TP1: Euler's Algorithm-Radioactive Decay-Introduction to Fortran 90/95

January 11, 2021
University of Eloued
Physic's department : Master

1 Introduction

Computer simulations are nowadays an integral part of contemporary basic and applied research in the sciences. Computation is becoming as important as theory and experiment. In physics, computational physics, theoretical physics and experimental physics are all equally important in our daily research and studies of physical systems. Physics is the unity of theory, experiment and computation. Moreover, the ability "to compute" forms part of the essential repertoire of research scientists. Several new fields within computational science have emerged and strengthened their positions in the last years, such as computational materials science, bioinformatics, computational mathematics and mechanics, computational chemistry and physics and so forth, just to mention a few.

Simulations are suited for nonlinear problems which can not generally solved by analytical methods. The starting point of a simulation is an idealized model of a physical system of interest. We want to check whether or not the behaviour of this model is consistent with observation. We specify an algorithm for the implementation of the model on a computer. The execution of this implementation is a simulation. Simulations are therefore virtual experiments.

2 About Software

2.1 Compiler

A crucial tool in computational physics is programming languages. In order to run a code written in any programming language, we must first translate it into machine language, i.e. a language that the computer can understand. The translation is done by an `interpreter` or by a `compiler`: the former translates and immediately executes each instruction, the latter takes the file, produces the so-called `object` code that together with other `object` codes and with libraries is finally assembled into an executable file. Python, Java, Matlab, Mathematica are examples of "interpreted" language. Fortran, C, C++ are high-level "compiled" languages.

Our codes are written in Fortran 90/95. This is a sophisticated and complex language ofering dynamical memory management, arrays operations (e.g. matrix-vector products), modular and object-based structure. Fortran 90 however can be as effcient as Fortran 77 and maintains a wide compatibility with existing Fortran 77 codes.

In all cases, we need a Fortran 90 compiler. In PCs running Linux, the `gcc` compiler is basically part of the operating system and is always present. Recent versions of `gcc` include a Fortran

compiler, called `gfortran`. If this is absent, or it is not easy to install it, one can download the free and rather reliable compiler, `g95`. It is possible to install on Windows either `gcc` with `gfortran` or `g95`.

2.2 Visualization

Visualization of data produced by the codes has a central role in the analysis and understanding of the results. Code `gnuplot` can be used to make two-dimensional or three-dimensional plots of data or of analytical expressions. Gnuplot is open-source software, available for all operating systems and usually found pre-installed on Linux PCs. An introduction to `gnuplot`, with many links to more resources, can be found here: `http://www.gnuplot.info/help.html`. Another software that can be used is `Origin7.5`. This is an executable Windows program and has a graphical user interface and thus it is easier to use than gnuplot. One can also exploites `Matlab` or `Octave` graphics capabiltities.

3 The Physics-Radioactive Decay

In a spontaneous radioactive decay a particle with no external influence will decay into other particles. A typical example is the nuclear isotope uranium 235. The exact moment of decay of any one particle is random. This means that the number $-dN(t) = N(t) - N(t + dt)$ of nuclei which will decay during a time inetrval dt must be proportional to dt and to the number $N(t)$ of particles present at time t. In other words the probability of decay per unit time given by $(-dN(t)/N(t))/dt$ is a constant which we denote $1/\tau$. (The minus sign is due to the fact that dN is negative since the number of particles decreases with time). We write

$$\frac{dN(t)}{dt} = -\frac{N(t)}{\tau}. \tag{1}$$

The solution of this first order differential equation is

$$N(t) = N_0 \exp(-t/\tau). \tag{2}$$

The number N_0 is the number of particles at time $t = 0$. The time τ is called the mean lifetime. It is the average time for decay. For the uranium 235 the mean lifetime is around 10^9 years.

4 Numerical Analysis-Euler's Algorithm

Because many types of systems can be modeled by differential equations such as (1), it is important to know how to solve such equations. In general, analytical solutions of differential equations, that is, solutions in terms of well-known functions, do not exist. We are therefore motivated to find numerical solutions of differential equations. However, analytical solutions are very important and often exist in special or limiting cases. We often use them to test our numerical solutions.

The standard technique for numerically solving a differential equation is to convert the differential equation to a finite difference equation. Let us consider a first-order differential equation of the form

$$\frac{dy}{dx} = f(x, y) \tag{3}$$

and analyze its meaning. Suppose that at $x = x_0$, y has the value y_0. Because (3) tells us how y changes at (x_0, y_0), we can find the approximate value of y at the neighboring point $x_1 = x_0 + \Delta x$,

if Δx is small. The simplest approximation is to assume that $f(x, y)$, the rate of change of y with respect to x, is constant over the interval x_0 to x_1. Then the approximate value of y at $x_1 = x_0 + \Delta x$ is given by

$$y_1 = y(x_0) + \Delta y \approx y(x_0) + f(x_0, y_0)\Delta x.$$

We can repeat this procedure to find the value of y at the point $x_2 = x_1 + \Delta x$:

