Introduction to Programming using FORTRAN 95

http://www.fortrantutorial.com/

These worksheets aim to provide an introduction to programming. The language chosen for this is FORTRAN 95. This is because FORTRAN is particularly suitable for science and engineering; it is also very widely available.

The skills you acquire working through these notes can be applied to any computing language. The concepts you will learn are shared in common with every other computing language.

This document and all the examples may be found online at:

http://www.fortrantutorial.com/

1 THE BASICS 3

1.1 AIMS 3
1.2 INSTALL FTN95 PERSONAL EDITION 3
1.3 YOUR FIRST PROGRAMMING SESSION 3
1.4 PLATO - A PROGRAMMING ENVIRONMENT 3
1.5 RUNNING YOUR FIRST FORTRAN 95 PROGRAM 4
1.6 PROGRAM STRUCTURE 6
1.7 MORE ON INPUT AND OUTPUT 6
1.8 MORE DATA TYPES – INTEGER AND CHARACTER 8
1.9 SAVING THE CONTENTS OF OUTPUT WINDOW 10

2 MAKING DECISIONS 11

2.1 AIMS 11
2.2 ASSIGNMENT 11
2.3 ARITHMETIC 11
2.4 INTRINSIC FUNCTIONS 12
2.5 MAKING DECISIONS 13
2.6 PROGRAM STYLE 14
2.7 MORE ON DECISION MAKING 14
2.8 OTHER LOGICAL OPERATORS 14
2.9 MULTIPLE CONDITIONS 15
2.10 THE SIMPLE IF STATEMENT 15
2.11 IMPORTANT NOTE – TESTING FOR ZERO 16

3 LOOPS 17

3.1 AIMS 17
3.2 MIXING VARIABLE TYPES 17
3.3 THE DO LOOP 18
3.4 NESTED DO LOOPS 19
3.5 USING LOOPS TO DO SUMMATION 20

4 USING FILES AND EXTENDING PRECISION 22

4.1 AIMS 22
4.2 READING FROM FILES 22
4.3 WRITING TO FILES 23
4.4 EXTENDING THE PRECISION 23
4.5 MAGNITUDE LIMITATIONS 25
4.6 CONVERGENCE – EXITING LOOPS ON A CONDITION 25

5 ARRAYS AND FORMATTED I/O 27

5.1 AIMS 27
5.2 ARRAYS 27
5.3 ARRAY MAGIC 29
5.4 MULTI DIMENSIONAL ARRAYS 30
5.5 FORMATTING YOUR OUTPUT 31
5.5.1 Integer Specification 32
5.5.2 Floating point Specification 32
5.5.3 Exponential Specification 32
5.5.4 Character Specification 33
5.6 IMPLIED DO LOOP TO WRITE ARRAYS 33

6 SUBROUTINES AND FUNCTIONS 35

6.1 AIMS 35
6.2 RE-USING CODE – THE SUBROUTINE 35
6.3 ARGUMENTS TO SUBROUTINES 36
6.4 USER DEFINED FUNCTIONS 38

7 ADVANCED TOPICS 40

7.1 AIMS 40
7.2 ARRAY FUNCTIONS 40
7.3 WRITING REAL PROGRAMS - FLOW CHARTS 42

1 The Basics

1.1 *Aims*

By the end of this worksheet, you will be able to:

- □ Create and run a FORTRAN 95 program
- □ Understand basic program structure
- □ Start to deal with programming errors
- □ Start to understand **real, integer** and **character** variable types.
- □ Save a copy of your output in Word.

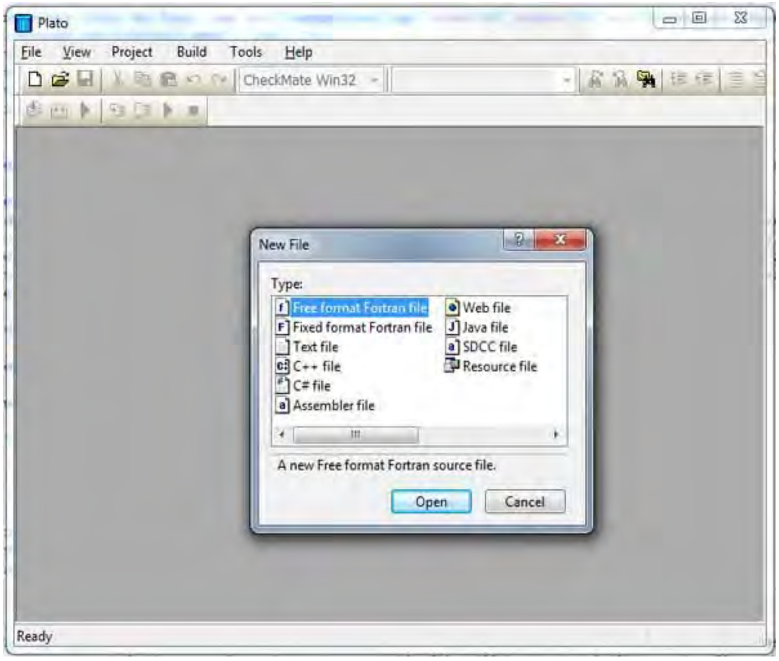
1.2 *Install FTN95 Personal Edition*

- Search for Silverfrost FTN5 personal edition or click this link http://www.silverfrost.com/32/ftn95/ftn95_personal_edition.aspx.
- Download and install the software accepting all the defaults.

1.3 *Your first programming session*

- Locate and double click the Plato icon
- Click **File, New**
- Select **Free Format Fortran File**
- Click **File, Save As**
- Create a directory called fortranprograms and open it
- Type **first.f95**

1.4 *Plato - a programming environment*

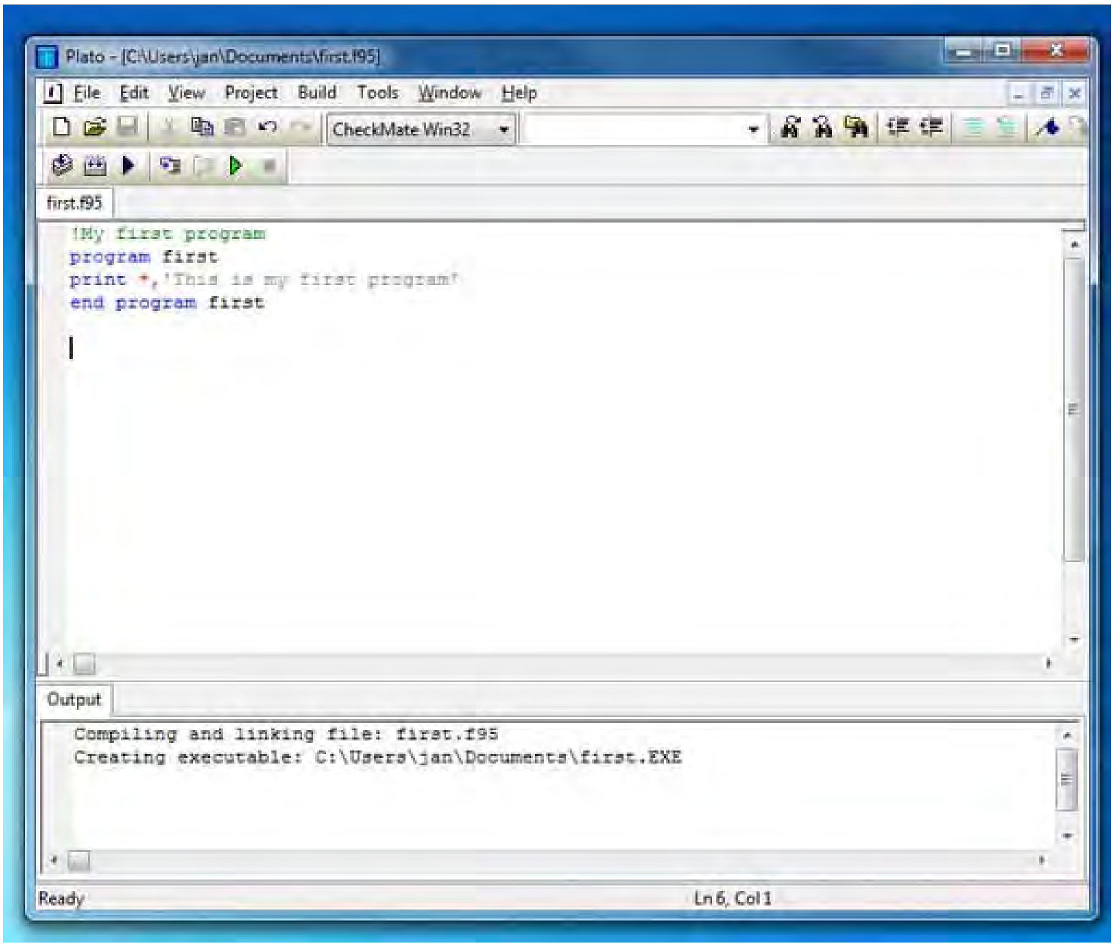


A new Free format Fortran source file.

Plato is a "programming environment". Within Plato, you can create and edit programs and get them to run. Plato's editor is special – it understands the syntax of various programming languages. We tell Plato which language we are using when we create our empty file and save it with a **.f95 (**FORTRAN 95) extension. Provided you have given your file the appropriate extension, Plato's editor will be able to check the syntax of the program, highlighting the various keywords that it knows about using a colour code to distinguish between the various elements of the language.

