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

Outline

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

Recall: memory addresses

In Week 5 we learned that...

- If (for example) `x` is some variable, then `&x` is its memory address.
- Usually, when you pass a variable to a function, you just pass a local copy. This is called **pass-by-value**.
- If you want the function to change the value of the variable in the calling function, you have to **pass-by-value** by passing the memory address of the variable. This is done by adding the `&` symbol before the variable name in the function header and definition.

Arrays

Much of Scientific Computing involves working with data, and often collections of data are stored as **arrays**, which are list-like structures that stores a collection of values all of the same type.

Example: declare an array to store five floats:

```
float vals[5];
vals[0]=1.0;  vals[1]=2.1;
vals[2]=3.14; vals[3]=-21.0;
vals[4]=-1.0;
```

Arrays

Consider the following piece of code:

00Array.cpp

```
  float vals[3];
  vals[0]=1.1;  vals[1]=2.2;  vals[2]=3.3;
  for (int i=0; i<3; i++)
    std::cout << "  vals["<<i<<"]=" << vals[i];
  std:: cout << std::endl;
  std::cout << "vals=" << vals << '\n';
```

The output I get looks like

```
  vals[0]=0  vals[1]=1.1  vals[2]=2.02
vals=0x7ffd9ab8ec9c
```

Can we explain the last line of output?

Arrays

So now it know that, if `vals` is the name of an array, then in fact the value stored in `vals` is the memory address of `vals[0]`.

We can check this with

```
std::cout << "vals=" << vals << '\n';
std::cout << "&vals[0]=" << &vals[0] << '\n';
std::cout << "&vals[1]=" << &vals[1] << '\n';
std::cout << "&vals[2]=" << &vals[2] << '\n';
```

For me, this gives

```
vals=0x7ffc932b960c
&vals[0]=0x7ffc932b960c
&vals[1]=0x7ffc932b9610
&vals[2]=0x7ffc932b9614
```

Can we explain?

Arrays

And in the same piece of code, if I changed the first line from

`float vals[3];`

to

`double vals[3];`

we get something like

```
vals=0x7ffd361abdc0
&vals[0]=0x7ffd361abdc0
&vals[1]=0x7ffd361abdc8
&vals[2]=0x7ffd361abdd0
```

Can we explain?

Arrays

So now we understand why C++ (and related languages) index their arrays from `0`:

- `vals[0]` is stored at the address in `vals`;
- `vals[1]` is stored at the address after the one in `vals`;
- `vals[k]` is stored at the *k*th address after the one in `vals`;

But there are numerous complications, not least that different data types are stored using different numbers of bytes. So the off-set between addresses changes.

To understand the subtleties, we need to know about **pointers**.

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`.

Pointers

01Pointers.cpp

```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';
```

Pointer arithmetic

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

```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++;
}
```

Pointers

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.

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

Dynamic Memory Allocation `new`

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];
  ```

Dynamic Memory Allocation `delete`

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 $x_0 = a$, $x_1 = a + h$, $\dots x_N = b$. 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]

Example: Quadrature 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);
```

Example: Quadrature 1

Next we skip to the function code...

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;
  return(Q);
}
```

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

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

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

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 if we didn't know the true solution?
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.

Quadrature 2: Simpson's Rule

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

Here is a rough idea of how it works:

Quadrature 2: Simpson's Rule

The Method is:

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

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

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:

Analysis

For the rest of today, we want to analyse, experimentally, the results given by these program.

We'll do the calculations, in detail, for the Trapezium Rule.

As an exercise, you can redo this for Simpson's Rule.

Analysis

05CheckConvergence.cpp

```
int main(void )
{
  unsigned K = 8;   // number of cases to check
  unsigned Ns[K];    // Number of intervals
  double Errors[K];
  double a=0.0, b=1.0; // limits of integration
  double *x, *y; // quadrature points and values.
```

Analysis

05CheckConvergence.cpp

```
for (unsigned k=0; k<K; k++)
{
  unsigned N = pow(2,k+2);
  Ns[k] = N;
  x = new double[N+1];
  y = new double[N+1];
  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);
  Errors[k] = fabs(ans_true - Est1);
  delete [] x; delete [] y;
}
```

Analysis

[Switch to Jupyter]