# 2 Numerical integration

Numerically integrate some function $f(x)$, over a closed domain $[a, b]$. There are many possibilities for form of $f$, e.g.,:

- We may have an analytic function form for $f$ – can evaluate at any $x$ simply
- We may have a numerical approximation for $f$ (e.g., result of solving differential equation) – $f(x)$ values stored as an array, for only fixed values of $x$ ($f[0] = f(x_0)$, $f[1] = f(x_1)$ etc).
- $f$ may be easy, or very difficult (numerically intensive) to evaluate at any given point
- $f$ may be smooth and well-behaved, or be highly unpredictable or quickly oscillating

The method we choose to integrate will depend on these factors; understanding how different methods work, and their pros and cons, will allow you to make this decision wisely.

## 2.1 Simplest case

Riemann sum: $N$ discrete points distributed equally along the closed interval $[a, b]$ – i.e., $N - 1$ sub-intervals of equal width. We call these points sample points, integration points, or grid points. Break function region into $N-1$ equal-width rectangles; area under curve is sum of area of rectangles, giving:

$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_{i=0}^{N-1} f(x_i)\,\delta x, \tag{1}$$

where

$$\delta x = \frac{b-a}{N-1}, \qquad x_i = a + i\,\delta x.$$

This is exact as $N \to \infty$, however, for finite $N$ it counts end-points twice. To avoid, we can choose left/right Riemann sum (by dropping the last/first point), or using mid-point.

Note: many places use '$N$' to denote the number of regions/sub-intervals; I use it to denote number of *points*, since this is more natural for coding algorithms. It usually doesn't matter, but pays to not get confused with off-by-one errors.

**Trapezoid rule:** Higher accuracy can be gained by approximating the area of each segment by a trapezoid. For each trapezoid, the area is:$[f(x) + f(x + \delta x)] * \delta x/2$. More generally, for $N$ equally-spaced integration points:

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{[f(x_0) + f(x_{N-1})]\delta x}{2} + \sum_{i=1}^{N-2} f(x_i)\,\delta x. \tag{2}$$

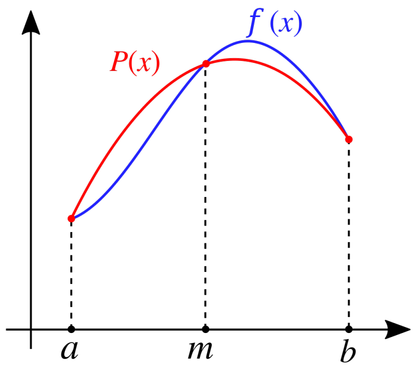

Figure 1: Simpsons rule

### 2.1.1 Simpson's rule

For each sub-interval, the area can be approximated more accurately again using Simpson's rule

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{b-a}{6}[f(a) + f(\tfrac{(a+b)}{2}) + f(b)]. \tag{3}$$

This rule is chosen so that it is exact for quadratic polynomials. It is equivalent to fitting a quadratic polynomial to the function in the region $(a, b)$, and integrating the polynomial exactly, and can be derived that way (see next section). It can also be derived simply by evaluating

$$\int_a^b f(x)\,\mathrm{d}x = \alpha f(a) + \beta f(\tfrac{(a+b)}{2}) + \gamma f(b), \tag{4}$$

for $f(x) = \{1, x, x^2\}$ – gives three equations with three unknowns; solve for $\alpha, \beta, \gamma$.

More generally, for $N$ equally-spaced integration points:

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{\delta x}{3}\left[f(x_0) + f(x_{N-1}) + 2\sum_{j=1}^{\frac{(n-1)}{2}-1} f(x_{2j}) + 4\sum_{j=1}^{\frac{(n-1)}{2}} f(x_{2j-1})\right] \tag{5}$$

$$= \frac{\delta x}{3}\left[f(x_0) + f(x_{N-1}) + 2\sum_{\text{even } j} f(x_j) + 4\sum_{\text{even } j} f(x_j)\right] \tag{6}$$

- $N$ must be odd (even number of intervals)
- Exact for order-2 polynomial
- Gives Runge-Kutta 4 (see last lecture)

In Example 2.1, we show various approximations for integral of

$$f(x) = e^{-x}\sin(5x)$$

over [0,5]. For this function, with $N = 101$, the trapezoid rule leads to 0.5% error, while Simpson's rule leads to 0.002% error.

**Note:** Links presented above the code examples marked – Interactive version: – take you to a CompilerExplorer page (godbolt.org) where you can play with the code.

