TUTORIAL ON SCIENTIFIC COMPUTING USING FORTRAN AND GNUPLOT

PRAVEEN. C

1. Introduction

In this tutorial, we will learn the basics of programming with a view towards numerical computing. We will study and write simple programs[1] in fortran to solve some elementary problems in numerical analysis. The best way to learn a new programming language is to read many examples of code and to actually write your own code. So the tutorial will give you many examples of small computer codes which solve some equation. The theory behind the numerical method will not be explained in this tutorial. It is assumed that you already have some knowledge with numerical analysis, like root finding, solution of system of linear equations, solution of ODE and PDE. However in every example the numerical recipe will be fully given so that you can use it to write a program. You will also learn the use of plotting software to visualize the results of numerical computing.

1.1. Pre-requisites.

This tutorial will assume that you are using a UNIX/Linux/MAC computer. We will assume that you know how to use a terminal and execute commands on the terminal. You are also expected to already know how to use a text editor. Examples of text editors are Vi/Vim and Emacs. We will use fortran as the programming language and all examples will be given in fortran. Fortran has been and still is the workhorse of numerical computing. A large number of scientific libraries are written in fortran and it is good to have some knowledge of this important language. However for the purpose of any exercises you are free to write the programs in C.

1.2. Language and compilers.

The fortran language is under considerable development. The old version of the language is called Fortran-77 which is fast going out of fashion due to emergence of new versions like Fortran-90, Fortran-95 and Fortran-2003. However each new version contains the previous versions as a subset; thus you can compile a Fortran-77 code using a compiler for Fortran-95. We will study fortran-77 first since it is still common to come across code which is written in this language.

The steps you would execute in solving some problem on the computer are:

(1) Choose a numerical method
(2) Plan the implementation of the method on paper
(3) Write a computer code
(4) Compile it to produce an executable
(5) Run it and look at the results

Of course in practice, you make mistakes which lead to *bugs* in the code. If you do any serious amount of numerical computing, you can expect to spend a considerable amount of time in finding and correcting your own mistakes; this is known as *debugging*.

Today there are several fortran compilers available; some of these are free while others cost money.

(1) g77 : This is the old Fortran-77 compiler from GNU; it is free.
(2) gfortran: This is the latest fortran compiler from GNU which supports fortran-95 also; it is free.
(3) g95: This is another free fortran-95 compiler which is mostly the work of one individual, Andy Vaught.

[1] We will use the terms *program* and *code* to refer to a computer program.

(4) ifort: This is the fortran compiler from Intel and is free for Linux and for non-commercial use. For X86-based computers, this is probably the fastest compiler since it is highly optimized for Intel processors.
(5) pgf: The Portland Group develops a fortran compiler called pgf77, pgf90 and pgf95; these are not free.

All the code examples in this tutorial will (should) compile with any fortran compiler. In these notes, we will denote the Linux terminal prompt by the dollar symbol

```
$
```

To check that the fortran compiler is installed on your computer, just type the name of the compiler at the command prompt. If you see the following message, it means that the compiler is installed

```
$ gfortran
gfortran: no input files
```

If you get the following

```
$ gfortran
gfortran: command not found
```

it probably means that the compiler is not installed. Even if it is installed, make sure that the path to the compiler is in your `PATH` variable.

2. Programming in fortran 77

Let us start with a simple program given in Fig. 1 to illustrate the structure of a fortran code. The first few lines start with the character `c` in the *first* column of the code and are considered by the compiler as *comments*. These are used to add some description and author information, and are ignored by the compiler. You can put a comment line anywhere in the program but it must start with a `c` on the first column. The actual program starts with a `program` statement[2] and ends with an `end` statement. *In fortran-77, all statements except comments start from the seventh column and end on the 72'nd column.* Columns one to five are reserved for line numbers, while column six is used to indicate line continuation when some line exceeds 72 columns. The `print` statement prints a *string* to the standard output. Type this program in your text editor and save it under the name `hello.f`; now we compile it with the following command

```
$ gfortran hello.f
```

This will create an executable program called `a.out` in the same directory; you can optionally create an executable with any name you desire, for example

```
$ gfortran -o hello hello.f
```

create an executable named `hello` in the same directory. Executing this program should print the string "Hello World !" on your screen

```
$ ./hello
 Hello World !
```

It is a good practice to write a small description of what the program is supposed to do at the beginning of the file along with some information about who wrote it and when. If you do a lot of programming, you are likely to forget after some time, what a particular piece of program was supposed to do. It is also useful for sharing your code with other people, so that they can more easily make sense of what you have written.

2.1. Example: Adding two numbers.

The next example given in Fig. 2 adds two real numbers and prints the sum to the screen. Fortran-77 has an implicit declaration of variable types; any variable whose name starts with the letters `a-h` or `o-z` is considered to be real number and all others are considered to be integers. It is however a good practice to explicitly declare every variable you use since it helps to keep track of all the variables you use and avoids mistakes. Hence the statement `implicit none` declares that all variables will be explicitly declared. If you do not add this statement, then the compiler assumes that variables are implicitly declared.

[2] This is optional.

