FORTRAN90 CFD Notes

By Dr Michal Kopera

Contents

Lesson 1: Introduction to FORTRAN90 Programming Language

1-1 Introduction

The goal of this session is to familiarise you with the most basic concepts of programming in FORTRAN language. Although it was historically first language used in scientific applications, it's more modern versions (like the one you are going to learn today) are still widely used in Computational Fluid Dynamics (CFD). In frame of the CFD module you will be asked to do an assignment that will involve some programming skills, which you will gradually learn during this lab.

Today we are going to focus on following elements:

• Compiling and running simple code

• Inputting data from the keyboard and printing data on the screen

• Doing simple arithmetic using variables

• Input/output operations using files

1-2 Compiling and Running a Code

In any programming task you first need to write a code – a list of commands that define what you want a computer to do. You use a programming language for it – we are going to use FORTRAN. The programming language is a set of commands and rules that can be translated to a language understandable by a computer. To do the translation from a human-readable code to a machine-readable program we use a tool called a compiler. We are going to use g95

compiler that will translate our Fortran90 files (with an extension .f90) to executable files (.exe). First of all let's have a look on a simplest possible code:

```
PROGRAM HELLO
IMPLICIT NONE
! write out "Hello World" to your screen
PRINT *, "Hello World!"
END PROGRAM HELLO
```

First thing to notice is that exclamation mark at the beginning of the line says that this line is only a comment and therefore will be ignored by a compiler and not executed while running a code. Secondly, the compiler does not pay attention to capital letters, but it is a good programming practice to write the commands in uppercase and variables, comments and others in lowercase. Each f90 program should start with PROGRAM <name> command. In our case the program is called HELLO. Then we usually put a command IMPLICIT NONE which will not be discussed here – let assume that it always should be just below the PROGRAM command. After IMPLICIT NONE starts the main body of the program. Here it only consists of one command, which will print some text (included in apostrophes) on your screen.

Now let us make this code running. First create a directory called FORTRAN in your My documents directory on drive I. It will be your working directory. Then open the Notepad application. You may do this by selecting Accessories > Notepad from Start menu. Then copy the code above into Notepad and save it as a file called hello.f90 in your working directory. Make sure that the file has extension .f90, since it will be very important for compiler. If you end up with a file called something like hello.f90.txt (check that in your directory!) rename the file to hello.f90.

So you have a code ready for compiling. On your desktop there should be a shortcut called Fortran G95. Double click on it. A window with a command prompt should appear. It will look like this:

```
Microsoft Windows XP [Version 5.1.2600]
(C)Copyright 1985-2001 Microsoft Corp.

I:\>
```

Now type cd "My documents"\Fortran and press enter to change the directory to My documents\Fortran. To compile your code type g95 hello.f90.

```
I:\>cd "My documents"\Fortran
I:\My documents\Fortran>g95 hello.f90
```

After a while take a look into your FORTRAN directory. A file called a.exe should appear there. This is your executable file. You can now type a.exe in the command prompt. The result should look as follows:

```
I:\My documents\Fortran>a.exe
Hello world!

I:\My documents\Fortran>
```

a.exe is standard name for compiled code. If you would like to name it otherwise, you should compile your code using –o <file_name> option:a

```
I:\My documents\Fortran>g95 hello.f90 –o hello.exe
```

Congratulations! You have just compiled and run your first FORTRAN code. All it did was to print a greeting on your screen. Now take a while and change the code that you will get the program to print some other text, like "Hello Mike! Welcome to the CFD lab". Save changes, compile and run your modified code to see if you get the result you want.

1-3 Simple Arithmetics

Now is the time to learn how make basic calculations. After all, this is the main thing we want the computers do for us in CFD. Fortran uses five basic arithmetic operators:

Addition	+
Subtraction	-
Multiplication	*
Division	/
Exponentiation	**

Another important concept is variables. Before you can use a variable you have to declare its type. For now we are going to use only *integer* and *real* variables, but other types are possible in FORTRAN. Example below will explain how to declare variables and do some basic arithmetic. We declare variables x and f. f is a quadratic function of x:

$$f(x) = x^2$$

The code will calculate the value of a function and print it on the screen.

```
PROGRAM FINITE_DIFF
IMPLICIT NONE
!declaration of real variables x and f
REAL::x,f
!now let's assign a value to variable x
x = 1.0
!value of f at x
f = x**2
!print the value of x and f on the screen
PRINT *, 'x = ',x
PRINT *, 'f(x) = ',f
END PROGRAM FINITE_DIFF
```

Now copy the code above to some .f90 file, compile using g95 and run it. You should get following result:

```
I:\My documents\Fortran>a.exe
x = 1.
f(x) = 1.
I:\My documents\Fortran>
```

Now it's your turn. Let's calculate the derivative of f analytically:

$$\frac{\partial f}{\partial x} = 2x$$

Add proper lines to the code below so that it will calculate and print the value of derivative of f at point x. You will need to declare a variable for the derivative, called, say df. Then you need to add a line where you calculate the derivative analytically and finally a line to print it on the screen.

```
PROGRAM FINITE_DIFF
IMPLICIT NONE
!declaration of real variables x and f
REAL::x,f
!here put the line to declare variable df
!now let's assign a value to variable x
x = 1.0
!value of f at x
f = x**2
!here put the line to calculate df
!print the value of x and f on the screen
PRINT *, 'x = ',x
PRINT *, 'f(x) = ',f
!here put the line to print the value of df on the screen
END PROGRAM FINITE_DIFF
```

After you fill the proper lines compile, run and check results. Do not proceed to the next page before you are done with this task!

