Design and Computing 1

Introduction to Scientific Computing

Daniel Poole
Department of Aerospace Engineering
University of Bristol
`d.j.poole@bristol.ac.uk`

2017

LECTURE 4

Intro to Scientific Computing

- Have used programs to perform simple calculations
- Conditionals can be used to branch off during a program
- Loops are used to perform calculations multiple times

TODAY

- Splitting code into managable chunks - ”divide and conquer”
- Writing those chunks as functions
- An in-depth worked example to demonstrate how to write code as functions

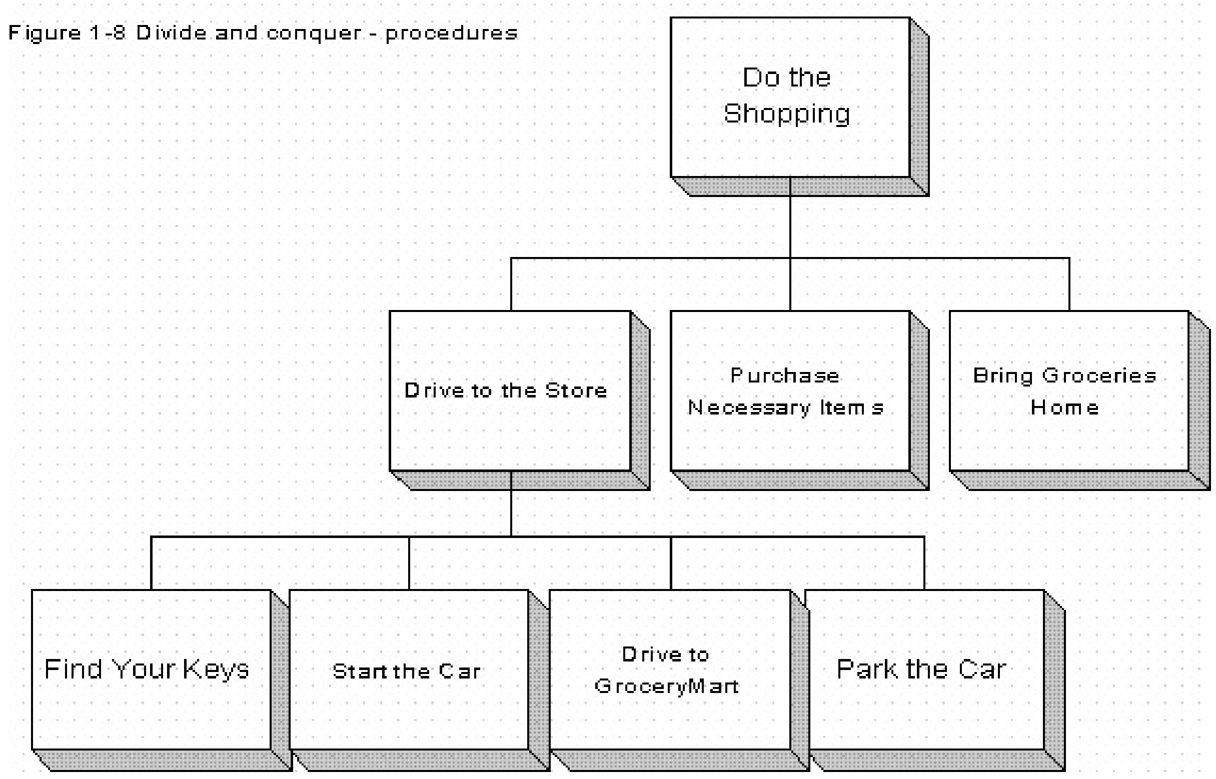
Functions

In the last two lectures we have begun to understand how coding can be used to solve problems, and the tools that the programming paradigm gives us to solve those problems. In the labs, emphasise has been placed on, firstly, taking a problem and splitting it down in a logical manner to be able to programme and solve it, and secondly, that it is important to write code that is logical and easily understandable. The latter of these two has, so far, been achieved using comments, indenting and good variable names. However, this lecture will introduce the concept of functions.