Operator	Math symbol	Fortran operator
Equal to	$=$	`.eq.`
Less than	$<$	`.lt.`
Greater than	$>$	`.gt.`
Less than or equal to	$\leq$	`.le.`
Greater than or equal to	$\geq$	`.ge.`

TABLE 1. Arithmetic comparison operators in fortran-77

In the same example, we also see how to declare `real` variables. When you declare a variable like this, memory is allocated for the variable, but there is no value store in it. You must assign some value to the variable as shown in the example before using it in some computation. A real number is a floating point number that is stored with a finite precision on a computer. The number of decimal places that are stored depends on the computer; `real` numbers (also called *single* precision) usually have 6-7 decimal places while *double precision* numbers have 15-16 decimal places. You can declare a double precision number using the statement

```
real*8 a, b, sum
```

or equivalently

```
double precision a, b, sum
```

A double precision variable is initialized as follows

```
a = 3.141d0
```

Variable names can be made up of alphabets, numbers (for example `x1, a12`) and even the underscore character (for example `x_vel`). Even if a program is written using single precision variables, you can tell the compiler to convert all single precision variables to double precision. With gfortran, this can be achieved as follows[3]

```
$ gfortran -fdefault-real-8 -o add add.f
```

Running the program now produces the following output: note that the number of decimal places is more now.[4]

```
$ ./add
 a   =  1.00000000000000000
 b   =   2.0000000000000000
 sum =   3.0000000000000000
```

Remark: More accurately, the precision of a number type on a computer is the largest number ϵ such that the computer cannot distinguish between 1 and $1 + \epsilon$.

2.2. **Example: Comparing two numbers.** This example shows how to use comparison operators in fortran. The result of applying a comparison operator to two numbers is a *logical*, i.e., the result is either true or false. Table 1 gives the comparison operators in fortran. Run the example code in Fig. 3 and see the output for different values of input. This example also introduces the `if ... else ... endif` statement which naturally arises when you want to compare numbers.

Because of the finite precision of a computer, real numbers cannot be stored exactly. For example the number 1/3 which has an infinite decimal representation is only stored approximately in a computer. If you compute `3.0 * (1.0 / 3.0)` on a computer, the answer is not exactly 1.0; hence one should be careful in making comparisons of real numbers.

[3]With ifort or g95, the equivalent flag is `-r8`

[4]Warning: In-spite of this, do not depend on what is printed on the screen to decide the precision; how many decimal places are printed depends on the compiler and not on the actual precision with which the numbers are stored.

2.3. **Example: Sum of a sequence of numbers.** This example illustrates the use of an `array` of numbers. The code computes the sum of `n` numbers. The numbers are stored in an array of real variables called `val`; note that the memory for this array is allocated at the same time the variable is declared. In fortran-77, the size of an array is hard-coded into the program. In this example we declare a *constant* `nmax` using the `parameter` statement, which is the size of the array. The actual size `n` of the array that the user wants to add is then read by prompting the user to enter a number. It is always good practice to check that any value entered by the user is *valid* for the purpose that it will be used. For example, in the present case, if the user requests to add 1000 numbers, the program will not work because it can handle at most 100 numbers (since we have `nmax=100`). Also `n` should be a positive integer, since otherwise the sum does not make sense.

The `do ... enddo` construct is useful to perform a fixed sequence of operations. Here, it is first used to initialize the array `val` and then to compute the sum. The general structure of this loop is

```
do i = ibegin, iend, incr
   perform some operations
enddo
```

For example, the following loop

```
do i = 0, 10, 2
   perform some operations
enddo
```

is executed for `i=0,2,4,6,8,10` only. One can also loop from $n > 1$ to 1 using a negative increment as follows

```
do i = n, 1, -1
   perform some operations
enddo
```

It is also common to use the following form of the `do` loop.

```
      do 20 i = n, 1, -1
         perform some operations
20    continue
```

In this case, the line number 20 must be written in columns 1-5 only.

Remark: Note that a statement like `sum = sum + val(i)` is not a mathematical equation. What this means is that we want to take the current value of `sum`, compute the right hand side and assign the result to the variable `sum`. This situation arises very often when writing an iterative numerical scheme.

2.4. **Example: Writing to a file.** In this example given in Fig. 5, we will see how to write data to a file. We will compute the following function

$$y = \sin(2\pi x), \quad x \in [0, 1]$$

on a one dimensional *grid* and save the values to a file. Let us divide the interval $[0, 1]$ into $n-1$ parts each of length $\Delta x = 1/(n-1)$, so that there are n points

$$x_i = (i-1)\Delta x, \quad i = 1, \ldots, n \tag{1}$$

We compute the function values $y_i = y(x_i)$ at each $x_i, i = 1, \ldots, n$, and save them to file. After running this program, a new file called `sine.dat` will be created in the same directory; open this file in your text editor and check that there are two columns of numbers corresponding to the x_i and y_i.

The example uses some standard mathematical functions like `atan` which is $\tan^{-1}$ and `sin` which is just the sine function. Table 2 lists some of the standard mathematical functions available in fortran. Note that there is a different function for single and double precision arguments and you should be careful to use the correct version in your own program.

