Computer assignment MT5012, VT2019

Monte-Carlo integration

Crude Monte-Carlo integration aims at numerically approximate an expectation/integral $E(h(X)) = \int h(x) f_X(x)\, dx$ by

- Simulate a realisation x of X
- Compute $h(x)$
- Repeat the above two steps a large number of times and return the average computed h

The law of large numbers ensures convergence and given finite variance $Var(h(X))$ we can also us the central limit theorem to approximate the error involved. According to the latter, the magnitude of approximation error will decrease at the rate $N^{-1/2}$, which is painfully slow. In comparison to many other numerical integration techniques, it does however have the advantage of not depending directly on dimensionality of X. Another advamtage is that f_X need not be analytically available as long as we can simulate from it.

As a basic example, we can approximate an integral on the unit interval using that $\int_0^1 h(x)\, dx = E(h(X))$ if X is uniformly distributed on $[0, 1]$. In R with $h(x) = \sin(x)$ we obtain

```r
N <- 10000
x <- runif(N)
h <- sin(x)
mean(h)
```

```
## [1] 0.4606767
```

which is reasonably close to the true value $1 - \cos(1) = 0.4596977...$. An alternative coding-approach in R that generalises more easier to the tasks later in this assignment is to use the function `replicate` as in

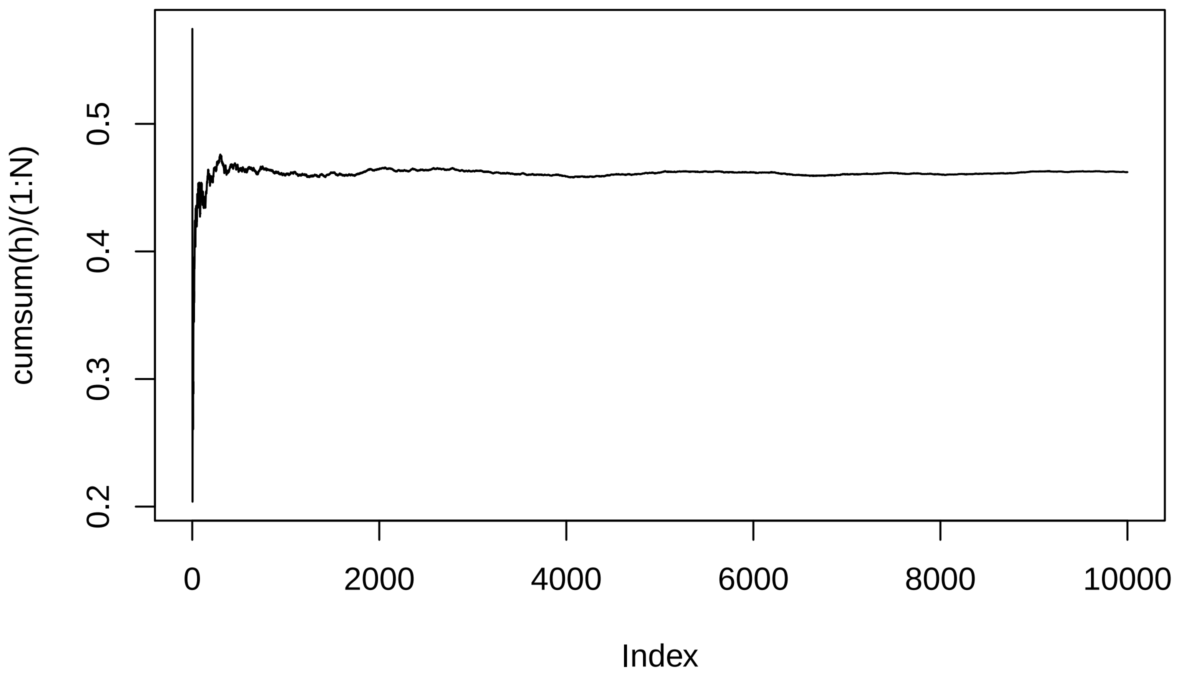
```r
h <- replicate(N,
               {
                   x <- runif(1)
                   sin(x)
               })
mean(h)
```

```
## [1] 0.4621594
```

the advantage being that it works also when the simulation function (`runif` in this case) does not have the option of returning multiple independent copies of x. Of course, the same thing could be done with a `for`-loop.

