

Chapter 7

Numerical integration

7.1 Introduction

In this chapter we discuss some of the classic formulae such as the trapezoidal rule and Simpson's rule for equally spaced abscissas and formulae based on Gaussian quadrature. The latter are more suitable for the case where the abscissas are not equally spaced. The emphasis is on methods for evaluating one-dimensional integrals. In chapter 8 we show how Monte Carlo methods can be used to compute multi-dimensional integrals. We discuss also how to compute singular integrals and outline a physics project which combines numerical integration techniques and inverse of a matrix to solve quantum mechanical scattering problems.

We end this chapter with an extensive discussion on MPI and parallel computing. The examples focus on parallelization of algorithms for computing integrals.

The integral

$$I = \int_a^b f(x)dx \quad (7.1)$$

has a very simple meaning. If we consider Fig. 7.1 the integral I simply represents the area enclosed by the function $f(x)$ starting from $x = a$ and ending at $x = b$. Two main methods will be discussed below, the first one being based on equal (or allowing for slight modifications) steps and the other on more adaptive steps, namely so-called Gaussian quadrature methods. Both main methods encompass a plethora of approximations and only some of them will be discussed here.

7.2 Newton-Cotes quadrature: equal step methods

In considering equal step methods, our basic tool is the Taylor expansion of the function $f(x)$ around a point x and a set of surrounding neighbouring points. The algorithm is rather simple, and the number of approximations perhaps unlimited!

- Choose a step size

$$h = \frac{b - a}{N}$$

where N is the number of steps and a and b the lower and upper limits of integration.

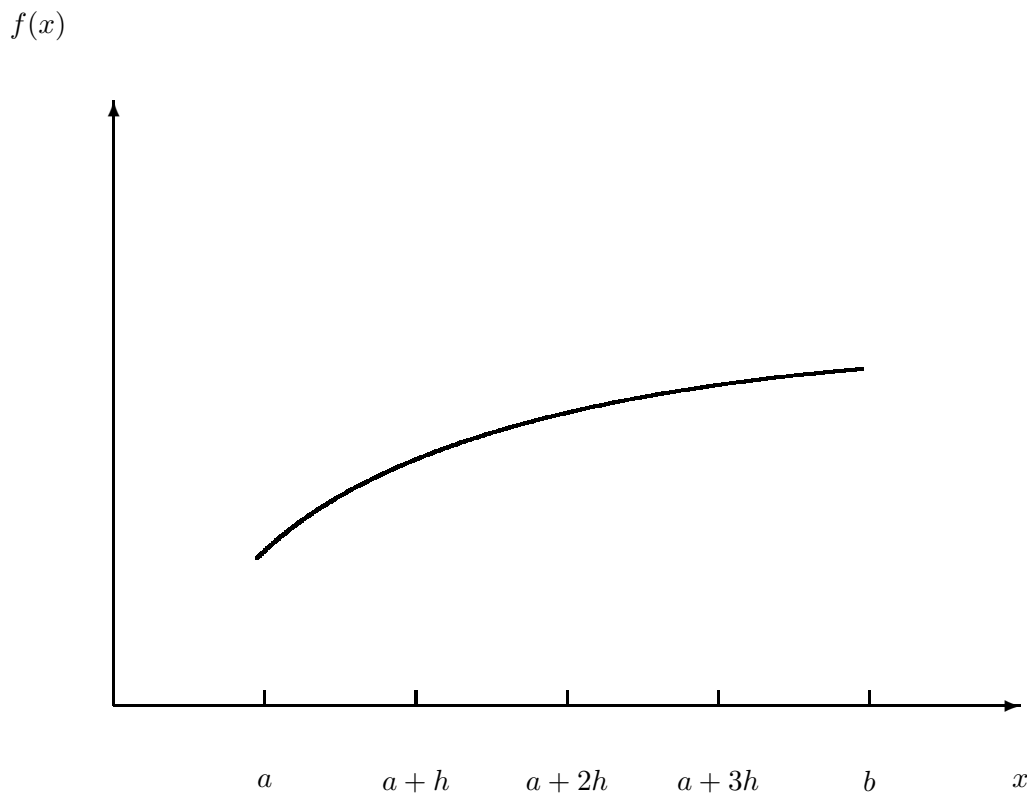


Figure 7.1: The area encribed by the function $f(x)$ starting from $x = a$ to $x = b$. It is subdivided in several smaller areas whose evaluation is to be approximated by the techniques discussed in the text. The areas under the curve can for example be approximated by rectangular boxes or trapezoids.

- With a given step length we rewrite the integral as

$$\int_a^b f(x)dx = \int_a^{a+h} f(x)dx + \int_{a+h}^{a+2h} f(x)dx + \dots + \int_{b-h}^b f(x)dx.$$

- The strategy then is to find a reliable Taylor expansion for $f(x)$ in the various intervals. Choosing a given truncation of the Taylor expansion of $f(x)$ at a certain derivative we obtain a specific approximation to the integral.
- With this approximation to $f(x)$ we perform the integration by computing the integrals over all subintervals.

Such a small measure may seemingly allow for the derivation of various integrals. To see this, let us briefly recall the discussion in the previous section and Fig. 3.1. First we rewrite the integral as

$$\int_a^b f(x)dx = \int_a^{a+2h} f(x)dx + \int_{a+2h}^{a+4h} f(x)dx + \dots + \int_{b-2h}^b f(x)dx.$$

The strategy then is to find a reliable Taylor expansion for $f(x)$ in the smaller sub intervals. Consider for example evaluating

$$\int_a^{a+2h} f(x)dx,$$

which we rewrite as

$$\int_a^{a+2h} f(x)dx = \int_{x_0-h}^{x_0+h} f(x)dx, \tag{7.2}$$

where we will Taylor expand $f(x)$ around a point x_0 , see Fig. 3.1. We have defined $x_0 = a + h$ and use x_0 as the midpoint.

The general form for the Taylor expansion around x_0 goes like

$$f(x = x_0 \pm h) = f(x_0) \pm hf' + \frac{h^2 f''}{2} \pm \frac{h^3 f'''}{6} + O(h^4).$$

Let us now suppose that we split the integral in Eq. (7.2) in two parts, one from $x_0 - h$ to x_0 and the other from x_0 to $x_0 + h$, that is, our integral is rewritten as

$$\int_a^{a+2h} f(x)dx = \int_{x_0-h}^{x_0} f(x)dx + \int_{x_0}^{x_0+h} f(x)dx.$$

Next we assume that we can use the two-point formula for the derivative, meaning that we approximate $f(x)$ in these two regions by a straight line, as indicated in the figure. This means that every small element under the function $f(x)$ looks like a trapezoid. The pertinent numerical approach to the integral bears thus the predictable name 'trapezoidal rule'. It means also that we are trying to approximate our function $f(x)$ with a first order polynomial, that is $f(x) = a + bx$. The constant b is the slope given by the first derivative at $x = x_0$

$$f' = \frac{f(x_0 + h) - f(x_0)}{h} + O(h),$$

or

$$f' = \frac{f(x_0) - f(x_0 - h)}{h} + O(h),$$

and if we stop the Taylor expansion at that point our function becomes,

$$f(x) \approx f_0 + \frac{f_h - f_0}{h}x,$$

for $x \in [x_0, x_0 + h]$ and

$$f(x) \approx f_0 + \frac{f_0 - f_{-h}}{h}x,$$

for $x \in [x_0 - h, x_0]$. The error is proportional with $O(h^2)$. If we then evaluate the integral we obtain

$$\int_{x_0-h}^{x_0+h} f(x)dx = \frac{h}{2} (f_{x_0+h} + 2f_{x_0} + f_{x_0-h}) + O(h^3), \quad (7.3)$$

which is the well-known trapezoidal rule. Concerning the error in the approximation made, $O(h^3) = O((b-a)^3/N^3)$, you should note the following. *This is the local error!* Since we are splitting the integral from a to b in N pieces, we will have to perform approximately N such operations. This means that the *global error* goes like $\approx O(h^2)$. To see that, we use the trapezoidal rule to compute the integral of Eq. (7.1),

$$I = \int_a^b f(x)dx = h (f(a)/2 + f(a+h) + f(a+2h) + \dots + f(b-h) + f_b/2), \quad (7.4)$$

with a global error which goes like $O(h^2)$.

Hereafter we use the shorthand notations $f_{-h} = f(x_0 - h)$, $f_0 = f(x_0)$ and $f_h = f(x_0 + h)$. The correct mathematical expression for the local error for the trapezoidal rule is

$$\int_a^b f(x)dx - \frac{b-a}{2} [f(a) + f(b)] = -\frac{h^3}{12} f^{(2)}(\xi),$$

and the global error reads

$$\int_a^b f(x)dx - T_h(f) = -\frac{b-a}{12} h^2 f^{(2)}(\xi),$$

where T_h is the trapezoidal result and $\xi \in [a, b]$.

The trapezoidal rule is easy to implement numerically through the following simple algorithm

- Choose the number of mesh points and fix the step.
- calculate $f(a)$ and $f(b)$ and multiply with $h/2$
- Perform a loop over $n = 1$ to $n - 1$ ($f(a)$ and $f(b)$ are known) and sum up the terms $f(a+h) + f(a+2h) + f(a+3h) + \dots + f(b-h)$. Each step in the loop corresponds to a given value $a + nh$.
- Multiply the final result by h and add $hf(a)/2$ and $hf(b)/2$.

A simple function which implements this algorithm is as follows

```

double TrapezoidalRule(double a, double b, int n, double (*func)(double))
{
    double TrapezSum;
    double fa, fb, x, step;
    int j;
    step=(b-a)/((double) n);
    fa=(*func)(a)/2. ;
    fb=(*func)(b)/2. ;
    TrapezSum=0.;
    for (j=1; j <= n-1; j++){
        x=j*step+a;
        TrapezSum+=(*func)(x);
    }
    TrapezSum=(TrapezSum+fb+fa)*step ;
    return TrapezSum;
} // end TrapezoidalRule

```

The function returns a new value for the specific integral through the variable **TrapezSum**. There is one new feature to note here, namely the transfer of a user defined function called **func** in the definition

```

void TrapezoidalRule(double a, double b, int n, double *TrapezSum,
                    double (*func)(double) )

```

What happens here is that we are transferring a pointer to the name of a user defined function, which has as input a double precision variable and returns a double precision number. The function **TrapezoidalRule** is called as

```

TrapezoidalRule(a, b, n, &MyFunction )

```

in the calling function. We note that **a**, **b** and **n** are called by value, while **TrapezSum** and the user defined function **MyFunction** are called by reference.

Another very simple approach is the so-called midpoint or rectangle method. In this case the integration area is split in a given number of rectangles with length h and height given by the mid-point value of the function. This gives the following simple rule for approximating an integral

$$I = \int_a^b f(x)dx \approx h \sum_{i=1}^N f(x_{i-1/2}), \quad (7.5)$$

where $f(x_{i-1/2})$ is the midpoint value of f for a given rectangle. We will discuss its truncation error below. It is easy to implement this algorithm, as shown here

```

double RectangleRule(double a, double b, int n, double (*func)(double))
{
    double RectangleSum;
    double fa, fb, x, step;
    int j;
    step=(b-a)/((double) n);
    RectangleSum=0.;
    for (j = 0; j <= n; j++){
        x = (j+0.5)*step+; // midpoint of a given rectangle
        RectangleSum+=(*func)(x); // add value of function.
    }
}

```

```

    }
    RectangleSum *= step; // multiply with step length.
    return RectangleSum;
} // end RectangleRule

```

The correct mathematical expression for the local error for the rectangular rule $R_i(h)$ for element i is

$$\int_{-h}^h f(x)dx - R_i(h) = -\frac{h^3}{24} f^{(2)}(\xi),$$

and the global error reads

$$\int_a^b f(x)dx - R_h(f) = -\frac{b-a}{24} h^2 f^{(2)}(\xi),$$

where R_h is the result obtained with rectangular rule and $\xi \in [a, b]$.