TABLE 2. Some mathematical functions in fortran

Function	Single precision	Double precision
sine	`sin`	`dsin`
cosine	`cos`	`dcos`
tangent	`tan`	`dtan`
Sine inverse	`asin`	`dasin`
Natural log	`log`	`dlog`
Log base 10	`log10`	`dlog10`

The next new concept in this example is writing to a file. Every file is assigned an integer number using which you can access the file. In the example, we assign the integer id 10 to the file `sine.dat`, then open the file, write some data into it and finally close the file. We use `write` in this case because `print*` can write to the standard output and not to a file. The asterisk in the `write` statement means that you let the compiler decide how many decimal places to print, etc. You can control the way numbers are printed using `formatting`, for example

```
write(fid,'(i5,3x,f12.4,3x,e12.4)') i, x(i), y(i)
```

The meaning of the format is as follows:

- `i5`: Print an integer in the first five columns, right justified
- `3x`: Leave three blank spaces
- `f12.4`: Print a floating point number in 12 columns with four decimal places
- `3x`: Leave three blank spaces
- `e12.4`: Print a floating point number in exponential form with four decimal places

Using a format statement has several advantages. It achieves consistency in the output across all compilers. You can print only the required number of digits for your purpose which can save memory. Also it is easier to read when you print numbers in a formatted way. The format string can also be specified in a separate statement as follows

```
      write(fid,20) i, x(i), y(i)
20    format(i5,3x,f12.4,3x,e12.4)
```

When you explicitly specify the format, make sure there is atleast one empty space between two adjacent columns. For printing floating point numbers, use the appropriate type of format `f` or `e` depending on the numerical magnitude of the numbers; if the number is very small or very large, it is better to use exponential format.

Remark: If a file of the same name already exists, then it will be overwritten by the write statement. To prevent over-writing an existing file, add the `status` identifier

```
open(unit=fid, file='sine.dat', status='new')
```

If the file `sine.dat` already exists, the program will abort. This is useful to prevent accidentally over-writing a file that you do not want to modify.

2.5. **Example: Reading from a file.** Reading data from a file is also performed similar to writing. Below is an example to read a file containing two columns of numbers, the first column being integers and the second column reals.

```
integer i, num(100)
real    x(100)
open(unit=20, file='sample.dat', status='old')
do i=1,100
   read(20,*) num(i), x(i)
enddo
close(20)
```

The `status` identifier will cause the program to check if the file exists; if it does not exist, then the program will abort with an error message. You do not have to specify formatting while reading unless you know the exact format in which the file has been written.

2.6. **Example: Finding zeros of a function using Newton method.** Consider the problem of finding the roots of a non-linear function $f : \mathbb{R} \to \mathbb{R}$ using Newton's method. In this method we make an initial guess x_o and then *iteratively* improve this guess using the formula

$$x^{i+1} = x^i - \frac{f(x^i)}{f'(x^i)} \tag{2}$$

Every numerical scheme has some or all of these ingredients:

- initial condition
- boundary condition
- stability condition
- iterative scheme
- convergence test

Fig. 6 gives an implementation of this method. The new programming concepts we see in this example are the `do while` loop and *functions*. The `do` loop which we have encountered before is useful when we know the end point of the loop, for example when summing an array of numbers. In the present case, we do not know how many iterations are required to achieve convergence. In the example, we stop the iterations when $|f(x^i)| < \epsilon$ for some small $\epsilon > 0$ for which the `do while` construct is useful. In some cases, you may never achieve convergence, but you do not want the program to continue forever. So we also impose a restriction on the maximum number of iterations, after which program will terminate even if convergence has not been achieved. It is a good practice to print a warning message in such situations.

This example uses two fortran functions, one (`fun_val`) for computing the function value and another (`fun_der`) for computing its derivative. A function takes one or more arguments and returns some value; here each function takes one double precision argument and returns a double precision value. If tomorrow you want to solve a new equation, you have to just modify the two functions while the main program remains unchanged. This modularity helps to manage your code better, improves understanding and promotes code reuse.

Remark: In the statement `x = x - f/fd`, the division is first performed and then the subtraction, which is what we want in this case. When a complex arithmetic expression is used, it is important to ensure the correct precedence of operations. For example `a*t**2` evaluates at^2 and not $(at)^2$; to evaluate the second form, use `(a*t)**2`. Whenever in doubt about the order of evaluation of an expression, use brackets to avoid the ambiguity.

2.7. **Example: Solving two dimensional Laplace equation.** In this example, we solve the following problem

$$\begin{aligned} \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} &= f(x,y), \quad (x,y) \in \Omega = (0,1) \times (0,1) \\ u &= 0 \quad \text{on} \quad \partial\Omega \end{aligned}$$

Construct a 2-D mesh using n_x partitions along x-axis and n_y partitions along y-axis, so that the grid points are $(x_i, y_j) = ((i-1)\Delta x, (j-1)\Delta y)$, $i = 1, \ldots, n_x$, $j = 1, \ldots, n_y$ and where $\Delta x = 1/(n_x - 1)$ and $\Delta y = 1/(n_y - 1)$. Using finite differences, the numerical approximation to the above PDE is given by

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2} = f_{i,j}, \quad 2 \le i \le n_x - 1, \quad 2 \le j \le n_y - 1$$

where $f_{i,j} = f(x_i, y_j)$ and $u_{i,j}$ is an approximation to the solution at (x_i, y_j). The above equation is a system of linear equations of the form $AU = F$ where A is a matrix of size $N \times N$ where $N = (n_x - 2)(n_y - 2)$. While there are many good methods to solve such equations, we will use a simple method here. Solving for $u_{i,j}$ we obtain

$$u_{i,j} = \frac{1}{2}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)^{-1} \left[\frac{u_{i-1,j} + u_{i+1,j}}{\Delta x^2} + \frac{u_{i,j-1} + u_{i,j+1}}{\Delta y^2} - f_{i,j}\right]$$

