Computers in Engineering COMP 208

Numerical Integration

Michael A. Hawker

Integration

- Many applications require evaluating the integral of a function
- The integrals of many elementary functions cannot be derived analytically
- As we have seen, we may not even have an analytic form for the function. We may just be able to sample it at various points

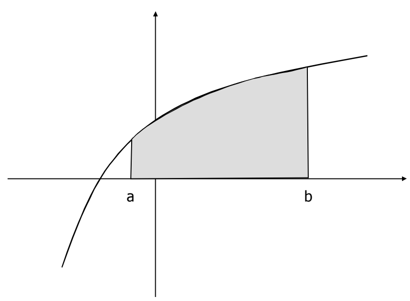
Integration

- This lead to the development of techniques for evaluating such integrals numerically
- Numerical integration techniques predate the use of electronic computers

Definite Integral

- The definite integral of a function of a single variable, f(x), between two limits a and b can be viewed as the area under the curve defined by the function
- Numerical integration algorithms try to estimate this area

Integral



Numerical Integration

- Our approach will be to divide the region between a and b into n segments
- We then estimate the area under the curve in each segment
- Finally, we sum these areas

Numerical Integration

- We consider three algorithms for estimating this area
  - The Midpoint method
  - The Trapezoidal method
  - Simpson's method

Midpoint Method

We estimate the area under the curve in each segment using the value of f at the midpoint of this segment

```
area = dx * f(x+dx/2)
```

To compute an approximation to the interval, we just have to sum these areas

The Midpoint Method

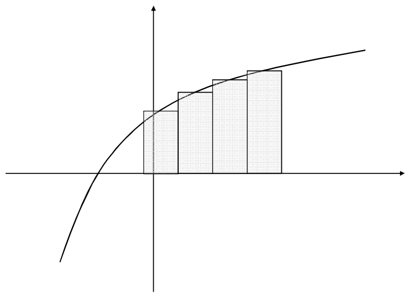
- We begin by dividing the region from a to b into n equal segments
- The width of each segment is

```
dx = (b-a)/n
```

- The endpoint of the segments are

```
a, a+dx, a+2dx, … ,a+ndx
```

Midpoint Method



Midpoint Method

- Multiplication (especially by small values) may cause roundoff errors
- To reduce the effect of these errors, we try to simplify expressions to reduce the number of multiplications
- We can factor out the dx and add all of the function values before multiplying by dx

Midpoint Method

```
double midpoint_int (DfD f,
                double x0, double x1, int n){
  int i;
  double x, dx, sum = 0.0;
  dx = (x1-x0)/ n;
  for (i = 0, x = x0 + dx/2; i < n;
       i++, x += dx)
    sum += f(x);
  return sum * dx;
}
```

Trapezoidal Method

To improve the accuracy of our estimate for the area of each segment we use the area of the trapezoid rather than the rectangle

The area of the trapezoid formed by `x`, `x+dx`, `f(x)` and `f(x+dx)` is

```
area = dx * (f(x) + f(x+dx))/2
```


Trapezoidal Method

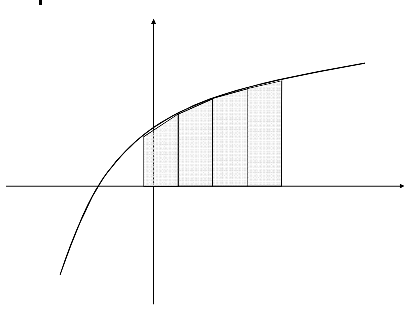
The area of each segment is given by

```
area = dx * (f(x) + f(x+dx))/2
```

We sum these areas for each panel to approximate the integral

To reduce the number of operations and the roundoff, we can factor out the dx

Trapezoid Method



Trapezoidal Method Simplifications

Moreover the sum telescopes:

```
(f(x0)+f(x1))/2 + (f(x1)+f(x2))/2
  = f(x0)/2 + f(x1) + f(x2)/2
  = (f(x0)+f(x2))/2 + f(x1)
```

This collapsing of terms effects all the terms except for the first and last

Trapezoidal Method

```
double trapezoidal_int (DfD f,
            double x0, double x1, int n){
  double x, dx, sum;
  int i;
  dx = (x1-x0)/ n;
  sum = (f(x0) + f(x1))/2;
  for (i=1,x = x0 + dx; i < n; x += dx)
    sum += f(x);
  return sum * dx;
}
```


Simpson’s Method

- Simpson’s method fits a parabola through the curve at three points, the value of the function at the two endpoints and at the midpoint of the interval
- Simpson’s method generally finds a better approximation to the area under the curve in each segment

Simpson's Method

Given three points
`(a,f(a)),  (b,f(b))` and `(c,f(c))`
there is a unique polynomial, called the *interpolating polynomial* , that passes through these points.

