Chapter 5 - *Quadrature*

The Goal: Adaptive Quadrature

Historically in mathematics, quadrature refers to the act of trying to find a square with the same area as a given circle.

In mathematical computing, quadrature refers to the numerical approximation of definite integrals.

Let $f(x)$ be a real-valued function of a real variable, defined on a finite interval $a \leq x \leq b$.

We seek to compute the value of the definite integral

$$\int_a^b f(x)\,dx.$$

Note: This is a real number.

In line with its historical meaning, "quadrature" might conjure up the vision of plotting the function on graph paper and counting the little boxes (and their fractions!) that lie underneath the curve.

Observations:

- The use of smaller boxes gives a more accurate answer (but requires more work!).

- It is possible to get away with larger boxes in some regions (where $f(x)$ doesn't change much) and not make a large error.

Adaptive quadrature is the act of carefully selecting the points where $f(x)$ is sampled.

Of course, we want a computer to do this automatically for us!

So, we will have to figure out strategies that can be programmed.

In general, we also want to evaluate the function at as few points as possible while approximating the integral to within some specified accuracy; i.e., adaptive quadrature is meant to give *accurate* answers in an *efficient* manner.

The basis for adaptive quadrature is the following fundamental additive property of the definite integral.

If c is any point between[1] a and b, then

$$\int_a^b f(x)\,dx = \int_a^c f(x)\,dx + \int_c^b f(x)\,dx.$$

The idea of adaptive quadrature is that if we can approximate each of the two integrals on the right to within a specified tolerance, then the sum gives us the desired result.

If not, we can recursively apply the additive property to each of the intervals $[a, c]$ and $[c, b]$; e.g., if the error is too large on $[a, c]$ the we subdivide it into smaller subintervals in order to reduce the overall error.

[1]Actually, the ensuing equation usually is true even if $c \notin [a, b]$ (unless $f(x)$ is misbehaved), but such values of c are not used by adaptive quadrature.

We expect the quadrature strategy to adapt to the integrand automatically, partitioning the interval into subintervals with fine spacing where the integrand is varying rapidly and coarse spacing where the integrand is varying slowly.

However, before we can learn the details of adaptive quadrature, we need to understand the basic quadrature rules.

Basic Quadrature Rules

We begin by deriving of two basic yet useful rules: the midpoint rule and the trapezoidal rule.

Both of these rules take their names from the approximations used to perform the quadrature.

Let $h = b - a$ be the length of the integration interval.

The midpoint rule M approximates the integral by the area of a rectangle whose base has length h and whose height is the value of $f(x)$ at the midpoint:

$$M = hf\left(\frac{a+b}{2}\right).$$

The trapezoidal rule T approximates the integral by the area of a trapezoid with base h and sides equal to the values of $f(x)$ at the two end points.

$$T = h\left(\frac{f(a) + f(b)}{2}\right).$$

What makes one quadrature rule more accurate than another?

We would like to be able to quantify this somehow.

This leads us to the following definition:

The order of a quadrature rule is the degree of the lowest-degree polynomial that the rule does not integrate exactly.

An equivalent interpretation of order (and accuracy) of a quadrature rule is as follows:

If a quadrature rule of order p is used to integrate a smooth function over a small interval of length h, then a Taylor series analysis shows that the error is proportional to h^p.

This means that if h is decreased by a factor of 2, the error decreases by a factor of 2^p.

So, rules with high order converge more quickly than those with low order.

The midpoint rule and the trapezoidal rule are both exact for constant functions and linear functions of x.

This should be obvious for the trapezoidal rule:

The trapezoidal rule forms a *linear interpolant* between $(a, f(a))$ and $(b, f(b))$ and integrates the interpolant exactly to define the rule.

If the underlying function is linear (so that the interpolant and the underlying function are identical), it should not be surprising that the trapezoidal rule gives the exact result.

The order is a little more subtle for the midpoint rule!

Already there is the notion of a free lunch here: the midpoint rule does a *constant* (degree-0) interpolation, yet it can integrate linear (degree-1) functions exactly!

Coming back to reality, neither of them is exact for a quadratic in x, so they both have order 2.

There are two methods that you may have seen but that we did not cover: the so-called *left-hand rectangle rule* and the *right-hand rectangle rule*.

These rules approximate the area under $f(x)$ by rectangles with height $f(a)$ or $f(b)$, respectively.

It can be shown that their orders are only 1, and so they are not good methods to use in practice.

The accuracy of the midpoint and trapezoidal rules can be compared by examining their behavior on the simple integral

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

The midpoint rule gives

$$M = 1\left(\frac{1}{2}\right)^2 = \frac{1}{4}.$$

The trapezoidal rule gives

$$T = 1\frac{0^2 + 1^2}{2} = \frac{1}{2}.$$

So, we see that the error in M is $1/12$, whereas the error in T is $-1/6$.