If we set some *initial* condition $u^o_{i,j}$ we can iterate using the above equation

$$u^{n+1}_{i,j} = \frac{1}{2}\left(\frac{1}{\Delta x^2} + \frac{1}{\Delta y^2}\right)^{-1}\left[\frac{u^n_{i-1,j} + u^n_{i+1,j}}{\Delta x^2} + \frac{u^n_{i,j-1} + u^n_{i,j+1}}{\Delta y^2} - f_{i,j}\right]$$

which is known as the *Gauss-Seidel* method. The iterations must be performed until the norm of the residual R is below some specified tolerance where the residual is defined as

$$R = \sum_{i=2}^{n_x}\sum_{j=2}^{n_y}\left[\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2} - f_{i,j}\right]^2$$

The above scheme is implemented in fortran-95 and shown in figures (7)-(11). For a discussion of fortran-95 language, see section (3). This example shows the use of two dimensional arrays, which is useful when solving a PDE using finite difference or finite volume method. The program is divided into many *subroutines*; a subroutine takes one or arguments, performs some computations and returns one or more results (unlike a function which returns only one value). For example, the subroutine `InitCond` takes in `nx,ny,dx,dy`, initializes the values for `f, u` which are then returned back to the calling program. For continuing a long statement on a new line, an ampersand & is used as in the subrutines `Iterate` and `Residue`. Compile this program as

```
$ gfortran -o laplace lap1.f95 lap2.f95 lap3.f95 lap4.f95 lap5.f95
```

Run this program and visualize the output files using gnuplot as explained in section (4).

Remark: Fortran is by default not case sensitive; `InitCond` and `initcond` will refer to the same subroutine. This rule applies to variable names also.

Remark: When the program is spread over many files, it can also be compiled in two steps. In the first step we create *object* files from every fortran source file

```
$ gfortran -c lap1.f95
$ gfortran -c lap2.f95
$ gfortran -c lap3.f95
$ gfortran -c lap4.f95
$ gfortran -c lap5.f95
```

You should see a `.o` file corresponding to every file you compiled above. Then in the second step we *link* all the object files to create the executable

```
$ gfortran -o laplace lap1.o lap2.o lap3.o lap4.o lap5.o
```

If you change any of the files, it is sufficient to create a new object file for that file alone and perform the linking step. For large programs it is better to automate this process using *makefile*.

2.8. **Variable types and arithmetic.** Fortran has integer, single precision and double precision variables for arithmetic operations.

```
integer p, q, r
real    x

r = p/q
x = p/q
```

The result for different cases is given in table

p	q	r = p/q	x = p/q
1	2	0	0.0
2	1	2	2.0
3	2	1	1.0

In the example

```
integer i
real    x, y
```

```
y = x * i / 100
```

the answer will depend on whether the multiplication is executed first or the division. If division is executed first, then the answer may be wrong (with `i=1`, we get `y=0`). In this case it can be written as

```
integer i
real    x, y

y = (x * i) / 100
```

which forces the multiplication to be performed first. Consider another example

```
integer i
real    x, y

y = x + i
```

This statement will be correctly executed; the compiler converts `i` to the same type as `y` and performs the computation.

Hence when using mixed variable types in a statement, you must be careful to check that you are getting the desired type of computation. When in doubt, you can convert an integer to a float using the `real` and `dble` function.

2.9. **Common variables.** Fortran has a feature to use global variables through a common block declaration. This is useful for storing values of constants like π which may be required in different parts of a large program and also for storing array sizes. Many programmers put all variables in a common block to avoid having to pass them to subroutines. This is however bad programming practice since it does not have a clear distinction between data and procedures.

In Fig. (12) two common blocks are defined, one to store π and another to store some dimensions. These variables are initialized in the main program and they can be accessed in a subroutine.

3. Programming in Fortran 95

Fortran 95 is a relatively new fortran standard that has several useful features. The syntax from fortran-77 is still valid in a fortran-95 program.

- The restriction of columns is removed; you can begin and end the program in any column.
- Memory for arrays can be dynamically allocated. This is a very useful feature since you do not have to hard-code array sizes in the program which would have to be changed every time the problem size changes. Remember to deallocate the memory once the variable is no longer requires. Otherwise, your program can consume all the memory in the computer.

  ```
  integer :: n
  real, allocatable :: x(:)
  read*,n
  allocate(x(n))
  .
  .
  .
  deallocate(x)
  ```

- Vector operations are possible. To assign zero to all elements of an array `x(:)`, just use

  ```
  x = 0.0
  ```

  or

  ```
  x(:) = 0.0
  ```

  You can also use only a part of the array, for example

  ```
  x(10:20) = 0.0
  ```

will set the value zero to the array elements x(10) to x(20) only. Arrays can also be added or subtracted

```
z(:) = x(:) + y(:)
```

Of course, this is possible only if all three arrays `x, y, z` have the same size. Scalar multiplication is also possible,

```
y(:) = c * x(:)
```

where `c` is a number. The above rules also extend to multi-dimensional arrays in an obvious way.

