Integration

Dom Owens

10/12/2019

Integrals often occur in statistics, for example as posterior expectations of a Bayesian model. Many are analytically intractable, so we approximate these with numerical methods. These can normally be approximated to arbitrary accuracy with sufficient computing resources.

These can be delineated into

-Deterministic methods

-Monte Carlo methods

Quadrature

Quadrature Rules approximate integrals with a finite number of function evaluations.

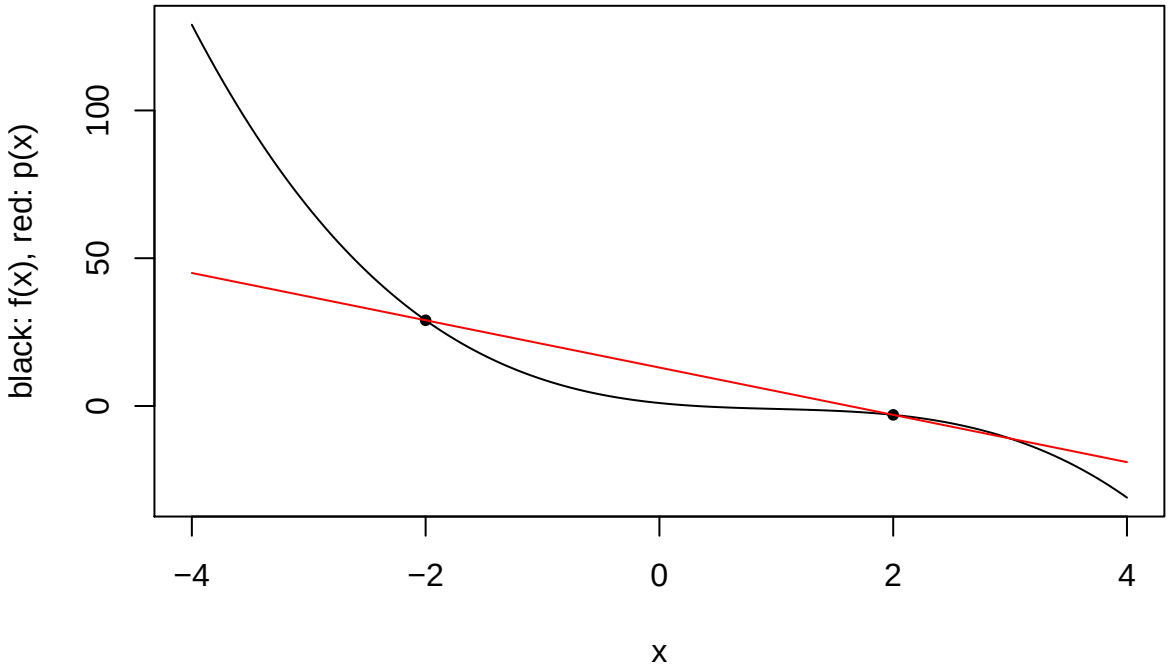
Polynomial Interpolation

We want to approximate a continuous function f on $[a, b]$ with a polynomial function p.

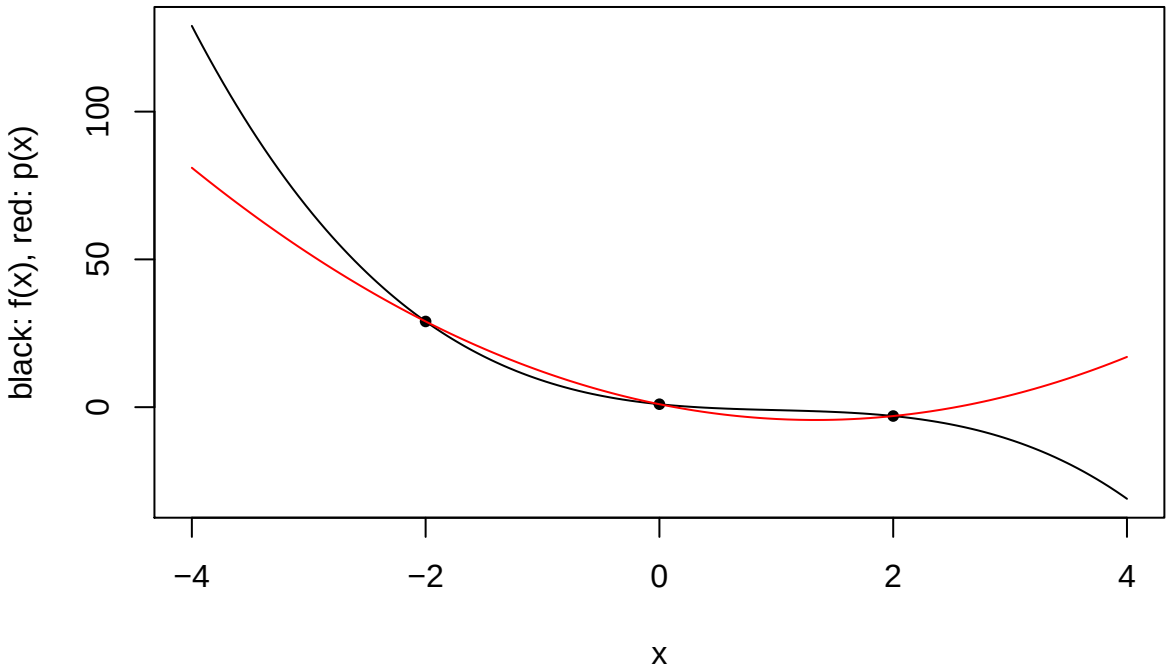
We can use k points $\{(x_i, f(x_i)\}_1^k$ to construct an interpolating polynomial with **Lagrange basis functions** $\lessdot_i$

```
construct.interpolating.polynomial <- function(f, xs) {
  k <- length(xs)
  fxs <- f(xs)
  p <- function(x) {
    value <- 0
    for (i in 1:k) {
      fi <- fxs[i]
      zs <- xs[setdiff(1:k,i)]
      li <- prod((x-zs)/(xs[i]-zs))
      value <- value + fi*li
    }
    return(value)
  }
  return(p)
}

plot.polynomial.approximation <- function(f, xs, a, b) {
  p <- construct.interpolating.polynomial(f, xs)
  vs <- seq(a, b, length.out=500)
  plot(vs, f(vs), type='l', xlab="x", ylab="black: f(x), red: p(x)")
  points(xs, f(xs), pch=20)
  lines(vs, vapply(vs, p, 0), col="red")
}

a <- -4
b <- 4
```

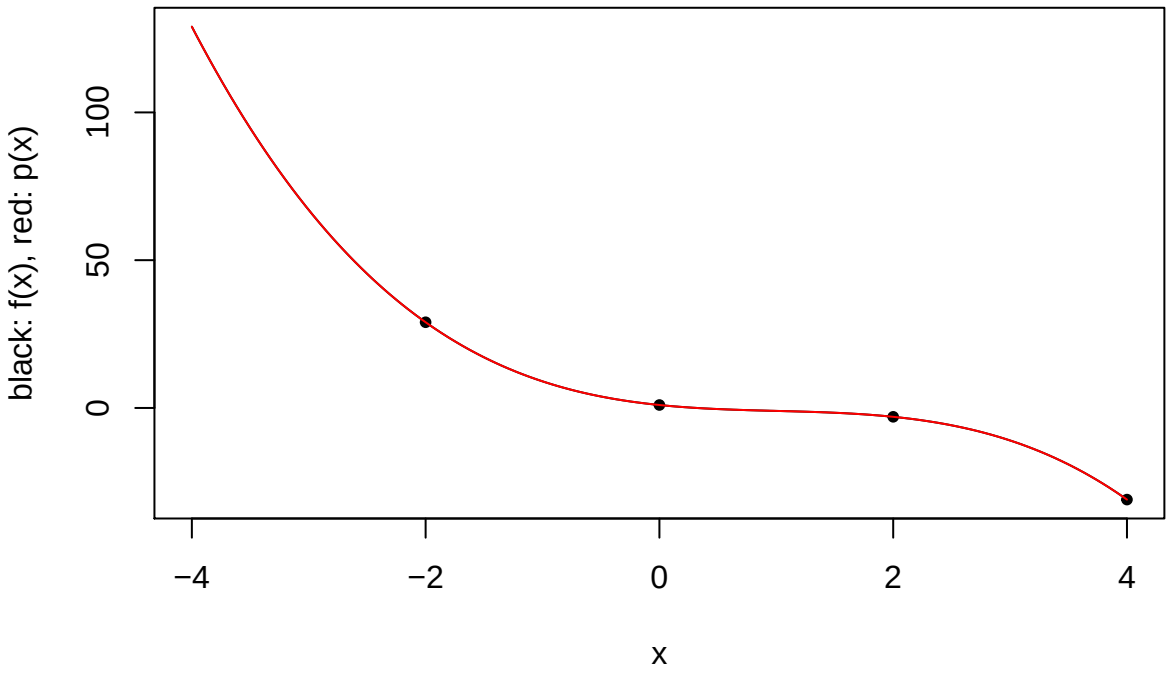
```r
f <- function(x) {
  return(-x^3 + 3*x^2 - 4*x + 1)
}

plot.polynomial.approximation(f, c(-2, 2), a, b) #inear approximation
```




```r
plot.polynomial.approximation(f, c(-2, 0, 2), a, b) #expand basis set to quadratic
```




```r
plot.polynomial.approximation(f, c(-2, 0, 2, 4), a, b)
```