Instead of using the above linear two-point approximations for f , we could use the three-point formula for the derivatives. This means that we will choose formulae based on function values which lie symmetrically around the point where we perform the Taylor expansion. It means also that we are approximating our function with a second-order polynomial $f(x) = a + bx + cx^2$. The first and second derivatives are given by

$$\frac{f_h - f_{-h}}{2h} = f'_0 + \sum_{j=1}^{\infty} \frac{f_0^{(2j+1)}}{(2j+1)!} h^{2j},$$

and

$$\frac{f_h - 2f_0 + f_{-h}}{h^2} = f''_0 + 2 \sum_{j=1}^{\infty} \frac{f_0^{(2j+2)}}{(2j+2)!} h^{2j},$$

and we note that in both cases the error goes like $O(h^{2j})$. With the latter two expressions we can now approximate the function f as

$$f(x) \approx f_0 + \frac{f_h - f_{-h}}{2h} x + \frac{f_h - 2f_0 + f_{-h}}{2h^2} x^2.$$

Inserting this formula in the integral of Eq. (7.2) we obtain

$$\int_{-h}^{+h} f(x)dx = \frac{h}{3} (f_h + 4f_0 + f_{-h}) + O(h^5),$$

which is Simpson's rule. Note that the improved accuracy in the evaluation of the derivatives gives a better error approximation, $O(h^5)$ vs. $O(h^3)$. But this is again the *local error approximation*. Using Simpson's rule we can easily compute the integral of Eq. (7.1) to be

$$I = \int_a^b f(x)dx = \frac{h}{3} (f(a) + 4f(a+h) + 2f(a+2h) + \dots + 4f(b-h) + f_b), \quad (7.6)$$

with a global error which goes like $O(h^4)$. More formal expressions for the local and global errors are for the local error

$$\int_a^b f(x)dx - \frac{b-a}{6} [f(a) + 4f((a+b)/2) + f(b)] = -\frac{h^5}{90} f^{(4)}(\xi),$$

and for the global error

$$\int_a^b f(x)dx - S_h(f) = -\frac{b-a}{180}h^4 f^{(4)}(\xi).$$

with $\xi \in [a, b]$ and S_h the results obtained with Simpson's method. The method can easily be implemented numerically through the following simple algorithm

- Choose the number of mesh points and fix the step.
- calculate $f(a)$ and $f(b)$
- Perform a loop over $n = 1$ to $n - 1$ ($f(a)$ and $f(b)$ are known) and sum up the terms $4f(a + h) + 2f(a + 2h) + 4f(a + 3h) + \dots + 4f(b - h)$. Each step in the loop corresponds to a given value $a + nh$. Odd values of n give 4 as factor while even values yield 2 as factor.
- Multiply the final result by $\frac{h}{3}$.

In more general terms, what we have done here is to approximate a given function $f(x)$ with a polynomial of a certain degree. One can show that given $n + 1$ distinct points $x_0, \dots, x_n \in [a, b]$ and $n + 1$ values y_0, \dots, y_n there exists a unique polynomial $P_n(x)$ with the property

$$p_n(x_j) = y_j \quad j = 0, \dots, n$$

In the Lagrange representation discussed in chapter 6, this interpolating polynomial is given by

$$P_n = \sum_{k=0}^n l_k y_k,$$

with the Lagrange factors

$$l_k(x) = \prod_{\substack{i=0 \\ i \neq k}}^n \frac{x - x_i}{x_k - x_i} \quad k = 0, \dots, n,$$

see for example the text of Kress [30] or Burlich and Stoer [40] for details. If we for example set $n = 1$, we obtain

$$P_1(x) = y_0 \frac{x - x_1}{x_0 - x_1} + y_1 \frac{x - x_0}{x_1 - x_0} = \frac{y_1 - y_0}{x_1 - x_0} x - \frac{y_1 x_0 + y_0 x_1}{x_1 - x_0},$$

which we recognize as the equation for a straight line.

The polynomial interpolatory quadrature of order n with equidistant quadrature points $x_k = a + kh$ and step $h = (b - a)/n$ is called the Newton-Cotes quadrature formula of order n . General expressions can be found in for example Refs. [30, 40].

7.3 Adaptive integration

Before we proceed with more advanced methods like Gaussian quadrature, we mention briefly how an adaptive integration method can be implemented.

The above methods are all based on a defined step length, normally provided by the user, dividing the integration domain with a fixed number of subintervals. This is rather simple to implement may be inefficient, in particular if the integrand varies considerably in certain areas of the integration domain. In these areas the number of fixed integration points may not be adequate. In other regions, the integrand may vary slowly and fewer integration points may be needed.

In order to account for such features, it may be convenient to first study the properties of integrand, via for example a plot of the function to integrate. If this function oscillates largely in some specific domain we may then opt for adding more integration points to that particular domain. However, this procedure needs to be repeated for every new integrand and lacks obviously the advantages of a more generic code.

The algorithm we present here is based on a recursive procedure and allows us to automate an adaptive domain. The procedure is very simple to implement.

Assume that we want to compute an integral using say the trapezoidal rule. We limit ourselves to a one-dimensional integral. Our integration domain is defined by $x \in [a, b]$. The algorithm goes as follows

- We compute our first approximation by computing the integral for the full domain. We label this as $I^{(0)}$. It is obtained by calling our previously discussed function **TrapezoidalRule** as

```
I0 = TrapezoidalRule(a, b, n, function);
```

- In the next step we split the integration in two, with $c = (a + b)/2$. We compute then the two integrals $I^{(1L)}$ and $I^{(1R)}$

```
I1L = TrapezoidalRule(a, c, n, function);
```

and

```
I1R = TrapezoidalRule(c, b, n, function);
```

With a given defined tolerance, being a small number provided by us, we estimate the difference $|I^{(1L)} + I^{(1R)} - I^{(0)}| < \text{tolerance}$. If this test is satisfied, our first approximation is satisfactory.

- If not, we can set up a recursive procedure where the integral is split into subsequent subintervals until our tolerance is satisfied.

This recursive procedure can be easily implemented via the following function

```
// Simple recursive function that implements the
// adaptive integration using the trapezoidal rule
// It is convenient to define as global variables
// the tolerance and the number of recursive steps
const int maxrecursions = 50;
const double tolerance = 1.0E-10;
// Takes as input the integration limits, number of points, function to
// integrate
// and the number of steps
void AdaptiveIntegration(double a, double b, double *Integral, int n, int
steps, double (*func)(double))
```



```

if ( steps > maxrecursions){
    cout << 'Too many recursive steps, the function varies too much' <<
        endl;
    break;
}
double c = (a+b)*0.5;
// the whole integral
double I0 = TrapezoidalRule(a, b,n, func);
// the left half
double I1L = TrapezoidalRule(a, c,n, func);
// the right half
double I1R = TrapezoidalRule(c, b,n, func);
if ( fabs(I1L+I1R-I0) < tolerance ) integral = I0;
else
{
    AdaptiveIntegration(a, c, Integral, int n, ++steps, func)
    AdaptiveIntegration(c, b, Integral, int n, ++steps, func)
}
}
// end function AdaptiveIntegration

```

The variables **Integral** and **steps** should be initialized to zero by the function that calls the adaptive procedure.

7.4 Gaussian quadrature

The methods we have presented hitherto are tailored to problems where the mesh points x_i are equidistantly spaced, x_i differing from x_{i+1} by the step h . These methods are well suited to cases where the integrand may vary strongly over a certain region or if we integrate over the solution of a differential equation.

If however our integrand varies only slowly over a large interval, then the methods we have discussed may only slowly converge towards a chosen precision¹. As an example,

$$I = \int_1^b x^{-2} f(x) dx,$$

may converge very slowly to a given precision if b is large and/or $f(x)$ varies slowly as function of x at large values. One can obviously rewrite such an integral by changing variables to $t = 1/x$ resulting in

$$I = \int_{b^{-1}}^1 f(t^{-1}) dt,$$

which has a small integration range and hopefully the number of mesh points needed is not that large.

However, there are cases where no trick may help and where the time expenditure in evaluating an integral is of importance. For such cases we would like to recommend methods based on Gaussian quadrature. Here one can catch at least two birds with a stone, namely, increased precision and fewer integration points. But it is important that the integrand varies smoothly over the interval, else we have to revert to splitting the interval into many small subintervals and the gain achieved may be lost.

¹You could e.g., impose that the integral should not change as function of increasing mesh points beyond the sixth digit.

The basic idea behind all integration methods is to approximate the integral

$$I = \int_a^b f(x)dx \approx \sum_{i=1}^N \omega_i f(x_i),$$

where ω and x are the weights and the chosen mesh points, respectively. In our previous discussion, these mesh points were fixed at the beginning, by choosing a given number of points N . The weights ω resulted then from the integration method we applied. Simpson's rule, see Eq. (7.6) would give

$$\omega : \{h/3, 4h/3, 2h/3, 4h/3, \dots, 4h/3, h/3\},$$

for the weights, while the trapezoidal rule resulted in

$$\omega : \{h/2, h, h, \dots, h, h/2\}.$$

In general, an integration formula which is based on a Taylor series using N points, will integrate exactly a polynomial P of degree $N - 1$. That is, the N weights ω_n can be chosen to satisfy N linear equations, see chapter 3 of Ref. [3]. A greater precision for a given amount of numerical work can be achieved if we are willing to give up the requirement of equally spaced integration points. In Gaussian quadrature (hereafter GQ), both the mesh points and the weights are to be determined. The points will not be equally spaced². The theory behind GQ is to obtain an arbitrary weight ω through the use of so-called orthogonal polynomials. These polynomials are orthogonal in some interval say e.g., $[-1,1]$. Our points x_i are chosen in some optimal sense subject only to the constraint that they should lie in this interval. Together with the weights we have then $2N$ (N the number of points) parameters at our disposal.

Even though the integrand is not smooth, we could render it smooth by extracting from it the weight function of an orthogonal polynomial, i.e., we are rewriting

$$I = \int_a^b f(x)dx = \int_a^b W(x)g(x)dx \approx \sum_{i=1}^N \omega_i g(x_i), \tag{7.7}$$

where g is smooth and W is the weight function, which is to be associated with a given orthogonal polynomial. Note that with a give weight function we end up evaluating the integrand for the function $g(x_i)$.

The weight function W is non-negative in the integration interval $x \in [a, b]$ such that for any $n \geq 0$ $\int_a^b |x|^n W(x)dx$ is integrable. The naming weight function arises from the fact that it may be used to give more emphasis to one part of the interval than another. A quadrature formula

$$\int_a^b W(x)f(x)dx \approx \sum_{i=1}^N \omega_i f(x_i), \tag{7.8}$$

with N distinct quadrature points (mesh points) is a called a Gaussian quadrature formula if it integrates all polynomials $p \in P_{2N-1}$ exactly, that is

$$\int_a^b W(x)p(x)dx = \sum_{i=1}^N \omega_i p(x_i), \tag{7.9}$$

²Typically, most points will be located near the origin, while few points are needed for large x values since the integrand is supposed to vary smoothly there. See below for an example.

It is assumed that $W(x)$ is continuous and positive and that the integral

$$\int_a^b W(x)dx$$

exists. Note that the replacement of $f \rightarrow Wg$ is normally a better approximation due to the fact that we may isolate possible singularities of W and its derivatives at the endpoints of the interval.

