Monte Carlo methods

AUTHOR
Karl Gregory

Now that we understand loops and can generate realizations of random variables in R, we can learn about Monte Carlo methods. These are methods which involve generating many random realizations of random variables in order to obtain approximations to quantities that are very difficult (or impossible) to compute exactly. Why are they called *Monte Carlo* methods? Well, Monte Carlo, in Monaco, was at one time a popular gambling destination, and as gambling typically involves random number generation, this name was given to methods which involved drawing many realizations of random variables. Keep in mind that Monte Carlo methods generally will not give exactly the same answer twice! However, we will find that if they are based on large enough samples of random variables, they can give reliable results.

In this note we will consider Monte Carlo methods for approximating the value of an integral and for approximating the minimizer (or maximizer) of a function.

Monte Carlo integral approximation

Suppose we want to integrate a function g over the interval $[a, b]$. That is, suppose we wish to compute

$$I = \int_a^b g(x)dx.$$

It may be that there is no nice anti-derivative of g that we can use to compute the integral exactly. In this case, we can approximate the integral using a Monte Carlo approach.

Hit-or-miss

One Monte Carlo approach is the so-called hit-or-miss approach. It goes like this: Given a value c such that $g(x) \leq c$ for all $x \in [a, b]$, choose some large N and generate independent realizations

$$\begin{aligned} X_1, \ldots, X_N &\sim \text{Uniform}(a, b) \\ Y_1, \ldots, Y_N &\sim \text{Uniform}(0, c). \end{aligned}$$

Then approximate I as

$$I \approx \frac{c(n-a)}{N} \sum_{i=1}^{N} \mathbb{I}(Y_i \leq g(X_i)),$$

where $\mathbb{I}(\cdot)$ is an indicator function (so we count how many times we have $Y_i \leq g(X_i)$).

Example

Suppose we wish to integrate the function

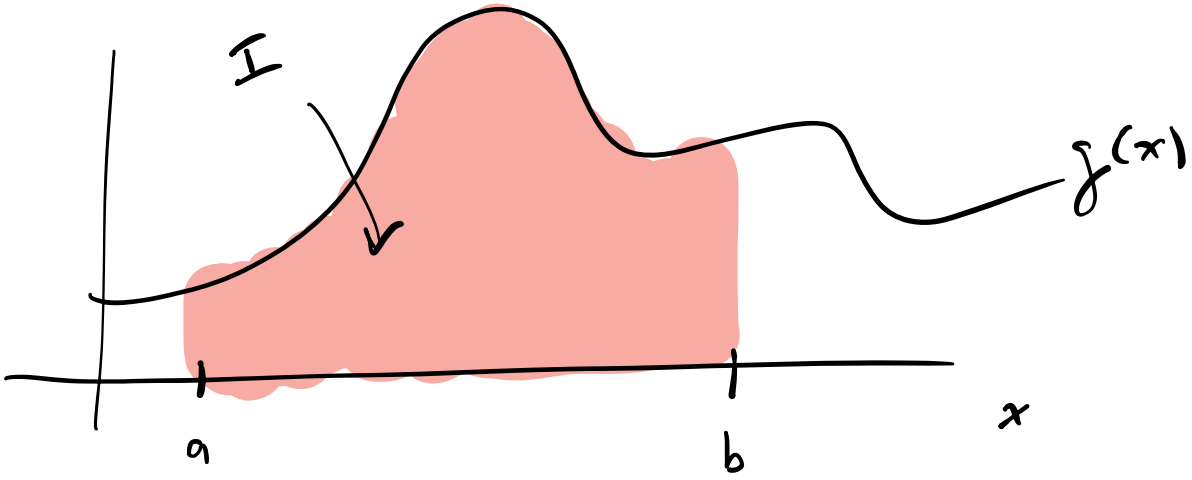
$$g(x) = -\frac{2x \log(x)}{\log(x+2)}$$

over the interval $[0, 1]$. The R code below obtains an approximation using the Monte Carlo hit-or-miss method:

Monte Carlo methods for approximately:

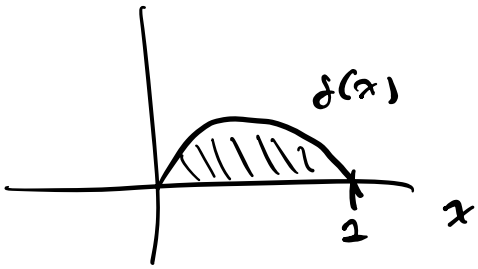
1. Integrating a function
2. Minimizing (maximizing) a function