1-4 Inputting Data from Keyboard

Now let's make our program a bit more interactive. Instead of assigning the value of x in the code, we will make it to ask for a value of x and then calculate the value of function f and its

analytical derivative.

```
PROGRAM FINITE_DIFF
IMPLICIT NONE
!declaration of real variables x and f
REAL::x,f
REAL::df
!ask user to input a value
PRINT *, 'Enter a value for x: '
!assign typed value to the variable x
READ(*,*) x
!value of f at x
f = x**2
!value of derivative of f at x
df = 2*x
!print the value of x and f on the screen
PRINT *, 'x = ',x
PRINT *, 'f(x) = ',f
PRINT *, 'df(x) = ',df
END PROGRAM FINITE_DIFF
```

As usual compile, run and check results. You can try to play around for a while and check different values for x.

1-5 Input/Output Operations using Files

Usually we do not want to ask user for every variable we want to feed to our program. When there is a lot of data we want to process, we store it in files. On the next sessions we are going to learn how to read multiple lines of data, but today let's just focus on reading simple number from a file. First create a file called input.dat. Open it and write there some number (you may do it using Notepad). We will read that number in the code as a variable x, and then write the values of function f and its derivative to another file.

```
PROGRAM FINITE_DIFF
IMPLICIT NONE
!declaration of real variables x and f
REAL::x,f
REAL::df
!open a file called input.dat and assign a 111 tag to it
OPEN(unit=111,file='input.dat',status='unknown')
```

```
!assign typed value to the variable x
READ(111,*) x
!close the file after use
CLOSE(111)
!value of f at x
f = x**2
!value of derivative of f at x
df = 2*x
!print the value of x and f on the screen
PRINT *, 'x = ',x
PRINT *, 'f(x) = ',f
PRINT *, 'df(x) = ',df
!open a file to write to
OPEN(unit=222,file='output.dat',status='unknown')
!write the results to a file
WRITE(222,*)x,f,df
!close the file after use
CLOSE(222)
END PROGRAM FINITE_DIFF
```

To use a file we must first open it. We use command OPEN to do this. This command assigns a number tag (unit) to a file (file='input.dat'). Let's do not discuss the status option just now. You may google for its meaning if you really want to.

To read the contents of a file we use command READ. We need to specify the unit number from which we want to read from and to which variable we want to assign read value.

Similarly to reading files, we can write to them using command WRITE. Again we need to supply a unit number and the list of variables to be written. Do not forget to open a file that you want your data to be saved to! I have chosen the name to be output.dat.

It is a good practice to always close the file after use with the command CLOSE. Now as you have read and understood what is happening in the code, you can compile and run it. You should see that a file called output.dat appeared in your working directory. Check its contents.

1-6 Exercise

It's your turn again. Change the code so that instead of quadratic function it will calculate the value and derivative of sin function. In FORTRAN you can calculate the value of sin or cos simply by typing f = sin(x) or f = cos(x). You can google for other intrinsic functions or you can find some examples there:

http://www.cs.mtu.edu/~shene/COURSES/cs201/NOTES/chap02/funct.html

Lesson 2: Introduction to FORTRAN90 Programming

2-1 Previously on CFD lab...

As you remember from the previous chapter we have learned:

1- How to compile and run a code

2- How to input data from keyboard

3- How to print data on the screen

4- How to make computer do some simple arithmetics

5- How to read and write data from/to a file

Before we head on to this week's material, let's do some exercise to recap some of most important features from last week. On the lecture you should hear about Finite Difference Schemes. They are numerical schemes for calculating approximate values of derivatives. Let's assume we have a function $f(x) = x^2$ and we want to calculate its derivative at a point $x_0 = 1$ using forward differencing scheme.

$$\left.\frac{df}{dx}\right|_{x=x_0} \approx \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x}$$

We need to choose our Δx first. The code on the next page calculates the value of the derivative of quadratic function at a given point with a given Δx. Read it and examine the changes that occurred since last week. Compile and run it to see how it works for different values of Δx. The code gives also the exact value of derivative for comparison.

```
PROGRAM FINITE_DIFF
IMPLICIT NONE
!declaration of variables
!fx0 is the value of a function at a point x0
!fx1 is the value of a function at a point x0+dx
!df is the exact value of the derivative
!dfc is the value of the calculated derivative
REAL::x0,fx0,fx1,dx
REAL::df,dfc
PRINT*,'Please input the value of x0 = '
READ(*,*) x0
PRINT*,'Please input the value of dx = '
READ(*,*) dx
!value of f at x0
fx0 = x0**2
!exact value of derivative of f at x0
df = 2*x0
!value of f at x0+dx
fx1 = (x0+dx)**2
!calculation of the derivative using forward finite difference
dfc = (fx1-fx0)/dx
!print the results on the screen
PRINT *, 'Derivative of f = x*2 at x0 = ',x0, 'with dx = ',dx
PRINT *, 'Calculated value = ',dfc
PRINT *, 'Exact value = ',df
END PROGRAM FINITE_DIFF
```

When you are done, save your code to a new file. Can you remember from the lecture what the Central Finite Difference was about? To help you recall, look at the formula below:

$$\left.\frac{df}{dx}\right|_{x=x_0} \approx \frac{f(x_0+\Delta x)-f(x_0)}{\Delta x}$$

Can you implement this formula to the code? Try to do it and compare the results of Forward and Central Finite Difference scheme. Which one is more accurate? Why?

