shows that there are two ways to improve the precision of Monte Carlo estimates:

(a) If the total number of points M tends to infinity, the variance vanishes as

$$\sigma_M \frac{\sigma}{\sqrt{M}} \overset{M\to\infty}{\longrightarrow} 0\ .$$

However, this decrease with M is slow. Standard integration methods in $d = 1$, for instance, converge much more rapidly with M.

(b) The second possibility consists in optimizing the probability $P_{\text{sim}}(\vec{x})$.

This idea leads us to the two commonly employed Monte Carlo methods, *simple sampling* and *importance sampling*, which we will discuss in the following chapter.

IV.3 Simple versus Importance Sampling

IV.3.1 Simple Sampling

Definition: Simple sampling (ss) := choose all $\vec{x}_m$ according to a uniform distribution, that is

$$P^{\text{ss}}_{\text{sim}}(\vec{x}) = \text{const}\ . \tag{IV.13}$$

Consequence for (IV.7):

$$\langle A\rangle \approx \langle A\rangle_{\text{ss}} = \frac{\sum_{m=1}^{M} A(\vec{x}_m)\mathrm{e}^{-\beta\mathrm{U}(\vec{\mathrm{x}}_\mathrm{m})}}{\sum_{m=1}^{M} \mathrm{e}^{-\beta\mathrm{U}(\vec{\mathrm{x}}_\mathrm{m})}} \tag{IV.14}$$

Here we immediately see the main caveat of the simple sampling method; it does not resolve the problem (b) mentioned before with respect to the regular selection of configuration space points because the method does not discriminate between important and unimportant regions of Γ for the calculation of thermal averages. Thus, it can only be a good simulation method for systems with $U(\vec{x}) = 0$, which is, of course, of minor interest in statistical physics.

Example: Modeling Brownian Motion as a Random Walk

Nevertheless, it is illustrative to consider an example where simple sampling can be applied. This example deals with the phenomenon of Brownian motion and its relationship with a *random walk*. To establish this relationship let us briefly consider the discovery of Brownian motion and its interpretation by Einstein.

In 1827 the botanist Robert Brown observed a highly irregular motion of pollen particles dispersed in water. Numerous experiments carried out by Brown himself and other researchers in the 19th century revealed that this irregular motion—later on called 'Brownian motion'—is not of biological nature, but must have a mechanistic origin. In 1905 Einstein suggested an interpretation by invoking the—at that time not established—concept of thermal motion. He interpreted the irregular motion of the (large) Brownian particle (BP) as a consequence of the incessant collisions with surrounding (much smaller) solvent particles. Indeed, we may write quite generally for the thermal velocity

$$v = \sqrt{\frac{3k_{\text{B}}T}{M}} \approx \begin{cases} 500\,\text{m/s},\ \text{solvent}\ (1\text{Å})\ , \\ 3\,\text{mm/s},\ \text{Brownian particles}\ (1\mu\text{m})\ , \end{cases} \tag{IV.15}$$

which implies a clear separation of time scales between the fast solvent particles and the slow Brownian particle

$$t_{\text{solvent}} \approx 10^{-13}\,\text{s} \ll t_{\text{BP}} \approx 10^{-4}\,\text{s}\ . \tag{IV.16}$$

Since only the Brownian particle is large and slow enough so that we can observe it under a microscope, this observation averages of the "fast degrees of freedom" given by the solvent particles, and the motion of the Brownian particles becomes random.

We can characterize the properties of this random motion by measuring the displacement of the Brownian particle via the following steps:

1. measure the position of the Brownian particle at time t_{i-1}: $\vec{r}_{i-1,m}$
2. wait a time interval $\tau = t_i - t_{i-1}$
3. measure the position of the Brownian particle again: $\vec{r}_{i,m}$
4. repeat steps 1. to 3. N times

   Steps 1. to 3. represent *one realization* of the stochastic process corresponding to Brownian motion. This realization is a point in configuration space given by $\vec{x}_m = (\vec{r}_{0,m}, \ldots, \vec{r}_{N,m})$.
5. calculate the total displacement $\vec{R}(\vec{x}_m) = \vec{r}_{N,m} - \vec{r}_{0,m}$
6. repeat steps 1. to 5. M times for statistical averaging

If this experiment is carried out, the following observations can be made:

(a) displacements $\Delta\vec{r}_{i,m} = \vec{r}_{i,m} - \vec{r}_{i-1,m}$ of successive time intervals are independent of each others

(b) the direction of $\Delta\vec{r}_{i,m}$ is random

Consequences:

- Elementary displacements have a vanishing mean value and are uncorrelated:

$$\Delta\vec{r}_{i,m} = \vec{r}_{i,m} - \vec{r}_{i-1,m} \tag{IV.17}$$

$$\Rightarrow \quad \langle \vec{r}_i \rangle = \lim_{M\to\infty} \frac{1}{M} \sum_{m=1}^{M} \Delta\vec{r}_{i,m} \overset{\text{observa.(b)}}{=} 0 \, . \tag{IV.18}$$

$$\Rightarrow \quad \langle \Delta\vec{r}_i \cdot \Delta\vec{r}_j \rangle \overset{\text{observa.(a)}}{=} \left\langle (\Delta\vec{r}_i)^2 \right\rangle \delta_{ij} =: l^2 \delta_{ij} \quad (l = \text{step length}) \, . \tag{IV.19}$$

- The total displacement ("end-to-end distance"),

$$\vec{R}(\vec{x}_m) = \vec{r}_{N,m} - \vec{r}_{0,m} = \sum_{i=1}^{N} \Delta\vec{r}_{i,m} \, , \tag{IV.20}$$

  has the following properties:

  - Its average value vanishes,

$$\langle \vec{R} \rangle \overset{\text{(IV.18)}}{=} 0 \, . \tag{IV.21}$$

  - Its variance is proportional to the number of steps N:

$$\langle \vec{R}^2 \rangle = \sum_{i,j=1}^{N} \langle \Delta\vec{r}_i \cdot \Delta\vec{r}_j \rangle \overset{\text{(IV.19)}}{=} \sum_{i,j=1}^{N} l^2 \delta_{i,j} = N l^2 \, . \tag{IV.22}$$

  - The distribution of $\vec{R}$ after N steps, $p_N(\vec{R})$, is gaussian because $\vec{R}$ is sum of the random variables $\Delta\vec{r}_{i,m}$ which
    - ∗ are independent and identically distributed,
    - ∗ have finite first and second moments, i.e., $\langle \Delta\vec{r}_i \rangle < \infty$, $\langle (\Delta\vec{r}_i)^2 \rangle < \infty$.

If a random sum variable satisfies these conditions, it is gaussian distributed due to the central limit theorem. So we have (in three dimensions)

$$p_N(\vec{R}) \overset{N\to\infty}{\longrightarrow} \left(\frac{3}{2\pi\langle\vec{R}^2\rangle}\right)^{3/2} \exp\left[-\frac{3\vec{R}^2}{2\langle\vec{R}^2\rangle}\right] . \tag{IV.23}$$

Starting from the experimental observations we may now introduce a simple model—the random walk—which enables us to simulate Brownian motion numerically. For reasons of simplicity we restrict ourselves to two dimensions. The model is characterized by the following features:

- We replace the spatial continuum by a square lattice. The position of the Brownian particle, i.e., of a random walker, is characterized by the lattice site (x, y).
- The walker can jump to adjacent sites only. So, l equals the lattice constant, and there are only 4 jump directions instead of infinitely many.
- All jump directions are equally probable. That is, they are chosen with probability $1/4$.
- Successive jumps are uncorrelated.

Clearly, since there is no potential energy involved in this simulation, simple sampling becomes correct, and we have

$$\overset{(\mathrm{IV}.14)}{\Longrightarrow} \quad \langle A\rangle \approx \langle A\rangle_{\mathrm{ss}} \overset{U=0}{=} \frac{1}{M}\sum_{m=1}^{M} A(\vec{x}_m) . \tag{IV.24}$$

A pseudo-code for the random walk can be written down as follows:

```
PROGRAM rw                             Monte Carlo program for a random walk on a
                                       square lattice
INTEGER irw                            uniformly distributed integer random number

read N,M                               read length N of a walk and total number M
                                       of repetitions for statistical averaging

do m = 1,M
   x =0                                definition of the origin
   y=0
   do i = 1,N
      irw = 4*ranf(i) + 1              ranf() generates a real random number
                                       uniformly distributed in [0,1[
      if(irw.eq.1) x = x + 1           step to the right
      if(irw.eq.2) y = y + 1           up-step
      if(irw.eq.3) x = x - 1           step to the left
      if(irw.eq.4) y = y - 1           down-step

      Rx2(i) = Rx2(i) + x*x            squared x-component of the total displacement
                                       after i steps

      Ry2(i) = Ry2(i) + y*y
   enddo
   calculate P(Rx), P(Ry)              probability to find the value Rx (Ry) for the
                                       x (y) component of the total displacement
enddo

do i =1,N
   write i, Rx2(i), Ry2(i)
   if(i.eq.N) write P(Rx), P(Ry)
enddo
END
```

IV.3.2 Importance Sampling

Definition: Importance sampling := choose $\vec{x}_m$ according to a distribution which minimizes the variance σ_M^2 [see Eq. (IV.12)]

$$\min_{P_{\rm sim}} \sigma_M^2[P_{\rm sim}] \quad \Rightarrow \quad P_{\rm sim}^{\rm is} \ . \tag{IV.25}$$

This implies that we have to calculate

$$\frac{\delta}{\delta P_{\rm sim}} \sigma_M^2[P_{\rm sim}] \overset{!}{=} 0 \quad \text{with the constraint} \quad \int {\rm d}^3\vec{x} P_{\rm sim}(\vec{x}) = 1 \ ,$$

where $\delta/\delta P_{\rm sim}$ denotes the functional derivative with respect to $P_{\rm sim}$. To account for the constraint we introduce a Lagrance multiplier λ. Thus,

$$\begin{aligned}
&\frac{\delta}{\delta P_{\rm sim}(\vec{x})} \left[\int {\rm d}^{3N}\vec{y}\, P_{\rm sim}(\vec{y}) \left(\frac{{\rm e}^{-\beta U(\vec{y})}}{P_{\rm sim}(\vec{y})} \right)^2 - Z_{\rm c}^2 + \lambda \left(\int {\rm d}^{3N}\vec{y}\, P_{\rm sim}(\vec{y}) - 1 \right) \right] \\
&= \int {\rm d}^{3N}\vec{y} \left\{ \underbrace{\frac{\delta P_{\rm sim}(\vec{y})}{\delta P_{\rm sim}(\vec{x})}}_{=\delta(\vec{y}-\vec{x})} \left(\frac{{\rm e}^{-\beta U}}{P_{\rm sim}} \right)^2 + P_{\rm sim}(\vec{y}) \left[2 \left(\frac{{\rm e}^{-\beta U}}{P_{\rm sim}} \right) \left(-\frac{{\rm e}^{-\beta U}}{P_{\rm sim}^2} \right) \right] \underbrace{\frac{\delta P_{\rm sim}(\vec{y})}{\delta P_{\rm sim}(\vec{x})}}_{=\delta(\vec{y}-\vec{x})} \right\} \\
&\quad + \lambda \int {\rm d}^{3N}\vec{x}\, \delta(\vec{y}-\vec{x}) \\
&= -\left(\frac{{\rm e}^{-\beta U(\vec{x})}}{P_{\rm sim}(\vec{x})} \right)^2 + \lambda \overset{!}{=} 0 \quad \Rightarrow \quad P_{\rm sim}^{\rm is}(\vec{x}) = \frac{1}{\sqrt{\lambda}} {\rm e}^{-\beta U(\vec{x})} \ .
\end{aligned}$$

The Lagrange multiplier is determined by the normalization condition, i.e., $\int {\rm d}^{3N}\, \vec{x} P_{\rm sim}^{\rm is}(\vec{x}) = 1 \overset{\rm (IV.4)}{=} \frac{1}{\lambda} Z_{\rm c}$ so that we finally have

$$P_{\rm sim}^{\rm is}(\vec{x}) = p_{\rm c}(\vec{x}) = \frac{1}{Z_{\rm c}} {\rm e}^{-\beta U(\vec{x})} \tag{IV.26}$$

Thus, the canonical distribution $p_{\rm c}(\vec{x})$ minimizes the variance σ_M.

This result may be interpreted as follows: The statistical accuracy of the Monte Carlo simulation will be optimal if we sample points from configuration space according to the canonical distribution. A method to achieve this goal was proposed by Metropolis (see Sec. IV.3.2). If the configuration space points are chosen according to $p_{\rm c}(\vec{x})$, Eq. (IV.7) simplifies; it becomes a simple arithmetic mean, i.e.,

$$\langle A \rangle \approx \langle A \rangle_{\rm is} = \frac{1}{M} \sum_{m=1}^{M} A(\vec{x}_m) \ . \tag{IV.27}$$

Metropolis Method

The Metropolis method is a Monte Carlo simulation method that generates a distribution of configuration space points, which tends toward the canonical distribution in the large-M limit. The key idea of the method consists in creating a sequence of points $\vec{x}_m$ via a Markov process.

Definition: Markov process := stochastic process in which the future state x_{m+1} depends *only* on the present state x_m, but *not* on the whole history of the motion $\{\vec{x}_i\}_{i<m}$.

To proceed let us introduce the following continuum notation:

$$\vec{x}_m \quad \rightarrow \quad (\vec{x}, t) \mathrel{\widehat{=}} \vec{x} \text{ is adopted at time } t \ (\mathrel{\widehat{=}} m) \ .$$

A Markov process may be characterized by the *Master equation*, an equation of motion for the probability $P(\vec{x},t)$ of finding $\vec{x}$ at time t. The Master equation is given by

$$\frac{\partial P(\vec{x},t)}{\partial t} = \underbrace{\sum_{\vec{x}'} W(\vec{x}|\vec{x}')P(\vec{x}',t)}_{\text{flux } \vec{x}' \to \vec{x}} - \underbrace{\sum_{\vec{x}'} W(\vec{x}'|\vec{x})P(\vec{x},t)}_{\text{flux } \vec{x} \to \vec{x}'} \ . \tag{IV.28}$$

where $W(\vec{x}|\vec{x}')$ denotes the conditional probability that the system performs a transition from state $\vec{x}'$ to state $\vec{x}$ in the time interval $\mathrm{d}t$ (therefore, W is also called 'transition rate').

The Master equation has the following interpretation: The term $W(\vec{x}|\vec{x}')P(\vec{x}',t)$ represents the probability (per unit time) that the system which occupies state $\vec{x}'$ at time t moves to state $\vec{x}$ in the following time interval $\mathrm{d}t$. By summing over $\vec{x}'$ we account for all transitions from all states $\vec{x}'$ toward the state $\vec{x}$. The first term in Eq. (IV.28) thus describes the increase of the population of state $\vec{x}$. Conversely, the second term represents the decrease of the population of $\vec{x}$ due to transitions from $\vec{x}$ to all possible states $\vec{x}'$.

The time evolution of $P(\vec{x},t)$ in the Master equation is determined by the transition rates W which we have to specify in such a way that the Markov process generates a stationary distribution that corresponds to the canonical distribution. That is, we require that

$$P(\vec{x},t) \overset{t\to\infty}{\longrightarrow} P^{\text{is}}_{\text{sim}}(\vec{x}) \overset{(\text{IV.26})}{\sim} \mathrm{e}^{-\beta U(\vec{x})} \ . \tag{IV.29}$$

Since the stationary distribution is independent of t, we have

$$\frac{\partial P^{\text{is}}_{\text{sim}}}{\partial t} = 0 \quad \Leftrightarrow \quad \sum_{\vec{x}'} \Big[W(\vec{x}|\vec{x}')P^{\text{is}}_{\text{sim}}(\vec{x}') - W(\vec{x}'|\vec{x})P^{\text{is}}_{\text{sim}}(\vec{x}) \Big] = 0 \ .$$

A sufficient (but not necessary) condition to satisfy this equation is that each term in the angular brackets vanishes. Thus, we require the condition of *detailed balance* which is given by

$$W(\vec{x}|\vec{x}')P^{\text{is}}_{\text{sim}}(\vec{x}') = W(\vec{x}'|\vec{x})P^{\text{is}}_{\text{sim}}(\vec{x}) \ . \tag{IV.30}$$

With $P^{\text{is}}_{\text{sim}}(\vec{x}') \sim \mathrm{e}^{-\beta U(\vec{x}')}$, Eq. (IV.30) becomes

$$\frac{W(\vec{x}'|\vec{x})}{W(\vec{x}|\vec{x}')} = \frac{P^{\text{is}}_{\text{sim}}(\vec{x}')}{P^{\text{is}}_{\text{sim}}(\vec{x})} = \exp\Big[-\beta\Big(U(\vec{x}') - U(\vec{x}) \Big) \Big] \ . \tag{IV.31}$$

The solution of Eq. (IV.31) is not unique because it only determines the ratio of the W's, but not W itself. However, to find a solution we can argue as follows:

- If $U(\vec{x}') - U(\vec{x}) < 0$, $P^{\text{is}}_{\text{sim}}(\vec{x}') > P^{\text{is}}_{\text{sim}}(\vec{x})$. So the new state $\vec{x}'$ to which the transition shall occur is more probable than the original state $\vec{x}$. A transition decreasing the potential energy should always be accepted.
- If $U(\vec{x}') - U(\vec{x}) > 0$, $P^{rmis}_{\text{sim}}(\vec{x}') < P^{rmis}_{\text{sim}}(\vec{x})$, and the new state is less likely than the original one. In this case, the transition would increase the potential energy. It may also be accepted, but only with probability according to the Boltzmann factor of Eq. (IV.31).