Example 2.1: Trapezoid and Simpson rules. Interactive version: godbolt.org/z/WhjbeKcWc

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

// Function, to integrate
double f(double x) { return std::exp(-x) * std::sin(5.0 * x); }

int main() {
  double exact = 0.1910576315007305;

  // Number of integration points:
  int n = 101;
  // Domain over which to integrate:
  double a = 0.0;
  double b = 5.0;
  double dx = (b - a) / (n - 1);

  std::cout << "\nTrapazoid rule:\n";
  double int2 = 0.5 * (f(a) + f(b)) * dx;
  for (int i = 1; i < n - 1; ++i) {
    double x = a + i * dx;
    int2 += f(x) * dx;
  }
  std::cout << "Error: " << 100.0 * (int2 - exact) / exact << "%\n";

  std::cout << "\nSimspons rule (n must be odd):\n";
  double int3 = (f(a) + f(b)) * dx / 3.0;
  for (int i = 1; i < n - 1; ++i) {
    double w = (i % 2 == 0) ? (2.0 / 3) : (4.0 / 3); // weight
    int3 += f(a + i * dx) * dx * w;
  }
  std::cout << "Error: " << 100.0 * (int3 - exact) / exact << "%\n";
}
```

## 2.2 Newton-Cotes (general polynomial interpolation)

Approximate $f(x) \approx P(x)$ for some general polynomial $P$. This can be done in several ways, but a common method is to use Lagrange basis polynomials $l^{(k)}(x)$;

$$f(x) \approx P(x) = \sum_{j=0}^{k} f(x_j)\, l_j^{(k)}(x), \tag{7}$$

where

$$l_j^{(k)}(x) = \prod_{\substack{m=0\\ m\neq j}}^{k} \frac{x - x_m}{x_j - x_m}.$$

This interpolating function takes a set of points $\{(x_0,\ f(x_0)), \dots (x_k,\ f(x_k))\}$ and constructs a polynomial that interpolates the given function. This isn't proved here, but proof can be easily found online (not important for our discussion).

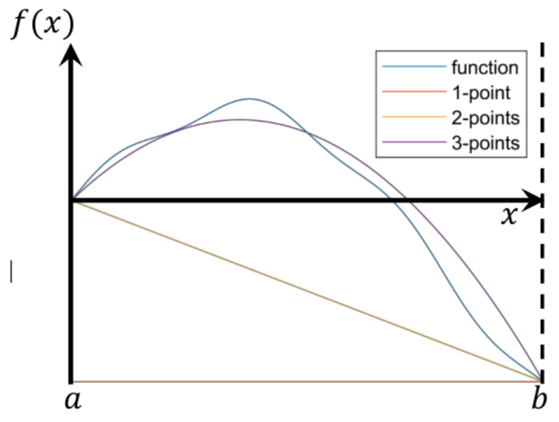

Figure 2: Newton-Cotes: polynomial interpolation (order $k$ means $k+1$-points)

Since the polynomial can be integrated exactly, this allows a good approximation for integration:

$$\int_a^b f(x)\,\mathrm{d}x = \sum_{j=0}^{k} f(x_j)\underbrace{\int_a^b l_j(x)\,\mathrm{d}x}_{w_j} \tag{8}$$

$$= \sum_{j=0}^{k} f(x_j)\,w_j. \tag{9}$$

The $w_i$ are known as weights, and the integration reduces to calculating these weights.

A major drawback of this technique is from over-fitting – becomes unstable for large $k$ – so $k$ must be chosen wisely; typically only small $k$ is used. If higher accuracy is required, techniques discussed below are used.

- $k = 0$ gives simple Riemann sum
- $k = 1$ gives trapezoid rule
- $k = 2$ gives Simpson's rule
- Unstable for large $k$

## 2.3 Gaussian Quadrature

In Newton-Cotes, polynomial interpolation with equally-spaced points $x_j$. In more general Gaussian quadrature, we write

$$\int_{-1}^{1} f(x)\,\mathrm{d}x = \sum_{j=1}^{k} f(x_j)\,w_j, \tag{10}$$

and now find both the weights $w_j$, and the sample-points (nodes) $x_j$ that give the best approximation. These are chosen so that the approximation is exact for polynomials of degree $2k-1$. The formulas are typically defined on the domain $[-1,\,1]$ – more general $[a,\,b]$ domain can be found via simple change of variables. More complex, but more accurate than Newton-Cotes formulas. However, since they rely on non-uniformly spaces nodes $x_j$ (which are not generally known before-hand), the method can be difficult to employ for cases where the function $f$ is only known at certain points, e.g., $f[i]$.

