CS319: Scientific Computing

Week 7: Analysis of Algorithms, NumPy, and Multidimensional Arrays

Dr Niall Madden

9am and **4pm**, 21 February, 2024

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

Outline

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

Recall: Quadrature

In Week 6 we re-visited the idea of **numerical integration** or **quadrature**, and considered two very simple algorithms for approximating

$$\int_a^b f(x)dx.$$

Most such algorithms are based the following:

- Form an array of **quadrature points** at which we will do some calculations.
- Form an array of **Quadrature values** which are values of f at the quadrature points.
- Compute some average of these to estimate the integral.

Recall: Quadrature

We considered two algorithms: the Trapezium Rule, and Simpson's Rule. In both cases we:

- Choose the number of intervals N, and set $h = (b-a)/N$.
- **Quadrature points** are $x_i = a + ih$ for $i = 0, 1, \ldots, N$.
- **Quadrature values** are $y_i = f(x_i)$ for $i = 0, 1, \ldots, N$.

Trapezium Rule

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

. . . .

Simpson's Rule:

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

Recall: Quadrature

We finished in Week 6 by running the program `04CompareRules.cpp`, which test both methods attempts at estimating

$$\int_0^1 e^x dx.$$

It's output is tabulated below.

N	**Trapezium Error**	**Simpson's Error**
8	2.236764e-03	2.326241e-06
16	5.593001e-04	1.455928e-07
32	1.398319e-04	9.102726e-09
64	3.495839e-05	5.689702e-10

From this we can quickly observe the Simpson's Rule to give smaller errors than the Trapezium Rule, for the same effort.

Can we quantify this?

Analysis

Next we want to analyse, experimentally, the results given by these program.

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

In Lab 5, you will redo this for Simpson's Rule.

. . . .

Let $E_N = |\int_a^b f(x)dx - Q_1(f)|$ where $Q_1(\cdot)$ is implemented for a given N.

We'll speculate that

$$E_N \approx CN^{-q},$$

for some positive constants C and q. If this was a numerical analysis module (like MA378) we'd determine C and p from theory. In CS319 we do this *experimentally*.

Analysis

The idea:

Analysis

To implement this, we need some data. That can be generated, for the Trapezium Rule, by the following programme.

Notice that we use dynamic memory allocation. That is because the size of the arrays, x and y change while the programme.

00CheckConvergence.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

00CheckConvergence.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

Our program outputs the results in the form of two numpy arrays. We'll have two different functions (with the same name!), since one is an array of ints and the other doubles.

Here is the code for creating outputting numpy array of doubles. The one for ints is similar.

00CheckConvergence.cpp

```
void print_nparray(double *x, int n, std::string str)
{
  std::cout << str << "=np.array([";
  std::cout << std::scientific << std::setprecision(6);
  std::cout << x[0];
  for (int i=1; i<n; i++)
    std::cout << ", " << x[i];
  std::cout << "])" << std::endl;
```

Juputer: lists and NumPpy

The next set of slides are in the Jupyter Notebook:
`CS319-Week07.ipynb`

Two-dimensional arrays

Recall the idea of a one dimensional array. For example, if we wanted to store a set of **five** `int`egers, we could declare an array

```
int v[5];
```

We could then access the five elements:
`v[0]`, `v[1]`, `v[2]`, `v[3]`, and `v[4]`.

This is a **one-dimensional array**: the array has a single index. It is similar to the idea of a **vector** in Mathematics.

Often we will have table/rectangles of data, in a way that is similar to a **matrix**.

Two-dimensional arrays

In C++, an two-dimensional $M \times N$ array of (say) `double`s can be declared as:

```
double A[M][N];
```

Then its members are

$$\begin{pmatrix} \text{A[0][0]} & \text{A[0][1]} & \text{A[0][2]} & \cdots & \text{A[0][N-1]} \\ \text{A[1][0]} & \text{A[1][1]} & \text{A[1][2]} & \cdots & \text{A[1][N-1]} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \text{A[M-1][0]} & \text{A[M-1][1]} & \text{A[M-1][2]} & \cdots & \text{A[M-1][N-1]} \end{pmatrix}$$

For example, `double A[3][4];`
gives a 2D array

$$\begin{pmatrix} \text{A[0][0]} & \text{A[0][1]} & \text{A[0][2]} & \text{A[0][3]} \\ \text{A[1][0]} & \text{A[1][1]} & \text{A[1][2]} & \text{A[1][3]} \\ \text{A[2][0]} & \text{A[2][1]} & \text{A[2][2]} & \text{A[2][3]} \end{pmatrix}$$

Two-dimensional arrays

Dynamic memory allocation in 2D is a little complicated, because a 2D array is actually just an “array of arrays”.