Functions are blocks of code, separate from the main program itself. Writing functions makes the code more logical and readable, and will usually make the main program simpler. In this lecture we will go through what a function is, and at the end we will take a problem, split it down into a number of functions, and go through the process of coding it. You will therefore see how adding the knowledge of functions to our programming repertoire allows us to write more logical and structured code.

Divide and Conquer

Before considering what a function is, like other programming aspects, it is useful to understand the logic of when a function is used. This will become more obvious with experience, but for now, we should consider programming from a **DIVIDE AND CONQUER** approach. Take your big problem and split it down into manageable chunks. Programme and test each chunk independently and then stitch together at the end.



Functions

We will now outline all of the necessary details to be able to use functions. At the end of the lecture we will go through and perform and ”divide and conquer” approach to a given example problem.

A function is a utility that calculates an output based on some input **arguments**:

```
y=sin(x);
```

or multiple inputs:

```
c=pow(a,b);
```

We don’t always have an output:

```
printf("");
```

Built-in Functions

As you have already seen in the first two labs, C has many built in functions. These are arranged in libraries. To load a library, we include it in the first lines of our program:

```
#include<library name>
```

For information on the different libraries, a good reference is:

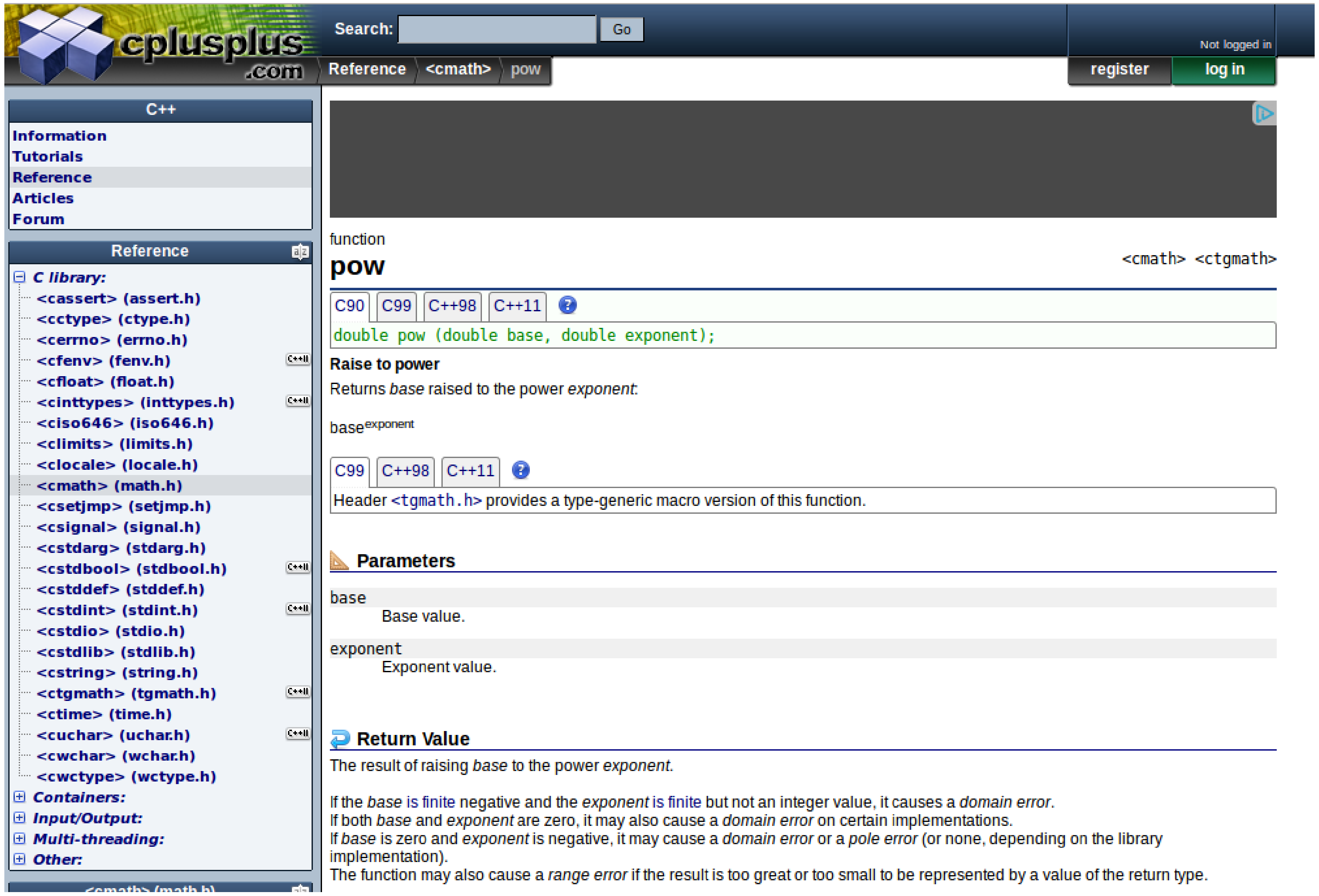
`http://www.cplusplus.com/reference/`

The most common ones you will probably use are:

```
#include<stdio.h>
#include<math.h>
#include<string.h>
```

Built-in Functions

If we consider the math library, we can go to the reference page and view one of the functions (`pow` for example):



function
pow
`<cmath> <ctgmath>`

C90 C99 C++98 C++11

```
double pow (double base, double exponent);
```

Raise to power

Returns *base* raised to the power *exponent*:

$\text{base}^{\text{exponent}}$

C99 C++98 C++11

Header `<tgmath.h>` provides a type-generic macro version of this function.

Parameters

base
Base value.

exponent
Exponent value.

Return Value

The result of raising *base* to the power *exponent*.