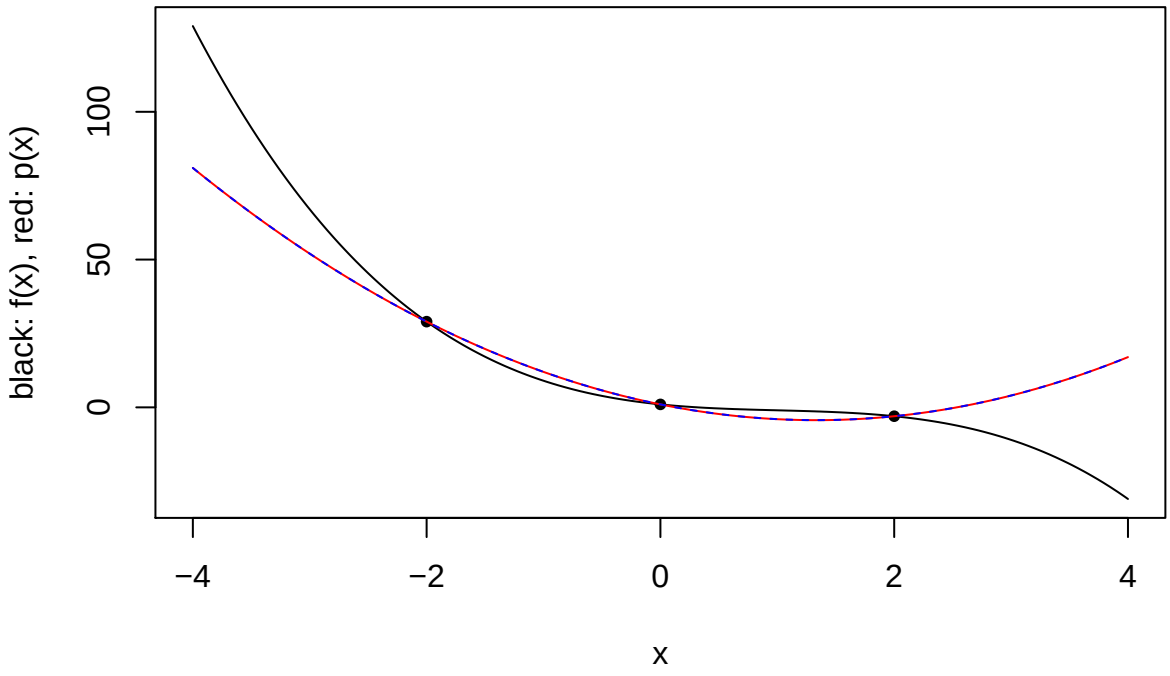
Sums of monomials

$$\sum_{i=1}^{k} a_i x^{i-1}$$

are a possible alternative, though solving the linear system can be unstable.

```
construct.vandermonde.matrix <- function(xs) {
  k <- length(xs)
  A <- matrix(0, k, k)
  for (i in 1:k) {
    A[i,] <- xs[i]^(0:(k-1))
  }
  return(A)
}

compute.monomial.coefficients <- function(f, xs) {
  fxs <- f(xs)
  A <- construct.vandermonde.matrix(xs)
  coefficients <- solve(A, fxs)
  return(coefficients)
}

construct.polynomial <- function(coefficients) {
  k <- length(coefficients)
  p <- function(x) {
    return(sum(coefficients*x^(0:(k-1))))
  }
  return(p)
}

xs <- c(-2,0,2)
p <- construct.polynomial(compute.monomial.coefficients(f, xs))
plot.polynomial.approximation(f, xs, a, b)
```

```
vs <- seq(-4, 4, length.out=100)
lines(vs, vapply(vs, p, 0), col="blue", lty="dashed")
```

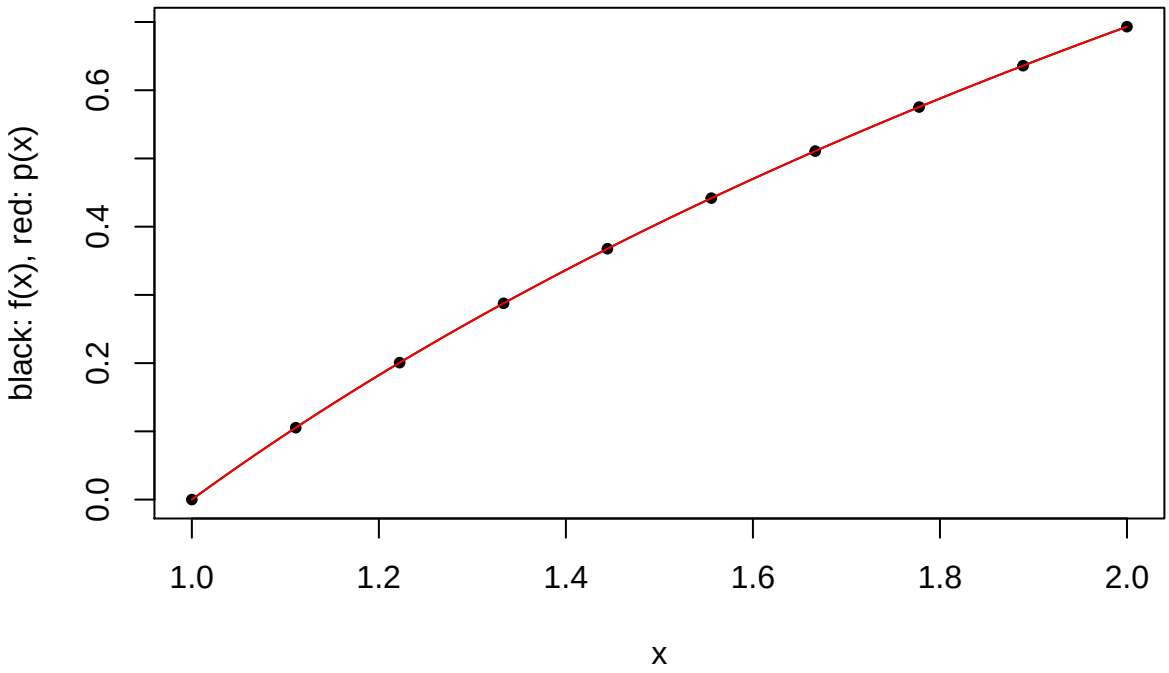


Error

With sufficiently large k we can approximate any polynomial. What if the problem is not a polynomial?

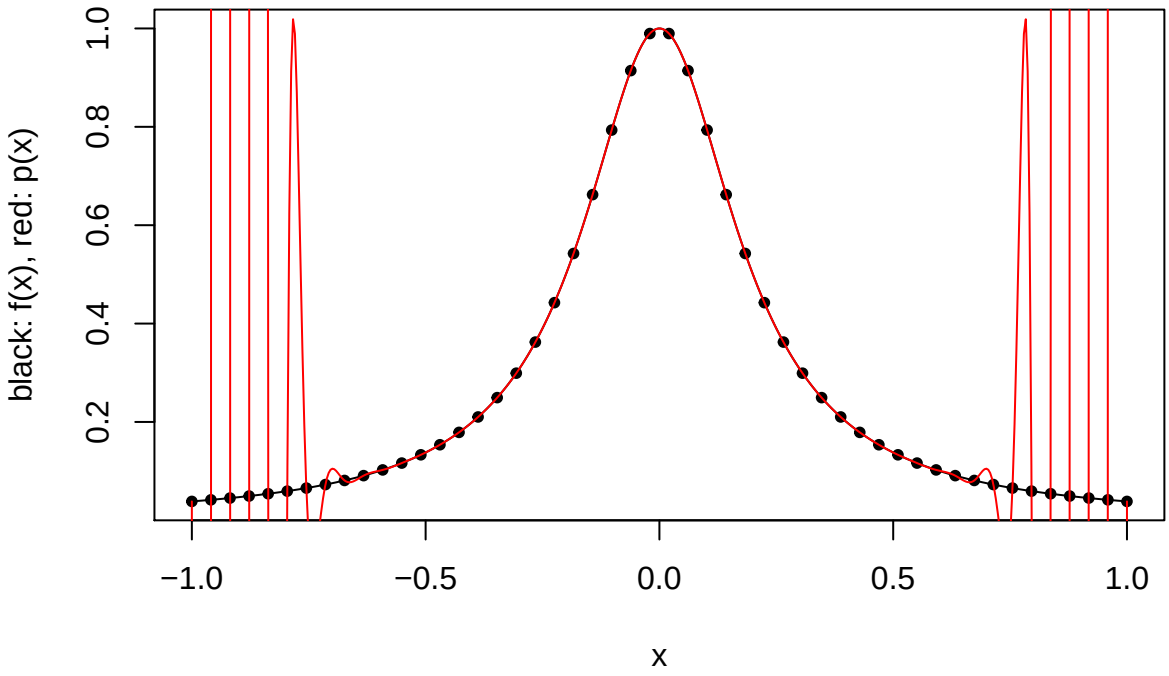
By the *interpolation error theorem*, there is a sequence of sets of interpolating points giving uniform convergence to f. There is no universal set of points independent of f guaranteeing convergence.

```
construct.uniform.point.set <- function(a, b, k) {
  if (k==1) return(a)
  return(seq(a, b, length.out=k))
}

a <- 1
b <- 2
plot.polynomial.approximation(log, construct.uniform.point.set(a, b, 10), a, b)
```



The **Runge function** shows uniformly-spaced points are not always suitable; the approximation is poor at tails.