Integration



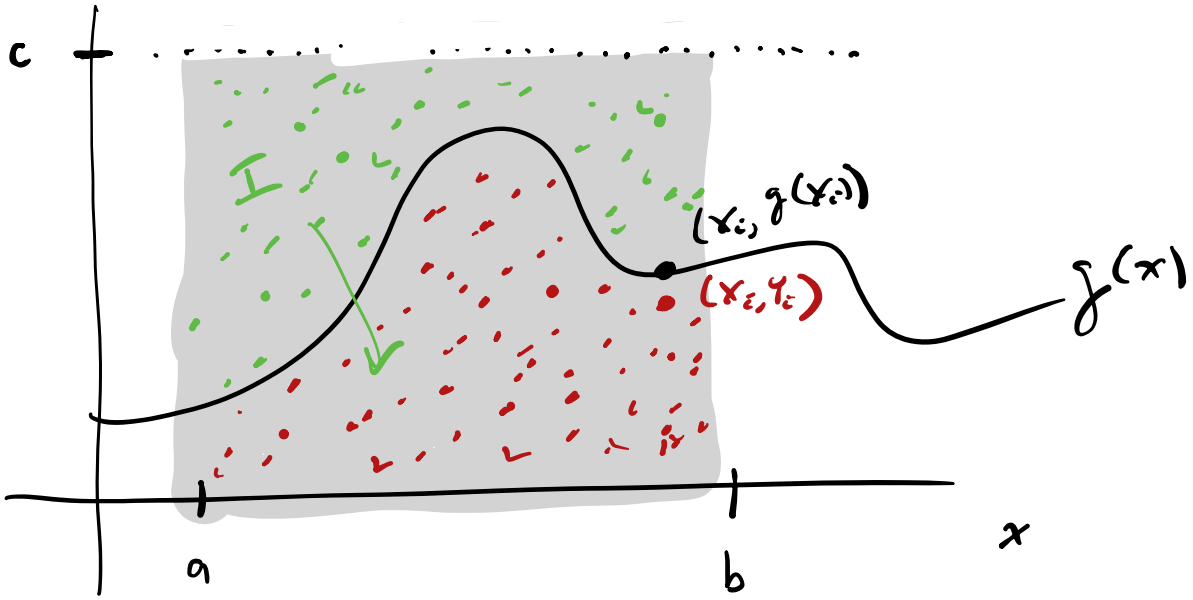
$$I = \int_a^b f(x)\,dx$$

example: $f(x) = -\frac{2x \log x}{\log(x+2)}$



$$I = \int_0^1 \frac{-2x \log x}{\log(x+2)}\,dx$$

M.C. "Hit-or-miss" method



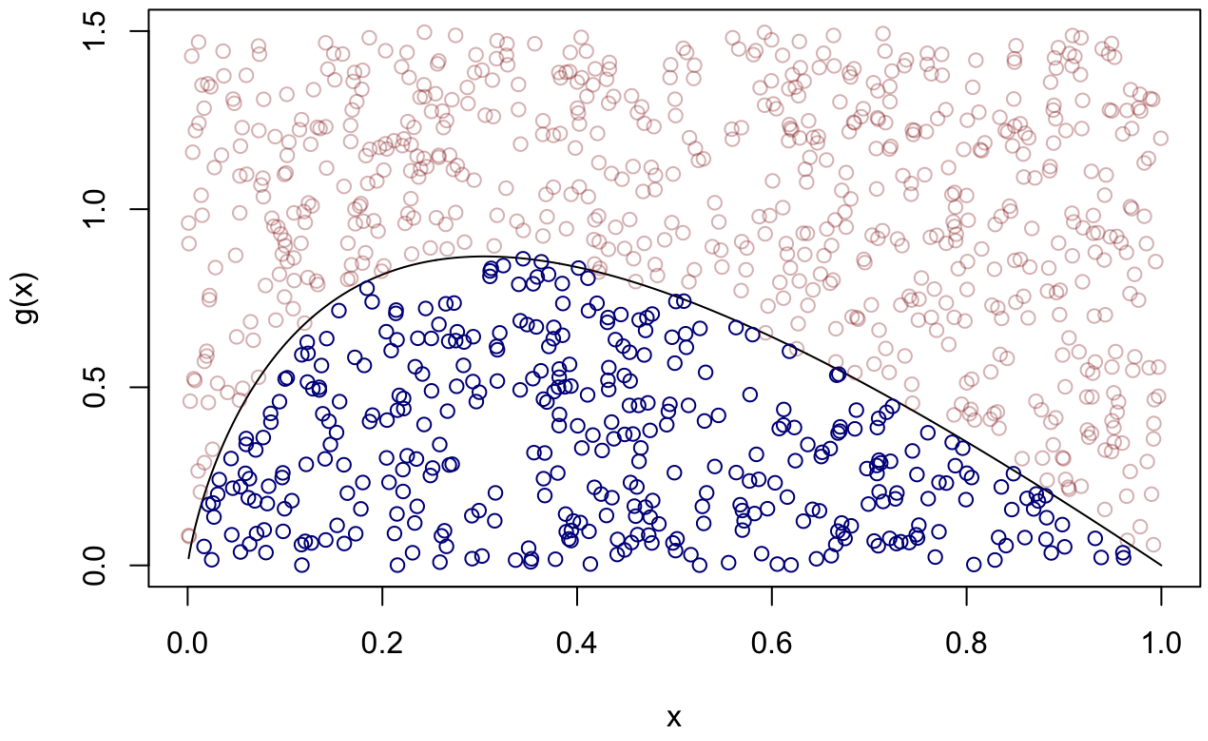
$$I \approx c(b-a) \frac{\#\{\text{dots under the function}\}}{\#\ \text{dots}}$$

Generate $X_1, \ldots, X_N \sim Unif(a, b)$

$Y_1, \ldots, Y_N \sim Unif(0, c)$

$$\#\{\text{dot under}\} = \#\{Y_i \le g(X_i)\}$$

```r
g <- function(x) -2*x*log(x) / log(x + 2)
x <- seq(.001,1, by = 0.001)
gx <- g(x)
plot(gx~x,type = "l",
     ylim = c(0,1.5),
     xlim = c(0,1),
     ylab = "g(x)")

a <- 0
b <- 1
c <- 1.5
N <- 1000
X <- runif(N,a,b)
Y <- runif(N,0,c)
I <- c*(b - a)*mean(Y <= g(X))
points(X,Y,col=ifelse(Y <= g(X),rgb(0,0,.545),rgb(.545,0,0,.3)))
```




```r
I
```

