Stat S610. Fall 2021. Homework 8.

Due 12/09/2021

Premise of the Assignment

In this assignment, you will build a workflow for (pseudo) random number generation (PRNG). We will assume low-level random number generation; the default generator in `R` is a Mersenne-Twister, which produces pseudo random numbers that are approximately uniformly distributed on a unit cube in 623 dimensions. While this is not the most modern PRNG, it is sufficient for most purposes and fairly quick. In addition to the `kind` of random number engine, the function `RNGkind` allows one to set the algorithm used for normal (`normal.kind`) and discrete uniform (`sample.kind`) random number generation. The default values for these are `Inversion` and `Rejection`, respectively.

This assignment focuses on algorithms that generate random variates from a univariate distribution when given a stream of random numbers that are uniformly distributed in the unit interval. There are three algorithms for which you will write general purpose functions: inverse canonical transformation sampling, rejection sampling, and one flavors of adaptive rejection sampling.

You will structure your programs in the following fashion.

(1) For each algorithm, you will have a project directory with the following subdirectories: `headers`, `headers_testing`, `main`, and `main_testing`.

(2) The `main` directory contains the functions that you want the user to access. The `main_testing` directory contains a `test_that` block for testing the functions in your `main` directory.

(3) The `headers` directory contains the functions that are used by the functions defined in the `main` and that you do not want the user to access. The `headers_testing` directory contains a `test_that` block for testing the functions in your `headers` directory.

(4) All of the directories for the three algorithms are to be housed in a common GitHub repository (private preferred) that will be made accessible to the grader and me for the purposes of cloning the repository and running tests for grading. All calls to source for an algorithm should be relative to the directory housing the four directories for that algorithm.

Algorithm 1: Inverse Canonical Transformation Sampling

Suppose that X is a continuous random variable following distribution function F_X and define Y to be the random variable given by $Y = F_X(X)$. Then the distribution function for Y is

$$F_Y(y) = P(Y \leq y) = P(F_X(X) \leq y) = P\left(X \leq F_X^{-1}(y)\right) = F_X\left(F_X^{-1}(y)\right) = y$$

and Y is uniformly distributed on ghe interval $(0, 1)$. This is the canonical transformation. To use this to generate random variables, we can generate $Y \sim U(0, 1)$ and then set $X = F_X^{-1}(Y)$.

When F_X^{-1} is readily available, then sampling is easy. When it is not available, we have to write a function that will invert F_X to produce the estimate.

(1) Write a function in your `main` directory that is called `inverse_sampling` and takes in four arguments: a sample size `n`, a function `cdf`, its inverse `cdf_inv`, and the argument `...` that will pass further arguments to `cdf` and `cdf_inv`. The function `inverse_sampling` does three things. First, `inverse_sampling` does input checks. If both `cdf` and `cdf_inv` are missing, stops and reports an appropriate error. It does appropriate checks for `n` being a positive integer, stopping on errors or warning about type changing if necessary. If `cdf_inv` is missing but `cdf` is not missing, then it defines `cdf_inv` by calling a function named `def_cdf_inv` and evaluating it at the arguments `cdf` and `...` (the further arguments passer). Second, `inverse_sampling` used `Vectorize` to define `V_cdf_inv` that is a vectorized version of `cdf_inv` that is vectorized in its first argument (obtained using `names(cdf_inv)[1]`). Third, `inverse_sampling` returns `V_cdf_inv(runif(n),...)`.

(2) Write a function `def_cdf_inv` in your `headers` directory that will be used to define `cdf_inv` in `inverse_sampling` when it is missing. The function `def_cdf_inv` is to take in two arguments the function `cdf` and `...` (the further arguments passer). The function `def_cdf_inv` does three things. First, it defines the function `f` which has three arguments: `x`, `p`, and `...` (the further arguments passer). The function `f` returns `cdf` evaluated at `x` and `...` (the further arguments passer) minus `p`. Second, `def_cdf_inv` defines the function `f_root` which takes in two arguments: `p` and `...` (the further arguments passer). The function `f_root` returns the root of `f` found using `uniroot` with the arguments: `f=f`, `interval=c(-1,1)*1e3`, `p=p`, `...`, `extendInt="upX"`, `tol=.Machine$double.eps^0.5`, and `maxiter=1e4`. The root is `root` in the list of the output from a call to `uniroot`. Third, `def_cdf_inv` returns the function `f_root`.

(3) Write a `test_that` block in the `headers_testing` that tests that the function returned from `def_cdf_inv( pnorm, mean=100, sd=25)` evaluated at inputs `p` being in `0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9` and additional arguments `mean=100, sd=25` matches the appropriate output from `qnorm`.

(4) Write a `test_that` block in the `main_testing` folder that tests whether the output from calling `inverse_sampling(100,pnorm,qnorm,mean=100,sd=25)` matches that from a normal distribution by first setting the seed to be `1234567890`, generating the output, and then running `ks.test` appropriately and checking that the output is larger than 0.01. Do a similar test on the output from `inverse_sampling(100,pnorm,mean=100,sd=25)`.