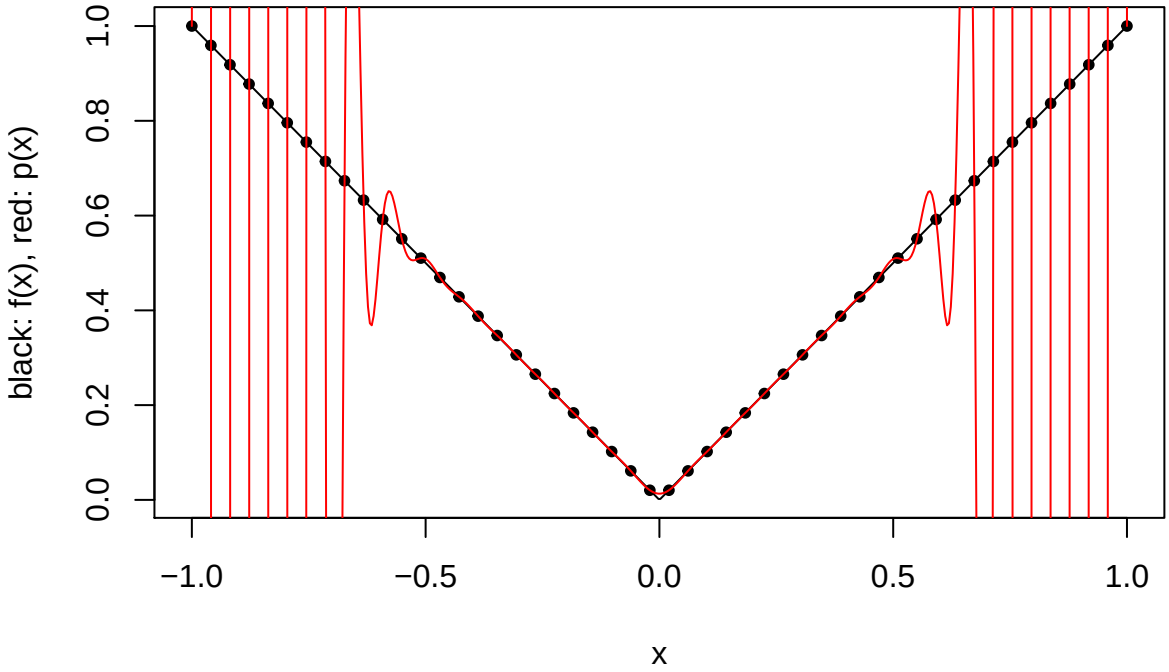
```
a <- -1
b <- 1
f <- function(x) return(1/(1+25*x^2))
plot.polynomial.approximation(f, construct.uniform.point.set(a,b,50), a, b)
```



The `abs` function does the same.

```
a <- -1
b <- 1
```

```
plot.polynomial.approximation(abs, construct.uniform.point.set(a,b,50), a, b)
```



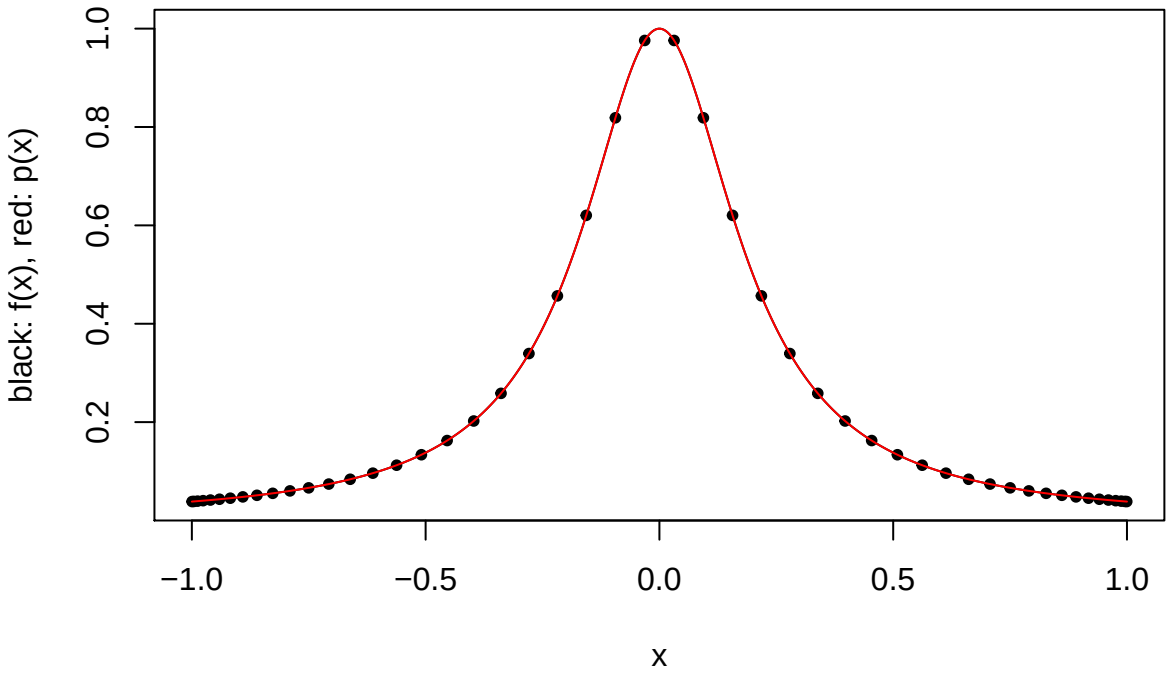
Chebyshev points are chosen to to minimise the maximum absolute value of the product in the interpolation error formula. These are, for

$$i \in \{1, \dots, k\}$$

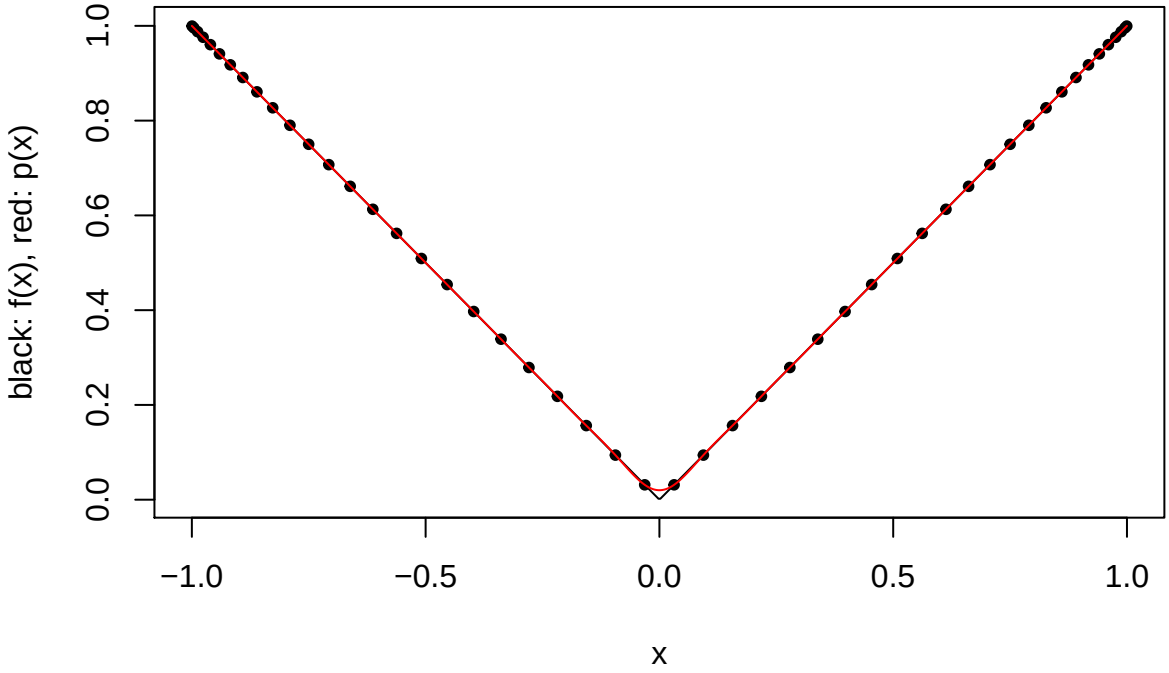
$$\cos\left(\frac{2i-1}{2k}\pi\right)$$

These do not minimise overall error.

```
construct.chebyshev.point.set <- function(k) { #define function constructing chebyshev points
  return(cos((2*(1:k)-1)/2/k*pi))
}

plot.polynomial.approximation(f, construct.chebyshev.point.set(50), a, b) #apply chebyshev points as ba
```