Always ensure that your program files have a .f95 extension

1.5 Running your first FORTRAN 95 Program



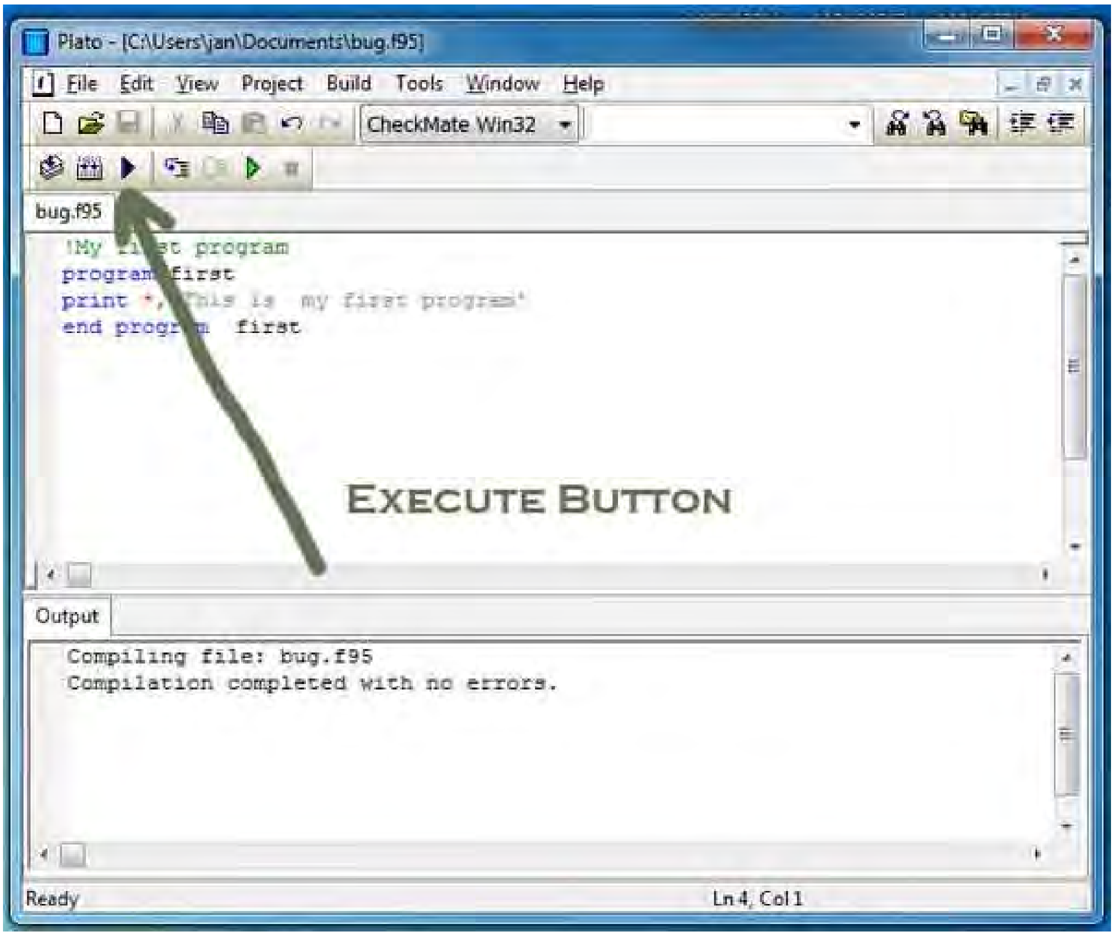
```
!My first program
program first
print *,'This is my first program'
end program first
```

```
Compiling and linking file: first.f95
Creating executable: C:\Users\jan\Documents\first.EXE
```

Exercise 1.1

- Type in the following *exactly* as shown:

```
!My first program
program first
print *,'This is  my first program'
end program  first
```



Plato - [C:\Users\jan\Documents\bug.f95]
File Edit View Project Build Tools Window Help
CheckMate Win32
bug.f95
EXECUTE BUTTON
Output
Ready
Ln 4, Col 1

```
!My first program
program first
print *, 'This is  my first program'
end program  first
```

```
Compiling file: bug.f95
Compilation completed with no errors.
```

- Click the **black** ►, (the **Execute** button).
- Plato will get FTN95 to check your program for errors. If it finds any problems, it will give you the details. If you have typed in the program *exactly* as shown above, an executable file will be generated (first.exe). Plato will then automatically get the program to start executing.
- A banner will appear for a couple of seconds and will then disappear (that"s the price we have to pay for using the free software)
- A black console window will appear.
- Press **Return** to close the window. **Do not** click the X at the top right of the window.

Plato can get upset if you do not press **Return** to close the window, try this…

- Save your program first!
- Run the program again (click ►)
- This time click the X at the top right of the window to close it.
- Make up your own mind about which is the better way to close this window in future!

1.6 Program Structure

Examine the following short program:

```
   program sum                           !a: name of program
!an example of program structure         !b: a comment
   real :: answer,x,y                    !c: declarations
   print *, 'Enter two numbers'          !d: output
   read *, x                             !e: input
   read *, y                             !e: input
   answer=x+y                            !f :arithmetic
   print *, 'The total is ', answer      !g: output
   end program sum                       !h: end of program
```

There are a number of general points here:

- The program is made up of a number of lines. Each line is called a **statement**.
- Each **statement** is made up of
  - **variable** names e.g. **`answer, x, y`**
  - **operators** e.g. **+, -** etc
  - **keywords** e.g. **`read, print`**
- The **statements** are executed sequentially.

Let's break the program down, line by line:

a) The name of the program. Keep it reasonably short and meaningful.

b) A comment explaining the purpose of the program. Comments are indicated by an exclamation mark. All text to the right of an exclamation mark is ignored by the compiler. Programmers use comments to help them remember how a program works. **Use of appropriate comments in programs aids understanding and is good practice**.

c) **Variables** - **`answer, x`** and **`y`** are used to store floating point numbers – we indicate this by **declaring** them as **real**.

d) **`print *,`** outputs to the screen – the asterisk means use the default number of decimal places when the number is written to the screen.

e) We **`read`** information from the keyboard and store the values in **`x`** and **`y`**.

f) Do some arithmetic and store the answer in **`answer`**.

g) Output the result to the screen

h) Conclude the program

1.7 More on Input and Output

Exercise 1.2

- Open a new file and call it io.f95.
- Type in the following program:

```
program io
real :: x,y,z
print *, 'enter the values x,y and z'
read *, x,y,z
print *, 'the values you typed are for z,y,x are: ',z,y,x
end program io
```

- Execute it by pressing ▶
- You can enter the numbers one at a time and press the **Enter** key each time.
- Execute the program again
- This time type all three numbers on one line separated by commas.

Look at the **print** statement

```
print *, 'the values you typed are for z,y,x are: ',z,y,x
```

In this statement, we are outputting four separate things, a literal string of characters,

```
'the values you typed are for z,y,x are: '
```

and the **variables** z, y, and x. We may output several items at one time, provided they are separated by commas.

Exercise 1.3

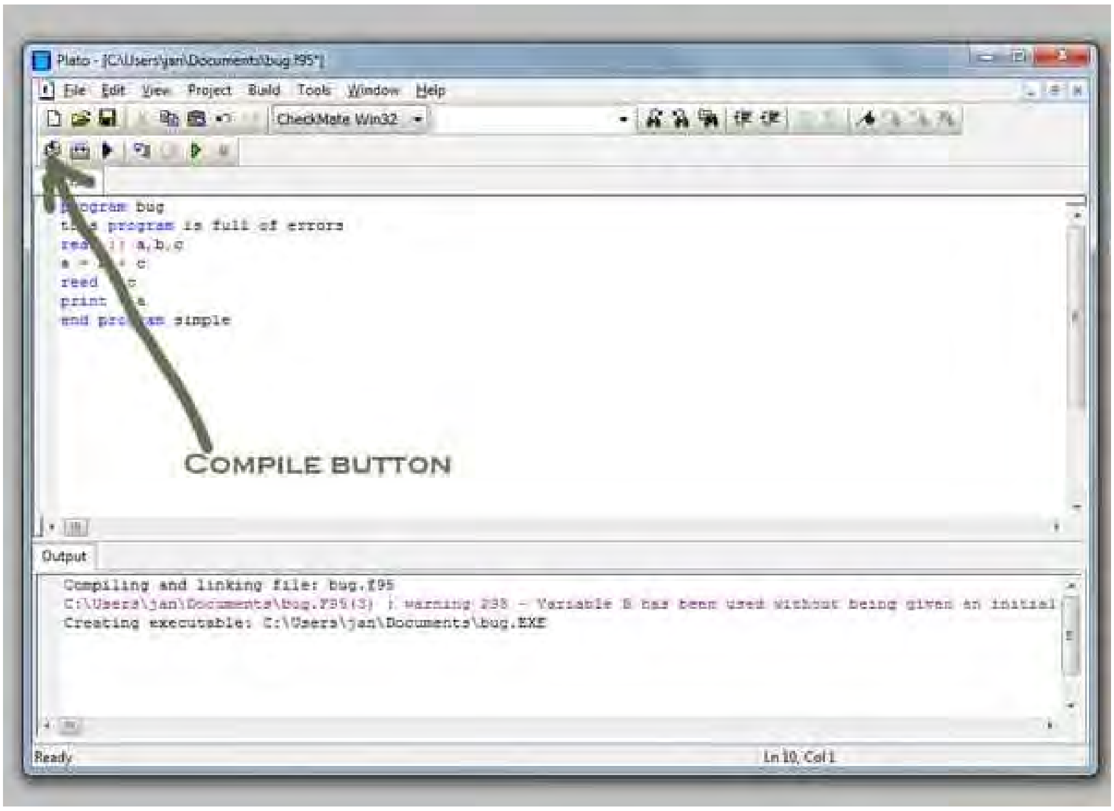
The following program has a number of errors.