The errors have opposite signs and, perhaps surprisingly, the midpoint rule is twice as accurate as the trapezoidal rule.

This turns out to not be just coincidence!

When integrating smooth $f(x)$ over short intervals $[a,b]$, M is roughly twice as accurate as T, and the errors have opposite signs.

We can extrapolate on this knowledge as follows:

Knowing these error estimates allows us to combine these two methods and get a rule that is usually more accurate than either one separately!

Let the exact value of the integral be I.

If the error in T were exactly -2 times the error in M, then

$$I - T = -2(I - M).$$

Solving this equation for I would then give us the exact value of the integral!

We settle for creating a new rule S that better approximates I:

$$S = \frac{2}{3}M + \frac{1}{3}T.$$

Indeed, S is usually a more accurate approximation than either M or T alone.

This rule is known as Simpson's rule.

Classically, Simpson's rule is derived by using a parabola to interpolate $f(x)$ at the two endpoints, a and b, and the midpoint, $c = (a+b)/2$ and integrating the parabola exactly.

This yields

$$S = \frac{h}{6}(f(a) + 4f(c) + f(b)).$$

It turns out that S also integrates cubics exactly (another free lunch!), but not quartics, so its order is 4.

So far, we have fit one interpolant to the entire interval $[a, b]$ and integrated it to obtain a quadrature rule.

One way to derive quadrature rules of higher and higher order is to fit polynomial interpolants of higher and higher degree to the integrand $f(x)$ and integrate these interpolants exactly.

However, as we know, high-degree polynomial interpolation based on equally spaced abscissae generically results in oscillatory interpolants that do not reflect the behaviour of the underlying function.

This would degrade the accuracy of the quadrature rules based on them!

With equally spaced abscissae, a better strategy is to use lower-order piecewise polynomial interpolation.

For example, apply Simpson's rule on the two halves of the interval, $[a, c]$ and $[c, b]$.

Let d and e be the midpoints of these two subintervals; i.e., $d = (a + c)/2$ and $e = (c + b)/2$.

Applying Simpson's rule to each subinterval, we obtain a quadrature rule over $[a, b]$:

$$S_2 = \frac{h}{12}(f(a) + 4f(d) + 2f(c) + 4f(e) + f(b)).$$

This is an example of a composite quadrature rule.

As before, the two approximations S and S_2 can be combined to get an even more accurate approximation.

Both rules are of order four, but the step size for S_2 is half the step size for S, so the error in S_2 is roughly $2^4 = 16$ times smaller than that of S.

Thus, a new, more accurate rule Q can be obtained by solving

$$Q - S = 16(Q - S_2).$$

The result is

$$Q = S_2 + \frac{(S_2 - S)}{15}.$$

As an exercise on the mock final asks you to express Q as a weighted combination of the five function values $f(a)$ through $f(e)$ and to establish that its order is six.

The rule is known as *Weddle's rule* or the sixth-order closed *Newton–Cotes* rule or the first step of *Romberg integration*.

We will simply call it the *extrapolated Simpson's rule* because it uses Simpson's rule for two different values of h and then extrapolates toward $h = 0$.

Romberg integration generalizes this to the use of an arbitrary number of different values of h.

An Adaptive Quadrature Strategy

Composite quadrature rules with error estimates can be used to produce *adaptive quadrature methods*.

In general, it is inefficient to use the same mesh on an integrand $f(x)$ that varies rapidly in one region but slowly in another.

$\rightarrow$ The uniform mesh will have a larger error where the integrand varies rapidly.

So if the mesh is uniform, we would have to use a very fine mesh everywhere.

This can be extremely inefficient in regions where $f(x)$ varies slowly and a coarser mesh would be adequate.

Ideally, we would like to equidistribute the error over the whole mesh; i.e., we would like each piece of the composite integration to contribute the same amount of error to the overall error.

Then we would be using a fine or coarse mesh where appropriate.

The basic problem is to approximate $\int_a^b f(x)\,dx$ to within some specified tolerance $\epsilon > 0$.

Suppose we wish to base our adaptive method on Simpson's rule.

Step 1: Apply Simpson's rule with $h = b - a$.

$$\int_a^b f(x)\,dx \approx S_h(a,b),$$

where

$$S_h(a,b) = \frac{h}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right].$$

Step 2: Determine an estimate of the accuracy of the approximation.

The standard way to do this is to halve the mesh and re-solve the problem; i.e., compute $S_{h/2}(a,b)$.