- Pointers are also available in fortran-95 though in most cases, allocatable arrays are sufficient.
- Derived data types are also possible, similar to structures in C. For example a three dimensional vector type can be defined as

```
type v3d
  real :: x, y, z
end type v3d
```

You can then define variables of type `v3d` as follows, and access the members of the type

```
Type(v3d) :: vec

vec%x = 0.1
vec%y = 0.2
vec%z = 0.3

norm = sqrt( vec%x**2 + vec%y**2 + vec%z**2 )
```

4. Visualization using `gnuplot`

Gnuplot is a free visualization tool for 1-, 2- and 3-D functions developed by the Free Software Foundation. For more information on gnuplot, go to the following website

`http://www.gnuplot.info`

Another excellent resource for gnuplot related information is the following website

`http://t16web.lanl.gov/Kawano/gnuplot/index-e.html`

To start gnuplot, just type its name in the terminal; you will get the command prompt `gnuplot>` which is where you type all the commands. There is a lot of online help available within gnuplot using the `help` command; for example type `help plot` at the gnuplot prompt and read the manual.

4.1. **Making xy plots or 1-D plots.**

Using the program `sine.f` or any other program of your choice, generate a file containing two columns of numbers $(x_i, f(x_i))$. The function f can be visualized in gnuplot using the following commands.

4.1.1. *Plotting single curve.*

Plotting a single curve from the file `sine.dat`:

- To plot using symbols

```
gnuplot> plot 'sine.dat' w p
```

- To plot using lines

```
gnuplot> plot 'sine.dat' w l
```

- To plot using both lines and symbols

```
gnuplot> plot 'sine.dat' w lp
```

Most gnuplot commands have a short form; thus `plot` can be replaced with just `p` as we do in the rest of the tutorial.

4.1.2. *Plotting two curves.* Plotting two curves on the same graph: Prepare a file `cossin.dat` with three columns of data $(x_i, \cos(x_i), \sin(x_i))$

- To plot using different symbols

```
gnuplot> p 'cossin.dat' u 1:2 w p pt 1, 'cossin.dat' u 1:3 w p pt 2
```

- To plot using different lines

```
gnuplot> p 'cossin.dat' u 1:2 w l lt 1, 'cossin.dat' u 1:3 w l lt 2
```

- To plot using both lines and symbols

```
gnuplot> p 'cossin.dat' u 1:2 w lp lt 1 pt 1, \
'cossin.dat' u 1:3 w lp lt 2 pt 2
```

4.1.3. *Using gnuplot from a script file.* Instead of typing these commands every time, we can save them in a file and use it repeatedly. Save the following in a file called `plt.gnu`

```
1 set xlabel 'x'
2 set ylabel 'f(x)'
3 set title 'Plot of x versus cos(x) and sin(x)'
4 p 'cossin.dat' u 1:2 t 'cos(x)' w lp lt 1 pt 3, \
5 'cossin.dat' u 1:3 t 'sin(x)' w lp lt 2 pt 6
```

and inside gnuplot, do

```
gnuplot> load 'plt.gnu'
```

All the commands in the file will be executed and you should see two curves with different symbols and line style. A legend will also appear for the two curves.

4.1.4. *Logarithmic plots.* Using the example program from section (2.7), generate the `res.dat` file which contains two columns, iteration number and residual error. The error value ranges from an $O(1)$ quantity to a very small quantity; it is then better to plot the error in log scale. An example is given in the `res.gnu` script

```
1 set nokey
2 set logscale y
3 set xlabel 'Number of iterations'
4 set ylabel 'Residual error'
5 set title 'Conv of residual error for soln of Poisson equation'
6 p 'res.dat' u 1:2 w l lw 2
```

4.1.5. *Producing postscript file.* Save the following gnuplot commands in a file `ps.gnu`

```
1 set term postscript enhanced
2 set out 'cossin.eps'
3 set xlabel 'x'
4 set ylabel 'f(x)'
5 set title 'Plot of x versus cos(x) and sin(x)'
6 p 'cossin.dat' u 1:2 t 'cos(x)' w lp lt 1 lw 2 pt 3, \
7 'cossin.dat' u 1:3 t 'sin(x)' w lp lt 2 lw 2 pt 6
```

and execute it from the terminal (not from inside gnuplot)

```
$ gnuplot ps.gnu
```

This should generate a postscript file called `cossin.eps` which you can open in a postscript viewer like `gv` or `ggv` or `kghostview`. To produce colored plots, use

```
set term postscript enhanced color
```

Gnuplot also supports saving the plots in other formats like pdf, jpg, png, gif, etc. To see the supported formats, type `help term` at the gnuplot command prompt.

4.2. **Making 2-D plots.** Two dimensional plots may be surface plots or contour plots. The data for a 2-D plot consists of a function defined on a 2-D grid of points of size $n_x \times n_y$. The data must be prepared using the following format:

```
do i=1,nx
   do j=1,ny
      write(fid,*) x(i,j), y(i,j), u(i,j)
   enddo
   write(fid,*)
enddo
```

If you execute the example code from section (2.7), it generates a file `u.dat` which is in the above format. Then use the following gnuplot script to plot the solution as a surface and contour.

```
set size square
set nokey
set contour
set surface
set cntrparam levels 25
splot 'u.dat' u 1:2:3 w l lt 1
```

You can use the left mouse button to rotate the figure and the middle mouse button to zoom in and out. To plot using only contours, use the following script

```
set size square
set nokey
set contour
set nosurface
set view 0,360
set cntrparam levels 25
#set cntrparam levels incremental 0, 0.1, 1.0
set term table
set out 'table.dat'
splot 'u.dat' u 1:2:3 w l
set term x11
set xlabel 'x'
set ylabel 'y'
p 'table.dat' w l
```

Note that number of contour levels to plot is given by the `set cntrparam` statement; in the example, 25 equally spaced values between and minimum and maximum are used. To specify the levels explictly, you can use

```
set cntrparam levels discrete 0, 0.1, 0.2, 0.3, 0.4, 0.5
```

or

```
set cntrparam levels incremental 0, 0.01, 1
```

5. Some advice on good programming habits

There can be quite a lot of work involved in taking a numerical scheme from paper to a working code. Mistakes invariably occur and can be quite difficult to detect. However they can be considerably minimized by following some programming habits.