$$y_2 = y(x_1 + \Delta x) \approx y(x_1) + f(x_1, y_1)\Delta x.$$

This procedure can be generalized to calculate the approximate value of y at any point $x_{n+1} = x_n + \Delta x$ by the iterative formula

$$y_{n+1} = y_n + f(x_n, y_n)\Delta x. \quad (Euler algorithm) \tag{4}$$

Equivalently we can consider that we have approximated the derivative in (3) with a *forward difference* using a truncated Taylor's expansion

$$\left.\frac{dy}{dx}\right|_n \approx \frac{y_{n+1} - y_n}{\Delta x}$$

which leads to the finite difference equation 4. Another way to derive the Euler Algoritm is by integrating the equation (3) over a small interval $\Delta x = x_{n+1} - x_n$

$$y(x_n + \Delta x) = y(x_n) + \int_{x_n}^{x_{n+1}} f(x', y)dx'$$

The approximation come with writing the integral as

$$\int_{x_n}^{x_{n+1}} f(x', y)dx' \approx f(x_n, y_n)\Delta x$$

to obtain

$$y_{n+1} = y_n + f(x_n, y_n)\Delta x.$$

The Euler algorithm sometimes is called the constant slope algorithm. We expect that (4) will yield a good approximation to the exact value of y if Δx is sufficiently small. From this later derivation it is clear that the Euler algorithm assumes the rate of change of y to be constant over the interval x_n to x_{n+1}, and that the rate of change can be evaluated at the *beginning* of the interval.

5 Order of Accuracy

The major factors to be considered in evaluating/comparing different numerical methods are the accuracy of the numerical solution and its computation time. It is important to note that the evaluation/comparison of numerical methods is not so simple because their performances may depend on the characteristic of the problem at hand. It should also be noted that there are other factors to be considered, such as stability, versatility, proof against run-time error, and so on. In this section, we will deal only with the accuracy of numerical solution. How accurate is the Euler method? To quantify this we use a Taylor expansion of $y(x_n + \Delta x)$ around x_n and substitute it into (4)

$$y(x_n + \Delta x) = y(x_n) + \Delta x \left.\frac{dy}{dx}\right|_n + \frac{1}{2}(\Delta x)^2 \left.\frac{d^2y}{dx^2}\right|_n + ... \approx y(x_n) + f(x_n, y_n)\Delta x = y(x_n) + \Delta x \left.\frac{dy}{dx}\right|_n \tag{5}$$

where we have used (3) to obtain the final form. Hence, we see that the term in Δx in the expansion has been correctly reproduced by the approximation, but that the higher order terms are wrong. We therefore describe the Euler method as 1^{st} order accurate.

In general, *an approximation to a quantity is n^{th} order accurate if the term in $(\Delta x)^n$ in the Taylor expansion of the quantity is correctly reproduced.* The order of accuracy of a *method* is the order of accuracy with which the unknown is approximated.

6 Error

There are two types of errors: truncation (algorithm) and roundoff . Truncation error is the error of truncating the Taylor's series after the n^{th} term. Thus in (5) the truncation error is the term in $(\Delta x)^2$. Roundoff error occurs because of the limited precision of computer arithmetic operations. The truncation error exists independently from any roundoff errors.

While it is possible these two kinds of error could cancel each other, given our pessimistic worldview, we usually assume they are additive. We also expect some tradeoff between these two errors as Δx varies. As Δx is made smaller, the truncation error presumably decreases, while the roundoff error increases. This results in an optimal Δx-value for a given method. Unfortunately, the optimal Δx-value will usually depend on the given differential equation and the approximation method. In general it is impossible to predict the optimal Δx if the differential equation solution is unknown.

Both kinds of error can accumulate, since numerical algorithms usually involve many steps. The sum of the errors in each step is the local error. Errors in each step will accumulate to produce a global error for the calculation. If the calculation has m steps:

- local errors with random signs usually results in a global error $\sim \sqrt{m} \times$ average local error.
- local errors with mostly the same sign can result in a larger global error $m \times$ average local error.

7 First example: Radioactive Decay

We first use the Euler algorithm to compute the numerical solution of the differential equation (1) with the initial condition $N(0) = N_0$ at $t = 0$. With this new variables of the problem we find $N \equiv y$, $t \equiv x$ and $-N/\tau \equiv f(x, y)$. In this case the Euler algoritm reads

$$N(t_n + \Delta t) = N(t_n) - \Delta t \frac{N(t_n)}{\tau}. \tag{6}$$

We will start from the number of particles at time $t = 0$ given by $N(0) = N_0 \equiv N(1)$. We substitute $t = 0$ in (6) to obtain $N(\Delta t) \equiv N(2)$ as a function of $N(1)$. Next the value $N(2)$ can be used in equation (6) to get $N(2\Delta t) \equiv N(3)$, etc. By choosing a number of steps m such that the total time interval is $T = m\Delta t$, we are led to the time discretization

$$t \equiv t(i) = (i-1)\Delta t \ , \ i = 1, ..., m+1. \tag{7}$$

and the numerical solution (6) can be rewritten as

$$N(i) = N(i-1) - \Delta t \frac{N(i-1)}{\tau} \ , \ i = 2, ..., m+1. \tag{8}$$

here $t(i)$ and $N(i)$ represents two arrays in which we store the values of time t and the number of nuclei $N(t)$ at each step.

