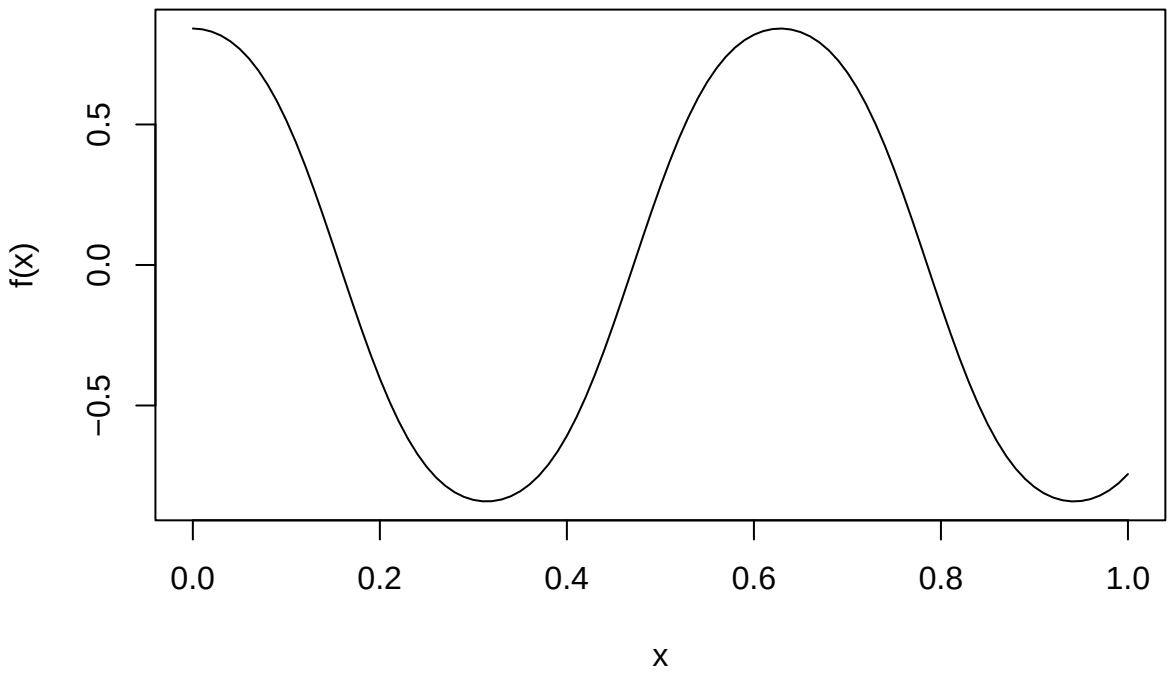
Portfolio 9

Sam Bowyer

Numerical Integration

In statistical analysis we are often required to evaluate integrals. For relatively simple integrals, such as $\int_0^1 \sin(\cos(10x))dx$, we can use R's `integrate` function:

```
f <- function(x) sin(cos(10*x))
curve(f)
```




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

```
## -0.0465934 with absolute error < 1.2e-06
```

However, many of the integrals we may want to evaluate may not have a closed-form expression, which can render exact computation extremely difficult if not impossible. To work around this we often then must use various approximation techniques, many of which fall into two groups: quadrature and Monte-Carlo methods.

Quadrature methods calculate an approximation of a definite integral by first splitting the desired integral into various smaller areas. For each of these smaller areas we can then approximate the integrand's behaviour

by some simpler function which will be much easier to integrate, polynomials in particular are very common choices here. For instance, if we have k points $\{(x_i, f(x_i)\}_{i=1}^{k}$of a polynomial f we can construct an interpolating *Lagrange polynomial* as