Algorithm 2: Rejection Sampling

Suppose that X is a continuous random variable following distribution function F that is continuous and has associated density function f. Suppose that Y is a random variable following distribution function G that is continuous and has associated density function g. Further suppose that

$$M = \operatorname*{ess\,sup}_{r\in\mathbb{R}} \frac{f(r)}{g(r)}$$

is finite. Here, ess sup is the essential supremum, which is the supremum excluding any set of probability 0. Any essential upper bound M for $f(r)/g(r)$ works in the algorithm. The least of the the essential upper bounds, the essential supremum, provides the most efficient algorithm that proposes from G and accepts or rejects the proposals. It is critical that f/g is essentially bounded above, otherwise M is infinite and rejection sampling cannot be employed. Any density g for which $f(r) \le Mg(r)$ almost everywhere for some M is said to be an enveloping density for f and M is often referred to as an enveloping constant for f relative to g.

Rejection sampling is the following algorithm. Draw (Y_1, U_1) by independently drawing Y_1 from G and U_1 from $U(0, 1)$. If $U \le \frac{f(Y_1)}{Mg(Y_1)}$ then set $X = Y_1$, otherwise draw a new (Y, U) pair (Y_2, U_2) and test it. Continue in this fashion until an X is drawn. To see that the drawn X follows F, note that

$$\begin{aligned} P(X \le x) &= \sum_{i=1}^{\infty} P\left(Y_i \le x, U_i \le \tfrac{f(Y_i)}{Mg(Y_i)}, U_j > \tfrac{f(Y_j)}{Mg(Y_j)} \text{ for } j < i\right) \\ &= \sum_{i=1}^{\infty} \left(1 - \tfrac{1}{M}\right)^{i-1} \int_{-\infty}^{x} \tfrac{f(y)}{Mg(y)} g(y)\mathrm{d}y \\ &= F(x)\tfrac{1}{M} \sum_{i=1}^{\infty} \left(1 - \tfrac{1}{M}\right)^{i-1} \\ &= F(x) \end{aligned}$$

The algorithm proposes variates from G and rejects or accepts them as variates from F based on

(1) Write a function in your `main` directory that is called `rejection_sampling` and takes in five arguments: a sample size `n`, a target density function `targ_pdf`, a proposal density function `prop_pdf`, a random variate generator for the proposal density `prop_ran`, and an enveloping constant `env_const`. The function `rejection_sampling` does three things. First, `rejection_sampling` does input checks. After doing appropriate checks for `n` being a positive integer, stopping on errors or warning about type changing if necessary, it checks to see if `env_const` is missing. If `env_const` is missing, then `env_const` is defined by calling a function `def_env_const`, evaluating it at the arguments `targ_pdf` and `prop_pdf`. Finding the enveloping constant using numerical methods can lead to a bad choice of constant (not large enough of an enveloping constant because a local and not global mode was found), and so output an appropriate warning when this happens. Second, `rejection_sampling` performs rejection sampling by proposing from `prop_ran` and accepting or rejecting based on `targ_pdf(prop_value)<=M*g(prop_value)`. This is easiest achieved with a `while` loop, though there are more efficient block generation methods. Third, `rejection_sampling` returns the vector of accepted proposals. The output must have length `n`.

(2) Write a function `def_env_const` in your `headers` directory that will be used to define `env_const` in `rejection_sampling` when it is missing. The function `def_env_const` is to take in two arguments the function `targ_pdf` and `prop_pdf`. The function `def_cdf_inv` does three things. First, it defines the function `h` which has one argument `x`. The function `f` checks if the value of `targ_pdf` is less than `.Machine$double.eps` and returns 0 if it is. If it is not, `f` returns `targ_pdf(x)/prop_pdf(x)`. Second, `def_env_const` calls `optimize` with a line like

```
opt = optimize(f,interval=c(-1,1)*10,maximum=TRUE,tol=.Machine$double.eps^0.5)
```

Note that this is a bad idea because the maximum is not guaranteed to be in the interval and the algorithm might only find a local maximum that is not a global maximum. If we were to want to implement an idea like this for real, we would want to find a more robust method of computing the maximum. Third, `def_env_const` returns the value `opt$objective`.

(3) Write a `test_that` block in the `headers_testing` that tests that the value returned from `def_env_const( dnorm, function(x) 1/pi/(1+x^2))` is less than 1.521 and that the value returned from `def_env_const( dexp, function(x) 1/(1+x)^2)` is less than 1.472.

(4) Write a `test_that` block in the `main_testing` folder that tests whether the output from calling `rejection_sampling(100,dnorm,function(x) dt(x,1),function(n) rt(n,1),1.52)` matches that from a standard normal distribution by first setting the seed to be `1234567890`, generating the output, and then running `ks.test` appropriately and checking that the output is larger than 0.01. Do a similar test on the output from `rejection_sampling(100,dnorm,function(x) 1/2*exp(-abs(x)),function(n) rexp(n,1)*(2*rbinom(n,1,0.5)-1),1.32)`. Repeat these two tests with the `env_const` input missing.

