```r
if_all_unique_append = function(x_vec, x){
  EPSILON = 1e-8
  distance_vec = abs(x_vec - x)
  if (all(distance_vec >= EPSILON)){
    x_vec = c(x_vec, x)
    x_vec = sort(x_vec)
    return(x_vec)
  } else {
    return(x_vec)
  }
}


get_initial_x_vec_and_D = function(target_density, h_of, h_prime_of,
                                   x_domain, ...){
  # optimization starting point
  paramemter_init = 0
  multiples = 2
  if (x_domain[1] == -Inf && x_domain[2] == Inf){
    paramemter_init = 0
  }
  else if (x_domain[1] == -Inf){
    paramemter_init = x_domain[2] - multiples*abs(x_domain[2])
  }
  else if (x_domain[2] == Inf){
    paramemter_init = x_domain[1] + multiples*abs(x_domain[1])
  }
  else {
    paramemter_init = (x_domain[1] + x_domain[2]) / 2
    if (target_density(paramemter_init, ...) < 1e-8){
      paramemter_init = 0
    }
  }

  # -g(x) to be optimized to get where h'(x) == 0
  negative_g_of_x = function(x){ -target_density(x, ...) }
  x_max_optim = optim(paramemter_init, negative_g_of_x, method='BFGS')

  # get the x where log(g(x)) is maximized
  x_mid = x_max_optim$par
  x_mid = min(x_mid, x_domain[2])
  x_mid = max(x_mid, x_domain[1])
  lower_bound = x_mid
  upper_bound = x_mid

  # get the lower bound
  pdf_at_x_mid = target_density(x_mid, ...)
  threshold_density = 0.0001 * pdf_at_x_mid
  exponent = 0
  while (target_density(lower_bound, ...) > threshold_density){
    exponent = exponent + 1
    lower_bound = x_mid - 2^exponent
  }
```

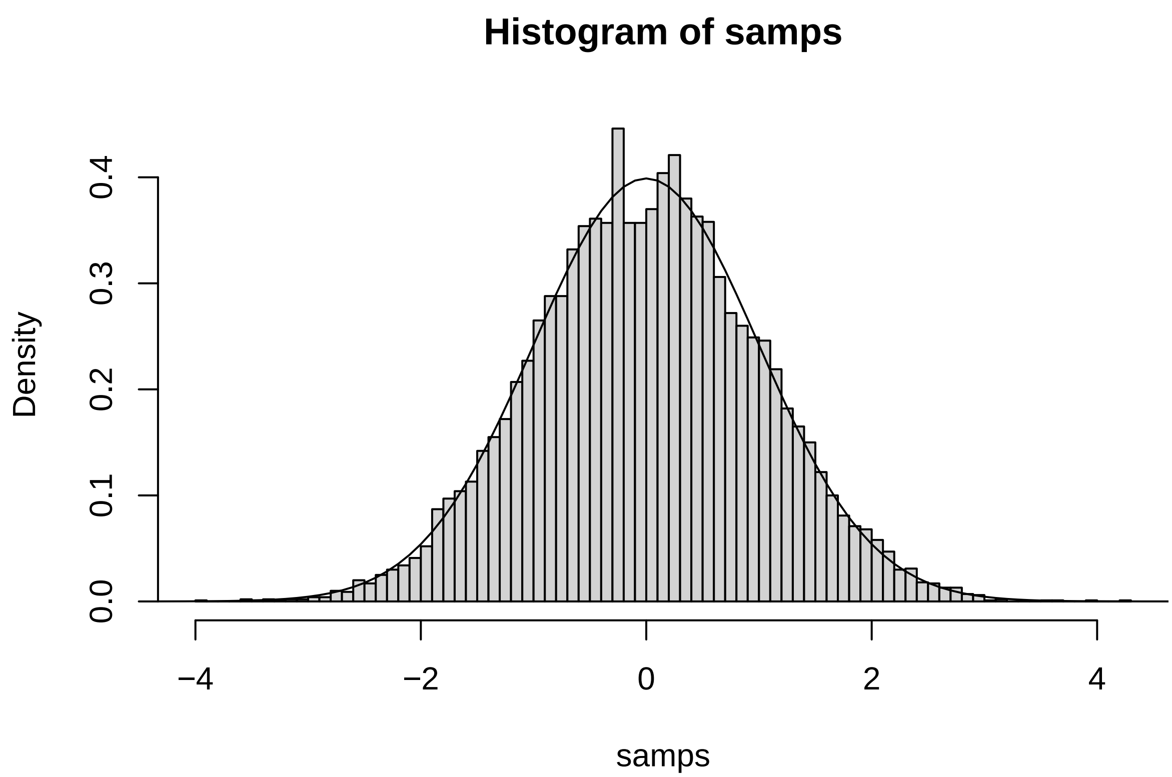
```
  exponent = 0
  while (target_density(upper_bound, ...) > threshold_density){
    exponent = exponent + 1
    upper_bound = x_mid + 2^exponent
  }

  # check with user input
  lower_bound = max(x_domain[1], lower_bound)
  upper_bound = min(x_domain[2], upper_bound)

  # return the x vector
  x_vec_part_1 = seq(lower_bound, x_mid, length.out = 4)
  x_vec_part_2 = seq(x_mid, upper_bound, length.out = 4)
  x_vec_prop = unique(c(x_vec_part_1[1:3] , x_vec_part_2[2:4]))
  x_vec = c()

  # form initial x_vec with the x with non-zero h'(x)
  for (x in x_vec_prop){
    abs_h_prm_x = abs(h_prime_of(x))
    if (is.na(abs_h_prm_x)){
      next;
    } else if (abs_h_prm_x == Inf){
      next;