The quadrature weights or just weights (not to be confused with the weight function) are positive and the sequence of Gaussian quadrature formulae is convergent if the sequence Q_N of quadrature formulae

$$Q_N(f) \rightarrow Q(f) = \int_a^b f(x)dx,$$

in the limit $N \rightarrow \infty$. Then we say that the sequence

$$Q_N(f) = \sum_{i=1}^N \omega_i^{(N)} f(x_i^{(N)}),$$

is convergent for all polynomials p , that is

$$Q_N(p) = Q(p)$$

if there exists a constant C such that

$$\sum_{i=1}^N |\omega_i^{(N)}| \leq C,$$

for all N which are natural numbers.

The error for the Gaussian quadrature formulae of order N is given by

$$\int_a^b W(x)f(x)dx - \sum_{k=1}^N w_k f(x_k) = \frac{f^{2N}(\xi)}{(2N)!} \int_a^b W(x)[q_N(x)]^2 dx$$

where q_N is the chosen orthogonal polynomial and ξ is a number in the interval $[a, b]$. We have assumed that $f \in C^{2N}[a, b]$, viz. the space of all real or complex $2N$ times continuously differentiable functions.

In physics there are several important orthogonal polynomials which arise from the solution of differential equations. Well-known examples are the Legendre, Hermite, Laguerre and Chebyshev polynomials. They have the following weight functions

Weight function	Interval	Polynomial
$W(x) = 1$	$x \in [-1, 1]$	Legendre
$W(x) = e^{-x^2}$	$-\infty \leq x \leq \infty$	Hermite
$W(x) = x^\alpha e^{-x}$	$0 \leq x \leq \infty$	Laguerre
$W(x) = 1/(\sqrt{1-x^2})$	$-1 \leq x \leq 1$	Chebyshev

The importance of the use of orthogonal polynomials in the evaluation of integrals can be summarized as follows.

- As stated above, methods based on Taylor series using N points will integrate exactly a polynomial P of degree $N - 1$. If a function $f(x)$ can be approximated with a polynomial of degree $N - 1$

$$f(x) \approx P_{N-1}(x),$$

with N mesh points we should be able to integrate exactly the polynomial P_{N-1} .

- Gaussian quadrature methods promise more than this. We can get a better polynomial approximation with order greater than N to $f(x)$ and still get away with only N mesh points. More precisely, we approximate

$$f(x) \approx P_{2N-1}(x),$$

and with only N mesh points these methods promise that

$$\int f(x)dx \approx \int P_{2N-1}(x)dx = \sum_{i=0}^{N-1} P_{2N-1}(x_i)\omega_i,$$

The reason why we can represent a function $f(x)$ with a polynomial of degree $2N - 1$ is due to the fact that we have $2N$ equations, N for the mesh points and N for the weights.

The mesh points are the zeros of the chosen orthogonal polynomial of order N , and the weights are determined from the inverse of a matrix. An orthogonal polynomials of degree N defined in an interval $[a, b]$ has precisely N distinct zeros on the open interval (a, b) .

Before we detail how to obtain mesh points and weights with orthogonal polynomials, let us revisit some features of orthogonal polynomials by specializing to Legendre polynomials. In the text below, we reserve hereafter the labelling L_N for a Legendre polynomial of order N , while P_N is an arbitrary polynomial of order N . These polynomials form then the basis for the Gauss-Legendre method.

7.4.1 Orthogonal polynomials, Legendre

The Legendre polynomials are the solutions of an important differential equation in physics, namely

$$C(1 - x^2)P - m_l^2 P + (1 - x^2) \frac{d}{dx} \left((1 - x^2) \frac{dP}{dx} \right) = 0.$$

C is a constant. For $m_l = 0$ we obtain the Legendre polynomials as solutions, whereas $m_l \neq 0$ yields the so-called associated Legendre polynomials. This differential equation arises in for example the solution of the angular dependence of Schrödinger's equation with spherically symmetric potentials such as the Coulomb potential.

The corresponding polynomials P are

$$L_k(x) = \frac{1}{2^k k!} \frac{d^k}{dx^k} (x^2 - 1)^k \quad k = 0, 1, 2, \dots,$$

which, up to a factor, are the Legendre polynomials L_k . The latter fulfil the orthogonality relation

$$\int_{-1}^1 L_i(x)L_j(x)dx = \frac{2}{2i+1} \delta_{ij}, \tag{7.10}$$

and the recursion relation

$$(j+1)L_{j+1}(x) + jL_{j-1}(x) - (2j+1)xL_j(x) = 0. \tag{7.11}$$

It is common to choose the normalization condition

$$L_N(1) = 1.$$

With these equations we can determine a Legendre polynomial of arbitrary order with input polynomials of order $N - 1$ and $N - 2$.

As an example, consider the determination of L_0 , L_1 and L_2 . We have that

$$L_0(x) = c,$$

with c a constant. Using the normalization equation $L_0(1) = 1$ we get that

$$L_0(x) = 1.$$

For $L_1(x)$ we have the general expression

$$L_1(x) = a + bx,$$

and using the orthogonality relation

$$\int_{-1}^1 L_0(x)L_1(x)dx = 0,$$

we obtain $a = 0$ and with the condition $L_1(1) = 1$, we obtain $b = 1$, yielding

$$L_1(x) = x.$$

We can proceed in a similar fashion in order to determine the coefficients of L_2

$$L_2(x) = a + bx + cx^2,$$

using the orthogonality relations

$$\int_{-1}^1 L_0(x)L_2(x)dx = 0,$$

and

$$\int_{-1}^1 L_1(x)L_2(x)dx = 0,$$

and the condition $L_2(1) = 1$ we would get

$$L_2(x) = \frac{1}{2}(3x^2 - 1). \tag{7.12}$$

We note that we have three equations to determine the three coefficients a , b and c .

Alternatively, we could have employed the recursion relation of Eq. (7.11), resulting in

$$2L_2(x) = 3xL_1(x) - L_0,$$

which leads to Eq. (7.12).

The orthogonality relation above is important in our discussion on how to obtain the weights and mesh points. Suppose we have an arbitrary polynomial Q_{N-1} of order $N - 1$ and a Legendre polynomial $L_N(x)$ of order N . We could represent Q_{N-1} by the Legendre polynomials through

$$Q_{N-1}(x) = \sum_{k=0}^{N-1} \alpha_k L_k(x), \tag{7.13}$$

where α_k 's are constants.

Using the orthogonality relation of Eq. (7.10) we see that

$$\int_{-1}^1 L_N(x)Q_{N-1}(x)dx = \sum_{k=0}^{N-1} \int_{-1}^1 L_N(x)\alpha_k L_k(x)dx = 0. \quad (7.14)$$

We will use this result in our construction of mesh points and weights in the next subsection.

In summary, the first few Legendre polynomials are

$$\begin{aligned} L_0(x) &= 1, \\ L_1(x) &= x, \\ L_2(x) &= (3x^2 - 1)/2, \\ L_3(x) &= (5x^3 - 3x)/2, \end{aligned}$$

and

$$L_4(x) = (35x^4 - 30x^2 + 3)/8.$$

The following simple function implements the above recursion relation of Eq. (7.11). for computing Legendre polynomials of order N .

```
// This function computes the Legendre polynomial of degree N
double Legendre( int n, double x)
{
    double r, s, t;
    int m;
    r = 0; s = 1.;
    // Use recursion relation to generate p1 and p2
    for (m=0; m < n; m++)
    {
        t = r; r = s;
        s = (2*m+1)*x*r - m*t;
        s /= (m+1);
    } // end of do loop
    return s;
} // end of function Legendre
```

The variable s represents $L_{j+1}(x)$, while r holds $L_j(x)$ and t the value $L_{j-1}(x)$.

7.4.2 Integration points and weights with orthogonal polynomials

To understand how the weights and the mesh points are generated, we define first a polynomial of degree $2N - 1$ (since we have $2N$ variables at hand, the mesh points and weights for N points). This polynomial can be represented through polynomial division by

$$P_{2N-1}(x) = L_N(x)P_{N-1}(x) + Q_{N-1}(x),$$

where $P_{N-1}(x)$ and $Q_{N-1}(x)$ are some polynomials of degree $N - 1$ or less. The function $L_N(x)$ is a Legendre polynomial of order N .

Recall that we wanted to approximate an arbitrary function $f(x)$ with a polynomial P_{2N-1} in order to evaluate

$$\int_{-1}^1 f(x)dx \approx \int_{-1}^1 P_{2N-1}(x)dx,$$

we can use Eq. (7.14) to rewrite the above integral as

$$\int_{-1}^1 P_{2N-1}(x)dx = \int_{-1}^1 (L_N(x)P_{N-1}(x) + Q_{N-1}(x))dx = \int_{-1}^1 Q_{N-1}(x)dx,$$

due to the orthogonality properties of the Legendre polynomials. We see that it suffices to evaluate the integral over $\int_{-1}^1 Q_{N-1}(x)dx$ in order to evaluate $\int_{-1}^1 P_{2N-1}(x)dx$. In addition, at the points x_k where L_N is zero, we have

$$P_{2N-1}(x_k) = Q_{N-1}(x_k) \quad k = 0, 1, \dots, N - 1,$$

and we see that through these N points we can fully define $Q_{N-1}(x)$ and thereby the integral. Note that we have chosen to let the numbering of the points run from 0 to $N - 1$. The reason for this choice is that we wish to have the same numbering as the order of a polynomial of degree $N - 1$. This numbering will be useful below when we introduce the matrix elements which define the integration weights w_i .

We develop then $Q_{N-1}(x)$ in terms of Legendre polynomials, as done in Eq. (7.13),

$$Q_{N-1}(x) = \sum_{i=0}^{N-1} \alpha_i L_i(x). \quad (7.15)$$

Using the orthogonality property of the Legendre polynomials we have

$$\int_{-1}^1 Q_{N-1}(x)dx = \sum_{i=0}^{N-1} \alpha_i \int_{-1}^1 L_0(x)L_i(x)dx = 2\alpha_0,$$

where we have just inserted $L_0(x) = 1!$ Instead of an integration problem we need now to define the coefficient α_0 . Since we know the values of Q_{N-1} at the zeros of L_N , we may rewrite Eq. (7.15) as

$$Q_{N-1}(x_k) = \sum_{i=0}^{N-1} \alpha_i L_i(x_k) = \sum_{i=0}^{N-1} \alpha_i L_{ik} \quad k = 0, 1, \dots, N - 1. \quad (7.16)$$

Since the Legendre polynomials are linearly independent of each other, none of the columns in the matrix L_{ik} are linear combinations of the others. This means that the matrix L_{ik} has an inverse with the properties

$$\hat{\mathbf{L}}^{-1}\hat{\mathbf{L}} = \hat{\mathbf{I}}.$$

Multiplying both sides of Eq. (7.16) with $\sum_{j=0}^{N-1} L_{ji}^{-1}$ results in

$$\sum_{i=0}^{N-1} (L^{-1})_{ki} Q_{N-1}(x_i) = \alpha_k. \quad (7.17)$$

We can derive this result in an alternative way by defining the vectors

$$\hat{\mathbf{x}}_k = \begin{pmatrix} x_0 \\ x_1 \\ \cdot \\ \cdot \\ x_{N-1} \end{pmatrix} \quad \hat{\alpha} = \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \cdot \\ \cdot \\ \alpha_{N-1} \end{pmatrix},$$

and the matrix

$$\hat{\mathbf{L}} = \begin{pmatrix} L_0(x_0) & L_1(x_0) & \dots & L_{N-1}(x_0) \\ L_0(x_1) & L_1(x_1) & \dots & L_{N-1}(x_1) \\ \dots & \dots & \dots & \dots \\ L_0(x_{N-1}) & L_1(x_{N-1}) & \dots & L_{N-1}(x_{N-1}) \end{pmatrix}.$$

We have then

$$Q_{N-1}(\hat{x}_k) = \hat{L}\hat{\alpha},$$

yielding (if \hat{L} has an inverse)

$$\hat{L}^{-1}Q_{N-1}(\hat{x}_k) = \hat{\alpha},$$

which is Eq. (7.17).

