Chapter_9

Henry Bourne

2022-12-12

Numerical Integration

Often in statistics we need to evaluate integrals, for example when finding the posterior expectation or evaluating risk/loss functions. However, often integrals cannot be expressed using elementary functions and so we must instead use approximations, which involve lots of computation. Often we can make these approximations arbitrarily accurate at the expense of increasingly large compute. We will cover two types of algorithms: deterministic numerical integration algorithms and Monte Carlo algorithms.

Quadrature (Deterministic Numerical Integration Method)

Quadrature rules are integral approximations that use a finite number of function evaluations, we will look mainly at the case where we are doing this over a finite interval. Here we will look at quadrature rules that involve interpolating polynomials. In practice we can use the **integrate()** function for one-dimensional functions and the **cubature** package for multi-variate integration. We will start by describing how we can approximate the function of the data from the datapoints and then how we can use this approximation to find the integral over our desired interval.

Polynomial interpolation

Let's consider the case where we want to approximate some continuous function, f, on interval, $[a, b]$, (ie. $f \in C^0([a, b])$) using a polynomial function p. A motivation for using this example is that polynomials can be approximated exactly, theoretically the Weierstrass approximation theorem also gives us motivation. It tells us that that there exists a sequence of functions that converges uniformly on our function f on $[a, b]$, so for some tolerance we define we know it;s possible to find a polynomial that can be used to approximate f (for some arbitrary f in $C^0([a, b])$). However, it does not tell us how to do this and how to do this in a computationally efficient way. Polynomial approximations are referred to as **quadrature rules**.

Lagrange polynomials

One way we could create an approximation is to approximate f using an **interpolating polynomial**, the polynomial of lowest degree that passes through all our points. The interpolating polynomial is unique and has degree at most k-1, we can express it as a **Lagrange polynomial**:

$$p_{k-1}(x) := \sum_{i=1}^{k} l_i(x) f(x_i) \tag{1}$$

where $l_i(x)$ (the **lagrange basis polynomials**) are:

$$l_i(x) = \prod_{j=1, j\neq i}^{k} \frac{x - x_j}{x_i - x_j} \tag{2}$$

So for the k points we are given we can construct the Lagrange polynomial that if we plot will be the polynomial of lowest degree that goes through all k points.

Polynomial interpolation error

By choosing a k large enough we can successfully approximate any polynomial, however, desirably we would like to also approximate functions that aren't in fact polynomial. Let's first introduce the **Interpolation Error Theorem**: Let $f \in C^k[a, b]$, ie. $f : [a, b] \to \mathbb{R}$ has k continuous derivatives and let p_{k-1} be the polynomial interpolating f at k points $x_1, ..., x_k$. Then for any $x \in [a, b]$ $\exists \xi \in (a, b)$ such that:

$$f(x) - p_{k-1}(x) = \frac{1}{k!} f^{(k)}(\xi) \prod_{i=1}^{k} (x - x_i) \tag{3}$$

From this we can drive a theorem that says there is a sequence of interpolation points that guarantee uniform convergence of a sequence of interpolating polynomials on f. However, for any fixed sequence of sets of interpolation points there exists a continuous function f for which the sequence of interpolating polynomials diverges. So although there exists a sequence of interpolation points that guarantee convergence this sequence will vary from function to function.

So how should we select our interpolating points? We can use the **Chebyshev points**: for a given k, we choose the points

$$\cos(\frac{2i - 1}{2k}\pi) \tag{4}$$

and the absolute value of the product term is then bounded above by 2^{1-k}. However, just using Chebyshevs points is not a complete solution as there exist functions for which the interpolating polynomial obtained using Chebyshevs points diverges.

Other Polynomial interpolation schemes

An alternative to the above would be to split the domain up and approximate a polynomial for each sub-interval in the domain, this is called **composite polynomial interpolation**. Note that this could result in a piecewise polynomial approximation that is not necessarily continuous. The error associated with this approximation can be obtained using the Interpolation error theorum, we can find that for a large number of sub intervals the product term can be made arbitrarily small in absolute value.