- Plan your program as much as possible on paper.
- Write the code in a clean and organized way. Use proper indentation.
- Give natural names to variables
- Write comments to explain what is happening in each part of the program.
- Encapsulate common procedures in a function or subroutine.
- Read parameters from an external file instead of modifying the program every time you change the value of some parameter.

- Always check the validity of every variable if you think that it may be wrong. For example if some variable must always be positive, check that it is actually so before using it.
- Print important variables to screen so that you can verify everything is initialized correctly.
- When you are using arrays, make sure you do not exceed the array bounds. For example if you have an array of size 100, say `val(100)`, then you should not try to use the value `val(101)` since it is out of bounds. Using an array out of its bounds leads to *segmentation fault*.
- Test the program on a problem whose solution is known to you. This can reveal mistakes you might have committed.
- Do not use `goto` statements. This makes the code difficult to understand and debug in case of mistakes.
- Beware of using uninitialized variables. Many compiler have options to check this. For example, with gfortran, use the flag `-Wall` which causes the compiler to do many checks and issue warnings.

6. Exercises

You can do the following exercises in any of the languages: fortran-77, fortran-95 or C. However I recommend fortran-95 or C. All programs must contain a header comment giving information about your name, date of creation of the program and a brief description of the program. There must be comments describing the action taking place in different parts of the program. Weightage will be given if you follow the good programming habits discussed above.

6.1. **Machine precision.** Find the smallest positive number n such that $1+1/2^n == 1$. Write two programs, one in single precision and the other in double precision.

6.2. **Roots of a quadratic.** Write a program to find the roots of a quadratic equation $ax^2 + bx + c = 0$. The program must read in the values of a, b, c and print the two roots to the screen. It must also be able to check and compute complex roots.

6.3. **Euler method.** Write a fortran program to solve an ODE using Euler method.

$$\begin{aligned} \frac{\mathrm{d}u}{\mathrm{d}t} &= f(u,t), \quad t > t_o \\ u(t_o) &= u_o \end{aligned}$$

The quantities t_o, u_o, t_f, N must be read from a file. The positive integer N is the number of time steps taken to go from t_o to t_f, so that the time-step interval is $\Delta t = (t_f - t_o)/(N-1)$. The function $f(u,t)$ must be computed by a fortran function. The Euler method is given by

$$u^{n+1} = u^n + \Delta t f(u^n, t^n), \quad t^n = t_0 + n\Delta t, \quad n = 0, 1, 2, \ldots \tag{3}$$

As a specific example, solve the ODE

$$\begin{aligned} \frac{\mathrm{d}u}{\mathrm{d}t} &= \cos t, \quad t > 0 \\ u(0) &= 0 \end{aligned}$$

until the final time $t_f = 2\pi$. The exact solution is $u(t) = \sin t$; plot the numerical solution and exact solution using gnuplot. Do this for $N = 100, 200, 300, 400, 500$. The program must ask the user to enter the values of N and t_f.

6.4. **Runge-Kutta method.** Consider the ODE problem from (6.3). The m'th order Runge-Kutta method is given by

$$\begin{aligned} u^{(0)} &= u^n \\ u^{(s+1)} &= u^n + \frac{\Delta t}{m-s} f(u^{(s)}, t^n + \Delta t/(m-s)), \quad s = 0, \ldots, m-1 \\ u^{n+1} &= u^{(m)} \end{aligned}$$

For $m = 1$ this becomes the Euler method. Write a program to solve the ODE using Runge-Kutta method of any order m. The program must ask the user for N, t_f, m. Compare the solution for $N = 100$ and $m = 1, 2, 3, 4$.

6.5. **Finite volume method.** Write a fortran program to solve the following conservation law

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{\partial}{\partial x} f(u) &= 0, \quad x \in (a,b), \quad 0 < t \leq t_f \\ u(x,0) &= u_o(x), \quad x \in [a,b] \end{aligned}$$

Divide the computational domain $[a, b]$ into N equal partitions of length $\Delta x = (b-a)/(N-1)$

$$a = x_{1/2} < x_{2/2} < x_{3/2} \ldots < x_{N-1/2} < x_{N+1/2} = b$$

The finite volume scheme is obtained by integrating the PDE over each cell; for the i'th cell $C_i = (x_{i-1/2}, x_{i+1/2})$ we get

$$\frac{\mathrm{d}u_i}{\mathrm{d}t} + \frac{F_{i+1/2} - F_{i-1/2}}{\Delta x} = 0$$

where $F_{i+1/2} = F(u_i, u_{i+1})$ is a numerical flux function. This gives us a system of ODE

$$\frac{\mathrm{d}u_i}{\mathrm{d}t} = R_i(u) := -\frac{F_{i+1/2} - F_{i-1/2}}{\Delta x}, \quad i = 2, \ldots N-1$$

