Rejection sampling

Computational Statistics

University of Copenhagen

September 25th 2019

Introduction

Sample from the probability distribution on $[0; \infty[$ with a density

$$f(y) \propto \prod_{i=1}^{100} \exp(y z_i x_i - e^{y x_i}), \ y \geq 0$$

Find a Gaussian envelope of f and implement rejection sampling from the distribution with density f using this envelope.

Implement the adaptive rejection sampling algorithm that uses a piecewise log-affine envelope and compare it with the one based on the Gaussian envelope

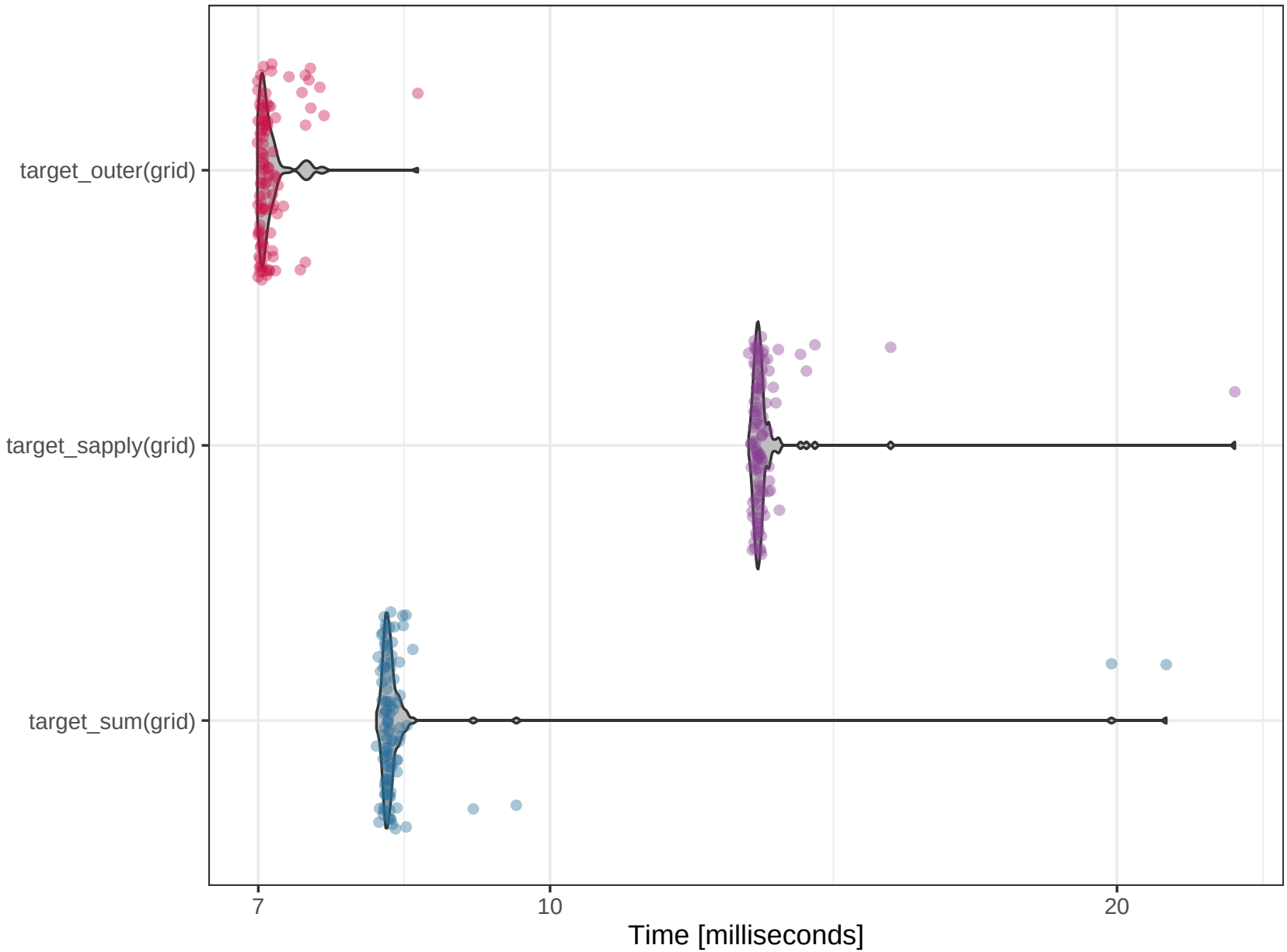
Target density and implementation

$$f(y) \propto \prod_{i=1}^{100} \exp(yz_i x_i - e^{yx_i}),\ y \geq 0$$

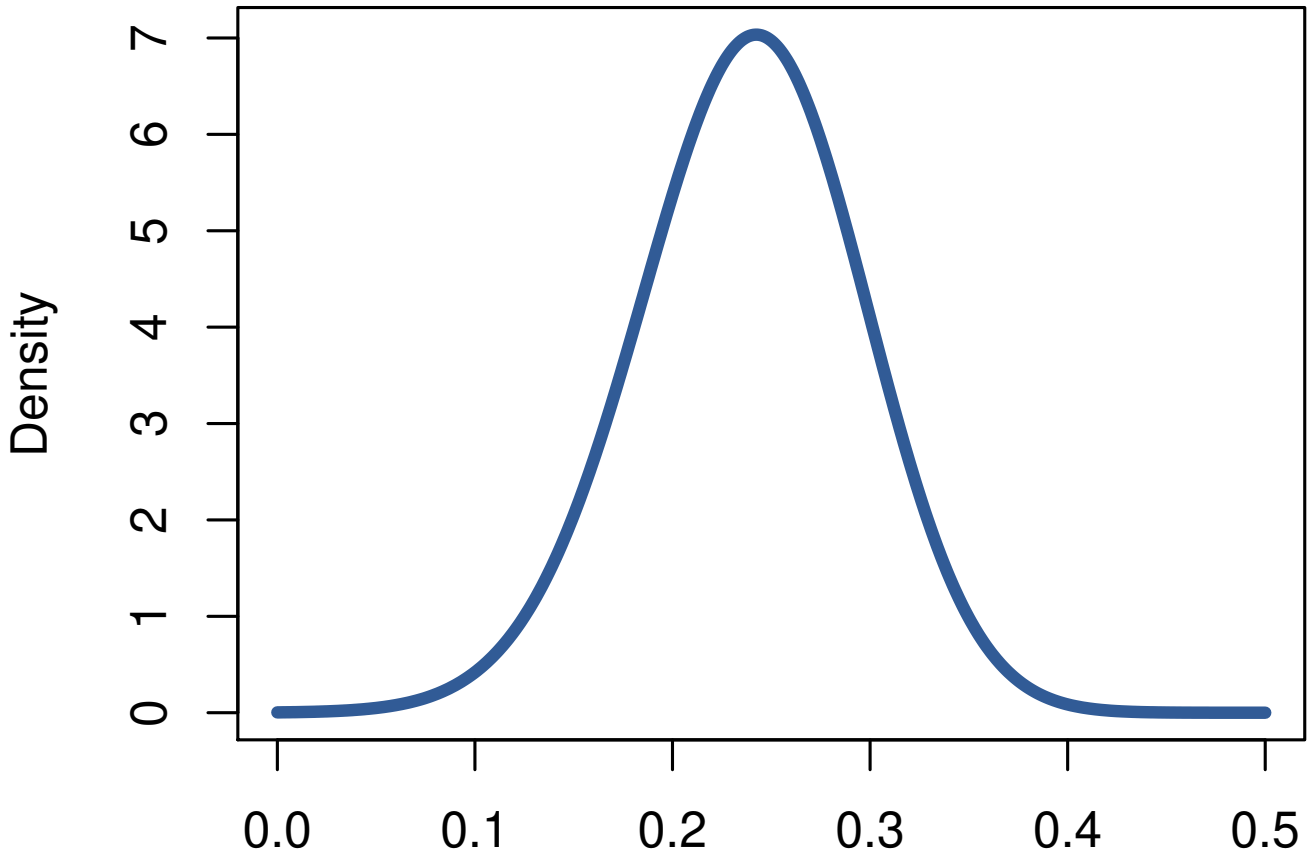
```
target_sapply <- function(y)
{
    sapply(y, function(yy) prod(exp(yy*zx - exp(x*yy)))) / 1.05009e-41
}


target_sum <- function(y)
{
    sapply(y, function(yy) exp( sum(yy*zx - exp(x*yy)) )) / 1.05009e-41
}


target_outer <- function(y)
{
    exp( rowSums(outer(y, zx) - exp(outer(y, x))))/ 1.05009e-41

}
```

Note that the normalization constant is approximated via *integrate*(), so the $\propto$-notation still applies. It does however allow us to later interpret α as being approximately the true probability of accept.