If the *base* is finite negative and the *exponent* is finite but not an integer value, it causes a *domain error*.
If both *base* and *exponent* are zero, it may also cause a *domain error* on certain implementations.
If *base* is zero and *exponent* is negative, it may cause a *domain error* or a *pole error* (or none, depending on the library implementation).
The function may also cause a *range error* if the result is too great or too small to be represented by a value of the return type.

Built-in Functions

We can see that the input arguments to `pow` are two doubles, and the return value is also a double.

```
double pow (double base, double exponent);
```

We must therefore write a code that adheres to these requirements:

```
#include<stdio.h>
#include<math.h>
int main () {
  double a,b,c;
  c=pow(a,b);
  return(0);
}
```

The intrinsic functions of C are wide ranging, and are easy to call. However, we must also understand how to write our own functions if we are to make our code elegant.

Function Syntax

The syntax for writing our own functions has the same anatomy as main:

```
<output type> <function name> (<arg1 type> <arg1> ...) {
  <local variable allocations>
  <statements>
  return(<output value>);
}
```

We can have as many input arguments to the function as we want, of any type we want.

Note that we define local variables inside the function. These are wiped when the function ends.

Function Syntax

The syntax for writing our own functions has the same anatomy as main:

```
<output type> <function name> (<arg1 type> <arg1> ...) {
  <local variable allocations>
  <statements>
  return(<output value>);
}
```

If we have an output value to our function, then we must call it like:

```
<output value> = <function name> (<var1> ...)
```

Otherwise, if there is no output, then it is called like:

```
<function name> (<var1> ...)
```

Function example

An example is calculating the distance between two points:

```
double dist(double x1, double y1, double x2, double y2){
  double d;
  d=sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
  return(d);
}
```

Notable items within this example are:

- We have four arguments, each of type double
- There is a local declaration used to store the value we wish to return
- The value of the variable `d` is returned from the function. This variable is then wiped from memory.