Recalling Weddle's rule (translated to the present notation),

$$Q(a,b) = S_{h/2}(a,b) + E_{h/2},$$

where

$$E_{h/2} = \frac{S_{h/2}(a,b) - S_h(a,b)}{15}.$$

This can be viewed as an approximation $S_{h/2}(a,b)$ to the definite integral, plus an error estimate $E_{h/2}$.

In the case of Weddle's rule, we add the error estimate to the approximation to (hopefully) obtain an even better approximation.

In the case of adaptive quadrature, we use $E_{h/2}$ to decide whether the approximation $S_{h/2}(a,b)$ is accurate enough.

If so, $S_{h/2}(a,b)$ should approximate $\int_a^b f(x)\,dx$ to within ϵ.

If not, we repeat this process on the subintervals $[a,(a+b)/2]$ and $[(a+b)/2,b]$ individually.

However, this time we will compare the error estimates to $\epsilon/2$.

This way, if each of the 2 subintervals has an error of at most $\epsilon/2$, the total error will be at most $\epsilon/2+\epsilon/2=\epsilon$.

Again, if any subinterval reports an error of more than $\epsilon/2$, it is halved again, and its error is recomputed and compared to $\epsilon/4$, etc.

We keep going until the error estimate on every subinterval satisfies the appropriate fraction of ϵ.

Two important notes:

1. The text by Moler argues that this strategy is *too conservative*, leading to too much accuracy.

   In fact, some textbooks advocate simply using the same accuracy tolerance ϵ on every subinterval.

2. This method is not foolproof!

   Integrands with discontinuities, singularities, or misbehaved higher derivatives can cause this method to never terminate.

   However, it tends to work well on many problems, roughly because for each subdivision, the accuracy increases by a factor of 16, but the tolerance is only restricted by a factor of 2.

Example: Evaluate

$$\int_1^3 \frac{100}{x^2} \sin \frac{10}{x} \, dx.$$

See `quadDemo.m`

Specifying Integrands

MATLAB has several different ways of specifying the function to be integrated by a quadrature routine.

The `inline` facility is convenient for simple formulas.

For example,

$$\int_0^1 \frac{dx}{\sqrt{1+x^4}}$$

can be computed with

```
f = inline('1./sqrt(1+x.^4)')
Q = quad(f,0,1)
```

If we want to compute

$$\int_0^\pi \frac{\sin x}{x}\, dx,$$

we could try

```
f = inline('sin(x)./x')
Q = quad(f,0,pi)
```

Unfortunately, this results in a division-by-zero message and an invalid answer.

One way to suppress the warning (but not necessarily to get the correct answer!) is to change the lower limit of integration from 0 to `realmin`.

```
Q = quad(f,realmin,pi)
```

In this case, the error made by this change is many orders of magnitude smaller than roundoff error because $|f(x)| \leq 1$ and the length of the omitted interval is less than 10^{-300}.

Another remedy is to use an M-file instead of the inline function that can handle the problematic point $x = 0$.

We create a file named `sinc.m` that contains

```
function f = sinc(x)

if x == 0
  f = 1;
else
  f = sin(x)/x;
end
```

Then the statement

```
Q = quad(@sinc,0,pi)
```

uses a function handle and computes the integral with no difficulty.

Definite integrals that depend on parameters are encountered frequently.

An example is the beta function, defined by

$$\beta(z, w) = \int_0^1 t^{z-1}(1-t)^{w-1}\, dt.$$

MATLAB already has a beta function, but we can use this example to illustrate how to handle parameters.

Create an inline function with three arguments,

```
F = inline('t.^(z-1).*(1-t).^(w-1)','t','z','w')
```

or create an M-file with a name like `betaf.m`,

```
function f = betaf(t,z,w)
f = t.^(z-1).*(1-t).^(w-1)
```

As with all functions, the order of the arguments is important!

The functions used with quadrature routines must have the variable of integration as the *first* argument.

Values for the parameters are then given as extra arguments to quad.

For example, to compute $\beta(8/3, 10/3)$, you could do

```
z = 8/3;
w = 10/3;
tol = 1e-6;
```

and then use

```
Q = quad(F,0,1,tol,[],z,w);
```

or

```
Q = quad(@betaf,0,1,tol,[],z,w);
```

The “function functions” in MATLAB usually expect the first argument to be in vectorized form.

In other words, the function you input to the function function should be able to accept a vector argument; i.e., it should be written in such a way as to be able to process a bunch of inputs *simultaneously*.

Usually, this means that the mathematical expressions should be specified with MATLAB array (.) notation.

For example,

$$f(x) = \frac{\sin x}{1 + x^2}$$

should be input as

```
sin(x)./(1 + x.^2)
```

Without the (.) operators, the code

```
sin(x)/(1 + x^2)
```

calls for linear algebraic vector operations that are not appropriate here.

