Loop Handout

This document is to provide an alternative look at the concept of a 'loop'. It will be focused on the use of loops within Fortran 90, but the underlying ideas are applicable to loops in essentially any language. In order to analyse how a loop works, we will look at its application to the mathematical structures of arithmetic progressions and geometric progressions.

Basic Structure

The basic structure of a loop, as also described in the course booklet is as follows:

```
DO counter_variable = start, finish, step
  !loop contents
END DO
```

Where '`counter_variable`', should be an `INTEGER` variable and `start`, `finish` and `step` are values for what value the counter should start at, what it will finish at/before and the value to increment the counter between each iteration.

So, we could have the following loop which counts to ten, writing the numbers into the terminal. After the code, the way the code will actually be executed is described, pay close attention to it, especially its linear nature. Everything in computing as covered in this module is entirely linear, statements are performed in fixed order as they are written in your source code, one after another.

```
INTEGER :: i

DO i = 1, 10, 1
  WRITE(*,*) i
END DO

WRITE(*,*) 'Finished'
```

1. First, we define the `INTEGER` variable to be used as our counter; `i`, `j` and `k` are the most commonly used names for counters.
2. The loop itself begins on line 3, when the code reaches this point it initialises (gives an initial value to) the variable `i`, in this case the value `1`.
3. It then proceeds to the contents of the loop, the indented portion between the `DO` and `END DO`, in this case only containing a single `WRITE` statement. This `WRITE` statement will cause the current value of `i` to be output, it will still be `1` at this point.
4. The code then moves to the `END DO` statement, this instructs the program to jump back to the point at which the `DO` loop began, the commands on line 3.
5. As the loop has already begun, `i` is known to have a value, this value is then shifted to the next one in the chain. In this case, because the increment is `1`, `i` has `1` added to it (resulting in `2`) and it is then checked to see whether it has exceeded the upper limit, `10` in this case.
6. As it has not exceeded `10` yet, the contents of the `DO` loop is executed by the code again, this time writing `i`'s new value of '`2`' to the terminal.
7. Reaching `END DO` again, the process repeats as before, incrementing `i` by the required amount, seeing if it exceeds the limit, and then executing the contents of the loop if it does not.
8. Once `i` reaches `10`, the contents of the loop is run one more time, writing `10` to the terminal, and then jumping back to line 3 upon reaching the `END DO`. It increments the counter, `i`, by one, resulting in `11`; being higher than `10`, the loop then knows that it must stop. The program jumps back down to the `END DO` line and continues on to the rest of the program from there. In this case, finally executing the `WRITE` statement placing '`Finished`' to the terminal.

It is of vital importance to understand loops executing in this step-by-step fashion, such that you can appreciate what effect any commands placed either inside or outside of the loop will have when the loop runs.

Arithmetic Progressions

We will now look at a simple application of loops to a mathematical problem. Arithmetic progressions describe a sequence of numbers for which the step between each number is an addition/subtraction (sometimes as a function of the index of the number being calculated). A simple arithmetic progression could therefore be phrased as follows.

$$a_{n+1} = a_n + b \tag{1}$$

The 'as' are the numbers of the sequence, the subscript indicates the index of that value for a (eg a_0 is the initial value a_1 is the first value calculated, a_2 the second value calculated, etc). b is the increment between values in the sequence. What this equation tells us is that the 'next' value along $(n + 1)$, from any particular value n^{th} in the sequence, will be the current value, a_n, plus the increment, b. Use of this indexed notation for elements in a series is very commonly used in multiple areas of computing, mathematics and physics.

So, if $b = 3$ and the current value of a, a_n is 8; the next element along will be $a_{n+1} = a_n + b = 8 + 3 = 11$. This can be directly expressed in a loop. You may notice that the example in the previous section is equivalent to a progression where the starting value, a_0 is 1 and the increment b is also 1. Were we to change the value of the `increment` parameter (the third value on the `DO` loop line), it would be equivilent to changing b.

For a loop defined with `DO i = 1, 10, 2`, the values of `i` would be given by the arithmetic progression $\texttt{i}_{n+1} = \texttt{i}_n + 2$, with the initial value $\texttt{i}_0$ being the start value for the loop, 1.

What happens if we wanted to find the first twenty values of a sequence $a_{n+1} = a_n + 6$, with $a_0 = 19$? We could construct a loop where the starting value was 19, and the increment 6, but then what do we set the final value as? We would have to perform a calculation on the loop first, to determine how large the 20th value was!

To circumvent this we can, instead of having the variable `i` represent the value itself, a, represent the index, n. This could be accomplished by the following code (with suitable `PROGRAM` statements, etc.).

```
INTEGER :: i, a

a = 19

DO i = 1, 20, 1
  a = a + 6
  WRITE(*,*) a
END DO
```

In this case, our loop goes from 1 to 20 in steps of 1, as we want every of the values with indices 1 to 20. The variable `a` is set equal to 19 before the loop, this sets our initial $a_0 = 19$ criteria.