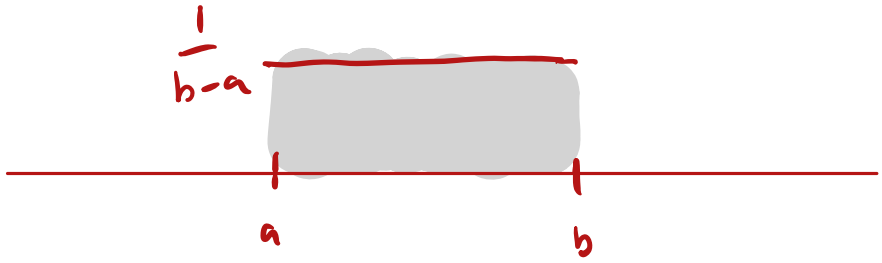
```
[1] 0.555
```

Wolfram Alpha gives the approximation 0.568501 to this integral. From the picture above, we see that all we are doing is approximating the proportion of the rectangular area $[a, b] \times [0, c]$ covered by the function and then scaling this by the total area $c(b - a)$.

Classical M.C. Integral approx.

$$I = \int_a^b g(x)\,dx = (b-a)\int_a^b g(x)\,\frac{1}{b-a}\,dx$$

$$= (b-a)\,\mathbb{E}\,g(U), \quad \text{when } U \sim \text{Unif}(a,b)$$



If X is a r.v. with pdf f then

$$\mathbb{E}\,g(X) = \int_{\mathbb{R}} g(x) f(x)\,dx$$

Draw $U_1, \ldots, U_N \sim \text{Unif}(a,b)$.

Set

$$\hat{I} = (b-a)\,\frac{1}{N}\sum_{i=1}^{N} g(U_i)$$

Classical

Note that we can write

$$I = \int_a^b g(x)dx = (b-a)\int_a^b g(x)\frac{1}{b-a}dx = (b-a)\mathbb{E}g(U),$$

where U is a random variable having the $\text{Uniform}(a,b)$ distribution (since this has pdf equal to $1/(b-a)$ on the interval $[a,b]$). Now, by a result in mathematical statistics called the Strong Law of Large Numbers, the mean of a random sample can be made arbitrarily close to its expected value if a large enough sample is drawn. So the classical Monte Carlo approach to approximating this integral is this: For some large N, generate $U_1, \ldots, U_N$ independently from the $\text{Uniform}(a,b)$ distribution. Then approximate the integral as

$$I \approx (b-a)\frac{1}{N}\sum_{i=1}^{N} g(U_i).$$

Example

For the same integral as in the hit-or-miss example, we have:

```
g <- function(x) -2*x*log(x) / log(x + 2)
N <- 1000
a <- 0
b <- 1
U <- runif(N,a,b)
I <- (b-a)*mean(g(U))
I
```

```
[1] 0.575542
```

Monte Carlo optimization

Suppose we wish to find the value of x which minimizes a function g over an interval $[a,b]$. That is, we wish to find

$$x = \underset{t\in[a,b]}{\text{argmin}}\ g(t).$$

Sometimes it is not possible to use standard calculus methods for finding the minimizer of a function, as the function may not be differentiable or may have many local minima. Such Monte Carlo methods When we are speaking of minimizing or maximizing a function, this function is often referred to as the *objective* function, so we will use this term.

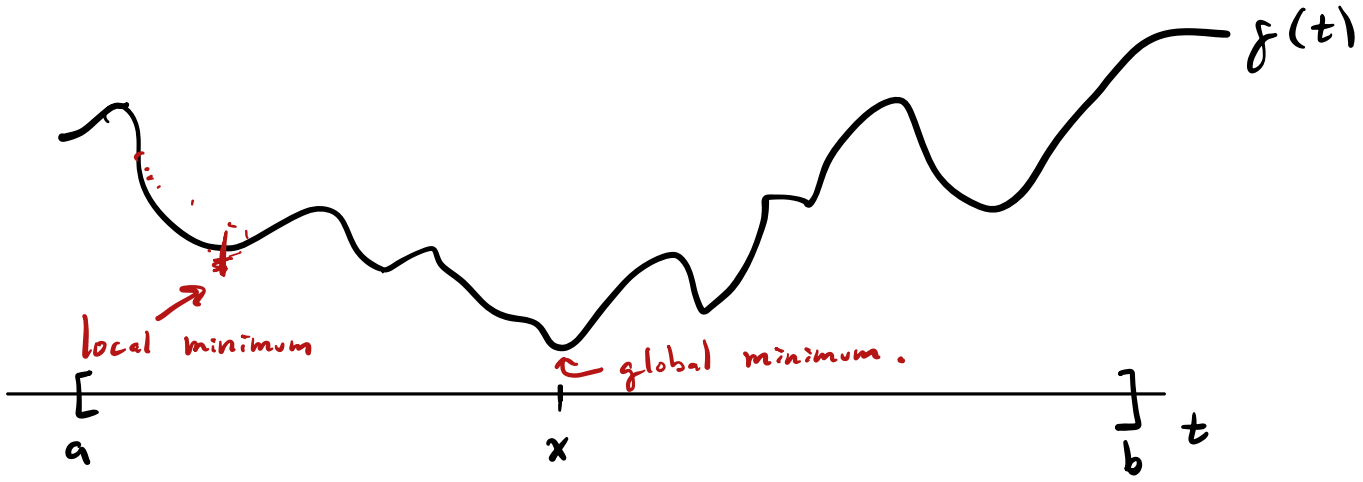
Uniform sampling

The most basic way to seek the minimizer of an objective function g over an interval $[a,b]$ is as follows: For some large N, generate $U_1, \ldots, U_N$ independently from the $\text{Uniform}(a,b)$ distribution and then use the approximation

$$x \approx U_j, \quad \text{where} \quad j = \underset{k\in\{1,\ldots,N\}}{\text{argmin}}\ g(U_k).$$

Example 1

Optimization (Minimization / Maximization) of a function.