The MATLAB function `vectorize` automatically transforms a scalar expression into a form that can be used as input to function functions.

So

```
vectorize('sin(x)/(1+x^2)')
```

produces

```
sin(x)./(1+x.^2)
```

Many of the function functions in MATLAB require the specification of an interval of the x-axis.

Mathematically, we have two notations, $a \leq x \leq b$ or $x \in [a, b]$.

With MATLAB, we also have two possibilities.

The endpoints can be given as two separate arguments, `a` and `b`, or they can be combined into one vector argument, `[a,b]`.

The quadrature functions, `quad`, `quadl`, and `quadgk`, use two separate arguments.

As we have seen, the root-finder `fzero` uses a single argument, but this can be either a single starting point or a two-element vector specifying the interval.

Integrating Discrete Data

So far, we have been concerned with computing an approximation to the definite integral of a specified function $f(x)$.

We have assumed that we can evaluate $f(x)$ at any point in a given interval.

However, in many situations, the function is only known at a discrete set of points, say (x_k, y_k), $k = 1, 2, 3, \ldots, n$. Assume the x_k are sorted in increasing order, with

$$a = x_1 < x_2 < \cdots < x_{n-1} < x_n = b.$$

When approximating $\int_a^b f(x)\,dx$, it is now not possible to sample $f(x)$ wherever we want.

This rules out the use of adaptive quadrature methods!

The most obvious approach is to do piecewise linear interpolation through the data and integrate the interpolant exactly.

i.e., use the *composite trapezoidal rule*

$$T = \sum_{k=1}^{n-1} h_k \frac{y_k + y_{k+1}}{2},$$

where $h_k = x_{k+1} - x_k$.

The composite trapezoidal rule can be implemented in MATLAB in one line:

```
T = sum(diff(x).*(y(1:end-1)+y(2:end))/2)
```

The MATLAB function `trapz` also provides an implementation.

For example, consider the data set

```
x = 1:6
y = [6 8 11 7 5 2]
```

For these data, the trapezoidal rule gives

$$T = 35.$$

The trapezoidal rule is often satisfactory in practice, and more complicated methods may not be necessary.

Nevertheless, methods based on higher-order interpolation can give other estimates of the integral.

Whether or not they are *better* is impossible to decide without further knowledge of the data!

Recall that both the `spline` and `pchip` interpolants are based on piecewise cubic Hermite interpolation:

$$\begin{aligned} P(x) =& \frac{3hs^2 - 2s^3}{h^3} y_{k+1} + \frac{h^3 - 3hs^2 + 2s^3}{h^3} y_k \\ &+ \frac{s^2(s-h)}{h^2} d_{k+1} + \frac{s(s-h)^2}{h^2} d_k, \end{aligned}$$

where $x_k \le x \le x_{k+1}$, $s = x - x_k$, and $h = h_k$.

Recall also that this is a cubic polynomial in s (and hence in x) that satisfies 4 interpolation conditions, 2 on function values and 2 on derivative values:

$$P(x_k) = y_k, \quad P(x_{k+1}) = y_{k+1},$$

$$P'(x_k) = d_k, \quad P'(x_{k+1}) = d_{k+1}.$$

The slopes d_k are computed in `spline` or `pchip`.

As an exercise, show that

$$\int_{x_k}^{x_{k+1}} P(x)\,dx = h_k \frac{y_{k+1} + y_k}{2} - h_k^2 \frac{d_{k+1} - d_k}{12}.$$

Hence

$$\int_a^b P(x)\,dx = T - D,$$

where T is the trapezoidal rule, and

$$D = \sum_{k=1}^{n-1} h_k^2 \, \frac{d_{k+1} - d_k}{12},$$

is a higher-order correction.

Note: If the x_k are equally spaced with spacing h, then D simplifies to

$$D = h^2 \, \frac{d_n - d_1}{12}.$$

For the data shown earlier, the area obtained by `trapz` (piecewise linear interpolation) is 35.00.

By `spline` interpolation, the area is 35.25.

By the `pchip`, the area is 35.41667.

The integration process tends to *average out* the variation in the interpolants, so even though the interpolants might have rather different shapes, the approximations to the integral are often quite close to each other.

High-Powered Tricks and Theory

- **The method of undetermined coefficients**

Quadrature formulas take the form

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

i.e., the definite integral is approximated by a weighted sum of function values.

We have seen how quadrature rules can be derived based on interpolating a set of function values by a polynomial or piecewise polynomial of a given degree and integrating the interpolant exactly.

An alternative derivation to explicitly forming and integrating an interpolant is the method of undetermined coefficients.

Suppose the nodes x_i have been chosen; e.g., they could be n equally spaced points in $[a, b]$, including the end-points.