We can illustrate convergence by plotting cumulative means

```r
plot(cumsum(h) / (1:N), type = "l")
```



or give an approximate 95% confidence interval for the unknown value by

```r
mean(h) + qnorm(c(.025, .975)) * sd(h) / sqrt(N)
```

```
## [1] 0.4572691 0.4670498
```

Importance sampling

Importance sampling generalises crude Monte Carlo by writing

$$E(h(X)) = \int h(x) f_X(x)\,dx = \int h(x) f_X(x) \frac{f_Y(x)}{f_Y(x)}\,dx = E\Big(\frac{h(Y) f_X(Y)}{f_Y(Y)}\Big) = E(h(Y)w(Y)),$$

which is valid as long as $f_Y(x) > 0$ for all x such that $f_X(x) > 0$ (the property of *absolute continuity*). The expression is then approximated by simulating from Y and averaging simulated $h(y)w(y)$.

There are two main arguments for this construction:

- The random variable/vector Y may be easier to simulate than X,
- the variance $Var(h(Y)w(Y))$ may be much smaller than $Var(h(X))$, leading to a smaller Monte-Carlo error.

The name of the method refers to applications of the latter; say we want to approximate $P(X > a) = \int \mathbf{1}(x > a) f_X(x)\,dx$ for a value a far out in the tail of f_X. If we proceed by simulating $X_1, \ldots, X_N$ from f_X, and take the average of $\mathbf{1}(X_i > a)$, we will be spending most of our simulation budget on a region that is *unimportant*; in this case where the integrand $\mathbf{1}(x > a) f_X(x)$ equals zero. We would like to spend more effort on the *important* region $x > a$ in order to increase efficiency/decrease variance.

For illustration, we consider the problem of approximating the probability of a rare event $p = P(X > 10)$ when X has a standard exponential distribution. Of course, Monte Carlo is not needed here since $P(X >$

$10) = \exp(-10) = 0.000045\ldots$. First, the crude Monte-Carlo gives the following based on 10000 draws:

```
x <- rexp(N)
h <- (x > 10)
mean(h)
```

```
## [1] 0
```

Not a single draw was larger than 10 and the approsimation returned is 0. While this is "close" to the true value in an absolute sense, when approximating rare events the relative error is more relevant (i.e. we are concerned whether the probability is 10^{-5} or 10^{-10}, not just that it is close to zero). Hence we want

$$\frac{\hat{p}}{p} - 1$$

rather than $p - \hat{p}$ to be small.

Using importance sampling, we may replace the standard exponential with a distribution that is more likely to excess 10.

Task 1: Complete the weight-function `w` in the code below such that it implements an importance sampler with $Y \sim Exponential(1/10)$. Also compute the relative error.

```
y <- rexp(10000, rate = 1 / 10)
h <- (y > 10)
w <- ...
mean(h * w)
```

Task 2: An "optimal" importance sampling distribution for the above problem is $Y = X + 10$, a standard exponential shifted to the right. Try it. In what sense is it "optimal"?

Default of an insurance company

Consider an insurance company that starts with capital C at year $t = 0$. Each year, it gains $p > 0$ in premiums and loses $X \sim F$ independently of other years. Its capital at the end of year t is thus

$$Z_t = C + tp - \sum_{i=1}^{t} X_i.$$

What is the probability of default within the next T years? We can write the probability of default as

$$P(\min_{0<t\leq T} Z_t < 0) = E(\mathbf{1}(\min_{0<t\leq T} Z_t < 0)).$$

Approximating this probability can be done by crude Monte-Carlo as follows:

- Simulate $x = (x_1, \ldots, x_T)$.
- Compute $h(x) = \mathbf{1}(\min_{0<t\leq T} z_t < 0)$.
- Repeat the above steps a large number of times and return the average of the computed indicators h.