The weights and nodes are found by solving Eq. (10) for each polynomial $f(x) = \{1,\, x,\, x^2,\, \dots x^{2k-1}\}$. For example, for $k = 2$, we take $f(x) = \{1,\, x,\, x^2,\, x^3\}$, giving:

$$\begin{aligned} 2 &= w_1 + w_2, \\ 0 &= w_1 x_1 + w_2 x_2, \\ \frac{2}{3} &= w_1 x_1^2 + w_2 x_2^2, \\ 0 &= w_1 x_1^3 + w_2 x_2^3, \end{aligned} \tag{11}$$

which gives $x_1 = -x_2 = 1/\sqrt{3}$, and $w_1 = w_2 = 1$. The results are more complex for higher orders. In general, we get $2k$ equations with $2k$ unknowns, allowing us to solve for $x_j$ and $w_j$.

There are many libraries available that implement general-order Gaussian quadrature integration algorithms. Libraries useful for C and C++ code available through GSL (GNU Scientific Libraries) – https://www.gnu.org/software/gsl/doc/html/integration.html (see 'fixed-point-quadratures')

- Exact for polynomials of degree up to $2k - 1$
- Very accurate for smooth functions (well-approximated by polynomial/Taylor series)
- More complicated: but implemented in many libraries
- Not always practical

## 2.4 Mixed quadrature rules

Often: most inaccuracy in integrals comes from end-points - error in middle part tends to cancel. We want high-accuracy, but also simple (and therefore fast/efficient) method. Particularly, we want high accuracy in situations where we only know function value as discrete points $f[x_i]$. We can use trapezoid rule for the middle of the integrals, with higher-order quadrature rule for the ends.

In the trapezoid rule we had:

$$\int_a^b f(x)\,\mathrm{d}x \approx \frac{[f(x_0) + f(x_{N-1})]\delta x}{2} + \sum_{i=1}^{N-2} f(x_i)\,\delta x. \tag{12}$$

Our mixed method, we instead have:

$$\int_a^b f(x)\,\mathrm{d}x \approx \sum_{i=0}^{k-1} w_i\,[f(x_i) + f(x_{N-i-1})]\;\delta x + \sum_{i=k}^{N-k-1} f(x_i)\,\delta x. \tag{13}$$

For odd values of $k$, this is exact for polynomials of degree $k$ (no need to use even $k$ rules, previous odd $k$ just as accurate). We find the weights in the same way as before, by solving the (end-point) part of the integral exactly for $f(x) = \{1, x, x^2, \dots x^k\}$. For $k = 3$, the error is on the same order as for Simpson's rule; for $k = 5$, it is much better while in fact being simpler and more efficient. Some weights are given in Table 1.

- Accurate and efficient, very simple to implement
- Very useful for array-stored functions $f[x_i]$

Example for same function as above in Ex. 2.2. The error now (with $k = 5$ and $N = 101$) is $\sim 10^{-5}\%$, a huge improvement.

Table 1: Mixed quadrature weights for $k = 1, 3, 5, 7$, where we write weights as $w_i = c_i/d$.

| $k$ | $d_k$ | $c_1$ | $c_2$ | $c_3$ | $c_4$ | $c_5$ | $c_6$ | $c_7$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 1 | | | | | | |
| 3 | 24 | 9 | 28 | 23 | | | | |
| 5 | 1440 | 475 | 1902 | 1104 | 1586 | 1413 | | |
| 7 | 120960 | 36799 | 176648 | 54851 | 177984 | 89437 | 130936 | 119585 |

Example 2.2: Mixed quadrature rile for $k = 5$. Interactive version: godbolt.org/z/48hG8eY7E

