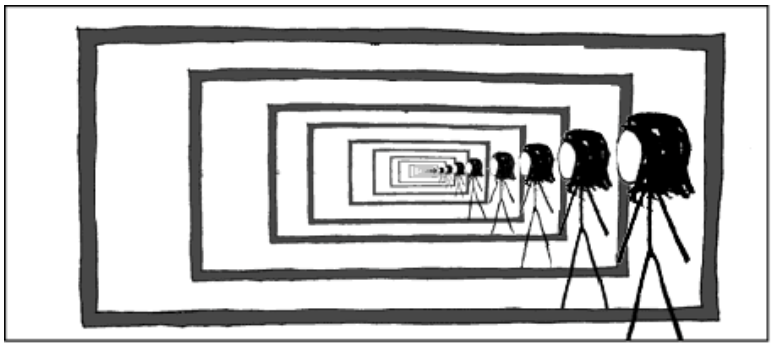
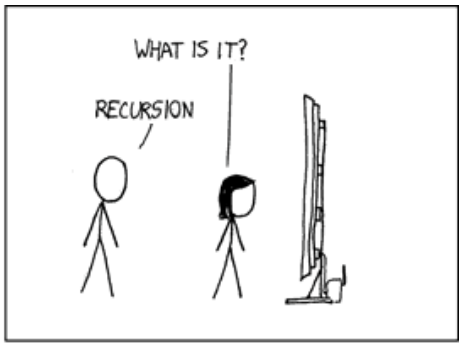
**CS319: Scientific Computing**

# Functions and Quadrature

Dr Niall Madden

Week 4: 5th and 7th, February, 2025






Slides and examples: https://www.niallmadden.ie/2425-CS319

# 0. Outline



Today

Slides and examples:
https://www.niallmadden.ie/2425-CS319

Next it makes sense to write a function that implements the Trapezium Rule, so that it can be used in different settings.

The idea is pretty simple:

- As before, `f` will be a globally defined function.
- We write a function that takes as arguments `a`, `b` and `N`.
- The function implements the Trapezium Rule for these values, and the globally defined `f`.

04QuadratureV02.cpp (header)

```
// 04QuadrateureV02.cpp: Trapezium Rule as a function
// Trapezium Rule (TR) quadrature for a 1D function
// Author: Niall Madden
// Date: Feb 2025
// Week 04: CS319 - Scientific Computing
#include <iostream>
#include <cmath>   // For exp()
#include <iomanip>

double f(double x) {  return(exp(x)); } // definition
double TrapRule(double a, double b, int N);
```

Prototype

04QuadratureV02.cpp (main)

```
int main(void )
{
  std::cout << "Using the TR to integrate in 1D\n";
  std::cout << "Integrate between x=0 and x=1.\n";
  double a=0.0, b=1.0;
  double Int_true_f = exp(1)-1; // for f(x)=exp(x)

  std::cout << "Enter value of N for the Trap Rule: ";
  int N;
  std::cin >> N; // Lazy! Should do input checking.

  double Int_TR_f = TrapRule(a,b,N);
  double error_f = fabs(Int_true_f - Int_TR_f);

  std::cout << "N=" << std::setw(6) << N <<
    ", Trap Rule=" << std::setprecision(6) <<
    Int_TR_f << ", error=" <<  std::scientific <<
    error_f << std::endl;
  return(0);
```

04QuadratureV02.cpp (function)

```
34 double TrapRule(double a, double b, int N)
   {
36   double h=(b-a)/double(N);
     double QFn = (h/2.0)*f(a);
38   for (int i=1; i<N; i++)
       QFn += h*f(a+i*h);
40   QFn += (h/2.0)*f(b);
     return(QFn);
42 }
```

Note: f is defined globally.

"Quadrature" fn - "function".

# 4. Functions as arguments to functions

We now have a function that implements the Trapezium Rule. However, it is rather limited, in several respects. This includes that the function, `f`, is hard-coded in the `TrapRule` function. If we want to change it, we'd edit the code, and recompile it.

Fortunately, it is relatively easy to give the name of one function as an argument to another.

The following example shows how it can be done.

# 4. Functions as arguments to functions

05QuadratureV03.cpp(header)

```
// 05QuadrateureV03.cpp: Trapezium Rule as a function
// that takes a function as argument
// Week 04: CS319 - Scientific Computing
#include <iostream>
#include <cmath>   // For exp()
#include <iomanip>

double f(double x) {  return(exp(x)); } // definition
double g(double x) {  return(6*x*x); } // definition

double TrapRule(double Fn(double), double a, double b,
                int N);
```

!! New !!

Note we specify the argument type & return type.

# 4. Functions as arguments to functions

05QuadratureV03.cpp (part of main())

```
20    std::cout << "Which shall we integrate: \n"
                << "\t 1. f(x)=exp(x) \n\t 2. g(x)=6*x^2?\n";
22    int choice;
      std::cin >> choice;
24    while (!(choice == 1 || choice  == 2) )
      {
26      std::cout << "You entered " << choice
                  <<". Please enter 1 or 2: ";
28      std::cin >> choice;
      }
30    double Int_TR=-1; // good place-holder
      if (choice == 1)
32      Int_TR = TrapRule(f,a,b,10);
      else
34      Int_TR = TrapRule(g,a,b,10);

36    std::cout << "N=10" << ", Trap Rule="
                << std::setprecision(6) << Int_TR  << std::endl;
38    return(0);
}
```

# 4. Functions as arguments to functions

05QuadratureV03.cpp (TrapRule())

```
42 double TrapRule(double Fn(double), double a,
                  double b, int N)
44 {
     double h=(b-a)/double(N);
46   double QFn = (h/2.0)*Fn(a);
     for (int i=1; i<N; i++)
48     QFn += h*Fn(a+i*h);
     QFn += (h/2.0)*Fn(b);

     return(QFn);
52 }
```

locally (ie, within this function) f(x) or g(x) are known as Fn(x).

