CS319: Scientific Computing

Week 6: Pointers, Arrays, and Quadrature again

Dr Niall Madden

9am and **4pm**, 14 February, 2024

Slides and examples: https://www.niallmadden.ie/2324-CS319

Annotated slides from 4pm

Pointers

To properly understand how to use arrays, we need to study **Pointers**.

- We already learned that if, say, `x` is a variable, then `&x` is its memory address.
- A **pointer** is a special type of variable that can store memory addresses. We use the `*` symbol before the variable name in the declaration.
- For example, if we declare

  ```
  int i;
  int *p
  ```

  then we can set `p=&i`.

If we declare
int p2[10];
then p2 is also a pointer: it stores the memory address of the start of the array. But the value of p2 is fixed, whereas p can be changed.

Pointers

01Pointers.cpp

```
int a=-3, b=12;
int *where;

std::cout << "The variable 'a' stores " << a <<
  '\n' << "The variable 'b' stores " << b << '\n';
std::cout << "'a' is stored at address " << &a <<
  '\n' "'b' is stored at address " << &b << '\n';

where = &a;
std::cout << "The variable 'where' stores "
          << (void *) where << std::endl;
std::cout << "... and that in turn stores " <<
  *where << '\n';
```

One can actually do calculations on memory addresses. This is called **pointer arithmetic**. One can't (for example) add two addresses, or compute their product, but you can, for example, increment them.

02PointerArithmetic.cpp

```
int vals[3];
vals[0]=10;  vals[1]=8;  vals[2]=-4;

int *p;
p = vals;

for (int i=0; i<3; i++)
{
  std::cout << "p=" << p << ", *p=" << *p << "\n";
  p++;
}
```

Here "*" is a "dereference" operator.
It is (kinda) an inverse operator of "&": it returns the value that is stored in the memory address stored in p!
In fact, if we set "a=10", then "*(&a)" is 10 as well.
But "&(*a)" does not mean anything.

Being able to manipulate memory addresses is one of the reasons C++ is considered a very **powerful** language. It is possible to preform (low-level) operations in C++ that are impossible in, say, Python.

But it is also possible to write programmes that will crash, or even crash your computer, since memory addresses are not well protected.

Dynamic Memory Allocation

In all examples we've had so far, we've specified the size of an array at the time it is defined.

In many practical cases, we don't have that information. For example, we might need to read data from a file, but not know the file size in advance.

It would be useful if, on the fly, we could set the size of an array.

Furthermore, for efficiency, we may want to free up memory allocated.

how much space is needed to store arrays

To add this functionality, we will use two new (to us) C++ operators for dynamic $\boxed{\text{memory allocation}}$ and deallocation: `new` and `delete`. (There are also functions `malloc()`, `calloc()` and `free()` inherited from C, but we won't use them).

This can change!

The `new` operator is used in C++ to allocate memory. The basic form is

var = new *type*

where *type* is the specifier of the object for which you want to allocate memory and *var* is a pointer to that type.

If insufficient memory is available then `new` will return a NULL pointer or generate an exception.

To dynamically allocate an array:

- First declare a pointer of the right type:
```
int *data;
```
- Then use `new`
```
data = new int[MAX_SIZE];
```

When it is no longer needed, the operator `delete` releases the memory allocated to an object.

To "delete" an array we use a slightly different syntax:

```
delete [] array;
```

where *array* is a pointer to an array allocated with `new`.

Example: Quadrature 1

In Week 4, we introduced the idea of **numerical integration** or **quadrature**.

We computed estimates for $\int_a^b f(x)dx$ by applying the Trapezium Rule:

- Choose the number of intervals N, and set $h = (b - a)/N$.
- Define the quadrature points $x_0 = a$, $x_1 = a + h$, $\ldots x_N = b$. In general, $x_i = a + ih$.
- Set $y_i = f(x_i)$ for $i = 0, 1, \ldots, N$.
- Compute $\displaystyle\int_a^b f(x)dx \approx Q_1(f) := h\Big(\frac{1}{2}y_0 + \sum_{i=1:(N-1)} y_i + \frac{1}{2}y_N\Big)$.

[Take notes for the next few slides]

Eg $x_N = a + Nh = a + N\left(\frac{b-a}{N}\right) = a + b - a = b.$

Example: Quadrature 1

In Week 4, we introduced the idea of **numerical integration** or **quadrature**.

We computed estimates for $\int_a^b f(x)dx$ by applying the Trapezium Rule:

- Choose the number of intervals N, and set $h = (b-a)/N$.
- Define the quadrature points $x_0 = a$, $x_1 = a + h$, $\dots x_N = b$. In general, $x_i = a + ih$.
- Set $y_i = f(x_i)$ for $i = 0, 1, \dots, N$.
- Compute $\displaystyle\int_a^b f(x)dx \approx Q_1(f) := h\Big(\frac{1}{2}y_0 + \sum_{i=1:(N-1)} y_i + \frac{1}{2}y_N\Big).$

[Take notes for the next few slides]

$i = 1:(N-1)$ is "Matlab" notation $i = 1, 2, 3, \dots, N-1$

03TrapeziumRule.cpp

```
#include <iostream>
#include <cmath>    // For exp()
#include <iomanip>

double f(double x) {  return(exp(x)); } // definition
double ans_true = exp(1.0)-1.0; // true value of integral

double Quad1(double *x, double *y, unsigned int N);
```

$f(x) = e^x$.