Within the loop, we have the statement `a = a + 6`. Depending on how you imagine this is evaluated, it may seem odd. What is important to remember is that, in Fortran, the entire right-hand-side of an assignment is calculated, before the left hand side is set equal to it. So, the `a` to the right of the equals represents our existing value of a, a_n, this has six added to it and the variable `a` is then given this new value. This means that the `a` to the left of the equals represents the 'next' value of a, a_{n+1}.

This can sometimes be difficult to appreciate, as it may be instinctive to interpret the assignment purely mathematically. Seeing it like this results in a statement which apparently makes no sense. If $a = a + 6$, there can exist no value of a which satisfies this as an equation. This is why it is important to realise that the left-hand-side of the statement is completely evaluated, before anything is done in relation to the left-hand-side. It is as though there is an additional 'hidden' variable (we'll call it h in this example) making the code the following:

```
h = a + 6
a = h
```

So, `a+6` is evaluated first, then stored. After that, the variable `a` is assigned its new value, `h`, which happens to be its previous value, plus six. This process of being able to assign a variable a new value based on some function of its old value is exceptionally useful.

It is worth noting that arithmetic progressions are very well defined mathematically. It is easily possible to determine the value at an arbitrary point in the sequence with an equation. It is primarily for the utility of the concepts in loops that we use them as an example; a more suitable way to find particular elements of a simple sequence is mathematically. However, complex patterns and sequences may be difficult to solve, or even unsolvable, at which point loop structures in programming can be an ideal way to further examine such problems. An arbitrary value within an arithmetic progression can be given by,

$$a_n = a_0 + n\,b \tag{2}$$

where a_0 is the initial value, and n is the index of the value in the sequence desired to be found. For example, in the case of $a_0 = 19$ and $b = 6$ given earlier, the 15th value in the sequence is found to be 109. This is also given by the previous formula,

$$a_{15} = a_0 + 15 \times b = 19 + 15 \times 6 = 109. \tag{3}$$

Practice

These practice problems are optional, but recommended to try if you are having difficulty understanding the loop concept. Solutions will be made available in a separate document.

1. Write a code which would output values a_1 to a_{15} of the progression $a_{n+1} = a_n + 42$, where $a_0 = 4$.
2. Write a code which would output values a_3 to a_6 of the progression $a_{n+1} = a_n + 8$, where $a_0 = 3$, this will require an `IF` statement to output correctly.
3. In either of the previous two practices, or the examples described at the end of the previous section, where the counter `i` is used as the index n of the progression, not the a value in the progression itself (a); if we were to alter the increment of the `DO` loop to be `2`, rather than `1`, would the results still be correct? Give reasoning for your answer, and consider what exactly would be happening in the code.

Geometric Progressions

In contrast to the progression described in the previous section, we can define a different type of sequence, one reliant on multiplication rather than addition. This is a geometric progression, and the next value in a sequence a_{n+1} based on the current value a_n can be expressed as,

$$a_{n+1} = m \times a_n \tag{4}$$

where m is similar to the increment b in the arithmetic progression. In this case, however, the current value is multiplied with, rather than added to, this parameter.

The first several values of a sequence where $m = 3$ and $a_0 = 2$, would therefore be $a_1 = 3 \times 2 = 6$, $a_2 = 3 \times 6 = 18$, $a_3 = 3 \times 18 = 54$... and so on. The code that would produce this sequence for the first 10 values, created by exactly the same concept as for the arithmetic progression, could be as follows.

```fortran
INTEGER :: i, a

a = 2

DO i = 1, 10, 1
  a = a * 3
  WRITE(*,*) a
END DO
```

Entirely identical to the format of the arithmetic progression code, but with the assignment `a = a * [value]`, rather than `a = a + [value]` (where `[value]` is whatever value was defined for the progression in question). So, we can now easily produce a sequence of values from a geometric progression, given a particular starting value a_0. As with the arithmetic progression, an equation can be found giving the value at any arbitrary index mathematically, this equation is,

$$a_n = a_0\, r^n. \tag{5}$$

This is also a straightforward way to determine what any particular value in a sequence happens to be. This equation had to be determined first, however, which means that for other unusual sequences, such an equation may not already exist, or may not be readily available.

Practice

These practice problems are optional, but recommended to try if you are having difficulty understanding the loop concept. Solutions will be made available in a separate document.

1. Write a code which would output values a_1 to a_5 of the progression $a_{n+1} = 10\, a_n$, where $a_0 = 1$.
2. Write a code which would output values a_1 to a_{16} of the progression $a_{n+1} = -2\, a_n$, where $a_0 = 3$.

Fibonacci Sequence

You may be familiar with the Fibonacci Sequence, or have heard of it before. We can define it in a similar way to the other progressions mathematically, but with a key difference.