- ❑ Create a new file called **bug.f95** and then type in the following program exactly as shown.
- ❑ You can also download this file from http://fortrantutorial.com/fortrantutorial-example-programs/index.php

```
program bug
this program is full of errors
real :: a,b,c
a = b + c
read *,c
print *,a
end program simple
```

The compiler will report two error messages when it attempts to compile. Click on the **details** button. Each error generates a message.

Double clicking on the message will
take you to the line in the program where the fault occurs.



```
program bug
this program is full of errors
real :: a,b,c
a = b + c
reed *,c
print *,a
end program simple
```

```
Compiling and linking file: bug.f95
C:\Users\jan\Documents\bug.F95(3) : warning 298 - Variable B has been used without being given an initial
Creating executable: C:\Users\jan\Documents\bug.EXE
```

- Correct the two errors.
- Click Execute ⍰
- There is now one further error, Plato will provide a yellow warning alert. **Watch the screen carefully! The window will close and then the program will start to execute. Something is not correct however… the program will "hang"**. It is actually waiting for you to input a value, because of the line   read *,c.   To the user of the program, this is not at all obvious – they may have thought that the program has crashed!
- Type in a number then press **enter**
- The program returns an strange value. This is an "**execution time**" error.
- We need to find out what the warning message was. Click the "compile" button (to the right of the binoculars). Then click the "details" button. Plato will tell you that the variable **b** has not been given a value.
- Correct the program to give **b** a value, and then execute the program again.
- There is **still** a problem. This time, it is a problem with the program's **logic**.

Need a Hint?   The program statements are executed **sequentially**.

```
a=b+c
read  *, c
print *, a
```

The statement `a=b+c` doesn't make sense, as at this stage of the program, we haven't yet given a value to `c`.

Important points to note

- There are two types of errors associated with this program: **compiler** errors and **run-time** errors.
- The program is also **user-unfriendly**. The program waits for input without telling the user what is needed.

Fix the run time error by:

- read in a value for b
- correct the order of the statements
- make the program more user-friendly,

then compare your program with the one called **bugfixed.f95** at

http://fortrantutorial.com/fortrantutorial-example-programs/

1.8 More Data types – integer and character

So far, we have only used **real** (floating point numbers) in our programs. We can also specify that numbers are **integer** and **character**. Program **convert**, below, demonstrates their use.

Within a given range, **integers** are always represented *exactly* whereas the precision of real numbers is limited by the architecture of the machine. The **real** variable type gives us 6 figure decimal precision. (If this doesn't seem enough – don't worry we'll come back later on when we examine how to increase the number of digits of precision in Section 4).

Character variables hold strings of characters like

```
'A happy day was had by all'
'Yes'
'N'
'3 + 4 equals 7'
```

When the **character variable** is declared, we show the maximum length that the string can occupy by following the name by a * then its maximum length. The example below has a maximum length of 10 characters allowed for a person's name – this might not always be enough! You have to make a judgement here.

```
  program convert
!This example shows the use of integer and character variables.
  implicit none
  integer   :: pounds,pence,total
  character :: name*10
  print *,'What is your name?'
  read *,name
  print *, 'Hi ',name,'! Enter number of pounds and  pence'
  read *, pounds,pence
  total =100 * pounds + pence
  print *,'the total money in pence is ',total
  end program convert
```

NOTE the inclusion of the line

`implicit none`

By including it in your program, FORTRAN will check that you have properly declared all your variable types. In the bad old days of programming, declaration of variables was thought to be unnecessary and the old FORTRAN compilers used an implicit convention that integers have names starting with the letters in the range i – n, all the others being real. FORTRAN still allows you to do this if we don't include the line, **implicit none**. Time has shown that **one of the commonest reasons for error in a program** is the incorrect use of variables.

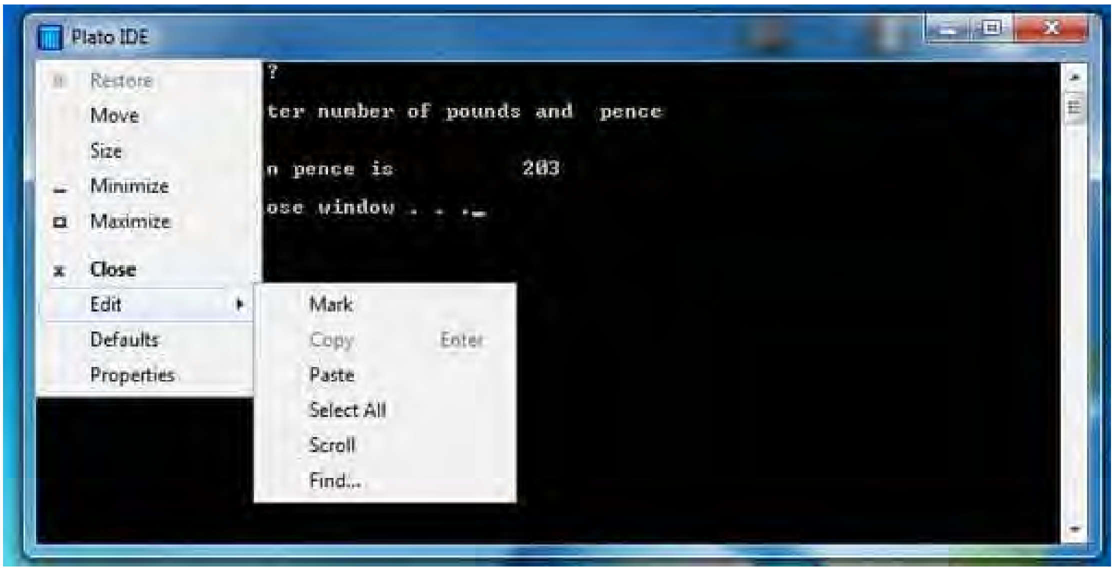
Always use *implicit none* at the start of every program.

Exercise 1.4

With the program **convert** in section 1.5 as a guide, write a program to test out everything you've learned so far. You might include different types of variables, for example **real**, **integer**, and **character**. Include input and output using **read** and **print**. An example might be a program that asks people questions, including things like their age and name and so on. It could, for example, print out their year of birth with a suitable message. It's up to you, just use your imagination.

1.9 Saving the contents of Output Window

Run your last program again. When the black output window opens right click on the Plato icon in the top left corner



```
?
ter number of pounds and  pence

n pence is          203
ose window . . ._
```

- Click on edit
- Click Select all
- Click copy
- Open a new document in Word or Notepad and click paste.

2 Making Decisions

2.1 Aims

By the end of this worksheet, you will be able to:

- ❑ Do **arithmetic**
- ❑ Start to use FORTRAN **intrinsic functions**
- ❑ Begin to understand program flow and logic
- ❑ Know how to **test for zero – important!**
- ❑ Learn more about good programming style

2.2 Assignment

When we start programming, the similarity between mathematical equations and FORTRAN statements can be confusing.
Consider the following FORTRAN statements:

```
x = 2              Store the value 2 in memory location x
y = 3              Store the value 3 in memory location y
z = x + y          Add the values stored in memory location
                   x and y and store the result in memory location z
```

In mathematics, “x = 2” means that the variable x is equal to 2. In FORTRAN it means **“store the value 2 in the memory location that we have given the name x”**.

The significance of this is made clearer by the following equation in mathematics:

$$x + y = z$$

In mathematics, this means that the left hand side of the equation is equal to the right hand side.
In FORTRAN, this expression is meaningless: there is **no** memory location "x+y" and so it would lead to a compiler error.

Rule – there can only ever be ONE variable name on the left hand side of an equals sign

Exercise 2.1

Write a program which reads in two numbers **a** and **b**. Get the program to swap the values around so that the value that was in **a** is now in **b**, and print out the result. *Hint* you need to declare a third variable for intermediate storage of the data. (Check your program by examining program **swap.f95** at http://fortrantutorial.com/fortrantutorial-example-programs/

2.3 Arithmetic

The arithmetic operators are

+, -	plus and minus
***, /**	multiply and divide
******	exponentiation (raise to the power)
( )	brackets

- ❑ The order of precedence in FORTRAN is identical to that of mathematics.
- ❑ Unlike algebra, the operator must always be present xy is *not* the same as x*y
- ❑ Where operations are of equal precedence they are evaluated left to right
- ❑ Consecutive exponentiations are evaluated right to left
- ❑ We can override the order of evaluation by use of brackets

Exercise 2.2

The following program is an example of the use of arithmetic.

```
    program calculate
    implicit none
! a simple calculator
    real :: x,y,z,answer
    x=1.5
    y=2.5
    z=3.5
    answer=x+y/z
    print *,'result is ',answer
    end program calculate
```

Explore the use of arithmetic operators by modifying program **calculate**. Use it to calculate the values:

1. $\dfrac{x+y}{x+z}$
2. xyz
3. $x^{y^{z}}$

2.4 Intrinsic Functions

FORTRAN is especially useful for mathematical computation because of its rich library of inbuilt functions (***intrinsic functions***). We shall mention a few briefly here:

function name	**type of argument**	**type of result**	**Definition**
sin(x)	real	real	sine
cos(x)	real	real	cosine
tan(x)	real	real	tangent
atan(x)	real	real	arctangent
abs(x)	real/integer	real/integer	absolute value
sqrt(x)	real	real	square root
exp(x)	real	real	e^x
log(x)	real	real	$\log_{10} x$

Trigonometric functions are calculated in radians (1 radian = 180/Pi degrees).

