1 Operations on a Numerical Grid

Several elementary mathematical operations are of primary importance in the Physics Sciences and Engineering. These include root finding, locating maxima and minima, as well as basic integration and differentiation. We learn about all of these in calculus class, in the context of continuous functions. But what if the function is only known via a table based on a numerical grid, or (as in most numerical simulations) is defined only at discrete points? This section reviews some numerical methods used to perform these operations.

1.1 Functions on a Grid

The numerical algorithms described here are designed to perform basic mathematical operations on functions tabulated on uniformly spaced (simpler) or arbitrary numerical grids. A uniformly spaced grid is defined by three of four parameters: (1) its starting point A, (2) its ending point B, and either (3) the number of grid points N or (4) the grid spacing Δx. The last two are related by

$$\Delta x = \frac{B - A}{N - 1}.$$

In what follows, for purposes of illustration we will use a simple function

$$f(x) = x\, e^{-x^2} = x\, \exp(-x^2).$$

The following `C++` program (`FunctionOnGrid.cpp`) lists the values of $f(x)$ on a numerical grid in tabular form.

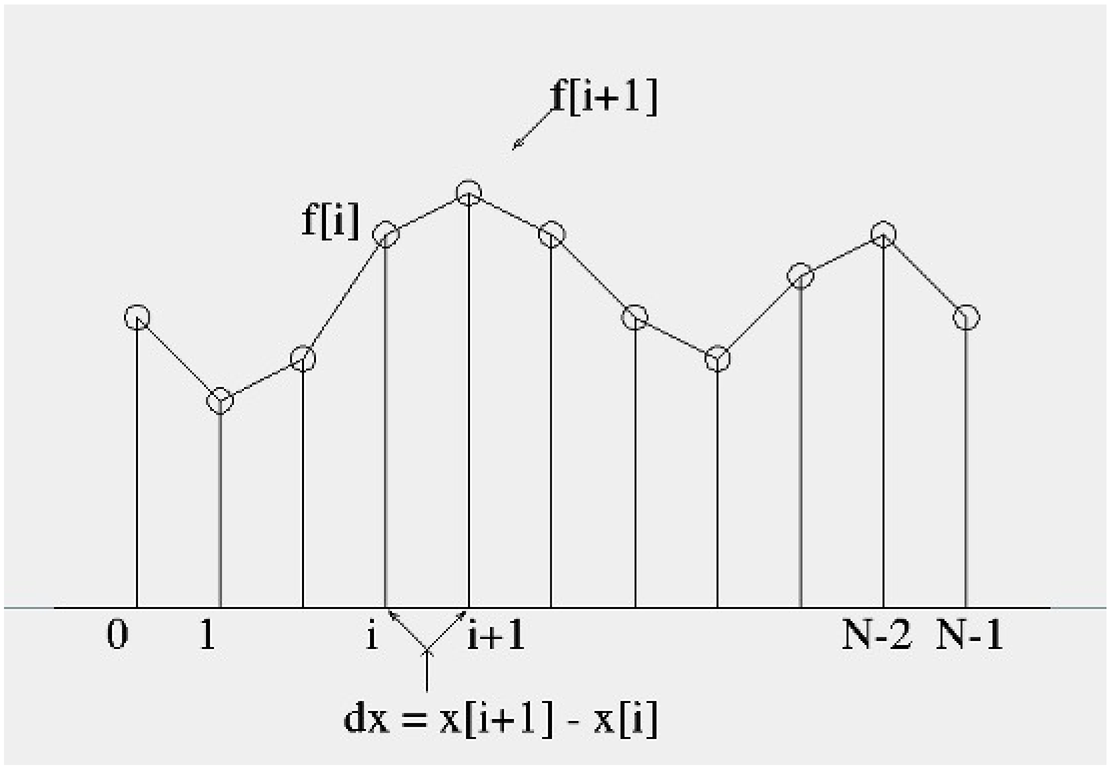
```
#include <iostream>
#include <cmath>

double F( double x )
{
    return x * exp( -( x * x ) );
}

int main()
{
    double x, func;
    int i;

    // Define the grid.

    double A = -1.0;
    double B = 3.0;
    int N = 51;
    double dx = ( B - A )/(N-1);

    // Function on this grid.
```

```
        for( i=0; i<N; i++ ) {
            x = A + i*dx;
            func = F( x );
            std::cout << i << "  " << x << "  " << func << "\n";
        }
    }
```

It lists i, x_i and $f(x_i)$ for the index i in the range $[0, N-1]$ and $x_i = A + i\,\Delta x$. The output stream of the program may be piped into a file.