The local truncation error estimate from Taylor series is

$$(\Delta t)^2 \frac{d^2N(t)}{dt^2} = (\Delta t)^2 \frac{N(t)}{\tau^2}$$

and hence the global error estimate reads

$$m \times (\Delta t)^2 \frac{N(t)}{\tau^2}, \ where \ \ m = \frac{t}{\Delta t} \ numberofsteps$$

We divide the global error by $N(t)$ to get relative error estimates.

8 Another Example-Air Resistance

We consider an athlete riding a bicycle moving on a flat terrain. The goal is to determine the velocity. Newton's second law is given by

$$m\frac{dv}{dt} = F.$$

F is the force exerted by the athlete on the bicycle. It is clearly very difficult to write down a precise expression for F. Formulating the problem in terms of the power generated by the athlete will avoid the use of an explicit formula for F. Multiplying the above equation by v we obtain

$$\frac{dE}{dt} = P.$$

E is the kinetic energy $E = \frac{1}{2}mv^2$ and P is the power $P = Fv$. Experimentaly we find that the output of well trained athletes is around $P = 400$ watts over periods of $1h$. The above equation can also be rewritten as

$$\frac{dv^2}{dt} = \frac{2P}{m}.$$

For P constant we get the solution

$$v^2 = \frac{2P}{m}t + v_0^2.$$

We remark the unphysical effect that $v \longrightarrow \infty$ as $t \longrightarrow \infty$. This is due to the absence of the effect of friction and in particular air resistance.

The most important form of friction is air resistance. The force due to air resistance (the drag force) is

$$F_{\rm drag} = -B_1 v - B_2 v^2. \tag{9}$$

At small velocities the first term dominates whereas at large velocities it is the second term that dominates. For very small velocities the dependence on v given by $F_{\rm drag} = -B_1 v$ is known as Stockes' law. For reasonable velocities the drag force is dominated by the second term, i.e it is given by $F_{\rm drag} = -B_2 v^2$ for most objects. The coefficient B_2 can be calculated as follows. As the bicycle-rider combination moves with velocity v it pushes in a time dt a mass of air given by $dm_{\rm air} = \rho A v dt$ where ρ is the air density and A is the frontal cross section. The corresponding kinetic

energy is $dE_{\text{air}} = dm_{\text{air}}v^2/2$. This is equal to the work done by the drag force,i.e $-F_{\text{drag}}vdt = dE_{\text{air}}$. From this we get

$$B_2 = C\rho A. \tag{10}$$

The drag coefficient is $C = \frac{1}{2}$. The drag force becomes

$$F_{\text{drag}} = -C\rho A v^2. \tag{11}$$

Taking into account the force due to air resistance we find that Newton's law becomes

$$m\frac{dv}{dt} = F + F_{\text{drag}}. \tag{12}$$

Equivalently

$$\frac{dv}{dt} = \frac{P}{mv} - \frac{C\rho A v^2}{m}. \tag{13}$$

9 Fortran Code

In order to appreciate what a program is and how to design a program in FORTRAN 90/95 programming language we will consider a simply physical problem: Radioactive Decay problem which we are going to solve using a computer. Clearly, the analytical solution in (2) is so simple that we do not need to write a program at all. But we have to start somewhere, so let us discuss how we might write a program to compute the nuclie number present at a particular time. A program is just a set of instructions to the computer to perform a series of operations. Those operations will often be mathematical calculations, decisions based on equalities and inequalities, or special instructions to say write output to the screen or to a given data file.

The program consists of "source code" which is "stored" in a text file. This code contains the instructions in a highly structured form – a "Programming Language". Each programming language has a different set of rules (or syntax) for specifying these operations, although the logical operations we ask the computer to do are independent of the language.

9.1 The Overall Program

A Fortran program generally consists of a main program (or driver) and possibly several subprograms (or procedures). For now we will assume all the statements are in the main program; subprograms will be treated later. Any main program may begin with a *program* statement (not mandatory) and must conclude with a corresponding *end program* statement. The *program* statement allows us to give a name to the program. The *end* statement may be preceded by a *return* or *stop* statement. This looks like

```
program radioactivity
! myfirstprogram
!
write(*,*)'startoftheradioactivityprogram'
return
endprogram radioactivity
```
(14)

This is a an overall structure of a complete Fortran 95 program. We have chosen the name "radioactivity" for our program. The program executes line by line from the first line to the last. The lines beginning with a '!' are not instructions to the computer but comments for us so that we can document what we are doing, and can be included anywhere in the program. Everything after the exclamation mark will be ignored by the compiler. The asterisks (*,*) following the *write* statement is a command telling the computer to output something to the screen, in this case the text 'Start of the radioactivity Program'. This `write`$(*,*)$ statement may be replaced by the `print*,` statement which has the same role, note the comma after the *. Because F95 dose not distinguishes between upper and lowercase characters, `write` and `WRITE` for example are identical within the source code.