Using the above results and the fact that

$$\int_{-1}^1 P_{2N-1}(x)dx = \int_{-1}^1 Q_{N-1}(x)dx,$$

we get

$$\int_{-1}^1 P_{2N-1}(x)dx = \int_{-1}^1 Q_{N-1}(x)dx = 2\alpha_0 = 2 \sum_{i=0}^{N-1} (L^{-1})_{0i} P_{2N-1}(x_i).$$

If we identify the weights with $2(L^{-1})_{0i}$, where the points x_i are the zeros of L_N , we have an integration formula of the type

$$\int_{-1}^1 P_{2N-1}(x)dx = \sum_{i=0}^{N-1} \omega_i P_{2N-1}(x_i)$$

and if our function $f(x)$ can be approximated by a polynomial P of degree $2N - 1$, we have finally that

$$\int_{-1}^1 f(x)dx \approx \int_{-1}^1 P_{2N-1}(x)dx = \sum_{i=0}^{N-1} \omega_i P_{2N-1}(x_i).$$

In summary, the mesh points x_i are defined by the zeros of an orthogonal polynomial of degree N L_N while the weights are given by $2(L^{-1})_{0i}$.

7.4.3 Application to the case $N = 2$

Let us visualize the above formal results for the case $N = 2$. This means that we can approximate a function $f(x)$ with a polynomial $P_3(x)$ of order $2N - 1 = 3$.

The mesh points are the zeros of $L_2(x) = 1/2(3x^2 - 1)$. These points are $x_0 = -1/\sqrt{3}$ and $x_1 = 1/\sqrt{3}$.

Specializing Eq. (7.16)

$$Q_{N-1}(x_k) = \sum_{i=0}^{N-1} \alpha_i L_i(x_k) \quad k = 0, 1, \dots, N - 1.$$

to $N = 2$ yields

$$Q_1(x_0) = \alpha_0 - \alpha_1 \frac{1}{\sqrt{3}},$$

and

$$Q_1(x_1) = \alpha_0 + \alpha_1 \frac{1}{\sqrt{3}},$$

since $L_0(x = \pm 1/\sqrt{3}) = 1$ and $L_1(x = \pm 1/\sqrt{3}) = \pm 1/\sqrt{3}$.

The matrix L_{ik} defined in Eq. (7.16) is then

$$L_{ik} = \begin{pmatrix} 1 & -\frac{1}{\sqrt{3}} \\ 1 & \frac{1}{\sqrt{3}} \end{pmatrix},$$

with an inverse given by

$$(L)_{ik}^{-1} = \frac{\sqrt{3}}{2} \begin{pmatrix} \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \\ -1 & 1 \end{pmatrix}.$$

The weights are given by the matrix elements $2(L_{0k})^{-1}$. We have thence $\omega_0 = 1$ and $\omega_1 = 1$.

Obviously, there is no problem in changing the numbering of the matrix elements $i, k = 0, 1, 2, \dots, N-1$ to $i, k = 1, 2, \dots, N$. We have chosen to start from zero, since we deal with polynomials of degree $N-1$.

Summarizing, for Legendre polynomials with $N = 2$ we have weights

$$\omega : \{1, 1\},$$

and mesh points

$$x : \left\{ -\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}} \right\}.$$

If we wish to integrate

$$\int_{-1}^1 f(x) dx,$$

with $f(x) = x^2$, we approximate

$$I = \int_{-1}^1 x^2 dx \approx \sum_{i=0}^{N-1} \omega_i x_i^2.$$

The exact answer is $2/3$. Using $N = 2$ with the above two weights and mesh points we get

$$I = \int_{-1}^1 x^2 dx = \sum_{i=0}^1 \omega_i x_i^2 = \frac{1}{3} + \frac{1}{3} = \frac{2}{3},$$

the exact answer!

If we were to employ the trapezoidal rule we would get

$$I = \int_{-1}^1 x^2 dx = \frac{b-a}{2} ((a)^2 + (b)^2) / 2 = \frac{1 - (-1)}{2} ((-1)^2 + (1)^2) / 2 = 1!$$

With just two points we can calculate exactly the integral for a second-order polynomial since our method approximates the exact function with higher order polynomial. How many points do you need with the trapezoidal rule in order to achieve a similar accuracy?

7.4.4 General integration intervals for Gauss-Legendre

Note that the Gauss-Legendre method is not limited to an interval $[-1,1]$, since we can always through a change of variable

$$t = \frac{b-a}{2}x + \frac{b+a}{2},$$

rewrite the integral for an interval $[a,b]$

$$\int_a^b f(t)dt = \frac{b-a}{2} \int_{-1}^1 f\left(\frac{(b-a)x}{2} + \frac{b+a}{2}\right) dx.$$

If we have an integral on the form

$$\int_0^\infty f(t)dt,$$

we can choose new mesh points and weights by using the mapping

$$\tilde{x}_i = \tan\left\{\frac{\pi}{4}(1+x_i)\right\},$$

and

$$\tilde{\omega}_i = \frac{\pi}{4} \frac{\omega_i}{\cos^2\left(\frac{\pi}{4}(1+x_i)\right)},$$

where x_i and ω_i are the original mesh points and weights in the interval $[-1, 1]$, while \tilde{x}_i and $\tilde{\omega}_i$ are the new mesh points and weights for the interval $[0, \infty]$.

To see that this is correct by inserting the value of $x_i = -1$ (the lower end of the interval $[-1, 1]$) into the expression for \tilde{x}_i . That gives $\tilde{x}_i = 0$, the lower end of the interval $[0, \infty]$. For $x_i = 1$, we obtain $\tilde{x}_i = \infty$. To check that the new weights are correct, recall that the weights should correspond to the derivative of the mesh points. Try to convince yourself that the above expression fulfils this condition.

7.4.5 Other orthogonal polynomials

Laguerre polynomials

If we are able to rewrite our integral of Eq. (7.7) with a weight function $W(x) = x^\alpha e^{-x}$ with integration limits $[0, \infty]$, we could then use the Laguerre polynomials. The polynomials form then the basis for the Gauss-Laguerre method which can be applied to integrals of the form

$$I = \int_0^\infty f(x)dx = \int_0^\infty x^\alpha e^{-x} g(x)dx.$$

These polynomials arise from the solution of the differential equation

$$\left(\frac{d^2}{dx^2} - \frac{d}{dx} + \frac{\lambda}{x} - \frac{l(l+1)}{x^2}\right) \mathcal{L}(x) = 0,$$

where l is an integer $l \geq 0$ and λ a constant. This equation arises e.g., from the solution of the radial Schrödinger equation with a centrally symmetric potential such as the Coulomb potential. The first few polynomials are

$$\mathcal{L}_0(x) = 1,$$

$$\mathcal{L}_1(x) = 1 - x,$$

$$\mathcal{L}_2(x) = 2 - 4x + x^2,$$

$$\mathcal{L}_3(x) = 6 - 18x + 9x^2 - x^3,$$

and

$$\mathcal{L}_4(x) = x^4 - 16x^3 + 72x^2 - 96x + 24.$$

They fulfil the orthogonality relation

$$\int_{-\infty}^{\infty} e^{-x} \mathcal{L}_n(x)^2 dx = 1,$$

and the recursion relation

$$(n + 1)\mathcal{L}_{n+1}(x) = (2n + 1 - x)\mathcal{L}_n(x) - n\mathcal{L}_{n-1}(x).$$

Hermite polynomials

In a similar way, for an integral which goes like

$$I = \int_{-\infty}^{\infty} f(x) dx = \int_{-\infty}^{\infty} e^{-x^2} g(x) dx.$$

we could use the Hermite polynomials in order to extract weights and mesh points. The Hermite polynomials are the solutions of the following differential equation

$$\frac{d^2 H(x)}{dx^2} - 2x \frac{dH(x)}{dx} + (\lambda - 1)H(x) = 0. \quad (7.18)$$

A typical example is again the solution of Schrödinger's equation, but this time with a harmonic oscillator potential. The first few polynomials are

$$H_0(x) = 1,$$

$$H_1(x) = 2x,$$

$$H_2(x) = 4x^2 - 2,$$

$$H_3(x) = 8x^3 - 12,$$

and

$$H_4(x) = 16x^4 - 48x^2 + 12.$$

They fulfil the orthogonality relation

$$\int_{-\infty}^{\infty} e^{-x^2} H_n(x)^2 dx = 2^n n! \sqrt{\pi},$$

and the recursion relation

$$H_{n+1}(x) = 2xH_n(x) - 2nH_{n-1}(x).$$

Table 7.1: Mesh points and weights for the integration interval [0,100] with $N = 10$ using the Gauss-Legendre method.

i	x_i	ω_i
1	1.305	3.334
2	6.747	7.473
3	16.030	10.954
4	28.330	13.463
5	42.556	14.776
6	57.444	14.776
7	71.670	13.463
8	83.970	10.954
9	93.253	7.473
10	98.695	3.334

7.4.6 Applications to selected integrals

Before we proceed with some selected applications, it is important to keep in mind that since the mesh points are not evenly distributed, a careful analysis of the behavior of the integrand as function of x and the location of mesh points is mandatory. To give you an example, in the Table below we show the mesh points and weights for the integration interval [0,100] for $N = 10$ points obtained by the Gauss-Legendre method. Clearly, if your function oscillates strongly in any subinterval, this approach needs to be refined, either by choosing more points or by choosing other integration methods. Note also that for integration intervals like for example $x \in [0, \infty]$, the Gauss-Legendre method places more points at the beginning of the integration interval. If your integrand varies slowly for large values of x , then this method may be appropriate.

Let us here compare three methods for integrating, namely the trapezoidal rule, Simpson's method and the Gauss-Legendre approach. We choose two functions to integrate:

$$\int_1^{100} \frac{\exp(-x)}{x} dx,$$

and

$$\int_0^3 \frac{1}{2+x^2} dx.$$

A program example which uses the trapezoidal rule, Simpson's rule and the Gauss-Legendre method is included here. For the corresponding Fortran program, replace program1.cpp with program1.f90. The Python program is listed as program1.py.

<http://www.fys.uio.no/compphys/cp/programs/FYS3150/chapter07/cpp/program1.cpp>

```
#include <iostream>
#include "lib.h"
using namespace std;
// Here we define various functions called by the main program
// this function defines the function to integrate
double IntFunction(double x);
// Main function begins here
int main()
```

```

{
    int n;
    double a, b;
    cout << "Read in the number of integration points" << endl;
    cin >> n;
    cout << "Read in integration limits" << endl;
    cin >> a >> b;
    // reserve space in memory for vectors containing the mesh points
    // weights and function values for the use of the gauss-legendre
    // method
    double *x = new double [n];
    double *w = new double [n];
    // set up the mesh points and weights
    GaussLegendre(a, b,x,w, n);
    // evaluate the integral with the Gauss-Legendre method
    // Note that we initialize the sum
    double IntGauss = 0.;
    for ( int i = 0; i < n; i++){
        IntGauss+=w[i]*IntFunction(x[i]);
    }
    // final output
    cout << "Trapez-rule = " << TrapezoidalRule(a, b,n, IntFunction)
        << endl;
    cout << "Simpson's rule = " << Simpson(a, b,n, IntFunction)
        << endl;
    cout << "Gaussian quad = " << IntGauss << endl;
    delete [] x;
    delete [] w;
    return 0;
} // end of main program
// this function defines the function to integrate
double IntFunction(double x)
{
    double value = 4./(1.+x*x);
    return value;
} // end of function to evaluate

```

To be noted in this program is that we can transfer the name of a given function to integrate. In Table 7.2 we show the results for the first integral using various mesh points, while Table 7.3 displays the corresponding results obtained with the second integral. We note here that, since the area over where we

Table 7.2: Results for $\int_1^{100} \exp(-x)/x dx$ using three different methods as functions of the number of mesh points N .