A solution which realizes this idea that the Markov process is capable of moving up and down the potential energy landscape, is the *Metropolis transition rate*:

$$W(\vec{x}'|\vec{x}) = \begin{cases} \exp\Big[-\beta\Big(U(\vec{x}') - U(\vec{x}) \Big) \Big] & U(\vec{x}') - U(\vec{x}) > 0 \ , \\ 1 & U(\vec{x}') - U(\vec{x}) \le 0 \ . \end{cases} \tag{IV.32}$$

The numerical procedure to turn this machinery into a practical simulation method employs the following steps (see Fig. IV.1):

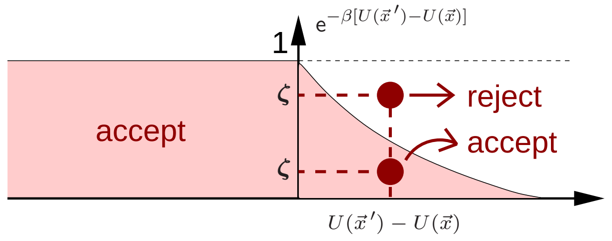
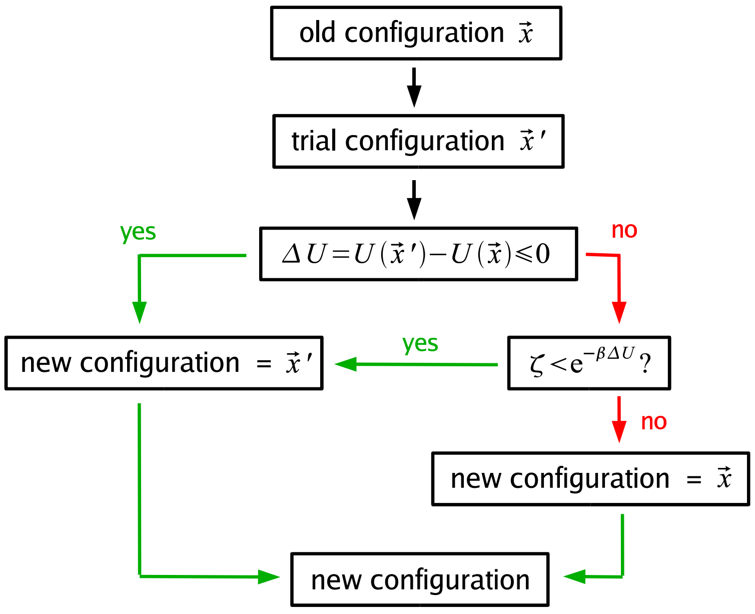


Figure IV.1: Illustration of the Metropolis criterion for the acceptance of a Monte Carlo move. If the potential energy $U(\vec{x}')$ of the (new) proposed configuration is smaller than that of the (old) initial configuration $\vec{x}$, the proposition ('move') is always excepted. Otherwise, it is only accepted according to the Boltzmann factor $e^{-\beta\Delta U}$. Practically, this is realized by comparing a real random number $\zeta \in [0,1[$ with $e^{-\beta\Delta U}$. The move is accepted if $e^{-\beta\Delta U} \geq \zeta$.

1. We calculate the energy difference $\Delta U = U(\vec{x}') - U(\vec{x})$.
2. We 'draw' a real random number ζ that is uniformly distributed between 0 and 1. (This step requires the application of a so-called 'random number generator', cf. Sec. IV.4).
3. We compare ζ to the Boltzmann factor $e^{-\beta\Delta U}$, and

$$\begin{aligned} &\text{accept the transition if } e^{-\beta\Delta U} \geq \zeta\,, \\ &\text{reject the transition if } e^{-\beta\Delta U} < \zeta\,. \end{aligned} \tag{IV.33}$$

Figure IV.2 shows the flowchart for the Monte Carlo importance sampling.



old configuration $\vec{x}$
trial configuration $\vec{x}'$
$\Delta U = U(\vec{x}') - U(\vec{x}) \leqslant 0$
yes
no
new configuration = $\vec{x}'$
yes
$\zeta < e^{-\beta\Delta U}$?
no
new configuration = $\vec{x}$
new configuration

Figure IV.2: Flowchart for Monte Carlo importance sampling. If the move is rejected according to Eq. (IV.33), the initial configuration has to counted once again, i.e., is taken as a new configuration, for statistical averaging [see discussion below Eqs. (IV.42) and (IV.43)].

We want to illustrate the Metropolis Monte Carlo method by the example of the simulation of a 'Lennard-Jones (LJ) system'.

Definition: LJ system := a system of electrically neutral, spherical particles which interact via a Lennard-Jones pair potential given by

$$U_{\rm LJ}(r) = 4\epsilon\Big[\Big(\frac{\sigma}{r}\Big)^{12} - \Big(\frac{\sigma}{r}\Big)^{6}\Big] \quad \text{with} \quad r = |\vec{r}_i - \vec{r}_j| \, , \tag{IV.34}$$

where $\vec{r}_i$ denotes the position of the ith particle ($i = 1, \ldots, N$).

The LJ potential has the following properties:

- The LJ potential is characterized by two parameters, the particle diameter σ and the strength ϵ of interaction between two particles.
- The LJ potential is strongly repulsive at short interparticle distances ($U_{\rm LJ}(r) \sim 1/r^{12}$), but attractive at larger distances ($U_{\rm LJ}(r) \sim -1/r^{6}$).

  While the repulsive part is empirical, the attractive part can be justified theoretically. It corresponds to the attraction of temporary electrical dipoles ('van der Waals attraction').

The LJ potential was introduced by J. Lennard-Jones in 1924 [Proc. Roy. Soc. **106**, 463 (1924)]. It has become a commonly employed model in simulations of condensed matter systems because it represents a simple—and thus computationally convenient—but realistic interaction potential for neutral particles.

In the following we want to discuss some details of the simulation:

(a) *Truncation of the interaction range.* In essentially all current simulations, one does not work with the full LJ potential of Eq. (IV.34), but with a LJ potential that is *truncated* (t) at some distance $r_{\rm c}$ and *shifted* (s) to zero there. That is,

$$U_{\rm LJ}(r) \;\rightarrow\; U_{\rm LJ}^{\rm ts} = \begin{cases} U_{\rm LJ}(r) - U_{\rm LJ}(r_{\rm c}) & r \le r_{\rm c} \, , \\ 0 & \text{otherwise} \, . \end{cases} \tag{IV.35}$$

The reason for this choice is as follows. Let us assume that in the course of the Monte Carlo simulation a displacement of particle i is proposed (cf. Fig. IV.3), i.e.,

$$\vec{r}_i \rightarrow \vec{r}_i^{\,\prime} = \vec{r}_i + \Delta\vec{r} \, . \tag{IV.36}$$

This displacement entails to a modification of the configuration from $\vec{x}$ to $\vec{x}'$ and thus a change of potential energy which is given by

$$\Delta U = \sum_{\substack{j=1\\ j\neq i}}^{N} U_{\rm LJ}\left(|\vec{r}_i^{\,\prime} - \vec{r}_j|\right) - \sum_{\substack{j=1\\ j\neq i}}^{N} U_{\rm LJ}\left(|\vec{r}_i - \vec{r}_j|\right) \, . \tag{IV.37}$$

The first term represents the interaction of particle i with all other particles j *after* the step. Since there are $N-1$ other particles, the numerical calculation of this term is an operation of order N—this is very time consuming in the large-N limit. Similarly, the second term represents the interaction of i with all other particles j *before* the step; this term is already known from previous calculation.

Thus, by truncating $U_{\rm LJ}$ one reduces the range of the interaction and thus the number of particles which have to be taken into account in the calculation of ΔU. Thus, the calculation of ΔU is no longer extensive in the particle number N. Of course, when comparing to theorectial or experimental results one has to correct for the error introduced by neglecting the 'tail' of the LJ potential. Such 'tail corrections' can be carried out after the simulation (for further details see Sec. VI.2.3).

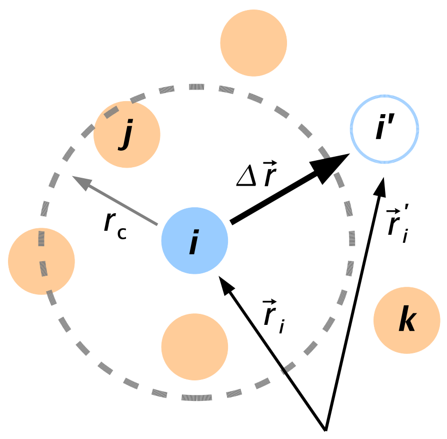


Figure IV.3: A displacement $\Delta\vec{r}$ of particle i from its initial position $\vec{r}_i$ to a new position $\vec{r}_i'$ is proposed. This displacement changes the potential energy of the system, from $U(\vec{x})$ with $\vec{x} = (\ldots, \vec{r}_i, \ldots)$ to $U(\vec{x}\,')$ with $\vec{x}\,' = (\ldots, \vec{r}_i', \ldots)$. If the pairwise interparticle potential is made short-ranged by truncating the potential at distance r_c [see Eq. (IV.35)], only the neighboring particles within the interaction range r_c around i and i' have to be taken into account in the calculation of potential energies $U(\vec{x})$ and $U(\vec{x}\,')$.

(b) *Realization of the transition $\vec{x} \to \vec{x}'$.* The elementary step of a Monte Carlo simulation can be decomposed into two steps:

1. We choose a particle i and a displacement $\Delta\vec{r}$ according to some probability $\alpha(\vec{x} \to \vec{x}')$.
2. We accept this proposition according to the probability $\mathrm{acc}(\vec{x} \to \vec{x}')$.

As these two steps are, by construction, independent of each others, we may pose

$$W(\vec{x}'|\vec{x}) \equiv W(\vec{x} \to \vec{x}') = \alpha(\vec{x} \to \vec{x}')\,\mathrm{acc}(\vec{x} \to \vec{x}') \,. \tag{IV.38}$$

Detailed balance now implies

$$\frac{W(\vec{x}'|\vec{x})}{W(\vec{x}|\vec{x}')} = \frac{\alpha(\vec{x} \to \vec{x}')\,\mathrm{acc}(\vec{x} \to \vec{x}')}{\alpha(\vec{x}' \to \vec{x})\,\mathrm{acc}(\vec{x}' \to \vec{x})} \overset{\text{(IV.31)}}{=} \exp\Big[-\beta\Big(U(\vec{x}') - U(\vec{x})\Big)\Big] \,. \tag{IV.39}$$

A commonly made choice is

$$\alpha(\vec{x} \to \vec{x}') = \alpha(\vec{x}' \to \vec{x}) \,. \tag{IV.40}$$

This means that the probability to propose a transition is symmetric for the forward and the backward moves. Equation (IV.40) is satisfied if

- we choose a particle i at random, i.e., with probability $1/N$;
- we also choose $\Delta\vec{r} = (\Delta x, \Delta y, \Delta z)$ at random. This means that the components of $\Delta\vec{r}$ are uniformly distributed between $-\Delta/2$ and $\Delta/2$,

$$\Delta x = \Big(\zeta_x - \frac{1}{2}\Big)\Delta, \;\; \Delta y = \Big(\zeta_y - \frac{1}{2}\Big)\Delta, \;\; \Delta z = \Big(\zeta_z - \frac{1}{2}\Big)\Delta \,, \tag{IV.41}$$

where $\zeta_{x,y,z}$ are real random numbers uniformly distributed between 0 and 1, and Δ represents the magnitude of the displacement. In Eq. (IV.41), $1/2$ must be substracted so that positive and negative displacements are equally likely to occur.

(c) *What to do if the transition $\vec{x} \to \vec{x}'$ is rejected?* In the time interval $\mathrm{d}t$ a transition from $\vec{x}$ to some state $\vec{x}'$ certainly occurs. This implies that

$$\sum_{\vec{x}'} W(\vec{x}'|\vec{x}) = 1 \,. \tag{IV.42}$$

A direct consequence of this normalization is that

$$\begin{aligned} W(\vec{x}|\vec{x}) &= \text{probability to remain in state } \vec{x} = \text{probability that the move } \vec{x} \to \vec{x}' \text{ is rejected} \\ &= 1 - \sum_{\vec{x}' \neq \vec{x}} W(\vec{x}'|\vec{x}) \geq 0 \; . \end{aligned} \tag{IV.43}$$

This means that the analysis of the simulation results may not distinguish between accepted and rejected moves. All moves contribute equally to the calculation of thermal averages (see Fig. IV.2).

IV.4 Introduction to the Generation of Random Numbers

Monte Carlo simulations make extensive use of random numbers. These numbers are produced by a computer program, a so-called *random number generator*. Of course, the quality of the results from Monte Carlo simulations sensitively depends on the quality of the random number generator. It is therefore important to have criteria which allow one to determine whether a random number generator is 'good' or 'bad'. We briefly sketch such criteria in the following.

Definition: A random number generator (RNG) is a computer program—that is, a deterministic sequence of instructions—which, starting from an initial value, the so-called "seed", generates a sequence of (pseudo-) random numbers $\{\zeta_i\}$.

This definition implies that the same seed always generates the same sequence of random numbers.

Properties: A 'good' random number generator has the following properties:

- The RNG generates random numbers that successfully pass various statistical tests. An important test is that the ζ_i must be uniformly distributed without any correlation.
- The RNG must not influence the results of the main program. That is, the physical results—explored by the main program—must obviously not depend on the employed RNG.
- The RNG should be fast and have a large period.

  A typical Monte Carlo simulation requires 10^{10} random numbers. Of course, one is not willing to accept an important slowing down of the execution of the main program due to the RNG—it should be fast. Furthermore, the period of the RNG—that is, how many random number it can generate before a repetition of the sequence occurs—must be large to avoid correlations.

IV.4.1 Uniformly Distributed Random Numbers

In this section, we briefly discuss two commonly employed RNG which provide uniformly distributed random numbers.

Linear Congruential Generator

Defintion: Starting from a seed ($= I_1 =$ positive integer) the linear congruential generator (LCG) produces a sequence of integer random numbers $\{I_i | 0 \leq I_i \leq m-1\}$ via the following recurrence relation

$$I_i = aI_{i-1} + c \pmod{m} = (aI_{i-1} + c) - \text{int}\left(\frac{(aI_{i-1} + c)}{m}\right) m \; , \tag{IV.44}$$

where $a \in \mathbb{N}$, $c \in \mathbb{N}_0$, and m is the maximum possible integer one can choose [see below the discussion of Eq. (IV.45)].

If we take $c = 0$, the random generator is called 'multiplicative LCG'. We will focus on this LCG in the following.

An important property of the LCG is that its periode is smaller than or equal to $m-1$, depending on the choice for a, c, and m. For the multiplicative LCG the maximum periode can be determined from the 'Carmichael theorem' which states:

The multiplicative LCG has the maximum period, $m-1$, if

- m is prime number and
- a is a 'primitive positive root (mod m)' of m, i.e., if

  $$a^{\frac{m-1}{q}} \pmod{m} \neq 1 \; , \tag{IV.45}$$

  where q is a prime factor of $m-1$.

Let us illustrate this theorem by an example. Suppose that $m = 41$, so $m-1 = 40$. We have to decompose 40 into prime factors, i.e. $40 = 2^3 \cdot 5$, so that $q = 3, 5$. Having determined the values of $m-1$ and q, a must simultaneously satisfy the following two equations

$$\begin{aligned} a^{40/2=20} &\pmod{41} \neq 1 \quad \text{and} \\ a^{40/5=8} &\pmod{41} \neq 1 \end{aligned}$$

to be a primitive positive root (mod 41). If we now try sucessively $a = 1, 2, 3, \ldots$, we will find that the first primitive positive root (mod 41) is $a = 6$.

In exactly the same fashion, one can determine all primitive positive roots of a 32 bit computer. In that case, the maximum possible integer,

$$m = 2^{32} - 1 = 2\,147\,483\,647 \tag{IV.46}$$

is a prime number, and analysis shows that for this (best) choice of m there are only three primitive positive roots,

$$a = 7, \; 7^5 = 16807, \; 65539 \; ,$$

from which $a = 16807$ should be chosen due to the following reason:

- The LCG displays correlations between successive random numbers; these correlations scale as $1/a$. This feature excludes $a = 7$.
- A further condition is that a should be much smaller than $\sqrt{m} \simeq 46341$. This excludes $a = 65539$.

Thus, the optimum multiplicative LCG is given by:

$$\begin{aligned} &\text{seed} = I_1 > 0 \; , \\ &I_i = 16807 I_{i-1} \pmod{2^{31} - 1} \quad \text{for } i > 1 \; . \end{aligned} \tag{IV.47}$$

Remarks:

- A sequence of real random numbers $\{\zeta_i | \zeta_i \in \mathbb{R}, 0 \leq \zeta_i < 1\}$ may be obtained by

  $$\zeta_i = \frac{I_i}{m} \; . \tag{IV.48}$$

- The maximum period equals $2^{31} - 2 \approx 2.1 \times 10^9$. This is not very large and barely sufficient for typical simulations.
- The LCG is fast, but exhibits correlations between the random numbers.

The last two points—the short period and the correlations—pose problems for Monte Carlo simulations. A solution to these problems by optimizing the LCG is suggested in *Numerical Recipes* [see `FUNCTION ran2`; W. H. Press *et al*, *Numerical Recipes* (Cambridge University Press, Cambridge, 1992)].

Instead of trying to improve the LCG one can also switch to another random number generator which has better properties by construction. Such a generator will be discussed in the next section.

Lagged-Fibonacci Generators

Definition: Starting from p integer random numbers (p seeds) a Lagged-Fibonacci generator (LFG) produces a sequence of integer random numbers $\{I_i\}_{i>p}$ by the following recurrence relation

$$I_i = I_{i-p} \diamond I_{i-q} \quad (q < p) \ , \tag{IV.49}$$

where '$\diamond$' denotes a binary operator and q is another integer.

Examples:

- If we choose $p = 2$, $q = 1$, and $\diamond = +$, we get

$$I_i = I_{i-1} + I_{i-2} \ ,$$

which yields the Fibonacci series (whence the name) $I_3 = 1$, $I_4 = 2$, $I_5 = 3$, $I_6 = 5$, ... with $I_1 = 0$ and $I_2 = 1$.