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

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

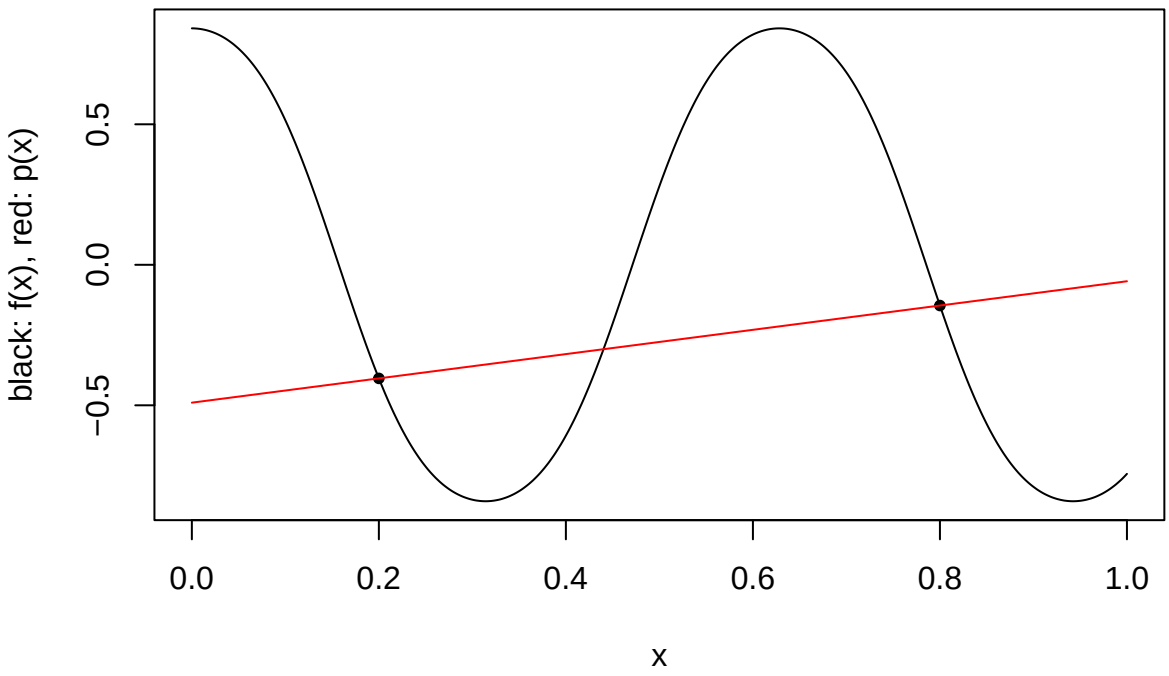
$$l_i(x) = \prod_{j \in [k] \setminus \{i\}} \frac{x - x_j}{x_i - x_j} \qquad i \in [k].$$

From this we can write the following code (for which I am heavily indebted to the SC1 course website) from which we can see how each section of any sufficiently well-behaved integral can be approximated by a polynomial:

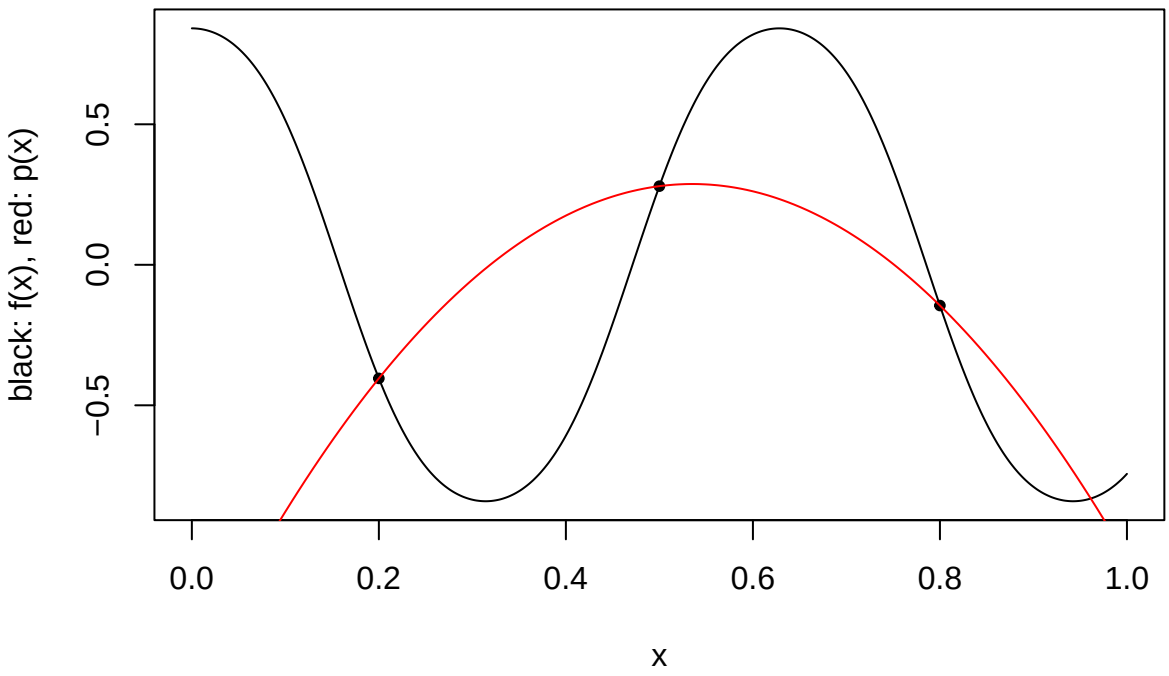
```
getLagrangePolynomial <- function(f, xs) {
  polynomial <- function(x) {
    total = 0
    for (i in 1:length(xs)) {
      zs = xs[setdiff(1:length(xs),i)]
      basisPolynomial <- prod((x-zs)/(xs[i]-zs))
      total = total + f(xs[i])*basisPolynomial
    }
    return(total)
  }
  return(polynomial)
}

plotPolyApproximation <- function(f, xs, start, end) {
  p = getLagrangePolynomial(f, xs)
  xs_full = seq(start, end, length.out=500)
  plot(xs_full, f(xs_full), type='l', xlab="x", ylab="black: f(x), red: p(x)")
  points(xs, f(xs), pch=20)
  lines(xs_full, vapply(xs_full, p, 0), col="red")
}

plotPolyApproximation(f, c(0.2, 0.8), 0, 1)
```