The idea is that we treat the weights w_i as unknowns and solve for them by forcing the quadrature rule to integrate the usual polynomial basis functions exactly.

If we have n nodes, we have n weights; so, we can require the quadrature rule to exactly integrate polynomials up to degree $n - 1$.

$\rightarrow$ a system of n linear equations in n unknowns.

Example: Derive a 3-point rule that uses $x_1 = a$, $x_2 = (a + b)/2$, and $x_3 = b$.

The quadrature rule is

$$\int_a^b f(x)\,dx \approx w_1 f(a) + w_2 f\left(\frac{a+b}{2}\right) + w_3 f(b).$$

We want the rule to be exact for $f(x) = 1,\ x,\ x^2$.

(Then by linearity, it will be exact for all polynomials of degree ≤ 2.)

$f(x) = 1$:

$$1 \cdot w_1 + 1 \cdot w_2 + 1 \cdot w_3 = \int_a^b 1 \, dx = b - a$$

$f(x) = x$:

$$a \cdot w_1 + \frac{a+b}{2} \cdot w_2 + b \cdot w_3 = \int_a^b x \, dx = \frac{b^2 - a^2}{2}$$

$f(x) = x^2$:

$$a^2 \cdot w_1 + \left(\frac{a+b}{2}\right)^2 \cdot w_2 + b^2 \cdot w_3 = \int_a^b x^2 \, dx = \frac{b^3 - a^3}{3}$$

Solution:

$$w_1 = \frac{b-a}{6}, \quad w_2 = \frac{2(b-a)}{3}, \quad w_3 = \frac{b-a}{6}.$$

This is of course Simpson's rule.

See `undetCoeffs.m`

Try Simpson's rule on $f(x) = x^3$.

Are you surprised by the result?

• **Gaussian quadrature**

In the quadrature rules that we have studied so far, we have always assumed that the integrand is evaluated at the end points and at equally spaced points in between.

There are disadvantages to this!

1. Insisting that the nodes be equally spaced removes degrees of freedom from the method of undetermined coefficients.

   $\rightarrow$ If we remove this restriction, we can derive higher-order quadrature rules for a given number of function evaluations!

2. Integrands are often singular (and yet integrable) at the endpoints.

   $\rightarrow$ Evaluating $f(x)$ at $x = a$ or $x = b$ is problematic, even though the integral itself is well defined.

Recall that when we fixed the x_i, we chose the w_i to integrate exactly the functions $f(x) = 1,\ x,\ x^2,\ \ldots$.

So for n nodes, we expect to integrate exactly polynomials of degree $n-1$.

The key idea behind Gaussian quadrature is to also treat the x_i as unknowns.

Now we have $2n$ degrees of freedom, and we expect to integrate exactly polynomials up to degree $2n-1$.

Example: Derive the Gaussian quadrature rule with $n=2$ and $[a,b]=[-1,1]$.

We are to determine $x_1,\ x_2,\ w_1,\ w_2$ such that

$$\int_{-1}^{1} f(x)\,dx = w_1 f(x_1) + w_2 f(x_2)$$

holds exactly for all polynomials of degree 3 or less.

$$\begin{aligned}
f(x) = 1: &\quad 1 \cdot w_1 + 1 \cdot w_2 &&= \int_{-1}^{1} 1\,dx = 2 \\
f(x) = x: &\quad x_1 \cdot w_1 + x_2 \cdot w_2 &&= \int_{-1}^{1} x\,dx = 0 \\
f(x) = x^2: &\quad x_1^2 \cdot w_1 + x_2^2 \cdot w_2 &&= \int_{-1}^{1} x^2\,dx = \frac{2}{3} \\
f(x) = x^3: &\quad x_1^3 \cdot w_1 + x_2^3 \cdot w_2 &&= \int_{-1}^{1} x^3\,dx = 0
\end{aligned}$$

One solution is

$$x_1 = -\frac{1}{\sqrt{3}},\ x_2 = \frac{1}{\sqrt{3}},\ w_1 = 1,\ w_2 = 1.$$

Thus, the Gaussian quadrature rule takes the form

$$\int_{-1}^{1} f(x)\,dx \approx f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right).$$

It turns out that the abscissae for these maximum-order $(2n)$ quadrature rules are the roots of the nth Legendre polynomials.

Hence, these methods are also sometimes called *Gauss–Legendre* quadrature rules.

Knowledge of the location of the x_i makes the computation of the w_i much easier!

$\rightarrow$ A linear system of size n can be solved for the w_i, as opposed solving to a nonlinear system of size $2n$ for the x_i and w_i simultaneously.

Note: The Legendre polynomials are defined on the interval $[-1, 1]$.