9.2 Variables, Declaration and Types:

We need some way of storing the current values of quantities such as time t and nuclei number N in our example and of performing mathematical operations on them. Computers can store information only as binary numbers, that is, sequences of ones and zeros. Every one or zero is called a bit and a group of 8 bits is referred to as a Byte. For example, the integer 362 is stored as 0000000101101011 using two Bytes of memory. It would be difficult to write a program if we had to write numbers such as the speed of light in binary format. High-level computer languages allow us to reference stored numbers using *identifiers* or *variable names*. A valid identifiers is a series of characters consisting of letters, digits, and underscores that does not begin with a digit nor contain any spaces. Variables may be of different types, the most important being *integer*, *real* (holding floating point variables), *character* and *complex*. In our example we shall only need *real* and *integer*. The general form for declaring variables is: `type  name_of_variable`.

In Fortran 90/95, a declaration of an integer is `interger(kind=m)`. If one choose m = 2, the programmer reserves 2 bytes (16 bits) of memory to store the integer variable wheras m = 4 reserves 4 bytes (32 bits). Although it may be compiler dependent, just declaring a variable as `interger` , reserves 4 bytes in memory as default. Note that the `(kind=2)` can be written as `(2)`. Normally however, we will for Fortran programs just use the 4 bytes default assignment `integer`. Here are some examples of program declarations

```
integer(kind=2) ::  i , m !  i and m are declared as two integer variables
real(kind=8)    ::  N, dt !  N and dt are declared as two real variables
```

In Fortran real variables are declared as `real(kind=8)` or `real(kind=4)` for double or single precision, respectively. In general we discourage the use of single precision in scientific computing, the achieved precision is in general not good enough. A constant is simply a number occuring directly in the program. The type of a constant is integer, unless it contains a decimal point or an exponent, when it is real. Real numbers are written as `2.0, 2.E0, 1.3D12` meaning 2.0, 2.0×10^0, 1.3×10^{12}. Using a `D` rather than an `E` for the exponent causes the constant to be double precision. The next program performs a single step of our calculation using our problem variables N, t, τ,

and Δt with some new featurs.

```
program radioactivity

  implicitnone

  ! Definevariablesusedtoholdtime and nuclei number

  real(kind = 8):: N, dt, t, tao

  ! Setinitialvaluesfortime t and nuclei number N

  t=0.D0 ;  N=100.D0
  ! Choosethetimestepandthemean lifetime

  dt=0.1D0 ;  tao=5.D0

  t=t + dt

  N=N - (dt * N)/tao

  write(*, *) t, N
endprogram radioactivity
```

Our program now has two types of statements. The line near the top beginning `real` is called a declaration statement and is used to define the variables in our program. The remaining lines are called executable statements and are instructions to the computer to perform certain operations, in this case a few simple calculations and assignments. A program may have many lines of declarations and all of them must precede the executable statements. There are few new important things to note:

- a single line may contain two or more statements and must be separated by semicolons ';'.
- all variables must be initialized (have a value) before we can use them.
- the line `t=0.D0 ;  N=100.D0` assigne the initial value `0.D0` and `100.D0` to the variable `t` and `N` respectively.
- a line such as: `t=t + dt` appears to be mathematical nonsense, but the operator '`=`' should be interpreted as meaning assignment not equality. Thus the line calculates `t + dt`, and stores the answer back into the original memory location that `t` occupied. The old value of `t` is overwritten (updated).
- The asterisk '`*`' and the slash '`/`' are simple arithmetic operators for Multiplication and Division respectively.

- the statement `implicit none` says that we require all variables we use to be declared, and therefore detect possible errors already while compiling. Without `implicit none;`simply referencing a variable causes it to be declared automatically, this is Fortran's implicit declaration rule: *integer* if the first letter of the variable is in the range i-n, *real* otherwise. In many Fortran 77 one can see statements like `implicit real*8 (a-h,o-z)`, meaning that all variables beginning with any of the above letters are by deafult floating numbers. However, such a usage makes it hard to spot eventual errors due to misspelling of variable names.
- the `write(*,*)`statement accepts a comma-separated list of things to print out.

9.3 Repetition: loops

For the radioactivity problem, we see from equation (8) that the same operation in calculating N will be repeated each time, but at different values. Therefore, the next step in our program is to get it to repeat the calculation many times. We do this using a *loop*. *Loops* are one of the key ideas in programming enabling us to repeat operations many times.