Algorithm 3: Adaptive Rejection Sampling Variant 1

Suppose that X is a continuous random variable following distribution function F that is continuous and has associated density function f. Further suppose that f is log-concave. That is, the function $h(x) = \log(f(x))$ is concave. If h is differentiable, then we can compute the tangent line to the the graph of h at any point x. This tangent line has to lie above or on the graph for h due to the concavity of h. If h were merely concave and continuous but had no derivative at some point x, then any tangent line of the graph of h at x would lie above the graph for h.

This provides a way to iteratively refine an enveloping function for h. Let (ℓ, u) define the support for f and assume that h is concave on its support. we need to break ourselves into three cases. Case 1, support bounded above and below, $-\infty < \ell < u < \infty$. Case 2, support bounded above, $-\infty = \ell < u < \infty$. Case 3, support bounded below, $-\infty < \ell < u = \infty$. Case 4, support unbounded, $-\infty = \ell < u = \infty$.

We begin with two points $x_1 < x_2$ and their values under h, given by $y_1 = h(x_1)$ and $y_2 = h(x_2)$. Let $s_1 = h'(x_1)$ and $s_2 = h'(x_2)$. If $\ell = -\infty$, then we need $s_1 > 0$ (Cases 2 and 4). If $u = \infty$, then we need $s_2 < 0$ (Cases 3 and 4). We make a continuous, piece-wise linear function using the tangent lines. The exponential of this function is an integrable envelope for f. We use the accompanying enveloping distribution to do rejection sampling from f.

The rejection or acceptance of a proposed point x is first determined using the squeeze principle. A random uniform variate v is drawn and if

$$v < \frac{g_{\text{lower}}(x)}{g_{\text{upper}}(x)}$$

then the point is accepted as coming from f (g_{lower} and g_{upper} are the lower and upper envelope functions). Alternatively, this could be written as

$$\log(v) < \log(g_{\text{lower}}(x)) - \log(g_{\text{upper}}(x))$$

so that calculations are done on the log-scale and are more numerically stable. If the point x is not accepted using the squeeze principle, then the comparison to f is made and the point is accepted if

$$v < \frac{f(x)}{g_{\text{upper}}(x)} \qquad \text{or alternatively} \qquad \log(v) < h(x) - \log(g_{\text{upper}}(x))$$

where the second inequality uses $h = \log(f)$ and is more numerically stable.

Whenever a point is not accepted using the squeeze principle, then the function f (or better, $h = \log(f)$) had to be evaluated at a new point. This point is $(x, h(x))$ is added to the upper and lower envelopes (on the log scale) and the envelopes are updated before doing the next draw. The point is added to the envelope whether it was accepted or rejected using the direct comparison to f after the failure to accept using the squeeze principle.

`R` functions for initializing the envelopes, updating the envelopes, evaluating the envelopes, and sampling the upper envelope are in the file `hw8_q3_header`. Below are examples of using the functions in that file that are called when programming an adaptive rejection sampler. There are other functions that assist these demonstrated functions, but you do not need to call any of these other functions in your main function. Note that the environment needs to be made first and that persists through the draws.

```r
source("hw8_q3_header.R")

# setting things up,
# this would all have to be input to the main function
h = function(x,mu,sigma) -(x-mu)^2/(2*sigma^2)-0.5*log(2*pi*sigma^2)
interval=c(-Inf,Inf)
mu = 3
sigma = 2
x = mu+c(-1,1)*sigma

# making an empty environment to keep the envelopes in
envelope_env = new.env()

initialize_envelope_info(envelope_env,h,interval,x,mu=mu,sigma=sigma)
# this function has no output
# just side effects updating the environment envelope_env
# mu and sigma are in the ... slot for this function
# they are the inputs to h in addition to x

# getting a new point to demonstrate the updating function
x_new = rnorm(1,mu,sigma)
y_new = h(x_new,mu,sigma)

update_environment_info(envelope_env,h,x_new,y_new,mu=mu,sigma=sigma)
# this function has no output
# just side effects updating the environment envelope_env
# mu and sigma are in the ... slot for this function
# they are the inputs to h in addition to x

# these are as their names suggest
x_val = upper_envelope_sample(envelope_env)
upper_y_val = upper_envelope_evaluate(x_val, envelope_env, log=TRUE)
lower_y_val = lower_envelope_evaluate(x_val, envelope_env, log=TRUE)
```

Your task is to write a main function called `adaptive_rejection_sampling` that takes in h, the support interval, initial points x, `...` which represents additional arguments to h other than x (x must be the first argument of h). This function is then to do adaptive rejection sampling from $f = exp(h)$ and return the sampled points. Write appropriate tests for this function (but not for any of the functions in the provided header file) and structure the files in an appropriate directory structure.

Remark: Adaptive Rejection Sampling (and Rejection Sampling more broadly) can be used when we only know f up to a multiplicative constant (for instance, we might have the functional form for f, but the area under that function is not 1; this is the same as knowing $h = \log(f)$ only up to an additive constant). The sample pairs that are accepted (U, X) would be uniformly distributed under the graph of whatever that function is. Having the area under f be something other than 1 only effects the scaling of the U margin and not the distribution of the X margin. So the X draws would be from the distribution whose density is the normalized f.