Comparisons

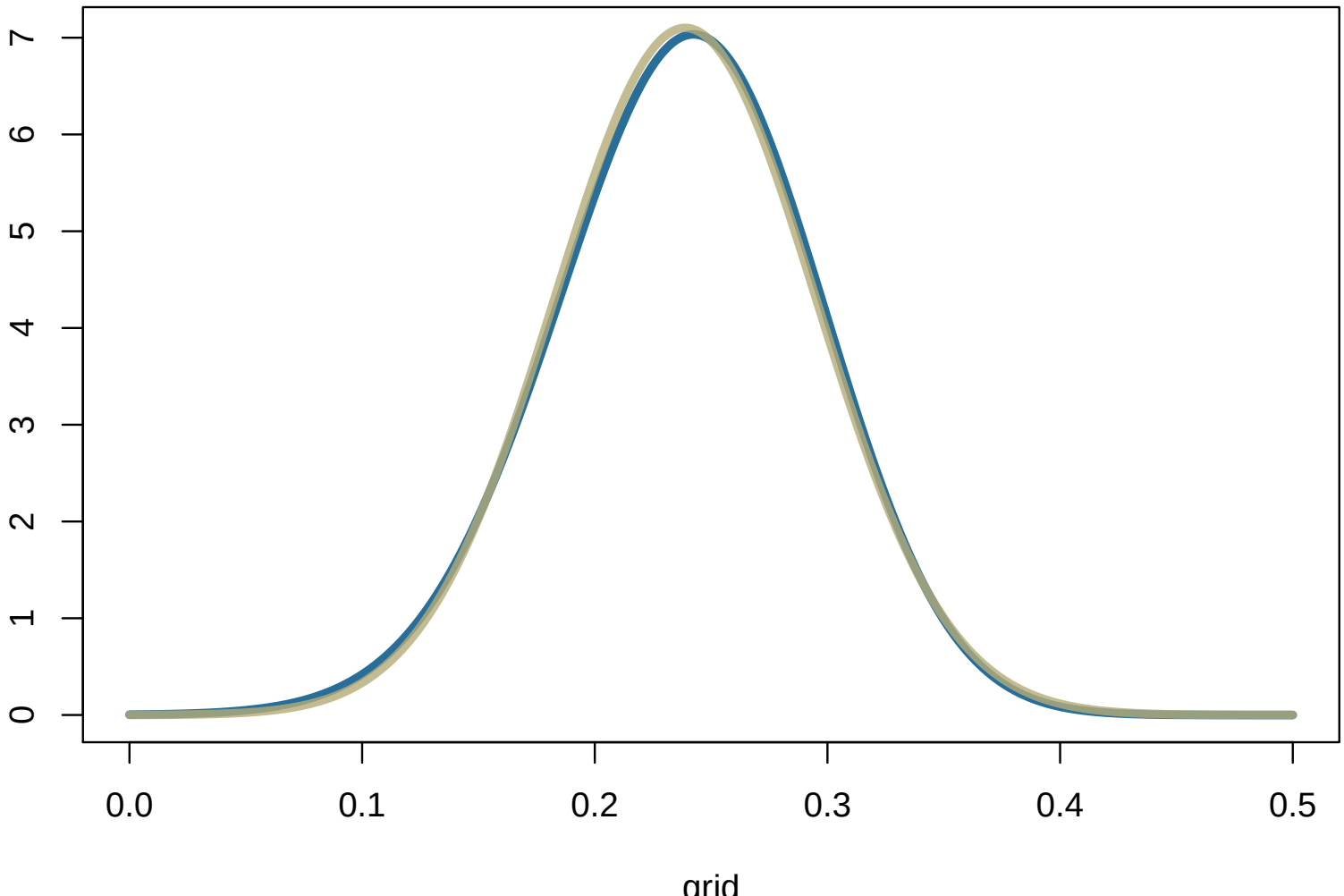


Target density



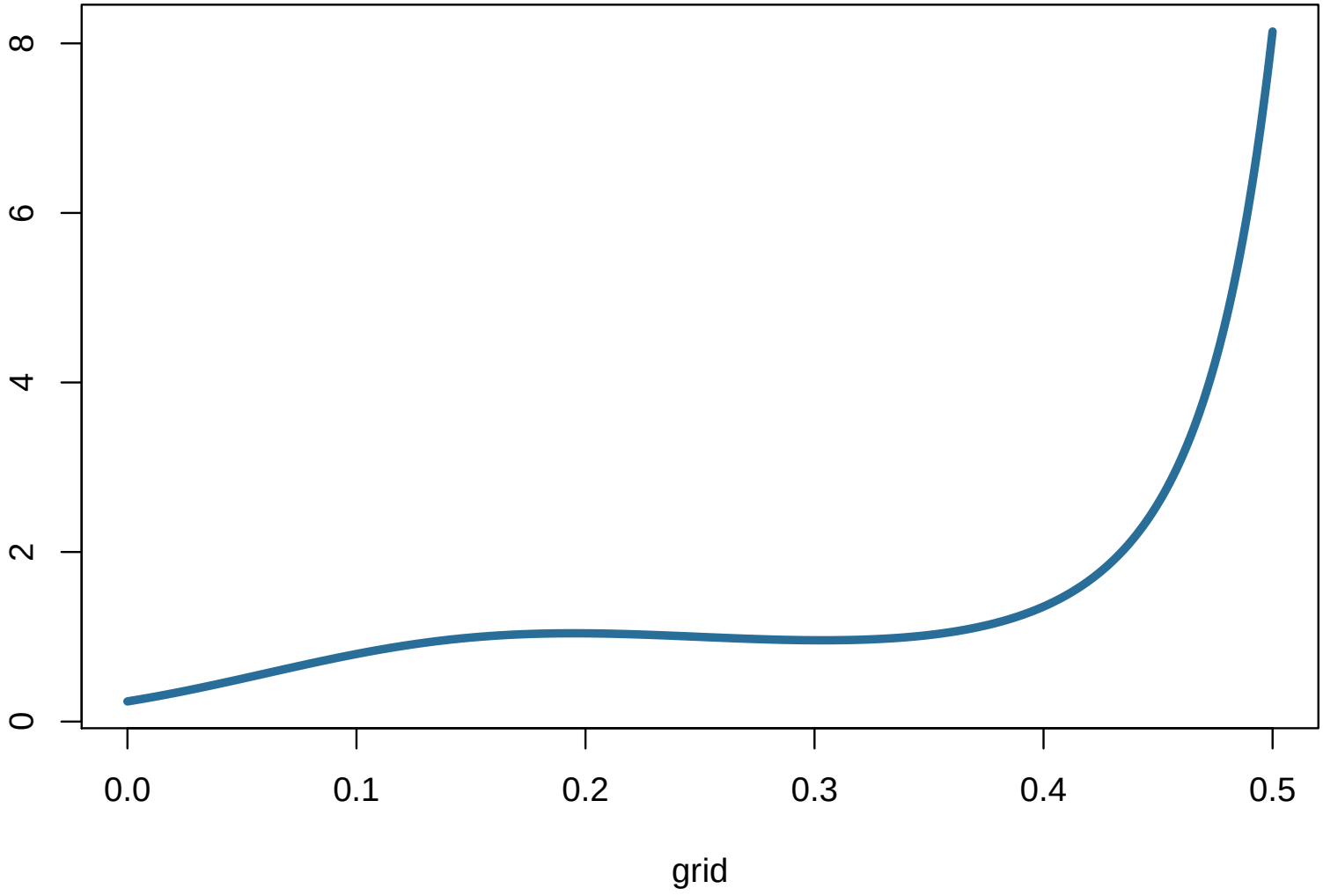
Gaussian envelope

```
mean <- integrate(function(v) target_outer(v)*v, 0, Inf)$value
sd <- sqrt(integrate(function(v) target_outer(v)*v^2, 0, Inf)$value - mean^2)
```

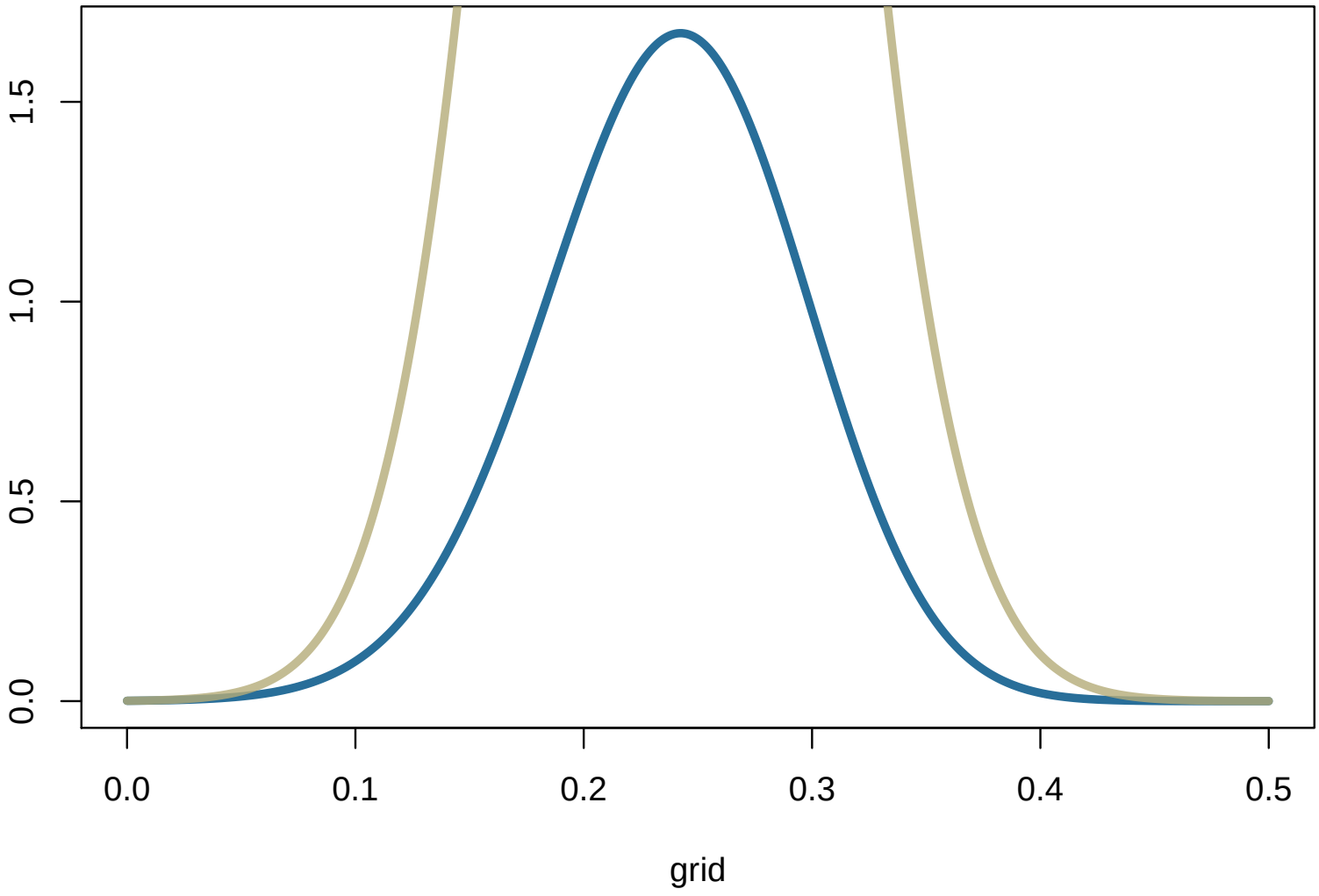


Ratio plot

The α-parameter was calculated to be approximately 0.237.