This is because, we declare, for example,
`double A[3][4];` what really happens is:

- `A` is assigned the base address of three **pointers**: `A[0]`, `A[1]`, `A[2]`.
- Each of those is a base address for a (1D) array of 4 doubles.

This approach has advantages: because of it the language can support arrays in as many dimensions as one would like.

But it makes DMA more complicated.

Two-dimensional arrays

To use dynamic memory allocation to reserve memory for a two-dimensional $M \times N$ matrix of doubles (for example):

- declare a "pointer to pointer to double"
- use `new` to assign memory for M pointers;
- for each of those, assign memory for N doubles.

Code:

```
double **A;
A = new double* [M];
for (int i=0; i<M; i++)
     A[i] = new double [N];
```

Two-dimensional arrays

OOPS!

This slide was accidentally omitted from the original version of these slides.

If we dynamically allocate memory for a 2D array, we need to de-allocate it too, using the `delete` operator (See Week 6).

If the array `A` as been allocated as on the previous slide, it is de-allocated as:

```
for (int i=0; i<M; i++)
     delete [] A[i];
delete[] A;
```

Quadrature in 2D

For the last time (in lectures) we'll look at **numerical integration**, this time of two dimensional functions.

That is, our goal is to estimate

$$\int_{a_1}^{b_1} \int_{a_2}^{b_2} f(x_1, x_2) dx_1 dx_2.$$

When we implement an algorithm for this, we will set

- `x1` and `x2` to be vectors of (one-dimensional) quadrature of $N+1$ points.
- `y` to be a **two-dimensional** array of $(N+1)^2$ quadrature values. That is, we will set
  `y[i][j] = f(x1[i], x2[j]);`

Derivation

Implementation

We'll implement this for estimating $\int_0^1 \int_0^1 e^{x_1+x_2} dx_1 dx_2$, with N quadrature points in each direction.

01Trap2D.cpp preamble

```
double f(double x1, double x2) {  return(exp(x1+x2)); }
double ans_true = pow(exp(1.0)-1.0,2); // true value

double Trap2D(double *x1, double *x2,
              double **y, unsigned int N);
```

01Trap2D.cpp main()

```cpp
int main(void )
{
  unsigned N = pow(2,4); // Number of points in each direction
  double a1=0.0, b1=1.0, a2=0.0, b2=1.0; // limits of int
  double h1, h2; // step-size in x1 and x2
  double *x1, *x2, **y; // quadrature points and values.

  x1 = new double[N+1];
  x2 = new double[N+1];

  h1 = (b1-a1)/double(N);
  h2 = (b2-a2)/double(N);
  for(unsigned i = 0; i < N+1; i++)
  {
    x1[i] = a1+i*h1;
    x2[i] = a2+i*h2;
  }
```

01Trap2D.cpp main() continued

```cpp
  y = new double * [N+1];
  for(unsigned i = 0; i < N+1; i++)
    y[i] = new double[N+1];

  for (unsigned i=0; i<N+1; i++)
    for (unsigned j=0; j<N+1; j++)
      y[i][j] = f(x1[i], x2[j]);

  double est1 = Trap2D(x1, x2, y, N);
  double error1 = fabs(ans_true - est1);

  std::cout << "N=" <<  N << " | est=" << est1
            << " | error = " << error1 << std::endl;
```

01Trap2D.cpp Trap2D()

```
double Trap2D(double *x1, double *x2, double **y,
              unsigned N)
{
  double Q, h1 = (x1[N]-x1[0])/double(N),
    h2 = (x2[N]-x2[0])/double(N);

  Q = 0.25*(f(x1[0],x2[0]) + f(x1[N],x2[0])   // 4 corners
          + f(x1[0],x2[N]) + f(x1[N],x2[N]));

  for (unsigned k=1; k<N; k++)    // 4 edges (not including corners)
    Q += 0.5*(f(x1[k],x2[0]) + f(x1[k],x2[N])
            + f(x1[0],x2[k]) + f(x1[N],x2[k]));

  for (unsigned i=1; i<N; i++) // All the points in the interior
    for (unsigned j=1; j<N; j++)
      Q += f(x1[i],x2[j]);

  Q *= h1*h2;
  return(Q);
```

Lab 5 preview

- Implement Simpson's Rule in 1D and 2D;
- Verify convergence using Python/NumPy/Jupyter.