The statement to implement a loop is called `do` and is terminated by `end do`. The statements between these two are repeated for a specified number of times. Here is a loop which will execute twenty times:

```
do i=1,20
write(*,*) i, i*i
end do
```

An integer variable, `i`, called *loop variable* is used to keep count of how many times we have gone round the loop. It is assigned an initial value of 1 and the loop continues until `i` exceeds 20. The full syntax of a do statement is `do i=start,end,step`, where the '`,step`' is optional. If given, it specifies how much to add to `i` between each iteration of the loop. The final iteration will be the last one which does not cause `i` to exceed `end`. Note that the do-loop variable must never be changed by other statements within the loop! This will cause great confusion. Fortran also provides other loops construct such as *do while* loop and infinite *do* loop:

```
do   while (condition)

       statments

end  do
```

```
do
     statments
  if ( condition ) exit
     statments
end do
```

The statements in the body will be repeated as long as the condition in the `while` statement is true. The `exit` statement provides a way to leave a *do* or *do while* loop before all the iterations of that loop are finished. When the condition in the `if` statement is true, the `exit` causes the program to jump to the statement following the current loop, whether the loop is infinite or finite. Here is an example that calculates and prints all the powers of two that are less than or equal to 100:

```
i=1
do while(i<=100)
 i=2*i
 write(*,*) i

end do
```

```
i=1
do
  i=2*i
   write(*,*) i
    if (i<=100) exit
end do
```

Now let's modify our program:

```
program radioactivity

 implicitnone

 ! Definevariablesusedtoholdtime and nuclei number

 real(kind = 8):: N, dt, t, tao
 integer :: i
 ! Setinitialvaluesfortime t and nuclei number N

 t=0.D0 ;  N=100.D0
 ! Choosethetimestepandthemean lifetime

 dt=0.1D0 ;  tao=5.D0
 ! loop updating the t , N, and outputting the values
 do i=1,200
    t=t + dt
    N=N - (dt * N)/tao
    write(*,*) t, N
    write(*,*)
 end do
endprogram radioactivity
```

We note here:

- how all the declarations must come at the top of the program.
- we put the output statement, i.e. the `write` inside the loop to keep track of the answer.
- we have indented the lines between `do` and `end do`. This is good practice but not essential, however it provides a clear visual clue that these statements are within the "do loop" and will be repeated a number of times (200 in this case).
- the write(*,*) with no text following it is valid and produces a blank line.

9.4 Arrays:

Simple scalar variables do not cover every need in Physics. F95 provides us with the concept of an array: variables containing multiple scalars. The arrays provide a very natural way of defining vectors and matrices. A one dimensional array (vector) A is an ordered list of m variables of a given data type called the elements of the array and denoted $A(1)$, $A(2)$,...,$A(m)$. `A(i)` is an element of the array, and `i` is the array index or subscript.The following statements show how array variables are declared:

```
real(kind=8)::  v(3),r(3)                !  define two vectors v and r each of 3 elements
real(kind=8),dimension(1:3)::  v , r     !  the same as above
real(kind=8),dimension(3)  ::  v , r     !  the same as above
real(kind=8)::  mat(3,4) !  define a matrix of 3-by-4 elements(two dimensional array)
real(kind=8),dimension(1:3,1:4)::  mat !  the same as above
real ::  x(-200:200) !  define a vector x of 401 elements
real, dimension(-1:3,2,0:5) ::  b!  define a three dimensional array
```

The Indices of an arrays must be integers or integer expressions: `mat(2*i,j+7)` is quite valid. In Fortran the default lower limit (bound) of the range of an array is 1 and not 0. The bounds of the array `mat` in the last example are `(1-3,1-4)` and any indices must lie within this range. Exceeding the bounds of an array is a common error, and is often referred to as '*falling off the end*' of the array. A reference to `mat(101,100)` would be such a mistake. Let's see how our program looks using two array variables `t` and `N` :

```
program radioactivity

 implicitnone

 ! Definevariablesusedtoholdtime and nuclei number
 real(kind = 8):: N(1 : 200), dt, t(1 : 200), tao
 integer :: i

 ! Setinitialvaluesfortime t and nuclei number N
 t=0.D0 ; N=0.D0    ! here we initialize all the array elements by zero
 t(1)=0.D0 ;  N(1)=100.D0

 ! Choosethetimestepandthemean lifetime
 dt=0.1D0 ;  tao=5.D0

 ! loop calculating t(i) , N(i), and outputting their values
 do i=2,200
    t(i)=t(i-1) + dt
    N(i)=N(i-1) - ( dt * N(i-1) )/tao
    write(*,*) t(i), N(i)
 end do
endprogram radioactivity
```

It is clear that our problem does not require to use arrays, but we just have explained how to use them. We note here some new points:

- with the statement `t=0.D0 ; N=0.D0` , we hold the constant `0.D0` in eash element of the arrays `t` and `N`. There is nothing intrinsic in this operation, it is just an assignment via an *implicit loop*: one way of storing a value to an array elements or copying array to another

would be:

```
do i=1,m
a(i)=b(i)
t(i)=0.D0
c(i)=a(i)*b(i)
end do
```

an abbreviated syntax for this provided by Fortran 90/95 is : `a=b ; t=0.D0.; c=a*b`. This syntax requires `c`, `a` and `b` to be the same length, or for `a` and/or `b` to be simple scalar variables or constants (e.g. `a=2*a` doubles all the elements of `a`).

- the loop variable `i` is assigned an initial value of `2` rather than `1`, to avoid exceeding the arrays limits. With an initial value of 1, it would be an error: the array index `i,` while evaluating the expressions inside the loop, falls at `0` which does not corresponds to the declared arrays' lower limts.