Task 3: Write a function

```
default <- function(x, C, p){
    # Takes a vector of simulated losses "x", a starting capital "C"
    # and a premium "p" and returns an indicator TRUE/FALSE of default
    T <- length(x)
    ...
}
```

that takes a length T vector of simulated losses $(x_1, \dots, x_T)$, a starting capital C, a premium p and returns an indicator of default. Cumulative sums can be computed with R´s `cumsum`.

If $T = 20$, $C = 15$, $p = 1$ and loss-distributions independent $LogNormal(0, 1/4)$, we can now compute 10000 indicators by

```
N <- 10000
h <- replicate(N,
               {
                   x <- exp(rnorm(20, mean = 0, sd = 1/2))
                   default(x, C = 15, p = 1)
               })
```

and approximate the probability of default by

```
mean(h)
```

```
## [1] 3e-04
```

Again this will be inefficient if the probability of default is very small. A simple importance sampling alternative is to replace $X_i \sim LogNormal(0, 1/4)$ by $Y_i \sim LogNormal(\mu, 1/4)$, where the aim is to increase the probability of default (but not make it too close to 1).

An importance sampling version of the solution is then

```
mu <- ...
hw <- replicate(N,
               {
                   y <- exp(rnorm(20, mean = mu, sd = 1/2))
                   default(y, C = 10, p = 1) * w(y, mu)
               })
}
mean(hw)
```

where `w` computes the importance sampling weigths $w(y)$.

Task 4: What is w? Write it in mathematical notation and complete the function below

```
w <- function(Y, mu){
    ...
}
```

Task 5: Use the above functions to improve on the approximation of probability of default by choosing a suitable μ.

Markov chain Monte Carlo

Simulating directly from a given distribution $X \sim P$ can be difficult, in particular when the dimension of X is high. It turns out to be relatively straightforward to simulate a Markov chain which has P as its stationary distribution though. Less straightforward is to do it in an efficient way! Again we want to approximate $E(h(X))$ based on random draws $X_1, \dots, X_N$. This time however

- $X_1, \dots, X_N$ form a Markov chain, hence they will in general not be independent
- X_i is only guaranteed to have the same distribution as X in the limit.

Due to dependence, variance of $\bar{h}$ is now

$$Var(n^{-1}\sum_{i=1}^{N} h(X_i)) = n^{-2}\sum_{i=1} Var(h(X_i)) + n^{-2}\sum_{i \neq j} Cov(h(X_i), h(X_j)),$$

where the first term will be approximately $n^{-1}Var(h(X_n))$ as for the independent case. For a Markov chain with strong serial dependence, the second term will dominate the Monte carlo error/variance.

Gibbs sampling a multivariate normal

Let (X_1, X_2) follow a bivariate Normal distribution with mean $\mu = (0, 0)$ and covariance matrix