2 Integrals

Calculating a definite integral of a function entails finding the *area under the curve* defined by the function. This graphical interpretation leads to some very simple and intuitive methods when combined with the listing of the function on a numerical grid. Specifically, numerical calculation of a definite integral amounts to summing the areas of a series of *strips* covering the integration domain, as illustrated in the figure below.



We will use our example function

$$f(x) = x\,\exp(-x^2)$$

and calculate the definite integral

$$I = \int_A^B f(x)dx$$

for various domains. The `C++` program listed earlier lists $f(x)$ in a tabular fashion.

2.1 The Trapezoidal Rule

The simplest methods follow from the assumption that every strip under the curve is a rectangle based on the function value at the left or at the right of each strip. However, intuition tells us that this will give a very inaccurate value for the area since we are approximating the underlying function by horizontal straight line segments.

A more accurate form follows if we assume that each strip has the shape of a trapezoid. The area of such a trapezoidal shaped strip, delimited by f_i and f_{i+1}, is

$$da_i = \tfrac{1}{2}\,\Delta x\,(f_i + f_{i+1}).$$

The area under the curve is simply the sum of the areas of all such strips between $A(i-0)$ and $B(i = N-1)$:

$$\begin{aligned} I &= \sum_{i=0}^{N-2} da_i \\ &= \left[\tfrac{1}{2}(f_0 + f_{N-1}) + \sum_{i=1}^{N-2} f_i\right] \Delta x. \end{aligned}$$

The method is illustrated in the following code (`Trapezoid.cpp`)

```
//  Read a function on a grid from a file; calculate the area under
//  the curve.

#include <iostream>
#include <cmath>

int main()
{
    double xx[1000], ff[1000];
    int i, j;

    // Read the function from cin.

    j = 0;
    for ( ; ; ) {
        std::cin >> i >> xx[j] >> ff[j] ;
        if ( std::cin.eof() == true )
            break;
        j++;
    }
```

```
        // numerical grid
        double dx = xx[1] - xx[0];
        int N = j;

        // Area via Trapezoidal rule.

        double area = 0.0;
        for( i=0; i<N-2; i++ ) {
            double areastrip = 0.5 * ( ff[i]+ff[i+1] ) * dx;
            area = area + areastrip;
        }
        std::cout << "#   Area (Trapezoidal) = " << area << "\n";

        //    Exact area (integral).

        std::cout << "#   Area ( exact )     = " <<
            - ( exp(-xx[N-1]*xx[N-1]) - exp(-xx[0]*xx[0]) ) / 2.0 << " \n" ;
    }
```

The earlier program may be used to pipe data into this code.

The advantage of using the simple form $f(x) = x\,\exp(-x^2)$ is that the exact integral is easily determined analytically:

$$I = \int x\,\exp(-x^2)\,dx = -\tfrac{1}{2}\exp(-x^2),$$

so

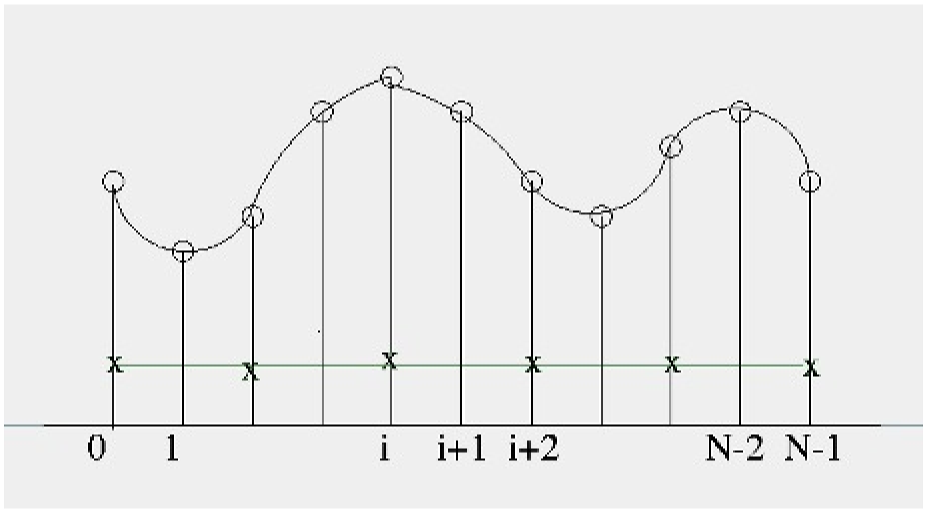
$$I = \int_A^B f(x)\,dx = -\tfrac{1}{2}\left[\exp(-B^2) - \exp(-A^2)\right].$$

Exercise: Can you derive this result?