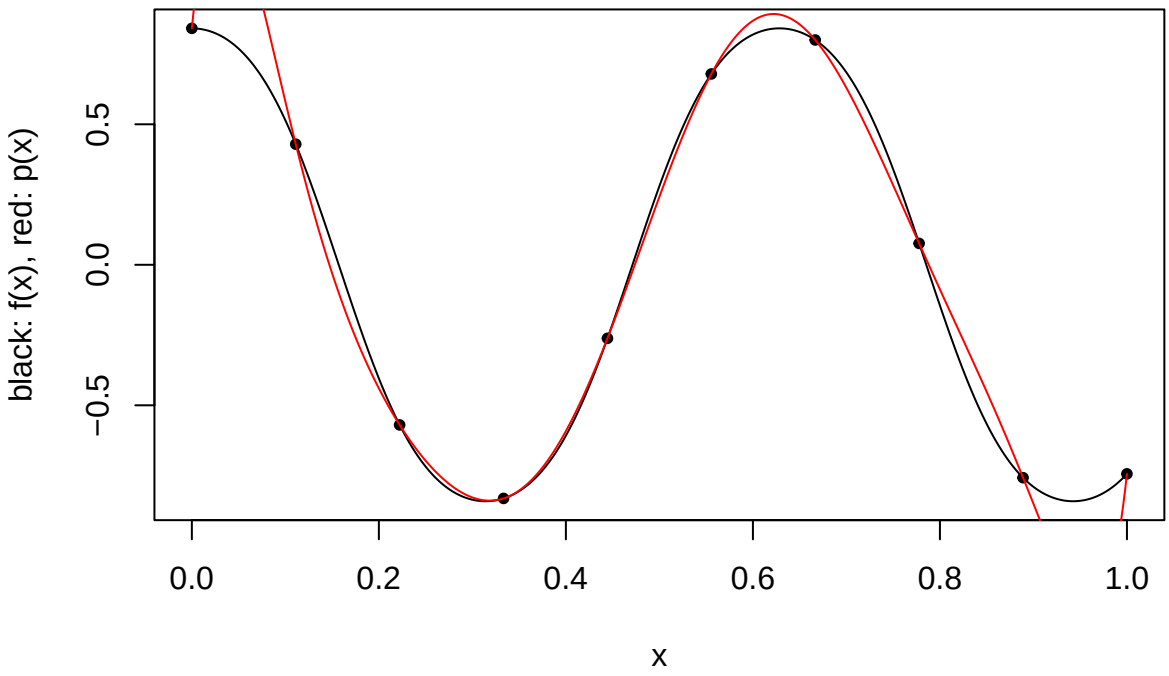



```
plotPolyApproximation(f, c(0.2,0.5, 0.8), 0, 1)
```