N	Trapez	Simpson	Gauss-Legendre
10	1.821020	1.214025	0.1460448
20	0.912678	0.609897	0.2178091
40	0.478456	0.333714	0.2193834
100	0.273724	0.231290	0.2193839
1000	0.219984	0.219387	0.2193839

integrate is rather large and the integrand goes slowly to zero for large values of x , both the trapezoidal rule and Simpson's method need quite many points in order to approach the Gauss-Legendre method. This integrand demonstrates clearly the strength of the Gauss-Legendre method (and other GQ methods as well), viz., few points are needed in order to achieve a very high precision.

The second Table however shows that for smaller integration intervals, both the trapezoidal rule and Simpson's method compare well with the results obtained with the Gauss-Legendre approach.

Table 7.3: Results for $\int_0^3 1/(2+x^2)dx$ using three different methods as functions of the number of mesh points N .

N	Trapez	Simpson	Gauss-Legendre
10	0.798861	0.799231	0.799233
20	0.799140	0.799233	0.799233
40	0.799209	0.799233	0.799233
100	0.799229	0.799233	0.799233
1000	0.799233	0.799233	0.799233

7.5 Treatment of singular Integrals

So-called principal value (PV) integrals are often employed in physics, from Green's functions for scattering to dispersion relations. Dispersion relations are often related to measurable quantities and provide important consistency checks in atomic, nuclear and particle physics. A PV integral is defined as

$$I(x) = \mathcal{P} \int_a^b dt \frac{f(t)}{t-x} = \lim_{\epsilon \rightarrow 0^+} \left[\int_a^{x-\epsilon} dt \frac{f(t)}{t-x} + \int_{x+\epsilon}^b dt \frac{f(t)}{t-x} \right],$$

and arises in applications of Cauchy's residue theorem when the pole x lies on the real axis within the interval of integration $[a, b]$. Here \mathcal{P} stands for the principal value. *An important assumption is that the function $f(t)$ is continuous on the interval of integration.*

In case $f(t)$ is a closed form expression or it has an analytic continuation in the complex plane, it may be possible to obtain an expression on closed form for the above integral.

However, the situation which we are often confronted with is that $f(t)$ is only known at some points t_i with corresponding values $f(t_i)$. In order to obtain $I(x)$ we need to resort to a numerical evaluation.

To evaluate such an integral, let us first rewrite it as

$$\mathcal{P} \int_a^b dt \frac{f(t)}{t-x} = \int_a^{x-\Delta} dt \frac{f(t)}{t-x} + \int_{x+\Delta}^b dt \frac{f(t)}{t-x} + \mathcal{P} \int_{x-\Delta}^{x+\Delta} dt \frac{f(t)}{t-x},$$

where we have isolated the principal value part in the last integral.

Defining a new variable $u = t - x$, we can rewrite the principal value integral as

$$I_{\Delta}(x) = \mathcal{P} \int_{-\Delta}^{+\Delta} du \frac{f(u+x)}{u}. \tag{7.19}$$

One possibility is to Taylor expand $f(u+x)$ around $u = 0$, and compute derivatives to a certain order as we did for the Trapezoidal rule or Simpson's rule. Since all terms with even powers of u in the Taylor

expansion disappear, we have that

$$I_{\Delta}(x) \approx \sum_{n=0}^{N_{max}} f^{(2n+1)}(x) \frac{\Delta^{2n+1}}{(2n+1)(2n+1)!}.$$

To evaluate higher-order derivatives may be both time consuming and delicate from a numerical point of view, since there is always the risk of losing precision when calculating derivatives numerically. Unless we have an analytic expression for $f(u+x)$ and can evaluate the derivatives in a closed form, the above approach is not the preferred one.

Rather, we show here how to use the Gauss-Legendre method to compute Eq. (7.19). Let us first introduce a new variable $s = u/\Delta$ and rewrite Eq. (7.19) as

$$I_{\Delta}(x) = \mathcal{P} \int_{-1}^{+1} ds \frac{f(\Delta s + x)}{s}. \quad (7.20)$$

The integration limits are now from -1 to 1 , as for the Legendre polynomials. The principal value in Eq. (7.20) is however rather tricky to evaluate numerically, mainly since computers have limited precision. We will here use a subtraction trick often used when dealing with singular integrals in numerical calculations. We introduce first the calculus relation

$$\int_{-1}^{+1} \frac{ds}{s} = 0.$$

It means that the curve $1/s$ has equal and opposite areas on both sides of the singular point $s = 0$.

If we then note that $f(x)$ is just a constant, we have also

$$f(x) \int_{-1}^{+1} \frac{ds}{s} = \int_{-1}^{+1} f(x) \frac{ds}{s} = 0.$$

Subtracting this equation from Eq. (7.20) yields

$$I_{\Delta}(x) = \mathcal{P} \int_{-1}^{+1} ds \frac{f(\Delta s + x)}{s} = \int_{-1}^{+1} ds \frac{f(\Delta s + x) - f(x)}{s}, \quad (7.21)$$

and the integrand is now longer singular since we have that $\lim_{s \rightarrow 0} (f(s+x) - f(x)) = 0$ and for the particular case $s = 0$ the integrand is now finite.

Eq. (7.21) is now rewritten using the Gauss-Legendre method resulting in

$$\int_{-1}^{+1} ds \frac{f(\Delta s + x) - f(x)}{s} = \sum_{i=1}^N \omega_i \frac{f(\Delta s_i + x) - f(x)}{s_i}, \quad (7.22)$$

where s_i are the mesh points (N in total) and ω_i are the weights.

In the selection of mesh points for a PV integral, it is important to use an even number of points, since an odd number of mesh points always picks $s_i = 0$ as one of the mesh points. The sum in Eq. (7.22) will then diverge.

Let us apply this method to the integral

$$I(x) = \mathcal{P} \int_{-1}^{+1} dt \frac{e^t}{t}. \quad (7.23)$$

The integrand diverges at $x = t = 0$. We rewrite it using Eq. (7.21) as

$$\mathcal{P} \int_{-1}^{+1} dt \frac{e^t}{t} = \int_{-1}^{+1} \frac{e^t - 1}{t}, \quad (7.24)$$

since $e^x = e^0 = 1$. With Eq. (7.22) we have then

$$\int_{-1}^{+1} \frac{e^t - 1}{t} \approx \sum_{i=1}^N \omega_i \frac{e^{t_i} - 1}{t_i}. \quad (7.25)$$

The exact results is 2.11450175075..... With just two mesh points we recall from the previous subsection that $\omega_1 = \omega_2 = 1$ and that the mesh points are the zeros of $L_2(x)$, namely $x_1 = -1/\sqrt{3}$ and $x_2 = 1/\sqrt{3}$. Setting $N = 2$ and inserting these values in the last equation gives

$$I_2(x=0) = \sqrt{3} \left(e^{1/\sqrt{3}} - e^{-1/\sqrt{3}} \right) = 2.1129772845.$$

With six mesh points we get even the exact result to the tenth digit

$$I_6(x=0) = 2.11450175075!$$

We can repeat the above subtraction trick for more complicated integrands. First we modify the integration limits to $\pm\infty$ and use the fact that

$$\int_{-\infty}^{\infty} \frac{dk}{k - k_0} = \int_{-\infty}^0 \frac{dk}{k - k_0} + \int_0^{\infty} \frac{dk}{k - k_0} = 0.$$

A change of variable $u = -k$ in the integral with limits from $-\infty$ to 0 gives

$$\int_{-\infty}^{\infty} \frac{dk}{k - k_0} = \int_{\infty}^0 \frac{-du}{-u - k_0} + \int_0^{\infty} \frac{dk}{k - k_0} = \int_0^{\infty} \frac{dk}{-k - k_0} + \int_0^{\infty} \frac{dk}{k - k_0} = 0.$$

It means that the curve $1/(k - k_0)$ has equal and opposite areas on both sides of the singular point k_0 . If we break the integral into one over positive k and one over negative k , a change of variable $k \rightarrow -k$ allows us to rewrite the last equation as

$$\int_0^{\infty} \frac{dk}{k^2 - k_0^2} = 0.$$

We can use this to express a principal values integral as

$$\mathcal{P} \int_0^{\infty} \frac{f(k)dk}{k^2 - k_0^2} = \int_0^{\infty} \frac{(f(k) - f(k_0))dk}{k^2 - k_0^2}, \quad (7.26)$$

where the right-hand side is no longer singular at $k = k_0$, it is proportional to the derivative df/dk , and can be evaluated numerically as any other integral.

Such a trick is often used when evaluating integral equations, as discussed in the next section.

7.6 Scattering equation and principal value integrals

In quantum mechanics, it is often common to rewrite Schrödinger's equation in momentum space, after having made a so-called partial wave expansion of the interaction. We will not go into the details of these expressions but limit ourselves to study the equivalent problem for so-called scattering states, meaning that the total energy of two particles which collide is larger than or equal zero. The benefit of rewriting the equation in momentum space, after having performed a Fourier transformation, is that the coordinate space equation, being an integro-differential equation, is transformed into an integral equation. The latter can be solved by standard matrix inversion techniques. Furthermore, the results of solving these equation can be related directly to experimental observables like the scattering phase shifts. The latter tell us how much the incoming two-particle wave function is modified by a collision. Here we take a more technical stand and consider the technical aspects of solving an integral equation with a principal value.

For scattering states, $E > 0$, the corresponding equation to solve is the so-called Lippman-Schwinger equation. This is an integral equation where we have to deal with the amplitude $R(k, k')$ (reaction matrix) defined through the integral equation

$$R_l(k, k') = V_l(k, k') + \frac{2}{\pi} \mathcal{P} \int_0^\infty dq q^2 V_l(k, q) \frac{1}{E - q^2/m} R_l(q, k'), \quad (7.27)$$

where the total kinetic energy of the two incoming particles in the center-of-mass system is

$$E = \frac{k_0^2}{m}. \quad (7.28)$$

The symbol \mathcal{P} indicates that Cauchy's principal-value prescription is used in order to avoid the singularity arising from the zero of the denominator. We will discuss below how to solve this problem. Equation (7.27) represents then the problem you will have to solve numerically. The interaction between the two particles is given by a partial-wave decomposed version $V_l(k, k')$, where l stands for a quantum number like the orbital momentum. We have assumed that interaction does not coupled to partial waves with different orbital momenta. The variables k and k' are the outgoing and incoming relative momenta of the two interacting particles.

The matrix $R_l(k, k')$ relates to the experimental the phase shifts δ_l through its diagonal elements as

$$R_l(k_0, k_0) = -\frac{\tan \delta_l}{m k_0}, \quad (7.29)$$

where m is the reduced mass of the interacting particles. Furthermore, the interaction between the particles, V , carries

In order to solve the Lippman-Schwinger equation in momentum space, we need first to write a function which sets up the integration points. We need to do that since we are going to approximate the integral through

$$\int_a^b f(x) dx \approx \sum_{i=1}^N w_i f(x_i),$$

where we have fixed N integration points through the corresponding weights w_i and points x_i . These points can for example be determined using Gaussian quadrature.