We want a M.C. approximation to

$$x = \underset{t \in [a,b]}{\operatorname{argmin}} \; f(t).$$

"Uniform sampling method":

Generate $U_1, \ldots, U_N \sim \text{Unif}(a,b)$.

Then set $\hat{x} = U_j$, where

$$j = \underset{k \in \{1,\ldots,N\}}{\operatorname{argmin}} \; f(U_k).$$

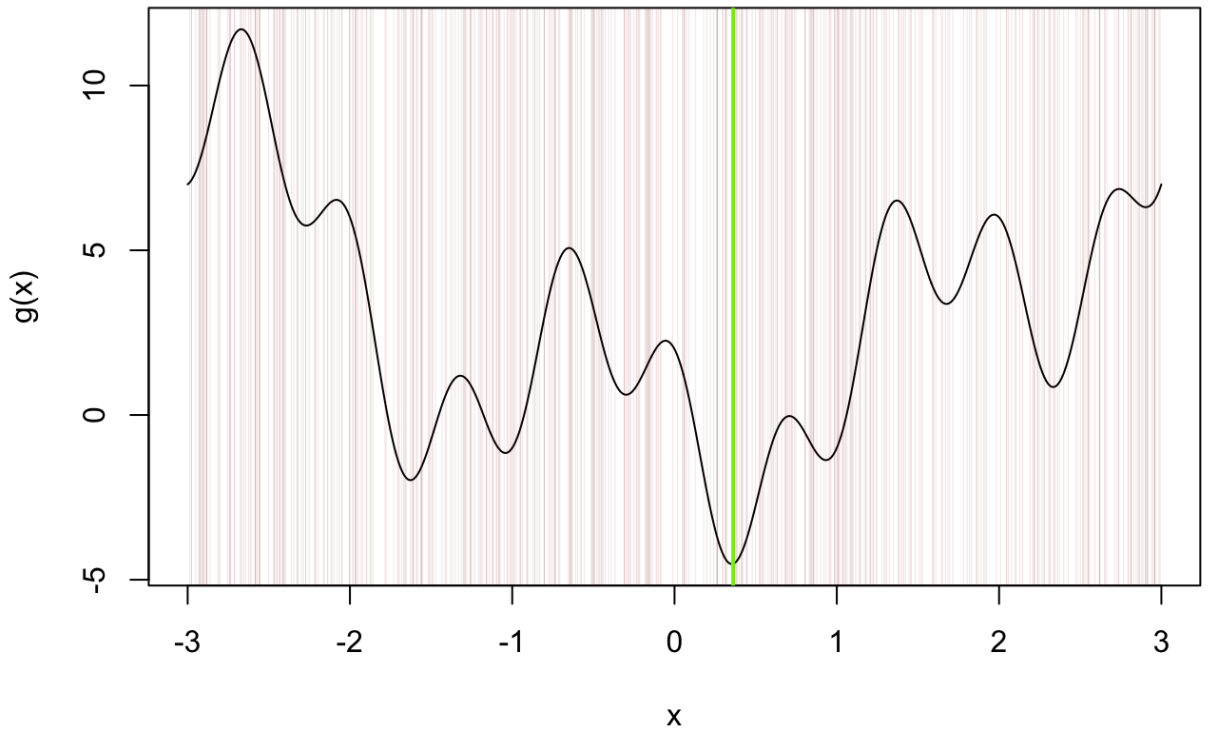
$$g(x) = x^2 - 3\sin(\pi x) + 2\cos(3\pi x)$$

Let's say we want to find the minimizer of the function

$$g(x) = x^2 - 3\sin(\pi x) + 2\cos(3\pi x)$$

over the interval $[-3, 3]$. The R code below finds an approximation with the uniform sampling approach:

```
g <- function(x) x^2 - 3*sin(x*pi) + 2*cos(x*3*pi)
a <- -3
b <- 3
N <- 500
U <- runif(N,a,b)
x0 <- U[which.min(g(U))]  # which.min is a handy function!

x <- seq(-3,3,length=500)
plot(g(x)~x,type = "l")
abline(v = x0,col="chartreuse",lwd = 2)
abline(v = U, col = rgb(0.545,0,0,0.1),lwd = .5)
```