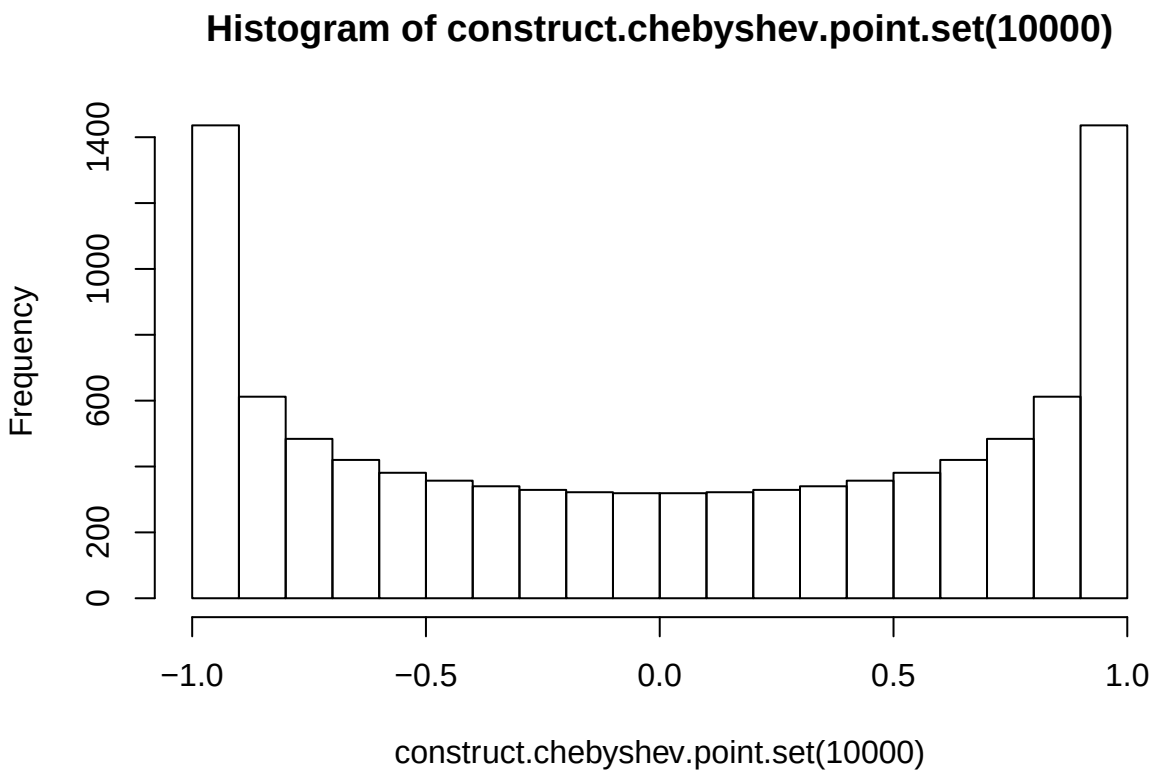



```
plot.polynomial.approximation(abs, construct.chebyshev.point.set(50), a, b) #same for abs function
```



These are clustered at -1 and 1.

```
hist(construct.chebyshev.point.set(10000))
```



For some functions the polynomials diverge on the Chebyshev set, but these are rare and complicated.

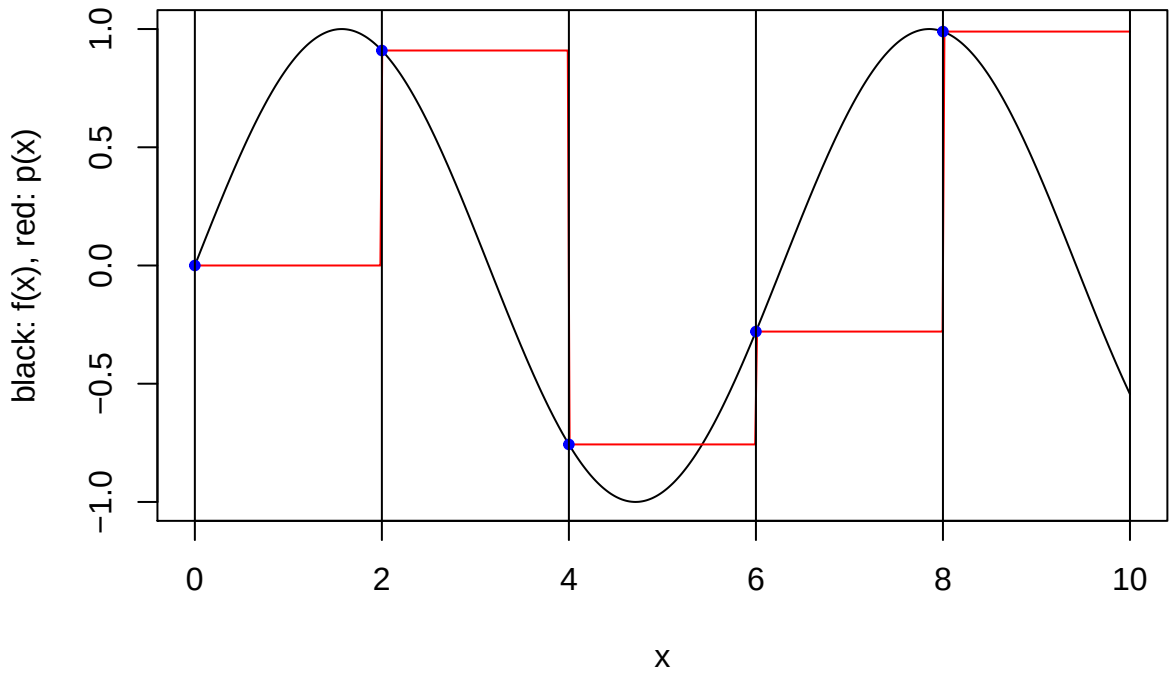
Composite Polynomials

We might instead split the interval $[a, b]$ into sub-intervals, giving a non-continuous approximation of f.

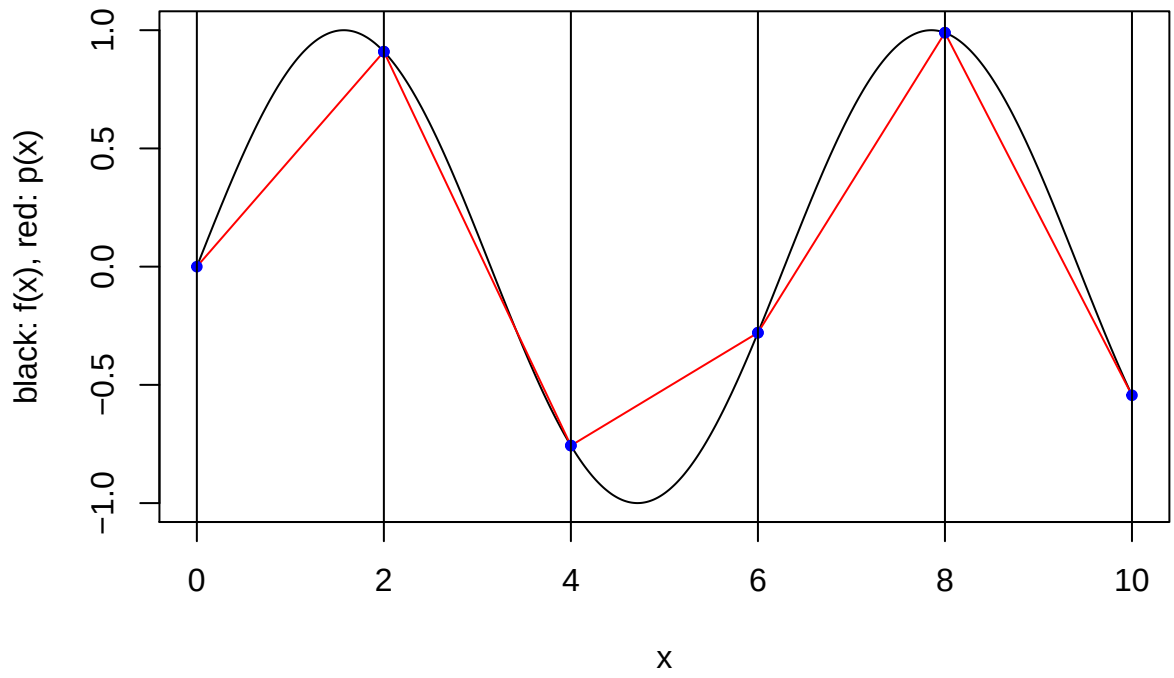
k points are chosen in each sub-interval, evenly spaced. **Closed** schemes use endpoints of the interval; **Open** schemes do not. We split the interval into equally-sized sub-intervals for simplicity.

```
# get the endpoints of the subintervals
get.subinterval.points <- function(a, b, nintervals) {
  return(seq(a, b, length.out=nintervals+1))
}

# returns which subinterval a point x is in
get.subinterval <- function(x, a, b, nintervals) {
  h <- (b-a)/nintervals
  return(min(max(1,ceiling((x-a)/h)),nintervals))
}

# get the k interpolation points in the interval
# this depends on the whether the scheme is open or closed
get.within.subinterval.points <- function(a, b, k, closed) {
  if (closed) {
    return(seq(a, b, length.out=k))
  } else {
    h <- (b-a)/(k+1)
    return(seq(a+h,b-h,h))
  }
}
```