$\int_0^1 f(x) = e^1 - e^0 = e-1$

define the f we want to integrate, and the true value of $\int_a^b f(x)\, dx$, which we'll use for estimating errors.

Header for function that implements the Trapezium Rule.

Example: Quadrature 1

Next we skip to the function code... Also /* comment */

03TrapeziumRule.cpp

```
double Quad1(double *x, double *y, unsigned int N)
{
  double h = (x[N]-x[0])/double(N);
  double Q = 0.5*(y[0] + y[N]);
  for (unsigned int i=1; i<N; i++)
    Q += y[i];
  Q *= h;   // Q = Q * h
  return(Q);
}
```

Source of confusion: * is used in two very different contexts here.

$$Q_1(f) = \left(\frac{1}{2} y_0 + \sum_{i=1:(N-1)} y_i + \frac{1}{2} y_N\right) h$$

Example: Quadrature 1

Back to the main function: declare the pointers, input N, and allocate memory.

03TrapeziumRule.cpp

```
int main(void )
{
  unsigned int N;
  double a=0.0, b=1.0; // limits of integration
  double *x; // quadrature points
  double *y; // quadrature values

  std::cout << "Enter the number of intervals: ";
  std::cin >> N; // not doing input checking

  x = new double[N+1];
  y = new double[N+1];
```

Example: Quadrature 1

Initialise the arrays, compute the estimates, and output the error.

03TrapeziumRule.cpp

```
double h = (b-a)/double(N);
for (unsigned int i=0; i<=N; i++)
{
  x[i] = a+i*h;
  y[i] = f(x[i]);
}
double Est1 = Quad1(x,y,N);
double error = fabs(ans_true - Est1);
std::cout << "N=" << N << ", Trap Rule="
          << std::setprecision(6) << Est1
          << ", error=" <<  std::scientific
          << error << std::endl;
```

Example: Quadrature 1

Finish by de-allocating memory (optional, in this instance).

03TrapeziumRule.cpp

```
  delete [] x;
  delete [] y;
  return(0);
}
```

deallocate the memory.

Enter the number of intervals: 8
N=8, Trap Rule=1.72052, error=2.236764e-03

Enter the number of intervals: 16
N=16, Trap Rule=1.71884, error=5.593001e-04

Enter the number of intervals: 32
N=32, Trap Rule=1.71842, error=1.398319e-04

Example: Quadrature 1

Although this was presented as an application of using arrays in C++, some questions arise...

1. What value of N should we pick the ensure the error is less than, say, 10^{-6}?
2. How could we predict that value of N if we didn't know the true solution? That is, how can we estimate the error?
3. What is the smallest error that can be achieved in practice? Why?
4. How does the time required depend on N? What would happen if we tried computing in two or more dimensions?
5. Are there any better methods? (And what does "better" mean?)

Example: Quadrature 1

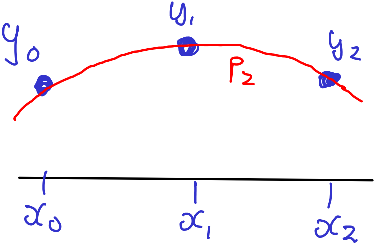
Some answers to those questions.

We'll just try to answer the last of these.

"better" might mean that, for the same effort, we get a smaller error.
But how much smaller, and when is it worth the effort?

Simpson's Rule is an improvement on the Trapezium Rule.

Here is a rough idea of how it works:



P_2 is quadratic polynomial.

$$\int_{x_0}^{x_2} f(x)\,dx \approx \int_{x_0}^{x} P_2(x)\,dx.$$

$$\approx \frac{h}{6}\left(y_0 + 4y_1 + y_2\right)$$

The Method is:

- Choose an **EVEN** number of intervals N, and set $h = (b-a)/N$.
- Define the quadrature points $x_0 = a$, $x_1 = a + h$, $\ldots x_N = b$. In general, $x_i = a + ih$.
- Set $y_i = f(x_i)$ for $i = 0, 1, \ldots, N$.
- Compute

$$Q_2(f) := \boxed{\frac{h}{3}\Big(y_0 + \sum_{\underline{i=1:2:N}-1} 4y_i + \sum_{i=2:2:N-2} 2y_i + y_N\Big).}$$

$i = 1:2:N-1$ is $1, 3, 5, \ldots, N-1$

Quadrature 2: Simpson's Rule

The program 04CompareRules.cpp implements both methods and compares the results for a given N. Here we just show the code for the implementation of Simpson's Rule.

04CompareRules.cpp

```
double Quad2(double *x, double *y, unsigned int N)
{
  double h = (x[N]-x[0])/double(N);
  double Q = y[0]+y[N];
  for (unsigned int i=1; i<=N-1; i+=2)
    Q += 4*y[i];
  for (unsigned int i=2; i<=N-2; i+=2)
    Q += 2*y[i];
  Q *= h/3.0;
  return(Q);
}
```

Quadrature 2: Simpson's Rule

Typical output:

```
Enter the number of intervals: 8
 N=8 | Trap Error=2.236764e-03 | Simp Error=2.326241e-06

Enter the number of intervals: 16
 N=16 | Trap Error=5.593001e-04 | Simp Error=1.455928e-07

Enter the number of intervals: 32
 N=32 | Trap Error=1.398319e-04 | Simp Error=9.102726e-09
```

Finished here 5pm