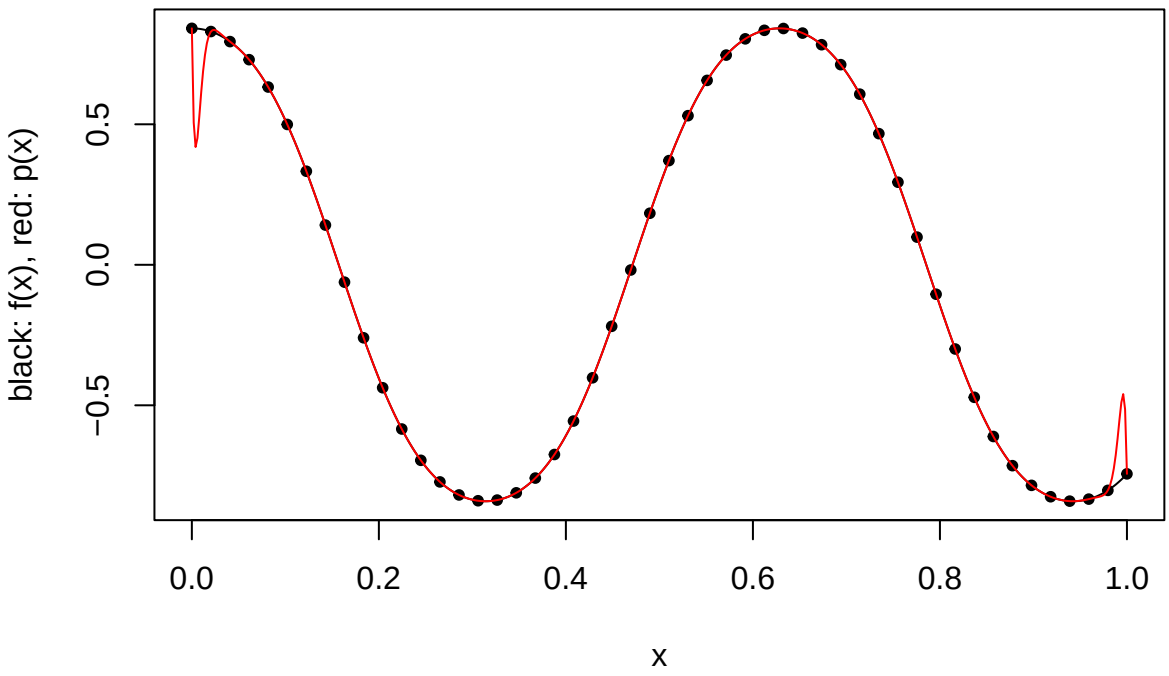


Clearly for this non-polynomial function $f(x) = \sin(\cos(10x))$ we have not arrived at a suitable approximation with 2 or 3 points, however, we can improve this by increasing the number of points:

```
plotPolyApproximation(f, seq(0, 1, length.out=10), 0, 1)
```



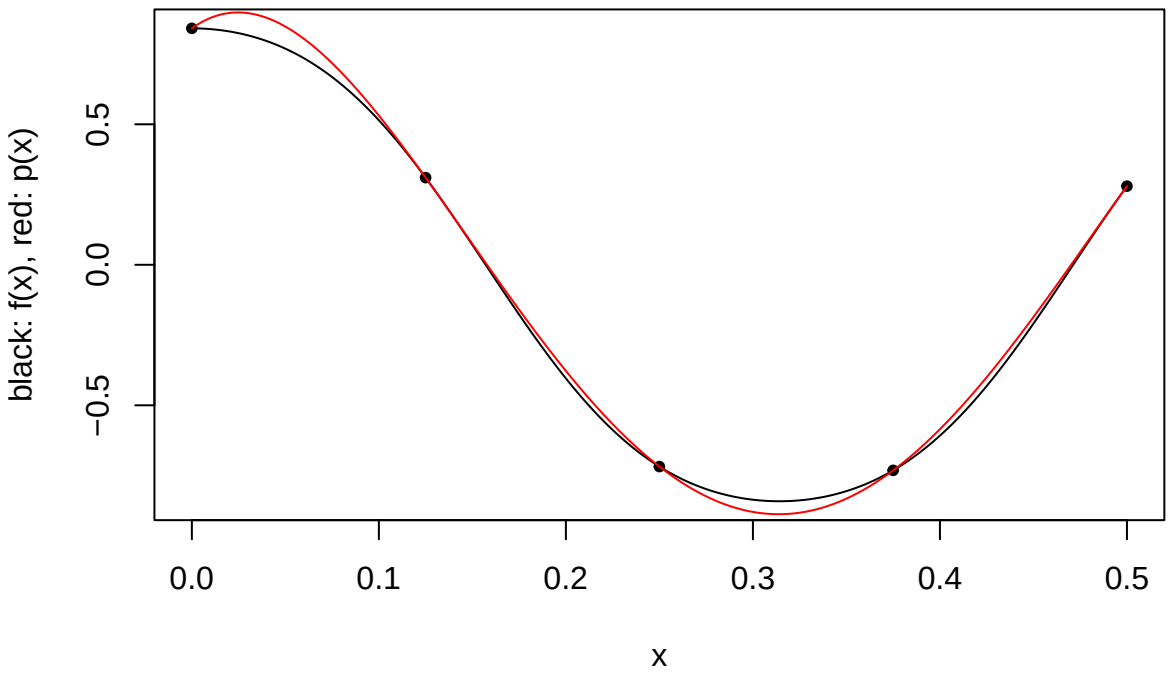

```
plotPolyApproximation(f, seq(0, 1, length.out=50), 0, 1)
```