- If we choose $p = 250$, $q = 103$, and $\diamond = \oplus$, where $\oplus$ denotes the 'exclusive or' operator, i.e. the addition of bits (mod 2), we have

$$I_i = I_{i-250} \oplus I_{i-103} \ . \tag{IV.50}$$

This equation adds corresponding bits of the integers I_{i-250} and I_{i-103} to obtain I_i. Let us take as an example $I_{i-250} = (1\,0\ \ldots 1)$ and $I_{i-103} = (0\,1\ \ldots 1)$, where both integers consist of $N_{\rm bit}$ bits (typically 32 bits). Then, $I_i = I_{i-250} \oplus I_{i-103}$ implies

$$\left.\begin{array}{rcccccc} & & 1 & & 1 & & 0 \\ 1+0 & (\text{mod } 2) \rightarrow & 1 & & 0 & & 1 \\ & & \vdots & = & \vdots & \oplus & \vdots \\ 1+1 & (\text{mod } 2) \rightarrow & 0 & & 1 & & 1 \end{array}\right\} N_{\rm bit} \text{ bits} \ .$$

The random number generator based on Eq. (IV.50) is a widely spread generator called 'R250'. It successfully passes typical statistical tests, has a long period (of order 10^{75}), but is not completely free of (subtle) correlations [for a discussion of the latter issue see e.g. F. Schmid and N. B. Wilding, Int. J. Mod. Phys. **6**, 781 (1995)].

IV.4.2 Random Numbers Distributed According to Some Distribution Function

The previously discussed random number generators engender real random numbers, uniformly distributed between 0 and 1, i.e. $\{\zeta | \zeta \in \mathbb{R}, 0 \le \zeta < 1\}$. This implies that the probability to find a random number between ζ and $\zeta + {\rm d}\zeta$ is given by

$$P(\zeta){\rm d}\zeta = \begin{cases} \frac{1}{\text{length of the interval=1}}\,{\rm d}\zeta = {\rm d}\zeta & 0 \le \zeta < 1 \ , \\ 0 & \text{otherwise} \ . \end{cases} \tag{IV.51}$$

If a simulation does not require uniformly distributed random numbers, but random numbers y that are distributed according to some probability distribution $\overline{P}(y){\rm d}y$, we can convert ζ to y by the following steps. First, we perform a variable transformation

$$\overline{P}(y) = P(\zeta)\left|\frac{{\rm d}\zeta}{{\rm d}y}\right| = \begin{cases} \left|\frac{{\rm d}\zeta}{{\rm d}y}\right| & 0 \le \zeta < 1 \ , \\ 0 & \text{otherwise} \ , \end{cases} \tag{IV.52}$$

where we exploited Eq. (IV.51), i.e. $P(\zeta) = 1$, in the last step. Next, we consider the probability to find the random variable y between $-\infty$ and y [see Eq. (IV.8)]

$$F(y) = \int_{-\infty}^{y} {\rm d}y'\, \overline{P}(y') = \int_{-\infty}^{y} {\rm d}y' \left|\frac{{\rm d}\zeta}{{\rm d}y}\right| = \zeta(y) \ . \tag{IV.53}$$

Because $F(y)$ is monotonous, we can invert this equation and obtain the desired conversion:
$y = y(\zeta) = F^{-1}(y)$ (F^{-1} denotes the inverse function of F).

Numerically, this inversion can be carried out as follows (for further details see Chap. 7.2 of *Numerical Recipes*):

1. We calculate $F(y)$ by numerical integration of $\overline{P}(y)$ and tabulate the result.
2. Then, we draw a random number ζ and pose $\zeta = F(y)$.
3. Finally, we can 'read off' the corresponding value of the abcissa, y. This 'reading' may be achieved by determining the zero of $f(y) := \zeta - F(y)$.

Chapter V

Computational Methods Pertaining to Electronic Structure Calculations

V.1 Physical Motivation

Materials properties, quite generally, are determined by the distribution of electrons—by the *electronic structure*—in the material. Therefore, electronic structure calculations play a key role in modern condensed matter physics. Due to the complicated many-body interactions (of electrons and nuclei) numerical approaches find important applications in this field, and we want to describe in this chapter some computational methods which are pertinent for electronic structure calculations.

In order to obtain a feeling which methods are relevant here, let us consider a specific example, the tight-binding approximation for the electronic structure of a metal or an insulator. To calculate the electronic structure of these systems different points of view may be taken. One point of view consists in regarding the solid as a collection of weakly interacting neutral atoms. This idea may be motivated by the following 'gedankenexperiment'. We start by imagining an atomic crystal whose the lattice sites are separated by a macroscopic lattice constant. In this case, all electrons occupy the atomic levels localized at the lattice sites. If we now allow the lattice constant to shrink towards its natural value, at some point the electronic structure of the crystal can no longer be identified with the atomic orbitals because an electron in some atomic level feels the presence of neighboring atoms. This will occur if the interatomic spacing becomes comparable to the spatial extent of the electron's wave function.

The so-called 'tight-binding approximation' deals with the case in which the overlap of atomic wave functions is sufficient to account for the corrections to the picture of isolated atoms. The approximation thus assumes that the delocalization of the electrons remains weak so that the full wave function $|\psi\rangle$ of the crystal can be approximated by a linear superposition of atomic orbitals $\{|\varphi_i\rangle\}_{i=1,\dots,N}$ ('LCAO'= L̲inear C̲ombination of A̲tomic O̲rbitals). That is,

$$|\psi_{\mathrm{LCAO}}\rangle = \sum_{i=1}^{N} c_i |\varphi_i\rangle \ , \tag{V.1}$$

where c_i are (complex) coefficients. The challenge then consists in optimizing the coefficients so as to obtain the best possible description of the electron structure, given the ansatz of Eq. (V.1).

The way to achieve this goal exploits the 'Rayleigh-Ritz variation principle'. This principle may be stated as follows: Let E denote the mean energy of the system's hamiltonian $\mathcal{H}$ for the state $|\psi\rangle$, i.e.,

$$E := \frac{\langle\psi|\widehat{\mathcal{H}}|\psi\rangle}{\langle\psi|\psi\rangle} \ . \tag{V.2}$$

Evidently, E depends on $|\psi\rangle$, but its variation with respect to $|\psi\rangle$ vanishes if $|\psi\rangle$ satisfies the Schrödinger equation. Then, $|\psi\rangle$ is an eigenstate of $\mathcal{H}$ and E the corresponding eigenvalue. More precisely, we have

$$\delta E = 0 \quad \Leftrightarrow \quad \widehat{\mathcal{H}}|\psi\rangle = E|\psi\rangle \; . \tag{V.3}$$

The Rayleigh-Ritz variation principle—the variation of E with respect to $|\psi\rangle$ vanishes—is thus equivalent to the Schrödinger equation. Thus, if we insert Eq. (V.1) into Eq. (V.3) and optimize E with respect to c_i, we get the best possible coefficients based on the LCAO hypothesis.

To realize this idea the following steps are necessary. Following Eq. (V.2) we have to calculate

$$\langle\psi_{\mathrm{LCAO}}|\widehat{\mathcal{H}}|\psi_{\mathrm{LCAO}}\rangle = \sum_{ij} c_i^* c_j \underbrace{\langle\varphi_i|\widehat{\mathcal{H}}|\varphi_j\rangle}_{=\mathcal{H}_{ij}} \; , \tag{V.4}$$

$$\langle\psi_{\mathrm{LCAO}}|\psi_{\mathrm{LCAO}}\rangle = \sum_{ij} c_i^* c_j \underbrace{\langle\varphi_i|\varphi_j\rangle}_{=S_{ij}} \; , \tag{V.5}$$

where $\mathcal{H}_{ij}$ is the matrix element of the hamiltonian obtained from the orbitals of atoms i and j, and S_{ij} is the overlap integral of these orbitals (it is also a matrix). Insertion of Eqs. (V.4,V.5) into Eq. (V.3) yields

$$\begin{aligned} \frac{\partial E}{\partial c_k^*} \quad &= \quad \frac{\sum_j c_j \langle\varphi_k|\widehat{\mathcal{H}}|\varphi_j\rangle}{\sum_{ij} c_i^* c_j S_{ij}} - \frac{\sum_{ij} c_i^* c_j \mathcal{H}_{ij} \sum_j c_j S_{kj}}{\left[\sum_{ij} c_i^* c_j S_{ij}\right]^2} \\ &\overset{(\mathrm{V.2,V.4,V.5})}{=} \quad \frac{\sum_j c_j [\mathcal{H}_{ij} - E S_{ij}]}{\sum_{ij} c_i^* c_j S_{ij}} \overset{!}{=} 0 \; , \end{aligned}$$

which is satisfied if the numerator vanishes, i.e.,

$$\sum_{j=1}^{N} c_j \big[\mathcal{H}_{ij} - E S_{ij}\big] = 0 \qquad (k = 1, \ldots, N) \; . \tag{V.6}$$

This *set of linear equations* has a solution (other than $c_j = 0 \; \forall j$) provided

$$\det\big(\mathsf{H} - E\mathsf{S}\big) = 0 \; , \tag{V.7}$$

where we introduced the notation H and S for the corresponding matrices.

This example shows us which numerical procedures will be important in electronic structure calculations. We must calculate the determinant, Eq. (V.7), to obtain the eigenvalues E_n (the energy band), and for every E_n, we must solve the set of linear equations, Eq. (V.6), to find the coefficients $\{c_j^{(n)}\}_{j=1,\ldots,N}$ in order to determine the wave function $|\psi_{\mathrm{LCAO}}^{(n)}\rangle$.

V.2 Systems of Linear Equations

V.2.1 Introduction

We consider the following set of linear algebraic equations:

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \ldots & + & a_{1N}x_N & = & b_1 \; , \\ \vdots & & & & & & \vdots & & \vdots \\ a_{N1}x_1 & + & a_{N2}x_2 & + & \ldots & + & a_{NN}x_N & = & b_N \; . \end{array} \tag{V.8}$$

In this set the coefficients a_{ij}, $i, j = 1, \ldots, N$ and the right-hand sides b_i, $i = 1, \ldots, N$ are known numbers. Our aim is to determine the N unknowns x_i, $i = 1, \ldots, N$.

One approach to solving this problem could consist in writing Eq. (V.8) in matrix form,

$$\mathsf{A} \cdot \vec{x} = \vec{b}\,, \tag{V.9}$$

and in exploiting the theorem that, provided the determinant of matrix A is not zero, Eq. (V.9) has the unique solution

$$x_i = \frac{\det \mathsf{A}_i}{\det \mathsf{A}} \qquad (i = 1, \ldots, N) \tag{V.10}$$

with A_i defined by

$$\mathsf{A}_i = \begin{array}{c} \begin{pmatrix} a_{11} & \cdots & b_1 & \cdots & a_{1N} \\ \vdots & & \vdots & & \vdots \\ a_{N1} & \cdots & b_N & \cdots & a_{NN} \end{pmatrix} \\ \uparrow \\ \text{column } i \end{array}\,, \tag{V.11}$$

that is, A_i equals matrix A except that $b_1, \ldots, b_N$ substitutes the elements of A in the ith column.

In practical applications this approach is not employed. There are two main reasons for that:

- It requires the knowledge of the determinants $\det \mathsf{A}_i$ and $\det \mathsf{A}$. We will see that the numerical procedure to calculate a determinant is an operation of order N^3, which makes Eq. (V.10) of order N^4. The method we will present in Sec. V.2.2 solves Eq. (V.8) by an operation of order N^3, a tremendous advantage for large N.
- Equation (V.10) cannot be applied if Eq. (V.8) is *singular* which implies that $\det \mathsf{A} = 0$. This occurs if one or more columns (or rows) of A are a linear combination of the others. In this not uncommon case, other methods to solve Eq. (V.8) must be used. Special numerical procedures have been developed to deal with this 'singular' situation [for details see W. H. Press *et al*, *Numerical Recipes* (Cambridge University Press, Cambridge, 1992)].

In the following we will always consider that matrix A is nonsingular. Then, an often employed numerical method to solve Eq. (V.8) is the so-called 'LU decomposition'. We will discuss this method in the following section.

V.2.2 LU Decomposition

Let A be a nonsingular matrix and suppose that we can decompose A into the product of two matrices

$$\mathsf{A} = \mathsf{L} \cdot \mathsf{U}\,, \tag{V.12}$$

where L is *lower triangular*—it has elements different from 0 only on the diagonal and below—and U is *upper triangular*—it has elements different from 0 only on the diagonal and above. That is,

$$\mathsf{L} = \begin{pmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{N1} & l_{N2} & \cdots & l_{NN} \end{pmatrix} \quad \text{and} \quad \mathsf{U} = \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1N} \\ 0 & u_{22} & \cdots & u_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{NN} \end{pmatrix}\,. \tag{V.13}$$

We can use this decomposition to solve the linear set

$$\mathsf{A} \cdot \vec{x} = \big(\mathsf{L} \cdot \mathsf{U}\big) \cdot \vec{x} = \mathsf{L} \cdot \underbrace{\big(\mathsf{U} \cdot \vec{x}\big)}_{=:\,\vec{y}} = \vec{b}$$

by first solving for the vector $\vec{y}$

$$\mathsf{L}\cdot\vec{y}=\vec{b}\quad\Rightarrow\quad\vec{y}\,, \tag{V.14}$$

and then solving

$$\mathsf{U}\cdot\vec{x}=\vec{y}\quad\Rightarrow\quad\vec{x} \tag{V.15}$$

to obtain $\vec{x}$. The advantage of breaking up one set of linear equations [Eq. (V.9)] into two successive ones is that Eq. (V.14) and (V.15) can be solved iteratively by substitutions. For Eq. (V.14) a forward substitution is carried out as follows

$$\begin{aligned}
l_{11}y_1=b_1\ &\Rightarrow\ y_1=\frac{b_1}{l_{11}}\\
l_{21}y_1+l_{22}y_2=b_2\ &\Rightarrow\ y_2=\frac{1}{l_{22}}\big[b_2-l_{21}y_1\big]\\
l_{31}y_1+l_{32}y_2+l_{33}y_3=b_3\ &\Rightarrow\ y_3=\frac{1}{l_{33}}\big[b_3-l_{32}y_2-l_{31}y_1\big]\\
&\vdots
\end{aligned}$$

and thus

$$\begin{aligned}
y_1&=\frac{b_1}{l_{11}}\\
y_i&=\frac{1}{l_{ii}}\left[b_i-\sum_{j=1}^{i-1}l_{ij}y_j\right]\quad(i=2,\cdots,N)\,,
\end{aligned} \tag{V.16}$$

while for Eq. (V.15) the substitution is carried out in reverse order

$$\begin{aligned}
x_N&=\frac{y_N}{u_{NN}}\\
x_i&=\frac{1}{u_{ii}}\left[y_i-\sum_{j=i+1}^{N}u_{ij}x_j\right]\quad(i=1,\cdots,N-1)\,.
\end{aligned} \tag{V.17}$$

This yields the desired solution, $\vec{x}$, provided we can decompose A as required by Eq. (V.12). To achieve this we first write Eq. (V.12) explicitly by using Eq. (V.13),

$$i<j:\quad l_{i1}u_{1j}+l_{i2}u_{2j}+\cdots+l_{ii}u_{ij}=a_{ij}\,, \tag{V.18}$$

$$i=j:\quad l_{i1}u_{1j}+l_{i2}u_{2j}+\cdots+l_{ii}u_{jj}=a_{ij}\,, \tag{V.19}$$

$$i>j:\quad l_{i1}u_{1j}+l_{i2}u_{2j}+\cdots+l_{ij}u_{jj}=a_{ij}\,. \tag{V.20}$$

Equations (V.18)–(V.20) total N^2 equations. However, there are more unknowns than equations because L and U contain $(N^2-N)/2$ off-diagonal and N diagonal elements, in total $2[(N^2-N)/2+N]=N^2+N$ elements. Thus, N unknowns can be chosen arbitrarily, for instance,

$$l_{ii}=1\,,\quad i=1,\ldots,N\,. \tag{V.21}$$

Based on this analysis *Crout's algorithm* represents a clever procedure which solves Eqs. (V.18)–(V.20) by just arranging the equations in a certain order. This algorithm utilizes two steps:

(a) If $i=1,\ldots,j$, we use Eqs. (V.18)–(V.20) to solve for u_{ij}, i.e.,

$$u_{ij}=\begin{cases} a_{ij}\,, & i=1\,,\\ a_{ij}-\sum_{k=1}^{i=1}l_{ik}u_{ki}\,, & i=2,\cdots,j\,. \end{cases} \tag{V.22}$$

(b) If $i = j+1, \ldots, N$, we use Eq. (V.20) to determine l_{ij}, i.e.,

$$l_{ij} = \frac{1}{u_{jj}} \left[a_{ij} - \sum_{k=1}^{j-1} l_{ik} u_{kj} \right] . \qquad \text{(V.23)}$$

For each i both steps have to be done before going to next j because that calculation requires all l_{ik} and u_{ki} determined in the previous steps.

Remarks:

- Equations (V.22,V.23) have to be excecuted for $i, j = 1, \ldots, N$ so that there are N^2 operations. As Eqs. (V.22,V.23) contain sums with N terms, the LU decomposition is of order N^3.
- By using the LU decomposition it is easy to find the inverse of a matrix, A^{-1}. Let us write A^{-1} and the unit matrix 1 as

  $$\mathsf{A}^{-1} = \begin{pmatrix} \tilde{a}_{11} & \cdots & \tilde{a}_{1N} \\ \vdots & & \vdots \\ \tilde{a}_{N1} & \cdots & \tilde{a}_{NN} \end{pmatrix} \quad \text{and} \quad 1 = \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix} . \qquad \text{(V.24)}$$

  The inverse is defined by

  $$\Rightarrow \mathsf{A} \cdot \mathsf{A}^{-1} = 1 ,$$

  which can be split up as follows

  $$\mathsf{A} \cdot \begin{pmatrix} \tilde{a}_{11} \\ \vdots \\ \tilde{a}_{1N} \end{pmatrix} = \begin{pmatrix} 1 \\ \vdots \\ 0 \end{pmatrix} \quad \ldots \quad \mathsf{A} \cdot \begin{pmatrix} \tilde{a}_{1N} \\ \vdots \\ \tilde{a}_{NN} \end{pmatrix} = \begin{pmatrix} 0 \\ \vdots \\ 1 \end{pmatrix} . \qquad \text{(V.25)}$$

  Equation (V.25) represents a set of N linear equations of the form of Eq. (V.9) for the columns of A^{-1}, which we can solve using the LU decomposition.
- It is also straightforward to calculate the determinant of A because

  $$\det \mathsf{A} \overset{\text{(V.12)}}{=} \det(\mathsf{L} \cdot \mathsf{U}) = (\det \mathsf{L}) \cdot (\det \mathsf{U}) = \left(\prod_{i=1}^{N} l_{ii} \right) \cdot \left(\prod_{i=1}^{N} u_{ii} \right) \overset{\text{(V.21)}}{=} \prod_{i=1}^{N} u_{ii} . \qquad \text{(V.26)}$$

- For large sets of linear equations it is not always easy to obtain numerically accurate results equal or close to the precision of the computer's limit (which is given by the precision of the computer's floating-point word). This is due to roundoff errors which may be magnified to the extent that one can easily lose two or three significant digits.