2-2 Session Plan

Today we are going to learn about following things:

1- Conditional statements

2- Loops

3- Multiple line input and output from/to a file

2-3 Conditional Statements

The conditional statements are used to control the flow of your code. Basic conditional statement in FORTRAN90 is the `if` statement. It allows you to define what the computer should do if certain condition is fulfilled and what should happen otherwise. The syntax of an if statement is as follows:

```
IF (condition is true) THEN
Execute this line
Execute this line
Etc.
ELSE
Execute this line
Execute this line
Etc.
ENDIF
```

First we should know something about how to formulate a condition. We need for that some comparison and logical operators:

```
a.gt.b - is true when a is greater than b
a.ge.b - is true when a is greater or equal than b
a.lt.b - is true when a is less than b
a.ge.b - is true when a is less or equal than b
a.eq.b - is true when a is equal to b
a.ne.b - is true when a is not equal to
b.not.a - is true when a is false
a.and.b - is true when a and b are both true
a.or.b - is true when either a or b are true
```

Let's see how it works on an example. We will follow from the code for Forward Finite Difference. We are going to apply if statement to judge whether our is sufficiently Δx small to calculate the derivative accurately.

```
PROGRAM FINITE_DIFF
IMPLICIT NONE
!declaration of variables
!fx0 is the value of a function at a point x0
```

```
!fx1 is the value of a function at a point x0+dx
!df is the exact value of the derivative
!dfc is the value of the calculated derivative
REAL::x0,fx0,fx1,dx
REAL::df,dfc
REAL::err
PRINT*,'Please input the value of x0 = '
READ(*,*) x0
PRINT*,'Please input the value of dx = '
READ(*,*) dx
!value of f at x0
fx0 = x0**2
!exact value of derivative of f at x0
df = 2*x0
!value of f at x0+dx
fx1 = (x0+dx)**2
!calculation of the derivative using central finite difference
dfc = (fx1-fx0)/(dx)
!calculate the error
err = abs((df-dfc)/df)
!print the results on the screen
PRINT *, 'Derivative of f = x*2 at x0 = ',x0, 'with dx = ',dx
PRINT *, 'Calculated value = ',dfc
PRINT *, 'Exact value = ',df
IF(err.le.0.001) THEN
PRINT *,'dx is small enough, results accepted'
ELSE
PRINT *,'dx is too big, results rejected'
ENDIF
END PROGRAM FINITE_DIFF
```

Compile and run the code. Try with different input values and try to get both possible answers on screen.

It is possible to make multiple mutually excluding if statements in one go. We use the ELSE IF command for that. The syntax will look like this:

```
IF (condition is true) THEN
Execute this line
Execute this line
Etc.
ELSE IF (another condition is true) THEN
Execute this line
Execute this line
Etc.
ELSE IF (yet another condition is true) THEN
Execute this line
Execute this line
Etc.
...
```

```
ELSE
Execute this line
Execute this line
Etc.
ENDIF
```

We want our code to judge whether our Δx is sufficiently small, too big, or maybe it is so small that we can allow it to be a bit bigger (generally in CFD we want our dx to be small enough to make error small, but at the same time dx should be largest possible since it is strongly connected to the computational effort our machine needs to make in order to get the result). Your task will be to apply following conditions using IF, ELSE IF and ELSE construct:

```
if error < 1e-06 - dx can be increased
if error > 1e-06 and error < 1e-03 – dx is just right
if error > 1e-03 – dx is too big
```

Apply the conditions above and try several input values to see what happens.

2-4 Loops

Another important concept in programming is a loop. It allows to repeat a group of commands given number of times. Let's see how it works on a simple example:

```
PROGRAM LOOPY
IMPLICIT NONE
INTEGER:: i
DO i=1,10
PRINT*,'Hello World!, count = ',i
END DO
END PROGRAM LOOPY
```

Compile and run the example to see what happens. Try to change the number of repetitions.

