Lecture 12: Interpolation and Numerical Integration

Cheng Lu

Overview

- Interpolation
  - Linear Interpolation
  - Spline Interpolation
- Numerical Integration
  - Trapezoidal rule
  - Simpson's rule

Interpolation

Interpolation: Constructing new data points using the data we know.

- Given data points $(x_i, y_i), i = 1, \ldots, n$, where $x_i < x_{i+1}, i = 1, \ldots n-1$ are well ordered.
- Given $x_{out} : x_1 < x_{out} < x_n$, want to find the corresponding y_{out}.

Two approaches:

- Linear interpolation: Approximate data points by piecewise linear functions
- Spline interpolation: Approximate data points by piecewise polynomials

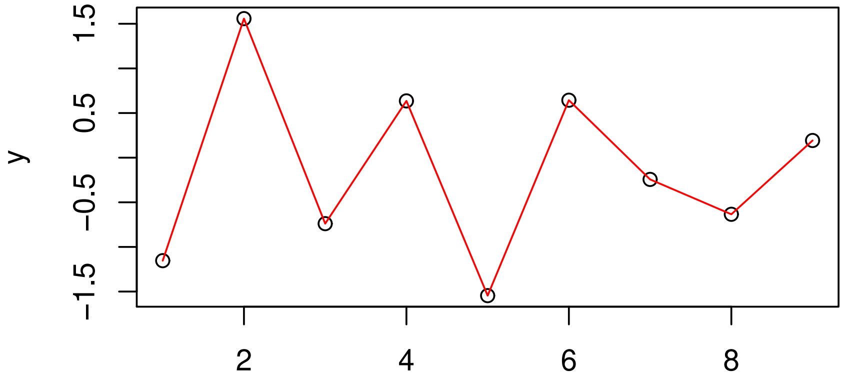
Linear Interpolation

Linear interpolation: Approximate data points by piecewise linear functions

Example

```
> n <- 9
> x <- 1:n # x = 1, 2, 3, ..., 9
> y <- rnorm(n) # generate y from standard normal
> plot(x, y)
> lines(x, y, col = "red")
```

Linear Interpolation



Linear Interpolation

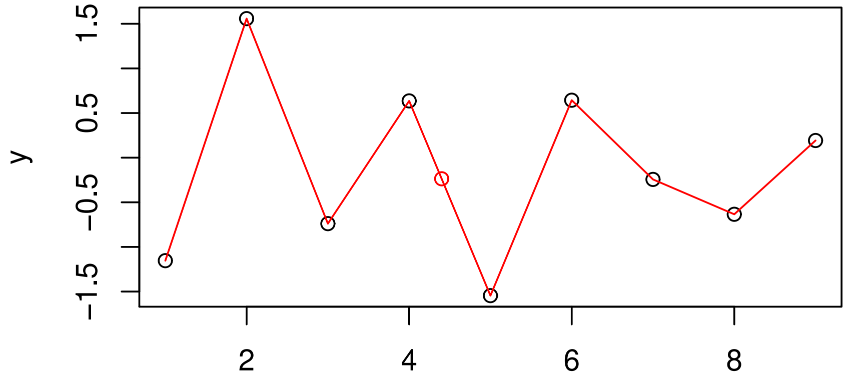
Suppose we want to find y_{out} such that $x_{out} = 4.4$. Then we need to find the 2 closest x, which are 4 and 5, and approximate y_{out} by:

$$y_{out} = y_4 + \frac{y_5 - y_4}{x_5 - x_4}(x_{out} - x_4)$$

Example

```
> xout <- 4.4
> which(x < xout) # find indices for x < x_out
[1] 1 2 3 4
> max(which(x < xout)) # max index
[1] 4
> id <- max(which(x < xout)) # id <- max index
> yout <- y[id] + (y[id+1] - y[id])/(x[id+1] - x[id])*(xout - x[id])
> yout
[1] -0.2366558
> points(xout, yout, col = "red")
```