So in general, the definite integral defined on $[a, b]$ is transformed to $[-1, 1]$ to make the evaluations more convenient.

This means we change the variable of integration by means of

$$t = \frac{2x - b - a}{b - a} \iff x = \frac{1}{2}[(b - a)t + (b + a)].$$

Thus,

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

Example: Approximate $\int_0^1 e^{-x^2}\,dx$ by the 2-point Gaussian quadrature rule.

First we transform the interval of integration:

$$a=0,\ b=1 \quad\Longrightarrow\quad x=\frac{1}{2}(t+1).$$

Thus

$$\begin{aligned}\int_0^1 e^{-x^2}\,dx &= \frac{1-0}{2}\int_{-1}^1 e^{-\{\frac{1}{2}(t+1)\}^2}\,dt\\ &\approx \frac{1}{2}\left[e^{-\left\{\frac{1}{2}\left(-\frac{1}{\sqrt{3}}+1\right)\right\}^2}+e^{-\left\{\frac{1}{2}\left(\frac{1}{\sqrt{3}}+1\right)\right\}^2}\right]\\ &= 0.746595.\end{aligned}$$

Note: The exact answer is 0.746824.

This approximation is more accurate than Simpson's rule and uses one less function evaluation!

See `gaussQuad.m`

• Infinite intervals of integration

The problem we consider here is

$$\int_a^b f(x)\,dx,$$

where now $a = -\infty$ and/or $b = \infty$.

There are 3 standard ways to approach such problems.

1. Replace the infinite limit(s) by finite value(s).

   In order to do this sensibly, an error estimate of the neglected "tail(s)" is required.

2. Change variables so that the new interval is finite;

   e.g.,

$$x = -\log t = \log\left(\frac{1}{t}\right) \iff t = e^{-x},$$

$$x = 0 \implies t = 1, \quad x = \infty \implies t = 0,$$

or

$$x = \frac{t}{1-t} \iff t = \frac{x}{x+1},$$
$$x = 0 \implies t = 0, \quad x = \infty \implies t = 1.$$

3. Apply a quadrature rule like Gauss–Laguerre (for $[0, \infty)$) or Gauss–Hermite (for $(-\infty, \infty)$).

We briefly describe Gauss–Laguerre quadrature.

Gauss–Laguerre quadrature rules are designed to evaluate definite integrals

$$\int_0^\infty e^{-x} f(x)\, dx = \sum_{i=1}^{n} w_i f(x_i)$$

by choosing the quadrature weights w_i and nodes x_i such that the quadrature rule is exact for $f(x)$ that are polynomials up to and including degree $2n-1$.

We can derive the 2-stage Gauss–Laguerre quadrature rule using the method of undetermined coefficients.

$$\begin{aligned}
f(x) = 1: &\quad 1 \cdot w_1 + 1 \cdot w_2 &= \int_0^\infty 1 e^{-x}\, dx = 1 \\
f(x) = x: &\quad x_1 \cdot w_1 + x_2 \cdot w_2 &= \int_0^\infty x e^{-x}\, dx = 1 \\
f(x) = x^2: &\quad x_1^2 \cdot w_1 + x_2^2 \cdot w_2 &= \int_0^\infty x^2 e^{-x}\, dx = 2 \\
f(x) = x^3: &\quad x_1^3 \cdot w_1 + x_2^3 \cdot w_2 &= \int_0^\infty x^3 e^{-x}\, dx = 6
\end{aligned}$$

Solution:

$$x_1 = 2 - \sqrt{2},\ x_2 = 2 + \sqrt{2},\ w_1 = \frac{2+\sqrt{2}}{4},\ w_2 = \frac{2-\sqrt{2}}{4}.$$

Implementation note: When solving

$$\int_0^\infty \tilde{f}(x)\,dx,$$

we re-write the integral as

$$\int_0^\infty e^{-x}[e^x\,\tilde{f}(x)]\,dx,$$

and use the quadrature rule in the form

$$\int_0^\infty \tilde{f}(x)\,dx = \sum_{i=1}^{n} w_i e^{x_i}\tilde{f}(x_i);$$

i.e., we identify $f(x)$ with $e^x\tilde{f}(x)$.

Example: Use 2-stage Gauss–Laguerre quadrature to evaluate

$$\int_0^\infty \ln(1+e^{-x})\,dx.$$

See `infiniteInterval.m`

• **Multiple integrals (cubature)**

Multiple integrals are integrals in dimension greater than 1.

Needless to say, the computational effort required to compute definite integrals generally increases greatly with increasing dimension.

Fortunately, definite integrals derived from physical problems are usually limited to 3 or 4 dimensions.