Function example

To call this function from our main program, it is much like calling an intrinsic function:

```
double dist(double x1, double y1, double x2, double y2){
  double d;
  d=sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
  return(d);
}

int main () {
  double xa=1.0, ya=1.0;
  double xb=2.0, yb=2.0;
  double ab;
  ab=dist(xa,ya,xb,yb);
  return(0);
}
```

NOTE: `main` function comes last in the file

Function example

We can call a function as many times as we want

```
double dist(double x1, double y1, double x2, double y2){
  double d;
  d=sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
  return(d);
}

int main () {
  double xa=1.0, ya=1.0;
  double xb=2.0, yb=2.0;
  double xc=2.0, yc=2.0;
  double ab, ac;
  ab=dist(xa,ya,xb,yb);
  ac=dist(xa,ya,xc,yc);
  return(0);
}
```

Pass-by Value

When we use a variable as an argument, C sends only the **value** of that variable to the function. The function then creates a **local copy** of that variables i.e. a new variable that has the same value as the argument passed-in is created. Therefore, the function cannot change the value of the variable. If we wanted to do this then we would have to use a pointer (see later lectures).

```
int settotwo(int a) {
  printf("F start: a=%i\n",a);
  a=2;
  printf("F end: a=%i\n",a);
  return(0);
}
int main () {
  int x=5;
  printf("M start: x=%i\n",x);
  settotwo(x);
  printf("M end: x=%i\n",x);
  return(0);
}
```

```
M start: x=5
F start: a=5
F end: a=2
M end: x=5
```

We can see here that the variable `x` has been passed to a function with argument `a`. The program has effectively read the value of `x` and copied it to another variable `a`. This can be modified, but will not be passed back out of the function as we have `return(0)`.

Worked Example

DEMO: Numerical Integration

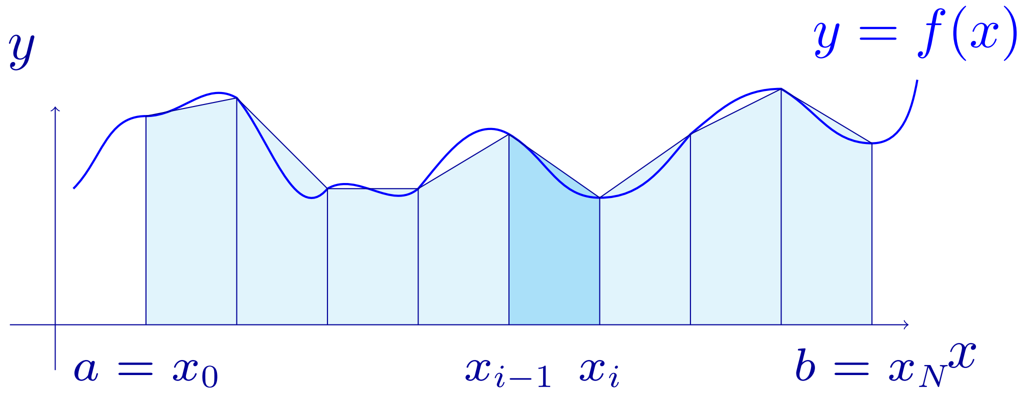
Numerical Integration

The best way to understand dividing and conquering your code is to do some examples. The lab this week will deal with this, however, we will do one exercise here to understand how this works. This example will be the numerical calculation of a definite integral:

$$\int_a^b f(x)dx \tag{1}$$

Numerically, the method for solving this integral is to split the interval into N sections and work out the area of each section and add them together. It is common to use trapeziums as the sections, so

$$\int_a^b f(x)dx \approx \sum_{i=1}^{N} \Delta x \frac{f(x_{i-1}) + f(x_i)}{2}$$



Numerical Integration

We will use the trapezium rule to evaluate the following integral:

$$\int_0^{10} \left(2 + 0.05x^2 + \sin(x)\right) dx$$

An integral like this we can do by hand, so we know what the real answer should be:

Problems with known answers are useful for testing your code.