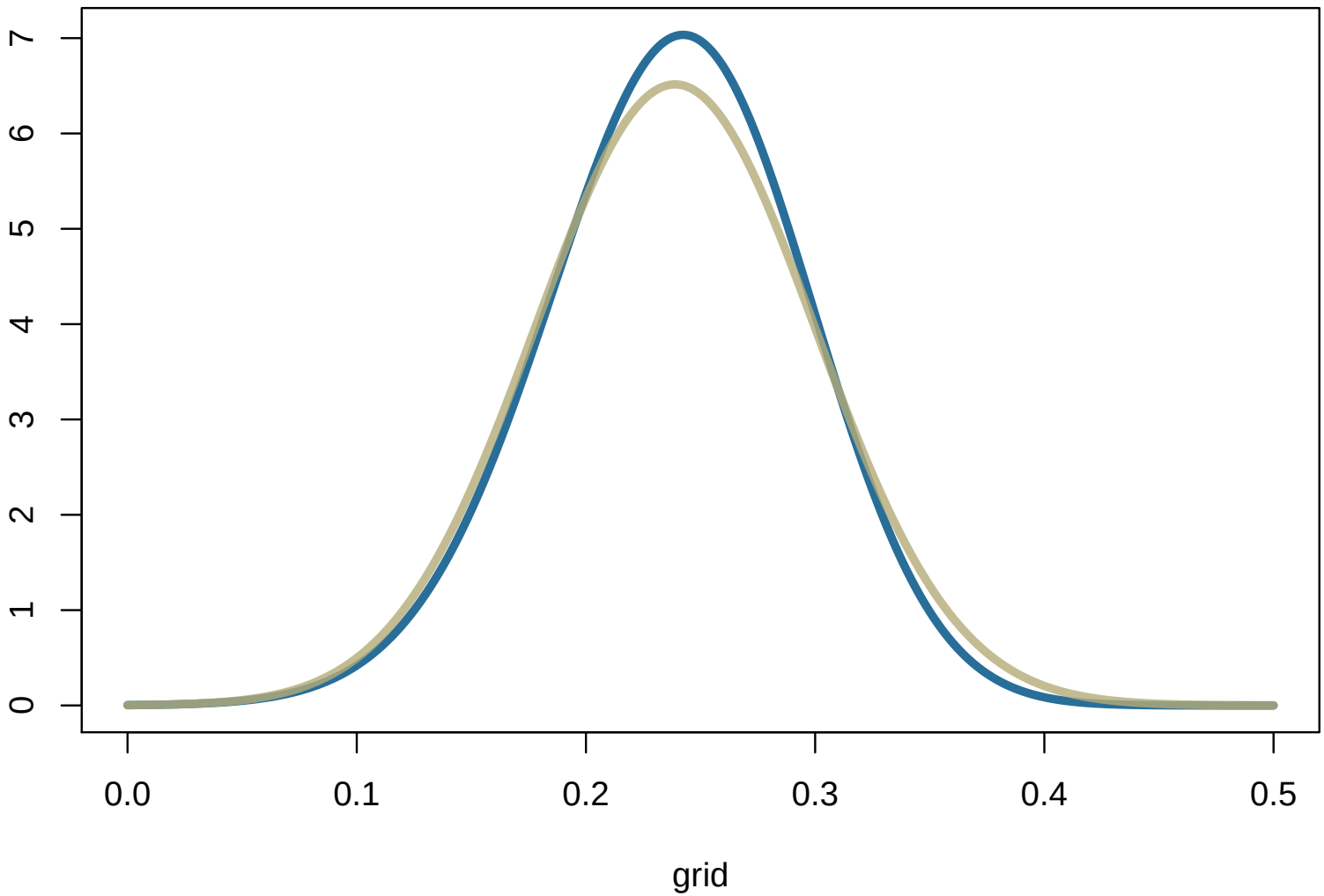


The envelope



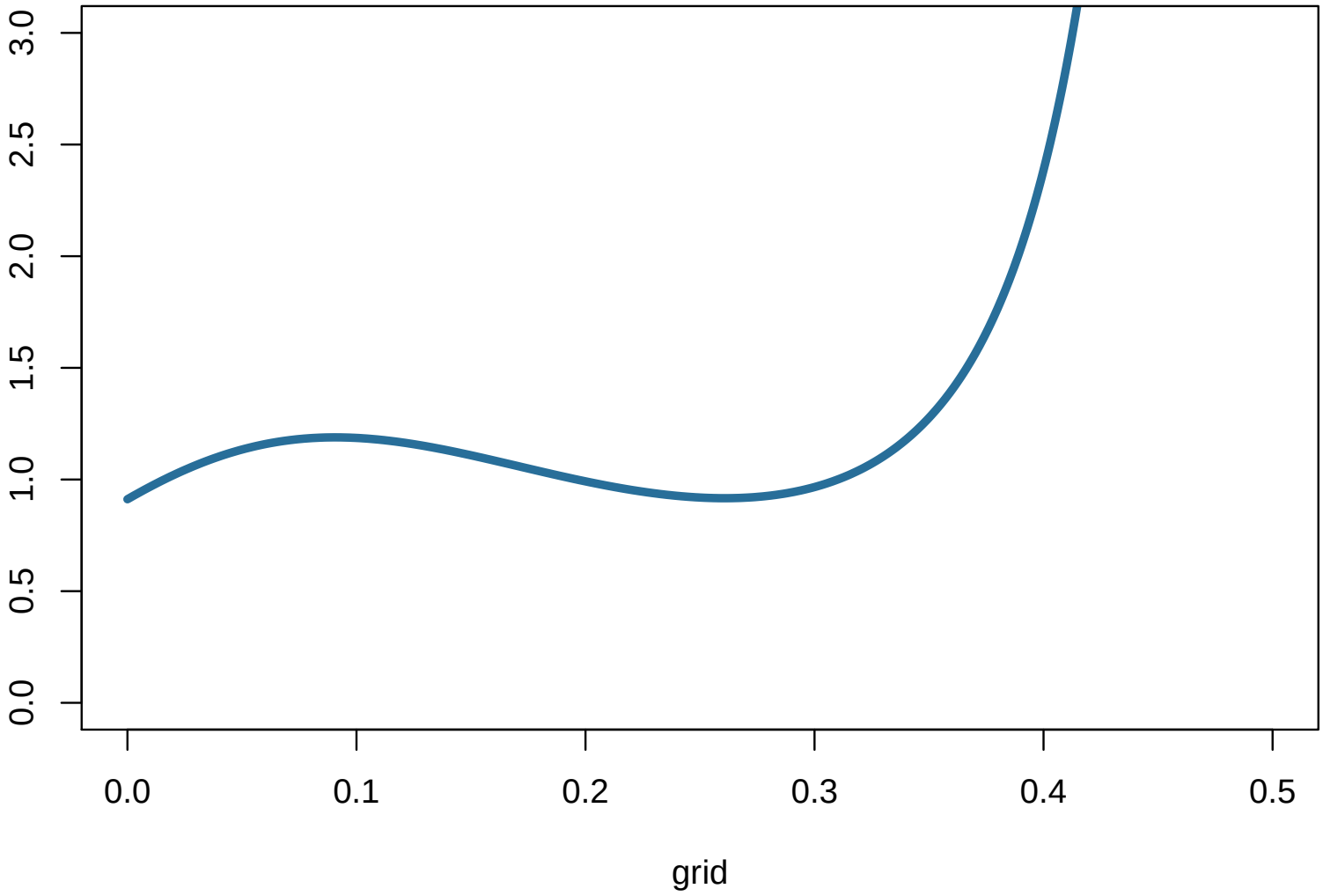
Improved envelope

Clearly, our proposal needs to have a higher standard deviation. By trial and error, we increase the sd by 9 percent.

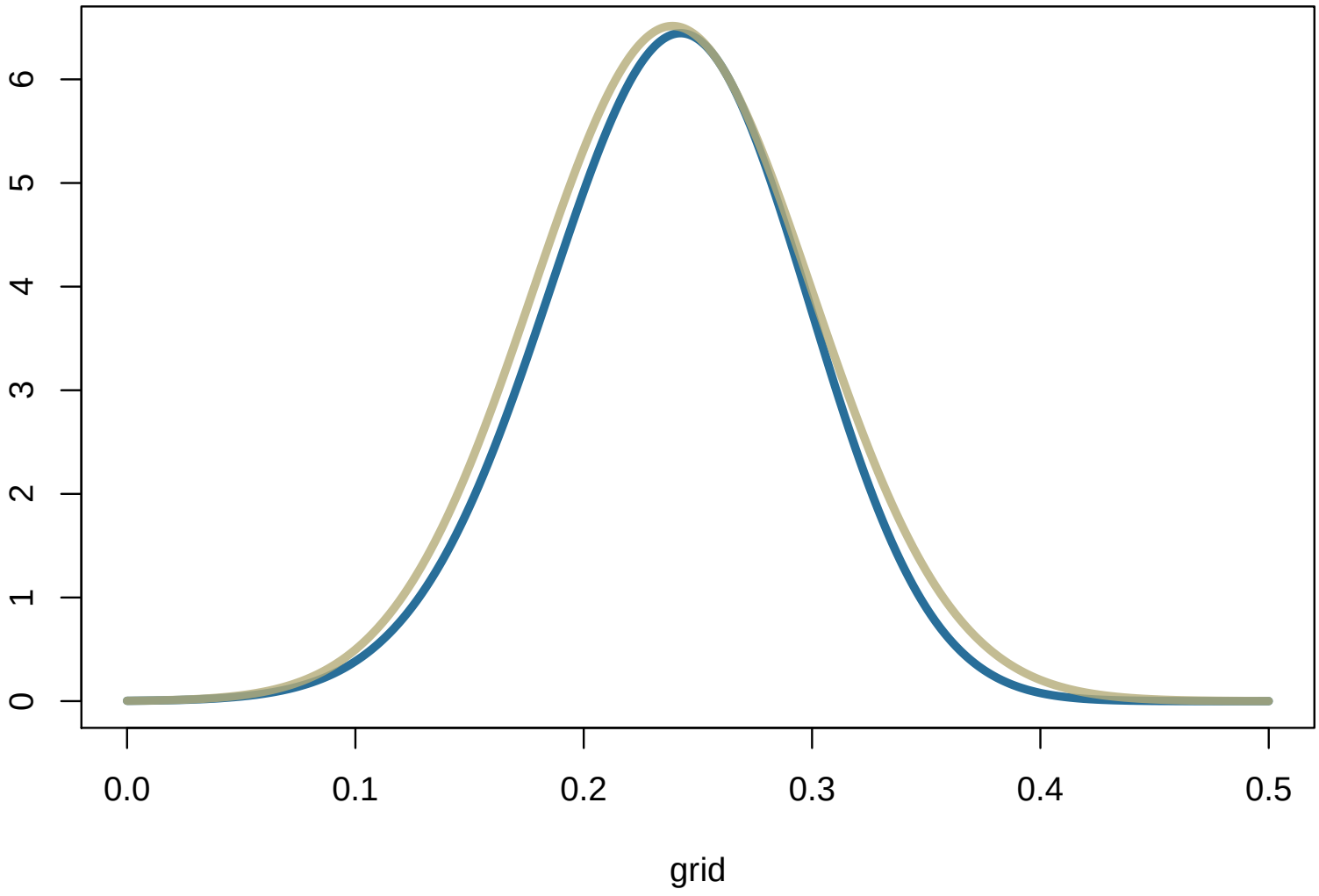


Improved envelope

The new α was calculated to be 0.9163.