However, applications such as mathematical finance can have integrals in an arbitrary number of dimensions; e.g., depending on how many different financial products are relevant.

There are 3 general approaches for computing multiple integrals.

1. Write the integral as an iterated integral and use 1D quadrature.

   $\rightarrow$ Keeping reasonable tolerance at each stage can be tricky!

This approach works well for regular (e.g., rectangular) domains.

You can use MATLAB's `dblquad` or `triplequad` functions to approximate double and triple integrals, respectively.

The default behaviour is to assume the domain is rectangular.

If it is not, you can enclose the domain in a rectangle and define the integrand to be 0 outside the domain (but still within the enclosing rectangle.)

2. Use a higher-dimensional quadrature rule.

   This is the analogue of how we have been doing quadrature but extended to higher dimensions.

   For example, a quadrature rule in 2D might look like

$$\int_a^b \int_c^d f(x,y)\,dy\,dx \approx \sum_{i=1}^{n} w_i f(x_i, y_i).$$

Again, this approach works well for regular domains; the various rules are derived for specific domains.

3. Use a *Monte Carlo* method.

   This is usually the method of choice for approximating definite integrals in dimensions higher than 2 or 3.

   The idea is to sample the integrand at N points distributed randomly in the domain of integration.

   These values are then averaged and multiplied by the "volume" of the domain to obtain an approximation for the integral.

   It can be shown that such a procedure converges like $N^{-1/2}$, which is pretty slow!

   For example, to gain an extra decimal of accuracy, you will need 100 times more points!

   This makes Monte Carlo uncompetitive for low-dimensional quadrature.

   However, the beauty is that the convergence rate is independent of the number of dimensions!

For example, 10^6 points in 6D is only 10 points per dimension, and this will be much better than any type of conventional quadrature rule for the same type of accuracy.

Also, the software is easy to write and parallelize.

Monte Carlo methods can be made more efficient by biasing the sampling in some way, e.g., concentrating the sampling where the integrand is large or rapidly varying.

$\rightarrow$ This makes it behave like adaptive quadrature.

Example: Estimate

$$\int_0^1 \int_0^1 \frac{1}{1 - xy}\, dy\, dx.$$

Note: Exact value is $\pi^2/6$.

See `multipleIntegral.m`

Shared-Memory Adaptive Quadrature

The natural progression of scientific discovery and industrial research is to continually seek out harder and harder problems to study.

Parallel programming and parallel computers allow us to take better advantage of computing resources to solve *larger* and *more memory-intensive* problems *in less time* than is possible on a single serial computer.

Trends in computer architecture towards multi-core systems make parallel programming and high-performance computing (HPC) a necessary practice going forward.

Computational scientists who ignore HPC do so at their own risk!

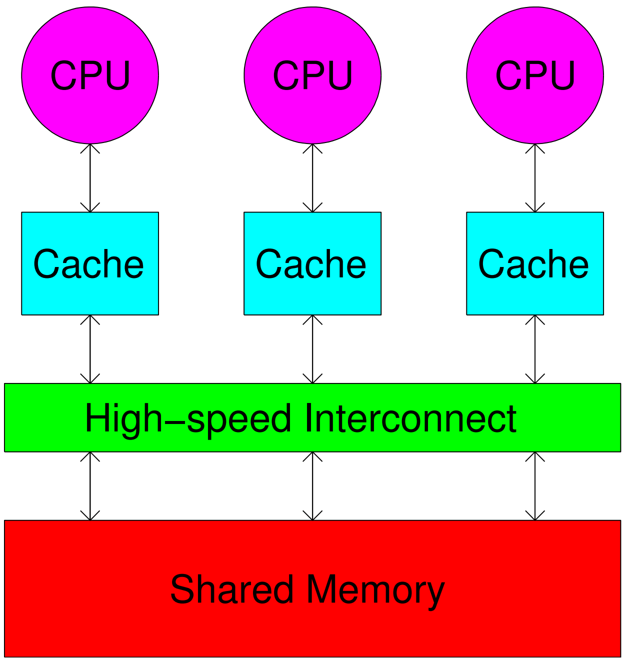
It is already difficult to do leading-edge computational science without HPC, and this situation can only get worse in the future.

The Shared-Memory Paradigm

Parallel computers can be organized in different ways.

In a *shared-memory environment*, each processor is able to read from and write to the same (global) memory.

Each processor (or *node*) also has its own local memory (cache) and runs its own program.



Parallel programming is the act of writing a program for each node, co-ordinating the work that they do.

We assume each node has access to the `quad` function and the integrand `f`.

First, we must initialize any required global variables, i.e., the integration limits `a`, `b` as well as any optional arguments such as tolerances, parameters, etc.

When parallel programs are executed, the number of desired processors p is known in advance.