We could also fit a polynomial using derivatives of f aswell as f, which is known as **Hermite interpolation**. There is also **spline interpolation** where we incorporate derivatives in piecewise polynomial interpolation by matching the derivatives at the boundaries of the sub-intervals, which means that our polynomial approximation we get is continuous.

We could also use techniques such as neural networks to create function approximations, however, it may not be easy to integrate these more complicated approximation functions.

Polynomial Integration

Now we have our polynomial approximation of the function we would like to find the approximating integral:

$$I(f) := \int_a^b f(x)dx \tag{5}$$

where our $f \in C^0([a,b])$.

Changing the limits of integration

The first thing we will discuss is how we can change the limits of our approximate integral. ie. how we can find a function g such that:

$$\int_a^b f(x)dx = \int_c^d g(y)dy \tag{6}$$

We can define a function g that fulfills the above as follows:

$$g(y) := \frac{b-a}{d-c} g(a + \frac{b-a}{d-c}(y-c)) \tag{7}$$

It is also possible to do a similar thing for semi-infinite (one limit is an infinity) and infinite integrals.

Integrating the interpolating polynomial approximation

Consider integrating a Lagrange polynomial (which we introduced earlier), p_{k-1}, over $[a,b]$. We have that:

$$I(p_{k-1}) = \int_a^b p_{k-1}(x)dx \tag{8}$$

$$= \int_a^b \sum_{i=1}^{k} l_i(x) f(x_i)dx \tag{9}$$

$$= \sum_{i=1}^{k} f(x_i) \int_a^b l_i(x)dx \tag{10}$$

$$= \sum_{i=1}^{k} w_i f(x_i) \tag{11}$$

where $w_i := \int_a^b l_i(x)dx$. The approximation of $I(f)$ is $I(p_{k-1})$, we therefore can calculate the approximate integral. Note, however, that the l_i functions can be complicated (for larger values of k).

Other methods

Alternatively we could use the **Newton-Cotes rules** which are a family of numerical integration methods where we apply each rule again to some polynomial that approximates the function we are trying to integrate. The rules include the **rectangular rule**, **midpoint rule**, **trapezoidal rule** and the **Simpson rule**.

We could also use **composite rules**, which is when we are integrating a composite polynomial interpolation approximation. The approximate integral here is simply the sum of the approximate integrals associated with each sub-interval, where we could use Newton-Cotes rules to compute the approximations over each sub-interval for example.

In practice

So far we have seen some simple rules that tell us how we can approximate a function using a polynomial by interpolating and then how to find an approximation for the integral of this approximating polynomial. These rules are popular in practice, however, we can tweak them to enhance them when using them in practice.