Now for something more complicated. We want our program to read couple of values of dx from a file, calculate the derivative for given dx and write the corresponding error to another

file. First we need to generate a file called input.dat. In the first line it should have the number of different dx we want to use. Following lines would hold the values of dx. The example below shows how such file might look like:

```
8
0.8
0.4
0.2
0.1
0.05
0.025
0.0125
0.00625
```

After you save the file read through the code below and see how the loop was implemented.

```
PROGRAM FINITE_DIFF
IMPLICIT NONE
!declaration of variables
REAL::x0,fx0,fx1,dx
REAL::df,dfc
REAL::err
INTEGER::i,n
PRINT*,'Please input the value of x0 = '
READ(*,*) x0
!opening of input and output files
OPEN(unit=111,file='input.dat',status='unknown')
OPEN(unit=222,file='output.dat',status='unknown')
!value of f at x0
fx0 = x0**2
!exact value of derivative of f at x0
df = 2*x0
!read the number of iterations from file
READ(111,*)n
!the loop begins
DO i=1,n
!read the value of dx from input file
READ(111,*)dx
!value of f at x0+dx
fx1 = (x0+dx)**2
!calculation of the derivative
dfc = (fx1-fx0)/(dx)
!calculate the error
err = abs((df-dfc)/df)
```

```
!write result to a file
WRITE(222,*) dx,err
!write result on a screen
PRINT*,i,dx,err
END DO
CLOSE(111)
CLOSE(222)
END PROGRAM FINITE_DIFF
```

You can open the output.dat file and copy the results to Excell in order to print the results on a graph. Now change the scheme to Central Difference, produce similar graph and check how those two methods compare. You can experiment with your input file in order to get good results. Remember that each time your code runs, it will overwrite the file output.dat!

We have one final concept to investigate today – do loops with conditional exit. Instead of running for a pre-definef number of iterations, such loop will terminate only if certain condition will be met. Again let's see how it works on example:

```
PROGRAM LOOPY
IMPLICIT NONE
INTEGER:: i
i=0
DO
i=i+1
PRINT*,'Hello World!, count = ',i
IF(i.ge.10) EXIT
END DO
END PROGRAM LOOPY
```

Note that this time I had to set the value of variable i before the loop and increment it's value inside the loop. I have put a condition on the value of i so that the loop will stop when i will be greater or equal than 10. I got the same result as with the normal do loop, but we shall see that those two constructions are significantly different. What would happen if I did not set the value of i to 0 before the loop? What if my exit condition would be insensible (like i.le.-1)?

Finally, what would happen if I do not increment the value of i inside the loop? Try to investigate those questions on your own. Little hint: in order to stop the program from working press CRTL+C.

2-5 Exercise

After you are done with it, we shall proceed to some more interesting code. Now since we know the conditional statements and loops, we want to investigate which value of dx would be best for Forward Difference scheme in case of our function. We will start with some large value of dx and will decrease it inside the loop as long as we will get small enough error.

Read the code below and fill in the missing lines (marked with description and stars ***).

```
PROGRAM FINITE_DIFF
IMPLICIT NONE
!declaration of variables
REAL::x0,fx0,fx1,dx
REAL::df,dfc
REAL::err
PRINT*,'Please input the value of x0 = '
READ(*,*) x0
OPEN(unit=222,file='output.dat',status='unknown')
!exact value of derivative of f at x0
df = 2*x0
!set the initial value of dx to 0.8
!***
!the loop begins
DO
!value of f at x0
fx0 = x0**2
!value of f at x0+dx
fx1 = (x0+dx)**2
!calculation of the derivative
dfc = (fx1-fx0)/(dx)
!calculate the error
err = abs((df-dfc)/df)
!write result to a file
WRITE(222,*) dx,err
!write result on a screen
PRINT*,dx,err
!condition for exit: err<1e-03
!***
dx = dx-0.001
IF(dx.le.0) THEN
```

```
PRINT*,'Exit condition not met, dx hit 0'
PRINT*,'Emergency exit'
EXIT
END IF
END DO
CLOSE(222)
END PROGRAM FINITE_DIFF
```

Compile and run the code. How would you count the number of iterations that the code performed? Try to print the number of each iteration on the screen to get the result similar to:

```
I:\My documents\Fortran\a.exe
Please input the value of dx:
4
1 0.8 0.10000
2 0.799 0.0998
…
```

You can notice that using linear progress it takes a lot of iterations to get to sufficiently small dx. Instead of decreasing dx linearly, try to divide it, say, by 2 each iteration.

Now try to change the scheme to Central Differences. See how the results compare.

2-6 Summary

During this session you have learned:

1- How to use conditional statements

2- How to use loops

3- How to use conditional statements to control the loops

4- What is the difference in accuracy between Forward and Central Difference schemes

Lesson 3: Introduction to FORTRAN90 Programming

3-1 Previously on CFD lab...

• Conditional statements

• loops

• forward and centre finite difference schemes

If you have not finished all exercises from the last lab, please do so now. It is crucial that you are familiar with conditional statements and loops, so we can move on to a new topic.

Today, we are going to learn about...

3-2 Arrays

Arrays are another very important concept, especially in CFD. They allow us to store and operate on multiple data. Arrays are defined by two basic characteristics:

type – just as in case of simple variables there are different types of arrays. We are still going to use integer or real arrays size – size defines how many single variables are stored in a variable. At first we are going to look at single dimension arrays, but more dimensional arrays are also possible

Let's have a look on a simple example of declaring and using an array, You will note that we are going to use concepts learned on previous sessions.

```
PROGRAM ARRAYS
IMPLICIT NONE
INTEGER::i
!we declare two arrays of type REAL and size 10
REAL,DIMENSION(10)::x,f
!we fill the array x with arguments of a function f
DO i=1,10
x(i) = 0.1*i
```

```
END DO
!we calculate the values of function f=x**2 for given x
DO i=1,10
f(i) = x(i)**2
END DO
!we print the results on the screen
DO i=1,10
PRINT*,x(i),f(i)
END DO
END PROGRAM ARRAYS
```

After you compile and run this code, please change it so it will write the output to some file. Then use Excell or any other software to print a graph of the function. You can change the code to calculate values of any other function you want. You can also change the values of x and the number of x points where the value of f is calculated (you will need to change the size of arrays and loops accordingly). Please spend some time to understand this simple code. It will be very important soon.