Linear Interpolation



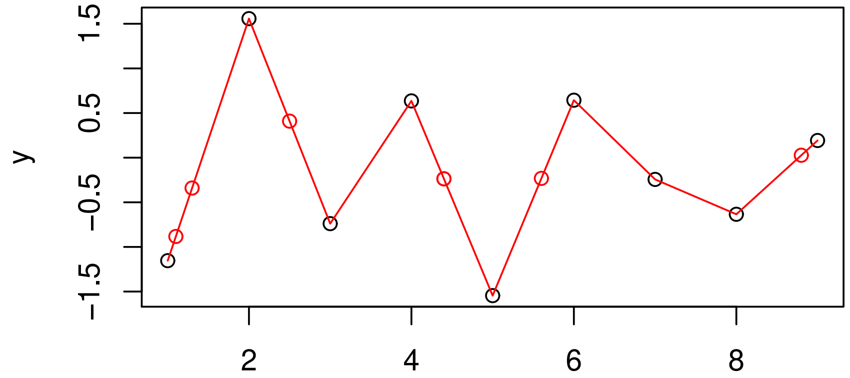
Linear Interpolation

Suppose x_{out} is a vector of points we need to interpolate, and we need to find the vector y_{out}, the first way is putting the code into a loop:

Example

```
> xout <- c(1.1, 1.3, 2.5, 4.4, 5.6, 8.8)
> # Loop
> yout <- NULL
> for(i in 1:length(xout)){
+   id <- max(which(x < xout[i]))
+   yout[i] <- y[id] +
+     (y[id+1] - y[id])/(x[id+1] - x[id])*(xout[i] - x[id])
+ }
> yout
[1] -0.88282846 -0.34046603  0.40930369 -0.23665581 -0.23205302  0.02701905
> points(xout, yout, col = "red")
```

Linear Interpolation



Linear Interpolation

We can also use vectorized computation:

Example

```
> # Vectorized
> func1 <- function(xout){
+   id <- max(which(x < xout))
+   yout <- y[id] + (y[id+1] - y[id])/(x[id+1] - x[id])*(xout - x[id])
+ }
> yout <- sapply(xout, func1)
> yout
[1] -0.88282846 -0.34046603  0.40930369 -0.23665581 -0.23205302  0.02701905
```

Linear Interpolation

- Built-in functions: **approx()** or **approxfun()**
- **approx()** returns a list of x_{out} and y_{out}
- **approxfun()** returns a function for evaluating x_{out}

Example

```
> # Built in
> yout <- approx(x,y,xout = xout)$y
> yout
[1] -0.88282846 -0.34046603  0.40930369 -0.23665581 -0.23205302  0.02701905
> yout.fun <- approxfun(x, y)# function
> yout.fun(xout)
[1] -0.88282846 -0.34046603  0.40930369 -0.23665581 -0.23205302  0.02701905
```

Linear Interpolation

In linear interpolation, we approximate data by piecewise linear functions

$$\begin{aligned}
P_1(x) &= y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1), x_1 \leq x \leq x_2 \\
P_2(x) &= y_2 + \frac{y_3 - y_2}{x_3 - x_2}(x - x_2), x_2 \leq x \leq x_3 \\
&\dots \\
P_{n-1}(x) &= y_{n-1} + \frac{y_n - y_{n-1}}{x_n - x_{n-1}}(x - x_{n-1}), x_{n-1} \leq x \leq x_n
\end{aligned}$$

- There are "kinks" on the curve at points $x_2, \dots, x_{n-1}$,
- Need a more smooth curve, approximate with higher order polynomials.

Spline Interpolation

Spline interpolation: Approximate data points by piecewise polynomials

- For example: the 3rd order piecewise polynomials (cubic spline):

$$P_1(x) = a_1 + b_1 x + c_1 x^2 + d_1 x^3, x_1 \le x \le x_2$$
$$P_2(x) = a_2 + b_2 x + c_2 x^2 + d_2 x^3, x_2 \le x \le x_3$$
$$\dots$$
$$P_{n-1}(x) = a_{n-1} + b_{n-1} x + c_{n-1} x^2 + d_{n-1} x^3, x_{n-1} \le x \le x_n$$

- To make the curve smooth, we need values and derivatives agree at each data points:

$$P_1(x_1) = y_1$$
$$P_1(x_2) = P_2(x_2) = y_2, P_1'(x_2) = P_2'(x_2), P_1''(x_2) = P_2''(x_2)$$
$$\dots$$
$$P_{n-2}(x_{n-1}) = P_{n-1}(x_{n-1}) = y_{n-1}, P_{n-2}'(x_{n-1}) = P_{n-1}'(x_{n-1}), P_{n-2}''(x_{n-1}) = P_{n-1}''(x_{n-1})$$
$$P_{n-1}(x_n) = y_n.$$

Spline Interpolation

- Total $4(n-1)$ parameters and $4(n-1)-2$ equations, to solve the parameters a_i, b_i, c_i, d_i completely, we need to impose boundary conditions, for example:

$$P_1''(x_1) = 0, P_{n-1}''(x_n) = 0$$

- The cubic spline with the boundary condition above is called *natural cubic spline*.
- Now total $4(n-1)$ parameters with $4(n-1)$ equations, thus can be uniquely solved.
- One can simplify the model by replacing $n-1$ polynomials to a single polynomial.