  However, there is trick to restore the full machine precision. This trick is called *iterative improvement* of the solution and works as follows. Suppose that the vector $\vec{x}$ has the exact solution

  $$\mathsf{A} \cdot \vec{x} = \vec{b} .$$

  The LU decompostion, on the other hand, yields only the approximate result $\vec{x} + \delta\vec{x}$. Of course, if we knew the error $\delta\vec{x}$, we could immediately corret the LU result and obtain $\vec{x}$. An approximate determination of $\delta\vec{x}$ is indeed possible by the following three steps:

  1. We pose

     $$\mathsf{A} \cdot (\vec{x} + \delta\vec{x}) = \vec{b} + \delta\vec{b} . \qquad \text{(V.27)}$$

  2. Substracting Eq. (V.9) from Eq. (V.27) we get

     $$\mathsf{A} \cdot \delta\vec{x} = \delta\vec{b} \overset{\text{(V.27)}}{=} \mathsf{A}(\vec{x} + \delta\vec{x}) - \vec{b} =: \vec{b}\,' . \qquad \text{(V.28)}$$

     Here, we expressed the unknown $\delta\vec{b}$ in terms of known quantities, $\vec{x}+\delta\vec{x}$—the numerical solution of the LU decomposition—and $\vec{b}$.

3. Starting from Eq. (V.28) we can determine $\delta\vec{x}$ by a further application of the LU decomposition and thus correct the original estimate: $(\vec{x}+\delta\vec{x}) \to (\vec{x}+\delta\vec{x}) - \delta\vec{x}$.

Of course, this second application of the LU decomposition to determine $\delta\vec{x}$ could again be improved by repeating the three steps above once or even several times. Usually, one repetition should suffice to improve the result, but another iteration might be reassuring to verify the convergence of the correction procedure.

V.3 Eigenvalue Problems

V.3.1 Introduction

We consider Eq. (V.9) with the choice $\vec{b} = \lambda\vec{x}$:

$$\mathsf{A} \cdot \vec{x} = \lambda \vec{x} \; . \tag{V.29}$$

This is an *eigensystem* with *eigenvector* $\vec{x}$ and *eigenvalue* λ. The eigensystem has a solution (other than $\vec{x} = \vec{0}$) provided

$$\det(\mathsf{A} - \lambda 1) = 0 \quad \text{(‘secular or characterisitic equation’)} \; , \tag{V.30}$$

where 1 denotes again the unit matrix

$$1 = \begin{pmatrix} 1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 1 \end{pmatrix} .$$

We may expand the characteristic equation (V.30) in a Nth degree polynomial in λ,

$$\det(\mathsf{A} - \lambda 1) = c_0 + c_1\lambda + \cdots + c_{N-1}\lambda^{N-1} + (-1)^N \lambda^N \; , \tag{V.31}$$

whose roots form the set of N eigenvalues $\{\lambda\}_{i=1,\ldots,N}$. Thus, root-searching algorithms could be applied to obtain $\{\lambda\}_{i=1,\ldots,N}$. However, this is usually a very poor computational method for finding eigenvalues. We will discuss a much better way in this chapter, as well as an efficient way for finding the corresponding eigenvectors.

In the following, we will always consider the special case of a symmetric matrix. A symmetric matrix has the property that it equals its transpose,

$$\mathsf{A} = \mathsf{A}^{\mathrm{T}} \quad \text{or} \quad a_{ij} = a_{ji} \; . \tag{V.32}$$

The eigenvalues of a symmetric matrix are real, $\lambda_i \in \mathbb{R}$ for $i = 1, \ldots, N$. We suppose further that they are ‘nondegenerate’, that is, that every λ_i has a unique eigenvector $\vec{x}_i$. These eigenvectors are mutually orthogonal, i.e. (we suppose the eigenvectors to be normalized)

$$\vec{x}_i \cdot \vec{x}_j = \sum_{\alpha=1}^{N} x_{\alpha i} x_{\alpha j} = \delta_{ij} \tag{V.33}$$

with

$$\vec{x}_i = \begin{pmatrix} x_{1i} \\ \vdots \\ x_{\alpha i} \\ \vdots \\ x_{Ni} \end{pmatrix} . \tag{V.34}$$

Thus, we rewrite Eq. (V.29) as

$$\mathsf{A}\cdot\vec{x}_i = \lambda_i\vec{x}_i$$

or more explicitly

$$\begin{pmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & & \vdots \\ a_{N1} & \cdots & a_{NN} \end{pmatrix} \begin{pmatrix} x_{1i} \\ \vdots \\ x_{Ni} \end{pmatrix} = \begin{pmatrix} \lambda_i x_{1i} \\ \vdots \\ \lambda_i x_{Ni} \end{pmatrix} .$$

Now, we introduce the matrix O whose columns are given by the eigenvectors, Eq. (V.34). That is,

$$\mathsf{O} := \begin{pmatrix} x_{11} & \cdots & x_{1i} & \cdots & x_{1N} \\ \vdots & & \vdots & & \vdots \\ x_{\alpha 1} & \cdots & x_{\alpha i} & \cdots & x_{\alpha N} \\ \vdots & & \vdots & & \vdots \\ x_{N1} & \cdots & x_{Ni} & \cdots & x_{NN} \end{pmatrix} . \tag{V.35}$$

Then, we have

$$\mathsf{A}\cdot\mathsf{O} \overset{\text{(V.29)}}{=} \begin{pmatrix} \lambda_1 x_{11} & \cdots & \lambda_N x_{1N} \\ \vdots & & \vdots \\ \lambda_1 x_{N1} & \cdots & \lambda_N x_{NN} \end{pmatrix} = \mathsf{O}\cdot \mathrm{diag}(\lambda_1,\cdots,\lambda_N) \tag{V.36}$$

with the diagonal matrix

$$\mathrm{diag}(\lambda_1,\cdots,\lambda_N) = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & \vdots \\ \vdots & \vdots & \ddots & 0 \\ 0 & 0 & \cdots & \lambda_N \end{pmatrix} . \tag{V.37}$$

Multiplying Eq. (V.36) from the left by the transpose of matrix O we get

$$\mathsf{O}^{\mathrm{T}}\cdot\mathsf{A}\cdot\mathsf{O} = \left(\mathsf{O}^{\mathrm{T}}\cdot\mathsf{O}\right)\mathrm{diag}(\lambda_1,\cdots,\lambda_N)$$
$$\Rightarrow \left(\mathsf{O}^{\mathrm{T}}\cdot\mathsf{O}\right)_{ij} = \sum_{\alpha=1}^{N} \underbrace{x^{\mathrm{T}}_{\alpha i}}_{=x_{i\alpha}} x_{\alpha j} \overset{\text{(V.33)}}{=} \delta_{ij} ,$$

which shows that the matrix O is orthogonal, i.e.,

$$\mathsf{O}^{\mathrm{T}} = \mathsf{O}^{-1} . \tag{V.38}$$

This finally yields the 'similarity transformation'

$$\mathsf{O}^{-1}\cdot\mathsf{A}\cdot\mathsf{O} = \mathrm{diag}(\lambda_1,\cdots,\lambda_N) . \tag{V.39}$$

In other words, we can diagonalize the matrix A by two matrix multiplications with an orthogonal matrix O containing the eigenvectors of A in its columns. Of course, we do not know the eigenvectors and thus O at the beginning of the calculation. Nevertheless, Eq. (V.39) suggests the following computational approach:

- Choose an orthogonal matrix $\mathsf{P}_{(pq)}$ such that

  $$\mathsf{P}^{\mathrm{T}}_{(pq)}\cdot\mathsf{A}\cdot\mathsf{P}_{(pq)} = \mathsf{A}' \tag{V.40}$$

  removes the element a_{pq} from A.

- Iterate this similarity transformation by varying (pq) so that all off-diagonal elements of A are removed successively. This implies

$$\begin{aligned} \mathsf{A} &\rightarrow \mathsf{P}^{\mathrm{T}}_{(pq)} \cdot \mathsf{A} \cdot \mathsf{P}_{(pq)} = \mathsf{A}^{'} \\ &\rightarrow \mathsf{P}^{\mathrm{T}}_{(p'q')} \cdot \mathsf{A}^{'} \cdot \mathsf{P}_{(p'q')} = \mathsf{A}^{''} \\ &\rightarrow \cdots . \end{aligned} \tag{V.41}$$

  If Eq. (V.41) converges, we will have

$$\begin{aligned} &\mathsf{A} = \mathrm{diag}(\lambda_1, \cdots, \lambda_N) , \\ &\mathsf{O} = \mathsf{P}_{(pq)} \cdot \mathsf{P}_{(p'q')} \cdot \mathsf{P}_{(p''q'')} \cdots . \end{aligned} \tag{V.42}$$

  where O is defined in Eq. (V.35).

Such a sequence of similarity transformations to push the matrix A step-by-step towards diagonal form is utilized in essentially all modern eigensystem routines. An example of such a routine is the Jacobi method which we will discuss next.

V.3.2 Example: Jacobi Method

The Jacobi method realizes the idea outlined by the Eqs. (V.40)—(V.42). It consists of a sequence of similarity transformations, each of which removes one of the off-diagonal matrix elements. Successive transformations undo previous steps, but the off-diagonal elements still become smaller and smaller until the matrix is diagonal to machine precision.

The Jacobi method is commendable: it is simple and it works very well for all real symmetric matrices. A caveat may be that the method is slow for matrices of order greater than about 10. For these cases, more efficient methods are available [for further details see W. H. Press *et al*, *Numerical Recipes* (Cambridge University Press, Cambridge, 1992)].

The Jacobi method introduces the following matrix $\mathsf{P}_{(pq)}$

$$\mathsf{P}_{(pq)} = \begin{pmatrix} 1 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 \\ \vdots & \ddots & & & & & \vdots \\ & & c & \cdots & 0 & \cdots & s & & \\ & & \vdots & \ddots & & & \vdots & & \\ & & 0 & & 1 & & 0 & & \\ & & \vdots & & & \ddots & \vdots & & \\ & & -s & \cdots & 0 & \cdots & c & & \\ \vdots & & & & & & & \ddots & \vdots \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & 1 \end{pmatrix} . \tag{V.43}$$

Here all diagonal elements are 1 except for the two elements c in rows (and columns) p and q. All off-diagonal elements are 0 except the two elements s and $-s$. As $\mathsf{P}_{(pq)}$ shall be orthogonal, we require

$$\mathsf{P}^{\mathrm{T}}_{(pq)} \cdot \mathsf{P}_{(pq)} = \begin{pmatrix} 1 & \cdots & & & & & 0 \\ \vdots & \ddots & & & & & \vdots \\ & & c^2+s^2 & & & & \\ & & & \ddots & & & \\ & & & & 1 & & \\ & & & & & \ddots & \\ & & & & & & c^2+s^2 \\ \vdots & & & & & \ddots & \vdots \\ 0 & \cdots & & & & & 1 \end{pmatrix} \overset{!}{=} 1 ,$$

which gives

$$c^2 + s^2 = 1 \quad \text{so that we define } c := \cos\Phi ,\ s := \sin\Phi . \tag{V.44}$$

Now, we carry out the similarity transformation of Eq. (V.40) and obtain the following result:

$$\begin{aligned} a'_{rp} &= c a_{rp} - s a_{rq} & (r \neq p, r \neq q) \quad &\text{(a)} , \\ a'_{rq} &= c a_{rq} + s a_{rp} & (r \neq p, r \neq q) \quad &\text{(b)} , \\ a'_{pp} &= c^2 a_{pp} + s^2 a_{qq} - 2sc a_{pq} & &\text{(c)} , \\ a'_{qq} &= c^2 a_{qq} + s^2 a_{pp} + 2sc a_{pq} & &\text{(d)} , \\ a'_{pq} &= (c^2 - s^2) a_{pq} + sc(a_{pp} - a_{qq}) & &\text{(e)} . \end{aligned} \tag{V.45}$$

To eliminate a'_{pq} we set

$$a'_{pq} = 0 . \tag{V.46}$$

Equation (V.45e) thus allows us to express a_{pp} and a_{qq} in the following way:

$$\begin{aligned} a_{pp} &= a_{qq} - \frac{c^2 - s^2}{sc} a_{pq} & (*) , \\ a_{qq} &= a_{pp} + \frac{c^2 - s^2}{sc} a_{pq} & (**) . \end{aligned}$$

Insertion of $(**)$ into Eq. (V.45c) and of $(*)$ into Eq. (V.45d) yields

$$\begin{aligned} a'_{pp} &= a_{pp} - t a_{pq} , \\ a'_{qq} &= a_{qq} - t a_{pq} , \end{aligned} \tag{V.47}$$

where

$$t = s/c \overset{\text{(V.44)}}{=} \tan\Phi . \tag{V.48}$$

Furthermore, we rewrite Eqs. (V.45 a,b) as

$$\begin{aligned} a'_{rp} &= a_{rp} - s(a_{rq} + \tau a_{rp}) , \\ a'_{rq} &= a_{rq} - s(a_{rp} + \tau a_{rq}) , \end{aligned} \tag{V.49}$$

where

$$\tau = \frac{s}{1 + c} . \tag{V.50}$$

Two comments shall be made here. First, Eqs. (V.47)-(V.49) reveal that the Jacobi method is of order N^3, since for every off-diagonal element—their total number is of order N^2—Eq. (V.49) has to be carried out for $r = 1, \ldots, N$. Second, the problem, alluded to above, concerning the annihilation of previous steps becomes apparent from Eq. (V.49). This equations modifies off-diagonal terms. Thus, previous steps are annihilated by restoring $a_{pq} > 0$. Nevertheless, one can see that the Jacobi method converges by considering the sum of squares of the off-diagonal elements. Equation (V.45) implies that

$$S^{'} := \sum_{r\neq s} |a'_{rs}|^2 = \sum_{r\neq s} |a_{rs}|^2 - 2|a_{pq}|^2 = S - 2|a_{pq}|^2 \; . \tag{V.51}$$

Thus, the sequence

$$s \to s^{'} \to s^{''} \to \cdots \to 0 \; ,$$

and the method converges.

The last bit of information that we need is how the parameters c and s of Eq. (V.44) should be chosen. From Eqs. (V.45,V.46) we have

$$\begin{aligned} \Theta &\equiv \frac{a_{qq} - a_{pp}}{2a_{pq}} && \text{(V.52)} \\ &= \frac{c^2 - s^2}{2sc} \overset{\text{(V.48)}}{=} \frac{1 - t^2}{2t} \; , \end{aligned}$$

and so

$$\begin{aligned} & t^2 + 2\Theta t - 1 = 0 && \text{(V.53)} \\ \Rightarrow \quad & t_{\pm} = -\Theta \pm \sqrt{\Theta^2 + 1} \; . && \text{(V.54)} \end{aligned}$$

Empirically, it was found that the smaller root gives the most stable reduction. Thus,

$$\text{if } \Theta < 0\text{:} \quad t_+ > t_- = |\Theta| - \sqrt{\Theta^2 + 1} = \frac{-1}{|\Theta| + \sqrt{\Theta^2 + 1}} \; ,$$

and

$$\text{if } \Theta > 0\text{:} \quad |t_-| > |t_+| \;\Rightarrow\; t+ = -|\Theta| + \sqrt{\Theta^2 + 1} = \frac{1}{|\Theta| + \sqrt{\Theta^2 + 1}} \; .$$

In summary, we thus find

$$t = \frac{\text{sgn}(\Theta)}{|\Theta| + \sqrt{\Theta^2 + 1}} \quad (\Theta \neq 0) \; , \tag{V.55}$$

and so

$$\begin{aligned} & \overset{\text{(V.44)}}{\Longrightarrow} \quad c = \frac{1}{\sqrt{1 + t^2}} \; , && \text{(V.56)} \\ & \overset{\text{(V.48)}}{\Longrightarrow} \quad s = tc = \frac{t}{\sqrt{1 + t^2}} \; . && \text{(V.57)} \end{aligned}$$

This concludes our presentation of the Jacobi method.

V.4 Finding the Extrema of a Function

V.4.1 Introduction

The computational problem we are facing here can be stated very simply: We have a function f that depends on one or more variables,

- $f(x) \quad (x \in \mathbb{R})$,
- $f(\vec{x}) \quad (\vec{x} = (x_1, \cdots, x_N) \in \mathbb{R}^N)$,

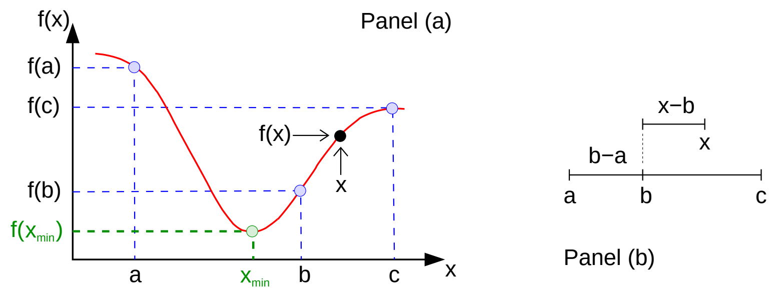


Figure V.1: Schematics to illustrate the golden section search method. Panel (a) shows the different abcissa values, a, b, c and x, as well as the corresponding ordinate values, which allow to bracket the minimum of the function $f(x)$. Panel (b) illustrates the 'similarity condition', Eq. (V.65), for the length of the subintervals in two sucessive iterations of the method.

and we want to find the value of those variables where f takes on a maximum or minimum value. The task of maximization and minimization are, of course, related to one another, since the maximum of f is the minimum of $-f$. So it suffices to focus on, say, procedures allowing to minimize a function.

The minimum can be global (truly the lowest function value) or local (lowest function value in some neighborhood). While every minimization routine will find local minima, finding a global minimum is a very difficult problem. Two standard approaches are commonly employed: (i) starting from widely varying initial values local minima are determined and the lowest of these is picked; (ii) or a previously found local minimum is perturbed by moving a finite step away from it and it is checked whether the routine returns to this minimum or finds another (deeper) one. An alternative to these trial-and-error approaches are *simulated annealing methods* (see Sec. V.4.4).