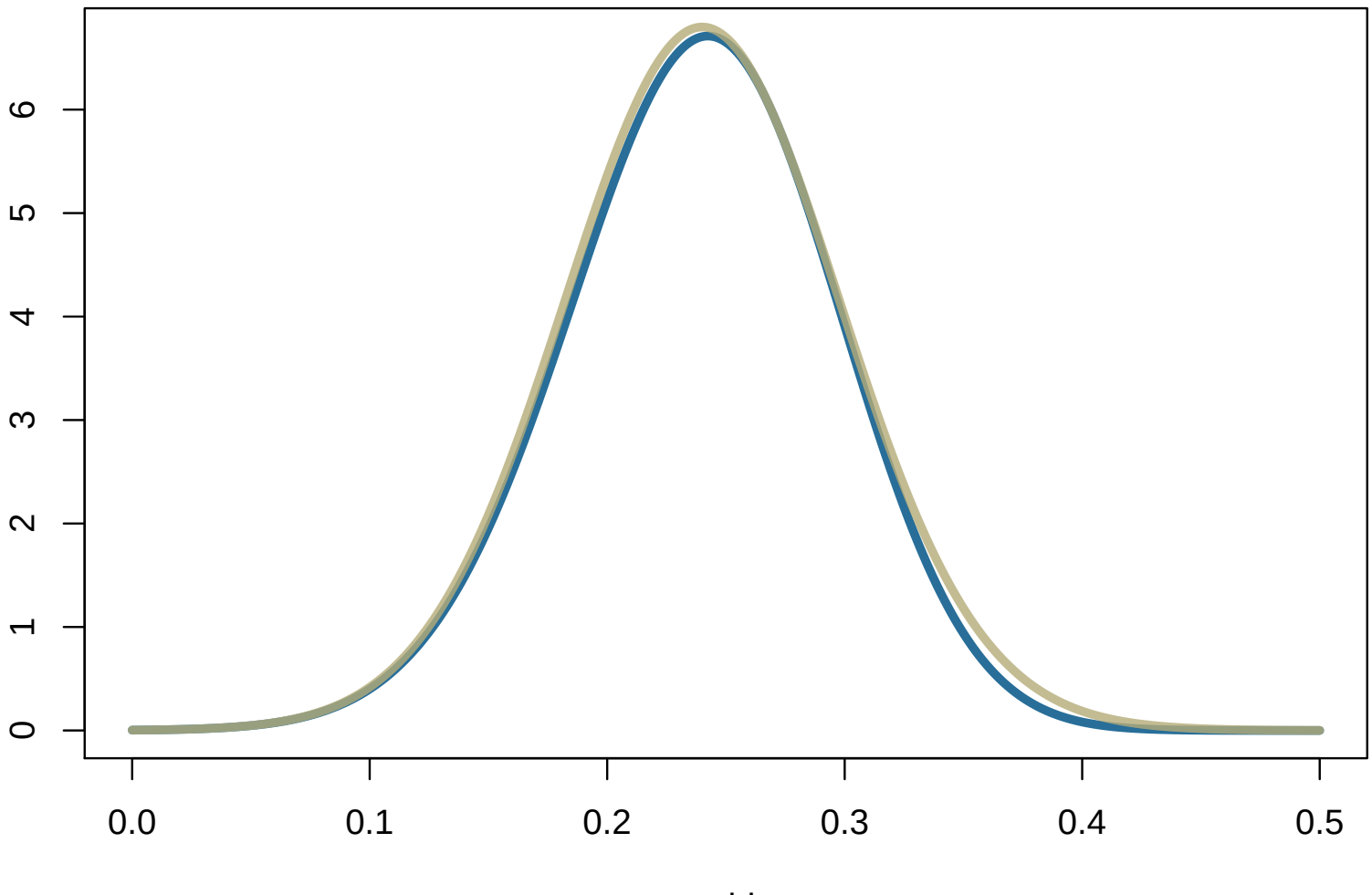


Improved envelope



A (generalized) t-distributed envelope

The α here is 0.954. $\mu = 0.24$, $\sigma = 0.058$, $df = 61$.
We found it by simulating the density from the Gaussian envelope, then fitting a t-distribution on the results.

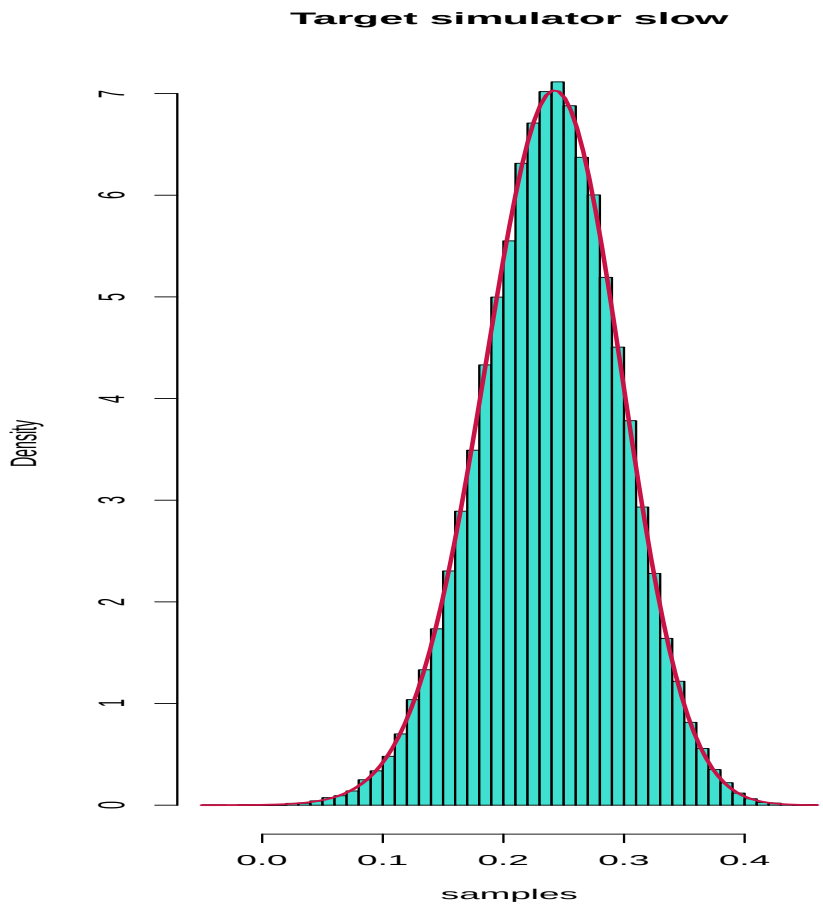


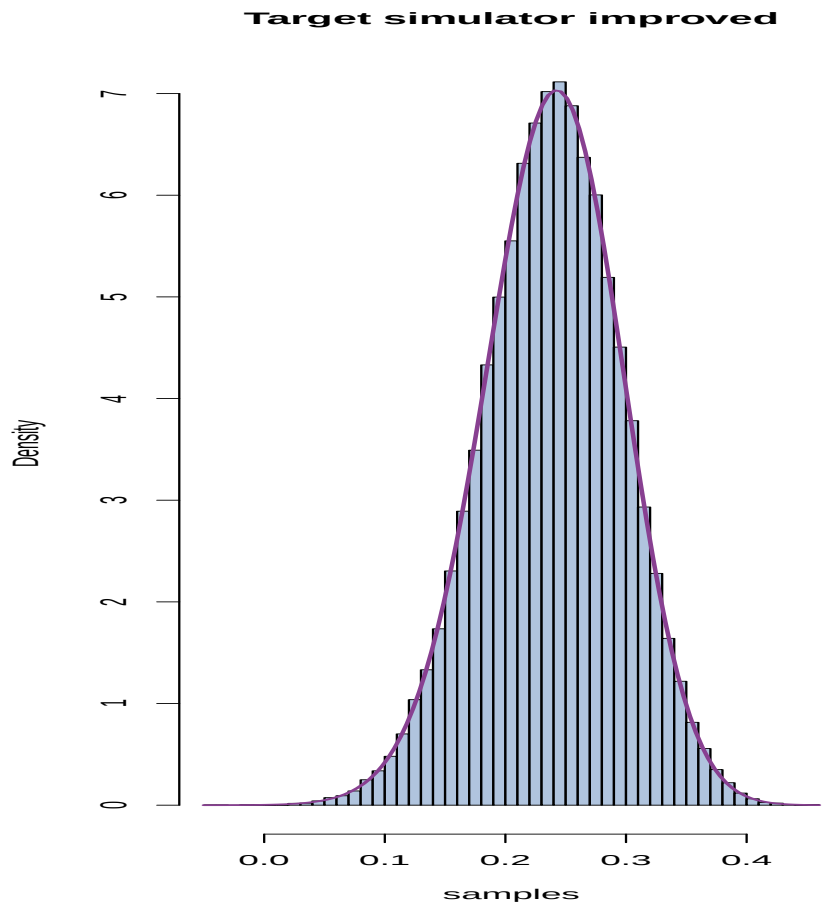
Rejection sampling algorithm slow

```
#We encapsulate all objects needed in a list.
#We will re-use this format later for the adaptive envelope generator
#this allows for easy extension to OO-programming


proposal_factory <- function(target_dens, proposal_dens, proposal_sim, a)
{
   list( tdens = target_dens,
         pdens = proposal_dens,
         sim = proposal_sim,
         alpha = a)}

target_simulator_slow <- function(n, proposal_object) {
  y <- numeric(n)
  for(i in 1:n) {
    reject <- TRUE
    while(reject) {
      y0 <- proposal_object$sim(1)
      u <- runif(1)
      reject <- u > proposal_object$a*proposal_object$tdens(y0) /
            proposal_object$pdens(y0)
    }
    y[i] <- y0
  }
  y}
```

Rejection sampling algorithm improved

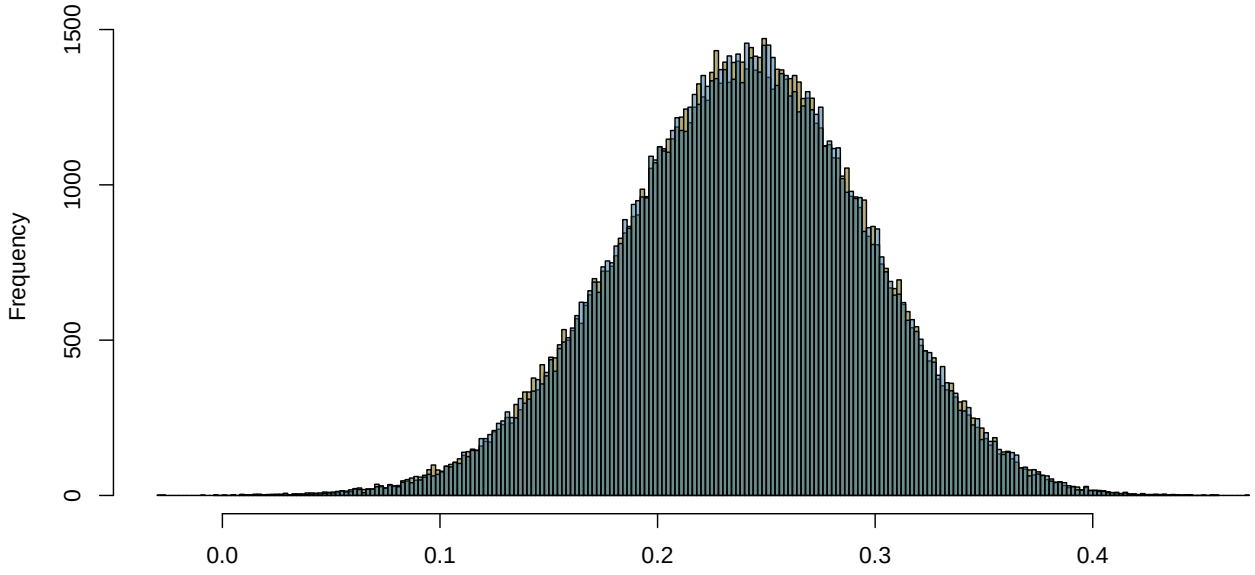
```
target_simulator <- function(n, proposal_object, scale = 1 )
{
    simulated_values <- numeric(n)  #we need n samples

    num_accepted <- 0 #we keep track of how many we have

    while (num_accepted < n)
    {
     samples <- ceiling( scale*(n - num_accepted)/proposal_object$a )
     #sample expected amount but possibly scaled up by a factor

     prop_samples <- proposal_object$sim(samples)  #get all the samples
     uniform_samples <- runif(samples)

     accept <- uniform_samples <= proposal_object$a*proposal_object$tdens(prop_samples) /
        proposal_object$pdens(prop_samples)

     new_additions <- min(n - num_accepted, sum(accept))

     simulated_values[(num_accepted + 1):(num_accepted + new_additions)]
        <- (prop_samples[accept])[1:new_additions]

     num_accepted <- num_accepted + new_additions

    }

    simulated_values
}
```