Numerical Integration

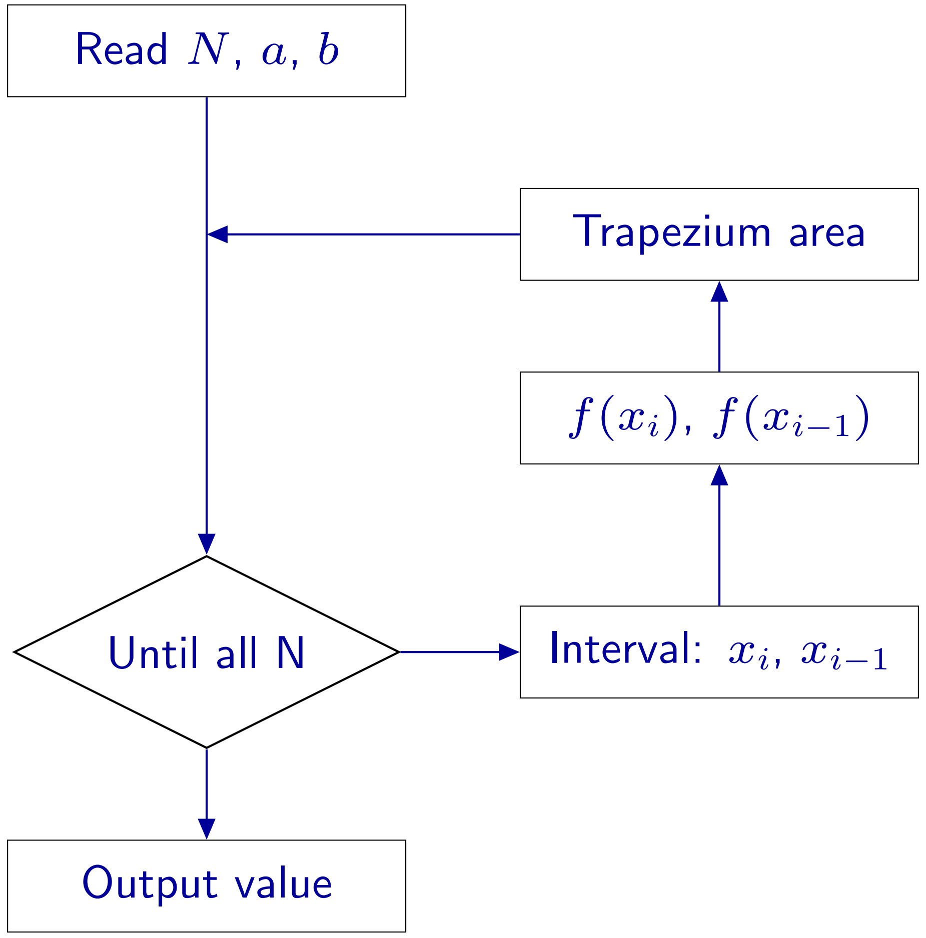
We should use a divide and conquer approach for solving this problem. Split the problem down into manageable chunks and solve them one by one:

1. Read the data
2. Set up the intervals along the domain
3. Produce a function value for a given input x value
4. Loop over all the trapeziums and calculate the areas
5. Add them all up to get the final answer
6. Make the output look pretty