Now let's go for something more in CFD area. Remember calculating the derivative of a function at a given point? Now we can calculate the derivative on a given domain and plot it. Let's have a look on the following example. It will calculate the derivative of sin(x) on a domain $(0, 2\pi)$ and print the results to a file. The code will automatically calculate the value of dx used by central difference scheme. Since we know the size of our domain, the dx will be defined by the number of points where we calculate values of derivatives.

```
PROGRAM FINITE_DIFF
IMPLICIT NONE
!df - calculated derivative of f, df_true - true value of
derivative
REAL,DIMENSION(20)::x,f,df,df_true
REAL::pi,dx
INTEGER::i
!we define the value of pi
pi = 2.0*asin(1.0)
!we calculate dx
dx = 2.0*pi/20
!we define x, corresponding values of f and true derivative
```

```
DO i=1,20
x(i) = 2.0*pi*i/20
f(i) = sin(x(i))
df_true(i) = cos(x(i))
END DO
!we calculate derivative of f using central difference
!note that we are not able to calculate the derivative at extreme
points of our domain
DO i=2,19
df(i) = (f(i+1)-f(i-1))/(2*dx)
END DO OPEN(unit=111,file='output.dat',status='unknown')
DO i=2,19
PRINT*, x(i),f(i),df(i),df_true(i)
WRITE(111,*)x(i),f(i),df(i),df_true(i)
END DO
CLOSE(111)
END PROGRAM FINITE_DIFF
```

Compile and run the code. Look at the results in output.dat file and use them to produce a graph. Check how well the calculated derivative matches the true one. Save the graph so you could show it at the end of the session to the tutor. Try to change the number of points where the derivative is calculated. Tip: you should alter the code in at least 6 lines.

3-3 Dynamic allocation of arrays

You have probably noticed that it is not very handful to declare the size of the array a priori. In case you want to make any change, you need to alter the code in many locations. Much easier would be to declare an array of a variable size that could be declared later. We can do this by using dynamic allocation. While declaring an array, we only need to say that it is allocatable. Later in the code, when we know the size of the array we can allocate it. Let's see how it works on the example. We are going to use our code for calculating the derivative. Read it carefully and look for changes. This time we are going to input the number of points from the keyboard.

```


PROGRAM FINITE_DIFF
IMPLICIT NONE
```

```
!df - calculated derivative of f, df_true - true value of derivative
REAL,DIMENSION(:),ALLOCATABLE::x,f,df,df_true
REAL::pi,dx
INTEGER::i,n
!we define the value of pi
pi = 2.0*asin(1.0)
!input number of points
PRINT*,'Please input number of points:'
READ(*,*)n
!allocate arrays
ALLOCATE(x(n),f(n),df(n),df_true(n))
!calculate dx
dx = 2*pi/n
!we define x, corresponding values of f and true derivative
DO i=1,n
x(i) = 2.0*pi*i/n
f(i) = sin(x(i))
df_true(i) = cos(x(i))
END DO
!we calculate derivative of f using central difference
!note that we are not able to calculate the derivative at extreme
points of our domain
DO i=2,n-1
df(i) = (f(i+1)-f(i-1))/(2*dx)
END DO
OPEN(unit=111,file='output.dat',status='unknown')
DO i=2,n-1
PRINT*,x(i),f(i),df(i),df_true(i)
WRITE(111,*)x(i),f(i),df(i),df_true(i)
END DO
CLOSE(111)
END PROGRAM FINITE_DIFF
```

Now you can easily change the number of points and therefore the size of dx. Make graphs for different values of n and see how they compare. Save the graphs for further reference.

Can you figure out a way to calculate the value of the derivative at extreme points i=1 and i=n? Can you calculate the error (as in previous session) and plot a graph of error vs x for different dx? Finally can you find maximum absolute error for each dx and plot a graph of the maximum error vs dx?

3-4 Exercise

Now is the time for your first exercise on your own. You should use all the knowledge you have learned during last sessions. The task is to write a code which will integrate sin(x) over an interval (0,2pi) using a trapezoid rule:

$$F(x_0) \approx \frac{f(x_0) - f(x_0 + \Delta x)}{2} \Delta x$$

Where

$$F(x) = \int f(x) dx$$

The algorithm of such code would be as follows:

1) declare necessary arrays and variables

2) define pi

3) read the number of points from keyboard

4) allocate the arrays

5) calculate dx

6) in a loop put the values of x to an array

7) in a loop calculate the values of function f

8) in a loop calculate the integral of f using trapezoid rule

9) save the the results to a file

Good luck! Do not hesitate to ask demonstrators for help. When you finish, show the results to one of the demonstrators. Try to perform similar error analysis as in previous example. If

you run out of time, do not worry. Just show the demonstrators the graphs you have saved before.

