CS319: Scientific Computing

Algorithm Analysis (Quadrature and Jupyter)

Dr Niall Madden

Week 6: 18 February, 2026

Slides and examples: https://www.niallmadden.ie/2526-CS319

0. Reminders

1. Grades for Lab 2 will be posted early next week.
2. Lab 4 will be posted shortly before 9am tomorrow.
3. **Class test**: here Friday at 11.

0. Outline

1. Recall: Quadrature
2. Quadrature 2: Simpson's Rule
   - Comparison
3. Analysis
   - Output numpy array
4. Juputer: lists and NumPpy

Slides and examples:
https://www.niallmadden.ie/2526-CS319

1. Recall: Quadrature

Last week, 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 $x_0 = a$, $x_1 = a + h$, ..., $x_N = b$. In general, $x_i = a + ih$.
- Set $y_i = f(x_i)$ for $i = 0, 1, \ldots, N$.
- Compute $$\int_a^b f(x)dx \approx Q_1(f) := h\Big(\frac{1}{2}y_0 + \sum_{i=1}^{N-1} y_i + \frac{1}{2}y_N\Big).$$

1. Recall: Quadrature

We then applied this method to estimate $\int_0^1 e^x dx$, for various values of N.

We got results like the following:

```
N=  8, Trap Rule=1.72052, error=2.236764e-03
N= 16, Trap Rule=1.71884, error=5.593001e-04
N= 32, Trap Rule=1.71842, error=1.398319e-04
N= 64, Trap Rule=1.71832, error=3.495839e-05
N=128, Trap Rule=1.71829, error=8.739624e-06
```

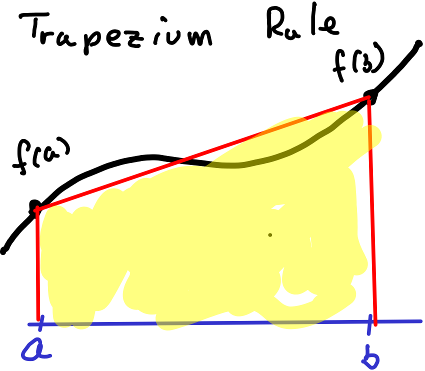
1. Recall: Quadrature

We then pondered some of the following questions:

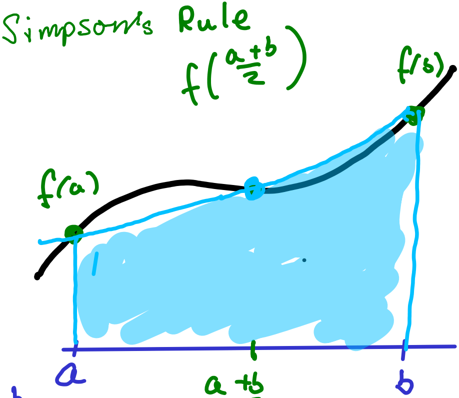
1. What value of N should we pick the ensure the error is less than, say, 10^{-6}?
2. How could we predict that value if we didn't know the true solution?
3. What is the smallest error that can be achieved in practice? Why?
4. How does the time required depend on N? What would happen if we tried computing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

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

Here is a rough idea of how it works: $N=1$



$$\int_a^b f(x)\,dx \simeq \frac{(b-a)}{2}\left(f(a)+f(b)\right)$$

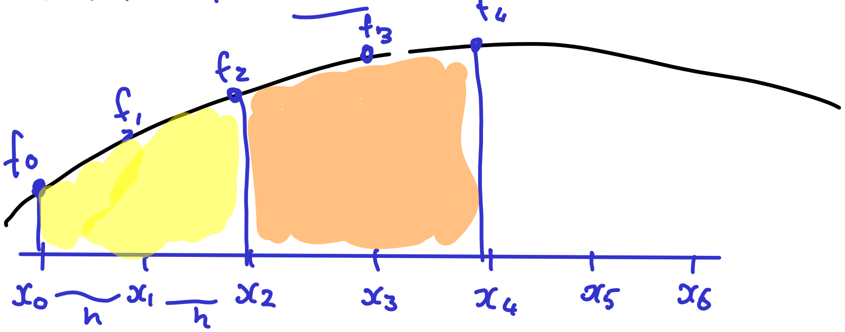


$$\int_a^b f(x)\,dx \simeq \frac{b-a}{6}\left(f(a)+f\left(\frac{a+b}{2}\right)4+f(b)\right)$$

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

Here is a rough idea of how it works:

Usually we apply Simpson's Rule with $N > 1$ & Even.



$$\frac{h}{3}(f_0 + 4f_1 + f_2) + \frac{h}{3}(f_2 + 4f_3 + f_4) \quad \text{etc.}$$

2. Quadrature 2: Simpson's Rule

Simpson's Rule

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

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

The program 00CompareRules.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.

00CompareRules.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)     ← odd i
    Q += 4*y[i];
  for (unsigned int i=2; i<=N-2; i+=2)     ← even i
    Q += 2*y[i];
  Q *= h/3.0;
  return(Q);
```

When we run `00CompareRules.cpp`, and h test both methods attempts at estimating

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

we get output like:

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?

3. Analysis

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.

C depends on f.

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**.

The idea: we expect that

$$\mathcal{E}_N \simeq C N^{-q}$$

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

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

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

Then

$$Y = K - qX$$

We can compute X & Y with our code. Then make a least-squares fit. From that we get K & q.

3. 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.

01CheckConvergence.cpp

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

01CheckConvergence.cpp

```
for (unsigned k=0; k<K; k++)
{
  unsigned N = pow(2,k+2);   // N=4,8,16,..., 512
  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;
}
```

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.

01CheckConvergence.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;
```

4. Juputer: lists and NumPpy

- The next set of slides are in the Jupyter Notebook: `CS319-Week06-notebook.ipynb`.
- Can be downloaded from https://www.niallmadden.ie/2526-CS319
- Can try that out on `https://cloudjupyter.universityofgalway.ie`
- Tips:
  - on that server, try: `File... Open from URL...` add `https://www.niallmadden.ie/2526-CS319/Week06/CS319-Week06-notebook.ipynb`
  - Or try just clicking on this nbgit-puller link.

4. Juputer: lists and NumPpy

- The next set of slides are in the Jupyter Notebook: `CS319-Week06-notebook.ipynb`.
- Can be downloaded from https://www.niallmadden.ie/2526-CS319
- Can try that out on `https://cloudjupyter.universityofgalway.ie`
- Tips:
  - on that server, try: `File... Open from URL...` add `https://www.niallmadden.ie/2526-CS319/Week06/CS319-Week06-notebook.ipynb`
  - Or try just clicking on this nbgit-puller link.

4. Juputer: lists and NumPpy

- The next set of slides are in the Jupyter Notebook: `CS319-Week06-notebook.ipynb`.
- Can be downloaded from https://www.niallmadden.ie/2526-CS319
- Can try that out on `https://cloudjupyter.universityofgalway.ie`
- Tips:
  - on that server, try: `File... Open from URL...` add `https://www.niallmadden.ie/2526-CS319/Week06/CS319-Week06-notebook.ipynb`
  - Or try just clicking on this nbgit-puller link.

Finished here Wed @ 5pm