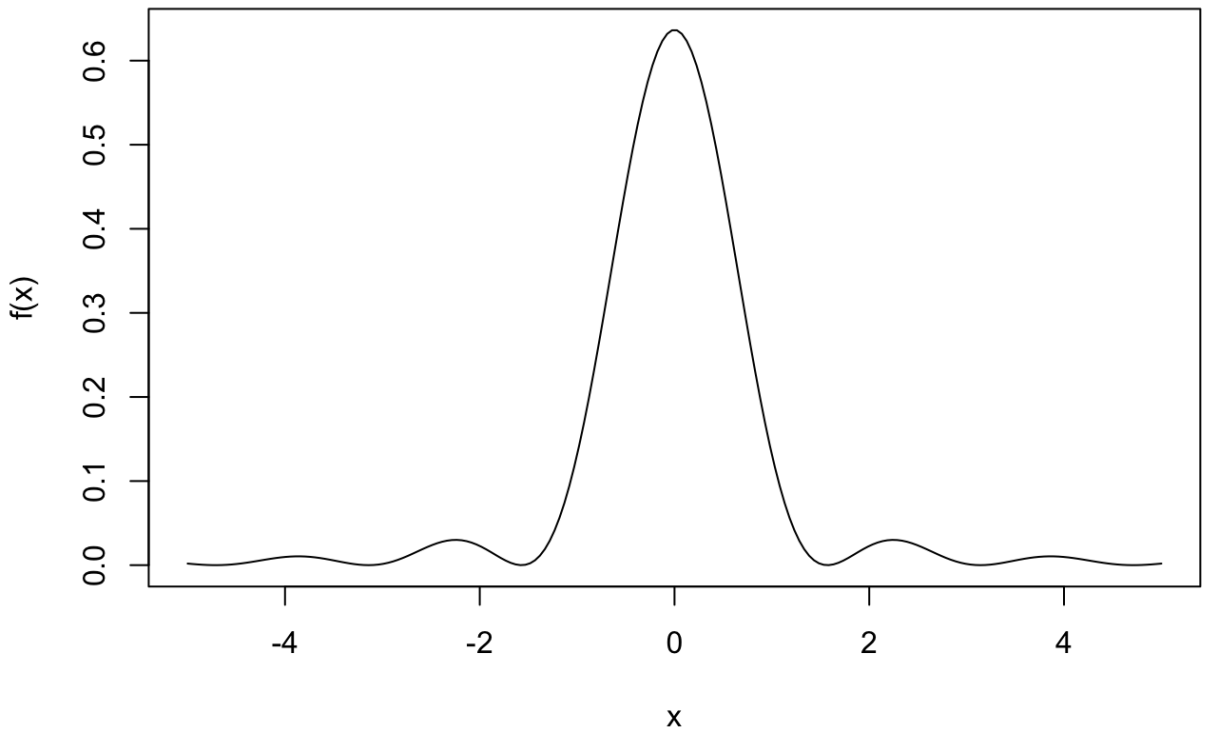
Example 2

Suppose we observe a random sample $X_1, \ldots, X_n$ from a distribution with probability density function given by

$$f(x) = \begin{cases} \dfrac{1}{\pi\lambda}, & x = 0 \\ \dfrac{\lambda}{\pi}\dfrac{\sin^2(x/\lambda)}{x^2}, & x \neq 0, \end{cases}$$

where $\lambda > 0$ is a parameter whose value is unknown. The density looks like this:

```
f <- function(x,lam) ifelse(x == 0, 1/(pi*lam) , sin(x/lam)^2/x^2  * lam / pi)
x <- seq(-5,5,length=200)
fx <- f(x,lam = 1/2)
plot(fx ~ x,
     type = "l",
     ylab = "f(x)")
```



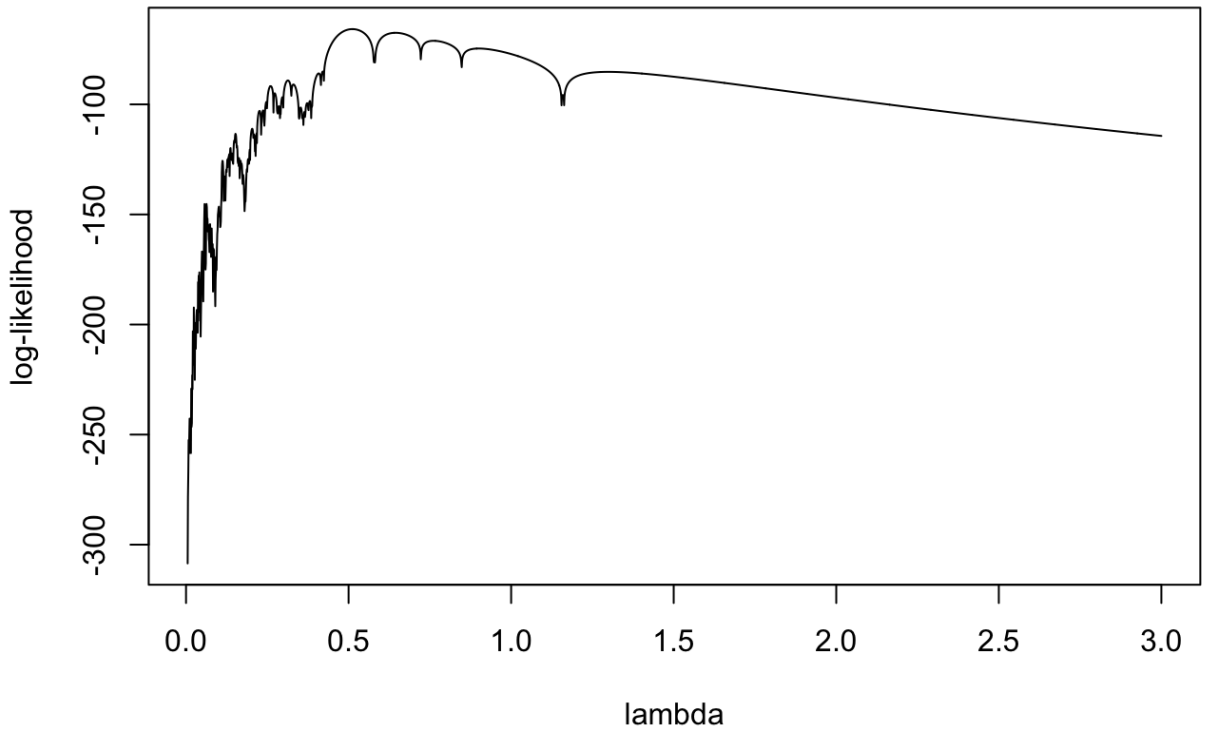
Then the maximum likelihood estimator of the parameter λ is found as the maximizer of the log-likelihood function

$$\ell(\lambda, X_1, \ldots, X_n) = n \log \lambda - n \log \pi + 2 \sum_{i=1}^{n} \log(\sin(X_i/\lambda)) - 2 \sum_{i=1}^{n} \log X_i,$$