We keep the solution in cell 1 and N fixed since otherwise some boundary condition will be required. The residual computation requires the fluxes $F_{3/2}$ to $F_{N-1/2}$. The finite volume residual must be computed without duplicating the flux computation. Use the flux function

$$F(u_i, u_{i+1}) = \frac{1}{2}(f_i + f_{i+1}) - \frac{1}{2}|a_{i+1/2}|(u_{i+1} - u_i)$$

where

$$a_{i+1/2} = \begin{cases} \frac{f_{i+1} - f_i}{u_{i+1} - u_i} & u_i \neq u_{i+1} \\ f'(u_i) & u_i = u_{i+1} \end{cases}$$

The time-step Δt must satisfy the stability condition

$$\frac{\Delta t \max_u |f'(u)|}{\Delta x} \leq 1$$

Take the following problem: $a = 0$, $b = 2$, $f(u) = u^2/2$,

$$u_o(x) = \begin{cases} \sin^2(2\pi(x - 1/4)) & 0.25 \leq x \leq 0.75 \\ 0 & \text{otherwise} \end{cases}$$

The program must have subroutines to:

- read $a, b, \Delta t, N, t_f$ from a file
- check that Δt satisfies stability condition
- set initial condition
- compute finite volume residual
- compute flux function
- save final solution to a file

6.6. **Finite difference method: Solution of Poisson equation.** Rewrite the program to solve the Poisson equation in an arbitrary rectangle $(x_m, x_M) \times (y_m, y_M)$ in two dimensions.

(1) The program must read $n_x, n_y, x_m, x_M, y_m, y_M$, MAXITER, TOL from an input file.
(2) Make separate subroutines to set the boundary condition, initial condition and the right hand side f.
(3) Plot the solution using surface and contours; put labels and title.

References

[1] S. Lipschultz and A. Poe, *Theory and problems of programming with Fortran*, McGraw-Hill Book Company.
[2] L. Nyhoff and S. Leestma, *Introduction to Fortran 90 for engineers and scientists*, Prentice Hall, 1997.
[3] S. J. Chapman, *Fortran 77 for engineers and scientists with an introduction to Fortran 90*, Harper Collins, 1995.
[4] W. H. Press et al., *Numerical recipes in Fortran: The art of scientific computing*, Cambridge University Press, 1992.
[5] M. Metcalf, J. Reid and M. Cohen, *Fortran 95/2003 Explained*, Oxford University Press, 2004.

```
c Canonical fortran program
c Author: Praveen. C <praveen@cfdlab.net>
c Date  : 31 August, 2008
c Desc  : Prints "hello world" on screen
      program hello

      print*,'Hello World !'

      end program hello
```

FIGURE 1. First fortran 77 program: `hello.f`

```
      program add
      implicit none
      real a, b, sum

      a = 1.0
      b = 2.0

      sum = a + b

      print*,'a   =', a
      print*,'b   =', b
      print*,'sum =', sum

      end
```

FIGURE 2. Adding two real numbers: `add.f`

```
      program compare
      implicit none
      real a, b
      print*,'Enter first number (a)'
      read*,a
      print*,'Enter second number (b)'
      read*,b
      if(a.lt.b)then
         print*,'a is less than b'
      else if(a.gt.b)then
         print*,'a is greater than b'
      else
         print*,'a is equal to b'
      endif
      end
```

FIGURE 3. Arithmetic comparison operators: `compare.f`

```
      program array_sum
      implicit none
      integer i, nmax, n
      parameter(nmax=100)
      real    val(nmax), sum

c Read how many number to add
      print*,'How many numbers to add ?'
      read*,n

c Check that n is a valid input
      if(n.gt.nmax)then
         print*,'Array size nmax is not sufficient'
         print*,'Increase nmax and then run'
         stop
      endif

      if(n.lt.0)then
         print*,'Error: give a positive integer'
         stop
      endif

c Initialize the numbers to add
      do i=1,n
         val(i) = i
      enddo

c Add the numbers
      sum = 0.0
      do i=1,n
         sum = sum + val(i)
      enddo

      print*,'sum =', sum

      end program array_sum
```

FIGURE 4. Adding a finite sequence of numbers: `array_sum.f`

```
1       program sine
2       implicit none
3       integer i, n, fid
4       parameter(n=101)
5       real x(n), y(n), dx, PI
6
7       PI = 4.0*atan(1.0)
8
9       dx = 1.0/(n-1)
10      do i=1,n
11         x(i) = (i-1)*dx
12         y(i) = sin(2.0*PI*x(i))
13      enddo
14
15      fid = 10
16      open(unit=fid, file='sine.dat')
17      do i=1,n
18         write(fid,*) i, x(i), y(i)
19      enddo
20      close(fid)
21
22      end
```

FIGURE 5. Writing to a file: `sine.f`

```
      program newton
      implicit none
      integer          iter, maxiter
      double precision EPSILON
      parameter(EPSILON=1.0d-10, maxiter=100)
      double precision x, f, fd
      double precision fun_val, fun_der

      iter = 0
      x    = 3.0d0
      f    = fun_val(x)
      fd   = fun_der(x)
      do while(dabs(f).gt.EPSILON .and. iter.lt.maxiter)
         x    = x - f/fd
         f    = fun_val(x)
         fd   = fun_der(x)
         iter = iter + 1
         write(*,10) iter, x, f
10       format("iter =",i5,3x," x =",e20.12,3x," f =",e15.4)
      enddo

      end

      double precision function fun_val(x)
      implicit none
      double precision x

      fun_val = x*x - 2.0d0

      end

      double precision function fun_der(x)
      implicit none
      double precision x

      fun_der = 2.0d0*x

      end
```