$$\Sigma = \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix}$$

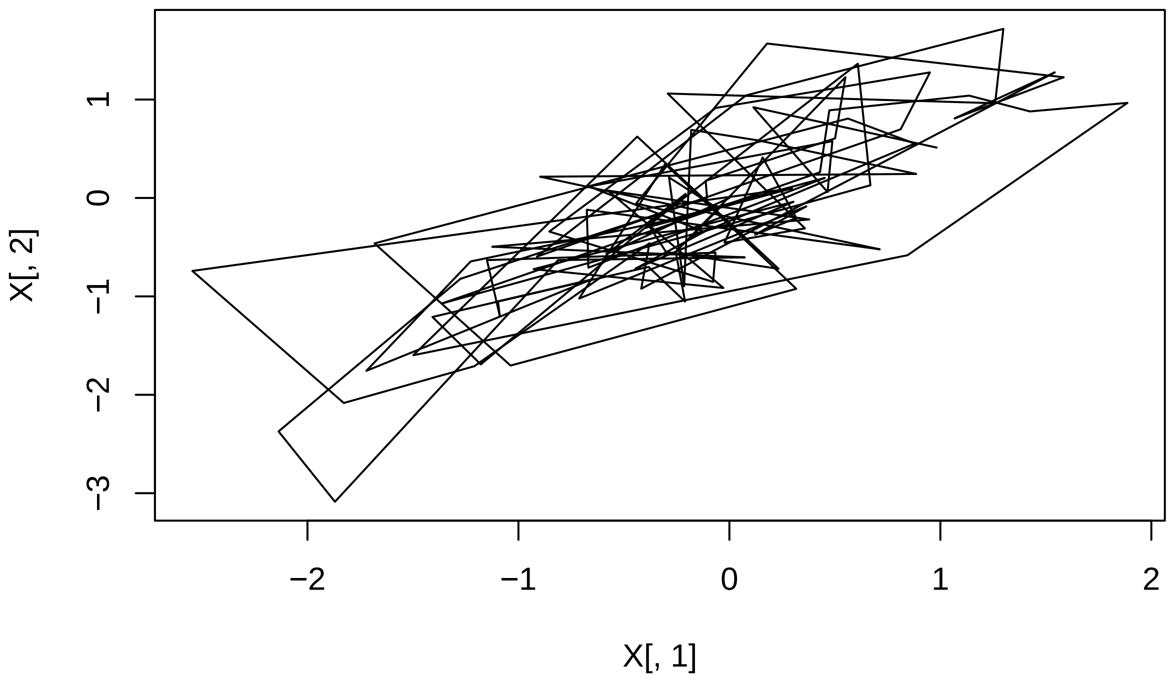
Task 6: Derive the conditional distributions $X_1|X_2 = x_2$ and $X_2|X_1 = x_1$.

Task 7: The following function implements a Gibbs-sampler according to the above, complete it by filling in the ...

```r
gibbs_bivn <- function(N, rho, start = c(0,0)){
    # Takes number of iterations "N", correlation "rho",
    # and a starting value "start". Returns a N by 2 matrix of
    # values Gibbs-sampled from a multivariate normal
    X <- matrix(ncol = 2, nrow = N)
    X[1, ] <- start
    for (i in 2:N){
        X[i, 1] <- ...
        X[i, 2] <- ...
    }
    return(X)
}
```

We may try it out as follows

```r
X <- gibbs_bivn(N = 100, rho = 0.8)
plot(X[, 1], X[,2], type = "l")
```



Task 8: Try the above for different values of $\rho \in (-1, 1)$ (some really close to 1 or -1). How does the Monte Carlo variance $Var(\bar{X}_1)$ approximated by e.g. (fill in `rho` in place of ...)

```r
m1hat <- replicate(100,
                   {
                       X <- gibbs_bivn(N = 1000, rho = ..., start = rnorm(1))
                       mean(X[, 1])
                   }
)
var(m1hat)
```

depend on ρ? Conclusion?

The ising model

The square lattice ising model is a classical model in statistical physics describing interacting magnetic spins. It defines a probability distribution on a $k \times k$ matrix σ, with entries $\sigma_i \in \{-1, 1\}$, $i \in J_k = \{1, \dots, k\}^2$, defined by

$$\pi_\sigma = C_\beta \exp\big(-\beta \sum_{i \sim j} \sigma_i \sigma_j\big)$$

where the sum is over all pairs of coordinates $i \in J_k$ and $j \in J_k$ that are *neighbours*. We here say that $i = (i_1, i_2)$ and $j = (j_1, j_2)$ are neighbours if $|i_1 - j_1| + |i_2 - j_2| = 1$. It can be shown that

$$p(\sigma_i = 1 | \{\sigma_j ; j \neq i\}) = 1/(1 + \exp(2\beta \sum_{j : i \sim j} \sigma_j)),$$

note that this does not depend on the hard-to-compute normalising constant C_β.