In this case, we will say that we are trying to find the value for an element in a sequence a_{n+1}, which is defined with regards to the most recently calculated element a_n and the element before it a_{n-1},

$$a_{n+1} = a_n + a_{n-1}. \tag{6}$$

We can see that our new element, a_{n+1}, is the sum of the element before it in the sequence a_n and the element before that in sequence a_{n-1}. The Fibonacci Sequence occurs with this relation, and two starting values of $a_0 = 0$ and $a_1 = 1$ (think about why two starting values are required in this case, rather than just one).

This gives us that $a_2 = a_1 + a_0 = 1 + 0 = 1$, $a_3 = a_2 + a_1 = 1 + 1 = 2$, $a_4 = a_3 + a_2 = 2 + 1 = 3$... and so on. One way of coding this (not the only method) would be a code as follows.

```fortran
INTEGER :: i, a_current, a_old, a_new

a_old = 0
a_current = 1

DO i = 1, 20, 1
  a_new = a_current + a_old
  WRITE(*,*) a_new
  a_old = a_current
  a_current = a_new
END DO
```

In this case, we have used three separate variables for a_{n+1}, a_n and a_{n-1}, `a_new`, `a_current` and `a_old` respectively. As mentioned, this is not the only way of doing this; you may want to try thinking about other ways with which you can achieve the same objective.

The first `a_new` calculated will be a_2, with `a_current` being a_n and `a_old` a_{n-1}. Within the loop, we first calculate the new value using the two existing ones with `a_new = a_current + a_old`. This is then written to the terminal. Next, we are effectively 'shuffling along' which elements of the sequence the variables `a_current` and `a_old` refer to. Remember that, at the beginning of our loop, `a_current=` a_1 and `a_old=` a_0'; with `a_new` the, as yet undetermined, a_2. At the end of an iteration of the loop, the line `a_old = a_current` is therefore giving `a_old` `a_current`'s value, effectively making `a_old` now equal a_1. Then, the line `a_current = a_new`, is giving `a_current` `a_new`'s value, that of a_2.

So, with an iteration of the loop having finished, we now have the condition that a_old= a_1 and a_current= a_2. This puts us in a position to find a_3, as we now have both elements previous to it. This means that in the next iteration, a_new will be used to represent the undetermined value a_3.

This process of calculating a new value, then shuffling along the values of variables such that the only values stored are those needed for the next iteration is a key part of how loops can be used.

You will recall that, in the sections on arithmetic and geometric progressions, it was possible to find a value in the sequence with any index in one step with an equation, requiring only that the initial value and the increment/multiplier were known (Equations 2 and 5).

Can the same be done for the Fibonacci Sequence? You can try determining it yourself, if you are interested, though it is definitely not required. The value at index n in the Fibonacci sequence is given by,

$$a_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\left(\frac{1+\sqrt{5}}{2}\right) - \left(\frac{1-\sqrt{5}}{2}\right)} = \frac{\left(1+\sqrt{5}\right)^n - \left(1-\sqrt{5}\right)^n}{2^n\sqrt{5}}. \tag{7}$$

As you can see, it is more complicated than Equations 2 and 5, especially given how simple its definition is for our program ($a_{n+1} = a_n + a_{n-1}$, $a_0 = 0$ and $a_1 = 1$). Deriving such an equation in the first place may be difficult for some problems which are otherwise simple to state. While it was comparitively easy with minimal mathematics knowledge to write the program, determining the full mathematical solution for any particular element requires rather more understanding and time put in to deriving it. While, frequently, the ideal way to solve a particular problem may be to find the exact general mathematical solution; for problems of increasing complexity, the corresponding mathematical solutions can become very greatly more complicated to determine (assuming that it is possible to determine one). It is for this reason that analysing mathematics using computer programs becomes an important element in solving physical problems. The application of this to the series presented here is a small fraction of what can be done in the wider field of numerical methods.

We will end this guide with an example program that asks the user to input the number of Fibonacci Sequence numbers they would like to be printed out, and then determines them. Try to ensure that you understand the reason for each piece of the code having been written in the way it is. In particular, consider why the loop might run from 2 to total-1, rather than 1 to total as in previous examples.

An additional note, related to data types rather than loops themselves: This program will only be correct up to a_{46}, try running to larger values, seeing what happens and work out why this may be. What is the problem, and how might it be fixed...? (hint: remember the description of the INTEGER data type in Section 1.8.2 of the course booklet!)

```fortran
PROGRAM fibonacci
  INTEGER :: i, a_current, a_old, a_new, total

  a_old = 0
  a_current = 1

  WRITE(*,*) 'User, please enter how many Fibonacci numbers to find:'
  READ(*,*) total

  WRITE(*,*) 'Finding', total, 'Fibonacci numbers'

  WRITE(*,*) 'Value', 0, '=', a_old
  WRITE(*,*) 'Value', 1, '=', a_current

  DO i = 2, total-1, 1
    a_new = a_current + a_old
    WRITE(*,*) 'Value', i, '=', a_new
    a_old = a_current
    a_current = a_new
  END DO
END PROGRAM fibonacci
```