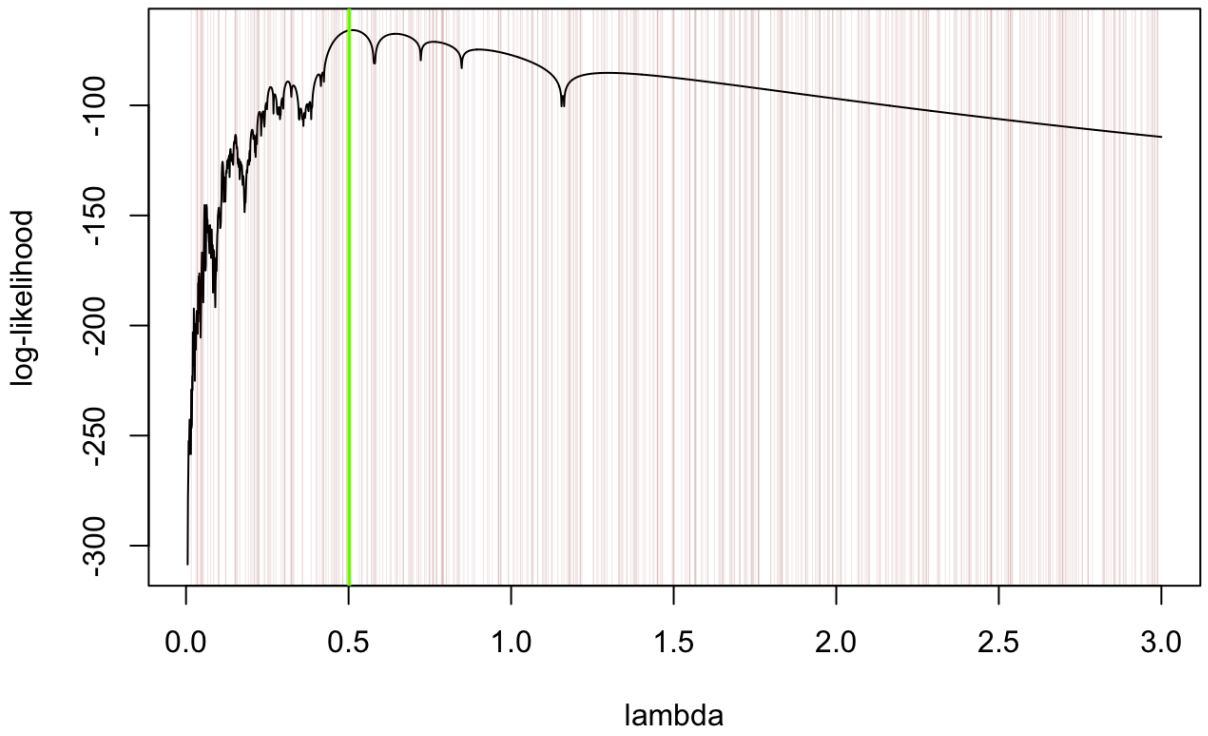
assuming we do not observe any $X_1, \ldots, X_n$ exactly equal to zero. This function, for one particular sample of data, looks like this:

```
# define the log-likelihood function for the parameter lambda based on sample data in X
ll <- function(lam,X) sum(log(f(X,lam)))

# values below generated with lambda = 1/2
X <- scan(text="-0.50101 -2.269045 -0.04700094 2.663053 0.4210084 1.303026 -0.6850137 -1.09102

lamseq <- seq(.005,3,by=0.001)
ll_lamseq <- numeric(length(lamseq))
```

```r
for(i in 1:length(lamseq)) ll_lamseq[i] <- ll(lam = lamseq[i],X)
plot(ll_lamseq~lamseq,
     type = "l",
     ylab="log-likelihood",
     xlab = "lambda")
```




```r
a <- 0.01
b <- 3
N <- 500
U <- runif(N,a,b)
ll_U <- numeric(N)
for(i in 1:N) ll_U[i] <- ll(U[i],X)
x0 <- U[which.max(ll_U)]  # which.max is great too :)

plot(ll_lamseq~lamseq,type = "l",
     xlab = "lambda",
     ylab = "log-likelihood")
abline(v = x0,col="chartreuse",lwd = 2)
abline(v = U, col = rgb(0.545,0,0,0.1),lwd = .5)
```