9.5 The parameter statement:

Occasionally it is useful to define a variable in such a way that it cannot be changed when the program runs. Such a variable is called a parameter, and everything about it can be determined when the program is compiled. This stops us from making possible mistakes in the rest of the program. For example we may change our program declarations as follows:

integer`, parameter :: m=200 !  the upper limit of the arrays`
real(`kind` = 8)`, parameter` :: $tao = 5.0D$
real(`kind` = 8):: $N(1:m), dt, t(1:m)$
integer :: `i`

Here tao can be used like any other variable, except that any attempt to change its value will result in an error. Since the size of our arrays will not change throughout the program, so that their upper limits must be declared with a *parameter statement.*

9.6 The write and format statements:

The output of the above program does not fall into the neat columns that one might hope for. This can be improved by using a `format` statement to tell the `write` statement precisely what to do. The second asterisk in `write(*,*)` is a placeholder for a format code, and implies a free format. We can specify the format of the data by using a proper format code instead of the asterisk; this allows to specify for example, the number of significant digits, ect. By replacing the line `write(*,*) t(i) , N(i)` in our program by

`write`($*$,'(`f11.5,E17.7`)') `t(i) , N(i)`

then this statement says: prints to the screen in the same line `t(i)` as simple decimal (floating) number, padded to a width of at most 11 characters, with precisely 5 characters after the decimal point, and `N(i)` as real number in scientific notation (exponential format) padded to a width of at most 17 characters, with precisely 7 characters after the decimal point. If the number does not fill up the entire specified width, blank spaces will be automatically padded on the left. Note that the default asterisk format usually specify a width of 14 characters for real numbers.

Instead of giving the format code directly in the write statement, one can specify the format code in a separate labled line anywhere within the program via the `format` statement. So the above variables could be output in the same format with the following:

```
write(*,23) t(i) , N(i)

23 format(f11.5,E17.7)
```

here the format code of the write statement has beeen replaced by a 23. This tells the compiler to look for a format statement somewhere in the program and labelled by a 23. The `format` statement may occur before or after the write. Many writes may refer to the same format statement. The format statement is completely ignored at the point when it would be due for execution. The label can be any integer of five or fewer digits.

The complete list of options for format is large, some of them for reference are;

ix	integer, at most x characters
i$x.y$	integer, at most x characters and at least y (padded with leading zeros)
f$x.y$	floating point, at most x characters and precisely y characters after the decimal point
e$x.y$	exponential, at most x characters and precisely y characters after the decimal point
d$x.y$	ditto, but use `'D'` not `'E'` to represent exponent
a	ASCII string
ax	ASCII string, at most x characters
g$x.y$	general – treated as **f**$x.y$ for numbers close to one, as **e**$x.y$ otherwise, and as **i**x if used on an integer
`'any text'`	literal text
x	horizontal skip (space)
/	vertical skip (newline)

Any of the above descriptors can be preceded by a repeat count, integer indicating how many sequential variables should be processed with the following edit descriptor:

```
      write(*,500) w, x, y, z
500   format(4e15.3)
```

This repeat count can be combined with parentheses to obtain more complicated results:

```
      write(11,200) i, x, j, y, k, z
200   format( 3(i5,e15.3))
```

this produces the proper pairs alternating between integer and exponential format.

For horizontal spacing, the **nx** code is often used. This means `n` horizontal spaces. If `n` is omitted, `n=1` is assumed. This produces spacing in a line not accounted for in extra length of a variable descriptor. For vertical spacing (newlines), use the code `/`. Each slash corresponds to one newline and may be preceded by a repeat count `n/`. Note that each write statement by default ends with a newline. As example:

```
write(*,23) t(i) , N(i)
23 format(5x,f11.5,10x,E17.7)
```

means precedes the variable t(i) with 5 blank spaces in the line and follows it with 10 spaces before printing N(i).

Sometimes text strings are given in the format statements:

```
      write(*,2050) x, dx
2050  format(f10.4,'+/-',f10.4)
```

As an advanced example, say we have a n by m two-dimensional array of integers and want to print it. Here is how:

```
      do i=1,n
      write(*,3000) (mat(i,j), j=1,m)
3000  format(i6)
```

here we have an explicit do loop over the rows and an *implicit* loop over the column index j .

It is also possible to allow repetition without explicitly stating how many times the format should be repeated. Suppose you have a vector where we want to print the first 50 elements, with ten elements on each line. Here is one way:

```
      write(*,3020) (vec(i), i=1,50)
3020  format(10i6)
```

Writing directly to a file is convenient when the output is too much to fit on the screen. This can be achieved by specifying an Input/Output *unit* number of a file, in the first argument of the write statement. The role of this unit is to tell the computer where our information would be sent. The first asterisk in the write statements shown so far, indicates now the default unit (usually the secreen). An integer value would indicate a number of a file that we had opened earlier with the open statement (will be seen later). By default, without open, the statement

```
write(10,'(f11.5,E17.7)') t(i) , N(i)
```

with an integer value of 10 for example by itself creates a file called 'fort.10'.