3-5 On the next session...

...we are going to learn about subroutines and functions. We will produce some really useful piece of code for derivation and integration and challenge some interesting numerical problems.

Lesson 4: Introduction to FORTRAN90 Programming

4-1 Previously on CFD lab...

we have learned about:

· arrays

· dynamic memory allocation

· numerical integration using trapezoid rule

By now you should be also familiar with programming concepts like:

· real and integer variables

· reading and writing data from/to a file

· conditional statements

· loops

Today, we are going to put some icing on our programming cake and learn about functions and subroutines. Knowlegde gained through this course will allow you to write really sensible and meaningful CFD code. During the session do not be afraid to experiment, ask questions like: “what will happen if I change this line/function/variable?” etc. If you will not be able to get the answers yourself, demonstrators will be more than happy to help you.

4-2 Functions

Functions are subprograms in the code that given some input data will produce a single variable output. Functions, like variables, can be of different types. We will focus only on REAL type functions. Simple example below will explain the concept best. Mind that after the main body of the program we put statement contains and after that the definition of some custom function.

```
PROGRAM FUNCTIONS
IMPLICIT NONE
INTEGER::i
REAL::x,y,a
PRINT*,'Please input the first length of a rectangle:'
READ*,x
PRINT*,'Please input the second length of a rectangle:'
READ*,y
!we calculate the area of the rectangle
a = area(x,y)
!we print the result on the screen
PRINT*,'The area of the rectangle is:',a
CONTAINS
REAL FUNCTION area(x,y)
REAL::x,y
area = x*y
END FUNCTION area
END PROGRAM FUNCTIONS
```

Compile and run this code. Try to write your own function for one and three input parameters. Check the results. Now let's see how can we use the concept of functions in CFD problems. Remember the integration code that you were asked to write yourself at the end of the previous session? Below is working version of this code, but if you like you can use your own. In the function declaration please specify the funcion of your choice.

```
PROGRAM INTEGRATION
IMPLICIT NONE
REAL,DIMENSION(:),ALLOCATABLE::x,f,ff,ff_sum
REAL::pi,dx
INTEGER::i,n
!we define the value of pi
pi = 2.0*asin(1.0)
!input number of points
PRINT*,'Please input number of points:'
```

```
READ(*,*)n
!allocate arrays
ALLOCATE(x(n),f(n),ff(n),ff_sum(n))
!calculate dx
dx = 2*pi/n
!we define x and corresponding values of f using custom function
DO i=1,n
x(i) = 2.0*pi*i/n
f(i) = func(x(i))
END DO
!we calculate intermediate integrals using trapezoid rule
ff = 0
DO i=1,n-1
ff(i) = (f(i)+f(i+1))/2.0*dx
END DO
!we calculate integral by suming up intermediate integrals
ff_sum = 0
ff_sum(1) = ff(1)
DO i=2,n-1
ff_sum(i) = ff(i) + ff_sum(i-1)
END DO
OPEN(unit=111,file='output.dat',status='unknown')
DO i=1,n-1
WRITE(111,*)x(i),f(i),ff_sum(i)
END DO
CLOSE(111)
DO i=1,n-1
PRINT*,x(i),f(i),ff_sum(i)
END DO
CONTAINS
REAL FUNCTION func(x)
REAL::x
!please specify your own function
func =
END FUNCTION func
END PROGRAM INTEGRATION
```

Try this with different functions. Plot the results and save graphs for further reference. You might notice that the result is shifted comparing to the expected result. Why is that?

4-3 Subroutines

Another, even more important than functions, concept are subroutines. They are subprograms that can get many variables or arrays as an input and produce many different types of output at the same time. You could imagine them as a building blocks of the code, each with different task to do. Again, we will have a look on some simple example that should explain everything. We will calculate the area and circumference of a rectangle.

```
PROGRAM SUBROUTINES
```

```
IMPLICIT NONE
REAL::x,y,area,circ
PRINT*,'Please input the first length of a rectangle:'
READ*,x
PRINT*,'Please input the second length of a rectangle:'
READ*,y
!we call the subroutine rect
call rect(x,y,area,circ)
!we print the result on the screen
PRINT*,'The area of the rectangle is:',area
PRINT*,'The circumference of the rectangle is:',circ
CONTAINS
SUBROUTINE rect(x,y,a,c)
REAL,INTENT(IN)::x,y
REAL,INTENT(OUT)::a,c
a = x*y
c = 2*x+2*y
END SUBROUTINE rect
END PROGRAM SUBROUTINES
```

Have a closer look on the declaration of the subroutine in the contains section. This time we do not specify the type of the subroutine. Instead, we need to specify whether each argument of the subroutine is input or output. We use INTENT statement for that. In this case x and y are input and a and c are output. It is worth to notice that when calling a subroutine in a main code, we do not need to provide variable of the same name as it is declared inside the subroutine. Only the type of the variable must match. It makes sense since one person in Australia can write a subroutine, and another person in UK might want to use it without changing all the variables in his code to match the names of those in subroutine. He or she would only need to know the type and order of variables needed.