Spline Interpolation

Consider using one polynomial P to replace $P_1, \ldots, P_{n-1}$:

$$P(x) = a + b(x - x_1) + c(x - x_1)^2 + \sum_{k=1}^{n-1} d_k (x - x_k)_+^3, x_1 \le x \le x_n$$

where $(x - x_i)_+ = \max\{x - x_i, 0\}$. Also the values and derivatives agree at each data points:

$$\lim_{x \to x_i^-} P(x) = \lim_{x \to x_i^+} P(x) = a + b(x_i - x_1) + c(x_i - x_1)^2 + \sum_{k=1}^{i-1} d_k (x_i - x_k)^3$$

$$\lim_{x \to x_i^-} P'(x) = \lim_{x \to x_i^+} P'(x) = b + 2c(x_i - x_1) + \sum_{k=1}^{i-1} 3d_k (x_i - x_k)^2$$

$$\lim_{x \to x_i^-} P''(x) = \lim_{x \to x_i^+} P''(x) = 2c + \sum_{k=1}^{i-1} 6d_k (x_i - x_k)$$

Spline interpolation

We can solve the parameters with the equations:

$$P(x_1) = a + b(x_1 - x_1) + c(x_1 - x_1)^2 + \sum_{k=1}^{n-1} d_k (x_1 - x_k)_+^3 = y_1$$

$$P(x_2) = a + b(x_2 - x_1) + c(x_2 - x_1)^2 + \sum_{k=1}^{n-1} d_k (x_2 - x_k)_+^3 = y_2$$

$$\dots$$

$$P(x_n) = a + b(x_n - x_1) + c(x_n - x_1)^2 + \sum_{k=1}^{n-1} d_k (x_n - x_k)_+^3 = y_n$$

For the natural cubic spline we have the boundary condition:

$$P''(x_1) = 2c = 0, P''(x_n) = 2c + \sum_{k=1}^{n-1} 6 d_k (x_n - x_k) = 0$$

Spline Interpolation

Now we have total $n+2$ parameters and $n+2$ equations. First we write the equations into matrix form:

$$A\theta = z$$

where

$$A = \begin{bmatrix} 1 & x_1 - x_1 & (x_1-x_1)^2 & (x_1-x_1)_+^3 & \dots & (x_1-x_{n-1})_+^3 \\ 1 & x_2 - x_1 & (x_2-x_1)^2 & (x_2-x_1)_+^3 & \dots & (x_2-x_{n-1})_+^3 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n - x_1 & (x_n-x_1)^2 & (x_n-x_1)_+^3 & \dots & (x_n-x_{n-1})_+^3 \\ 0 & 0 & 2 & 0 & \dots & 0 \\ 0 & 0 & 2 & 6(x_n-x_1) & \dots & 6(x_n-x_{n-1}) \end{bmatrix}; \theta = \begin{bmatrix} a \\ b \\ c \\ d_1 \\ \vdots \\ d_{n-1} \end{bmatrix}; z = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \\ 0 \\ 0 \end{bmatrix}$$

We only need to solve the linear system for θ.

Spline Interpolation

Example

```
> A <- cbind(1, x-x[1], (x-x[1])^2, pmax(outer(x, x[-n], "-"), 0)^3)
> # outer(x,x,"-") generates matrix M with M_ij = (x_i - x_j)
> A <- rbind(A, 0, 0) # initialize last 2 row
> A[c(n+1,n+2), 3] <- 2
> A[n+2, 4:(n+2)] <- 6*(x[n] - x[-n]) # last row
> z <- c(y, 0, 0)
> theta <- solve(A,z) # solve linear system
> a <- theta[1]
> b <- theta[2]
> c <- theta[3]
> d <- theta[-c(1,2,3)]
> yout <- as.vector(a + b*(xout-x[1]) + c*(xout-x[1])^2 +
+   pmax(outer(xout, x[-n], "-"), 0)^3%*%d)
> yout
[1] -0.71442587  0.12391686  0.37394158 -0.17408304 -0.34863316 -0.01175013
```