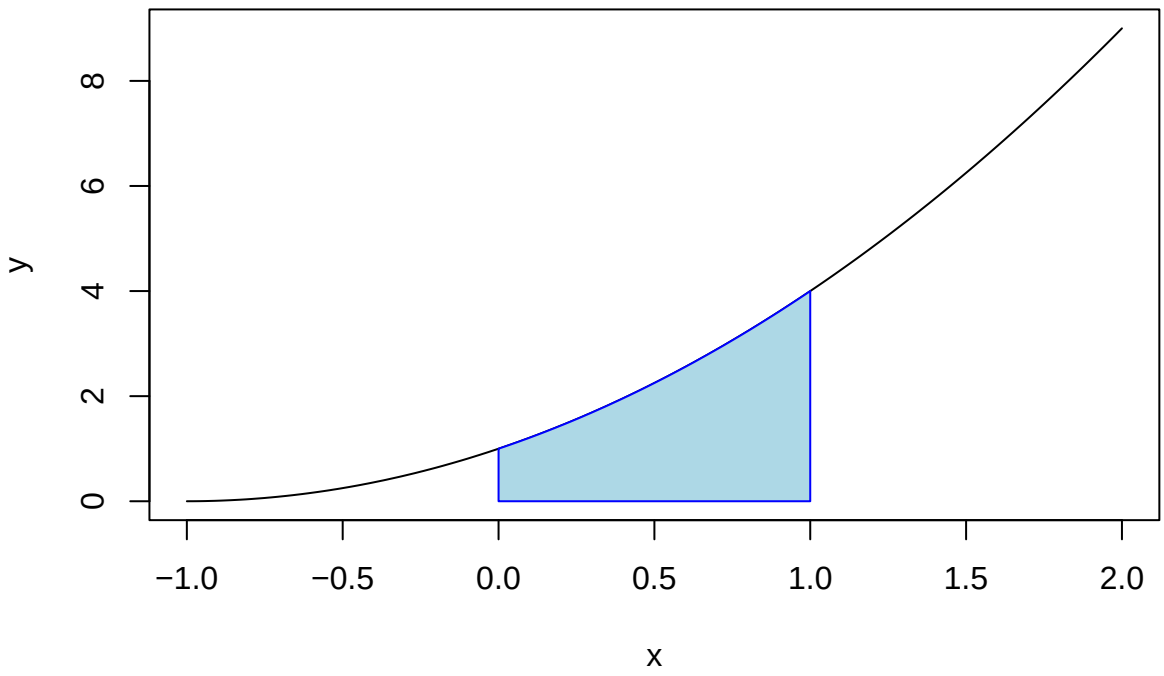
Ideally we would like to be able to compute an approximation of the integral to a given precision, however, to do this we need to compute estimates or bounds on the error. It is also advisable that we decrease computational cost, one way of doing this would be spending more compute on subintervals where the integral is estimated badly and less compute elsewhere (ie. we should focus the "attention" of our compute).

Additional features we would like would be incorporating change of variable formulas for dealing with semi infinite and infinite intervals aswell as techniques for dealing with singularities. We could even do things such as change the way we calculate weights (the w_i in equation (11)) to make them more easily integrable.

The R function **integrate** incorporates some of these desirable features. It uses adaptive quadrature which means it automatically subdivides the integration integral into sub-intervals to evaluate to meet a given accuracy. Like we just mentioned we wanted, this allows it to focus on regions of the integration interval that are more variable and need more compute. It can also deal with evaluating semi infinite and infinite integrals. It also uses the Wynn Epsilon algorithm to speed up convergence of the sequence of estimates produced by the adaptive quadrature algorithm. The quadrature algorithm it uses is the Gauss–Kronrod quadrature algorithm which uses Gaussian quadrature combined with Kronrod quadrature. Gaussian quadrature uses a set of points and weights that are optimized for approximating polynomials up to a certain degree and Kronrod quadrature uses a larger set of points and weights that are optimized for approximating integrals of smooth functions. By combining these methods we get the benefits of both and can achieve higher performance.

We can use the integrate function as follows; first lets define the function we want to integrate:

```
f <- function(x) {
  x^2 + 2 * x + 1
}
x <- seq(from = -1, to = 2, by = 0.01)
y <- f(x)
plot(x, y, type = "l")
y_filtered <- subset(y, x >= 0 & x <= 1)
x_filtered <- subset(x, x >= 0 & x <= 1)
polygon(c(0,x_filtered,1), c(0,y_filtered,0), col = "lightblue", border = "blue")
```



Now lets integrate it and print the result:

```
integrate(f,0,1)
```

```
## 2.333333 with absolute error < 2.6e-14
```

Sometimes in R we would like to integrate multi-dimensional functions which is where we run into an issue because *integrate* can only integrate one-dimensional functions. This is where the **cubature** package comes in. From the package description: the cubature package is "R wrappers around the cubature C library of Steven G. Johnson for adaptive multivariate integration over hypercubes and the Cuba C library of Thomas Hahn for deterministic and Monte Carlo integration". This package allows for one to use a plethora of techniques to carry out multi-dimensional integration. The methods include cubature methods (such as the **hcubature** method) which are essential quadrature methods but for when we are working in multiple-dimensions. It also includes Monte Carlo integration methods (such as the **vegas** method) which work by randomly sampling some function values, using this to compute an initial approximation of the integral and the assessing the accuracy of the approximation, it then adds additional points from the regions of the sub-domain where the approximation has the largest error using the Monte Carlo method.