```cpp
#include <array>
#include <cmath>
#include <iostream>
double f(double x) { return std::exp(-x) * std::sin(5.0 * x); }

int main() {
  double exact = 0.1910576315007305;
  int n = 101;
  double a = 0.0;
  double b = 5.0;
  double dx = (b - a) / (n - 1);

  std::cout << "\nQuadrature-trapezoid, k=5:\n";
  std::array<double, 5> c{475.0 / 1440, 1902.0 / 1440, 1104.0 / 1440,
                          1586.0 / 1440, 1413.0 / 1440};
  int k = c.size();
  double int4 = 0.0;
  // Quadrature rule for end-regions:
  for (int i = 0; i < k; ++i) {
    double x1 = a + i * dx;
    double x2 = a + (n - 1 - i) * dx;
    int4 += c.at(i) * (f(x1) + f(x2)) * dx;
  }
  // Trapezoid rule for middle part:
  for (int i = k; i < n - k; ++i) {
    double x = a + i * dx;
    int4 += f(x) * dx;
  }
  std::cout << 100.0 * (int4 - exact) / exact << "%\n";
}
```

## 2.5 Other important methods

### 2.5.1 Multi-dimensional integrals

For example

$$\iint f(x,y)\,\mathrm{d}x\mathrm{d}y. \tag{14}$$

In general, everything in the same as above, the generalisation is simple. However, such integrals are often numerically intensive. First, we should check if the function $f$ is separable, e.g., $f(x,y) = X(x)Y(y)$. Of course, this is not always possible, but if it is, we may write

$$\iint X(x)Y(y)\,\mathrm{d}x\mathrm{d}y = \int X(x)\,\mathrm{d}x \int Y(y)\,\mathrm{d}y. \tag{15}$$

Notice that if we used $N$ integration points for each variable, we would have a total of $N^2$ integration points using the naive formula, while we only need $2N$ points if the function is separable. Such scalings can make enormous impact on code run-time, so this is something that should always be checked.

### 2.5.2 Monte-Carlo integration

This will be covered in more depth in coming lectures.

Previously: chose $x_i$ integration/sample points on some uniformly spaced grid. In Monte-Carlo integration, these are chosen randomly. There are drawbacks and benefits to this:

- Fast and simple
- Sometimes less accurate
- Great for multi-dimensional integrals

For multi-dimensional integrals (say, $M$-dimension), we have

$$\int \mathrm{d}x_1 \ldots \int \mathrm{d}x_M f(x_1, \ldots, x_M).$$

If we use methods from above, with $N$ integration points, we would have $N^M$ total integration samples. This blows up very quickly. However, if we draw sets of points $\{x_1,\, x_2 \ldots x_M\}$ randomly (instead of from a defined grid), we often get much better overall coverage of the integration space with much fewer points. i.e., we can get the same accuracy with drastically fewer points.

### 2.5.3 Adaptive methods

In situations where the integrand is poorly behaved (e.g., oscillates quickly, or has unstable points), it can be very difficult to get accurate approximations to the integral. Adaptive methods are typically the best way to proceed in these cases. Adaptive methods continuously sub-divide certain "problem" areas of the integration region until some measure for the error drops below a specified level. This way, we end up using a very large number of points for problem sections of the integration.

A (very simplified) example is shown in Ex. 2.3. This uses the 3-point Simpson's rule, Eq. (3). For each iteration, it calculations the integrals on the sub-domains $[a, m]$ and $[m, b]$ ($m$ is the mid-point), and defines the error as the difference between the sum of these and the integral over the entire $[a, b]$ domain. If the error is too large, it recursively calls itself for each sub-interval, thus dividing each region in half until the error drops below a specified value.

- Very accurate, though can be slow
- Can be combined with any of the methods from above
- Often only option for badly-behaved functions
- Implemented in many libraries, e.g., FORTRAN 'QUADPACK' library
- And in GSL: gnu.org/software/gsl/doc/html/integration
- Program run time can easily explode if too small error is requested

Example 2.3: Simple recursive adaptive integration method, using Simpsons rule. Note: This is a *very* inefficient version; I have kept is as simple as possible to show the general idea. Note: it achieves extremely good accuracy. Interactive version: godbolt.org/z/ojsqKMGox

```cpp
#include <cmath>
#include <iostream>
double f(double x) { return std::exp(-x) * std::sin(5.0 * x); }

// Simpsons rule (for single interval)
template <typename Func> double Simpson(double a, double b, Func f) {
  double m = (a + b) / 2.0;
  return ((b - a) / 6.0) * (f(a) + f(b) + 4.0 * f(m));
}