Try to play around with different subroutines. Think how can you implement subroutine to the finite difference code we have been working with on the last session. If you think you understand the concept, you can move on to the final exercise.

4-4 Exercise

Now is the time for a major programming exercise. We will write a code which will solve 1D Navier-Stokes equation as explained on the lecture. You can access the notes here (see page ?): Chapter 5

We need to solve the governing equation:

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial p}{\partial x}$$

First we need to express it in a form that will allow us to solve it numerically. You already know how to express a derivative using central finite difference scheme:

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u(x_i - \Delta x) - 2u(x_i) + u(x_i + \Delta x)}{\Delta x^2} = \frac{u_{i-1} - 2u_i + u_{i+1}}{\Delta x^2}$$

The term on tight hand side we will consider as constant and feed its value to the program. So we have to solve discretised equation:

$$\frac{u_{i-1} - 2u_i + u_{i+1}}{\Delta x^2} = \frac{\partial p}{\partial x}$$

after making simple manipulations we get:

$$u_{i-1} - 2u_i + u_{i+1} = \frac{\partial p}{\partial x}\Delta x^2$$

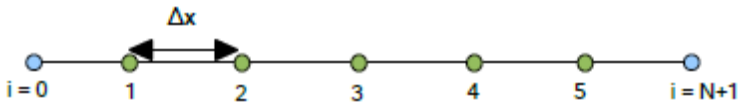
Let:

$$\frac{\partial p}{\partial x}\Delta x^2 = f_i$$

Now left-hand-side of the equation is unknown, while we know our right-handside.

$$u_{i-1} - 2u_i + u_{i+1} = f$$

Let's assume we select N=5 equispaced points inside our domain plus two at the boundaries:



For each internal point we are going to write an equation:

$$u_{i-1} - 2u_i + u_{i+1} = f$$

Points u_0 and u_{N+1} lie on the boundary, so they will represent constant boundary values (we will choose their values). We can write the entire system of equations in a matrix form:

$$\begin{bmatrix} -2 & 1 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 1 & -2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} = \begin{bmatrix} f_1 - u_0 \\ f_2 \\ f_3 \\ f_4 \\ f_5 - u_6 \end{bmatrix}$$

or using symbols:

$$\begin{bmatrix} b_1 & c_1 & 0 & 0 & 0 \\ a_2 & b_2 & c_2 & 0 & 0 \\ 0 & a_3 & b_3 & c_3 & 0 \\ 0 & 0 & a_4 & b_4 & c_4 \\ 0 & 0 & 0 & a_5 & b_5 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} = \begin{bmatrix} f_1 - a_1 \cdot u_0 \\ f_2 \\ f_3 \\ f_4 \\ f_5 - c_5 \cdot u_6 \end{bmatrix}$$

where:

$$a = [a_1, a_2, a_3, a_4, a_5] = [1,1,1,1,1]$$
$$b = [b_1, b_2, b_3, b_4, b_5] = [-2,-2,-2,-2,-2]$$
$$c = [c_1, c_2, c_3, c_4, c_5] = [1,1,1,1,1]$$

For the purpose of our example let's choose:

$$\frac{\partial p}{\partial x} = 2$$

One last thing is to solve the system of linear equations. Since the matrix that defines the system is tridiagonal (has only non-zero elements on diagonal, upper diagonal and lower diagonal) we can use fairly simple Tomas algorithm called also TDMA (you can google for the full name of the algotirhm).

The code below solves the 1D NS equation using TDMA. TDMA algorithm is coded in a subroutine that takes a,b,c and f as input and gives u as an output. Your task is to fill in the empty lines (marked with comments) and make the code run properly.

```
PROGRAM 1D_NS
IMPLICIT NONE
INTEGER::n
!dynamically declare arrays a,b,c,u,f
!***
REAL::Lx,dx,dpdx !Lx - domain size, dx - step length, dpdx -
pressure
gradient
INTEGER::i
PRINT*,'Please input the number of internal points:'
READ*,n
Lx = 1.0
dpdx = -2.0
!calculate dx
!***
!allocate arrays a,b,c,u,f of size n
!***
!allocate u - mind that is has entries indexed from 0 to n+1
ALLOCATE(u(0:n+1))
!intialize u with boundary and initial values
u(0) = 0 !boundary value
u(1:n) = 0 !initial value - does not matter what it is
u(n+1) = 0 !boundary value
!put proper values to a,b,c
DO i=1,n
a(i) = !***
b(i) = !***
c(i) = !***
END DO
!calculate right-hand-size f
f(1) = !***
DO i=2,n-1
f(i) = !***
END DO
f(n) = !***
!call subroutine TDMA
!hint: if you want to pass only elements indexexd 1 to 5 from
array u
!you can address them by typing u(1:5)
!***
!print results to a file and on the screen
OPEN(unit=111,file='output.dat',status='unknown')
DO i=0,n+1
WRITE(111,*) dx*i,u(i)
PRINT*,dx*i,u(i)
END DO
CLOSE(111)
DEALLOCATE(a,b,c,u,f)
CONTAINS
SUBROUTINE TDMA(a,b,c,u,f,n)
!****************************************
```

```
! subroutine solves a system of linear equations
! with tridiagonal matrix
! a - size n matrix holding lower diagonal values
! b - size n matrix holding diagonal values
! c - size n matrix holding upper diagonal values
! f - size n matrix holding right-hand-sides of equation
! u - size n matrix holding solution for internal points of
domain
! (without boundary values)
!
!mind that b and f will be overwritten during the execution of
subroutine
!*****************************************
INTEGER,INTENT(IN)::n
REAL,DIMENSION(n),INTENT(INOUT)::a,b,c,f
REAL,DIMENSION(n),INTENT(OUT)::u
INTEGER::i
!forward elimination
DO i=2,n
b(i) = b(i) - a(i)/b(i-1)*c(i-1)
f(i) = f(i) - a(i)/b(i-1)*f(i-1)
END DO
!backward substitution
!mind the inverse direction of the do loop
u(n) = f(n)/b(n)
DO i=n-1,1,-1
u(i) = (f(i) - c(i)*u(i+1))/b(i)
END DO
END SUBROUTINE tdma
```