Example 3

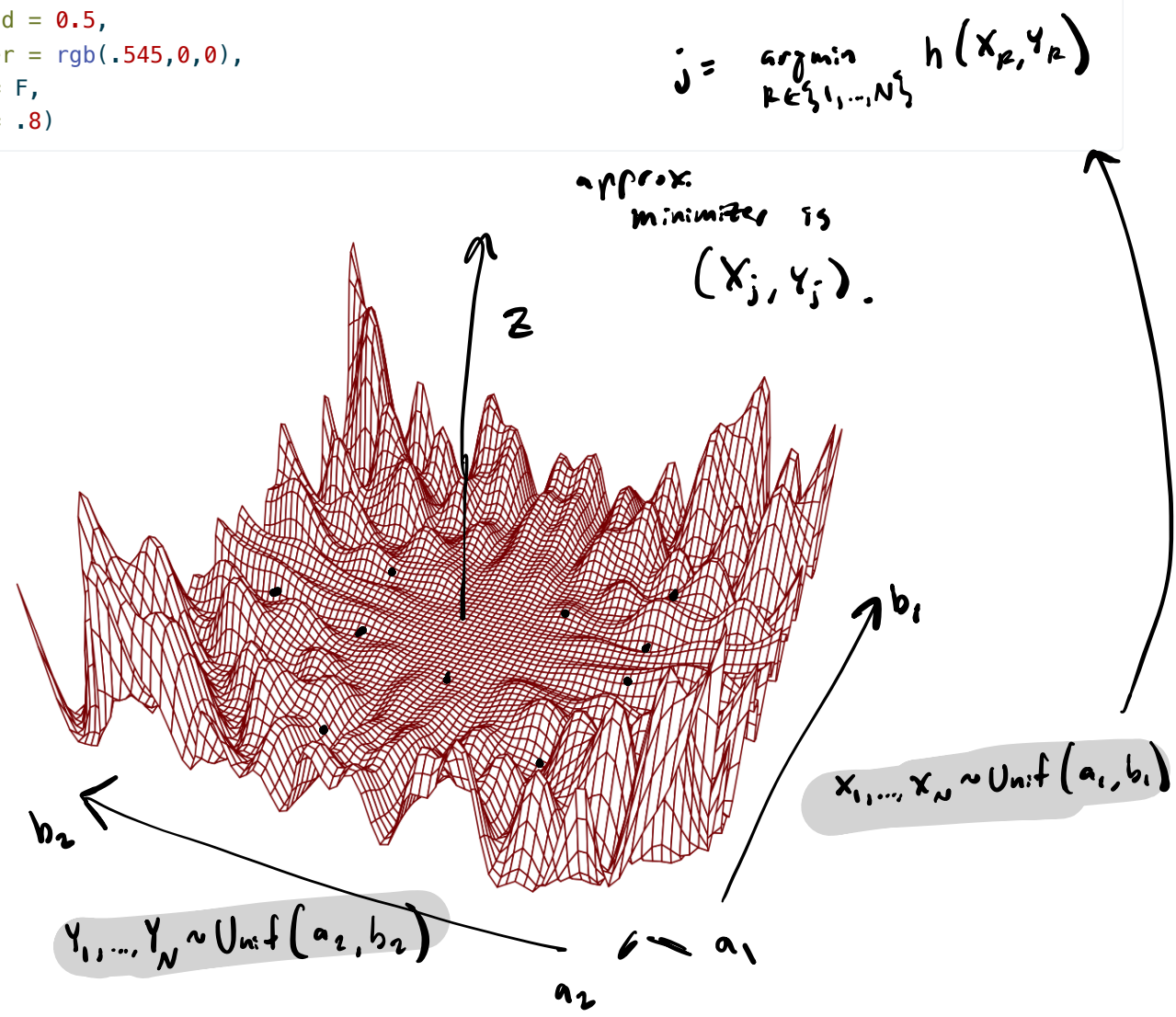
Consider minimizing the bivariate function

$$h(x,y) = (x\sin(20y) + y(\sin(20x))^2\cosh(\sin(10x)x) + (x\cos(10y) - y\sin(10x))^2\cosh(\cos(20y)y)$$

over the rectangle $[-1,1] \times [-1,1]$. We know that the function is minimized at $(x,y) = (0,0)$, but suppose we did not know this.

```
h <- function(x,y){

  a <- (x*sin(20*y) + y*sin(20*x))^2*cosh(sin(10*x)*x)
  b <- (x*cos(10*y) - y*cos(10*x))^2*cosh(cos(20*y)*y)
  val <- a + b
  return(val)

}

x <- seq(-1,1,length=81)
y <- x
z <- outer(x,y,FUN=h)

par(mar=c(1.1,1.1,1.1,1.1))
persp(x,y,z,
      theta = 24,
```

```
      phi = 30,
      cex.axis = 0.8,
      expand = 0.5,
      border = rgb(.545,0,0),
      box = F,
      lwd = .8)
```



In the bivariate case, we must generate $U_1, \ldots, U_n$ as points uniformly distributed in the rectangle $[-1, 1] \times [-1, 1]$, which we demonstrate here:

```
h2 <- function(x) h(x[1],x[2])
N <- 1000
U <- matrix(runif(2*N,-1,1),nrow = N)
gU <- apply(U,1,h2)
x0 <- U[which.min(gU),]
x0
```

```
[1] -0.1959214  0.1323608
```

When searching for the minimizer in a two-dimensional space, we need to generate a much larger number of points to obtain an accurate approximation. In spaces with dimension higher than two, the number of samples needed becomes so large that this method takes too much computational time to be practical.

Simulated annealing is an alternative to uniform sampling which is designed to seek the minimizer in a more computationally efficient way.

Simulated Annealing

The word *annealing* is used for the heating a cooling of a metal. *Simulated annealing* refers to a Monte Carlo method for finding the minimizer of a function which is more carefully constructed than the uniform sampling approach. It goes like this:

To find the minimizer of a objective function g, choose an initial guess x_0, an initial *temperature* T_0, a *step size* σ, and a number of iterations n. Then do the following for $i = 1, \ldots, n$:

1. Propose a new value $x^* = x_{i-1} + \sigma\varepsilon$, where ε represents a perturbation with, say, unit variance.
2. Compute $d = g(x^*) - g(x_{i-1})$. Then, if $d \leq 0$ set $x_i = x^*$ (this means that if the proposed value decreases the function we will keep it). On the other hand, if $d > 0$, set x_i equal to x^* with probability $\exp(-d/T_{i-1})$ and equal to x_{i-1} with probability $1 - \exp(-d/T_{i-1})$ (this means that even if the proposed value does *not* decrease the function but rather increases it, we may still keep it, but it is unlikely that we will keep it if the increase was large).
3. Update the temperature by setting $T_i = T_{i-1}/\log(i+1)$.