// Simple recursive adaptive method
template <typename Func>
double adaptive(double a, double b, Func f, double eps) {
  double m = (a + b) / 2; // mid-point
  // Find left, right, and total integrals: error is difference
  double left = Simpson(a, m, f);
  double right = Simpson(m, b, f);
  double total = Simpson(a, b, f);
  double err = std::abs(left + right - total);
  if (err < eps) {
    return total;
  } else {
    // Error too large: sub-divide (recursive)
    return adaptive(a, m, f, eps / 2.0) + adaptive(m, b, f, eps / 2.0);
  }
}

int main() {
  double exact = 0.1910576315007305;
  double a = 0.0;
  double b = 5.0;
  double error_target = 1.0e-6;
  double integral = adaptive(a, b, f, error_target);
  std::cout << "Error: " << 100.0 * (integral - exact) / exact << "%\n";
}
```

### 2.5.4 Non-uniform integration grid

We can define a non-uniform distribution for $x$, by introducing a new variable, $u$. For example, if want $x$ to be distributed according to a logarithmic grid, we set

$$u = \log x\,, \qquad \frac{\mathrm{d}x}{\mathrm{d}u} = x.$$

Then, with uniformly distributed $u$, we will have a logarithmically distributed $x$. This choice is very common, and is useful any time $x$ may span more than 1 order of magnitude. For other situations, we may chose different $u$ functions – but $u = \log x$ is very useful in a wide range of cases, including atomic physics.

Note that we then have:

$$\int_a^b f(x)\,\mathrm{d}x = \int_{u_a}^{u_b} f(x(u))\,\frac{\mathrm{d}x}{\mathrm{d}u}\,\mathrm{d}u, \tag{16}$$

and everything proceeds as per above (any of the above methods work with a non-uniform grid).

### 2.5.5 Importance Sampling

If we have a complex function, we may want to concentrate most of our integration samples in the important region of the domain. For example, if there is a sub-domain where we know $f(x)$ to be large, it will be most efficient to use a larger number of sample points in that region. One way to achieve this is with *inverse transform sampling.* This is a generalised way to chose a non-uniform grid, and is very often used in conjunction with Monte-Carlo integration.

Say we want the distribution of sample points to follow a probability distribution $p(x)$ (i.e., more points where $p(x)$ is large, fewer where it is small). Define cumulative distribution function

$$C(x) = \int_{-\infty}^{x} p(x')\,\mathrm{d}x', \tag{17}$$

and its inverse $C^{-1}(u)$. We will make the change of variables

$$x = C^{-1}(u). \tag{18}$$

Since $p$ is a normalised probability distribution, $C(x)$ has values over the range [0,1]. Therefore, $u$ assumes values on domain [0,1]. For a uniformly distributed set of $u$ values over [0,1], $x(u) = C^{-1}(u)$ will be distributed according to $p(x)$.

Since

$$\frac{\mathrm{d}}{\mathrm{d}x} f^{-1}|_a = \frac{1}{\frac{\mathrm{d}f}{\mathrm{d}x} \circ f^{-1}(a)},$$

we have

$$\frac{\mathrm{d}x}{\mathrm{d}u} = \frac{\mathrm{d}C^{-1}(u)}{\mathrm{d}u} = \frac{1}{C'(x)} = \frac{1}{p(x)}. \tag{19}$$

$$\int_a^b f(x)\,\mathrm{d}x = \int_{u_a}^{u_b} f(x(u))\,\frac{\mathrm{d}x}{\mathrm{d}u}\,\mathrm{d}u, \tag{20}$$

By choosing $p(x)$ to be large in the important regions for $f(x)$ [e.g., where $f$ is large, or where the integrand oscillates etc.], this leads to increased accuracy.

This is particularly useful in Bayesian analysis, where it is common to evaluate integrals of the form

$$p(D|m) = \int p(D|x)p(x|m)\,\mathrm{d}x. \tag{21}$$

This equation is called the marginalised likelihood; $p(D|m)$ means the probability of observing data $D$, given model $m$. $x$ is some parameter of the model, which needs to be integrated to find the total probability. $p(D|x)$ means probability of observing data $D$ given a specific value of $x$ – this is typically a complicated function that is calculated numerically and depends on the data. $p(x|m)$ is the probability distribution for parameter $x$, given model $m$ – it is called the Bayesian prior. By choosing $x$ to be distributed according to $p(x|m)$, we concentrate samples where the prior is large, and simplify the integration:

$$p(D|m) = \int p(D|x)p(x|m)\,\mathrm{d}x = \int p(D|x(u))p(x(u)|m)\,\frac{\mathrm{d}x}{\mathrm{d}u}\,\mathrm{d}u \tag{22}$$

$$= \int p(D|x(u))\,\mathrm{d}u. \tag{23}$$

I.e., priors can be taken into account using importance sampling – this simplifies the required integral, and also ensures we are concentrating sample points in the most important regions.