9.7 Subprograms: function and subroutine

A subprogram is a self-contained program segment that carries out some specific, well-defined task. In a large program, one often has to do the same sepecific task with many different data. Instead of replicating code, these tasks should be done by subprograms . Fortran 90/95 provides us with two different types of subprograms (procedures): called functions and subroutines, which returns the results of specific calculations given some parameters. Programmers are allowed to define their own subprograms, but F95 also provides some built-in or intrinsic functions and subroutine, for example trigonometric functions, square roots etc. The use of programmer defined subprograms permits a large complicated program to be broken down into a number of smaller, self-contained units. In other words, a main program can be modularized via the sensible use of programmer-defined subprograms.The use of subprograms alows other people to grasp the logical structure of a program with far greater ease than would otherwise be the case.

Subprograms of type function are invoked just like a mathematical function: both of them take a set of input arguments (parameters) and return a value of some type. In general, a function

always has a data type. The general syntax of a Fortran 90/95 function is :

```
type  function name (list-of-variables)
        .   .   .
        name= .   .   .

        return
      end function name
```

```
function name (list-of-variables)
  .   .   .
  type name
  name= .   .   .
  return
end function name
```

We see that the structure of a function closely resembles that of the main program. The main differences are:

- Functions have a type. This type must also be declared in the calling program.
- The return value should be stored in a variable with the same name as the function.
- Functions are terminated by the `return` statement instead of `stop`.

The function has to be declared with the correct type in the calling program unit. The function is then called by simply using the function name and listing the parameters in parenthesis.

A Fortran function can essentially only return one value. Often we want to return two or more values (or sometimes none!). For this purpose we use the `subroutine` construct. The syntax is as follows:

```
subroutine name (list-of-variables)
  .   .   .
  return
end subroutine name
```

Subroutines have no type and consequently should not (cannot) be declared in the calling program unit, and are invoked using a `call` statement. Subprograms are said to be *Internal* procedures (internal functions or internal subroutines) if are placed just before the `end` statement for a main program, or subprogram, and preceded by a special statement, `contains`. If they are placed just after the end statement or in a separate file, are said to be *external* procedures. Internal procedures have knowledge of variables within the unit in which they are contained, and only that program unit can reference them. However, they also have the power to hide variables from the calling program. Any variable declared by a type statement within the internal procedure is local to that procedure. Note that if a program contains multiple subprograms then their definitions may appear in any order. Once the subprogram has carried out its intended action, control is returned to the point from which the function was accessed. Note here, that many details about using subprograms have been intentionally omitted. In our next program we will only use subprogram of type function in one of its simple forms.

9.8 The final program

By putting now all thing together we get

```
program radioactivity

 implicitnone

 ! Definevariablesusedtoholdtime and nuclei number
 real(kind = 8):: N, dt, t, tao = 5.D0, N0, ti, tf
 integer :: i,m
 write(*,'(a$)') 'Give the initial nuclei number, N0 ='
 read(*,*) N0
 write(*,'(a$)') 'Give the integration interval,ti and tf: '
 read(*,*) ti,tf
 do while (tf¡ti)
    write(*,'(a$)') 'tf should not be less then ti, please give again,ti and tf: '
    read(*,*) ti,tf
  enddo
  ! read the step number
  write(*,'(a$)') 'Give the number of steps, m= '
  read(*,*) m
 ! Setinitialvaluesfortime t and nuclei number N
 t=ti ; N=N0
 ! Calculatethetimestep
 dt=(tf-ti)/m
!  open file for results
 open(10, file ='output.dat')
 ! loop calculating t(i) , N(i), and outputting their values
 do i=1,m+1
    write(10,'(f11.5, E17.7)') t, N
     t=t + dt
     N=N +dt*f(t, N)
 end do
!  close file
 close(10)
contains
!  this function provides the value of dNbydt at (t,N)
 real(kind = 8) function f(x, y)
    real(kind = 8), intent(in) :: x, y     !x and y are local variables
    f = −y/tao                             ! tao is a global variable
 end function f
endprogram radioactivity
```

There are lots of things to note here:

- a one line may contain a declaration and assignments of many variables.
- the `read(*,*)` statement reads a list of values from the keyboard and assign the values to the variables in the variable list, in this case one variable. The values read can be separated

by spaces or be on separate lines, but not separated by commas. The read statement also accepts a format code and a unit number as the write statement.

- The `open` statement creates a file called 'output.dat' and associates it with I/O unit number 10. This statement must come before the first use of that unit number in a `write` statement. Not all unit numbers are allowed in the `open` statement. Unit numbers 5 and 6 should not be used: 5 often corresponds to reading from the terminal (`read(*,*)`), 6 to writing to it (`write(*,*)`). Units 10 through 99 seem to work well with disk files.

- The `close` statement (close the file after we have written everything to it) is good practice, and should follow the last use of that I/O unit. This statement disconnects the external file from the I/O unit number of value 10.