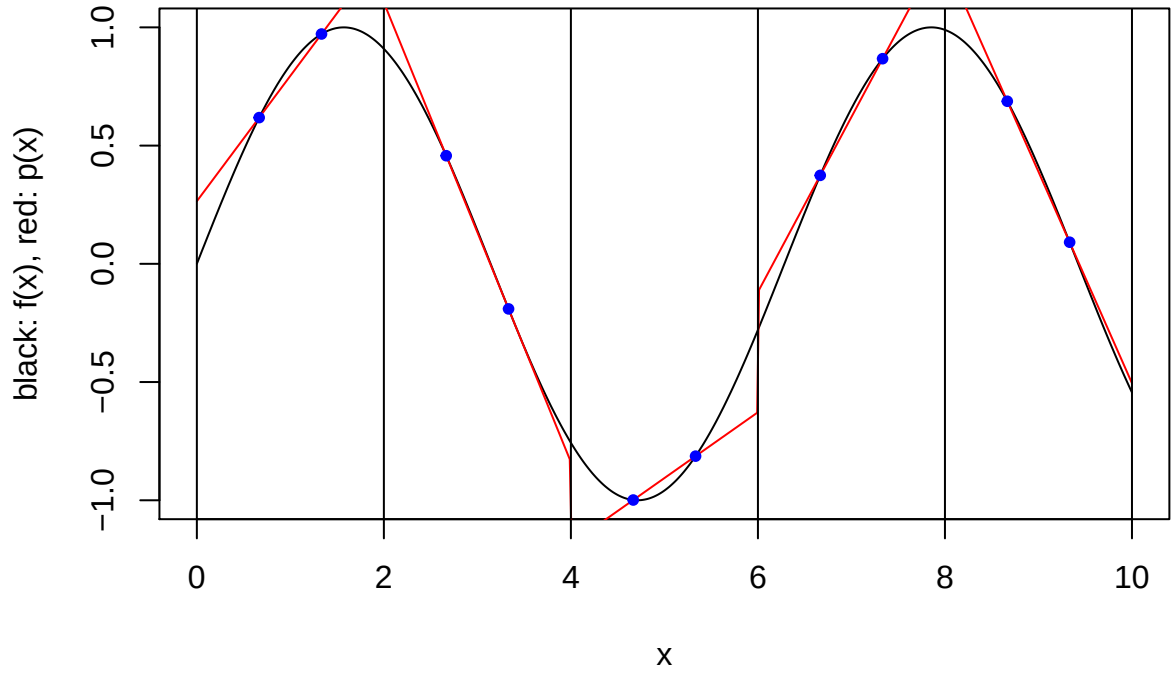
```r
construct.piecewise.polynomial.approximation <- function(f, a, b, nintervals, k, closed) {
  ps <- vector("list", nintervals)
  subinterval.points <- get.subinterval.points(a, b, nintervals)
  for (i in 1:nintervals) {
    left <- subinterval.points[i]
    right <- subinterval.points[i+1]
    points <- get.within.subinterval.points(left, right, k, closed)
    p <- construct.interpolating.polynomial(f, points)
    ps[[i]] <- p
  }
  p <- function(x) {
    return(ps[[get.subinterval(x, a, b, nintervals)]](x))
  }
  return(p)
}

plot.piecewise.polynomial.approximation <- function(f, a, b, nintervals, k, closed) {
  p <- construct.piecewise.polynomial.approximation(f, a, b, nintervals, k, closed)
  vs <- seq(a, b, length.out=500)
  plot(vs, f(vs), type='l', xlab="x", ylab="black: f(x), red: p(x)")
  lines(vs, vapply(vs, p, 0), col="red")
  subinterval.points <- get.subinterval.points(a, b, nintervals)
  for (i in 1:nintervals) {
    left <- subinterval.points[i]
    right <- subinterval.points[i+1]
    pts <- get.within.subinterval.points(left, right, k, closed)
    points(pts, f(pts), pch=20, col="blue")
  }
  abline(v = subinterval.points)
}
plot.piecewise.polynomial.approximation(sin, 0, 10, 5, 1, TRUE) #1 point/interval, closed
```



```
plot.piecewise.polynomial.approximation(sin, 0, 10, 5, 2, TRUE) #2 points/interval, closed
```




```
plot.piecewise.polynomial.approximation(sin, 0, 10, 5, 2, FALSE) #2 points/interval, open
```



Polynomial Integration

We instead want the integral

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

Domains can be changed using a linear function.

```
change.domain <- function(f, a, b, c, d) { #map [a,b] to [c,d]
  g <- function(y) {
    return((b-a)/(d-c)*f(a + (b-a)/(d-c)*(y-c)))
  }
  return(g)
}
```

This also works for infinite intervals.

Using Interpolating Polynomials

We use $I(f) = \hat{I}(p_{k-1})$, which only requires evaluating integrals of polynomials. Polynomials have analytical forms for their integrals. Each scheme for finding the interpolating polynomial corresponds to a **Newton-Cotes rule**.

Multiple Integrals can be rewritten iteratively over each dimension of the range, permitting recursive algorithms to perform quadrature on each specific dimension.

Monte Carlo Methods

For higher dimensional problems, we more often use **Monte Carlo Methods**. We want to approximate the quantity

$$\pi(f) = \int_X f(x)\pi(dx)$$

Monte Carlo with i.i.d. random variables

We rely on the **SLLN**:

$$\lim_{n\to\infty} n^{-1}S_n(f) = \mu(f) a.s.$$

$n^{-1}S_n(f)$ is a *Monte Carlo approximation* of $\mu(f)$.

This has variance

$$var[n^{-1}S_n(f)] = \frac{\mu(f^2) - \mu(f)^2}{n}$$

By the **CLT**:

$$n^{1/2}[n^{-1}S_n(f) - \mu(f)] \to^L X \sim N(0, \mu(\bar{f^2}))$$

The error is hence of order $\mu(\bar{f})^{1/2}n^{-1/2}$. This is slow compared to quadrature in $d = 1$, but performs independently of d, so applies better to high dimensions. f does not need to be continuous here, as in quadrature; error depends only on the variance.

```
# set the seed to fix the pseudo-random numbers
set.seed(12345)

monte.carlo <- function(mu, f, n) {
  S <- 0
  for (i in 1:n) {
    S <- S + f(mu())
```

```
  }
  return(S/n)
}

1 - cos(1)
```

```
## [1] 0.4596977
```

```
vapply(1:6, function(i) monte.carlo(function() runif(1), sin, 10^i), 0)
```

```
## [1] 0.5806699 0.4621368 0.4720845 0.4581522 0.4597137 0.4596813
```

Sampling

Perfect Sampling occurs when we can sample from $\pi = \mu$ *(conjugate prior?)*.

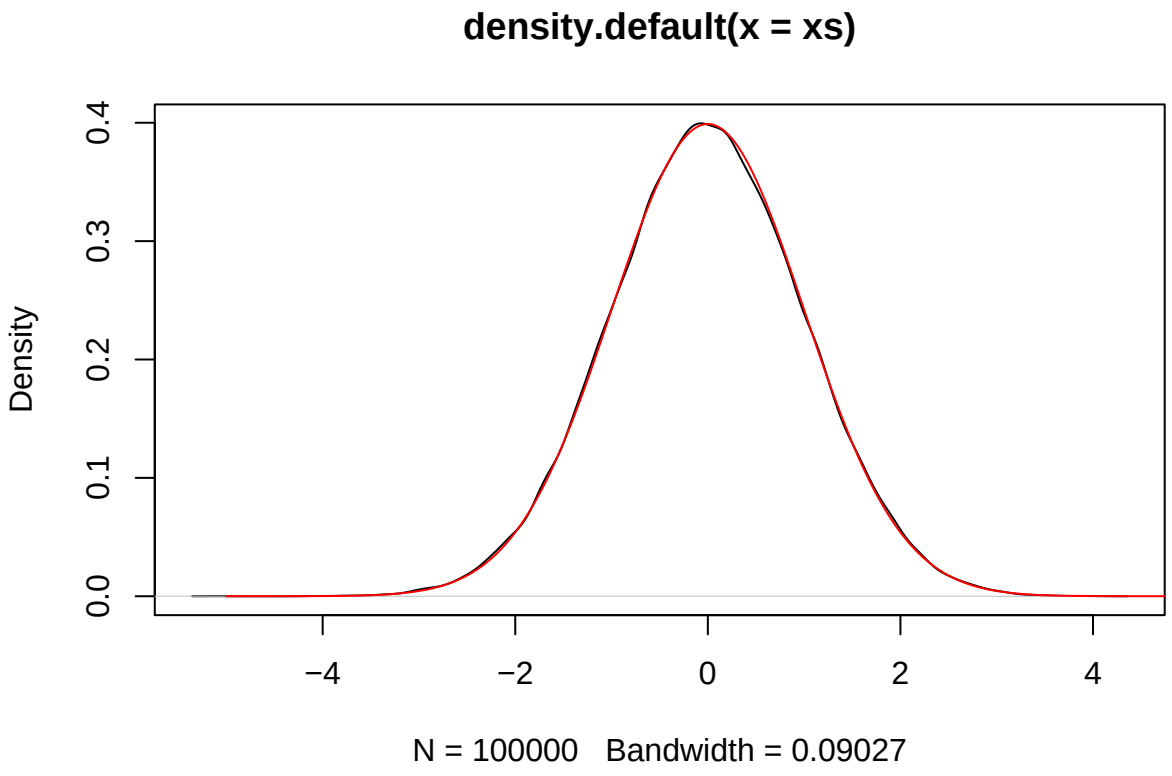
Rejection Sampling is used in general, when we can sample from μ and the supremum of π/μ is bounded above by some M.