    } else if (abs_h_prm_x == -Inf){
      next;
    } else if (abs_h_prm_x > 1e-8){
      x_vec = append(x_vec, x)
    }
  }

  # bound vector and return
  D_vec = c(lower_bound, upper_bound)
  return_list = list(x_vec, D_vec, x_mid)

  # check for log-concavity of function
  EPS = 1e-8
  l = length(x_vec)
  if (l < 3) {
    stop('Please Respecify Bounds and Target Density:
    Given Bound too Flat to form the Initial X Vector,
    Numerically Violated Log Concavity.')
  }
  h_prime_jumps = h_prime_of(x_vec[2:l]) - h_prime_of(x_vec[1:(l-1)])
  if(!all(h_prime_jumps <= EPS)) {
    stop('Please Respecify Bounds and Target Density
         Input Target Density is not Log-Concave Within the Domain.')
  }




  return(return_list)
}
```

```r
get_z_vec_and_I_vec = function(target_density, h_of, h_prime_of,
                               xk, d, ...) {

  l = length(xk)
  z = numeric(l+1)
  z[1] = d[1]
  z[l+1] = d[2]

  z[2:l] = ( h_of(xk[2:l])
             - h_of(xk[1:(l-1)])
             - xk[2:l] * h_prime_of( xk[2:l])
             + xk[1:(l-1)] * h_prime_of( xk[1:(l-1)])) /
    (h_prime_of( xk[1:(l-1)]) - h_prime_of( xk[2:l]))

  integ_first_part = exp(h_of(xk)) / h_prime_of(xk)
  integ_scnd_part = exp((z[2:(l+1)]-xk)*h_prime_of(xk))  -  exp((z[1:l]-xk)*h_prime_of(xk))
  integ = integ_first_part * integ_scnd_part

  integ_cum_sum = cumsum(integ)
  s = integ_cum_sum[l]
  I = integ_cum_sum/s

  return(list(z,I,s))
}

get_samples_from_density = function(target_density, h_of, h_prime_of,
                                    x_domain, n, num_iter_allowed, ...){
  # get initial x vector and modified appropriate domain
  x_vec_and_D_and_x_mid =  get_initial_x_vec_and_D(target_density, h_of, h_prime_of,
                                                   x_domain, ...)
  x_vec = x_vec_and_D_and_x_mid[[1]]
  D = x_vec_and_D_and_x_mid[[2]]
  x_mid = x_vec_and_D_and_x_mid[[3]]

  # loop over to get n samples
  samples = rep(NULL, n)
  curr_num = 0
  num_iter = 0
  while (curr_num < n && num_iter < num_iter_allowed){
    # record the number of generations
    num_iter = num_iter + 1
    if (num_iter == num_iter_allowed) {
      warning('Preset Maximum Allowed Number of Sampling Iterations Reached.')