FIGURE 6. Newton-Raphson method: `newton.f`

```
program laplace
   implicit none
   real, allocatable :: u(:,:), f(:,:)
   integer :: nx, ny, iter, MAXITER,fid
   real :: dx, dy, res, TOL
   parameter(MAXITER=1000, TOL=1.0e-7)

   print*,'Enter grid size nx, ny'
   read*,nx,ny
   allocate( u(nx,ny) )
   allocate( f(nx,ny) )
   dx = 1.0/(nx-1)
   dy = 1.0/(ny-1)
   call InitCond(nx,ny,dx,dy,f,u)
   res  = TOL+1.0 ! Residue to measure convergence
   iter = 0       ! Iteration counter
   fid  = 15      ! File to save residue
   open(unit=fid, file='res.dat')
   do while(res > TOL .and. iter < MAXITER)
      call Iterate(nx, ny, dx, dy, f, u)
      call Residue(nx, ny, dx, dy, f, u, res)
      iter = iter + 1
      write(*,'(i6,e12.4)') iter,res
      write(fid,'(i6,e12.4)') iter,res
   enddo
   close(fid)
   call SaveResult(nx,ny,dx,dy,u)
   deallocate(u)
   deallocate(f)

end
```

FIGURE 7. Solution of Poisson equation: `lap1.f95`

```
subroutine InitCond(nx,ny,dx,dy,f,u)
   implicit none
   integer :: nx, ny
   real    :: dx, dy, f(nx,ny), u(nx,ny)

   integer :: i, j
   real    :: x, y, PI

   PI = 4.0*atan(1.0)

   do i=1,nx
      do j=1,ny
         x      = (i-1)*dx
         y      = (j-1)*dy
         u(i,j) = 0.0
         f(i,j) = sin(2.0*PI*x) * cos(2.0*PI*y)
      enddo
   enddo

end
```

FIGURE 8. Solution of Poisson equation: `lap2.f95`

```
subroutine Iterate(nx, ny, dx, dy, f, u)
   implicit none
   integer :: nx, ny
   real    :: dx, dy, f(nx,ny), u(nx,ny)

   integer :: i, j
   real    :: fact

   fact = 0.5/( 1.0/dx**2 + 1.0/dy**2 )

   do i=2,nx-1
      do j=2,ny-1
         u(i,j) = fact*( (u(i-1,j  ) + u(i+1,j  ))/dx**2 + &
                         (u(i  ,j-1) + u(i  ,j+1))/dy**2 - &
                         f(i,j) )
      enddo
   enddo
end
```

FIGURE 9. Solution of Poisson equation: `lap3.f95`

```
subroutine Residue(nx, ny, dx, dy, f, u, res)
   implicit none
   integer :: nx, ny
   real    :: dx, dy, f(nx,ny), u(nx,ny), res

   integer :: i, j

   res = 0.0
   do i=2,nx-1
      do j=2,ny-1
         res = res &
             + ( (u(i-1,j  ) - 2.0*u(i,j) + u(i+1,j  ))/dx**2 &
             + (u(i  ,j-1) - 2.0*u(i,j) + u(i  ,j+1))/dy**2 &
             - f(i,j) )**2
      enddo
   enddo
   res = res/(nx-2)/(ny-2)
end
```

FIGURE 10. Solution of Poisson equation: `lap4.f95`

```
subroutine SaveResult(nx, ny, dx, dy, u)
   implicit none
   integer :: nx, ny
   real    :: dx, dy, u(nx,ny)

   integer :: fid, i, j
   real    :: x, y

   fid = 10
   open(unit=fid, file='u.dat')
   do i=1,nx
      do j=1,ny
         x = (i-1)*dx
         y = (j-1)*dy
         write(fid,*) x, y, u(i,j)
      enddo
      write(fid,*)
   enddo
   close(fid)
end
```

FIGURE 11. Solution of Poisson equation: `lap5.f95`

```
      program main
      implicit none
      integer nmax, smax
      real    pi
      common/math/pi
      common/dim/nmax, smax

      pi   = 4.0*atan(1.0)
      nmax = 100
      smax = 200

      call sub()
      end

      subroutine sub()
      implicit none
      integer nmax, smax
      real    pi
      common/math/pi
      common/dim/nmax, smax

      print*,'pi   =',pi
      print*,'nmax = ',nmax
      print*,'smax = ',smax
      end
```

FIGURE 12. Use of common variables: `common.f`

```
      integer nmax, smax
      real    pi
      common/math/pi
      common/dim/nmax, smax
```

FIGURE 13. Include file used in Fig. 14: `common.h`

```
1       program main
2       implicit none
3       include 'common.h'
4
5       pi    = 4.0*atan(1.0)
6       nmax = 100
7       smax = 200
8
9       call sub()
10      end
11
12      subroutine sub()
13      implicit none
14      include 'common.h'
15
16      print*,'pi    =',pi
17      print*,'nmax = ',nmax
18      print*,'smax = ',smax
19      end
```

FIGURE 14. Program of Fig. 12 written with include file: `common2.f`