Exercise: What is the percentage error in the Trapezoidal rule area compared to the exact answer?

2.2 Simpson's Rule

The relatively low accuracy of the Trapezoidal Rule is related the straight-line interpolation used to integrate the function. The more accurate Simpson's rule stems from the use of a *parabola* to fit two adjacent strips at a time. The parabola is constrained to fit the three function values within the two strips.



The following program computes the previous integral via Simpson's rule and compares the result to the Trapezoidal rule (`Simpson.cpp`).

```
//  Read a function on a grid from a file; calculate the area under
//  the curve.

#include <iostream>
#include <cmath>

int main()
{
    double xx[1000], ff[1000];
    int i, j;

    // Read the function from cin.

    j = 0;
    for ( ; ; ) {
        std::cin >> i >> xx[j] >> ff[j] ;
        if ( std::cin.eof() == true )
            break;
        j++;
    }

    // numerical grid
    double dx = xx[1] - xx[0];
    int N = j;

    // Area via Trapezoidal rule.
```

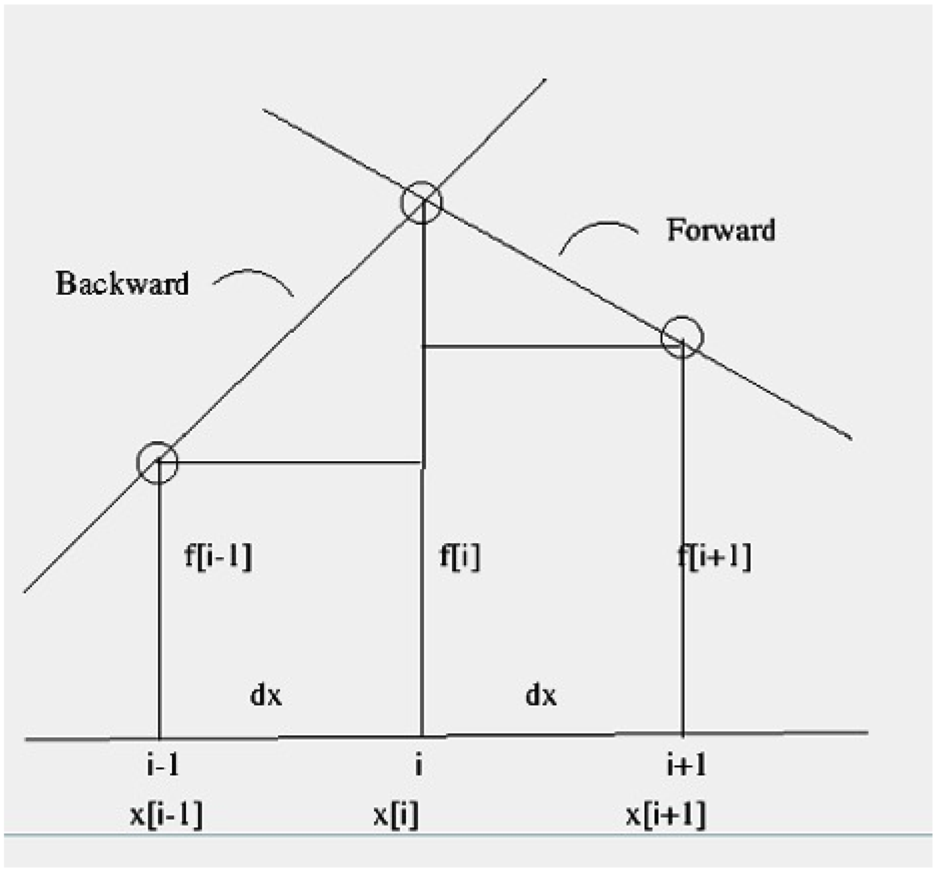
```
        double area = 0.0;
        for( i=0; i<N-2; i++ ) {
            double areastrip = 0.5 * ( ff[i]+ff[i+1] ) * dx;
            area = area + areastrip;
        }
        std::cout << "#   Area (Trapezoidal) = " << area << "\n";

        // Area via Simpson's rule.

        if ( ( N % 2 ) != 1 ) {
            std::cerr << "N must be odd \n";
            return 1;
        }
        area = 0.0;
        for( i=0; i<N-3; i=i+2 ) {
            double areastrip = ( ff[i] + 4.0*ff[i+1] + ff[i+2] ) * dx / 3.0;
            area = area + areastrip;
        }
        std::cout << "#   Area (Simpson)     = " << area << "\n";

        //    Exact area (integral).

        std::cout << "#   Area ( exact )     = " <<
            - ( exp(-xx[N-1]*xx[N-1]) - exp(-xx[0]*xx[0]) ) / 2.0 << " \n" ;
    }
```

Exercise: Is it more accurate?