We have the **Rejection sampling algorithm:**

1. Sample $X \sim \mu$
2. With probability $\frac{1}{M}\frac{\pi}{\mu}$ output X, otherwise go back to step 1.

This outputs a random variable distributed according to μ. The cost is random, distributed with $Geom(1/M)$.

```
# simple but not numerically stable
# one should use log densities for high-dimensional problems
rejection.sample <- function(pi, mu, M) {
  while (TRUE) {
    x <- mu$sample()
    y <- runif(1) < pi(x)/mu$density(x)/M
    if (y) {
      return(x)
    }
  }
}
# Use Laplace(0,1)
laplace <- list()
laplace$sample <- function() {
  v <- rexp(1)
  ifelse(runif(1) < 0.5, v, -v)
}
laplace$density <- function(x) {
  return(0.5 * exp(-abs(x)))
}
M <- sqrt(2/pi) * exp(0.5) # worked this out theoretically

xs <- replicate(100000, rejection.sample(dnorm, laplace, M))
plot(density(xs))
vs <- seq(-5,5,0.01)
lines(vs, dnorm(vs), col="red")
```



Importance Sampling relies on the density ratio $w = \pi/\mu$ to express π as an integral wrt. μ, approximating π with

$$n^{-1}S_n(f.w)$$

This has variance

$$\pi(f.w^2) - \pi(f)^2$$

This is used because sampling from π can be difficult, or to reduce variance. This does not often scale well with dimension.

```
importance.sample <- function(pi, mu, f, n) {
  w <- function(x) {
    return(pi(x)/mu$density(x))
  }
  fw <- function(x) {
    return(f(x)*w(x))
  }
  monte.carlo(mu$sample, fw, n)
}

# approximate the mean of a N(2,1) r.v. using Laplace(0,1) r.v.s
importance.sample(function(x) dnorm(x, mean=2), laplace, identity, 10000)
```

```
## [1] 1.930252
```

MCMC

Markov Chain techniques use memoryless stochastic processes, justified by MCLLN and MCCLT, for the transition kernel. We only use a restricted class of chains, those which are

- **Time-homogeneous**
- Have a **stationary distribution** μ
- ϕ-irreducible
- **Harris-recurrent**
- Have **total variation** from π bounded above by $M(x)\rho^n$

Metropolis-Hastings Sampling

MH Kernels are often used in practice. These accept jumps according to the probability

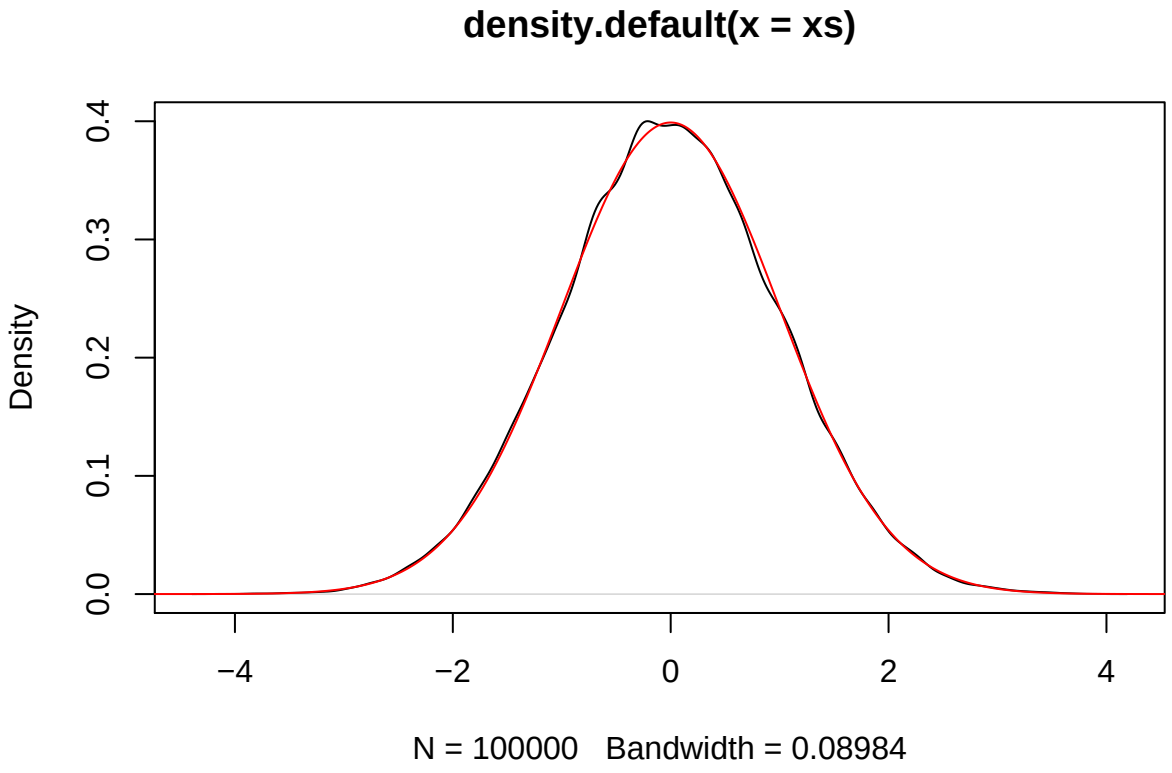
$$\min\left\{1, \frac{\pi(z)q(z,x)}{\pi(x)q(x,z)}\right\}$$

where $Q(x, dz) = q(x, z)\lambda(dz)$ is the proposition kernel.

```
make.metropolis.hastings.kernel <- function(pi, Q) {
  q <- Q$density
  P <- function(x) {
    z <- Q$sample(x)
    alpha <- min(1, pi(z)*q(z,x)/pi(x)/q(x,z))
    ifelse(runif(1) < alpha, z, x)
  }
  return(P)
}

# univariate normal proposal
make.normal.proposal <- function(sigma) {
  Q <- list()
  Q$sample <- function(x) {
    x + sigma*rnorm(1)
  }
  Q$density <- function(x,y) {
    dnorm(y-x, sd=sigma)
  }
  return(Q)
}

# simulate a Markov chain of length n of one-dimensional points
# initial point is x0, P simulates according to Markov kernel
simulate.chain <- function(P, x0, n) {
  xs <- rep(0, n)
  x <- x0
  for (i in 1:n) {
    x <- P(x)
    xs[i] <- x
  }
  return(xs)
}
```

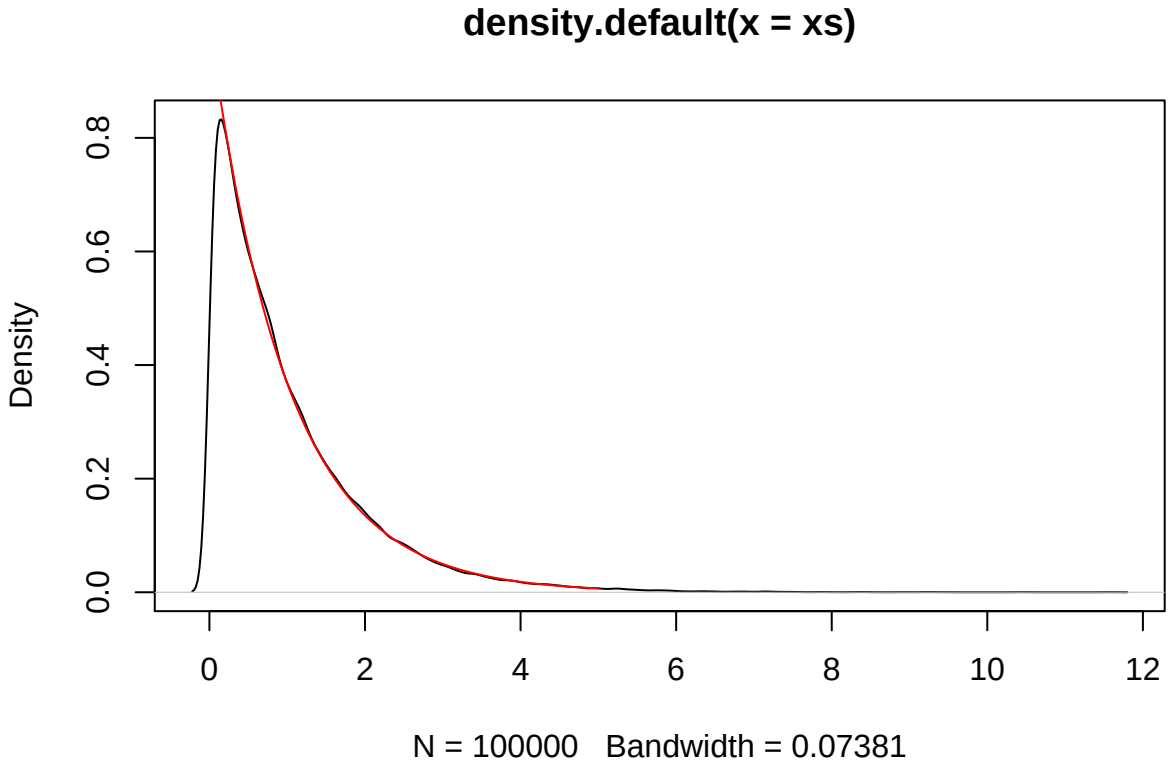
We simulate a standard normal with a normal proposal

```
P <- make.metropolis.hastings.kernel(dnorm, make.normal.proposal(1.0))
xs <- simulate.chain(P, 1, 100000)
plot(density(xs))
vs <- seq(-5,5,0.01)
lines(vs, dnorm(vs), col="red")
```