There are, of course, many more, and this list doesn't cover all FORTRAN variable types. The following example shows the use of some of the inbuilt functions.

```
program trig
implicit none
real :: a,pi
print *,'Enter an angle between 0 and 90'
read *, a
pi=4.0*atan(1.0)
print *,'the sine of ',a,' is ',sin(a*pi/180)
end program trig
```

2.5 Making Decisions

So far, our programs have worked as little more than basic calculators. The power of programming comes in when we have to make decisions. Copy the example program, **test.f95,** to your own file space. See if you can understand what is going on.

```
      program test
      implicit none
   !use of a simple menu
      real :: x,y,answer
      integer :: choice
   !set up the menu – the user may enter 1, 2 or 3
      print *,'Choose an option'
      print *,'1    Multiply'
      print *,'2    Divide'
      print *,'3    Add'
      read *,choice
      x=3.4
      y=2.9
   !the following line has 2 consecutive
   !equals signs – (no spaces in between)
      if (choice = = 1) then
            answer=x*y
            print *,'result = ',answer
      end if
      if (choice = = 2) then
             answer=x/y
             print *,'result = ',answer
      end if
      if (choice = = 3) then
         answer=x+y
         print *,'result = ',answer
      end if
      end program test
```

The bolded lines in the above program are called **if** … **end if** statements. They work like this:

```
if (condition is true) then
   execute this line
   and this
   and so on until we get to …
end if
```

It follows that if the condition is NOT true then the code 'jumps' to the next statement following the 'end if'. The statements between the **if** and the **end if** are deliberately indented, this makes the program easier to read.

We use two consecutive equals signs (no space in the middle) to test for equality. Compare

```
        if (choice == 3) then                  test
        choice = 3                             assignment
```

Exercise 2.3

Examine program **test** above. The line

```
print *,'result = ',answer
```

is repeated several times. Is this a good idea? Can you modify the program to make it more efficient?

2.6 Program Style

A good program:

- ❑ Uses comments *appropriately* to explain what is happening.
- ❑ Uses indentation to make the program easier to read.
- ❑ Uses *meaningful* variable names.
- ❑ Uses sensible prompts to let the user know what is going on.
- ❑ Uses `implicit none` at the start of every program.
- ❑ Is efficient!

If you want to get maximum marks for your assignments keep the above points firmly in mind. *It is not enough just to get a program to work!*

2.7 More on decision making

In our **test.f95** above, there was a problem if the user entered a value that wasn't catered for by the program.

What happens if the user doesn't enter one of the values 1, 2 or 3?

We are going to look at a new structure, called **if, else, endif** that handles this situation. Examine the following code snippet:

```
if (choice = = 1) then
    do something
else if (choice = =2) then
    do something else
else
     do this if nothing else satisfies the conditions
end if
```

2.8 Other logical operators

So far, all our tests have been for **equality**. There are several tests we can make:

= =	equal to (there is **no** space between the equals signs)
/ =	not equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

2.9 Multiple Conditions

Suppose we need to test if x is greater than y and y is greater than z. There are different ways of doing this:

```
if (x > y) then
        if (y > z) then
                do something
        end if
end if
```

This can also be handled by the following:

```
if (x > y .and. y > z) then
        do something
end if
```

notice the **.and.**

If we wanted to check whether a number were less than a given value or greater than a given value we could write:

```
if (x  < 10 .or. x > 20) then
        do something
end if
```

notice the **.or.**

Exercise 2.4

Write a program that reads a number from the keyboard. Get the program to decide whether:

- ❑ the value of the number is greater than 0 but less than 1
- ❑ or is greater than 1 but less than 10
- ❑ or is outside of both these ranges

Print out a suitable message to inform the user.

2.10 The simple if statement

There is a simpler, one line form of the **if** statement. Say we just wanted to print out a simple message such as

```
print *, 'enter a positive number'
read *, num
if (num <0)    stop
if (num < 10) print *, 'less than 10'
if (num > 10) print *, 'greater than 10'
 print *,'It is a positive number'
```

This snippet also introduces a useful, simple statement **stop** – it simply stops the program.

2.11 Important note – testing for zero

Suppose that you wish to test whether a **real** variable is zero. The test

Make sure you understand this !

```
if (x = = 0) then ….
```

is ***not*** a satisfactory test. Although **integer** numbers are held exactly by the computer, **real** numbers are not.

The way around this is to test if the absolute value of the variable is less than some small predefined value. For example:

```
if (abs(x) < .000001) then
  print *,'No zero values! Please enter another number'
  read *, x
end if
```

3 Loops

3.1 Aims

By the end of this worksheet, you will be able to:

- ❑ Understand more about the use of **real** and **integer** variables and how to use a mixture of data types in expressions
- ❑ Understand how to re-use code by **looping**
- ❑ Know how to control the number of times a section of code is executed by using a **do loop**

3.2 Mixing variable types

Exercise 3.1

Copy **divide.f95**

Make sure you understand this thoroughly!

```
program divide
implicit none
integer :: x
real :: y
x = 1
y = x/3
print *, y
end program divide
```

And run it. This program produces the following output:

```
  0.00000
```

Something odd is happening. The problem is the line:

```
y=x/3
```

FORTRAN evaluates the **right hand side** of the assignment **first** using **integer** arithmetic, because both x and 3 are integer. 1 divided by 3 cannot be stored as an integer, and so the value 0 is returned. The result, 0, is then converted to a real number and the assigned to y.

Replace the line in program **divide**

```
x = 1        by
x = 10
```

Your output should now be:

```
  3.00000
```

Can you see what is happening? FORTRAN is keeping the integer part of the answer and throwing the rest away.

To get over this problem, we have to signal to FORTRAN that we want it to calculate the right hand side of the expression using **real** arithmetic. If we want to keep x as **integer** data type, we could re-write our expression as follows:

```
y=x/3.0
```

The presence of a **real** number on the right hand side causes the right hand side of the expression to be evaluated using floating point arithmetic.

Actually, the problem is even more complicated! Where we have an expression like

```
y=x * ((2**i)/3)
```

where **x** and **y** are real and **i** is integer, FORTRAN computes the result in stages:

First it calculates (2**i)/3 and evaluates it as an integer number, then multiplies the result by x and evaluates it as real.

Exercise 3.2

Copy check.f95 to your computer.

```
    program check
!Integer and real arithmetic
    implicit none
    real :: x,y
    integer i
    x=2.0
    i=2
    y=x*((2**i)/3)
    print *,y
    y=x*((2.0**i)/3)
    print *,y
    end program check
```

… and examine its output. Make sure you understand why this is happening.

3.3 The do loop

Unless we are able to re-execute code, we might as well use a calculator… Now we start to take advantage of the power of the computer.

Exercise 3.3

Copy program loop.f95

```
program loop
implicit none
integer :: i
do i=0,20
      print *,i
end do
end program loop
```

Run the program. It prints out the numbers from 0 to 20 in steps of 1.

Note:

- ❑ **i** is called a **loop counter**. In this example, it has a start value of zero.
- ❑ All the statements within the **do** and **end do** are executed. In this example there is just the one statement, ie **print**.
- ❑ Each time the statements are executed, the **loop counter, i,** is **incremented** by 1.
- ❑ When the value of **i** is 20, the loop terminates, and the program resumes after the **end do**.

Change the **do** statement in program **loop** as follows:

```
do i = 50,70,2
```

Run the program. What happens?

The third argument in the **do statement**, is the **increment step**. If omitted, the value is taken as 1.

Loops can also decrement: try this

```
do i = 5,-5,-2
```

Exercise 3.4

Using a **do loop** to generate integer values of x between –10 and 10 in steps of 1, write a program that constructs a table of values of

y=1.0/x

What happened when x had the value zero? Use an **if, end if** to test for the condition that gives the incorrect value, and print out an appropriate message. Compare your result with divbyzero.f95.

Division by zero is one of the commonest reasons for a program to fail.

3.4 Nested Do Loops

We want to construct a table of values for z where

$z = x^y$

for values of x in the range 1 to 2 in steps of 0.5 and
y in the range 1 to 2 in steps of 0.5

Work through the next exercise which illustrates this:

Exercise 3.5

Copy program **xytab.f95** to your filespace.

```
      program xytab
      implicit none
  !constructs a table of z=x/y for values of x from 1 to 2 and
  !y from 1 to 4 in steps of .5
      real        ::  x, y, z
      print *,'     x      y      z'
      do x = 1,2
            do y = 1,4,0.5
                  z = x/y
                  print *, x,y,z
            end do
      end do
      end program xytab
```

Examine its output. Notice the use of the first **print** to give a heading to the table.

3.5 Using loops to do summation

Earlier on, we discussed the idea of assignments.

```
x = 1.0
```

means store the value 1.0 in the memory location called x.

If we had the following code:

```
x = 1.0
x = x + 1.0
print *, x
```

Can you guess what value would be printed out for x?

The answer would be 2.0.

Really important!

Bearing in mind the definition of an assignment, the statement

```
x = x +1.0
```

means "**add 1.0 to the value currently stored in memory location x and then store the result in memory location x**".

Exercise 3.6

Copy file **increment.f95** to your file space and examine its output.

```
program increment
implicit none
integer :: i
real :: x
x=1.0
do i=1,10
      x=x+1.0
      print *, i,x
end do
end program increment
```

- **Note carefully** that we have set the initial value of x *outside* of the **do** loop. Why have we done this? If you aren't sure – change the code to put the line x = 1.0 *inside* the loop – then examine the output.
- It is **important** to understand that if we use constructions such as **x = x + 1.0**, then it is vital to initialise x to some value. If we don't, it is possible that the value might be set to **any** random number. Run the program, make a note of the final value of x then put an exclamation mark in front of the **x = 1.0** statement and run the program again.

Exercise 3.7

Edit the line x = x + 1.0 in program **increment.f95**, and change it to x = x * i. Re-run the program and examine the output. What is significant mathematically about the sequence of numbers that has been generated?