Task 9: The function `gibbs_ising` below simulates am Ising model using a Gibbs-sampler, complete the function by writing the helper-function `sim_sigma` that simulates the value of `sigma` at position `(i1, i2)` conditionally on its values at other positions. R's `sample` is useful for simulating from discrete distributions.

```r
gibbs_ising <- function(N, k, beta,
                        start = matrix(sample(c(-1, 1), k * k, replace = TRUE),
                                       ncol = k, nrow = k)){
    # Takes number of iterations "N", grid-size "k", parameter "beta"
    # and a starting grid "start". Returns a list of matrices simulated
    # by Gibbs-sampling the Ising model.
    sigma_list <- lapply(rep(NA, N), matrix, ncol = k, nrow = k)
    sigma_list[[1]] <- start
    sigma <- start
    for (j in 2:N){
        for (i1 in 1:k){
            for (i2 in 1:k){
                sigma[i1, i2] <- sim_sigma(i1, i2, sigma, beta)
            }
        }
        sigma_list[[j]] <- sigma
    }
    sigma_list
}
```

As a helper you may use the function `sum_neigh` below that computes $\sum_{j:i\sim j}\sigma_j$ given $i=(i_1,i_2)$ and σ.

```r
sum_neigh <- function(i1, i2, sigma){
    # Takes a row number "i1", column number"i2",
    # matrix "sigma" and computes the sum of neighbour values
    sigma_pad <- rbind(0, cbind(0, sigma, 0), 0)
    sum(c(sigma_pad[i1 + 1, i2], sigma_pad[i1 + 1, i2 + 2],
          sigma_pad[i1, i2 + 1], sigma_pad[i1 + 2, i2 + 1]))
}
```

Note that `gibbs_ising` is computationally demanding for large values of k, try it with a small value (e.g. $k = 10$) first and then increase if computer power allows. A nice way to visualise the output is through an animated gif, which can be constructed with the `animation` package:

```r
library(animation)
saveGIF(for (i in seq_along(sigma_list)) image(sigma_list[[i]], axes = FALSE))
```

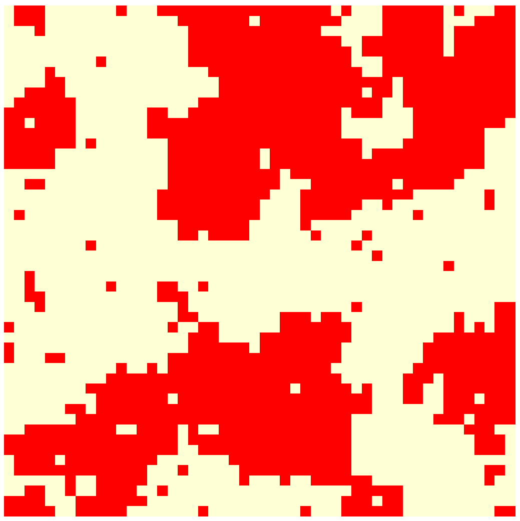
where `sigma_list` is the output from `gibbs_ising`. The gif is saved as `animation.gif` by default.

Task 10: Simulate and visualise the Ising model for positive and negative values of β and report the result.

Task 11: Note that by symmetry, $P(\sigma_i = 1) = P(\sigma_i = -1) = 1/2$. Simulate a long chain (with a modest k) and $\beta = 10$. What is the proportion of values of, say, $\sigma_{(2,2)}$ that equals 1? Conclusion?

