CS319: Scientific Computing

Week 7: ③ Multidimensional Arrays, and ① Analysis of Algorithms , ② Python

Dr Niall Madden

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

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

Annotated slides from 9am class.

Outline

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

Draft!

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. $\{x_0, x_1, x_2, \ldots \}$
- Form an array of **Quadrature values** which are values of f at the quadrature points. $\{y_0, y_1, y_2 \ldots, \}$
- Compute some average of these to estimate the integral.

$y_i = f(x_i)$. In C++: `y[i] = f[x(i)]`

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 $i = 0, 1, \dots, N$.
- **Quadrature values** are $y_i = f(x_i)$ for $i = 0, 1, \dots, N$.

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

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

Error decreases as N increases: "Convergence".

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

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

Actually: $E_N \leq C N^{-q}$ for all $N \geq N_0$

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

Experimentally!

Analysis

[notes edited after class]

The idea:

Suppose that $\mathcal{E}_N \approx C N^{-q}$.

Then $\log(\mathcal{E}_N) \approx \log(C N^{-q})$

$= \log(C) + \log(N^{-q})$

$= \log(C) - q \log(N)$.

Let $Y = \log(\mathcal{E}_N)$, $K = \log(C)$ and $X = \log(N)$.

Then $Y \approx K - q X$.

So Y resembles the equation of a line with slope $-q$, and Y-intercept K.

[PTO]

Analysis

The idea: $\mathcal{E}_N \cong C N^{-q} \iff Y = K - qX$.

Idea:

- Choose some values of N (eg, 8, 16, 32, ...)
- Compute associated values of $\mathcal{E}_N$
- Compute Y and X from $\mathcal{E}_N$ and N.
- Find the equation of the line $Y = K - qX$
  That is, estimate K and q. In practice, this is a least squares fit.
- Compute $C = e^K$. So now we have both C and q.

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]; // E_N
  double a=0.0, b=1.0;  // limits of integration
  double *x, *y;  // quadrature points and values.
```

Check 8 different values of N

pointers for use in dynamic memory allocation.

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;
}
```

$N = 2^k$

DMA.

define Quadrature points

define Quadrature values.

apply the Trapezium Rule to x, y. See notes from Week 6.

deallocation.

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 `int`s and the other `double`s.

Here is the code for creating outputting `numpy` array of doubles. The one for `int`s 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`

[Finished the 9am class discussing lists in Python]