4 Using Files and Extending Precision

4.1 *Aims*

By the end of this worksheet, you will be able to:

- ❑ **Read from** and **write to** files
- ❑ Use extended precision

4.2 *Reading from files*

In the real world, most of the data we use for our programs will be kept in files. We just need a modification to the **read** statement that we are already familiar with to do this.
This program reads 3 numbers from a file called 'mydata.txt' into an array. Use Windows **Notepad** to create such a file for yourself, or copy the file from **mydata.txt** which is on the website.

```
        program readdata
        implicit none
!reads data from a file called mydata.txt
        real :: x,y,z
        open(10,file='mydata.txt')
        read(10,*) x,y,z
        print *,x,y,z
        end program readdata
```

The new material here are the lines

```
        open(10,file='mydata.txt')
        read(10,*) x,y,z
```

The open statement links the file called 'mydata.txt' with an input device numbered 10 (it doesn't **have** to be **10**, it could be **any** positive integer). To read from device 10, we just use it as the first argument in the **read** statement.

Exercise 4.1

Use Notepad to create a file called evenodd.txt. In the file type 10 numbers, one per line. Write a program that reads data from evenodd.txt one line at a time. Check if each number is even or odd and print out a suitable message. One way to check if a number is even or odd is to use the **mod** intrinsic function, like this…

```
    if (mod(num,2)>0) then……
```

mod returns the remainder of the first argument divided by the second. If the return value is greater than zero, then the number must be odd. Check program **evenodd.f95** to see if you are correct.

4.3 Writing to files

This is a similar idea to reading from files. We need a new statement, though, instead of **print**, we use **write**.

```
program io2
!illustrates writing arrays to files
implicit none
real :: num
integer :: i
open(12,file='myoutput')
do i = 1,100
      num = i/3.0
      write(12,*) nums
end do
print *, 'finished'
end program io2
```

Exercise 4.2

Write a program which reads in numbers from a file one at a time. If the number is positive, it should store it in a file called 'positive.txt' and negative numbers in a file called 'negative.txt'.

4.4 Extending the precision

So far, we have used two types of variables, **real** and **integer**. The problem so far, as you will have noticed on output, is that we are extremely limited by the number of significant digits that are available for computation. Clearly, when we are dealing with iterative processes, this will lead rapidly to errors. We can, however, extend the precision available from the **single precision** default, which gives us 6 figure decimal precision to 15 figures by using a new specification for real numbers.

```
program extended
implicit none
integer, parameter :: ikind=selected_real_kind(p=15)
real (kind=ikind) :: sum,x
integer :: i
sum=0.0
do i=1,100
     x=i
     sum = sum + 1.0/(x**6)
end do
print *, sum
end program extended
```

produces the following output:

```
1.01734306196
```

Don't be put off by the odd looking code. In practice, the way of setting up this extended precision, is pretty much the same for every program.

We state the precision we want by the argument **p**

```
integer, parameter :: ikind=selected_real_kind(p=15)
```

in this case, 15 decimal places. **ikind** is a new data type – a **parameter**. FORTRAN returns a value to the parameter **ikind** that will be adequate to provide 15 digit precision. This code will work on **any** machine irrespective of the architecture.

We declare that the variables are using extended precision by

```
             real (kind=ikind) :: sum,x
```

Valid values for **p** are 6, 15 and 18. The default value for **p** is 6. If you ask for more precision than 18 digits, the compiler will complain with an error message. Try changing the values of **p** and see what effect this has on the output.

The trouble with PRINT is that the programmer has no control over the number of digits output irrespective of the selected precision .

Later on we'll come back to this when we learn about the WRITE statement, and output formatting.

Note Unlike variables, parameters may not change once they are declared.

If we want to use constants in a program that uses extended precision, we have to tell FORTRAN that they are also extended precision explicitly. This leads to the rather strange syntax you can see in the following program.

```
   program extendedconstants
!demonstrates use of extended precision
   implicit none
   integer, parameter :: ikind=selected_real_kind(p=18)
   real (kind=ikind) :: val,x,y
   val=10/3
   print*,val                  !10/3 calculated as integer  - wrong!
   x=10.0
   y=3.0
   val=x/y              !x/y assigned to extended precision - right!
   print*,val
   val=10.0_ikind/3              !extend precision constant - right!
   print*,val
   val=10.0/3.0                            !real constants - wrong!
   print*,val
   val = .12345678901234567890             !real constants - wrong!
   print *, val
   val = .12345678901234567890_ikind  !ext precision consts - right!
   print *, val
   end program extendedconstants
```

You should run this program for yourself and think carefully about its implications. This program demonstrates how easy it is to get calculations wrong. I'll leave this to you to experiment to ensure that you fully understand the importance of properly declaring variables and the use of constants in FORTRAN programming. A systematic approach to your programming will reduce the risk of errors as will running programs with test data that have known solutions so that you can confirm that your program is error free.

4.5 Magnitude limitations

We have already observed that there is a limitation of the accuracy with which we can do calculations in FORTRAN (and indeed, **any**, computer language). There are also limitations on the magnitude of a number. The various magnitude and precision limits are summarized in the following table:

Value of p	Decimal places	Range
6	**6 (default)**	$\pm 10^{38}$
15	**15**	$\pm 10^{307}$
18	**18**	$\pm 10^{4931}$

Exercise 5.3

To illustrate these limits copy file **magnitude.f95** and run the program. Take a while to get a feel for what is going on. Try inputting various values for the variable maxpower (eg 400). Can you confirm that the table above is correct?

One interesting construct is

```
print *,i,2.0_ikind**i
```

Here, we are telling the compiler that the real constant 2.0 is also using extended precision. Check what happens if you select extended precision (option 3) and enter a value of maxpower of 400. See what happens if you rewrite the line to be

```
print *,i,2.0**i
```

Run the program again and enter the same values. Can you explain what is going on?

4.6 Convergence – exiting loops on a condition

In the program **extended.f95**, we found the sum of

$$\sum_{x=1}^{x=10} \frac{1}{x^6}$$

It is useful to determine at what point such sums converge to a steady value – otherwise we may make arbitrary assumptions about the summation range. Clearly, a point will be reached, within the precision range that can be handled on our computer, that the term

$$\frac{1}{x^6}$$

will be too small to contribute to the sum. At this point we should exit the loop otherwise the program will do more computation than is required.

One way to do this is to compare the value of the variable **sum** with its previous value, and if the difference between the two is very small, then exit the loop.

```
program whileloop
implicit none
integer, parameter :: ikind=selected_real_kind(p=15)
real (kind=ikind) :: sum,previoussum,x,smallnumber,error
integer :: i
sum=0.0
previoussum=0.0
smallnumber = 10.0**(-15.0)
do i=1,1000
     x=i
     sum = sum + 1.0 /(x**6)
     error=abs(sum-previoussum)
     if (error<smallnumber) then
          print *,'sum ',sum,' number of loops ',i
          exit
     end if
     previoussum = sum
end do
end program whileloop
```

IMPORTANT NOTE
In the real world, we have to make choices about the amount of precision we need to work to. It is pointless asking for 15 digits of precision if, for example, we can only take a measurement to + or – 1% accuracy!

It is not **necessary** to always use a loop counter in a **do loop.** If we don't actually specify a counter, the program will loop forever. Constructs like this are OK:

```
smallnumber = .0000001_ikind
do
      print *, 'enter a positive number '
       read *, number
       if (number <= smallnumber) exit
 end do
```

The disadvantage is that, if you get the code wrong, you run the risk of the program looping forever – generally it's safer to use a loop counter!

5 Arrays and Formatted I/O

5.1 Aims

By the end of this worksheet you will be able to:

❑ Understand the use of arrays
❑ Improve the appearance of your output

5.2 Arrays

Let us imagine that we want to find the average of 10 numbers. One (crude) method is shown in the next program.

```
program av
real :: x1,x2,x3,x4,x5,x6,x7,x8,x9,x10,average
read *, x1,x2,x3,x4,x5,x6,x7,x8,x9,x10
average= (x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10)/10
print *, 'the average is ',average
print *, 'the numbers are:'
print *, x1
print *, x2
print *, x3
print *, x4
print *, x5
print *, x6
print *, x7
print *, x8
print *, x9
print *, x10
end program av
```

This approach is messy, involves a lot of typing and is prone to error. Imagine if we had to deal with thousands of numbers!

The way around this is to use **arrays**. An array is a list that we can access through a subscript. To indicate to FORTRAN that we are using an array, we just specify its size when we declare it.

```
real, dimension(100)  ::x
    .
    .
x(1) = 3
x(66) = 4
```

This snippet of code allocates 100 memory locations to the array **x**. To access an individual location, called an **array element**, we use a **subscript** – here we are assigning the number 4 to the 66th element of array x and 3 to the 1st element.

Now let's return to program **av** at the start of this worksheet, we'll re-write it using an array.

```
        program av2
        implicit none
        real ,dimension(10) :: x
        real                :: average,sum
        integer             :: i
        print *, 'enter 10 numbers'
        sum=0.0
        do i=1,10
             read *, x(i)
             sum=sum+x(i)
        end do
        average=sum/10
        print *, 'the average is ',average
        print *, 'the numbers are'
        print *,x
        end program av2
```

Notice that if we type

```
        print*, x
```

the program will print out the entire contents of the array.

The additional benefit of this program is that with very few changes, we could make it deal with any number of items in our list. We can improve on this still further by making use the **parameter** data type:

```
        program av3
!just change the value of the parameter to change the size of the
!array
        implicit none
        integer, parameter    :: imax = 10
        real,dimension(imax)  :: x
        real                  :: average,sum
        integer               :: i
        print *, 'enter’ ,imax, ’ numbers'
        sum=0.0
        do i=1,imax
             read *, x(i)
             sum=sum+x(i)
        end do
        average=sum/imax
        print *, 'the average is ',average
        print *, 'the numbers are'
        print *,x
        end program av3
```

Note this is an example of good programming. The code is easily maintainable – all we have to do to find an average of a list of numbers of *any* size is just to change the size of the parameter **imax**. We can also allocate the size of the array at **run time** by dynamically allocating memory.