The temperatures T_i decrease across the iterations, so as the algorithm proceeds, it becomes less and less likely to accept proposals which do not decrease the objective function.

In Step 1. one option for ε is to generate it from the standard Normal distribution. Note that this algorithm can also work when g takes vector-valued arguments, so we may be searching for the minimizer in a space with dimension greater than 1. In Step 3., one may update the temperatures according to some other rule—the point being that the temperatures should decrease, so in the latter iterations one insists strongly that the objective function should not increase.

In Step 2. we can do an action "with probability" *something* by generating $U \sim \text{Uniform}(0,1)$ and doing the action if U is less than *something*.

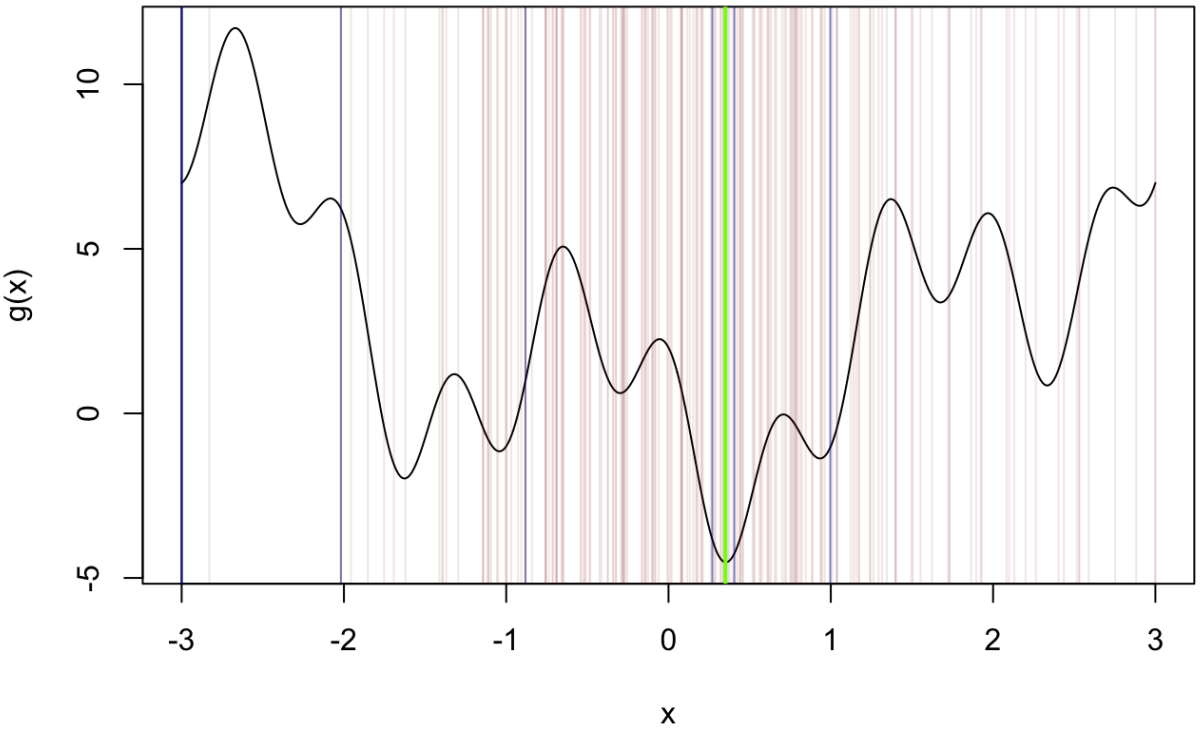
Example

The R code below defines a function which performs simulated annealing to search for the minimizer of a function g over an interval $[a, b]$. Then simulated annealing is demonstrated for the same minimization as in Example 1 in the uniform sampling section.

Note that the function defined below is a function which take as one of its arguments another function. This is something new for us. The `...` which appears in the list of arguments in the function definition stands in for any arguments which may need to be passed along to the function put in for `f`. We will make use of the `...` in the next example.

```
simann <- function(g,a,b,tmp,st,iter,...){

  x0 <- runif(1,a,b)
  vals <- numeric(iter)
  keep <- logical(iter)
  for(i in 1:iter){

    x1 <- x0 + st * rnorm(1)                                  # generate candidate next value
    x1 <- a*(x1 < a) + b*(x1 > b) + x1*((x1 >= a) & (x1 <= b)) # keep value of x1 inside inter
    vals[i] <- x1

    g0 <- g(x0,...)
    g1 <- g(x1,...)
```

```
    d <- g1 - g0

    tmp <- tmp / log(i+1)
    ap <- exp(-d / tmp)

    if((d <= 0) | (runif(1) < ap)){

      x0 <- x1
      keep[i] <- T

    }

  }

  output <- list(x = x0,
                 vals = vals,
                 keep = keep)

}

g <- function(x) x^2 - 3*sin(x*pi) + 2*cos(x*3*pi)
simann_out <- simann(g,a=-3,b=3,tmp=100,st=1,iter=200)

# make a plot
x <- seq(-3,3,length=500)
plot(g(x)~x,type = "l")
abline(v = simann_out$vals, col = ifelse(simann_out$keep,rgb(0,0,.545,.5),rgb(0.545,0,0,.1)))
abline(v = simann_out$x,col="chartreuse",lwd = 2)
```



Practice

Practice writing code and anticipating the output of code with the following exercises.

Write code

Read code