3 Numerical Derivatives

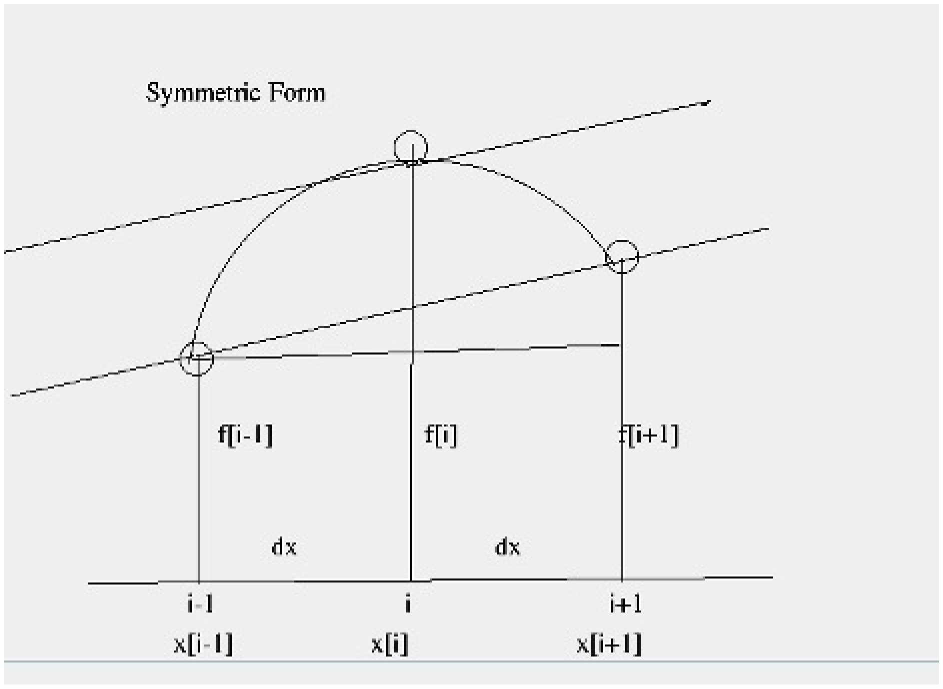
Calculating the derivative of a function is as fundamental as calculating its integral. Some numerical approaches are illustrated below. Be careful, though—numerical differentiation of noisy data will lead to very poor results!

The first figure illustrates three points of a function defined on an equally spaced grid.



A low-order approach follows by using the points i and $i+1$—the Forward formula. The Backward formula follows the same procedure but is based on the points $i-1$ and i. Both formulae for the derivative at i yield the same slope in the limit $\Delta x \to 0$, as discussed in any calculus textbook. However the numerical approach always implies finite Δx, and therefore not so accurate results.

A more accurate way to compute the derivatives is illustrated in the figure below. The so-called Symmetric form uses the function values at points $i+1$ and $i-1$.



The following program (`Derivative.cpp`) reads a tabulated function from the input stream and calculates its numerical derivative using the three numerical forms, Forward, Backward, and Symmetric.

```
// Read a function on a grid from file; calculate the derivative

#include <iostream>
#include <cmath>

int main()
{
    double dx, xx[1000], ff[1000];
    int i, j;
    double Fwd, Bwd, Symm, Exact;

    // Read the function from cin.

    j = 0;
    for ( ; ; )
      {
          std::cin >> i >> xx[j] >> ff[j] ;
          if ( std::cin.eof() == true )
             break;
          j++;
      }
    int N = j;
    // Derivatives (skip first and last points).

    dx =  xx[1] - xx[0];
    for( j=1; j < N-1 ; j++ )
      {
           Fwd = ( ff[j+1] - ff[j] ) / dx;
           Bkwd = ( ff[j] - ff[j-1] ) / dx;
           Symm = ( ff[j+1] - ff[j-1] ) / ( 2.0*dx );
           Exact = ( 1.0 - 2.0 * xx[j] * xx[j] ) * exp( - xx[j]*xx[j] );
           std::cout << j << "  "  << xx[j] << "  " << Fwd << "  "
                     << Bkwd << "  "
                     << Symm << "  " << Exact << "\n";
    }
}
```

The exact derivative of $f(x)$ is also calculated as a reference. It is obvious both graphically and by comparison with the exact answer that the the symmetrical form is the most accurate.

Note that (some of) these formulae do not allow us to calculate the derivatives at the first or last points on the grid. Can you see why?

Note also that the calculation of derivatives is intrinsically not very accurate, and suffers from round-off error as $\Delta x \to 0$.