The following program demonstrates the use of arrays where we do not know the size of the array.

```
program alloc
implicit none
integer, allocatable,dimension(:):: vector
!note syntax - dimension(:)
integer                           :: elements,i
print *,'enter the number of elements in the vector'

read *,elements
allocate(vector(elements))
!allocates the correct amount of memory
print *,' your vector is of size ',elements,'. Now enter each
element'
do i=1,elements
   read *,vector(i)
end do
print *,'This is your vector'

do i=1,elements
   print *,vector(i)
end do

deallocate(vector)
!tidies up the memory

end program alloc
```

The program is called alloc.f95 and can be copied from the web page. Note in particular the bolded lines. The new way of declaring the array **vector** tells the compiler that it is allocatable – ie the size will be determined at **run time.**

We shall look at this further in Section 7.

Exercise 5.1

Write a program that asks the user how many numbers they want to enter, call this value **imax**. Allocate **imax** elements to two arrays, **a** and **b**. Read in **imax** numbers to **a** and do the same to **b.** Print out the arrays **a, b** and print out the sum of **a** and **b**. Compare your attempt with sumalloc.f95.

5.3 Array magic

One of the benefits of arrays is that you can easily do operations on every element by using simple arithmetic operators.

```
program ramagic
implicit none
real ,dimension(100) :: a,b,c,d
open(10,file='data.txt')
read(10,*) a
b=a*10
c=b-a
```

```
        d=1
        print *, 'a= ',a
        print *, 'b= ',b
        print *, 'c= ',c
        print *, 'd= ',d
        end program ramagic
```

Exercise 5.2

Copy program **ramagic.f95** and file **data.txt** to your own filespace. Run the program and examine the output.

Exercise 5.3

Write a program that fills a 10 element array **x** with values between 0 and .9 in steps of .1. Print the values of sin(x) and cos(x) using the properties of arrays to simplify your program. Compare your answer with ramagic2.f95.

5.4 Multi dimensional arrays

The arrays we have looked at so far have been **one dimensional**, that is a single list of numbers that are accessed using a single subscript. In concept, 1 dimensional arrays work in a similar way to vectors. We can also use two dimensional arrays which conceptually are equivalent to matrices.

So, for example,

Integer, dimension(5,5) :: a

sets up a storage space with 25 integer locations.

The next program creates a 2 dimensional array with 2 rows and 3 columns. It fills all locations in column 1 with 1, columns 2 with 2, column 3 with 3 and so on.

```
        program twodra
        implicit none
        integer,dimension(2,3)      :: a
        integer                     ::row,col,count
        count = 0
!creates an array with 3 cols and 2 rows
!sets col 1 to 1, col2 to 2 and so on
        do row=1,2
           count=0
           do col =1,3
                count=count+1
                a(row,col)=count
           end do
        end do
        do row=1,2
             do col =1,3
                  print *,a(row,col)
             end do
        end do
        end program twodra
```

FORTRAN actually allows the use of arrays of up to 7 dimensions, a feature which is rarely needed. To specify a extended precision 3 dimensional array b with subscripts ranging from 1 to 10, 1 to 20 and 1 to 30 we would write:

```
real  (kind=ikind),dimension(10,20,30) :: b
```

Exercise 5.4

Using a 4*4 array create an identity matrix, that is, a matrix of the form:

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

and output it. **Wouldn't it be nice if we could actually output the matrix elements in rows and columns?** At the end of this section we shall see exactly how to do this.

5.5 Formatting your output

You may now be wondering if there is any way to have better control over what your output looks like. So far we have been using the default output option – that's what the *'s are for in the **write** and **print** statements:

```
write(10,*) x,y,z
print *, 'program finished'
```

Exercise 5.5

Copy **format.f95,** and run it

```
      program format
      implicit none
!demonstrates use of the format statement
      integer, parameter :: ikind=selected_real_kind(p=15)
      real , dimension(4)             :: x
      integer, dimension(4)           :: nums
   integer                            :: i
      real(kind=ikind),dimension(4)   :: computed
!fill up the arrays with something
      do i = 1,4
          nums(i)          = i * 10
          computed(i)      = cos(0.1*i)
          x(i)             = computed(i)
      end do
      print *,'nums - integer'
      write(*,1) nums
1     format(2i10)
      print *, 'x - real'
      write(*,2) x
2     format(f6.2)
      print *, 'computed - double precision'
      write(*,3) computed
3     format(f20.7)
```

```
                    end program format
```

You can see that the **write** and **format** statements come in pairs.

```
        write(output device,label) variable(s)
label   format(specification)
```

We are using in this example a * as the output device – in other words, the screen.

The **format** statement can actually go anywhere in the program, but by convention we usually place them just after the associated **write** or all together at the end of the program. It's just a matter of taste.

The tricky part here is the **specification**. There are different specifications for integer, real, and character variables.

5.5.1 Integer Specification

General form : n**i**m

- ❑ Right justified
- ❑ m is the number of character spaces reserved for printing (including the sign if there is one)
- ❑ If the actual width is less than m, blanks are printed
- ❑ n is the number of integers to output per line. If omitted, one number is output per line.

5.5.2 Floating point Specification

General form : n**f**m.d

- ❑ Right justified
- ❑ m is the number of character spaces reserved for printing (including the sign if there is one), and the decimal point.
- ❑ If the actual width is less than m, blanks are printed
- ❑ n is the number of real numbers to output per line. If omitted, one number is output per line.
- ❑ d is the number of spaces reserved for the fractional part of the number – filled with 0's if fewer spaces are needed. If the fractional part is too wide it is rounded.

If the total width for output (m) is too small, FORTRAN will just output *'s.

Rule m >= width of the integer part **plus**
d **plus**
1 (space for decimal point) **plus**
1 (space for sign – if negative)

Essentially, make m nice and wide and you won't have any trouble!

5.5.3 Exponential Specification

General form nEm.d

- ❑ Alternative specification for outputting **real**
- ❑ **d** is the number of decimal places
- ❑ **m** is the total width of the field including the sign (if any), the character E and its sign, the decimal point and the number of places of decimals. Again make **m** nice and wide to ensure the field is properly printed out.
- ❑ n is the number of exponential numbers to output per line. If omitted, one number is output per line.

Example

```
            real :: a,b
            a = sqrt(5.0)
            b = -sqrt(a)
            write(*,10) a,b
        10  format(2E14.5)
```

produces:

```
        0.22361E+01  -0.14953E+01
```

5.5.4 Character Specification

General form nAm

- n is the number of strings to print
- m is the maximum number of characters to output

Example:

```
              program chars
              implicit none
              character ::a*10,b*10
              a='hello'
              b='goodbye'
              write(*,10) a,b
        10    format(2a10)
              end program chars
```

Exercise 5.6

Using the format specifications in **format.f95** as a guide, produce a table of
x e^x
where $0 \leq x \leq 1$, for values of x in increments of 0.1. Write your output to a file called myoutput. Ensure that your output lines up neatly in columns. An example program is neatoutput.f95 is available on the website.

5.6 Implied Do Loop to write arrays

So far, the method we have used for input and output of arrays is:

```
           integer :: col,row
           real :: ra(10,10)
        !using do loop
           do row = 1,10
               do col = 1,10
                  read *,       ra(row,col)
                  write(*,*)    ra(row,col)
               end do
            end do
```

The trouble with this method is that the rows and columns are not preserved on output. An alternative, and neater method is to use **an implied do loop** in the write statement.

```
     real :: ra(10,10)
     integer :: row,col
!use implied do
     do row = 1,10
       do col = 1,10
          read *,        ra(row,col)
       end do
    end do
    do row=1,10
       write(*,10) (ra(row,col),col=1,10)
    end do
10  format(10f5.1)
```

Exercise 5.7

In Exercise 5.4 you wrote a program to produce and identity matrix. Apply what you know about formatting now to make a neatly formatted matrix onscreen. There is an example identity1.f95 available on the website.

6 Subroutines and Functions

6.1 *Aims*

By the end of this worksheet you will be able to:

- ❑ Understand the use of subroutines and functions to make your code more efficient and easier to read.

6.2 *Re-using code – the subroutine*

Examine the following program

```
 program output
 implicit none
 real,dimension(3) :: a,b,c
 character :: answer*1
!initialise arrays
 a = 1.5
 b = 2.5
 c = 3.5
 write(*,1) 'a',a
 print *, 'type y to continue or any other key to finish'
 read *, answer
 if (answer /= 'y') stop
 write(*,1) 'b',b
 print *, 'type y to continue or any other key to finish'
 read *, answer
 if (answer /= 'y') stop
 write(*,1) 'c',c
 print *, 'type y to continue or any other key to finish'
 read *, answer
 if (answer /= 'y') stop
 write(*,1) 'a*b*c',a * b * c
1 format(a,3f8.3)
 end program output
```

The program sets up some arrays and then outputs them. At three stages in the program (bolded), it asks whether it should continue; it stops if the answer is not 'y'. Notice that the three bolded parts of the code are *identical.*

Simple enough – but look at the amount of code! Most of it is the same – wouldn't it be nice to re-use the code and cut down on the typing? The answer is to use **subroutines**.

```
        program output1
        implicit none
        real,dimension(3) :: a,b,c
!initialise arrays
        a = 1.5
        b = 2.5
        c = 3.5
        write(*,1) 'a',a
        call prompt()
        write(*,1) 'b',b
        call prompt()
        write(*,1) 'c',c
        call prompt()
        write(*,1) 'a*b*c',a * b * c
1       format(a,3f8.3)
        end program output1
!++++++++++++++++++++++++++++++++++++++++++++
        subroutine prompt()
!prompts for a keypress
        implicit none
        character answer*1
        print *, 'type y to continue or any other key to finish'
        read *, answer
        if (answer /= 'y') stop
        end subroutine prompt
```

Examine the code, each time we use type

call prompt()

the program jumps to the line

subroutine prompt()

then executes each line of the code it finds in the subroutine until it reaches the line

end subroutine prompt

and then returns to the main program and carries on where it left off.
The program is much easier to understand now. All the code for prompting is in one place. If we ever need to change the code which prompts the user to continue, we will only ever need to change it once. This makes the program more **maintainable.**

6.3 Arguments to subroutines

We have seen that subroutines are very useful where we need to execute the same bit of code repeatedly.