- Note the way of calling a function, this is very similar to obvious mathematical notation. A function typically returns a one value.

- The function receives a dummy arguments `x` and `y`. `intent(in)` means that the dummy argument cannot be changed within the subprogram. `intent(out)` means that the dummy argument cannot be used within the subprogram until it is given a value with the intent of passing a value back to the calling program. The statement `intent(inout)` means that the dummy argument has an initial value which is changed and passed back to the calling program. We recommend using these options when calling subprograms. This allows better control when transfering variables from one function to another.

- You should match arguments one for one in count and type between the reference to the function and the function itself.

- Names used for argument need not match between the function argument list and the calling sequence.

- A variable not in the argument list, but with the same name used in another subprogram (or the main program) list does not refer to the same address in memory as its namesake elsewhere, unless you take special action (common blocks and modules). Subprograms are completely separate entities with very limited means of communicating information. You don't worry about conflicts in variable names or label values from a program to a subprogram or between subprograms.

- Values of arguments can be changed within the function, providing a means for a function to generate more information for use elsewhere in the program.

- Note the way in which the value of the variable tao is known in the function without passing it through the function argument.

10 Lab Work 1

1-Editing Fortran Program: Under Windows operating system, we open Compaq Visual Fortran by double clicking its icone in the desktop.Then Fortran editor appears for you. Following the above steps write a code for the radioactive decay problem. In PCs running Linux or In PCs running Windows with MinGW/Cygwin shell, you need an editor, like Emacs, Programer's Notepad

or Notepad++. In all cases after writing the code, save the file with a `.f90` extension, e.g. something like `decay.f90`. MinGW a contraction of "Minimalist GNU for Windows", is a minimalist development environment for native Microsoft Windows applications, and Cygwin is a collection of tools which provide a Linux look and feel environment for Windows.

2-Compilation and Execution: With Compaq Visual Fortran, we compile our file program by clicking the push button *compile*. If there is no errors we run the program by clicking the push button *Go* or by clicking the button *Execute decay.exe* in the *Build menu*. Under Linux we open a terminal or under Windows we open MinGW/Cygwin shell. Then we will use the `gfortran` or `g95` compiler by typing the command for this and hitting return:

$$\texttt{gfortran} \; -o \; \texttt{decay} \; decay.f90$$

The option -o (for output) tells the compiler to place the result of his work in an executable program called decay. We run the program by typing and then hitting return:

$$\texttt{./decay} \; \textit{or} \; \texttt{decay.exe}$$

We obtain the output file output.dat (assuming that you have written the output to a file named 'output.dat'). We can open it using Matlab, Origine or the `cat` command in the shell.

3-Visualization: We start gnuplot by typing `gnuplot` in the shell, you will then see the following prompt

$$\texttt{gnuplot} >$$

then you may need type `help` for a list of various commands and help options. Our data file output.dat contains two columns of data points, where the first column refers to the argument t while the second one refers to the computed number of nuclei $N(t)$. If we wish to plot these sets of points with gnuplot we just need to type and then hit return

$$\texttt{gnuplot} > \texttt{plot 'output.dat' using 1 : 2 w l}$$

or

$$\texttt{gnuplot} > \texttt{plot 'output.dat' w l}$$

since gnuplot assigns as default the first column as the x-axis. The abbreviations `w l` stand for 'with lines'. If you prefer to plot the data points only, type

$$\texttt{gnuplot} > \texttt{plot 'output.dat' wp}$$

This will typically produce the graph on the screen. If we wish to save this graph as a PostScript or Jpeg file, we can proceed as follows

```
gnuplot > set terminal postscript
gnuplot > set output 'filename.ps'
gnuplot > plot 'output.dat' w l
```

the first line of sequence changes the terminal type to PostScript. By typing this, we will be the owner of a postscript file called filename.ps, which can be displayed with ghostview through the call

$$\texttt{gv filename.ps}$$

After we have finished plotting to a file, we set the terminal type back to 'windows' or to 'X11':

```
setterminal win
```

We may need to fit our data by choosing a function $f(x)$. The function $f(x)$ will depend on a some set of parameters a, b,.... To fit this function to the data we type in gnuplot window and hit return

$$\texttt{gnuplot} > \texttt{f(x)} = \texttt{a} + \texttt{b} * \texttt{x} * \texttt{x}$$
$$\texttt{gnuplot} > \texttt{fit f(x) `output.dat` via a, b, ...}$$

To plot the fit $f(x)$ together with the data we plot the data then use the command *replot* to plot the function $f(x)$ on the top of it.

We can also use Matlab/Octave. We run Matlab by double clicking its icone and typing in its command window and then hitting return

$$\texttt{load output.dat; plot(output(:,1); output(:,2))}$$

or with Octave

$$\texttt{octave} > \texttt{load output.dat; plot(output(:,1); output(:,2))}$$

10.1 Home work:

Write a fortran program for the Air Resistance problem. We comment on the diference between the cases with and without air resistance. We also discuss the limit $\Delta t \to 0$. We write down the obtained fits with the corresponding errors. We write a report with conclusions.