Target simulator slow
Density
0
1
2
3
4
5
6
7
0.0
0.1
0.2
0.3
0.4
samples


Target simulator improved
Density
0
1
2
3
4
5
6
7
0.0
0.1
0.2
0.3
0.4
samples

Comparing the Gaussian and t-distributed envelopes

```
Unit: milliseconds
                                 expr      min       lq     mean   median       uq
 target_simulator(10000, propn, 1) 165.3658 167.6069 180.1751 172.6749 187.1796
 target_simulator(10000, propt, 1) 164.4175 165.5598 176.6923 171.1025 180.7648
      max neval
 273.6954   100
 268.6328   100
```



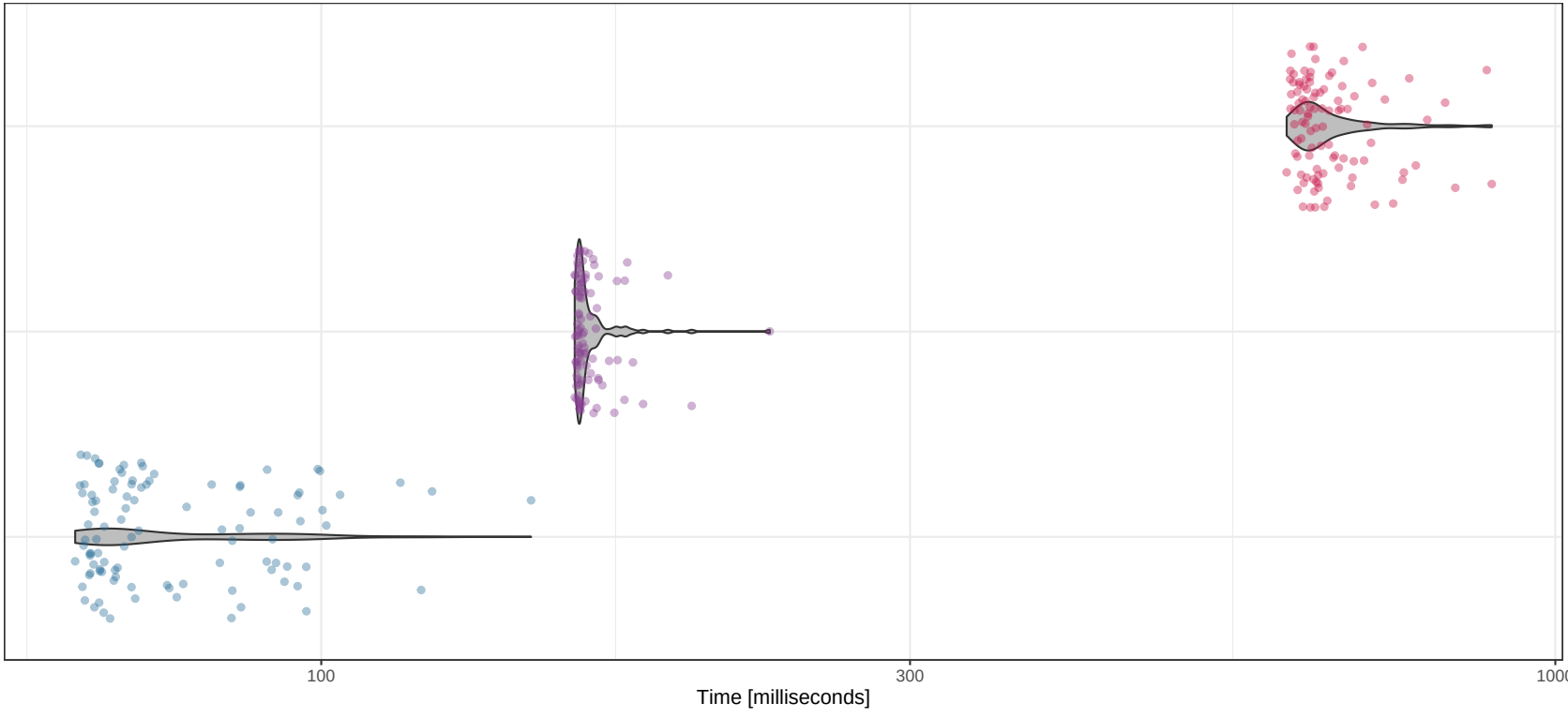
The benefit of a higher accept probability is largely cancelled out by the slower simulation and evaluation of the t-envelope.

Main implementation with parallelization

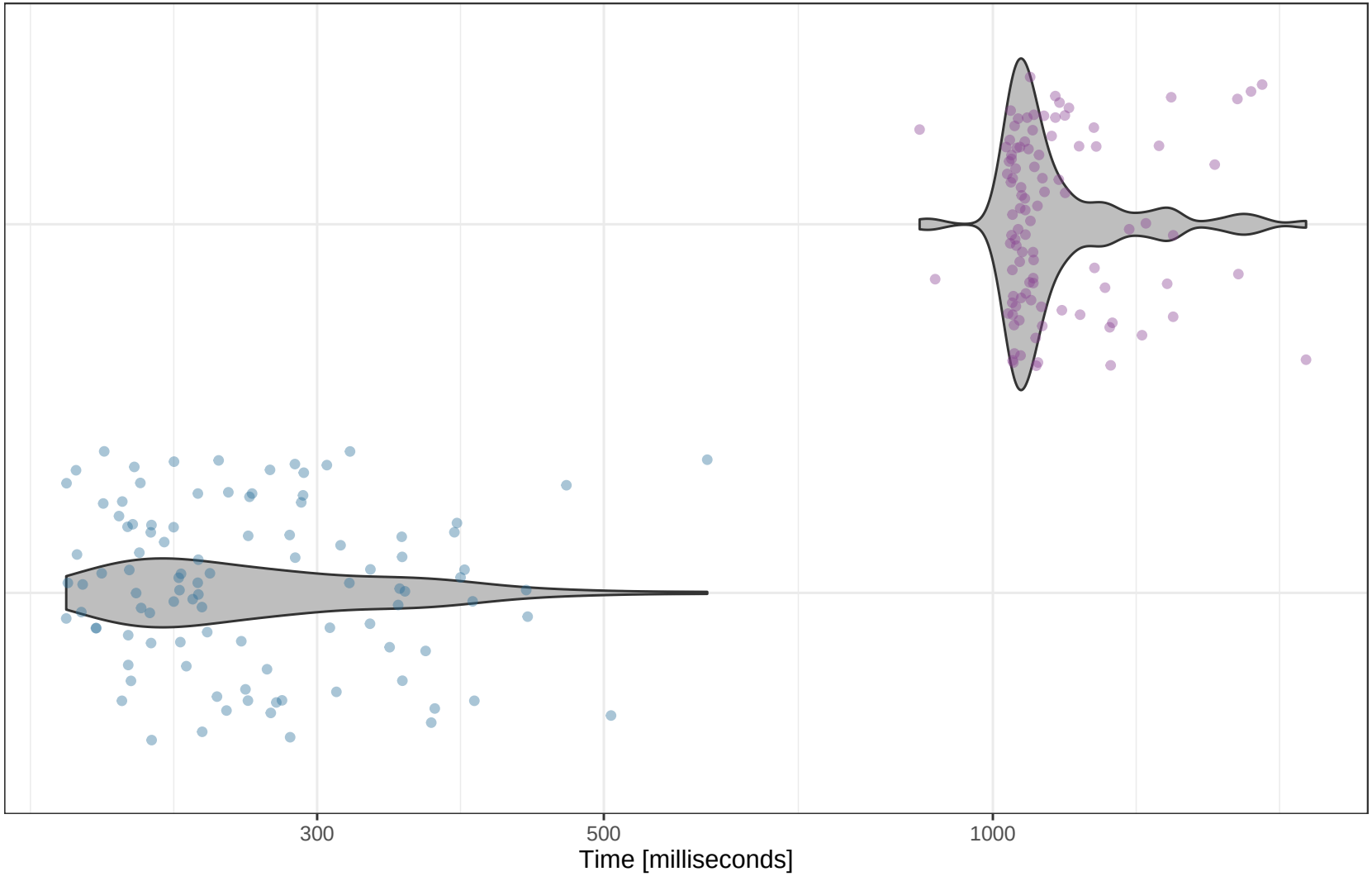
A makeshift example, using the doParallel library:

```
foreach(i = rep(10000/8, 8),  .combine = 'c') %dopar%
{
    propn <- proposal_factory(target_outer, function(x) dnorm(x, mean, sd*1.09),
        function(x) rnorm(x, mean, sd*1.09), alpha)
    target_simulator(i, propn, 1)
    }
```