The subroutine can be thought of as a separate program which we can call on whenever we wish to do a specific task. It is independent of the main program – it knows nothing about the variables used in the main program. Also, the main program knows nothing about the variables used in the subroutine. This can be useful – we can write a subroutine using any variable names we wish and we know that they will not interfere with anything we have already set up in the main program.

This immediately poses a problem – what if we **want** the subroutine to do calculations for us that we can use in the main program? The following program uses **arguments** to do just that.

Example: a program that calculates the difference in volume between 2 spheres.

```
        program vols
!Calculates difference in volume of 2 spheres
        implicit none
        real :: rad1,rad2,vol1,vol2
     character :: response
     do
        print *, 'Please enter the two radii'
        read *, rad1,rad2
        call volume(rad1,vol1)
        call volume(rad2,vol2)
        write(*,10) 'The difference in volumes is, ',abs(vol1-vol2)
10      format(a,2f10.3)
        print *, 'Any more? - hit Y for yes, otherwise hit any key'
        read *, response
        if (response /= 'Y' .and. response /= 'y') stop
     end do
     end program vols
!________________________________________________
     subroutine volume(rad,vol)
     implicit none
     real :: rad,vol,pi
!calculates the volume of a sphere
     pi=4.0*atan(1.0)
     vol=4./3.*pi*rad*rad*rad
!It's a little quicker in processing to do r*r*r than r**3!
     end subroutine volume
```

When the program reaches the lines

```
        call volume(rad1,vol1)
```

It jumps to the line

```
        subroutine volume(rad,vol)
```

The values, **rad1** and **vol1** are passed to the subroutine. The subroutine calculates a value for the volume and when the line :

```
        end subroutine volume
```

is reached, the value of the volume is returned to the main program

Points to notice – these are very important – please read carefully

- You may have several subroutines in your program. Ideally, a subroutine should do a specific task – reflected by its name.
- All the variables in subroutines, apart from the ones passed as arguments, are 'hidden' from the main program. That means that you can use the same names in your subroutine as in the main program and the values stored in each will be unaffected – unless the variable is passed as an argument to the subroutine.
- It is very easy to forget to declare variables in subroutines. Always use **implicit none** in your subroutines to guard against this.

- All the variables in the subroutine **must** be declared.
- The positioning of the arguments (in this case, rad and vol) is **important**. The subroutine has no knowledge of what the variables are called in the main program. It is **vital** that the arguments agree both in **position** and **type**. So, if an argument to the subroutine is **real** in the main program, it must also be **real** in the subroutine.
- If an argument to the subroutine is an array, it must also be declared as an array in the subroutine.

Exercise 6.1

Write a program that calculates the difference in area between two triangles. Your program should prompt the user for the information it needs to do the calculation. Use a subroutine to calculate the actual area. Pass information to the subroutine using arguments.

6.4 User Defined Functions

We have already met FORTRAN **intrinsic functions** like **abs, cos, sqrt**. We can also define our own functions – they work in a similar way to **subroutines**.

As an example, let's write a program (**func.f95**) that does some trigonometry. As you know, the trig routines in FORTRAN use radians, not degrees - so it would be nice to write a function that does all the conversion for us.

```
print *,'Enter a number'
read *, a
pi=4.0*atan(1.0)
print *,'the sine of ',a,' is ',sin(a*pi/180)
```

In this snippet, we are having to code the conversion from degrees to radians directly into the main part of the program. That's OK for a 'one-off', but what if we needed to do the conversion several times. Now look at this:

```
     program func
!demonstrates use of user defined functions
     implicit none
     integer, parameter :: ikind=selected_real_kind(p=15)
     real (kind=ikind):: deg,rads
     print *, 'Enter an angle in degrees'
     read  *, deg
     write(*,10) 'sin = ',sin(rads(deg))
     write(*,10) 'tan = ',tan(rads(deg))
     write(*,10) 'cos = ',cos(rads(deg))
10   format(a,f10.8)
     end program func
!_____________________________________________
     function rads(degrees)
     implicit none
     integer, parameter :: ikind=selected_real_kind(p=15)
!    returns radians
     real (kind=ikind) :: pi,degrees,rads
     pi=4.0_ikind*atan(1.0_ikind)
     rads=(degrees*pi/180.0_ikind)
     end function rads
```

What we have done, in effect, is to create our own function **rads**, which is used in an identical way to the intrinsic ones you have used already like **sqrt**, **cos**, and **abs**.

When the line

```
write(*,10) 'sin = ',sin(rads(deg))
```

is reached, the program jumps to

```
function rads(degrees)
```

the value, **degrees**, is passed to the function. The function does some computation, then finally returns the calculated value to the main program with the line

```
rads=(degrees*pi/180.0_ikind)
```

Note carefully that it doesn't return the value in the argument list (as does a subroutine) but actually **assigns** the value to its own name **rads**.

- The function **rads** converts the value of the argument, degrees, to radians.
- Notice that we must declare the data type of the function both in the main program, and in the function itself **as if it were a variable**.
- Functions return **one** value. This value, when calculated, is assigned to the name of the function as if it were a variable –
  rads=(degrees*pi/180.0_ikind)

Exercise 6.2

Write a program that includes a function called

real function average(n,list)

where **n** is integer and is the number of items in the **list**, and **list** is a **real array**.

Write suitable code for reading the numbers from a file (or keyboard), and output the average of the numbers.

Exercise 6.3

Write a program that allows a user to enter the size of a square matrix. In the program write a subroutine to compute a **finite difference matrix**. Ensure your output is neatly formatted in rows and columns.

So, for a 10 by 10 matrix, we expect output to look like this

```
  2 -1  0  0  0  0  0  0  0  0
 -1  2 -1  0  0  0  0  0  0  0
  0 -1  2 -1  0  0  0  0  0  0
  0  0 -1  2 -1  0  0  0  0  0
  0  0  0 -1  2 -1  0  0  0  0
  0  0  0  0 -1  2 -1  0  0  0
  0  0  0  0  0 -1  2 -1  0  0
  0  0  0  0  0  0 -1  2 -1  0
  0  0  0  0  0  0  0 -1  2 -1
  0  0  0  0  0  0  0  0 -1  2
```

Check your attempt with finite.diffs.f95 on the website.

7 Advanced Topics

7.1 Aims

By the end of this worksheet you will be able to:

- ❑ Use array functions
- ❑ Create larger programs aided by "Flow Charts"

7.2 Array Functions

FORTRAN provides a number of intrinsic functions that are useful for working with arrays. Among these are some which are specifically aimed at working with matrices and vectors.

MATMUL	**Matrix/vector**	**Matrix multiplication of two matrices or a matrix and a vector.**
DOT_PRODUCT	**Vector**	**Scalar (dot) product of two vectors**
TRANSPOSE	**Matrix**	**Transpose of a matrix**
MAXVAL	**Any array**	**Maximum value of an array, or of all the elements along a specified dimension of an array.**
MINVAL	**Any array**	**Minimum value of an array, or of all the elements along a specified dimension of an array.**
SUM	**Any array**	**Sum of all the elements of an array, or of all the elements along a specified dimension of an array.**

Program **matrixmul.f95**, demonstrates the use of these functions. Additionally, it includes two subroutines that are likely to be useful when handling matrix/array manipulations: **fill_array** which fills the array elements and **outputra** which prints the values of the array elements to the screen. This program is also an example of **dynamic memory allocation**.

```
program matrixmul
!demonstrates use of matmul array function and dynamic
!allocation of array
     real,  allocatable, dimension(:,:) :: ra1,ra2,ra3
     integer                            :: size
!initialize the arrays
     print*, 'Shows array manipulation using SQUARE arrays.'
     print*, 'Allocate the space for the array at run time.'
     print*, 'Enter the size of your array'
     read *, size
     allocate(ra1(size,size),ra2(size,size),ra3(size,size))
     print*, 'enter matrix elements for ra1 row by row'
     call fill_array(size,ra1)
     print*, 'enter matrix elements for ra2 row by row'
     call fill_array(size,ra2)
!echo the arrays
     print *,'ra1'
      call outputra(size,ra1)
      print *,'ra2'
      call outputra(size,ra2)
!demonstrate the use of matmul and transpose intrinsic
!functions
      ra3=matmul(ra1,ra2)
      print *,'matmul of ra1 and ra2'
      call outputra(size,ra3)
      ra3=transpose(ra1)
      print *,'transpose of ra1'
      call outputra(size,ra3)
      deallocate(ra1,ra2,ra3)
      end program matrixmul
!---------------------------------------------------------
      subroutine outputra(size,ra)
      implicit none
!will output a real square array nicely
      integer                                :: size,row,col
      real,dimension(size,size)              :: ra
      character                              :: reply*1
      do row =1,size
         write(*,10) (ra(row,col),col=1,size)
10    format(100f10.2)
!as we don't know how many numbers are to be output, specify
!more than we need - the rest are ignored
      end do

print*,'__________________________________________________'
      print*,'Hit a key and press enter to continue'
      read *,reply
      end subroutine outputra
!---------------------------------------------------------
      subroutine fill_array(size,ra)
      implicit none
!fills the array by prompting from keyboard
      integer                             :: row,col,size
      real                                :: num
      real, dimension(size,size)          :: ra
      do row=1,size
         do col=1,size
         print *, row,col
         read *,num
         ra(row,col)=num
         end do
      end do
      end subroutine fill_array
```

Exercise 7.1

Write a program to read in 2 square matrices (of any size). Confirm that the matrices obey the rule

$(AB)^T = B^T A^T$

where A^T is the transpose of matrix A.