Unfortunately, there is no 'perfect' minimization algorithm, in sense of being the optimal method for all conceivable problems. In the case of minimization, one should rather try more than one method and compare the results in order to single out the best one for the problem under consideration. The following sections should be taken as an example only. They present commonly employed methods for one-dimensional minimization (Sec. V.4.2) and multidimensional minimization (Sec. V.4.3).

V.4.2 Minimization in One Dimension: Golden Section Method

The 'golden section search' method can be considered as a direct 'translation' of the bisection method for finding the roots of a function to the minimization of a function. The key idea of the bisection method is to approach the root iteratively by bracketing it more and more. A root is bracketed by a pair of points, a and b, when the function has opposite sign at those two points. For a minimum, however, the situation is more complicated. It is known to be bracketed only when there is a triplet of points

$$a < b < c \quad \text{such that} \quad f(b) < f(a) \;\text{ and }\; f(b) < f(c)\,. \tag{V.58}$$

The analog of the bisection method therefore runs as follows [see Fig. V.1, panel (a)]:

1. First, we choose a point x, for instance, between b and c, so that $b < x < c$.

2. If $f(b) < f(x)$, we take (a, b, x) as the new interval for bracketing the minimum. Otherwise, we choose (b, x, c).

3. We iterate steps 1. and 2. until the method converges.

A criterion to decide whether the search has converged or not can be established as follows. We make a Taylor expansion of $f(x)$ around the minimum position $x_{\min}$,

$$f(x) \approx f(x_{\min}) + \frac{1}{2} f''(x)\big|_{x=x_{\min}} (x - x_{\min})^2 \ ,$$

and we stipulate to stop the search if

$$\frac{1}{2} f''(x_{\min})(x - x_{\min})^2 \ll |f(x_{\min})| \ ,$$

that is, if

$$\frac{1}{2} f''(x_{\min})(x - x_{\min})^2 = \epsilon |f(x_{\min})|$$

or

$$|x - x_{\min}| = \sqrt{\epsilon}\, |x_{\min}| \sqrt{\frac{2|f(x_{\min})|}{x_{\min}^2 f''(x_{\min})}} \ , \tag{V.59}$$

where ϵ denotes the computer's floating-point precision. In many cases, the square-root term is of order 1 so that, as a rule of thumb, the search can be stopped if

$$\left.\begin{array}{l} \dfrac{|x - x_{\min}|}{x_{\min}} = \sqrt{\epsilon} \\ \text{with } \epsilon = \text{ numerical precision of } x\text{:} \\ \bullet\ x = \text{double precision: } \epsilon \approx 10^{-15} \ , \\ \bullet\ x = \text{real: } \epsilon \approx 3 \cdot 10^{-8} \ . \end{array}\right\} \tag{V.60}$$

Now, the remaining task is to decide how to choose x, given (a, b, c). To this end, we introduce the ratios

$$\frac{b-a}{c-a} =: \omega < 1 \, , \quad \frac{c-b}{c-a} = 1 - \omega \tag{V.61}$$

to eliminate the explicit dependence on the length of the interval $(c - a)$. With that choice, we then also introduce

$$0 < z = \frac{x-b}{c-a} < 1 \ , \tag{V.62}$$

which requires $b < x < c$, as initially assumed. Following the above description of the golden section method we distinguish the following cases:

$$\text{If } f(b) < f(x) \overset{(2.)}{\Longrightarrow} (a, b, x) \overset{(\text{V.61,V.62})}{\Longleftrightarrow} z + \omega = \frac{x-a}{c-a} \ .$$
$$\text{If } f(b) > f(x) \overset{(2.)}{\Longrightarrow} (b, x, c) \overset{(\text{V.61})}{\Longleftrightarrow} 1 - \omega \ .$$

The best compromise between both cases is to pose

$$z + \omega \overset{!}{=} 1 - \omega \ , \tag{V.63}$$

such that

$$z = 1 - 2\omega \ \Leftrightarrow \ x = c - (b - a) \ . \tag{V.64}$$

Since $b < x$, this implies that

$$b < x = c - (b - a) \ \Leftrightarrow \ b - z < c - b \quad \text{or} \quad b < \frac{1}{2}(c + a) \ \Rightarrow \ b - a < \frac{1}{2}(c - a) \overset{(\text{V.61})}{=} \omega < \frac{1}{2} \ .$$

In other words, Eq. (V.63) places x in the larger of the two subintervals $[a, b]$ and $[b, c]$. Let us assume that this is the interval $[b, c]$; so $[a, b]$ is the smaller subinterval of length $b - a$. Since ω is determined by the ratio $(b-a)/(c-a)$ [Eq. (V.61)], and we want ω to be constant for all iterations, we require [see Fig. V.1, panel (b)]

$$\frac{b-a}{c-a} \overset{!}{=} \frac{x-b}{c-b} \tag{V.65}$$

$$\overset{\text{(V.61)}}{\Longrightarrow} \quad \omega = \frac{x-b}{c-b}\left[\frac{c-a}{c-a}\right] \overset{\text{(V.61,V.62)}}{=} \frac{z}{1-\omega} \overset{\text{(V.64)}}{\Longrightarrow} \omega^2 - 3\omega + 1 = 0 \;\Rightarrow\; w_\pm = \frac{3 \pm \sqrt{5}}{2} \,. \tag{V.66}$$

We have to choose the smaller root because $\omega < 1$. Thus,

$$\omega = \frac{3-\sqrt{5}}{2} \approx 0.38197 \quad \text{and} \quad 1-\omega \approx 0.61803 \,. \tag{V.67}$$

In other words, the optimal bracketing interval (a, b, c) has its middle point b a fractional distance 0.38197 from one end and 0.61803 from the other. This decompostion of the interval is (almost) identical to the so-called *golden mean* or *golden section* (whence the name of the minimization method) which is defined in the following way.

Definition: golden section. The decomposition

$$a = x + (a-x) \quad (a, x \in \mathbb{R})$$

is called 'golden section' if x is the geometric mean of a and $a-x$:

$$x = \sqrt{a(a-x)} \;\Rightarrow\; x = \frac{1}{2}(\sqrt{5}-1)a \approx 0.618\, a \,. \tag{V.68}$$

V.4.3 Minimization in Higher Dimension: Conjugate Gradient Method

The previous section explained how to minimize a function of one variable. Can we exploit this knowledge to minimize also a function of many variables? The answer to this question is 'Yes', and the basic idea of the algorithm runs as follows:

1. First, we choose a point $\vec{P}$ and a direction $\vec{h}$, and we pose

$$\vec{x} = \vec{P} + \lambda\vec{h} \quad (\lambda \in \mathbb{R} \text{ and } \vec{x}, \vec{P} \in \mathbb{R}^N) \,. \tag{V.69}$$

2. Then, we find the minimum of f along the direction $\vec{h}$. This corresponds to a minimization in one dimension:

$$\frac{\mathrm{d}f}{\mathrm{d}\lambda} \overset{!}{=} 0 \;\Rightarrow\; \lambda_{\min} \,. \tag{V.70}$$

3. Next, we calculate the point corresponding to this minimum, $\vec{x}_{\min} = \vec{P} + \lambda_{\min}\vec{h}$, choose a new direction $\vec{h}'$, and repeat step 2. starting from $\vec{x}_{\min}$

$$\vec{x}' = \vec{x}_{\min} + \lambda'\vec{h}' \quad (\lambda' \in \mathbb{R}) \,. \tag{V.71}$$

Thus, a sequence of such line minimization will eventually lead us to the minimum of the function $f(\vec{x})$.

Obviously, this algorithm requires at each step that we choose a new direction $\vec{h}$. This choice has to be done with care because of the risk that subsequent minimizations can 'spoil' or even annihilate previous steps. Therefore, the key question arises of how the directions have to be chosen. The answer to this question will be developed in the following. It will introduce the concept of *non-interfering* directions, more conventionally called *conjugate directions*. These direcions have the important property that they do not spoil previous minimizations. We can make this idea explicit by the following line of reasoning.

First, we make a Taylor expansion of $f(\vec{x})$ around the origin

$$\begin{aligned} f(\vec{x}) &= f(\vec{0}) + \sum_{i=1}^{N} \frac{\partial f}{\partial x_i}\bigg|_{\vec{x}=\vec{0}} x_i + \frac{1}{2} \sum_{i,j=1}^{N} \frac{\partial f}{\partial x_i \partial x_j}\bigg|_{\vec{x}=\vec{0}} x_i x_j + \cdots \\ &= c - \vec{b}\cdot\vec{x} + \frac{1}{2}\,\vec{x}\cdot \mathsf{A}\cdot\vec{x} + \cdots \ , \end{aligned} \tag{V.72}$$

where we introduced the abbreviations

$$\left.\begin{aligned} c = f(\vec{0})\,,\ \vec{b} = -\nabla f(\vec{0})\ , \\ (\mathsf{A})_{ij} = \frac{\partial f}{\partial x_i \partial x_j}\bigg|_{\vec{x}=\vec{0}} \quad \text{(symmetric matrix)}\ . \end{aligned}\right\} \tag{V.73}$$

The matrix A consists of the second partial derivatives of f and is called the 'Hessian matrix' of f at $\vec{0}$. From the quadratic approximation of Eq. (V.72) the gradient of $f(\vec{x})$ is readily calculated

$$\nabla f(\vec{x}) = \mathsf{A}\cdot\vec{x} - \vec{b}\,. \tag{V.74}$$

The gradient obviously represents an important direction at any point $\vec{x}$. Let us therefore determine the gradient at the point $\vec{P}_{i+1}$ obtained from a minimization along $\vec{h}_i$ with starting point $\vec{P}_i$. According to Eq. (V.71) $\vec{P}_{i+1}$ reads

$$\vec{P}_{i+1} = \vec{P}_i + \lambda_i \vec{h}_i \quad (\lambda_i \mathrel{\widehat{=}} \lambda_{\min} \text{ in Eq. (V.70)})\ , \tag{V.75}$$

and its gradient is given by

$$\begin{aligned} \vec{g}_{i+1} \quad &:= \quad -\nabla f(\vec{P}_{i+1}) \\ &\overset{(\text{V.74,V.75})}{=} \quad -\mathsf{A}\cdot(\vec{P}_i + \lambda_i\vec{h}_i)\ +\ \vec{b} \\ &= \quad \underbrace{-\mathsf{A}\cdot\vec{P}_i + \vec{b}}_{\overset{(\text{V.74})}{=}\vec{g}_i} - \lambda_i \mathsf{A}\cdot\vec{h}_i\ . \end{aligned}$$

So we finally have

$$\vec{g}_{i+1} = \vec{g}_i - \lambda_i \mathsf{A}\cdot\vec{h}_i\ . \tag{V.76}$$

Since the derivative along $\vec{h}_i$ vanishes at the minimum, we must impose

$$\nabla f(\vec{P}_i)\cdot\vec{h}_i \overset{!}{=} 0 \quad\Rightarrow\quad \vec{g}_{i+1}\cdot\vec{h}_i = 0 \Leftrightarrow \vec{g}_{i+1} \perp \vec{h}_i\ , \tag{V.77}$$

and hence we can write for λ_i:

$$\vec{h}_i\cdot\vec{g}_{i+1} \overset{(\text{V.76})}{=} \vec{h}_i\cdot\vec{g}_i - \lambda_i\vec{h}_i\cdot\mathsf{A}\cdot\vec{h}_i \overset{(\text{V.77})}{=} 0 \quad\Rightarrow\quad \lambda_i = \frac{\vec{h}_i\cdot\vec{g}_i}{\vec{h}_i\cdot\mathsf{A}\cdot\vec{h}_i}\ . \tag{V.78}$$

This is an 'intermediate result' in the sense that it contains the numerical value of λ_i, which we have already determined via the minimization in one dimension. However, the expression of Eq. (V.78) is very valuable because it enables us to calculate a new direction, $\vec{h}_{i+1}$, in which we continue our minimization procedure. Since the new and the old direction are orthogonal, a minimization along $\vec{h}_{i+1}$ conserves the minimum with respect to the old direction, $\vec{h}_i$–the directions are *conjugate*!

Thus, we proceed as follows. The transformation $\vec{h}_i \to \vec{h}_{i+1}$ leads to a variation of the gradient:

$$\delta\vec{g}_{i+1} \equiv \vec{g}_{i+2} - \vec{g}_{i+1} \overset{(\text{V.76})}{=} -\lambda_{i+1}\mathsf{A}\cdot\vec{h}_{i+1}\ . \tag{V.79}$$

In order to conserve the minimum along $\vec{h}_i$, we choose the new direction $\vec{h}_{i+1}$ according to:

$$\vec{h}_i\cdot\delta\vec{g}_{i+1} \overset{!}{=} 0 \overset{(\text{V.79})}{=} -\lambda_{i+1}[\vec{h}_i\cdot\mathsf{A}\cdot\vec{h}_{i+1}]\ . \tag{V.80}$$

This result makes the concept of 'non-interfering directions' mathematically explicit. If Eq. (V.80) is fulfilled, the directions $\vec{h}_i$ and $\vec{h}_{i+1}$ are said to be 'conjugated'. By construction, the method discussed above provides successive directions which are conjugated. This allows us to find the minimum of a function in an iterative way.

In order to proceed, we make the following hypothesis:

$$\vec{h}_{i+1} \overset{!}{=} \vec{g}_{i+1} + \gamma_i \vec{h}_i \ , \tag{V.81}$$

which introduces the coefficient γ_i that we have to determine next. To achieve this, we insert Eq. (V.81) into Eq. (V.80)

$$\vec{h}_i \cdot \mathsf{A} \cdot \vec{h}_{i+1} = \vec{h}_i \cdot \mathsf{A} \cdot \vec{g}_{i+1} + \gamma_i \vec{h}_i \cdot \mathsf{A} \cdot \vec{h}_i = 0 \ \Rightarrow \ \gamma_i = -\frac{\vec{h}_i \cdot \mathsf{A} \cdot \vec{g}_{i+1}}{\vec{h}_i \cdot \mathsf{A} \cdot \vec{h}_i} \ . \tag{V.82}$$

Equation (V.82) is not yet convenient, as it depends on the matrix A, and we do not know the matrix A. However, it is possible to eliminate the dependence of Eq. (V.82) on A if we use the following properties:

- Consider $\vec{g}_k \cdot \vec{g}_{i+1}$. Since the scalar product is commutative, we have

$$\vec{g}_k \cdot \vec{g}_{i+1} \overset{(V.76)}{=} \vec{g}_k \cdot \vec{g}_i - \lambda_i \vec{g}_k \cdot \mathsf{A} \cdot \vec{h}_i = \vec{g}_{i+1} \cdot \vec{g}_k \quad \Rightarrow \quad \vec{g}_k \cdot \mathsf{A} \cdot \vec{h}_i = \vec{h}_i \cdot \mathsf{A} \vec{g}_k \quad \forall i,k \ . \tag{V.83}$$

- Multiply Eq. (V.81) with $\vec{g}_{i+1}$ to find another expression for λ_i:

$$\vec{h}_{i+1} \cdot \vec{g}_{i+1} = \vec{g}_{i+1} \cdot \vec{g}_{i+1} - \underbrace{\gamma_i \vec{h}_i \cdot \vec{g}_{i+1}}_{\overset{(V.77)}{=} 0} \quad \overset{(V.78)}{\Longrightarrow} \quad \lambda_i = \frac{\vec{h}_i \cdot \vec{g}_i}{\vec{h}_i \cdot \mathsf{A} \cdot \vec{h}_i} = \frac{\vec{g}_i \cdot \vec{g}_i}{\vec{h}_i \cdot \mathsf{A} \cdot \vec{h}_i} \ . \tag{V.84}$$

These two results can be used to rewrite the numerator of Eq. (V.82) in the following way:

$$\begin{aligned} \vec{h}_i \cdot \mathsf{A} \cdot \vec{g}_{i+1} \quad &\overset{(V.83)}{=} \quad \vec{g}_{i+1} \cdot (\mathsf{A} \cdot \vec{h}_i) \overset{(V.76)}{=} \vec{g}_{i+1} \cdot \Big[-\frac{1}{\lambda_i}(\vec{g}_{i+1} - \vec{g}_i) \Big] \\ &= \quad -\frac{1}{\lambda_i} \vec{g}_{i+1} \cdot \vec{g}_{i+1} - \frac{1}{\lambda_i} \underbrace{\vec{g}_{i+1} \cdot \vec{g}_i}_{\overset{!}{=} 0, \text{ orthogonal}} \qquad \qquad (V.85) \\ &\overset{(V.84)}{=} \quad -\frac{\vec{g}_{i+1} \cdot \vec{g}_{i+1}}{\vec{g}_i \cdot \vec{g}_i} (\vec{h}_i \cdot \mathsf{A} \cdot \vec{h}_i) \\ \overset{(V.82)}{\Longrightarrow} \quad \gamma_i \quad &= \quad \frac{\vec{g}_{i+1} \cdot \vec{g}_{i+1}}{\vec{g}_i \cdot \vec{g}_i} \ , \qquad \qquad (V.86) \end{aligned}$$

which is the desired result. This concludes our presentation of the conjugate gradient method. The method can be summarized as follows.

1. Choose a starting point $\vec{P}_0$ and calculate:

$$\vec{g}_0 = -\nabla f(\vec{P}_0) \ . \tag{V.87}$$

2. Choose a starting direction:

$$\vec{h}_0 = \vec{g}_0 \ . \tag{V.88}$$

3. Minimize along $\vec{h}_i$ with a one-dimensional method:

$$\Rightarrow \text{position of the minimum at } \vec{P}_{i+1} \ [\rightarrow \text{Eq. (V.75)}] \ .$$

4. Calculate the gradient

$$\vec{g}_{i+1} = -\nabla f(\vec{P}_{i+1}) \ .$$

5. Choose a new direction according to Eqs. (V.81,V.86):

$$\vec{h}_{i+1} = \vec{g}_{i+1} + \Big[\frac{\vec{g}_{i+1} \cdot \vec{g}_{i+1}}{\vec{g}_i \cdot \vec{g}_i} \Big] \vec{h}_i \ .$$

6. Repeat steps 3. to 5. until the method converges.

V.4.4 A Method to Find the Absolute Minimum: Simulated Annealing

Let us recapitulate the key problem mentioned in the Introducion (Sec. V.4.1). A minimum may be local or global. Finding the global (or absolute) minimum is a difficult task because the algorithm must explore the 'landscape' defined by $f(\vec{x})$ and be able to spot the global minimum among all other minima.