Sample run:

```r
sigma_list <- gibbs_ising(100, 50, -1)
image(sigma_list[[50]], axes = FALSE, asp = 1)
```



A single server queue

A single-server queuing system is completely determined by a set of arrival times $(a_1, \ldots, a_N)$ and a set of service times $(s_1, \ldots, s_2)$. In order to compute $N(t)$, the number of customers in the queue at time t, it is convenient to first compute the exit times. This can be done recursively as follows:

- The first customer exits at time $e_1 = a_1 + s_1$.
- The second customer exits at time $e_2 = \max(a_2, e_1) + s_2$.
- …
- The N:th customer exits at time $e_N = \max(a_N, e_{N-1}) + s_N$.

that is, if the queue is empty when customer k arrives he/she exits at s_k time-units after arrival. If there are people in the system, he/she exits at s_k time-units after the previous customer.

Task 12: Write a function

```
get_exits <- function(arrival_times, service_times){
    # Takes vectors "arrival_times" and "service_times"
    # and returns the corresponding vector of exit times
    ...
}
```

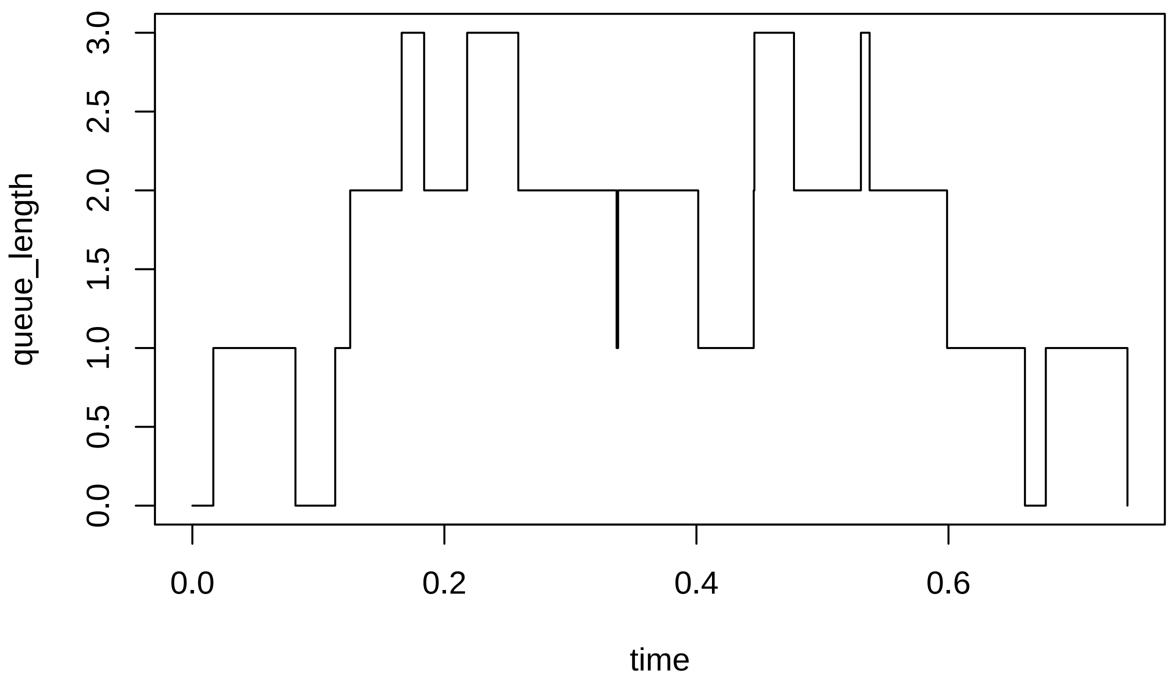
that returns a vector of exit-times given arrival and service times.