    }

    # get z-vector and I-vector
    z_vec_and_I_vec = get_z_vec_and_I_vec(target_density, h_of, h_prime_of,
                                          x_vec, D, ...)
    z_vec = z_vec_and_I_vec[[1]]
    I_vec = z_vec_and_I_vec[[2]]
    I_sum = z_vec_and_I_vec[[3]]

    c = runif(1,0,1)
```

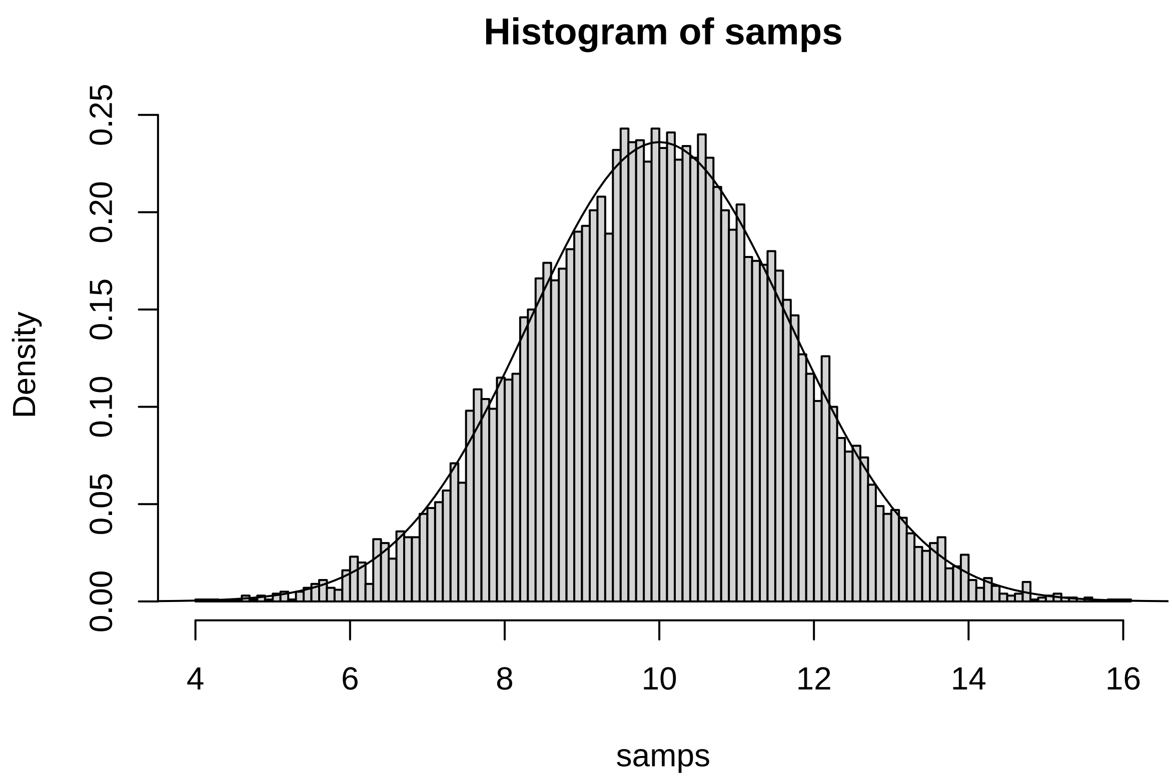
```r
  w = runif(1,0,1)

  # get index of where x_star would fall into
  j = sum(c > I_vec) + 1

  # define functions l_k_of_x and u_k_of_x
  x_j = x_vec[j]
  x_j_plus_one = x_vec[j+1]

  if (j == length(x_vec)){
    x_j_plus_one = D[2]
  }

  I_c = 0
  if(j != 1) {
    I_c = I_vec[j-1]
  }

  s1_first_part = I_sum*(c-I_c)*h_prime_of(x_j)/exp(h_of(x_j))
  s1_scnd_part = exp(h_prime_of(x_j)*(z_vec[j]-x_vec[j]))
  s1 = s1_first_part + s1_scnd_part

  x_star = log(s1)/h_prime_of(x_j) + x_vec[j]

  l_k_of = function(x, x_j, x_j_plus_one){
    numerator = (x_j_plus_one-x)*h_of(x_j) + (x-x_j)*h_of(x_j_plus_one)
    denominator = max(x_j_plus_one - x_j , 1e-8 )
    return(numerator/denominator)
  }
  u_k_of = function(x, x_j){ h_of(x_j) + (x-x_j)*h_prime_of(x_j) }

  # acceptation criterion
  first_threshold = exp(l_k_of(x_star, x_j, x_j_plus_one) - u_k_of(x_star, x_j))
  if (w <= first_threshold){
    curr_num = curr_num + 1
    samples[curr_num] = x_star
  }