There are two different ways in which we can try to achieve this task:

- Minimization procedures, like the Conjugate Gradient Method, are 'downhill' methods. They determine the local minimum nearest to the starting point of the minimization and remain stuck there. In order to search for the absolute minimum the algorithm must be started at different points. These repeated minimizations (hopefully) lead to different minima which can be compared to pick out the deepest one.

- The second possibility involves algorithms that do not only go 'downhill', but also 'uphill', thereby allowing to cross barriers (local maxima) in the landscape. These methods are called *simulated annealing*. The term 'annealing' means 'very careful cooling'. Thus, simulated annealing determines the absolute minimum by cooling the system ($\widehat{=} \vec{x}$) *quasistatically* ('annealing') to its ground state ($\widehat{=}$ minimum of $f(\vec{x})$).

A caveat of the first approach should be evident immediately. One has to choose a sufficient number of starting points. This might be very time consuming, and it is hard to ensure that the number was sufficient so that the absolute minimum is really among the minima found. This may be particularly problematic if $f(\vec{x})$ exhibits several local minima which are in the vicinity of the global minimum. Then, the method can be extremely inefficient, as the distance between the points has to be rather small, and a global search for minima becomes hence inconvenient.

Simulated annealing is an ellegant way to avoid such problems. The idea is as follows: Let us interpret $\vec{x}$ as a configuration of a thermodynamic system, such that the function $f(\vec{x})$ can be identified with the energy. The global minimum then represents the ground state of the system, which will be approached as the temperature is gradually decreased. If the system is in a state that corresponds to a local energy minimum, it can leave this minimum due to thermal fluctuations (with amplitude of order $k_{\mathrm{B}}T$) and gradually walk down the landscape until the absolute minimum is found.

Now, the question arises of how we can introduce temperature into the problem. We will illustrate this by an example: the 'traveling salesman problem (TSP)'.

Traveling Salesman Problem

Definition: *Traveling Salesman Problem (TSP).* A traveling salesman wants to visit N cities. What is the *shortest* way to visit all cities exactly *once* and to finally return to the starting point?

If $\vec{r}_i$ denotes the position of city i (with $i = 1, \cdots, N$), a configuration $\vec{x}$ contains a series of $\vec{r}_i$, where each position vector appears exactly one time, e.g., $\vec{x} = \{\vec{r}_1, \vec{r}_2, \cdots, \vec{r}_N\}$. Different configurations thus follow from each others by permutation of the r_i. If we choose one city as a (fixed) starting point, we can get

$$\frac{1}{2} \cdot 1 \cdot (N-1)(N-2)\cdots 1 = \frac{1}{2}(N-1)! \tag{V.89}$$

different configurations. The factor $1/2$ in Eq. (V.89) accounts for the possibility to invert the journey. The factor 1 means that one out of N cities has been chosen as the starting point. If the first city is chosen, there are $N-1$ cities left to be the second city, then $N-2$ cities, and so on. For large N, Eq. (V.89) can be simplified by using the 'Stirling formula':

$$N! \approx \sqrt{2\pi N}\left(\frac{N}{\mathrm{e}}\right)^N, \quad N \gg 1\,, \tag{V.90}$$

such that we get

$$M \approx \exp\left[N \ln\left(\frac{N}{\mathrm{e}}\right)\right] \tag{V.91}$$

possible permutations of $\vec{x}$. One of these configurations represents the 'ground state', i.e, the circuit of shortest length. We therefore have to calculate

$$U_{\min} = \min_{\vec{x}}\Big[\sum_{i=1}^{N}\sqrt{(\vec{r}_{i+1}-\vec{r}_i)^2}\Big] . \tag{V.92}$$

Remark: The TSP belongs to the class of 'NP-complete' problems (NP=non-deterministic polynomial), which is a class of problems in which the time t_{sol} to find the solution grows as $\exp(\mathrm{const}\times N)$. In contrast to this, with simulated annealing using the Metropolis method [cf. Sec. IV.3.2], t_{sol} scales as a power law of N (with a small exponent).

Let us now present the different steps of the latter method—simulated annealing via the Metropolis method:

1. Distribute N cities ($=$ points) randomly on a square (surface area $L^2 = (Na)^2 = N^2$, $a = 1$ being the lattice constant). Let
$$\vec{x}_0 = (\vec{r}_1^{\,(0)}, \vec{r}_2^{\,(0)}, \cdots, \vec{r}_N^{\,(0)}) \tag{V.93}$$
be the starting configuration.
2. Calculate the perimeter of the circuit:
$$\mathcal{L} = \mathcal{L}(\vec{x})\sum_{i=1}^{N}\sqrt{(\vec{r}_{i+1}-\vec{r}_i)^2} \qquad (\vec{r}_{N+1} = \vec{r}_1) . \tag{V.94}$$
3. Produce a new configuration by the following steps:
   (a) Choose k ($\leq N/2$) successive points, $\vec{r}_j, \vec{r}_{j+1}, \cdots, \vec{r}_{j+k}$ (the starting point $\vec{r}_j$ is randomly chosen);
   (b) Inverse the order and reinsert into the circuit.

   That is,
$$\begin{aligned} \vec{x} &= \{\vec{r}_1, \cdots, \underbrace{\vec{r}_j, \cdots, \vec{r}_{j+k}, \vec{r}_{j+k+1}}, \cdots, \vec{r}_N\} \\ &\qquad\qquad\qquad \downarrow \\ \vec{x}\,' &= \{\vec{r}_1, \cdots, \overbrace{\vec{r}_{j+k}, \cdots, \vec{r}_j, \vec{r}_{j+k+1}}, \cdots, \vec{r}_N\} \\ &= \{\vec{r}\,'_1, \cdots, \vec{r}\,'_j, \cdots, \vec{r}\,'_{j+k}, \vec{r}\,'_{j+k+1}, \cdots, \vec{r}\,'_N\} . \end{aligned} \tag{V.95}$$
4. Calculate the 'energy' of the new configuration,
$$\mathcal{L}' = \mathcal{L}(\vec{x}\,') = \sum_{i=1}^{N}\sqrt{(\vec{r}\,'_{i+1}-\vec{r}\,'_i)^2} .$$
5. Calculate
$$\beta\Delta U := \beta(\mathcal{L}' - \mathcal{L}) , \tag{V.96}$$
where we introduced the 'inverse temperature' $\beta = 1/T$ which we defined by
$$T = T_0(1-\Gamma t) , \tag{V.97}$$
and apply the Metropolis acceptance criterion [see Eq. (IV.32)],
$$W(\vec{x}'|\vec{x}) = \begin{cases} \exp\Big[-\beta\Big(U(\vec{x}\,') - U(\vec{x})\Big)\Big] & U(\vec{x}\,') - U(\vec{x}) > 0 , \\ 1 & U(\vec{x}\,') - U(\vec{x}) \leq 0 , \end{cases}$$

to decide whether the new configuration shall be accepted or rejected.

In Eq. (V.97), T_0 denotes the initial temperature and Γ the ‘cooling rate’. The initial temperature should be high, corresponding to a large perimeter. For instance, one can choose $T_0 \propto L \propto N$, which means that the initial configuration is a ‘rod’ whose end points are connected (by a bond of length L). The cooling rate Γ is an important control parameter: it defines the time $t = 1/\Gamma$ to cool the system from T_0 to $T = 0$. If Γ is choosen too small, thermal energy is removed too rapidly from the system so that it might get stuck in local minima. This parameter must thus be carefully optimized.

6. Repeat steps 3.–5. until the method converges.

Chapter VI

Molecular Dynamics Simulations

VI.1 Introduction

We consider a thermodynamic system of classical particles in the microcanonical (mc) ensemble. In this ensemble, the total energy E, the volume V, and the number of particles N are constant. Macroscopically, the system is characterized by three parameters only: E, V, and N. Microscopically, however, its description is more complex. At the microscopic level, the system is caracterized by a point $\vec{x}$ in phase space $\Omega(E, V, N)$, i.e.,

$$\vec{x} = (\vec{r}^N, \vec{p}^N) = (\vec{r}_1, \ldots, \vec{r}_N; \vec{p}_1, \ldots, \vec{p}_N) \; . \tag{VI.1}$$

Here $\vec{x} = \vec{x}(t)$ is the solution of the classical equations of motion of the many-body system:

$$\underset{\text{Newton}}{\frac{\mathrm{d}^2\vec{r}_i}{\mathrm{d}t^2} = \frac{1}{m}\vec{F}_i} \quad \Longleftrightarrow \quad \underset{\text{Hamilton}}{\left\{ \begin{array}{l} \dfrac{\mathrm{d}\vec{r}_i}{\mathrm{d}t} = \dfrac{1}{m}\vec{p}_i = \dfrac{\partial \mathcal{H}}{\partial \vec{p}_i} \\ \dfrac{\mathrm{d}\vec{p}_i}{\mathrm{d}t} = \vec{F}_i = -\dfrac{\partial \mathcal{H}}{\partial \vec{r}_i} \end{array} \right.} \tag{VI.2}$$

where the hamiltonian $\mathcal{H}$ is the total energy of the system

$$\mathcal{H}(\vec{x}) = \sum_{i=1}^{N} \frac{\vec{p}_i^{\,2}}{2m} + U(\vec{r}^N) \; . \tag{VI.3}$$

In Eq. (VI.3), $\vec{p}_i$ denotes the momentum of particle i and $U(\vec{r}^N) = U(\vec{r}_1, \ldots, \vec{r}_N)$ is the interaction potential between the particles. Equation (VI.2) provides a full description of the microstate of the system if the initial conditions $\{\vec{r}_i(0), \vec{p}_i(0)\}_{i=1,\ldots,N}$ are specified.

However, the drawback of this formally exact microscopic approach is known since the 19th century. An analytical solution of Eq. (VI.2) cannot be obtained in general for a thermodynamic system. The solution to this problem—and a solution to reconcile the microscopic and macroscopic approaches—was suggested by Boltzmann. This solution starts from the observation that the microscopic information—it specifies the full instantaneous configuration $\vec{x}(t)$ of the system—is too detailed. Most of this information is lost when passing from the microscopic to the macroscopic description:

$$\begin{array}{ccc} \textit{microscopic level} & & \textit{macroscopic level} \\ \text{classical mechanics} & \xrightarrow{\text{reduction of complexity}} & \text{thermodynamique} \\ \vec{x}(t) \;\widehat{=}\; 6N \text{ variables} & & E, V, N \;\widehat{=}\; 3 \text{ variables} \\ \text{observables} = \text{function of } t\text{: } A(\vec{x}(t)) & & \text{observables} \neq \text{function of } t\text{: } A(E, V, N) \end{array}$$

Apparently, not every point $\vec{x}(t)$ is important for the macroscopic description, but only the trajectories that lie on a hypersurface in phase space of constant energy E, that is the ensemble of phase space points satisfying $\{\vec{x}(t) \in \Omega(E,V,N) \,|\, \mathcal{H}(\vec{x}) = E\}$.

Boltzmann proceeded by the following line of reasoning:

- A typical measurement of thermodynamic observables lasts much longer than the time it takes the system to pass from one microstate to another. The thermodynamic observable $A(E,V,N)$ should thus not be identified with $A(\vec{x}(t)$, but with the *time average*

$$A(E,V,N) \overset{!}{=} \overline{A}(\vec{x}_0) = \lim_{t\to\infty} \frac{1}{t} \int_0^t \mathrm{d}t\, A(\vec{x}(t)) \, . \qquad \text{(VI.4)}$$

- This consideration eliminates the explicit dependence of A on time present at the microscopic level. However, *a priori* the time average depends on the initial condition $\vec{x}_0$, and the calculation of $A(E,V,N)$ still requires the knowledge of $\vec{x}(t)$, i.e., the solution of Eq. (VI.2).

  To solve these problems Boltzmann suggested that the trajectory visits all points $\vec{x}$ of Ω ('ergodic hypothesis'). Thus, all points should contribute *with the same weight* to the time average. Since there is consequently no objective criterion to distinguish phase space points, all being equally probable, the point $\vec{x}_0$ plays no particular role, and $\overline{A}(\vec{x}_0)$ must be independent of $\vec{x}_0$. Furthermore, the argument of equal probability of all points suggests to replace the time average by an *ensemble average* $\langle A\rangle_{\mathrm{mc}}$

$$\overline{A} = \langle A\rangle_{\mathrm{mc}} = \int_\Omega \mathrm{d}^{6N}\vec{x}\, A(\vec{x}) P_{\mathrm{mc}}(\vec{x}) \, ; \qquad \text{(VI.5)}$$

  with the uniform distribution

$$P_{\mathrm{mc}}(\vec{x}) = \frac{1}{\mathcal{Z}_{\mathrm{mc}}}\, \delta(\mathcal{H}(\vec{x}) - E)\, , \quad \mathcal{Z}_{\mathrm{mc}}(E,V,N) = \int_\Omega \mathrm{d}^{6N}\vec{x}\, \delta(\mathcal{H}(\vec{x}) - E) \, , \qquad \text{(VI.6)}$$

  where $\mathcal{Z}_{\mathrm{mc}}(E,V,N)$ denotes the partition function in the microcanonical ensemble.

Equation (VI.5) replaces the nontractable problem of classical dynamics by an analytically tractable problem of probability theory. Therefore, Boltzmann is the mental father of statistical mechanics.

Although it is not possible to prove the ergodic hypothesis in general, we can test its validity—and it has been verified numerically. To this end, both sides of Eq. (VI.5) must be determined independently. The right-hand side, the ensemble average, can be obtained from Monte Carlo simulations (see Sec. IV). The left-hand side can be determined by another simulation method, the *molecular dynamics* (MD). MD calculates thermodynamic observables as time averages $\overline{A}$ by numerical integration of Eq. (VI.2) for the N-body system. That is,

$$\overline{A} = \lim_{t\to\infty} \frac{1}{t} \int_0^t \mathrm{d}t'\, A(\vec{x}(t')) \overset{t\gg 1}{\approx} \frac{1}{t} \int_0^t \mathrm{d}t'\, A(\vec{x}(t')) \underset{\substack{\uparrow \\ \text{discretization: } 0, t_1 = h, \ldots, t_m = mh, \ldots, t = Mh}}{\approx} \frac{1}{(Mh)} \sum_{m=0}^{M} h\, A(\vec{x}_m) \qquad (\vec{x}_m = \vec{x}(t_m)) \, . \qquad \text{(VI.7)}$$

Equation (VI.7) shows that MD requires a discretization of the classical trajectory, introducing the time interval h as a 'small' parameter, the so-called 'integration step'. In the following, we shall give an introduction to this modern simulation method.

VI.2 Molecular Dynamics Simulations in the Microcanonical Ensemble

VI.2.1 Basic Structure of the Algorithm

The following flowchart shows the basic structure of the MD algorithm:

```
PROGRAM md                            MD program

call init                             initialization = choose initial positions
                                      and momenta for all particles
t = 0
do m = 1,M
   call force(F,E)                    calculate the forces F and the total energy E
   call integrate(F,E)                integrate equations of motion
   t = t + h                          increment the time by the time step h
   call analysis                      calculate observables
enddo
END
```

The details of the different steps in this flowchart will be elaborated in the following sections.

VI.2.2 Initialization

To start the MD simulation we need the initial conditions, i.e., $\{\vec{r}_i(0), \vec{p}_i(0)\}_{i=1,\ldots,N}$ (`SUBROUTINE init`). Here, we can first make a 'convenient' choice and then allow the system to relax. Convenient choices for the positions and momenta of the N particles are:

- $\vec{r}_i(0)$: We place the particles on a crystalline lattice. This is simple and efficient because the position of the first particle fixes those of all other particles, and strongly repulsive particle overlaps leading to very high forces are avoided.
- $\vec{p}_i(0) = m\vec{v}_i(0)$: We choose the initial velocities of the particles according to the Maxwell-Boltzmann distribution
$$P(v_{\alpha i}) = \sqrt{\frac{m}{2\pi k_\mathrm{B}T}} \exp\left(-\frac{mv_{\alpha i}^2}{2k_\mathrm{B}T}\right), \quad \alpha = x, y, z;\; i = 1, \ldots, N. \tag{VI.8}$$

Remarks:

- Equation (VI.8) impose the temperature T.
- During equilibration it will be necessary to reinitialize the velocities in order to keep T constant.
- In an isolated system, i.e., a system that does not experience an external force, the total momentum is conserved:
$$\vec{p}_\mathrm{tot} = \sum_{i=1}^{N} m\vec{v}_i = \mathrm{constant} = 0\,. \tag{VI.9}$$
We have to correct the particle velocities so that they satisfy this condition.