Now we should do them one-by-one and test as we go

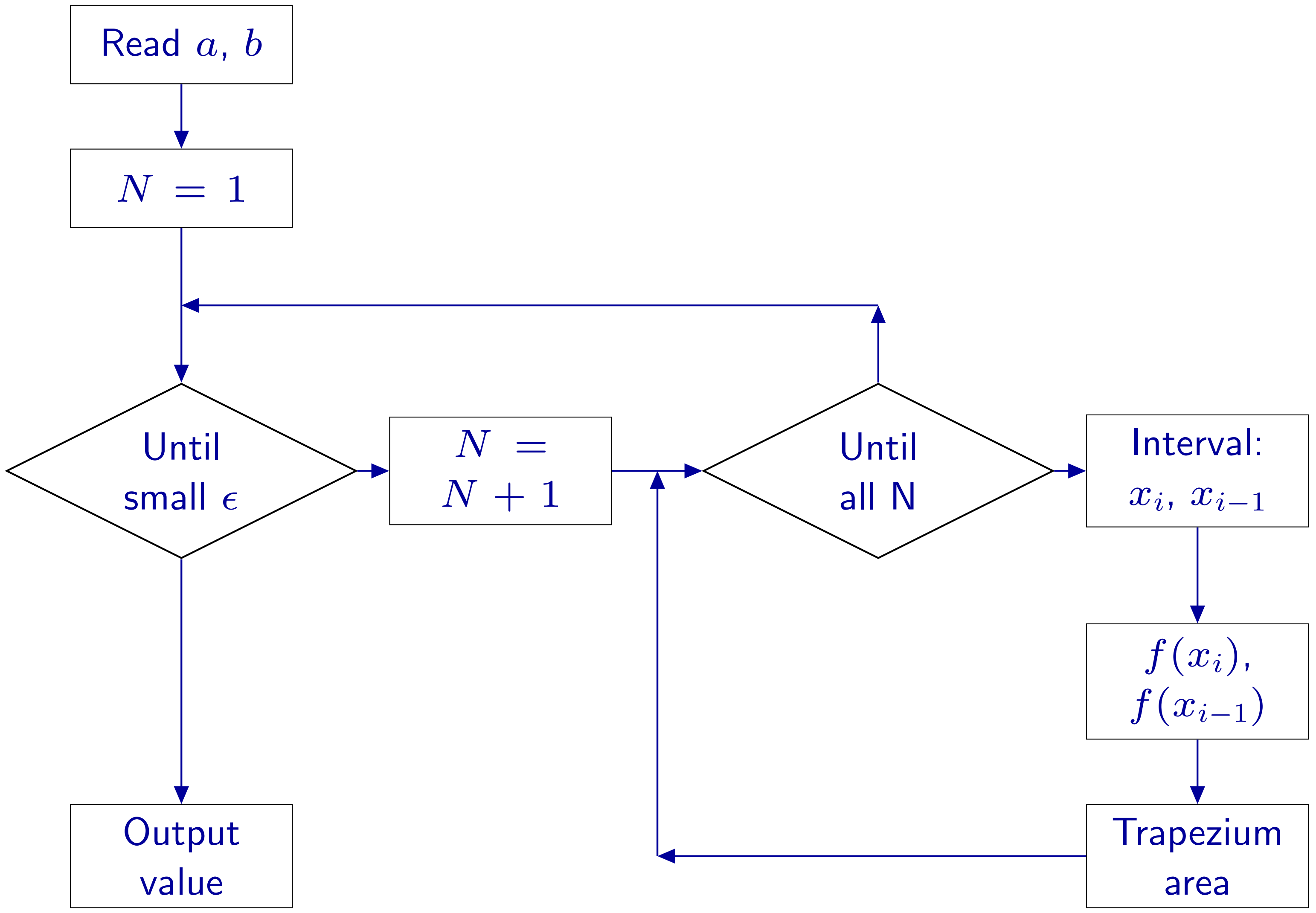
Numerical Integration

We can draw a flowchart for the processes we will have to go through:



Numerical Integration

Because we know the answer we could also put a while loop around to work out the number of N required to obtain a small enough error



Summary

- Have introduced functions
- Can now use a divide and conquer approach to coding
- Have demonstrated this on a detailed worked example

LAB

- Implementing coding examples using functions