Comparing the implementations, $n = 10.000$



Comparing the implementations, $n = 50.000$

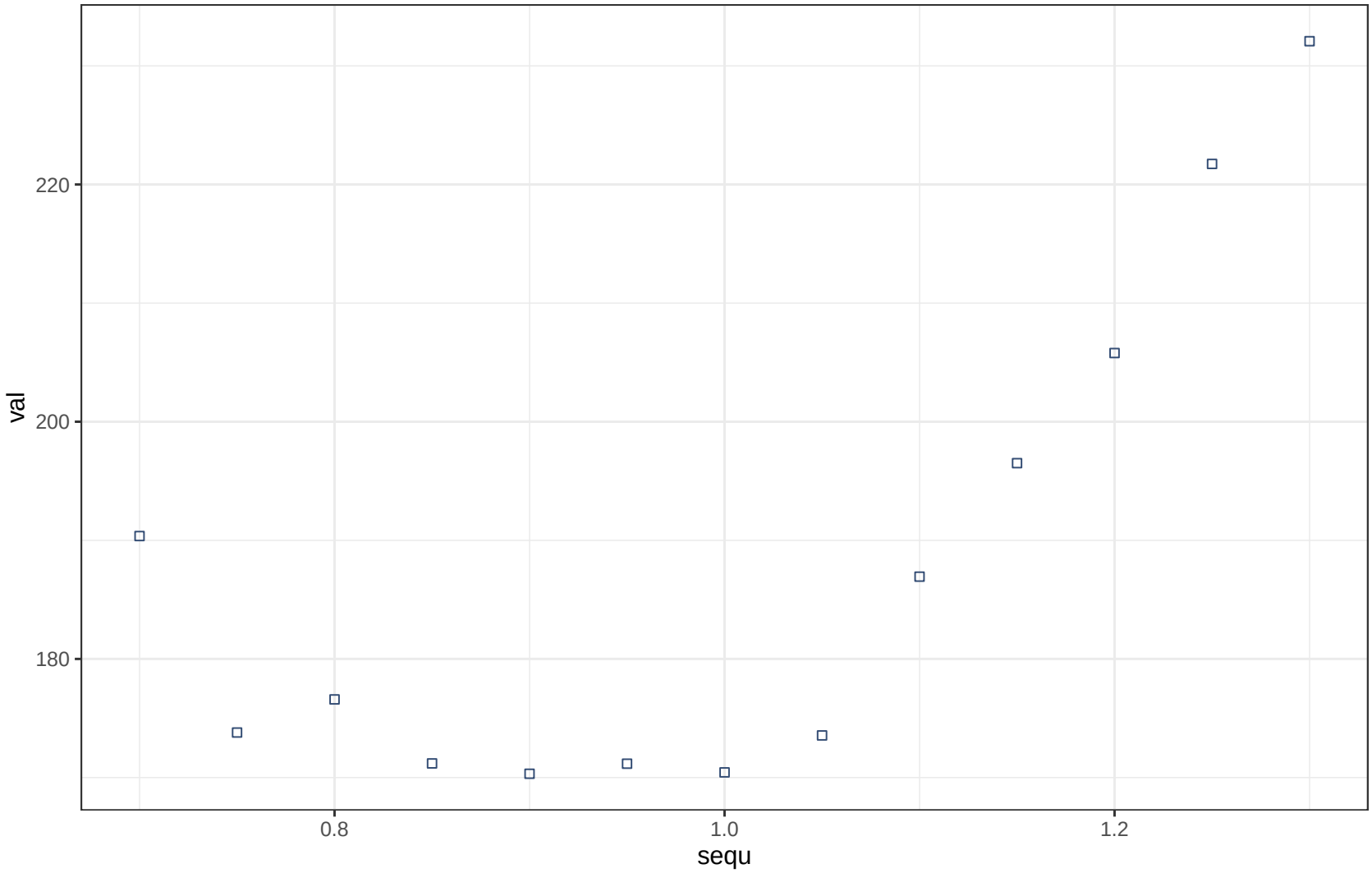


Can Rcpp speed things up?

```cpp
// [[Rcpp::export]]
NumericVector target_simulator_cpp(int n, List prop_object, double scale)
{
    Function target = as<Function>(prop_object["tdens"]);
    Function proposal = as<Function>(prop_object["pdens"]);
    Function sim = as<Function>(prop_object["sim"]);
    double alpha = as<double>(prop_object["alpha"]);

    NumericVector simulated_values(n);

    int num_accepted = 0;
    while (num_accepted < n)
    {
        int samples = ceil(scale*(double(n - num_accepted)) / alpha);
        NumericVector prop_samples = sim(samples);
        NumericVector uniform_samples = runif(samples);

        NumericVector fx = target(prop_samples);
        NumericVector gx = proposal(prop_samples);

        for (int i = 0; i < samples; i++)
        {
            if (alpha*fx[i] / gx[i] >= uniform_samples[i])
            {
                simulated_values[num_accepted] = prop_samples[i];
                num_accepted += 1;
                if (num_accepted >= n)
                {
                    goto exit;
                }
            }
        }
    }
exit:
    return simulated_values;
}
```

```
Unit: milliseconds
                                  expr      min       lq     mean   median       uq      max neval
     target_simulator(5000, propn, 1) 82.96958 83.35984 87.55625 83.86785 93.26021  99.72847   100
 target_simulator_cpp(5000, propn, 1) 82.67603 83.09416 89.41752 83.52766 89.09366 222.22728   100
```

Comparing the main implementation for different values of $scale$, $n = 10000$



Profiling ($n = 100.000$, Gaussian envelope)

```
profvis({
 simulated_values <- numeric(n)  #we need n samples

   num_accepted <- 0 #we keep track of how many we have

   while (num_accepted < n)                                                    0.1          20
   {
       samples <- ceiling( scale*(n - num_accepted)/proposal_object$a ) #sample expected amount

       prop_samples <- proposal_object$sim(samples)  #get all the samples      0.9          20
       uniform_samples <- runif(samples)

       accept <- uniform_samples <=  proposal_object$a*proposal_object$tdens(prop_samples) /   -1.1   169.0        1740
           proposal_object$pdens(prop_samples)

       new_additions <- min(n - num_accepted, sum(accept))

       simulated_values[(num_accepted + 1):(num_accepted + new_additions)] <- (prop_samples[accept])   3.1          10
[1:new_additions]

       num_accepted <- num_accepted + new_additions

   }

   simulated_values})
```

Vast majority of time is spent in the evaluation of the densities. It can further be shown that it is the target density which is causing the problem.

This specific density could be implemented in C++, but the generic parts of the code seem otherwise well-optimized.

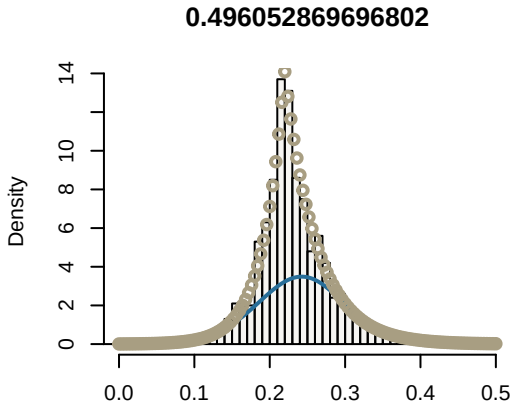
Adaptive envelope factory

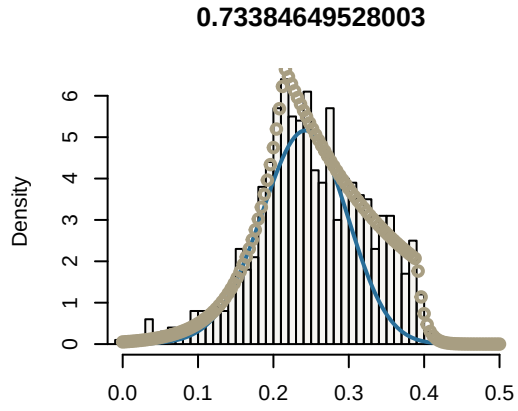
```
envelope_factory <- function(x,
                             target_dens,
                             logderiv = NULL,
                             lower_support = -Inf,
                             upper_support = Inf)
{

    #check if the log-derivative was supplied, otherwise we get it ourselves:
    if (is.null(logderiv))
    {
        logderiv <- function(xx) grad(function(x) log(target_dens(x)), xx)
    }

    #calculate a-vector and stop if there are integrability issues

    a <- logderiv(x)

    #check if we need to stop
    continue <- (a[1] > 0 & a[length(a)] < 0) | (a[length(a)] > 0 & lower_support > -Inf)
    | a[1] < 0 & upper_support < Inf

    if (!continue)
    {
        stop("Envelope is not integrable. Re-submit new x")
    }

    #now calculate b, z, Fz, Q, and const

    b <- log(target_dens(x)) - a*x

    z <- c(lower_support, -diff(b)/diff(a), upper_support)

    Fz <- numeric(length(x))

    for (i in seq_along(Fz))
    {
        Fz[i] <- exp(b[i]) * ( exp( a[i] * z[i+1]) - exp(a[i]*z[i])) / a[i]
    }

    Q <- c(0, cumsum(Fz))
```

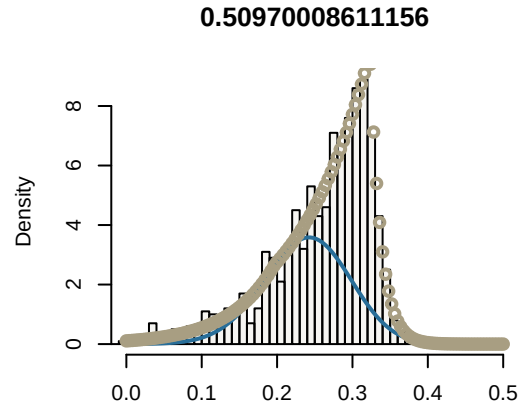
Adaptive envelope factory continued

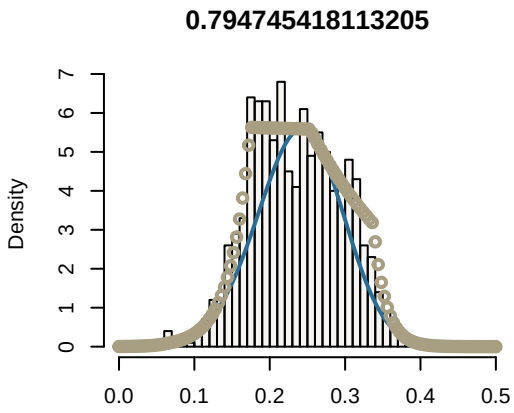
```
const <- Q[length(Q)]

#now define simulator and envelope -- must be vectorized

proposal_density <- function(x)
    index <- findInterval(x, z)  #Given x, find z-interval that x belongs to

    #findInterval is optimized and O(log(length(z)*length(x))

    #now just evaluate the function and return
    V <- a[index]*x + b[index]
    exp(V)/const
}

simulator <- function(n)
{
     #we need n uniform samples
     q <- runif(n)

     #find the index to which const*q belongs to.
     index <- findInterval(const*q, Q) + 1

     #solve the equation for x
     log( (const*q - Q[index])*a[index ]*exp(-b[index])
     + exp(a[index]*z[index]))/a[index ]
}

#return the list-object that can be used directly in previous functions.
 list( tdens = dens$tdens, pdens = proposal_density, sim = simulator, alpha = 1/const)
```

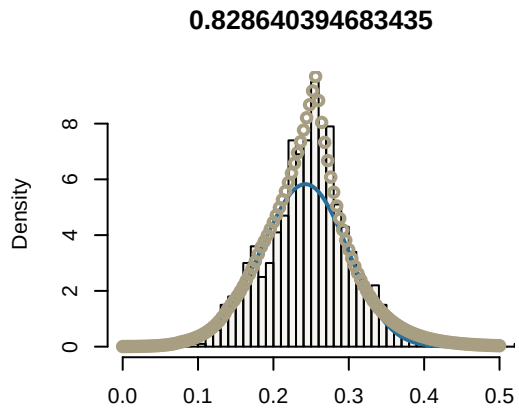
Comparing envelopes (3 to 8 nodes)