The algorithm corresponding to this initialization is sketched in the following flowchart:

```
SUBROUTINE init                       initialization of the MD program
```

```
sumv   = 0                        set sum variable for the velocity and for the
sumv2  = 0                        square velocity to 0
sigma2 = T                        variance = k_B T (Boltzmann constant k_B = 1)

do i = 1,N
   x(i)  = lattice(i)             place the particles on a lattice
   v(i)  = gauss(sigma2)          draw particle velocities from a Gaussian distribution
   sumv  = sumv + v(i)            determine velocity of the center of mass
   sumv2 = sumv2 + v(i)**2        determine square velocity
enddo

sumv  = sumv/N                    velocity of the center of mass
sumv2 = sumv2/N                   mean square velocity
RETURN
END
```

VI.2.3 Force Calculation

After the initialization we have to calculate the forces acting on each particle (`SUBROUTINE force`). For this calculation we make the assumption that the N-body potential is pair-decomposable

$$\begin{aligned} r_{ij} &= |\vec{r}_{ij}| = |\vec{r}_i - \vec{r}_j| \ , \\ U(\vec{r}^{\,N}) &= \frac{1}{2}\sum_{i\neq j} U(r_{ij}) \ . \end{aligned} \tag{VI.10}$$

This implies e.g. for the total force on particle i in x direction

$$\begin{aligned} F_{i,x} &= \sum_{j=i+1}^{N}\left[-\frac{\partial U(r_{ij})}{\partial x_{ij}}\right] \qquad (i = 1,\ldots,N-1) \\ &= \sum_{j=i+1}^{N}\left[-\frac{x_{ij}}{r_{ij}}\right]\frac{\partial U(r_{ij})}{\partial r_{ij}} \ . \end{aligned} \tag{VI.11}$$

Let us consider a specific example. We assume that the pair-potential $U(r_{ij})$ is given by the Lennard-Jones potential $U_{\mathrm{LJ}}(r_{ij})$, Eq. (IV.34). Then, the force $F_{i,x}$ in reduced units is given by

$$F_{i,x} = 48\left[\frac{\epsilon}{\sigma}\right]\sum_{j=i+1}^{N}\frac{x_{ij}^*}{(r_{ij}^*)^2}\left[\frac{1}{(r_{ij}^*)^{12}} - \frac{1}{2}\frac{1}{(r_{ij}^*)^6}\right] \ . \tag{VI.12}$$

Here the term "reduced units" means that we introduce

$$\begin{aligned} \text{the length scale:} &\quad \sigma \ , \\ \text{the energy scale:} &\quad \epsilon \ , \\ \text{the mass scale:} &\quad m \end{aligned} \tag{VI.13}$$

as units for physical quantities:

$$\left.\begin{array}{rcrcll} r & \to & r^* & = & r/\sigma \ , & \\ U_{\mathrm{LJ}} & \to & U_{\mathrm{LJ}}^* & = & U_{\mathrm{LJ}}/\epsilon \ , & \\ T & \to & T^* & = & k_{\mathrm{B}}T/\epsilon \ , & \\ t & \to & t^* & = & t/(\sigma\sqrt{m/\epsilon}) & (\text{time}) \ , \\ \rho & \to & \rho^* & = & \rho\sigma^3 & (\text{density}) \ , \\ p & \to & p^* & = & p\sigma^3/\epsilon & (\text{pressure}) \ . \end{array}\right\} \tag{VI.14}$$

The reason for introducing reduced units is that the constants (σ, ϵ, m) can take vastly different values. For instance, $\rho^* = 0.5$ and $T^* = 0.5$ correspond to

$$\begin{array}{rclrcll} \rho &=& 840\,\mathrm{kg/m^3}, & T &=& 60\,\mathrm{K} & \text{for Ar ,} \\ \rho &=& 1617\,\mathrm{kg/m^3}, & T &=& 112\,\mathrm{K} & \text{for Xe .} \end{array}$$

In reduced units, one simulation thus suffices to simulate various systems in different states.

Remarks concerning the potential:

- Equation (VI.11) is of order N^2. This implies that the force calculation is the most time-consuming step of the MD simulation.
- This fact has two consequences:
  - The total number of particles that can be treated is small compared to a macroscopic system:
    $$10^3 \lesssim N < 10^6 \ll 10^{23} \ . \tag{VI.15}$$
    This may raise questions about the finite size of the system ("finite-size effects").
  - Usually one simplifies the potential by truncating its range [cf. Eq. (IV.35)] because this allows a faster execution by means of "cell boxing" or "neighbor lists".

We will return to this optimization of the force calculation by cell boxing or neighbor lists later. First, we want to address the issue of how the boundary conditions of a simulation are typically chosen.

Periodic Boundary Conditions

To motive the choice of the boundary conditions we consider a system with N particles and density ρ. Then, the linear dimension L of the simulation box scales with N as

$$\rho = \frac{N}{L^3} \quad \Rightarrow \quad L \propto N^{1/3} \tag{VI.16}$$

so that

$$\text{the fraction of particles close to surfaces} \sim \frac{L^2}{L^3} \propto \frac{1}{N^{1/3}} \ . \tag{VI.17}$$

This implies, for instance, that in a system of 10^3 particles roughly 10% of the particles feel the surfaces, and still 1% for $N = 10^6$. Consequently, surface effects are not negligible.

Being interested in bulk properties of a thermodynamic system such (artificial) surface effects have to be minimized. A way to achieve this consists in employing periodic boundary conditions (pbc). This means that the simulation box is replicated in all spatial dimensions; it thus represents a unit cell of a simple cubic lattice

$$\vec{r}_i \to \vec{r}_i + \vec{n}L \,, \quad \vec{n} = (n_x, n_y, n_z), \ n_{x,y,z} \in \mathbb{Z} \ . \tag{VI.18}$$

These boundary conditions are schematically represented in Fig VI.1. Comments to the figure:

- If a particle moves in the simulation box, all of its periodic images also move in exactly the same way. In particular, if a particle leaves the simulation box on the right side, its left copy enters the box. This preserves the density of the system.
- The original simulation cell only serves to define the origin of the coordinate system, but otherwise has no particular significance for the calculation of the distance $d(\vec{r}_i, \vec{r}_j)$ between two particles i and j. One calculates this difference according to the *minimum-image convention*
  $$d(\vec{r}_i, \vec{r}_j) := \min_{\vec{n} \in \mathbb{Z}^3} \left| \vec{r}_i - \vec{r}_j + \vec{n}L \right| \ . \tag{VI.19}$$

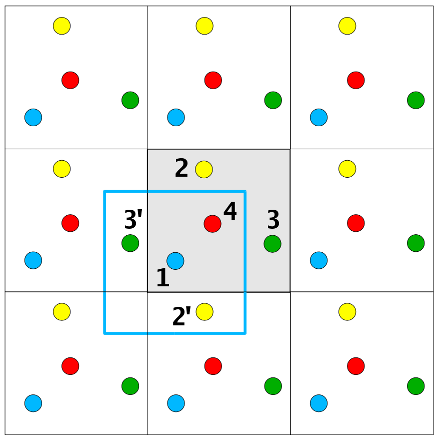


Figure VI.1: Periodic boundary conditions (pbc). The simulation cell (grey) is periodically replicated in all spatial directions. For instance, the particles 2 and 3 have image particles $2'$ and $3'$. These image particles have to be taken into account—instead of 2 and 3—when calculating the interaction of particle 1 with the other particles in the system because the original simulation cell loses its significance by the pbc. The cell that is important for the calculation of the interaction of particle 1 is thus the blue cell whose center is the position of particle 1.

For instance,

$$\text{instead of } \left\{ \begin{array}{c} 1 \leftrightarrow 2 \\ 1 \leftrightarrow 3 \end{array} \right\} \text{ the following particle pairs interact: } \left\{ \begin{array}{c} 1 \leftrightarrow 2' \\ 1 \leftrightarrow 3' \end{array} \right\}.$$

The pseudo-code for the implementation of the minimum-image convention looks as follows:

```
box    = L
boxinv = 2/L

do i = 1,N-1
   do j = i+1,N
   dx = x(i) - x(j)                  determine the distance between particles
   dy = y(i) - y(j)                  i and j
   dz = z(i) - z(j)

   dx = dx - box*int(dx*boxinv)      apply the minimum-image convention to dx
   dy = dy - box*int(dy*boxinv)      apply the minimum-image convention to dy
   dz = dz - box*int(dz*boxinv)      apply the minimum-image convention to dz

   rsqr = dx**2 + dy**2 + dz**2      calculate the distance between particles
                                     i and j with the minimum-image convention
enddo
```

Discussion of this algorithm:

$$\begin{aligned} \text{if } |\mathrm{d}x| < L/2: &\quad \Rightarrow\ |\mathrm{d}x \cdot \mathrm{boxinv}| < 1 \ \Rightarrow\ \mathrm{int}(\mathrm{d}x \cdot \mathrm{boxinv}) = 0 \ \Rightarrow\ \mathrm{d}x \text{ unchanged}\,, \\ \text{if } L/2 < \mathrm{d}x < L: &\quad \Rightarrow\ (\mathrm{d}x \cdot \mathrm{boxinv}) > 1 \ \Rightarrow\ \mathrm{int}(\mathrm{d}x \cdot \mathrm{boxinv}) = 1 \ \Rightarrow\ \mathrm{d}x \to \mathrm{d}x - L\,, \\ \text{if } -L < \mathrm{d}x < -L/2: &\quad \Rightarrow\ \mathrm{int}(\mathrm{d}x \cdot \mathrm{boxinv}) = -1 \ \Rightarrow\ \mathrm{d}x \to \mathrm{d}x + L\,. \end{aligned}$$

If $|\mathrm{d}x|$ is smaller than $L/2$ (first line), particle j is closer to i than the nearest image of j. So the initial distance remains unchanged. On the other hand, if $|\mathrm{d}x|$ is larger than $L/2$ (second and third lines), there is a periodic image of j that is closer to i.

- Periodic boundary conditions truncate fluctuations of order L, for instance
  - at a second order phase transtition. Close to a second order phase transition, strong correlations develop between the degrees of freedom which order—e.g. between the magnetic moments at a paramagnetic/ferromagnetic phase transition. These correlations make the ordering degrees of freedom to cluster. The size of these clusters is much larger than typical microscopic distances and diverges at the phase transition provided the system is infinite. For a finite system size the divergence is truncated, and the phase transition will be rounded. Computational physics has developed a method—the method of *finite-size scaling*—to extract the behavior of the infinite (thermodynamic) system from such finite-size simulation results [for a pedagogical introduction see K. Binder and D. W. Heermann, *Monte Carlo Simulation in Statistical Physics* (Springer, Berlin, 1997)].
  - for long range interactions

$$U(r) \sim \frac{1}{r^{\alpha}}\,, \quad \alpha \leq d \text{ (= spatial dimension).} \tag{VI.20}$$

    This implies that a particle will interact with all of its periodic images. Special methods have been invented to cope with this problem, e.g., "Ewald summation" [see D. Frenkel and B. Smit, *Understanding Molecular Simulation* (Academic Press, San Diego, 1996)].

Optimization of the Force Calculation

If we apply periodic boundary conditions (pbc), the potential energy of a pair-decomposable system is written as

$$U(\vec{r}^N) = \frac{1}{2}{\sum_{ij,\vec{n}}}' U(|\vec{r}_{ij} + \vec{n}|)\ . \tag{VI.21}$$

This implies that particle i interacts with all other particles j in this infinite periodic system; that is, with all other particles in the simulation box—the fact that $i \neq j$, if $\vec{n} = 0$, is indicated by a prime $\sum'$—and with all particles including its own image in all other cells (for $\vec{n} \neq 0$).

In this most general form, pbc are not very useful. The interaction energy involves an infinite sum over all periodic images. Since this is very unfavorable, one usually truncates the range of the interaction. For instance, for the Lennard-Jones potential one writes [cf. Eq. (IV.35)]

$$U_{\rm LJ}(r) \ \rightarrow\ U_{\rm LJ}^{\rm ts} = \begin{cases} U_{\rm LJ}(r) - U_{\rm LJ}(r_{\rm c}) & r \leq r_{\rm c}\ , \\ 0 & \text{otherwise}\ . \end{cases} \tag{VI.22}$$

This truncation has two consequences:

- It introduces a discontinuity of the force at $r = r_{\rm c}$, e.g. for $r_{\rm c} = 2.5\,\sigma$,

$$F_{\rm LJ}(r_{\rm c}) = -\left.\frac{{\rm d}U_{\rm LJ}(r)}{{\rm d}r}\right|_{r=r_{\rm c}} = \begin{cases} -0.039 & r \leq r_{\rm c}\ , \\ 0 & r > r_{\rm c}\ . \end{cases} \tag{VI.23}$$

  This jump can entail numerical instabilities.

- In order to compare with experiments or theory, which do not truncate the potential, a "tail correction" has to be carried out [see also the discussion below Eq. (IV.35)]. For example, for the potential energy this tail correction (underlined term) is given by

$$\begin{aligned} U &= \frac{1}{2}\sum_{i\neq j} U_{\rm LJ}(r_{ij}) \\ &= \frac{1}{2}\sum_{\langle ij\rangle}\Big[U_{\rm LJ}^{\rm ts}(r_{ij}) + U_{\rm LJ}(r_{\rm c})\Big] + \underline{\frac{N\rho}{2}\int_{r_{\rm c}}^{\infty} {\rm d}r\, 4\pi r^2\, g(r)\, U_{\rm LJ}(r)}\ , \end{aligned} \tag{VI.24}$$

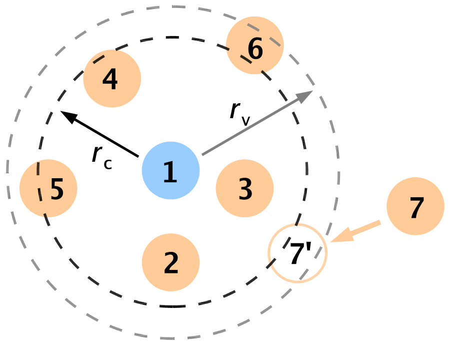


Figure VI.2: Verlet list for particle 1. Not only the particles within the cut-off radius $r_{\rm c}$ of the potential appear in this list, but also particles within the 'skin layer' of thickness $r_{\rm v} - r_{\rm c}$ (particle 6 in the present example). These particles can move into the interaction range between two updates of the list. On the other hand, particle 7 is outside this region and does not have to be taken into account. If the displacement of a particle becomes larger than $r_{\rm v} - r_{\rm c}$ (such as, $7 \to 7'$), the list has to be updated.

where $\langle ij \rangle$ denotes the sum over all neighbors within the interaction range $r_{\rm c}$, ρ the particle density, $g(r)$ the pair-distribution function, and $U_{\rm LJ}(r)$ the nontruncated LJ-potential. The pair-distribution function is a measure for the probability to find another particle j at distance r from particle i. For simple liquids, the choice $r_{\rm c} = 2.5\,\sigma$ implies that $g(r) \approx 1$, and the tail correction becomes

$$\int_{r_{\rm c}}^{\infty} {\rm d}r\, 4\pi r^2\, g(r)\, U(r) \approx \int_{r_{\rm c}}^{\infty} {\rm d}r\, 4\pi r^2\, U(r) \tag{VI.25}$$

$$\sim \int_{r_{\rm c}}^{\infty} {\rm d}r\, r^{2-\alpha} \overset{\alpha \leq 3}{=} \infty \qquad \text{for } U(r) \sim r^{-\alpha}\,. \tag{VI.26}$$

Thus, the tail correction is only a small correction if $\alpha > 3$, such as for $U_{\rm LJ}(r)$.

However, even if we use a truncated pair potential, a naive programming of the force calculation according to Eq. (VI.12)—that is, a simple sum over all $N(N-1)/2$ particle pairs—would tremendously slow down the execution of the program for large system sizes because it is an operation of order N^2. There are methods to improve this, the 'Verlet list' and 'linked cell' methods. Both methods make use of the fact that the interaction range of the potential is finite. Thus, only the neighbors within this range have to be included in the force calculation. Keeping these neighbors in lists for each particle should—and does—ameliorate the scaling of the force calculation with N. We will sketch the Verlet list and linked cell methods in the following.

Verlet lists. The basic idea consists in setting up a list of neighbors with which a particle can interact. Only these particles have to be taken into account in the calculation. The number of particles inside a sphere of radius $r_{\rm v}$ is given by:

$$n_{\rm v} = \frac{4}{3}\pi\rho r_{\rm v}^3\,. \tag{VI.27}$$

Here, we introduced the radius $r_{\rm v} > r_{\rm c}$ (typically, $r_{\rm v} = 2.7$ and $r_{\rm c} = 2.5$) because (see Fig. VI.2)

- particles with $r > r_{\rm v}$ cannot penetrate into the sphere with radius $r_{\rm c}$ in the time lapse between two updates of the Verlet list of particle 1;
- particles, such as particle 3 and 4 or 5 and 6, which can enter or leave the sphere of radius $r_{\rm c}$, have to be in the list.

If the displacement of a particle is larger than $r_{\rm v} - r_{\rm c}$, the list has to be updated.

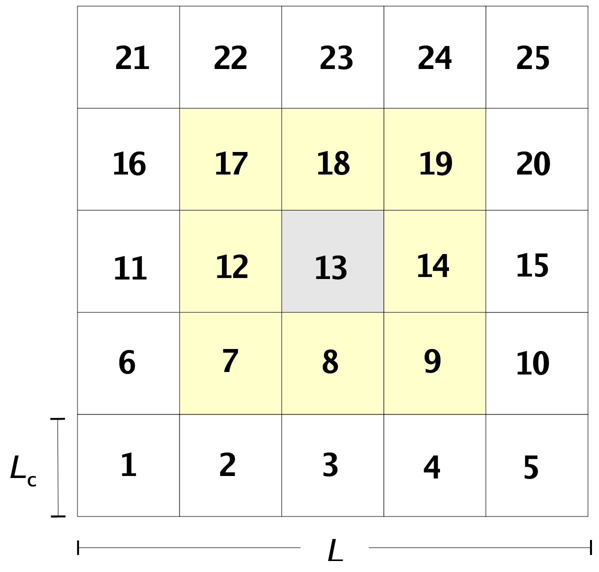


Figure VI.3: Illustration of the linked cell list in two dimensions. Cell 13 (grey) is surrounded by 8 neighbor cells (yellow); only these cells have to be taken into account when calculating the interactions of particles in cell 13 (in addition to those particles contained in cell 13). L denotes the linear dimension of the simulation box, and L_{c} is the linear dimension of a cell.

The size of the list is of order $n_{\mathrm{v}}N$. Thus, the algorithm is only efficient if $n_{\mathrm{v}} \ll N$. Example:

$$\rho = 0.8, \;\; r_{\mathrm{v}} = 2.7 \quad \Rightarrow \quad n_{\mathrm{v}} \approx 66 \quad \Rightarrow \quad N > 10^2 \,. \tag{VI.28}$$

The CPU time of the algorithm scales as

$$t_{\mathrm{V}} = c_1 n_{\mathrm{v}} N + c_2 N^2 \,, \tag{VI.29}$$

where c_1 and c_2 are constants. The second term of Eq. (VI.29) accounts for the update of the Verlet list, where all particle pairs have to be checked and redistributed.