The `get_queue` function below uses `get_exits` to compute the queue-length at its change-points:

```
get_queue <- function(arrival_times, service_times, n0 = 0){
    # Takes vectors "arrival_times" and "service_times" and
    # queue length at time "n0" at time 0. Returns a data-frame
    # with event times (including time 0) together with queue length
    N <- length(arrival_times)
    exit_times <- get_exits(arrival_times, service_times)
    times <- c(0, arrival_times, exit_times)
    event_order <- order(times)
    changes <- c(n0, rep(1, N), rep(-1, N))[event_order]
    data.frame(time = sort(times), queue_length = cumsum(changes))
}
```

We can try it out as follows:

```
N <- rpois(1, lambda = 10)
arrival_times <- sort(runif(N))
service_times <- rexp(N, rate = 50) + 0.05
queue <- get_queue(arrival_times, service_times)
with(queue, plot(time, queue_length, type = "s"))
```



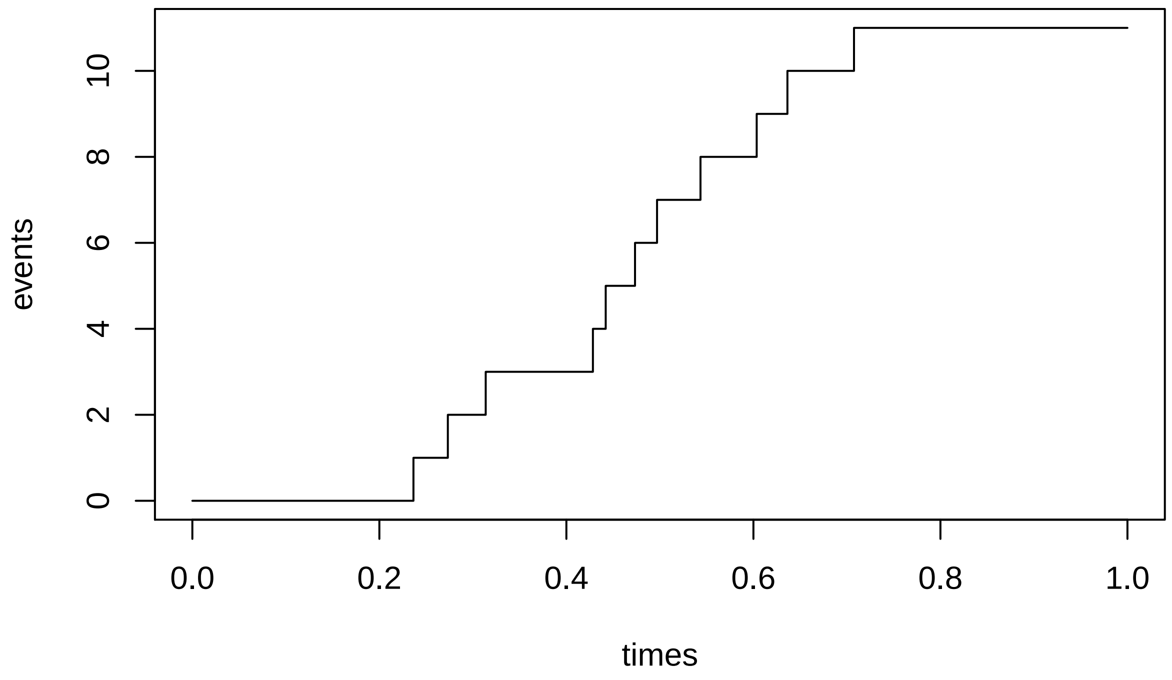
which plots a queuing process with Poisson arrivals and service times being exponential plus an offset of 0.05.

A queuing process with non-homogenous Poisson arrivals

The homogenous Poisson process used for arrivals above is easily generalised to a non-homogenous one by replacing the uniformly distributed arrival times with e.g. a Beta distribution.

Task 13: What is the intensity function of the non-homogenous Poisson process on the unit interval simulated below?

```r
N <- rpois(1, lambda = 10)
arrival_times <- sort(rbeta(N, 5, 5))
times <- c(0, arrival_times, 1)
events <- c(0, 1:N, N)
plot(times, events, type = "s", xlim = c(0, 1))
```



Task 14: Write a function that simulates a queuing system with non-homogenous Poisson arrivals (your choice!) and non-exponential service times (your choice, make them positive though!) and returns the maximum observed queue-length. Run the function 1000 times using `replicate` (or a for-loop) and visualise the maximum queue-length distribution with a histogram.