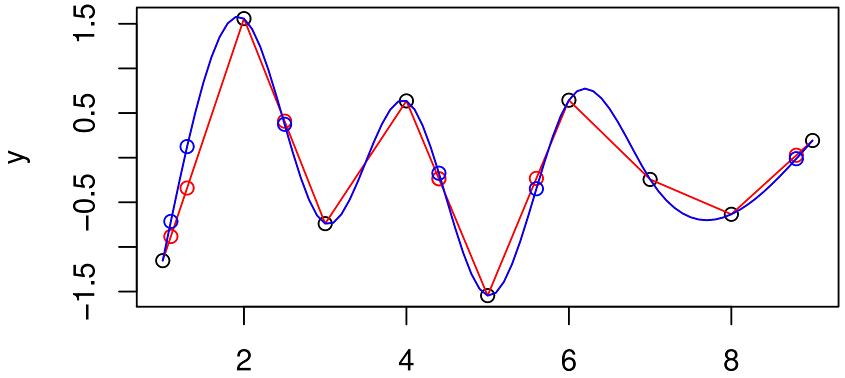
Interpolation

- Built-in functions: **spline()** and **splinefun()**.
- Usage is similar to **approx()** and **approxfun()**.
- Use natural cubic spline by letting **method = "natural"**.

Example

```
> yout.s <- spline(x,y, xout = xout, method = "natural")
> yout.s$y
[1] -0.71442587  0.12391686  0.37394158 -0.17408304 -0.34863316 -0.01175013
> yout.s.fun <- splinefun(x, y, method = "natural")
> yout.s.fun(xout)
[1] -0.71442587  0.12391686  0.37394158 -0.17408304 -0.34863316 -0.01175013
> lines(seq(1,9,by=0.1),yout.s.fun(seq(1,9,by=0.1)), col = "blue")
> points(xout, yout, col = "blue")
```

Interpolation



Numerical Integration

Numerical integration (quadrature): Calculating numeric value of integrals:

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

- f: integrand function, a continuous function;
- a: lower bound, a real number;
- b: upper bound, a real number.

First we evaluate f in finite many points on a partition $a = x_0 < x_1 < \ldots < x_n = b$

$$f(x_0), f(x_1), \ldots, f(x_n)$$

Then approximate f by piecewise polynomials and integrate them. Two approaches:

- Trapezoidal rule: Approximate f by piecewise linear functions
- Simpson's rule: Approximate f by piecewise quadratic functions

Trapezoidal Rule

Trapezoidal rule: Approximate function f by piecewise linear functions

$$f(x) \approx f(x_{i-1}) + \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}(x - x_{i-1}), \forall x \in [x_{i-1}, x_i]$$

Then

$$\begin{aligned}\int_a^b f(x)dx = \sum_{i=1}^n \int_{x_{i-1}}^{x_i} f(x)dx &\approx \sum_{i=1}^n \int_{x_{i-1}}^{x_i} f(x_{i-1}) + \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}(x - x_{i-1})dx \\ &= \sum_{i=1}^n \frac{1}{2}(f(x_{i-1}) + f(x_i))(x_i - x_{i-1})\end{aligned}$$

Trapezoidal Rule

If the partition has uniform grid: i.e. $\forall 1 \leq i \leq n$

$$x_i - x_{i-1} = \frac{b-a}{n}$$

Then

$$\int_a^b f(x)dx \approx \sum_{i=1}^{n} \frac{1}{2}(f(x_{i-1}) + f(x_i))(x_i - x_{i-1}) = \frac{b-a}{n}\left(\frac{1}{2}f(x_0) + \sum_{i=1}^{n-1} f(x_i) + \frac{1}{2}f(x_n)\right)$$

And we can calculate numeric integral using this formula.

Trapezoidal Rule

Suppose we are going to calculate $E[(Z^3-8)_+]$, where Z follows standard normal distribution, then we have

$$E[(Z^3-8)_+] = \int_{-\infty}^{+\infty}(x^3-8)_+\phi(x)dx = \int_{2}^{+\infty}(x^3-8)\phi(x)dx \approx \int_{2}^{10}(x^3-8)\phi(x)dx^1$$

where ϕ is the density function of standard normal distribution.