Linked cell method. The key idea of this method consists in decomposing the simulation box (of size $L \times L \times L$) in M^3 (sub-) cells (of size $L_{\mathrm{c}} \times L_{\mathrm{c}} \times L_{\mathrm{c}}$) with

$$\frac{L}{M} = L_{\mathrm{c}} \overset{!}{=} r_{\mathrm{c}} \,. \tag{VI.30}$$

This implies that a particle can only interaction with the particles in its cell and those of its neighbor cells, as illustrated in Fig. VI.3. The figure shows that

- in total 3^d cells have to be considered, d being the spatial dimension (for instance, 9 cells in two-dimensions);
- in each cell there are approximately $N_{\mathrm{c}} = N/M^d$ particles so that the number of pair interactions is given by

$$n_\ell = 3^d N_{\mathrm{c}} = 3^d \frac{N}{M^d} = 3^d \frac{N}{L^d} \left(\frac{L}{M}\right)^d = 3^d \rho r_{\mathrm{c}}^d \,. \tag{VI.31}$$

This implies that the CPU time scales as

$$t_\ell = c_1 n_\ell N + c_3 N \,, \tag{VI.32}$$

where c_3 is another constant and the last term represents again the update of the cells.

Summary: comparison of the methods

- The CPU time of the naive algorithm scales as

$$t_{N^2} = \frac{1}{2} c_1 N(N-1) \; . \tag{VI.33}$$

  This approach is simple to program, but much slower than Eqs. (VI.29) or (VI.32) if $N > 10^2$.

- Verlet versus linked cell ($d = 3$):

$$\frac{n_\ell}{n_\mathrm{v}} = \frac{81}{4\pi}\left(\frac{r_\mathrm{c}}{n_\mathrm{v}}\right)^3 \simeq 5 \qquad (\text{for } r_\mathrm{c} = 2.5\,\sigma, r_\mathrm{v} = 2.7\,\sigma) \; . \tag{VI.34}$$

  Thus, the Verlet list is faster than the linked cell algorithm; however, the update of the Verlet lists scales as N^2, where it is only of order N for the linked cell method. So, the best algorithm should consist of a combination of both ideas: one calculates the interactions according to the Verlet lists, but builds the lists according to the linked cell prescription. Then,

$$t_{\mathrm{V}\ell} = c_1 n_\mathrm{v} N + c_4 N \; . \tag{VI.35}$$

A detailed comparison of all these approaches may be found in D. Frenkel, B. Smit, *Understanding Molecular Simulation* (Academic Press,San Diego, 1996). It turns out that both, the Verlet list and the linked cell method, are indeed more efficient than the naive approach, but the combination of the Verlet list and the linked cell methods does not necessarily lead to a faster force calculation than the linked cell alone (which is more efficient, in the discussed example, than the Verlet method).

At the end of this section we provide the flowchart for the force calculation, using the naive approach (of order N^2):

```
SUBROUTINE force(F,E)                 calculate the force and the potential energy

E = 0                                 initialize sum variable for the energy
do i = 1,N
   F(i)  = 0                          set forces for all particle to zero
enddo

do i = 1,N-1
   do j =i+1,N
      dx = x(i) - x(j)                determine the distance between particles
      dy = y(i) - y(j)                i and j
      dz = z(i) - z(j)

      dx = dx - box*int(dx*boxinv)    apply the minimum-image convention to dx
      dy = dy - box*int(dy*boxinv)    apply the minimum-image convention to dy
      dz = dz - box*int(dz*boxinv)    apply the minimum-image convention to dz

      rsqr = dx**2 + dy**2 + dz**2    calculate the distance between particles
                                      i and j with the minimum-image convention

      if(rsqr.lt.rc2) then            test if distance is smaller than cut-off
        r2i = 1/rsqr
        r6i = 1/rsqr**3
        ff  = 48*r2i*r6i*(r6i-0.5)    Lennard-Jones potential
        F(i)= F(i) + ff*sqrt(rsqr)    update force on particle i
        F(j)= F(j) - ff*sqrt(rsqr)    update force on particle j
        E   = E+4*r6i*(r6i-1)-ecut    update energy of truncated LJ potential
      endif
   enddo
enddo
RETURN
END
```

VI.2.4 Numerical Integration of the Equations of Motion

The basic idea is simple. One discretizes Newton's equations of motion for an N-body system by expanding the equations in a Taylor series ($i = 1, \ldots, N$)

$$\vec{r}_i(t_k + h) = \vec{r}_i(t_k) + \vec{v}_i(t_k)\, h + \frac{\vec{F}_i(t_k)}{2m} h^2 + \frac{1}{3!} \frac{\mathrm{d}^3 \vec{r}_i(t)}{\mathrm{d}t}\bigg|_{t=t_k} h^3 + \mathcal{O}(h^4) \, , \tag{VI.36}$$

where h denotes the integration step and $t_k = kh$ with $k = 0, 1, \ldots, M$.

This approach to the numerical integration of the equations of motion immediately leads a conflict: To avoid integration errors, we should choose h as small as possible. But a small h implies that with a maximum number of M integration steps, the total time $t = Mh$ is small. Then, phase space is not well probed, and consequently the calculated thermodynamic properties are inaccurate.

In order to alleviate this problem several approaches have been proposed. We will discuss algorithms which utilize only the first two (physical) derivatives and also an algorithm which pushes the expansion Eq. (VI.36) to higher than second order. In the following, to simplify the notation we drop the indices i and k, i.e.,

$$\vec{r}_i \;\rightarrow\; \vec{r} \quad \text{and} \quad t_k \;\rightarrow\; t \, .$$

Verlet Algorithm and Similar Methods

The simplest algorithm is the Euler algorithm:

$$\begin{aligned} \vec{r}(t+h) &= \vec{r}(t) + \vec{v}(t)h \, , \\ \vec{v}(t+h) &= \vec{v}(t) + \frac{\vec{F}(t)}{m} h \, . \end{aligned} \tag{VI.37}$$

Except its simplicity there is not much to commend the Euler algorithm: the discretization error is of order h^2 and thus large; it does not satisfy the intrinsic properties of Newtonian dynamics, that is

- Newtonian dynamics is reversible, i.e., invariant with respect to $t \rightarrow -t$,
- Newtonian dynamics conserves the phase space volume—it is 'sympletic' . (VI.38)

In summary, the Euler algorithm should be avoided because better integration methods are available, such as the Runge-Kutta method [cf. Sec. II.3], in which one expresses $\vec{r}(t+h)$ and $\vec{r}(t)$ in terms of the derivatives at $t + h/2$. That is,

$$\begin{aligned} \vec{r}(t+h) &= \vec{r}(t+h/2) + \vec{v}(t+h/2)\,\frac{h}{2} + \frac{\vec{F}(t+h/2)}{2m}\left(\frac{h}{2}\right)^2 + \mathcal{O}(h^3) \, , \\ \vec{r}(t) &= \vec{r}(t+h/2) - \vec{v}(t+h/2)\,\frac{h}{2} + \frac{\vec{F}(t+h/2)}{2m}\left(\frac{h}{2}\right)^2 - \mathcal{O}(h^3) \, . \end{aligned} \tag{VI.39}$$

Substraction yields

$$\vec{r}(t+h) = \vec{r}(t) + \vec{v}(t+h/2)h + \mathcal{O}(h^3)\ . \tag{VI.40}$$

Similarly for the velocities, we can write down equations analogous to Eq. (VI.39) from which we obtain by substraction:

$$\vec{v}(t+h/2) = \vec{v}(t-h/2) + \frac{\vec{F}(t)}{m}h + \mathcal{O}(h^3)\ . \tag{VI.41}$$

This algorithm is called *leap-frog algorithm*. The name stems from the fact that the algorithm resembles a 'leap-frog' motion: By using the velocities at $t+h/2$ Eq. (VI.39) calculates the positions at a later time $t+h$. So the positions 'jump' ahead of the velocities. Then, $\vec{r}(t+h)$ is employed to calculate the new forces which in turn determine the velocities at time $t+(3/2)h$ via Eq. (VI.41). Now, the velocities are ahead of the positions, and so on. The disadvantage of the algorithm thus is that velocities and positions are never available at the same time; the velocity at time t has to be obtained by interpolation, i.e.,

$$\vec{v}(t) = \frac{1}{2}\Big[\vec{v}(t+h/2) + \vec{v}(t-h/2)\Big]\ . \tag{VI.42}$$

Besides this drawback the leap-frog algorithm has two main advantages compared to the Euler algorithm: it reduces the discretization error (it is only of order h^3) and it can be shown that it satisfies Eq. (VI.38).

Nonetheless, it is possible to go a step further and propose an algorithm which is equivalent to the leap-frog algorithm, but removes the asymmetry between velocity and position calculations. This algorithm, the *velocity-Verlet algorithm*, is commonly employed in MD simulations. To derive the velocity-Verlet algorithm we start from Eq. (VI.36)

$$\vec{r}(t+h) = \vec{r}(t) + \vec{v}(t)\,h + \frac{\vec{F}(t)}{2m}h^2 + \mathcal{O}(h^3)\ , \tag{VI.43}$$

and we write similar Taylor expansions for the velocities $\vec{v}(t+h/2)$, i.e.

$$\vec{v}(t+h/2) = \vec{v}(t) + \frac{\vec{F}(t)}{m}\left(\frac{h}{2}\right) + \mathcal{O}(h^2)\ , \tag{VI.44}$$

$$\vec{v}(t+h/2) = \vec{v}(t+h) - \frac{\vec{F}(t+h)}{m}\left(\frac{h}{2}\right) + \mathcal{O}(h^2)\ . \tag{VI.45}$$

When substracting Eq. (VI.44) from Eq. (VI.45) we obtain

$$\vec{v}(t+h) = \vec{v}(t) + \frac{1}{2}\frac{[\vec{F}(t+h) + \vec{F}(t)]}{m}\,h + \mathcal{O}(h^2)\ . \tag{VI.46}$$

Remarks:

- Equations (VI.43) and (VI.46) define the velocity-Verlet algorithm. In this algorithm, the positions and velocities are calculated at the same time; it is symmetric.
- The algorithm satisfies Eq. (VI.38). The invariance under time reversal, that is under the transformation $h \to -h$, is not difficult to prove. To see this, we write

  $$\vec{r}(t) = \vec{r}(t-h) - \vec{v}(t-h)\,h + \frac{\vec{F}(t-h)}{2m}h^2 \tag{VI.47}$$

  and substract this equation from Eq. (VI.43), yielding

  $$\vec{r}(t+h) + \vec{r}(t-h) = 2\vec{r}(t) + \frac{\vec{F}(t)}{m}\,h^2\ . \tag{VI.48}$$

  Obviously, Eq. (VI.48) is invariant under the transformation $h \to -h$.

- The error for the calculation of the positions is of order h^4, that of the velocities is of order h^2. This allows to choose a "large" value for h.

- What does "large" mean? How do we have to choose h to begin with?

  The choice of h is dictated by the interaction potential. Let us consider an example. For a Lennard-Jones system Eq. (VI.48) reads for the x-component in reduced units [see Eqs. (VI.13), (VI.14)]

$$x^*(t+h) = 2x^*(t) - x^*(t-h) + F_x^*(t)\left(\frac{h}{\tau}\right)^2 \tag{VI.49}$$

  with

$$\tau = \sqrt{\frac{m\sigma^2}{48\epsilon}} \overset{\mathrm{Ar}}{=} 3.112 \cdot 10^{-13}\,\mathrm{s}\,, \tag{VI.50}$$

  where we employed the mass and LJ parameters of Argon in the last step. The characteristic time scale τ is of the order of the time period for an oscillation around the equilibrium position of an Ar atom. If we want to probe the dynamics on this time scale, we have to choose

$$h \ll \tau \qquad \text{for instance } \; h = 10^{-2}\tau \simeq 10^{-15}\,\mathrm{s}\,. \tag{VI.51}$$

The following pseudo-code illustrates the integration subroutine by the example of the velocity Verlet algorithm:

```
SUBROUTINE integrate(F,E)                     integrate Newton's equations

sumv  = 0
sumv2 = 0

do i = 1,N
   xx = 2*x(i)-xm(i)+F_x(i)*h*h               detemine the positions at time t+h
   yy = 2*y(i)-ym(i)+F_y(i)*h*h               according to the velocity-Verlet
   zz = 2*z(i)-zm(i)+F_z(i)*h*h               algorithm

   vx = (xx-xm(i))/(2*h)                      determine the velocities
   vy = (yy-ym(i))/(2*h)
   vz = (zz-zm(i))/(2*h)

   xm(i) = x(i)                               update the positions at time t-h
   xm(i) = x(i)
   zm(i) = z(i)

   x(i)  = xx                                 update the positions at time t
   y(i)  = yy
   z(i)  = zz

   sumv = sumv + vx + vy + vz                 velocity of center of mass
   sum v2 = sumv2 + vx**2 + vy**2 + vz**2     calculate total kinetic energy
enddo
T    = sumv2/(3*N)                            instantaneous temperature
Etot = (E + sumv2)/(2*N)                      total energy per particle

RETURN
END
```

Predictor-Corrector Algorithm

Contrary to previously discussed algorithms the predictor-corrector algorithm includes derivatives of higher than second order. The key idea of this algorithm is to integrate Eq. (VI.36) in two steps:

(a) *Predictor step*: This first step predicts the future positions from the first n derivatives of $\vec{r}(t)$ ('algorithm of order n').

(b) *Corrector step*: This second step corrects the predictions.

Let us illustrate this general procedure by a 3rd order algorithm:

- Step (a):

$$\begin{aligned}
\vec{r}^{\,\mathrm{p}}(t+h) &= \vec{r}(t) + \vec{v}(t)\,h + \frac{1}{2}\,\vec{a}(t)\,h^2 + \frac{1}{6}\,\dddot{\vec{r}}(t)\,h^3 \ , \\
\vec{v}^{\,\mathrm{p}}(t+h) &= \vec{v}(t) + \vec{a}(t)\,h + \frac{1}{2}\,\dddot{\vec{r}}(t)\,h^2 \ , \\
\vec{a}^{\,\mathrm{p}}(t+h) &= \vec{a}(t) + \dddot{\vec{r}}(t)\,h \qquad \text{(accelaration)} \ , \\
\dddot{\vec{r}}^{\,\mathrm{p}}(t+h) &= \dddot{\vec{r}}(t) \ .
\end{aligned} \tag{VI.52}$$

  Remark: Here the acceleration is given by $\vec{a}(t) = \ddot{\vec{r}}$ and is not calculated from the forces, which would imply $\vec{a}(t) = \vec{F}/m$.

- Step(b): Using the predicted positions $\vec{r}^{\,\mathrm{p}}(t+h)$ the force $\vec{F}(t+h)$ is calculated. From $\vec{F}$ we get the acceleration via $\vec{a}(t+h) = \vec{F}(t+h)/m$, and this enables us to determine the error of the prediction for the acceleration,

$$\Delta\vec{a}(t+h) = \vec{a}(t+h) - \vec{a}^{\,\mathrm{p}}(t+h) \qquad \text{[error of step (a)]} \ . \tag{VI.53}$$

  This error is then employed to correct all predictions of the predictor step, i.e.,

$$\begin{aligned}
\vec{r}^{\,\mathrm{c}}(t+h) &= \vec{r}^{\,\mathrm{p}}(t+h) + c_0\,\Delta\vec{a}(t+h) \ , \\
\vec{v}^{\,\mathrm{c}}(t+h) &= \vec{v}^{\,\mathrm{p}}(t+h) + c_1\,\Delta\vec{a}(t+h) \ , \\
\vec{a}^{\,\mathrm{c}}(t+h) &= \vec{a}^{\,\mathrm{p}}(t+h) + c_2\,\Delta\vec{a}(t+h) \ , \\
\dddot{\vec{r}}^{\,\mathrm{c}}(t+h) &= \dddot{\vec{r}}^{\,\mathrm{p}}(t+h) + c_3\,\Delta\vec{a}(t+h) \ .
\end{aligned} \tag{VI.54}$$

  The constants $c_o, .., c_3$ are chosen to optimize the *stability* and the *precision* of the integration (Gear 1971) (see M. P. Allen, D. J. Tildesley, *Computer Simulation of Liquids* (Clarendon, Oxford, 1987), appendix E for details).

A few remarks concerning the predictor-corrector algorithm shall be added here:

- One may think that the precision of the integration could be increased by iterating the corrector step, starting from the $\vec{r}^{\,\mathrm{c}}(t+h)$. In practice, this is never done, due to two main reasons. First, it requires to calculate the forces again, which is the most time consuming step and should thus be avoided. Second, an algorithm of order n will always have an error of order h^n so that the accuracy cannot be improved arbitrarily.

- Shouldn't one then increase n to improve the precision? Empirical tests show that an increase of n does not yield the desired result.

VI.3 Comparison of Molecular Dynamics and Monte Carlo Simulations

This final section presents a key-word like synopsis of the two molecular simulation methods discussed, the Monte Carlo and Molecular Dynamics simulation methods.

Monte Carlo (MC):

- **Variables:** $\{\vec{r}_i(t)\}_{i=1,\ldots,N} = \vec{x}$: point in *configuration space*
- **Thermodynamic ensemble (natural):** canonical ensemble, i.e., thermodynamic variables (T, V, N) $\rightarrow$ thermostated system
- **Thermodynamic observables:** calculation of a *time average* of a *stochastic trajectory*, a Markov process (importance sampling), i.e.:

$$A = \langle A \rangle_{\mathrm{c}} \overset{\text{(IV.27)}}{\approx} \langle A \rangle_{\mathrm{is}} = \frac{1}{M} \sum_{m=1}^{M} A(\vec{x}_m) \; .$$

- **Results:**
  - static properties (same as for MD)
  - Markovian dynamics: can be *physically realistic* on large time scales
  - possibility to simulate continuous and lattice systems

Molecular Dynamics (MD):

- **Variables:** $\{\vec{r}_i(t), \vec{v}_i(t)\}_{i=1,\ldots,N} = \vec{x}$: point in *phase space*
- **Thermodynamic ensemble (natural):** microcanonical, i.e., thermodynamic variables $(E, V, N) \rightarrow$ isolated system
- **Thermodynamic observables:** calculation of a *time average* of a *deterministic trajectory*: integration of Newton's equations, i.e.:

$$A = \langle A \rangle_{\mathrm{mc}} \overset{\text{(VI.7)}}{\approx} \frac{1}{M} \sum_{m=0}^{M} A(\vec{x}_m) \; .$$

- **Results:**
  - static properties (same as for MC)
  - Newtonian dynamics: *microscopically realistic* for classical systems
  - treatment of continuous systems