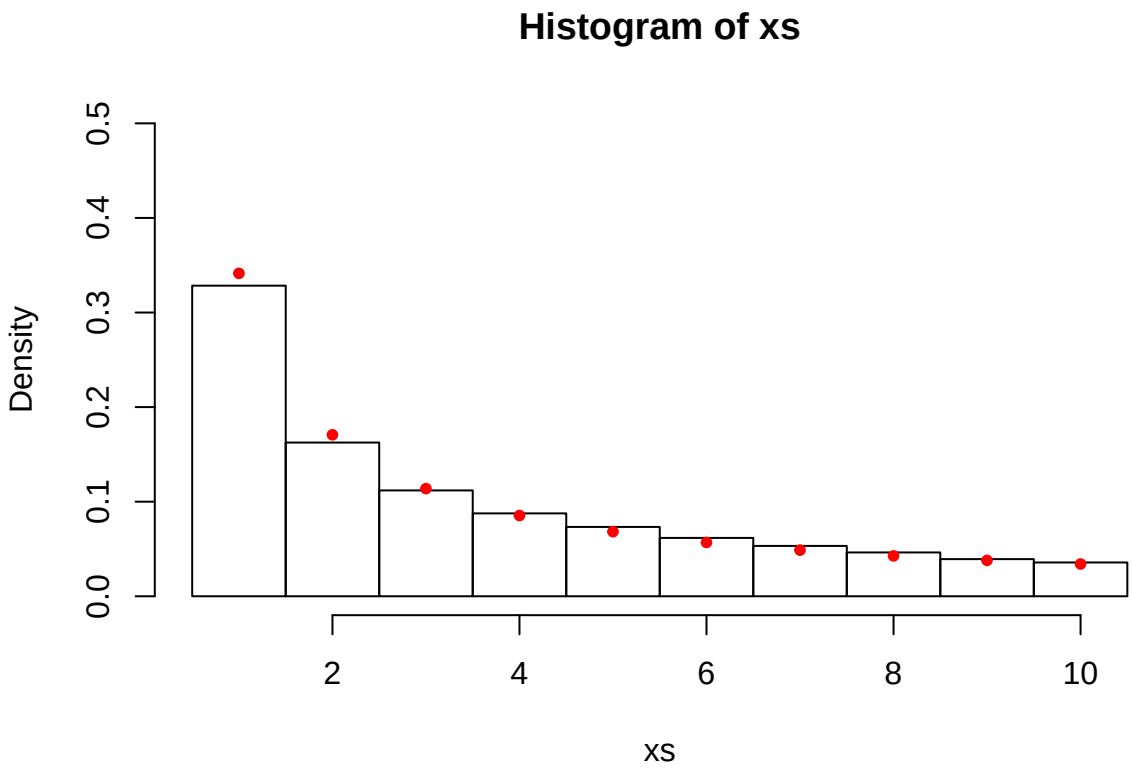
We simulate $Exp(1)$ with a normal proposal

```
P <- make.metropolis.hastings.kernel(dexp, make.normal.proposal(1.0))
xs <- simulate.chain(P, 1, 100000)
plot(density(xs))
vs <- seq(0,5,0.01)
lines(vs, dexp(vs), col="red")
```



This uses a symmetric proposal. We try with an asymmetric proposal

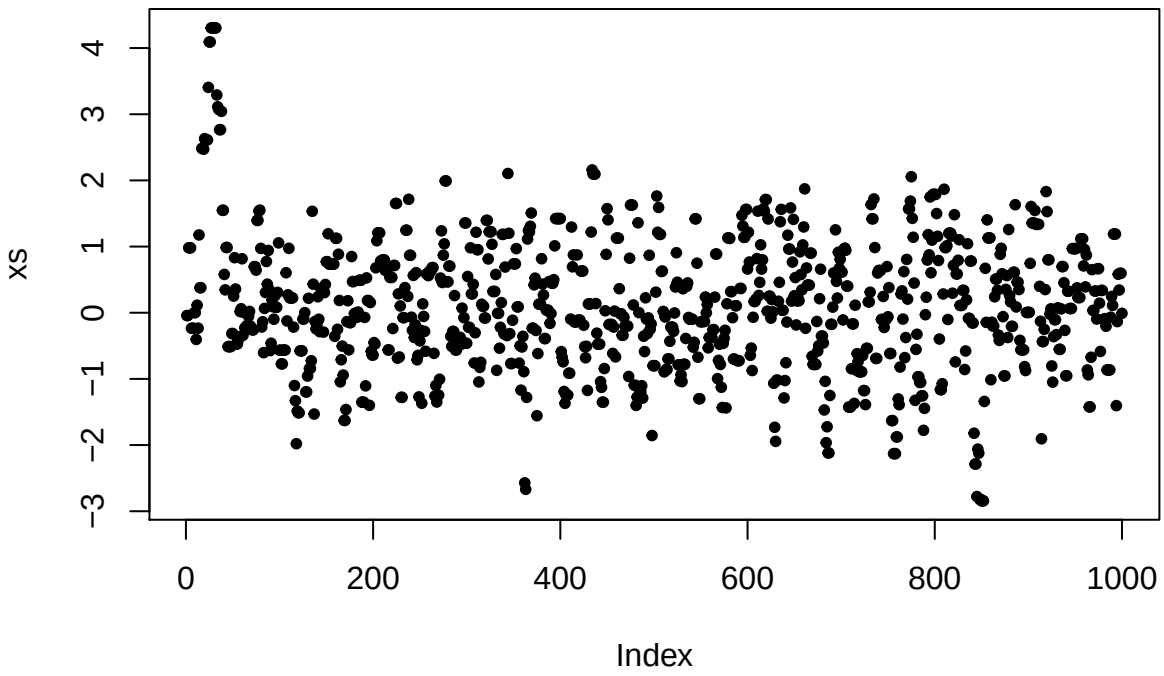
```
step.proposal <- list()
step.proposal$sample <- function(x) x + sample(c(-1,1), 1, prob = c(0.4,0.6))
step.proposal$density <- function(x,y) ifelse(y < x, 0.4, 0.6)

P <- make.metropolis.hastings.kernel(function(x) ifelse(x >= 1 && x <= 10, 1/x, 0), step.proposal)
xs <- simulate.chain(P, 1, 100000)
hist(xs, breaks=seq(min(xs)-0.5, max(xs)+0.5, 1), probability = TRUE, ylim=c(0,0.5))
vs <- seq(1,10,1)
points(vs, 1/vs/sum(1/vs), col="red", pch=20)
```



How do the chains depend on the proposal?