Simpson's Method

The area under this parabola between two points x and x+dx is given by

```
[f(x) +
 4*f(x+dx/2) +
 f(x+dx)]*dx/6
```

The integral is again approximated by summing these areas

Implementing Simpson's Method

Again, in order to minimize roundoff errors and improve efficiency, we simplify the sum

We factor out the `dx/6`

We can also telescope some terms of the sum

This results in the following algorithm

Simpson's Method

```
double simpsons_int(DfD f,
            double x0, double x1, int n){
  double x, sum, dx = (x1 - x0) / n;
  sum = f(x1) - f(x0);

  for(x = x0; x+dx/2 < x1; x += dx)
    sum += 2.0 * f(x) +
           4.0 * f(x + dx/2);

  return sum * dx / 6.0;
}
```


Accuracy of Integration

- Midpoint Method
  Exact for constant and piecewise linear functions
- Trapezoidal Method
  Exact for constant and piecewise linear functions
- Simpson's Rule
  Exact for polynomials of degree three or less

Accuracy of Integration

If we divide our original interval into subintervals of width h, it is possible to derive estimates on the accuracy of these methods for more general functions

- Midpoint Method
  The error is of order h^2
- Trapezoidal Method
  The error is the same
- Simpson's Rule
  The error is of order h^5

Monte Carlo Methods

- Monte Carlo methods use pseudo-random numbers to approximate definite integrals
- These methods are sometimes used for multidimensional functions integrated over a region that has a complicated shape

Monte Carlo Methods

- Consider the computing the volume of the intersection of two cylinders
- We can consider a cube that contains this shape.
- We then generate a large number of points and count how many fall within the region we are interested in
- Since we can compute the volume of the simple shape, we obtain an estimate of the volume of the complex shape

Simple Monte Carlo Integration

- For simple one dimensional functions, we have at least two different ways to apply this concept
- Method 1:
  - Bound a region containing the definite integral by a rectangle.
  - Divide the number of random points that fall under the curve by the total number of points used

Simple Monte Carlo Integration

Method 2:

- Sum the value of f at a large number of random points.
- Then divide by the total number of points used.

Random Number Generation (Review)

We first have to seed the pseudo-random number generator with an initial value using `srand(seed)`

The function `time(x)` in `time.h` gives us the current clock time of our computer and can be coerced to an unsigned integer type.

```
srand((unsigned int) time(NULL));
```


Random Number Generation (Review)

The function `rand()` returns a random unsigned integer less than the system defined value `RAND_MAX`

By coercing it to be a double precision real and dividing by `RAND_MAX`, we get a real number between 0 (inclusive) and 1 (exclusive)

We can then scale this value to fall between x0 and x1

```
((double) rand() / RAND_MAX) * (x1-x0)+x0);
```

Monte Carlo Integration

```
double monte_carlo_int(DfD f,
                       double x0, double x1, int n){
  double sum = 0;
  int i;
  srand((unsigned int) time(NULL));
  // Sum n random values of f(x) with x in [x0, x1].
  for(i = 0; i < n; ++i)
    sum += f(((double) rand() / RAND_MAX)*(x1-x0)+x0);

  // Divide the sum by n to get average value of f(x)
  // Multiply by (x1 - x0) to get the area.
  return (sum / n) * (x1 - x0);
}
```


Pi using Monte Carlo

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(){
  int i, numin = 0, num = 1000000;
  double x_coord, y_coord, pi;
  srand((unsigned int)time(NULL));
  for (i=0; i<num; i++){
    x_coord = ((double)rand())/RAND_MAX;
    y_coord = ((double)rand())/RAND_MAX;
    if (x_coord*x_coord + y_coord*y_coord < 1.0) numin++;
    pi = 4.0*(double)numin/(double)num;
  }
   printf("Pi after %i steps is %g \n", num, pi);
   return 0;
}
```