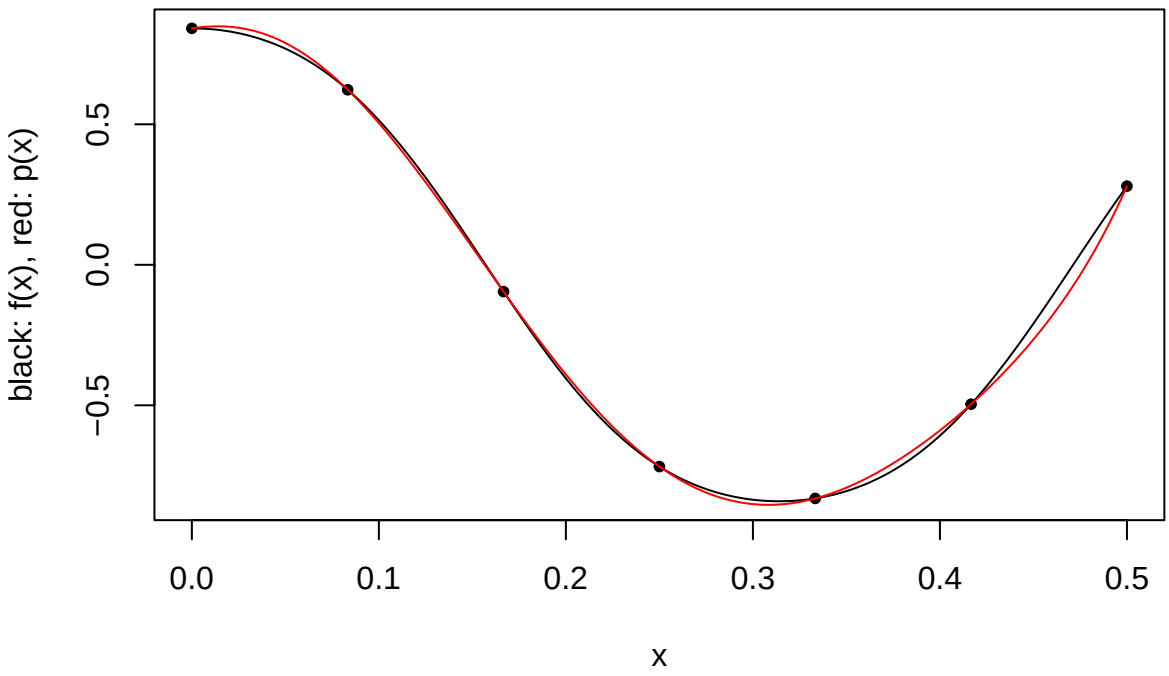
Note the odd behaviour near the endpoints of our function, something which can often crop up in these approximations. (It is also worth noting that if f were a polynomial of degree k we would interpolate f exactly using this method if we were given k points).

The other way we can improve these approximations without having to use so many data points is by splitting up our domain into smaller sections—this is really what's at the heart of quadrature. For example, see how we can obtain a fairly good approximation of f over the interval $[0, 0.5]$ with only 5 data points, with a particularly good approximation at 7 points.

```
plotPolyApproximation(f, seq(0, 0.5, length.out=5), 0, 0.5)
```




```
plotPolyApproximation(f, seq(0, 0.5, length.out=7), 0, 0.5)
```



Whilst quadrature methods are fairly computationally cheap, they can suffer greatly for high-dimensional integrals, in which case we might instead opt for a Monte-Carlo approach. One large family of algorithms of this type is that of Markov chain Monte Carlo (MCMC) methods, which, on a very basic level, works by finding a Markov chain with an equilibrium distribution that can be used to approximate the integral at hand. Then we can obtain samples from this equilibrium distribution by sampling a number of (ideally, sufficiently separated) points from some initial distribution and examining how they move throughout the state space of the Markov chain over a large number of time steps.