4-5 Congratulations!

You have just solved numerically your first equation. You must know that using this method you can tackle many computational problems. Try different number of points and plot some graphs for further reference. Now let's try to improve our code even further. Below is the formula to calculate a residual. The residual will show us how accurate is our solution without referring to true solution (it will show us how accurately our discretised system of linear equations is solved rather than how accurately the real problem is solved). Write a subroutine that will take arrays a,b,c,u and f as an imput and produces array r as an output. Mind that you need to input entire array u to the subroutine (u will have size (0:n)). Then print the maximum absolute value of r and check how does it change for different values of

dx. Try to plot a graph of maximum absolute r against dx. Following function will calculate the maximum absolute value of array (copy it into the contains section of your code):

```
REAL FUNCTION maxabsr(r,n)
INTEGER::n
REAL,DIMENSION(n)::r
maxabsr = 0
DO i=1,n
IF(abs(r(i)).ge.maxabsr)THEN
maxabsr = abs(r(i))
END IF
END DO
END FUNCTION maxabsr
```

Calculate maximum absolute value of residual for different values of dx and create a graph. Discuss it with one of demonstrators or a lecturer.

4-6 Summary

Given the techinques explained during last four session you are able to write a broad spectrum of CFD codes. If any time you are in trouble or doubt, google for solutions as Internet has many answers. If that fails, you can always write on Michal's address. Next session will be a troubleshooting session for you to throw your assigment problems at us. Please make good use of it since later it might be harder to catch any of demonstrators or lecturer. Good luck in your programming career!

Lesson 5: Introduction to FORTRAN90 Programming

This handout will give you some hints for your CFD assignment 1. It will provide a framework for construction of your own Fortran programs. It is by no means complete, so feel free to change it as you like. If you feel something could be done in a different way, do not be afraid to use your own ideas.

5-1 Exercise 1

In this exercise you are asked to produce a graph of error against different dx for four different differentiation schemes.

```
!Allocate variables
!x – point where derivative is calculated
!dx – size(n) array of different step sizes
!df – size(n,4) 2D array of calculated derivative for four
different schemes
!df_true – true value of derivative
!err – size (n,4) 2D array of errors for different dx and four
methods
!plus some other variables as needed
!Select the value of x from keyboard
!
!Calculate the true value of derivative
!df_true =
!In a loop set the values for dx
!do i=...
!dx(i) =
!end do
!Loop over elements of dx array
!do i=...
!calculate value of df(i,1) for first method
!calculate value of df(i,2) for second method
!etc...
!
!calculate error for four methods
!err(i,1) = (df_true – df(i,1))/df_true
```

```
!end do
!write results to a file
```

5-2 Exercise 2

You are asked first to write a code for solving a steady 1D Navier-Stokes equations. First step would be to run a steady simulation and then add an unsteady part. The main code for steady part should look something like that:

```
!allocate variables
!a,b,c – arrays for three diagonal coefficients
!f – array for right hand side
!u – array for results
!dx (or dy), dpdx value and anything else you need
!loop over space
do i=
!calculate a(i),b(i),c(i), and f(i)
!end of your DO loop
end do
!CALL TDMA
!save the results to a file
!close the file
```

The main code for **unsteady** simulation will look like:

```
!allocate variables
!a,b,c – arrays for tridiagonal coefficients
!f – array for right hand side
!u – array for results
!dx (or dy), dt, and anything else you need
!specify initial condition from the steady simulation above
!do i=
u(i) =
!end do
!Outer DO loop for your time index
do n=
!Inner DO for your space index
do i= !calculate a(i),b(i),c(i), and f(i)
!end of your inner DO loop
end do
!CALL TDMA
!save the results to a file
```

```
!do NOT close the file
!end of your outer DO loop
end do
!close the file here
```

You run the above simulation for 0 < t < 10. Then, you continue the unsteady simulation for t > 10 with the new_dpdx value, and the initial condition from the first part of the unsteady simulation at t = 10. Do not hesitate to ask if you have any problem. Remember that the algorithms above might not be complete, and you might need to add some lines in order to make them work. Good luck!