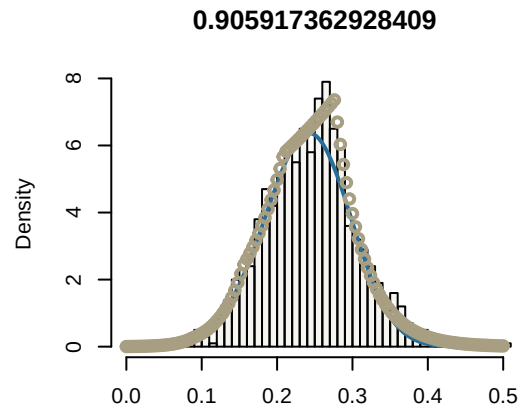




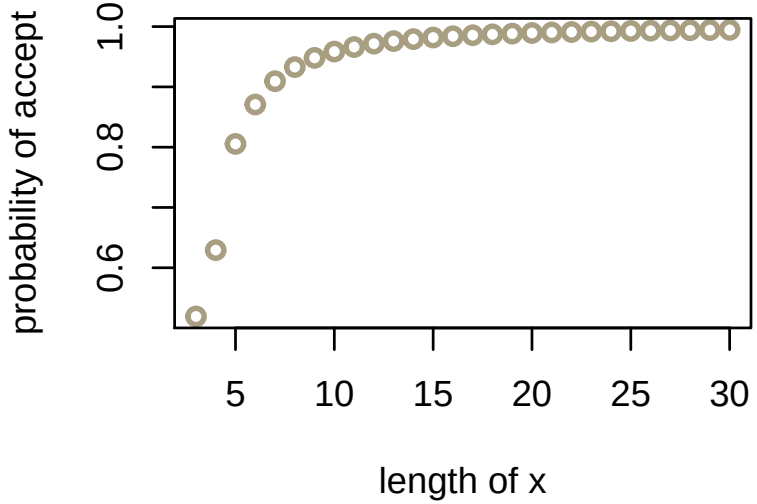


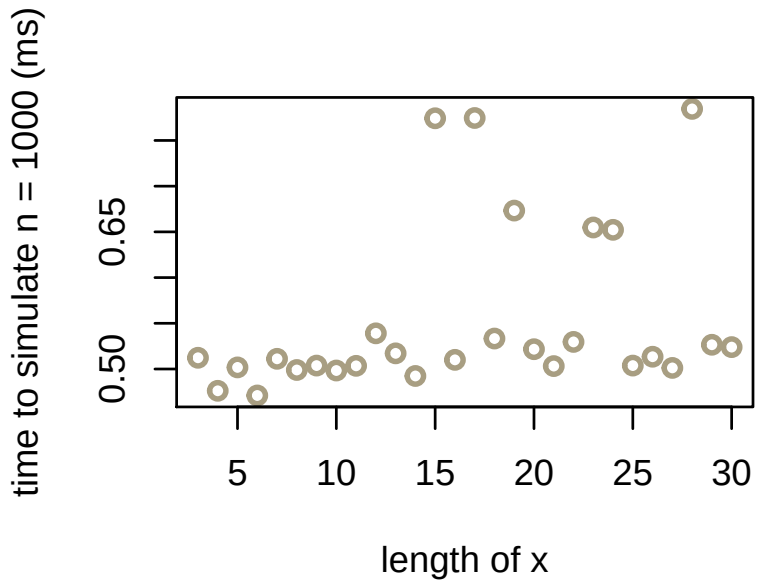


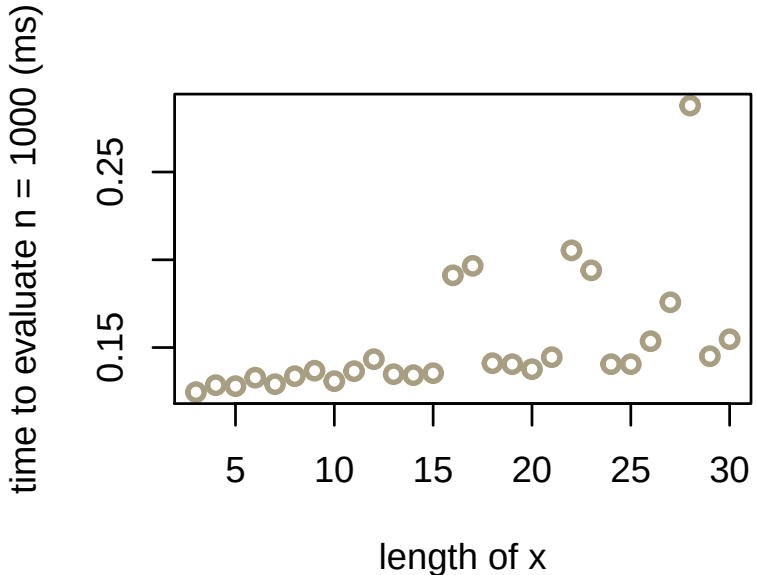


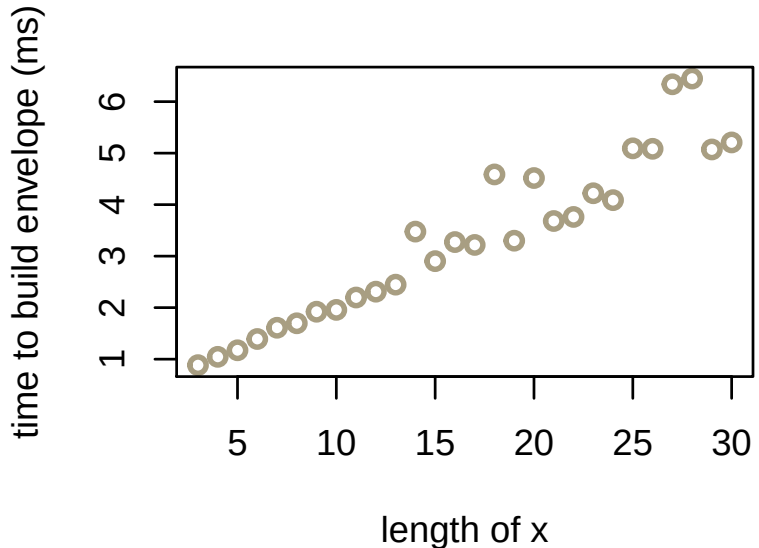


Testing the generator for various lengths of the x-vector

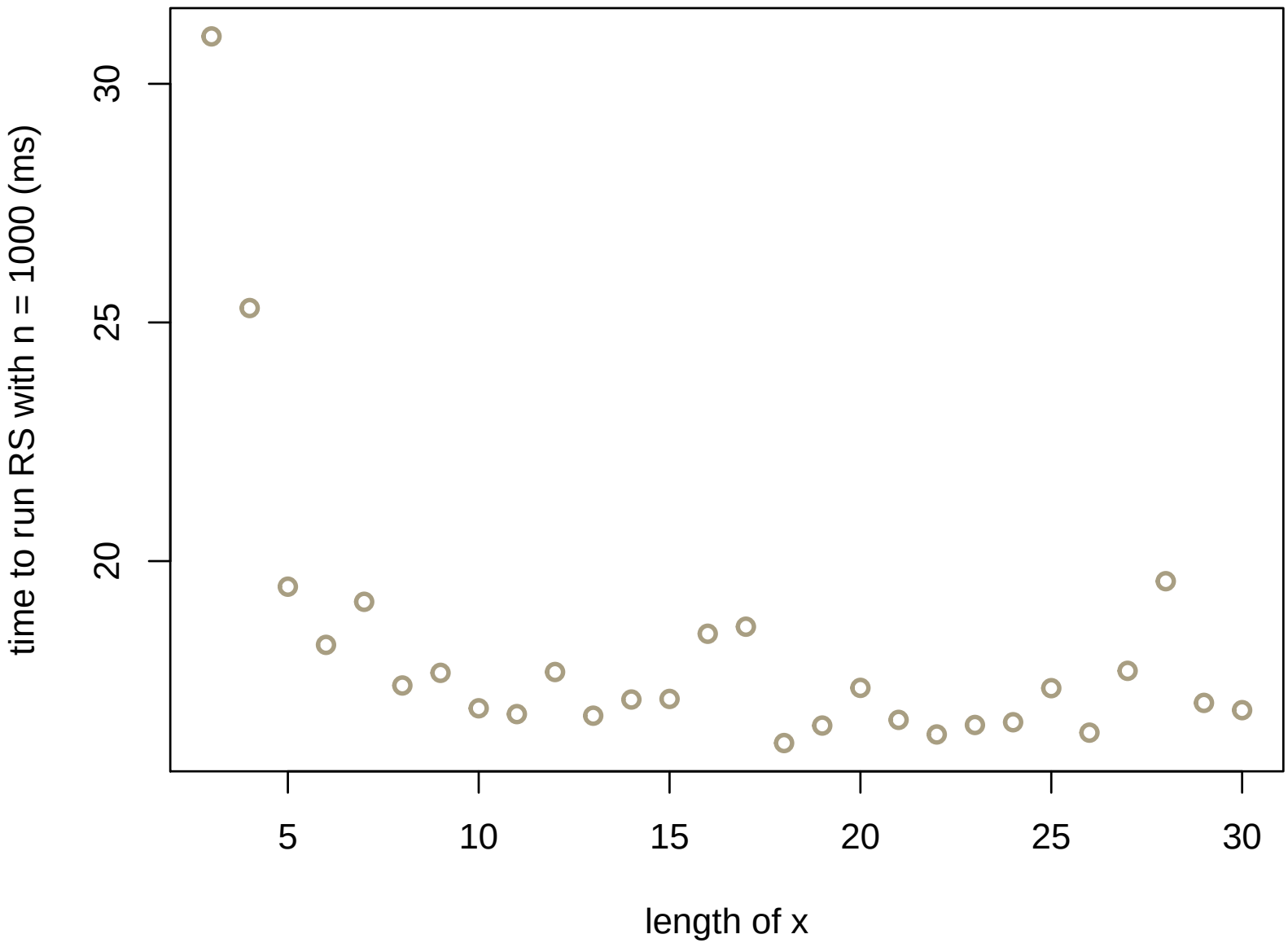




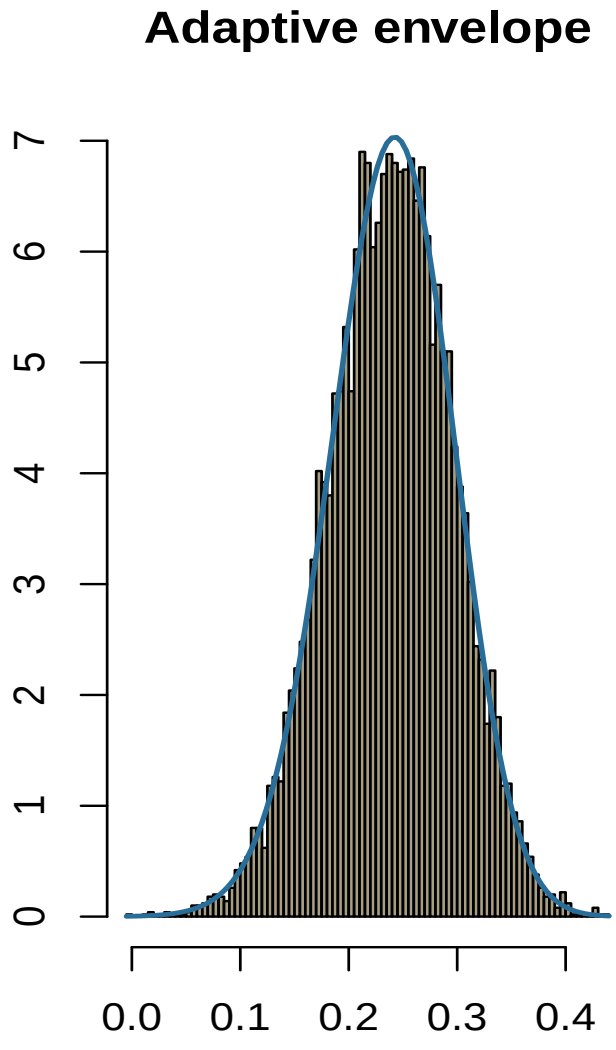


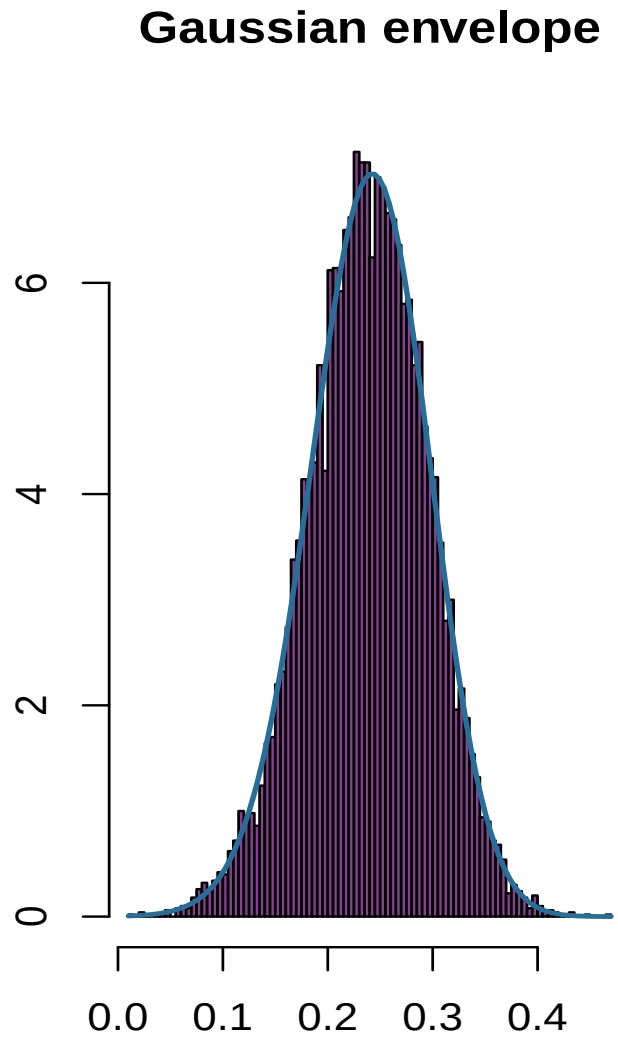


Timing the generator for various lengths of the x-vector



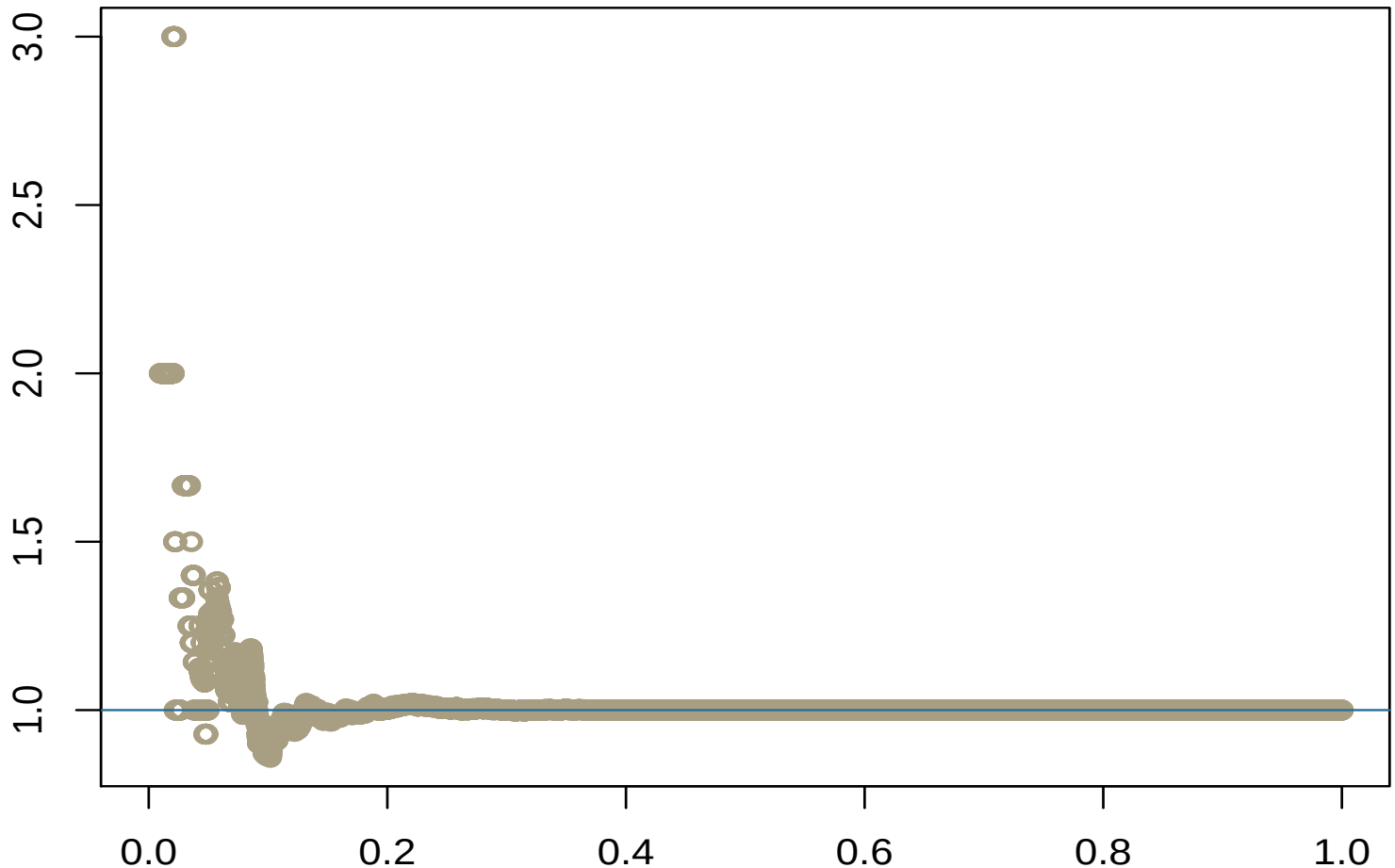
Comparing the adaptive envelope ($\alpha = 0.9839$) with gaussian envelope ($\alpha = 0.9163$), $n = 20000$



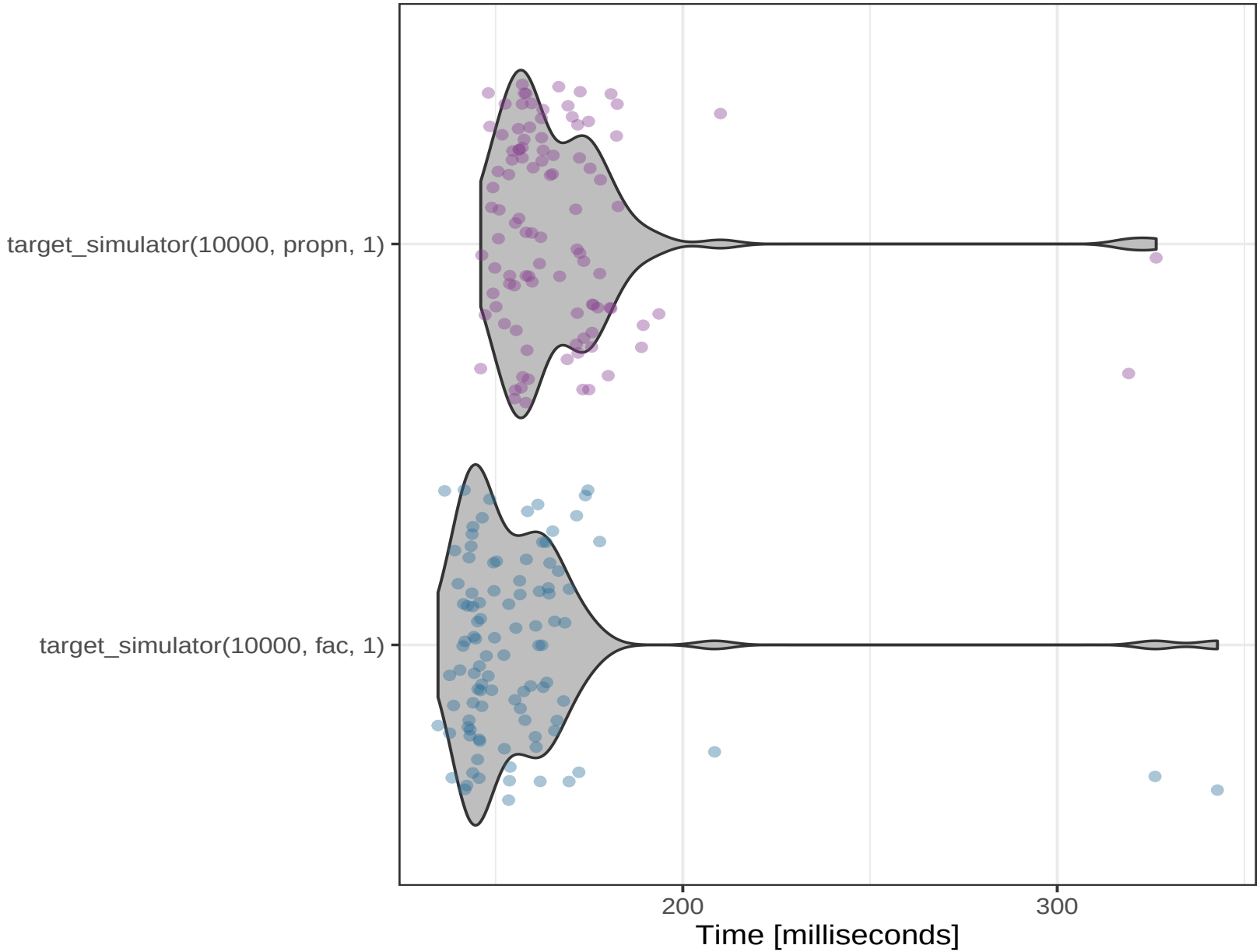


Comparing the adaptive envelope ($\alpha = 0.9839$) with gaussian envelope ($\alpha = 0.9163$), $n = 20000$

Ratio of empirical CDFs (easily passes the Kolmogorov-Smirnov test)



Comparing the adaptive envelope ($\alpha = 0.9839$) with gaussian envelope ($\alpha = 0.9163$), $n = 10000$



Using OOP

```
proposal_factory <- function(target_dens, proposal_dens, proposal_sim, a)
{
   structure(list( tdens = target_dens,
         pdens = proposal_dens,
         sim = proposal_sim,
         alpha = a), class = "RS")
}

plot.RS <- function(object, support)
{
    gridx <- seq(support[1], support[2], length.out = 512)

    plot(gridx, object$alpha*object$tdens(gridx), col = "#286e99", xlab = "", ylab = "",
    main = paste("Alpha equals", propn$alpha), type = "l", lwd = 2)

    points(gridx, object$pdens(gridx), col = "#a89e82")

}

simulate.RS <- function(object, nsim = 1, scale = 1, plot = FALSE)
{
    simulations <- target_simulator(nsim, object, scale)

    if (plot)
    {
        hist(simulations, prob = TRUE,  main = "", ylab = "", xlab = "", col = "#a89e82")
        curve(object$tdens(x), add = TRUE)
    }
    simulations
}
```