  else
  {
    scnd_threshold = exp(h_of(x_star) - u_k_of(x_star, x_j))
    if (w <= scnd_threshold){
      curr_num = curr_num + 1
      samples[curr_num] = x_star
      if (abs(x_star - x_mid) > 1e-8){
        x_vec = if_all_unique_append(x_vec, x_star)
      }
    } else {
      if (abs(x_star - x_mid) > 1e-8){
        x_vec = if_all_unique_append(x_vec, x_star)
      }
    }
  }
}
```

```
  return(samples)
}

ars = function(target_density, n, x_domain=c(-Inf,Inf), num_iter_allowed=10*n, ...){
  # Input check
  if (!is.function(target_density)) { stop('Target Density is Not Function Object.') }
  if (!is.numeric(x_domain)) { stop('Domain Input is Not Numeric.') }
  if (!is.numeric(n)) { stop('n (number of sample) is Not Numeric.') }
  if (!is.numeric(num_iter_allowed)) {
    stop('maximum number of sampling iteration allowed is Not Numeric.') }
  # domain input wrong
  if (length(x_domain) != 2) { stop('Domain Input should have 2 Arguments.') }
  # domain too tiny
  if (abs(x_domain[1] - x_domain[2]) < 1e-8) { stop('Domain Error, Specified Domain too Small.') }
  # correct domain input if it's in reversed order
  if (x_domain[1] > x_domain[2]) {  x_domain = x_domain(bounds) }

  # get h_of(x) = log(g(x)), and, h_prime(x)
  h_of = function(x_vec){ log(target_density(x_vec, ...)) }