# 5. Functions with default arguments

In our previous example, we wrote a function with the header
`double TrapRule(double Fn(double), double a, double b, int N);`

And then we called it as
`Int_TR = TrapRule(f,a,b,10);`

That is, when we were not particularly interested in the value of `N`, we took it to be 10.

It is easy to adjust the function so that, for example, if we called the function as
`Int_TR = TrapRule(f,a,b);`
it would just be assumed that $N = 10$. All we have to do is adjust the function header.

# 5. Functions with default arguments

To do, this we specify the value of $N$ in the **function prototype**. You can see this in `05QuadratureV04.cpp`. In particular, note Line 10:

`06QuadratureV04.cpp` (line 10)

```
10 double TrapRule(double Fn(double), double a,
                   double b, int N=10); // default N=10
```

This means that, if the user does not specify a value of $N$, then it is taken that $N = 10$.

# 5. Functions with default arguments

**Important:**

- You can specify default values for as many arguments as you like. For example:

```
1 double TrapRule(double Fn(double), double a=0.0,
          double b=1.0, int N=10);
```

- If you specify a default value for an argument, you must specify it for any following arguments. For example, the following would cause an error.

```
2 double TrapRule(double Fn(double), double a=0.0,
          double b=1.0, int N);
```

because we've default values for a and b, but not for N.

# 6. Pass-by-value

In C++ we need to distinguish between

- the value stored in the variable.
- a variable's identifier (might not be unique)
- a variable's (unique) memory address

In C++, if (say) `v` is a variable, then `&v` is the memory address of that variable.

We'll return to this at a later point, but for now we'll check the output of some lines of code that output a memory address.

```
int v = 20;
```

Then

- the value stored in the variable is 20
- the name/identifier is v
- Address of v is &v.

# 6. Pass-by-value

07MemoryAddresses.cpp

```
  int i=12;
10  std::cout << "main: Value stored in i: " << i << '\n';
  std::cout << "main: address of i: " << &i << '\n';
12  Address(i);
  std::cout << "main: Value stored in i: " << i << '\n';
```

Typical output might be something like:

```
main: The value stored in i  is 12
main: The address of i is 0x7ffcd1338314
```

# 6. Pass-by-value

A while back we learned that, when we pass a variable as an argument to a function, a new **copy** of the variable is made.

This is called **pass-by-value**.

Even if the variable has the same name in both `main()` and the function called, and the same value, they are different: the variables are **local** to the function (or block) in which they are defined.

We'll test this by writing a function that

- Takes a `int` as input;
- Displays its value and its memory address;
- Changes the value;
- Displays the new value and its memory address.

# 6. Pass-by-value

07MemoryAddresses.cpp

```
18 void Address(int i)
   {
20   std::cout << "Address: Value stored in i: " << i << '\n';
     std::cout << "Address: address of i: " << &i << '\n';
22   i+=10; // Change value of i
     std::cout << "Address: New val stored in i: " << i << '\n';
24   std::cout << "Address: address of i: " << &i << '\n';
   }
```

# 6. Pass-by-value

Finally, let's call this function:

07MemoryAddresses.cpp

```
    int i=12;
10  std::cout << "main: Value stored in i: " << i << '\n';
    std::cout << "main: address of i: " << &i << '\n';
12  Address(i);
    std::cout << "main: Value stored in i: " << i << '\n';
14  std::cout << "main: address of i: " << &i << '\n';
```

# 6. Pass-by-value

In many case, "pass-by-value" is a good idea: a function can change the value of a variable passed to it, without changing the data of the calling function.

But sometimes we **want** a function to be able to change the value of a variable in the calling function.

The classic example is function that

- takes two `integer` inputs, `a` and `b`;
- after calling the function, the values of `a` and `b` are swapped.

# 6. Pass-by-value

08SwapByValue.cpp

```
4  #include <iostream>
   void Swap(int a, int b);  // bad!

   int main(void )
8  {
     int a, b;

     std::cout << "Enter two integers: ";
12   std::cin >> a >> b;

14   std::cout << "Before Swap: a=" << a << ", b=" << b
               << std::endl;
16   Swap(a,b);
     std::cout << "After Swap: a=" << a << ", b=" << b
18             << std::endl;

20   return(0);
   }
```

# 6. Pass-by-value

```
void Swap(int x, int y)
{
  int tmp;

  tmp=x;
  x=y;
  y=tmp;
}
```

**This won't work.**

We have passed only the *values stored in the variables a and b*. In the `swap` function these values are copied to local variables $x$ and $y$. Although the local variables are swapped, they remained unchanged in the calling function.

What we really wanted to do here was to use **Pass-By-Reference** where we modify the contents of the memory space referred to by `a` and `b`. This is easily done...

# 6. Pass-by-value

...we just change the declaration and prototype from

```
void Swap(int x, int y)  // Pass by value
```

to

```
void Swap(int &x, int &y)  // Pass by Reference
```

the pass-by-reference is used.

void Swap(int &x, int &y)

# 7. Exercises

## Exercise (Simpson's Rule)

- *Find the formula for Simpson's Rule for estimating* $\int_a^b f(x)dx$.
- *Write a function that implements it.*
- *Compare the Trapezium Rule and Simpson's Rule. Which appears more accurate for a given N?*

## Exercise

*Change the* `Address()` *function in* `07MemoryAddresses.cpp` *so that the variable* `i` *is passed by reference.*
*How does the output change?*

Finished here