The principal value in Eq. (7.27) is rather tricky to evaluate numerically, mainly since computers have limited precision. We will here use a subtraction trick often used when dealing with singular integrals in numerical calculations. We use the calculus relation from the previous section

$$\int_{-\infty}^{\infty} \frac{dk}{k - k_0} = 0,$$

or

$$\int_0^\infty \frac{dk}{k^2 - k_0^2} = 0.$$

We can use this to express a principal values integral as

$$\mathcal{P} \int_0^\infty \frac{f(k)dk}{k^2 - k_0^2} = \int_0^\infty \frac{(f(k) - f(k_0))dk}{k^2 - k_0^2}, \quad (7.30)$$

where the right-hand side is no longer singular at $k = k_0$, it is proportional to the derivative df/dk , and can be evaluated numerically as any other integral.

We can then use the trick in Eq. (7.30) to rewrite Eq. (7.27) as

$$R(k, k') = V(k, k') + \frac{2}{\pi} \int_0^\infty dq \frac{q^2 V(k, q) R(q, k') - k_0^2 V(k, k_0) R(k_0, k')}{(k_0^2 - q^2)/m}. \quad (7.31)$$

We are interested in obtaining $R(k_0, k_0)$, since this is the quantity we want to relate to experimental data like the phase shifts.

How do we proceed in order to solve Eq. (7.31)?

1. Using the mesh points k_j and the weights ω_j , we can rewrite Eq. (7.31) as

$$R(k, k') = V(k, k') + \frac{2}{\pi} \sum_{j=1}^N \frac{\omega_j k_j^2 V(k, k_j) R(k_j, k')}{(k_0^2 - k_j^2)/m} - \frac{2}{\pi} k_0^2 V(k, k_0) R(k_0, k') \sum_{n=1}^N \frac{\omega_n}{(k_0^2 - k_n^2)/m}. \quad (7.32)$$

This equation contains now the unknowns $R(k_i, k_j)$ (with dimension $N \times N$) and $R(k_0, k_0)$.

2. We can turn Eq. (7.32) into an equation with dimension $(N + 1) \times (N + 1)$ with an integration domain which contains the original mesh points k_j for $j = 1, N$ and the point which corresponds to the energy k_0 . Consider the latter as the 'observable' point. The mesh points become then k_j for $j = 1, n$ and $k_{N+1} = k_0$.
3. With these new mesh points we define the matrix

$$A_{i,j} = \delta_{i,j} - V(k_i, k_j) u_j, \quad (7.33)$$

where δ is the Kronecker δ and

$$u_j = \frac{2}{\pi} \frac{\omega_j k_j^2}{(k_0^2 - k_j^2)/m} \quad j = 1, N \quad (7.34)$$

and

$$u_{N+1} = -\frac{2}{\pi} \sum_{j=1}^N \frac{k_0^2 \omega_j}{(k_0^2 - k_j^2)/m}. \quad (7.35)$$

The first task is then to set up the matrix A for a given k_0 . This is an $(N + 1) \times (N + 1)$ matrix. It can be convenient to have an outer loop which runs over the chosen observable values for the energy k_0^2/m . *Note that all mesh points k_j for $j = 1, N$ must be different from k_0 . Note also that $V(k_i, k_j)$ is an $(N + 1) \times (N + 1)$ matrix.*

4. With the matrix A we can rewrite Eq. (7.32) as a matrix problem of dimension $(N + 1) \times (N + 1)$. All matrices R , A and V have this dimension and we get

$$A_{i,l}R_{l,j} = V_{i,j}, \quad (7.36)$$

or just

$$AR = V. \quad (7.37)$$

5. Since we already have defined A and V (these are stored as $(N + 1) \times (N + 1)$ matrices) Eq. (7.37) involves only the unknown R . We obtain it by matrix inversion, i.e.,

$$R = A^{-1}V. \quad (7.38)$$

Thus, to obtain R , we need to set up the matrices A and V and invert the matrix A . With the inverse A^{-1} we perform a matrix multiplication with V and obtain R .

With R we can in turn evaluate the phase shifts by noting that

$$R(k_{N+1}, k_{N+1}) = R(k_0, k_0), \quad (7.39)$$

and we are done.

7.7 Parallel computing

We end this chapter by discussing modern supercomputing concepts like parallel computing. In particular, we will introduce you to the usage of the Message Passing Interface (MPI) library. MPI is a library, not a programming language. It specifies the names, calling sequences and results of functions or subroutines to be called from C++ or Fortran programs, and the classes and methods that make up the MPI C++ library. The programs that users write in Fortran or C++ are compiled with ordinary compilers and linked with the MPI library. MPI programs should be able to run on all possible machines and run all MPI implementations without change. An excellent reference is the text by Karniadakis and Kirby II [17].

7.7.1 Brief survey of supercomputing concepts and terminologies

Since many discoveries in science are nowadays obtained via large-scale simulations, there is an everlasting wish and need to do larger simulations using shorter computer time. The development of the capacity for single-processor computers (even with increased processor speed and memory) can hardly keep up with the pace of scientific computing. The solution to the needs of the scientific computing and high-performance computing (HPC) communities has therefore been parallel computing.

The basic ideas of parallel computing is that multiple processors are involved to solve a global problem. The essence is to divide the entire computation evenly among collaborative processors.

Today's supercomputers are parallel machines and can achieve peak performances almost up to 10^{15} floating point operations per second, so-called peta-scale computers, see for example the list over the world's top 500 supercomputers at www.top500.org. This list gets updated twice per year and sets up the ranking according to a given supercomputer's performance on a benchmark code from the LINPACK library. The benchmark solves a set of linear equations using the best software for a given platform.

To understand the basic philosophy, it is useful to have a rough picture of how to classify different hardware models. We distinguish between three major groups, (i) conventional single-processor computers, normally called SISD (single-instruction-single-data) machines, (ii) so-called SIMD machines

(single-instruction-multiple-data), which incorporate the idea of parallel processing using a large number of processing units to execute the same instruction on different data and finally (iii) modern parallel computers, so-called MIMD (multiple-instruction- multiple-data) machines that can execute different instruction streams in parallel on different data. On a MIMD machine the different parallel processing units perform operations independently of each others, only subject to synchronization via a given message passing interface at specified time intervals. MIMD machines are the dominating ones among present supercomputers, and we distinguish between two types of MIMD computers, namely shared memory machines and distributed memory machines. In shared memory systems the central processing units (CPU) share the same address space. Any CPU can access any data in the global memory. In distributed memory systems each CPU has its own memory. The CPUs are connected by some network and may exchange messages. A recent trend are so-called ccNUMA (cache-coherent-non-uniform-memory-access) systems which are clusters of SMP (symmetric multi-processing) machines and have a virtual shared memory.

Distributed memory machines, in particular those based on PC clusters, are nowadays the most widely used and cost-effective, although farms of PC clusters require large infrastructures and yield additional expenses for cooling. PC clusters with Linux as operating systems are easy to setup and offer several advantages, since they are built from standard commodity hardware with the open source software (Linux) infrastructure. The designer can improve performance proportionally with added machines. The commodity hardware can be any of a number of mass-market, stand-alone compute nodes as simple as two networked computers each running Linux and sharing a file system or as complex as thousands of nodes with a high-speed, low-latency network. In addition to the increased speed of present individual processors (and most machines come today with dual cores or four cores, so-called quad-cores) the position of such commodity supercomputers has been strengthened by the fact that a library like MPI has made parallel computing portable and easy. Although there are several implementations, they share the same core commands. Message-passing is a mature programming paradigm and widely accepted. It often provides an efficient match to the hardware.

7.7.2 Parallelism

When we discuss parallelism, it is common to subdivide different algorithms in three major groups.

- **Task parallelism:** the work of a global problem can be divided into a number of independent tasks, which rarely need to synchronize. Monte Carlo simulations and numerical integration are examples of possible applications. Since there is more or less no communication between different processors, task parallelism results in almost a perfect mathematical parallelism and is commonly dubbed embarrassingly parallel (EP). The examples in this chapter fall under that category. The use of the MPI library is then limited to some few function calls and the programming is normally very simple.
- **Data parallelism:** use of multiple threads (e.g., one thread per processor) to dissect loops over arrays etc. This paradigm requires a single memory address space. Communication and synchronization between the processors are often hidden, and it is thus easy to program. However, the user surrenders much control to a specialized compiler. An example of data parallelism is compiler-based parallelization.
- **Message-passing:** all involved processors have an independent memory address space. The user is responsible for partitioning the data/work of a global problem and distributing the subproblems to the processors. Collaboration between processors is achieved by explicit message passing, which is used for data transfer plus synchronization.

This paradigm is the most general one where the user has full control. Better parallel efficiency is usually achieved by explicit message passing. However, message-passing programming is more difficult. We will meet examples of this in connection with the solution eigenvalue problems in chapter 12 and of partial differential equations in chapter 15.

Before we proceed, let us look at two simple examples. We will also use these simple examples to define the speedup factor of a parallel computation. The first case is that of the additions of two vectors of dimension n ,

$$\mathbf{z} = \alpha\mathbf{x} + \beta\mathbf{y},$$

where α and β are two real or complex numbers and $\mathbf{z}, \mathbf{x}, \mathbf{y} \in \mathbb{R}^n$ or $\in \mathbb{C}^n$. For every element we have thus

$$z_i = \alpha x_i + \beta y_i.$$

For every element z_i we have three floating point operations, two multiplications and one addition. If we assume that these operations take the same time Δt , then the total time spent by one processor is

$$T_1 = 3n\Delta t.$$

Suppose now that we have access to a parallel supercomputer with P processors. Assume also that $P \leq n$. We split then these addition and multiplication operations on every processor so that every processor performs $3n/P$ operations in total, resulting in a time $T_P = 3n\Delta t/P$ for every single processor. We also assume that the time needed to gather together these subsums is negligible

If we have a perfect parallelism, our speedup should be P , the number of processors available. We see that this is case by computing the relation between the time used in case of only one processor and the time used if we can access P processors. The speedup S_P is defined as

$$S_P = \frac{T_1}{T_P} = \frac{3n\Delta t}{3n\Delta t/P} = P,$$

a perfect speedup. As mentioned above, we call calculations that yield a perfect speedup for embarrassingly parallel. The efficiency is defined as

$$\eta(P) = \frac{S(P)}{P}.$$

Our next example is that of the inner product of two vectors defined in Eq. (4.5),

$$c = \sum_{j=1}^n x_j y_j.$$

We assume again that $P \leq n$ and define $I = n/P$. Each processor is assigned with its own subset of local multiplications $c_P = \sum_p x_p y_p$, where p runs over all possible terms for processor P . As an example, assume that we have four processors. Then we have

$$\begin{aligned} c_1 &= \sum_{j=1}^{n/4} x_j y_j, & c_2 &= \sum_{j=n/4+1}^{n/2} x_j y_j, \\ c_3 &= \sum_{j=n/2+1}^{3n/4} x_j y_j, & c_4 &= \sum_{j=3n/4+1}^n x_j y_j. \end{aligned}$$

We assume again that the time for every operation is Δt . If we have only one processor, the total time is $T_1 = (2n - 1)\Delta t$. For four processors, we must now add the time needed to add $c_1 + c_2 + c_3 + c_4$, which is $3\Delta t$ (three additions) and the time needed to communicate the local result c_P to all other processors. This takes roughly $(P - 1)\Delta t_c$, where Δt_c need not equal Δt .