Example

```
> f <- function(x)(x^3-8)*dnorm(x) # define integrand function
> a <- 2 # lower bound of x
> b <- 10 # upper bound of x
> n <- 1000
> x <- seq(a, b, length.out = n+1)
> (b - a)/n*(sum(f(x)) - 0.5*f(a) - 0.5*f(b))
[1] 0.1419413
```

[1]Since $P(Z \leq 10) = \text{pnorm}(10) \approx 1$

Simpson's Rule

Simpson's rule: Approximate function f by piecewise quadratic functions:[2]

$$f(x) \approx \frac{(x-x_i)(x-x_{i+1})}{(x_{i-1}-x_i)(x_{i-1}-x_{i+1})} f(x_{i-1}) + \frac{(x-x_{i-1})(x-x_{i+1})}{(x_i-x_{i-1})(x_i-x_{i+1})} f(x_i) + \frac{(x-x_{i-1})(x-x_i)}{(x_{i+1}-x_{i-1})(x_{i+1}-x_i)} f(x_{i+1}), \forall x \in [x_{i-1}, x_{i+1}]$$

If the partition has uniform grid, we have $h = x_i - x_{i-1}$:

$$\int_{x_{i-1}}^{x_{i+1}} \frac{(x-x_i)(x-x_{i+1})}{(x_{i-1}-x_i)(x_{i-1}-x_{i+1})} dx = \int_0^{x_{i+1}-x_{i-1}} \frac{(x+x_{i-1}-x_i)(x+x_{i-1}-x_{i+1})}{(x_{i-1}-x_i)(x_{i-1}-x_{i+1})} dx$$

$$= \frac{1}{2h^2} \int_0^{2h} (x-h)(x-2h) dx = \frac{h}{3}$$

[2]Here we use Lagrange polynomials, one can verify it by substituting x_{i-1}, x_i and x_{i+1} for x.

Simpson's Rule

Similarly we have

$$\int_{x_{i-1}}^{x_{i+1}} \frac{(x-x_{i-1})(x-x_{i+1})}{(x_i-x_{i-1})(x_i-x_{i+1})}dx = \frac{4h}{3}; \int_{x_{i-1}}^{x_{i+1}} \frac{(x-x_{i-1})(x-x_i)}{(x_{i+1}-x_{i-1})(x_{i+1}-x_i)}dx = \frac{h}{3}$$

Then

$$\int_{x_{i-1}}^{x_{i+1}} f(x)dx \approx \frac{h}{3}(f(x_{i-1}) + 4f(x_i) + f(x_{i+1}))$$

Consider n as even number of interval, i.e. $n \bmod 2 = 0$, we have

$$\int_a^b f(x)dx = \sum_{i=1}^{n/2} \int_{x_{2i-2}}^{x_{2i}} f(x)dx \approx \sum_{i=1}^{n/2} \frac{h}{3}(f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i}))$$

$$= \frac{h}{3}(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \cdots + 4f(x_{n-1}) + f(x_n))$$

Simpson's Rule

Consider the same integral $\int_2^{10}(x^3-8)\phi(x)dx$

Example

```
> f <- function(x)(x^3-8)*dnorm(x) # define integrand function
> a <- 2 # lower bound for x
> b <- 10 # upper bound for x
> n <- 1000
> x <- seq(a, b, length.out = n+1)
> as.vector((b - a)/(3*n)*c(1,rep(c(4,2),n/2)[-n],1)%*%f(x))
[1] 0.1419447
```

- If we have $a=-\infty$ or $b=+\infty$, we need to make $b-a$ large enough and grid $h=\frac{b-a}{n}$ small enough for convergence.
- If we have large $b-a$, then we need a much larger n to make the size $h=\frac{b-a}{n}$ small enough to improve accuracy.

Numerical Integration

We can also use built-in function **integrate()** (when the integral is absolutely convergent[3]).
Consider the same integral: $\int_2^{\infty}(x^3-8)\phi(x)dx \approx \int_2^{10}(x^3-8)\phi(x)dx$

Example

```
> f <- function(x)(x^3-8)*dnorm(x)# define integrand function
> integrate(f, 2, 10)
0.1419447 with absolute error < 9.4e-05
> s <- integrate(f, 2, 10)
> s$value
[1] 0.1419447
> integrate(f, 2, Inf)$value
[1] 0.1419447
```

[3]Function **integrate()** may fails when the integral is divergent or conditional convergent.