Exercise 7.2

Write a program that will read a 3 X 3 matrix from a data file. In the program, include a subroutine that will generate any cofactor **cof** of the matrix **mat**. Call the subroutine cofactor and use these arguments:

```
subroutine cofactor(i,j,mat,cof)
implicit none
real :: mat(3,3),minor(2,2),cof
integer :: elrow,elcol
! cof – the cofactor of matrix mat for element i,j
.
.
```

Exercise 7.3

Use the program you developed Exercise 7.2 to calculate the **determinant** of a 3 X 3 matrix.

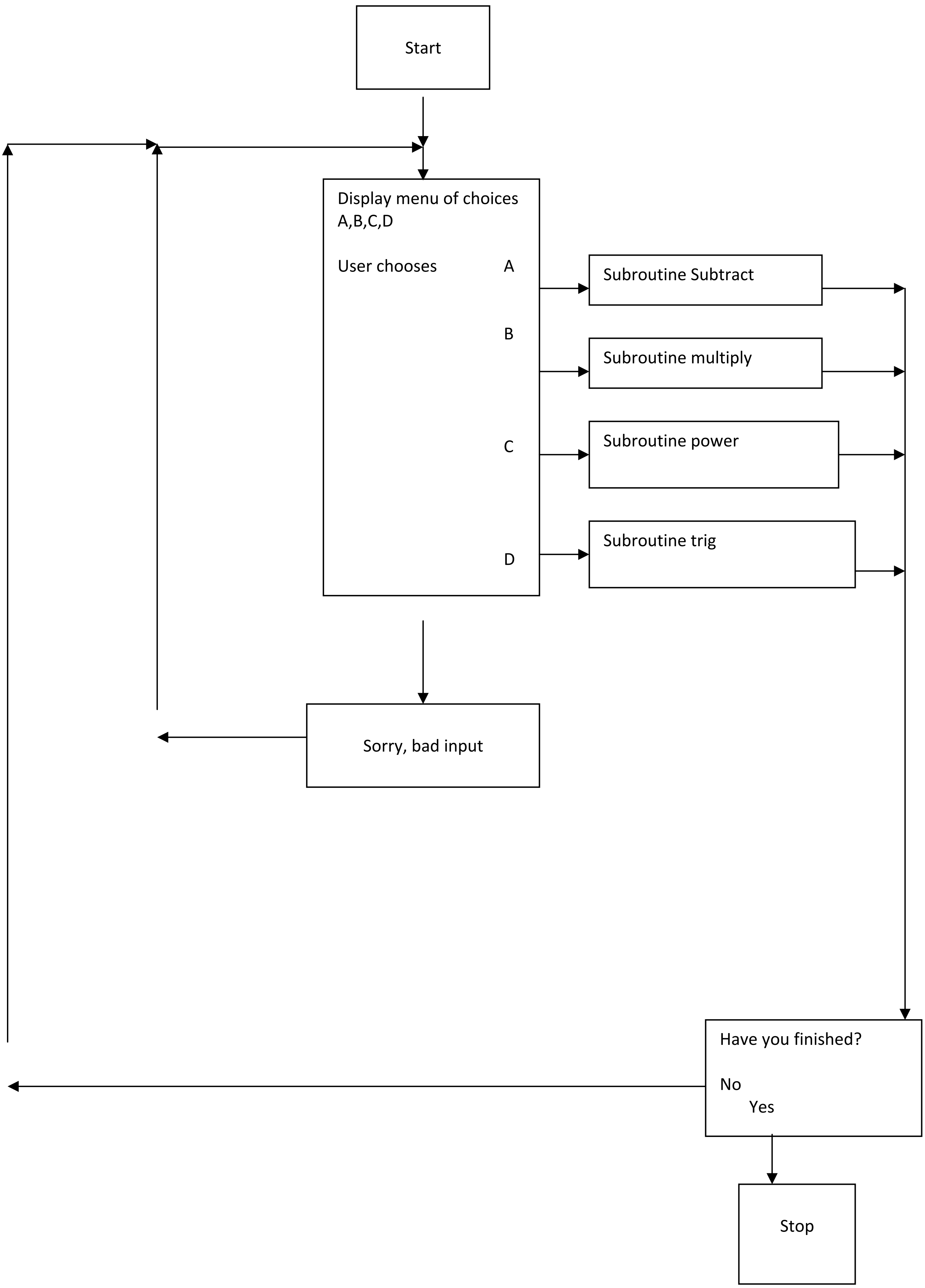
7.3 Writing REAL programs - Flow Charts

Now that you know all the main elements of FORTRAN 95, you are in a position to apply your skills to writing REAL programs. Unlike most of the exercises in these worksheets, REAL programs tend to be rather large. In large programs, the underlying logic can often be difficult to follow.

It helps, therefore, both in the devising of a program and later in its maintenance, to have a plan of what you intend the program to do. Let’s take, as an example, a program that works like a calculator.

The flowchart is shown on the next page. The logic of the program, as a whole, is clear. Details like what will happen in the subroutines is glossed over at this stage.

In commercial programming, flowcharts are usually formalized, with specific shapes for boxes that do different things. That need not concern us here. Essentially, we use flowcharts to provide a ‘map’ of the underlying logic flow in the program – what connects with what.


Start
Display menu of choices A,B,C,D
User chooses
A
B
C
D
Subroutine Subtract
Subroutine multiply
Subroutine power
Subroutine trig
Sorry, bad input
Have you finished?
No
Yes
Stop

De-bugging Tips

7.4 *Symptoms and probable causes*

Have you got rounding errors?

- Don't do floating point calculations using integers. Make sure your precision is consistent and adequate. Make sure the precision of the right hand side of an equation matches the type of variable you are assigning to.

Are your calculations completely wrong?

- Initialise all your variables – don't forget arrays!
- Make sure your arrays are big enough to hold all the data.
- Check that arguments passed to subroutines agree exactly in size, type and position
- Is the program's logic working the way it should?

7.5 *Wise precautions and time saving tips*

- You must **not** test floating point numbers for equality. Example:

```
if (x == 1) then...
```

does not work.

- Should you be using the absolute value of a calculation? Example:

```
if   (abs(x-y)<.00001) then
```

- Don't have overly elaborate logical tests. It's probably better to test one or two things at a time rather than this sort of thing…

```
if (((x.AND.y).OR.z > 10).OR.(.NOT. xx < 0))  then …
```

you might think you understood it when you wrote it, but imagine trying to figure out what's happening if the program breaks!

- **Don't try and write a complicated program all at once. Write it a piece at a time and check that each piece is working correctly before doing the next bit.**
- Use 'implicit none' at the start of all programs and subroutines.
- If your program needs data to be entered by the user, you will save hours of time by taking the trouble to read in the data from a file while the program is in development.
- Always do a test for 'division by zero' when dividing.
- **BE NEAT!** Good programming is like plain speaking – there's no mileage in tricky, difficult to read code.

7.6 *How to find the bugs*

Use a print statement to print out the values within the program – take a look at this code…

```
x   = x + 1
z   = x * y
print  *, ‘debug statement 1 value of x,y,z‘, x,y,z
do  ii  =1,10
   z = x *  ii
   if (ii == 5) then
     print *, ‘debug do loop value of z when ii =  5’,z
   end if
end do
if (z>2000) then
   print *, ‘debug statement – z>2000  value of z ‘,z
   stop
end if
```

Notice how we can print out values of specific variables, stopping the program if necessary.

Index

Arithmetic 11
arithmetic operators 11
Arrays 27
Assignment 11
character 8
Character Specification 33
Convergence 25
Debugging Tips 44
De-bugging Tips 44
dot product 40
dynamic memory allocation 40
dynamically allocating memory 28
errors 8
Exercise 1.1 4
Exercise 1.2 6
Exercise 1.3 7
Exercise 1.4 9
Exercise 2.1 11
Exercise 2.2 12
Exercise 2.3 14
Exercise 2.4 15
Exercise 3.1 17
Exercise 3.2 18
Exercise 3.3 18
Exercise 3.4 19
Exercise 3.5 20
Exercise 3.6 21
Exercise 3.7 21
Exercise 4.1 22
Exercise 4.2 23
Exercise 5.1 29
Exercise 5.2 30
Exercise 5.3 25, 30
Exercise 5.4 31
Exercise 5.5 31
Exercise 5.6 33
Exercise 5.7 34
Exercise 6.1 38
Exercise 6.2 39
Exercise 6.3 39
Exercise 7.1 42
Exercise 7.2 42
Exercise 7.3 42
exiting loops 25
Exponential Specification 32
exponentiations 11
finite difference 39
floating point numbers 8
Floating point Specification 32
Flow Charts 40, 42
Formatting your output 31
function 39
identity 31
ikind 26
Implied Do Loop to write arrays 33
integer 8
Integer Specification 32
intrinsic functions 11, 12, 40
logical operators 14
loop 18
loop counter 19
Magnitude 25
maintainable 36
matrix 39
Matrix multiplication 40
Mixing variable types 17
Multi dimensional arrays 30
Nested Do Loops 19
operators 6
parameter 24
Plato 3
precedence 11
precision 8, 23
programming style 11
Reading from files 22
simple if statement 15
statement 6
stop 15
Style 14
subroutine 35
summation 20
Transpose of a matrix 40
User Defined Functions 38
variable 6
Writing to files 23