Suppose they are numbered from 1 to p.

Each processor can find out its local ID number.

We usually view one processor (say the first one) as being the co-ordinating processor (or server); the rest are workers (or clients).

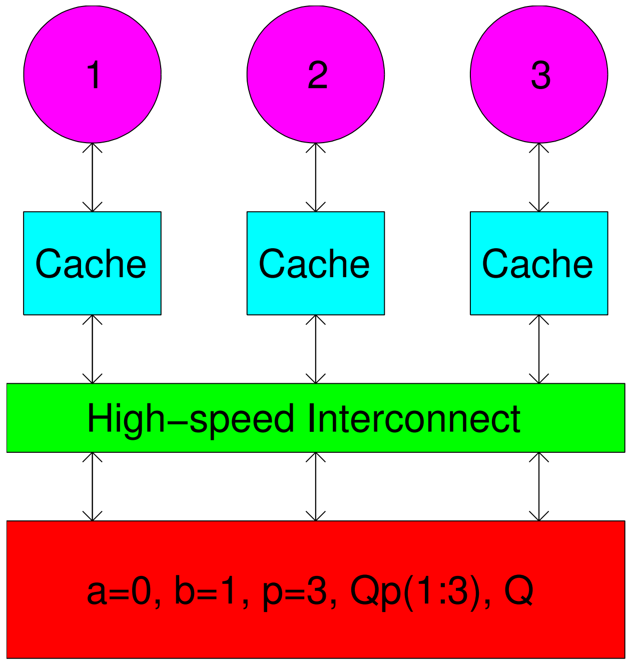
The idea is that we will get each processor to compute part of the integral, and then one processor can sum up the partial results and report the answer.

We will also allocate an array `Qp` of dimension p in global memory for the partial results and one space `Q` for the final answer.

For concreteness, suppose we are trying to compute[2]

$$Q = \int_0^1 \frac{1}{x} \cos\left(\frac{\log(x)}{x}\right)\, dx$$

using $p = 3$ processors.



[2]This is the first problem from the SIAM 100-Digit Challenge.

Static Scheduling

A natural way to partition (or *schedule*) the work among the p processors is to assign

$$Qp(i) = \int_{a+(i-1)h}^{a+ih} f(x)\,dx, \quad h := \frac{b-a}{p},$$

to processor i, $i = 1, 2, \ldots, p$.

The plan is for processor i to work on its part of Q then write its result to $Qp(i)$.

When all processors have finished, processor 1 sums the entries of Qp and reports the final answer Q.

This mapping is called "static" because the assignment of the tasks is fixed before the computation begins.

Because the time taken to solve the problem is governed by the time it takes the slowest processor to finish its task, this strategy works well if the workload can be divided evenly among the processors.

Unfortunately, this is often not the case in quadrature problems.

(If it were, adaptive quadrature would not be so important!)

One way to mitigate this problem is to subdivide the interval $[a, b]$ into rp subintervals and assign r different subintervals to each processor.

For example, if $r = 5$ and $p = 3$, processor 1 would be assigned intervals $1, 4, 7, 10, 13$, processor 2 would be assigned intervals $2, 5, 8, 11, 14$, etc.

The idea is that the processors would share the parts of $[a, b]$ where $\int_a^b f(x)\,dx$ is tough.

Clearly, this becomes more likely for large r.

But if r is too large, some intervals may be computed "too accurately"!

Hence there is a tradeoff in the choice of r.

It seems that imposing a static scheduling on an adaptive quadrature method may be ineffective.

Dynamic Scheduling

In parallel computing, we strive to keep all processors busy doing useful work all the time.

Such a computation is said to be *load-balanced*.

An idle processor is a wasted resource!

We would like a way for processors with easy assignments to help out with the hard parts of the integration when they are done.

The most common way to do this is instead of fixing the assignment of the rp intervals, we allocate assignments on a first-come, first-served basis.

For example, we begin by assigning interval i to processor i for $i = 1, 2, \ldots, p$.

Then interval $p + 1$ is assigned to whichever processor is done its initial assignment first, interval $p + 2$ is assigned to the first processor done its assignment after that, etc.

This process continues until all rp intervals have been assigned.

In this way, processors that have easy assignments and are done quickly can return to the "assignment pool" (or *queue*) to get a new assignment.

There is still no guarantee of good performance (e.g., interval rp turns out to be extremely difficult), but it is likely that with some foresight extremely poor performance can be avoided.

Care must be taken to ensure more than one processor is not assigned the same job!

This means there may be idling while processors determine which interval is to be handled next.

A more esoteric way to accomplish load balancing is to have the roles of servers and clients be dynamic, so a client with a subinterval that is too difficult can act as the server for that subinterval.