The speedup for four processors becomes now

$$S_4 = \frac{T_1}{T_4} = \frac{(2n - 1)\Delta t}{(n/2 - 1)\Delta t + 3\Delta t + 3\Delta t_c} = \frac{4n - 2}{10 + n},$$

if $\Delta t = \Delta t_c$. For $n = 100$, the speedup is $S_4 = 3.62 < 4$. For P processors the inner products yields a speedup

$$S_P = \frac{(2n - 1)}{(2I + P - 2) + (P - 1)\gamma},$$

with $\gamma = \Delta t_c/\Delta t$. Even with $\gamma = 0$, we see that the speedup is less than P .

The communication time Δt_c can reduce significantly the speedup. However, even if it is small, there are other factors as well which may reduce the efficiency η_p . For example, we may have an uneven load balance, meaning that not all the processors can perform useful work at all time, or that the number of processors doesn't match properly the size of the problem, or memory problems, or that a so-called startup time penalty known as latency may slow down the transfer of data. Crucial here is the rate at which messages are transferred

7.7.3 MPI with simple examples

When we want to parallelize a sequential algorithm, there are at least two aspects we need to consider, namely

- Identify the part(s) of a sequential algorithm that can be executed in parallel. This can be difficult.
- Distribute the global work and data among P processors. Stated differently, here you need to understand how you can get computers to run in parallel. From a practical point of view it means to implement parallel programming tools.

In this chapter we focus mainly on the last point. MPI is then a tool for writing programs to run in parallel, without needing to know much (in most cases nothing) about a given machine's architecture. MPI programs work on both shared memory and distributed memory machines. Furthermore, MPI is a very rich and complicated library. But it is not necessary to use all the features. The basic and most used functions have been optimized for most machine architectures

Before we proceed, we need to clarify some concepts, in particular the usage of the words process and processor. We refer to process as a logical unit which executes its own code, in an MIMD style. The processor is a physical device on which one or several processes are executed. The MPI standard uses the concept process consistently throughout its documentation. However, since we only consider situations where one processor is responsible for one process, we therefore use the two terms interchangeably in the discussion below, hopefully without creating ambiguities.

The six most important MPI functions are

- MPI_Init - initiate an MPI computation
- MPI_Finalize - terminate the MPI computation and clean up
- MPI_Comm_size - how many processes participate in a given MPI computation.

- MPI_Comm_rank - which rank does a given process have. The rank is a number between 0 and size-1, the latter representing the total number of processes.
- MPI_Send - send a message to a particular process within an MPI computation
- MPI_Recv - receive a message from a particular process within an MPI computation.

The first MPI C++ program is a rewriting of our 'hello world' program (without the computation of the sine function) from chapter 2. We let every process write "Hello world" on the standard output.

<http://www.fys.uio.no/compphys/cp/programs/FYS3150/MPI/chapter07/program2.cpp>

```
// First C++ example of MPI Hello world
using namespace std;
#include <mpi.h>
#include <iostream>

int main (int nargs, char* args [])
{
    int numprocs, my_rank;
// MPI initializations
    MPI_Init (&nargs, &args);
    MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    cout << "Hello world, I have rank " << my_rank << " out of " <<
        numprocs << endl;
// End MPI
    MPI_Finalize ();
    return 0;
}
```

The corresponding Fortran program reads

```
PROGRAM hello
  INCLUDE "mpif.h"
  INTEGER:: numprocs, my_rank, ierr

  CALL MPI_INIT(ierr)
  CALL MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
  CALL MPI_COMM_RANK(MPI_COMM_WORLD, my_rank, ierr)
  WRITE(*,*)"Hello world, I've rank ",my_rank," out of ",numprocs
  CALL MPI_FINALIZE(ierr)

END PROGRAM hello
```

MPI is a message-passing library where all the routines have a corresponding C++-bindings³ MPI_Command_name or Fortran-bindings (function names are by convention in uppercase, but can also be in lower case) MPI_COMMAND_NAME

To use the MPI library you must include header files which contain definitions and declarations that are needed by the MPI library routines. The following line must appear at the top of any source code file that will make an MPI call. For Fortran you must put in the beginning **INCLUDE** 'mpif.h' while for

³The C++ bindings used in practice are the same as the C bindings, although reading older texts like [15–17] one finds extensive discussions on the difference between C and C++ bindings. Throughout this text we will use the C bindings.

C++ you need to include the statement `#include "mpi.h"`. These header files contain the declarations of functions, variables etc. needed by the MPI library.

The first MPI call must be `MPI_INIT`, which initializes the message passing routines, as defined in for example `INTEGER::ierr` and `CALL MPI_INIT(ierr)` for the Fortran example. The variable `ierr` is an integer which holds an error code when the call returns. The value of `ierr` is however of little use since, by default, MPI aborts the program when it encounters an error. However, `ierr` must be included when MPI starts. For the C++ code we have the call to the function `int MPI_Init(int *argc, char *argv)` where `argc` and `argv` are arguments passed to `main`. MPI does not use these arguments in any way, however, and in MPI-2 implementations, `NULL` may be passed instead. When you have finished you must call the function `MPI_Finalize`. In Fortran you use the statement `CALL MPI_FINALIZE(ierr)` while for C++ we use the function `int MPI_Finalize(void)`.

In addition to these calls, we have also included calls to so-called inquiry functions. There are two MPI calls that are usually made soon after initialization. They are for C++, `MPI_COMM_SIZE` (`(MPI_COMM_WORLD, &numprocs)`) and `CALL MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)` for Fortran. The function `MPI_COMM_SIZE` returns the number of tasks in a specified MPI communicator (`comm` when we refer to it in generic function calls below).

In MPI you can divide your total number of tasks into groups, called communicators. What does that mean? All MPI communication is associated with what one calls a communicator that describes a group of MPI processes with a name (context). The communicator designates a collection of processes which can communicate with each other. Every process is then identified by its rank. The rank is only meaningful within a particular communicator. A communicator is thus used as a mechanism to identify subsets of processes. MPI has the flexibility to allow you to define different types of communicators, see for example [16]. However, here we have used the communicator `MPI_COMM_WORLD` that contains all the MPI processes that are initiated when we run the program.

The variable `numprocs` refers to the number of processes we have at our disposal. The function `MPI_COMM_RANK` returns the rank (the name or identifier) of the tasks running the code. Each task (or processor) in a communicator is assigned a number `my_rank` from 0 to `numprocs - 1`.

We are now ready to perform our first MPI calculations.

Running codes with MPI

To compile and load the above C++ code (after having understood how to use a local cluster), we can use the command

```
mpicxx -O2 -o program2.x program2.cpp
```

and try to run with ten nodes using the command

```
mpiexec -np 10 ./program2.x
```

If we wish to use the Fortran version we need to replace the C++ compiler statement `mpicc` with `mpif90` or equivalent compilers. The name of the compiler is obviously system dependent. The command `mpirun` may be instead of `mpiexec`. Here you need to check your own system.

When we run MPI all processes use the same binary executable version of the code and all processes are running exactly the same code. The question is then how can we tell the difference between our parallel code running on a given number of processes and a serial code? There are two major distinctions you should keep in mind: (i) MPI lets each process have a particular rank to determine which instructions are run on a particular process and (ii) the processes communicate with each other in order to finalize a

task. Even if all processes receive the same set of instructions, they will normally not execute the same instructions. We will exemplify this in connection with our integration example below.

The above example spits out the following output

```
Hello world, I've rank 0 out of 10 procs.
Hello world, I've rank 1 out of 10 procs.
Hello world, I've rank 4 out of 10 procs.
Hello world, I've rank 3 out of 10 procs.
Hello world, I've rank 9 out of 10 procs.
Hello world, I've rank 8 out of 10 procs.
Hello world, I've rank 2 out of 10 procs.
Hello world, I've rank 5 out of 10 procs.
Hello world, I've rank 7 out of 10 procs.
Hello world, I've rank 6 out of 10 procs.
```

The output to screen is not ordered since all processes are trying to write to screen simultaneously. It is then the operating system which opts for an ordering. If we wish to have an organized output, starting from the first process, we may rewrite our program as follows

<http://www.fys.uio.no/compphys/cp/programs/FYS3150/MPI/chapter07/program3.cpp>

```
// Second C++ example of MPI Hello world
using namespace std;
#include <mpi.h>
#include <iostream>

int main (int nargs, char* args [])
{
    int numprocs, my_rank, i;
    // MPI initializations
    MPI_Init (&nargs, &args);
    MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    for (i = 0; i < numprocs; i++) {
        MPI_Barrier (MPI_COMM_WORLD);
        if (i == my_rank) {
            cout << "Hello world, I have rank " << my_rank << " out of " <<
                numprocs << endl;
            fflush (stdout);
        }
    }
    // End MPI
    MPI_Finalize ();
    return 0;
}
```

Here we have used the MPI_Barrier function to ensure that every process has completed its set of instructions in a particular order. A barrier is a special collective operation that does not allow the processes to continue until all processes in the communicator (here MPI_COMM_WORLD) have called MPI_Barrier. The output is now

```
Hello world, I've rank 0 out of 10 procs.
```

Numerical integration

```
Hello world, I've rank 1 out of 10 procs.
Hello world, I've rank 2 out of 10 procs.
Hello world, I've rank 3 out of 10 procs.
Hello world, I've rank 4 out of 10 procs.
Hello world, I've rank 5 out of 10 procs.
Hello world, I've rank 6 out of 10 procs.
Hello world, I've rank 7 out of 10 procs.
Hello world, I've rank 8 out of 10 procs.
Hello world, I've rank 9 out of 10 procs.
```

The barriers make sure that all processes have reached the same point in the code. Many of the collective operations like `MPI_ALLREDUCE` to be discussed later, have the same property; viz. no process can exit the operation until all processes have started. However, this is slightly more time-consuming since the processes synchronize between themselves as many times as there are processes. In the next Hello world example we use the send and receive functions in order to have a synchronized action.