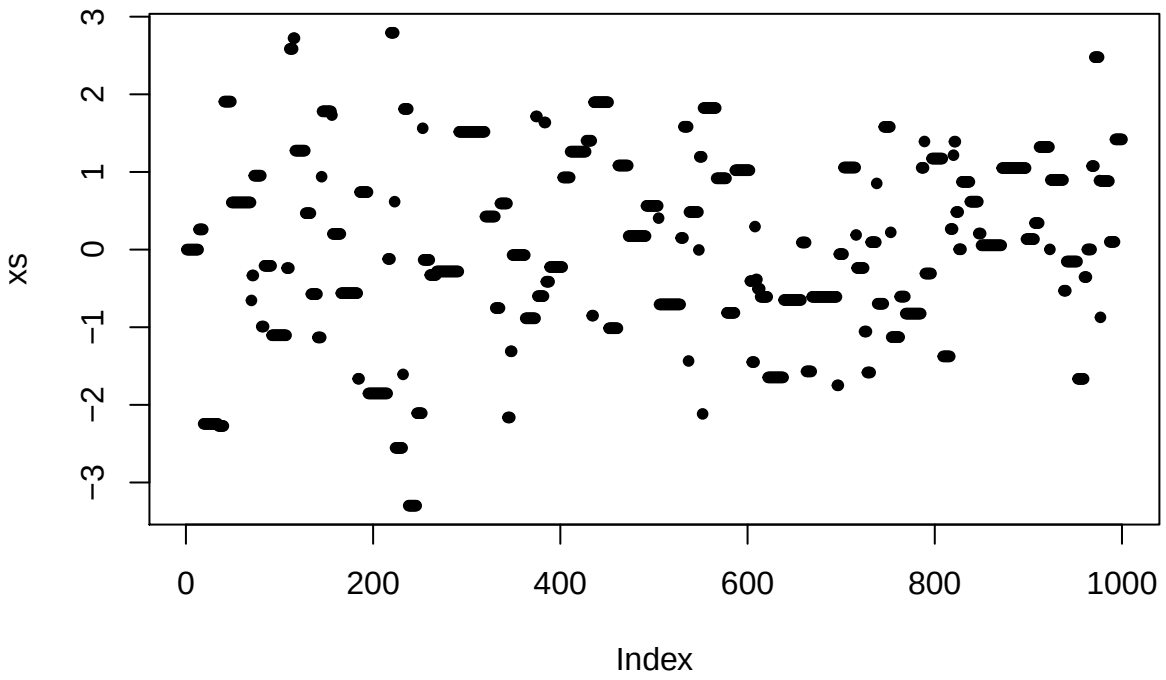
The variance controls the number and size of jumps

```
P <- make.metropolis.hastings.kernel(dnorm, make.normal.proposal(1.0)) #sigma = 1
xs <- simulate.chain(P, 0, 1000)
plot(xs, pch=20)
```




```
P <- make.metropolis.hastings.kernel(dnorm, make.normal.proposal(10.0)) #sigma = 10
```

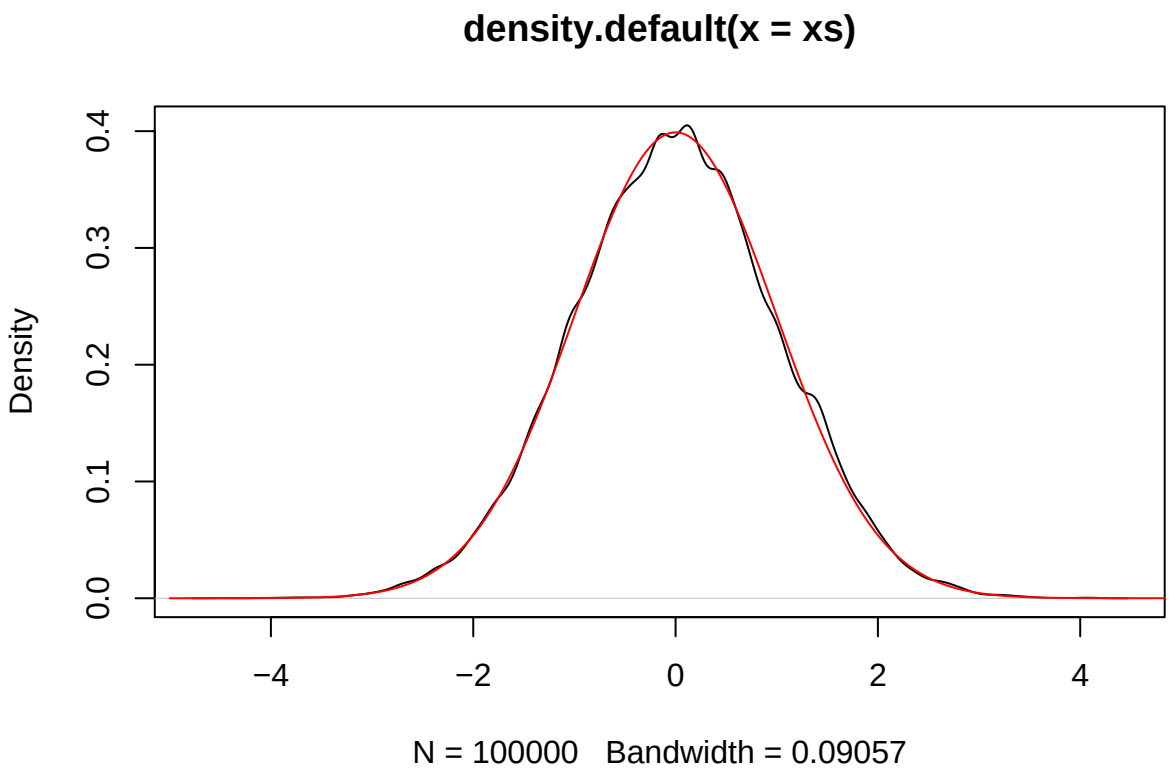
```
xs <- simulate.chain(P, 0, 1000)
plot(xs, pch=20)
```



Verifying stationarity of π is important, and very difficult in general.

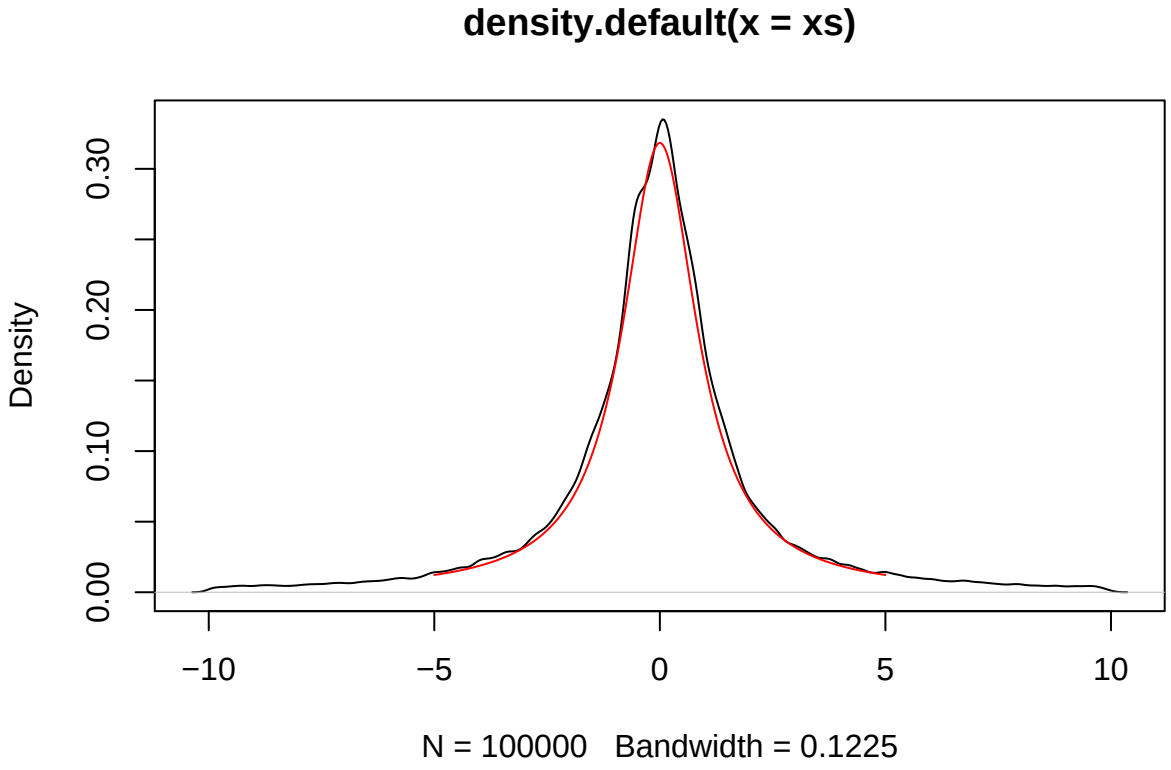
We demonstrate the above with a $Uniform[-10, 10]$ proposal for a normal random variable

```
# univariate uniform proposal
make.uniform.proposal <- function(a,b) {
  Q <- list()
  Q$sample <- function(x) runif(1, a, b)
  Q$density <- function(x,y) {
    dunif(x, a, b)
  }
  return(Q)
}

P <- make.metropolis.hastings.kernel(dnorm, make.uniform.proposal(-10,10)) #Uniform[-10,10] proposal
xs <- simulate.chain(P, 1, 100000)
plot(density(xs))
vs <- seq(-5,5,0.01)
lines(vs, dnorm(vs), col="red")
```



and for a $Cauchy(0, 1)$ variable

```
P <- make.metropolis.hastings.kernel(dcauchy, make.uniform.proposal(-10,10)) #Uniform[-10,10] proposal
xs <- simulate.chain(P, 1, 100000)
plot(density(xs))
vs <- seq(-5,5,0.01)
lines(vs, dcauchy(vs), col="red")
```


density.default(x = xs)
Density
0.00
0.10
0.20
0.30
−10
−5
0
5
10
N = 100000 Bandwidth = 0.1225