<http://www.fys.uio.no/compphys/cp/programs/FYS3150/MPI/chapter07/program4.cpp>

```
// Third C++ example of MPI Hello world
using namespace std;
#include <mpi.h>
#include <iostream>

int main (int nargs, char* args [])
{
    int numprocs, my_rank, flag;
    // MPI initializations
    MPI_Status status;
    MPI_Init (&nargs, &args);
    MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    // Send and Receive example
    if (my_rank > 0)
        MPI_Recv (&flag, 1, MPI_INT, my_rank-1, 100, MPI_COMM_WORLD, &status)
            ;
        cout << "Hello world, I have rank " << my_rank << " out of " <<
            numprocs << endl;
    if (my_rank < numprocs-1)
        MPI_Send (&my_rank, 1, MPI_INT, my_rank+1, 100, MPI_COMM_WORLD);
    // End MPI
    MPI_Finalize ();
    return 0;
}
```

The basic sending of messages is given by the function `MPI_SEND`, which in C++ is defined as

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
             MPI_Comm comm)
```

while in Fortran we would call this function with the following parameters

```
CALL MPI_SEND(buf, count, MPI_TYPE, dest, tag, comm, ierr).
```

This single command allows the passing of any kind of variable, even a large array, to any group of tasks. The variable `buf` is the variable we wish to send while `count` is the number of variables we are passing. If we are passing only a single value, this should be 1. If we transfer an array, it is the overall size of the array. For example, if we want to send a 10 by 10 array, `count` would be $10 \times 10 = 100$ since we are actually passing 100 values.

We define the type of variable using `MPI_TYPE` in order to let MPI function know what to expect. The destination of the send is declared via the variable `dest`, which gives the ID number of the task we are sending the message to. The variable `tag` is a way for the receiver to verify that it is getting the message it expects. The message tag is an integer number that we can assign any value, normally a large number (larger than the expected number of processes). The communicator `comm` is the group ID of tasks that the message is going to. For complex programs, tasks may be divided into groups to speed up connections and transfers. In small programs, this will more than likely be in `MPI_COMM_WORLD`.

Furthermore, when an MPI routine is called, the Fortran or C++ data type which is passed must match the corresponding MPI integer constant. An integer is defined as `MPI_INT` in C++ and `MPI_INTEGER` in Fortran. A double precision real is `MPI_DOUBLE` in C++ and `MPI_DOUBLE_PRECISION` in Fortran and single precision real is `MPI_FLOAT` in C++ and `MPI_REAL` in Fortran. For further definitions of data types see chapter five of Ref. [16].

Once you have sent a message, you must receive it on another task. The function `MPI_RECV` is similar to the send call. In C++ we would define this as

```
int MPI_Recv( void *buf , int count , MPI_Datatype datatype , int source , int
tag , MPI_Comm comm , MPI_Status *status )
```

while in Fortran we would use the call

```
CALL MPI_RECV(buf , count , MPI_TYPE , source , tag , comm , status , ierr) }
```

The arguments that are different from those in `MPI_SEND` are `buf` which is the name of the variable where you will be storing the received data, `source` which replaces the destination in the send command. This is the return ID of the sender.

Finally, we have used `MPI_Status status`; where one can check if the receive was completed. The source or tag of a received message may not be known if wildcard values are used in the receive function. In C++, MPI Status is a structure that contains further information. One can obtain this information using

```
MPI_Get_count ( MPI_Status *status , MPI_Datatype datatype , int *count ) }
```

The output of this code is the same as the previous example, but now process 0 sends a message to process 1, which forwards it further to process 2, and so forth.

Armed with this wisdom, performed all hello world greetings, we are now ready for serious work.

7.7.4 Numerical integration with MPI

To integrate numerically with MPI we need to define how to send and receive data types. This means also that we need to specify which data types to send to MPI functions.

The program listed here integrates

$$\pi = \int_0^1 dx \frac{4}{1+x^2}$$

by simply adding up areas of rectangles according to the algorithm discussed in Eq. (7.5), rewritten here

$$I = \int_a^b f(x)dx \approx h \sum_{i=1}^N f(x_{i-1/2}),$$

where $f(x) = 4/(1+x^2)$. This is a brute force way of obtaining an integral but suffices to demonstrate our first application of MPI to mathematical problems. What we do is to subdivide the integration range $x \in [0, 1]$ into n rectangles. Increasing n should obviously increase the precision of the result, as discussed in the beginning of this chapter. The parallel part proceeds by letting every process collect a part of the sum of the rectangles. At the end of the computation all the sums from the processes are summed up to give the final global sum. The program below serves thus as a simple example on how to integrate in parallel. We will refine it in the next examples and we will also add a simple example on how to implement the trapezoidal rule.

<http://www.fys.uio.no/compphys/cp/programs/FYS3150/MPI/chapter07/program5.cpp>

```
1  //   Rectangle rule and numerical integration using MPI send and
    Receive
2  using namespace std;
3  #include <mpi.h>
4  #include <iostream>
5
6  int main (int nargs, char* args [])
7  {
8      int numprocs, my_rank, i, n = 1000;
9      double local_sum, rectangle_sum, x, h;
10     //   MPI initializations
11     MPI_Init (&nargs, &args);
12     MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
13     MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
14     //   Read from screen a possible new value of n
15     if (my_rank == 0 && nargs > 1) {
16         n = atoi(args[1]);
17     }
18     h = 1.0/n;
19     //   Broadcast n and h to all processes
20     MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
21     MPI_Bcast (&h, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
22     //   Every process sets up its contribution to the integral
23     local_sum = 0.;
24     for (i = my_rank; i < n; i += numprocs) {
25         x = (i+0.5)*h;
26         local_sum += 4.0/(1.0+x*x);
27     }
28     local_sum *= h;
29     if (my_rank == 0) {
30         MPI_Status status;
31         rectangle_sum = local_sum;
32         for (i=1; i < numprocs; i++) {
33             MPI_Recv(&local_sum, 1, MPI_DOUBLE, MPI_ANY_SOURCE, 500,
MPI_COMM_WORLD, &status);
34             rectangle_sum += local_sum;
```

```

34     }
35     cout << "Result: " << rectangle_sum << endl;
36 }   else
37     MPI_Send(&local_sum, 1, MPI_DOUBLE, 0, 500, MPI_COMM_WORLD);
38 // End MPI
39 MPI_Finalize ();
40 return 0;
41 }

```

After the standard initializations with MPI such as

```
MPI_Init, MPI_Comm_size, MPI_Comm_rank,
```

MPI_COMM_WORLD contains now the number of processes defined by using for example

```
mpirun -np 10 ./prog.x
```

In line 4 we check if we have read in from screen the number of mesh points n . Note that in line 7 we fix $n = 1000$, however we have the possibility to run the code with a different number of mesh points as well. If `my_rank` equals zero, which corresponds to the master node, then we read a new value of n if the number of arguments is larger than two. This can be done as follows when we run the code

```
mpiexec -np 10 ./prog.x 10000
```

In line 17 we define also the step length h . In lines 19 and 20 we use the broadcast function `MPI_Bcast`. We use this particular function because we want data on one processor (our master node) to be shared with all other processors. The broadcast function sends data to a group of processes. The MPI routine `MPI_Bcast` transfers data from one task to a group of others. The format for the call is in C++ given by the parameters of `MPI_Bcast (&n, 1, MPI_INT, 0, MPI_COMM_WORLD);`, `MPI_Bcast (&h, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);` in case of a double. The general structure of this function is `int MPI_Bcast(void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`. All processes call this function, both the process sending the data (with rank zero) and all the other processes in `MPI_COMM_WORLD`. Every process has now copies of n and h , the number of mesh points and the step length, respectively.

We transfer the addresses of n and h . The second argument represents the number of data sent. In case of a one-dimensional array, one needs to transfer the number of array elements. If you have an $n \times m$ matrix, you must transfer $n \times m$. We need also to specify whether the variable type we transfer is a non-numerical such as a logical or character variable or numerical of the integer, real or complex type.

We transfer also an integer variable `int root`. This variable specifies the process which has the original copy of the data. Since we fix this value to zero in the call in lines 19 and 20, it means that it is the master process which keeps this information. For Fortran, this function is called via the statement `CALL MPI_BCAST(buff, count, MPI_TYPE, root, comm, ierr)`.

In lines 23-27, every process sums its own part of the final sum used by the rectangle rule. The receive statement collects the sums from all other processes in case `my_rank == 0`, else an MPI send is performed.

The above function is not very elegant. Furthermore, the MPI instructions can be simplified by using the functions `MPI_Reduce` or `MPI_Allreduce`. The first function takes information from all processes and sends the result of the MPI operation to one process only, typically the master node. If we use `MPI_Allreduce`, the result is sent back to all processes, a feature which is useful when all nodes need the value of a joint operation. We limit ourselves to `MPI_Reduce` since it is only one process which will print out the final number of our calculation, The arguments to `MPI_Allreduce` are the same.

The `MPI_Reduce` function is defined as follows `int MPI_Bcast(void *senddata, void* resultdata, int count, MPI_Datatype datatype, MPI_Op, int root, MPI_Comm comm)`. The two variables `senddata` and

resultdata are obvious, besides the fact that one sends the address of the variable or the first element of an array. If they are arrays they need to have the same size. The variable count represents the total dimensionality, 1 in case of just one variable, while MPI_Datatype defines the type of variable which is sent and received. The new feature is MPI_Op. MPI_Op defines the type of operation we want to do. There are many options, see again Refs. [15–17] for full list. In our case, since we are summing the rectangle contributions from every process we define MPI_Op = MPI_SUM. If we have an array or matrix we can search for the largest or smallest element by sending either MPI_MAX or MPI_MIN. If we want the location as well (which array element) we simply transfer MPI_MAXLOC or MPI_MINLOC. If we want the product we write MPI_PROD. MPI_Allreduce is defined as **int** MPI_Bcast(**void** *senddata, **void** *resultdata, **int** count, MPI_Datatype datatype, MPI_Op, MPI_Comm comm).

The function we list in the next example is the MPI extension of program1.cpp. The difference is that we employ only the trapezoidal rule. It is easy to extend this code to include gaussian quadrature or other methods.

It is also worth noting that every process has now its own starting and ending point. We read in the number of integration points n and the integration limits a and b . These are called a and b . They serve to define the local integration limits used by every process. The local integration limits are defined as $\text{local_a} = a + \text{my_rank} * (b - a) / \text{numprocs}$ and $\text{local_b} = a + (\text{my_rank} - 1) * (b - a) / \text{numprocs}$. These two variables are transferred to the method for the trapezoidal rule. These two methods return the local sum variable local_sum. MPI_Reduce collects all the local sums and returns the total sum, which is written out by the master node. The program below implements this. We have also added the possibility to measure the total time used by the code via the calls to MPI_Wtime.

<http://www.fys.uio.no/compphys/cp/programs/FYS3150/MPI/chapter07/program6.cpp>

```
// Trapezoidal rule and numerical integration using MPI with MPI_Reduce
using namespace std;
#include <mpi.h>
#include <iostream>

// Here we define various functions called by the main program

double int_function(double );
double trapezoidal_rule(double , double , int , double (*)(double));

// Main function begins here
int main (int nargs, char* args[])
{
    int n, local_n, numprocs, my_rank;
    double a, b, h, local_a, local_b, total_sum, local_sum;
    double time_start, time_end, total_time;
    // MPI initializations
    MPI_Init (&nargs, &args);
    MPI_Comm_size (MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);
    time_start = MPI_Wtime();
    // Fixed values for a, b and n
    a = 0.0 ; b = 1.0; n = 1000;
    h = (b-a)/n; // h is the same for all processes
    local_n = n/numprocs; // make sure n > numprocs, else integer division
                        gives zero
    // Length of each process' interval of
```

```

// integration = local_n*h.
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
total_sum = 0.0;
local_sum = trapezoidal_rule(local_a, local_b, local_n, &int_function);
MPI_Reduce(&local_sum, &total_sum, 1, MPI_DOUBLE, MPI_SUM, 0,
          MPI_COMM_WORLD);
time_end = MPI_Wtime();
total_time = time_end-time_start;
if ( my_rank == 0) {
    cout << "Trapezoidal rule = " << total_sum << endl;
    cout << "Time = " << total_time << " on number of processors: " <<
        numprocs << endl;
}
// End MPI
MPI_Finalize ();
return 0;
} // end of main program

// this function defines the function to integrate
double int_function(double x)
{
    double value = 4./(1.+x*x);
    return value;
} // end of function to evaluate

// this function defines the trapezoidal rule
double trapezoidal_rule(double a, double b, int n, double (*func)(double))
{
    double trapez_sum;
    double fa, fb, x, step;
    int j;
    step=(b-a)/((double) n);
    fa=(*func)(a)/2. ;
    fb=(*func)(b)/2. ;
    trapez_sum=0.;
    for (j=1; j <= n-1; j++){
        x=j*step+a;
        trapez_sum+=(*func)(x);
    }
    trapez_sum=(trapez_sum+fb+fa)*step;
    return trapez_sum;
} // end trapezoidal_rule

```

An obvious extension of this code is to read from file or screen the integration variables. One could also use the program library to call a particular integration method.