  h_prime_of = function(x_vec){
    dx = 1e-8
    derivative = (h_of(x_vec+dx) - h_of(x_vec)) / dx
  }
  samps = get_samples_from_density(target_density, h_of, h_prime_of,
                                   x_domain, n, num_iter_allowed=num_iter_allowed, ...)
  if ( length(samps) < n ){
    warning('Sampling Inefficientcy Encountered.
            To Get Input Sized Samples, Please Increase { num_iter_allowed }.')
  }
  return(samps)
}
```

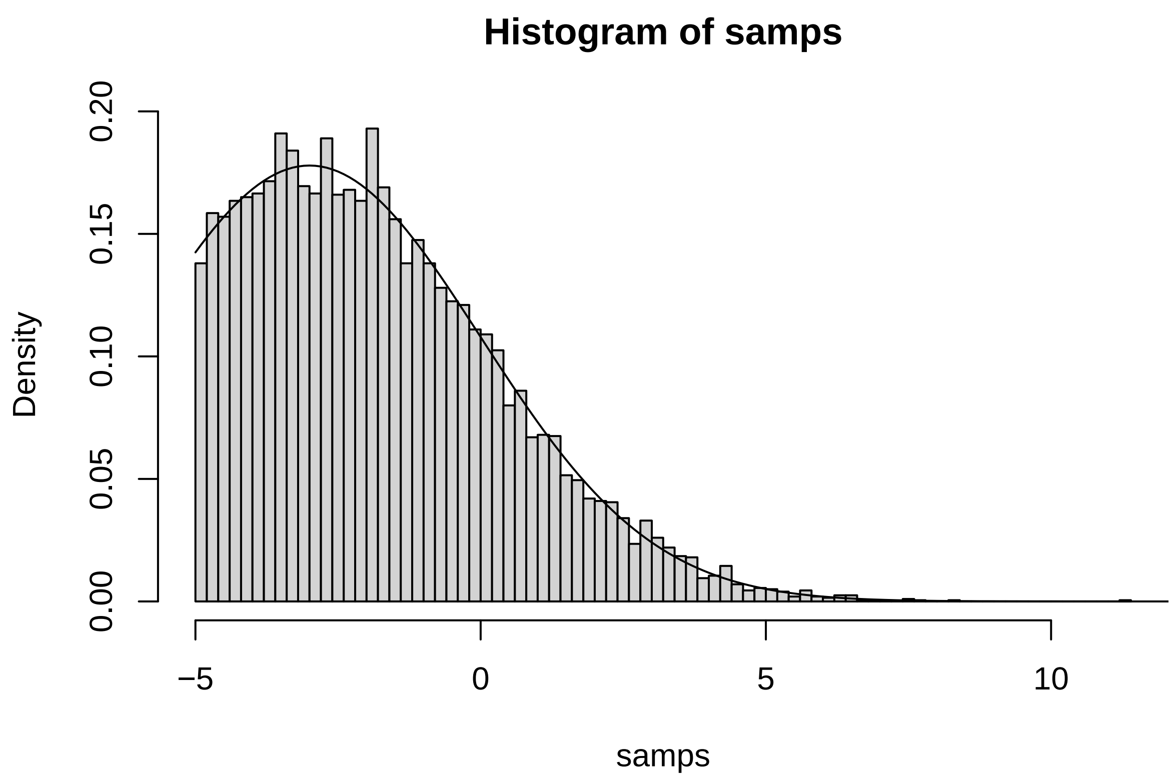
```r
# Standard Normal Distribution

mu = 0
std = 1

lower = -Inf
upper = Inf

samps = ars(dnorm, x_domain=c(lower, upper), n=10000, mean=mu, sd=std)

hist(samps, breaks=100, freq=FALSE)
plt_range = ((-1000):(1000))/10
lines(plt_range, dnorm(plt_range, mean=mu, sd=std) / (pnorm(upper, mean=mu, sd=std)
                                                       - pnorm(lower, mean=mu, sd=std)))
```



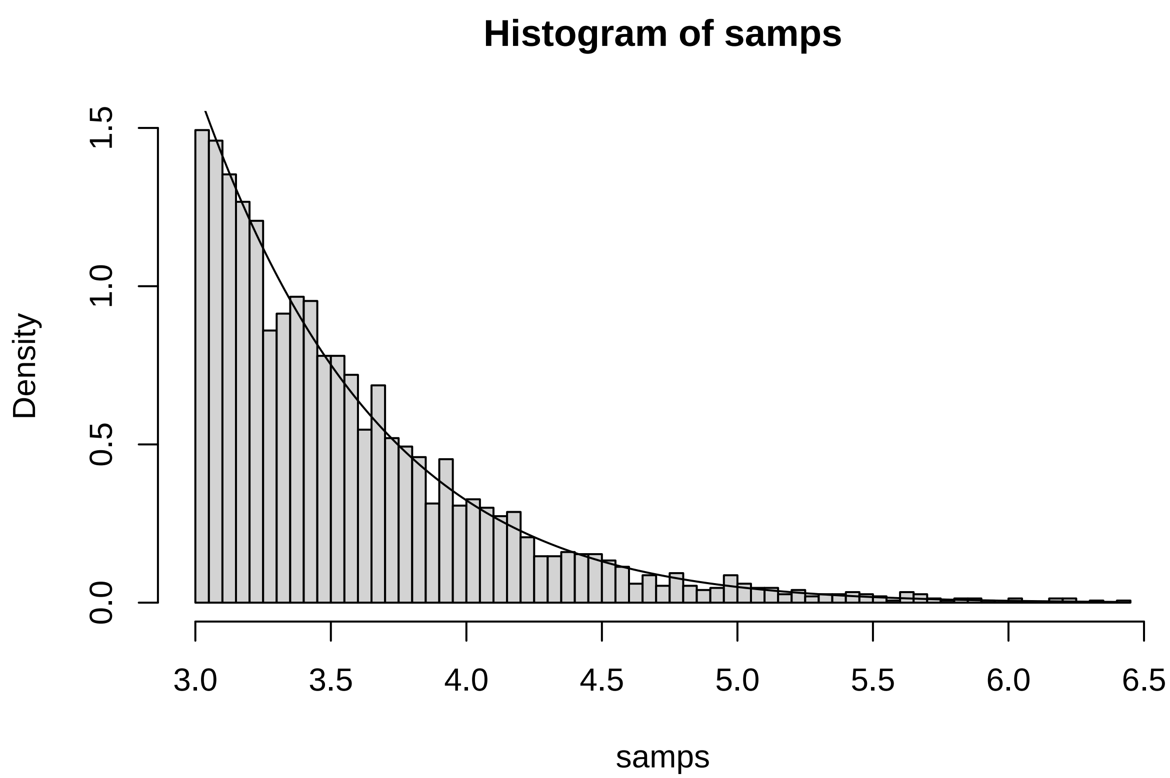
```r
# Normal Distribution

mu = 10
std = 1.69

lower = -Inf
upper = Inf

samps = ars(dnorm, x_domain=c(lower, upper), n=10000, mean=mu, sd=std)

hist(samps, breaks=100, freq=FALSE)
plt_range = ((-1000):(1000))/10
lines(plt_range, dnorm(plt_range, mean=mu, sd=std) / (pnorm(upper, mean=mu, sd=std)
                                                      - pnorm(lower, mean=mu, sd=std)))
```



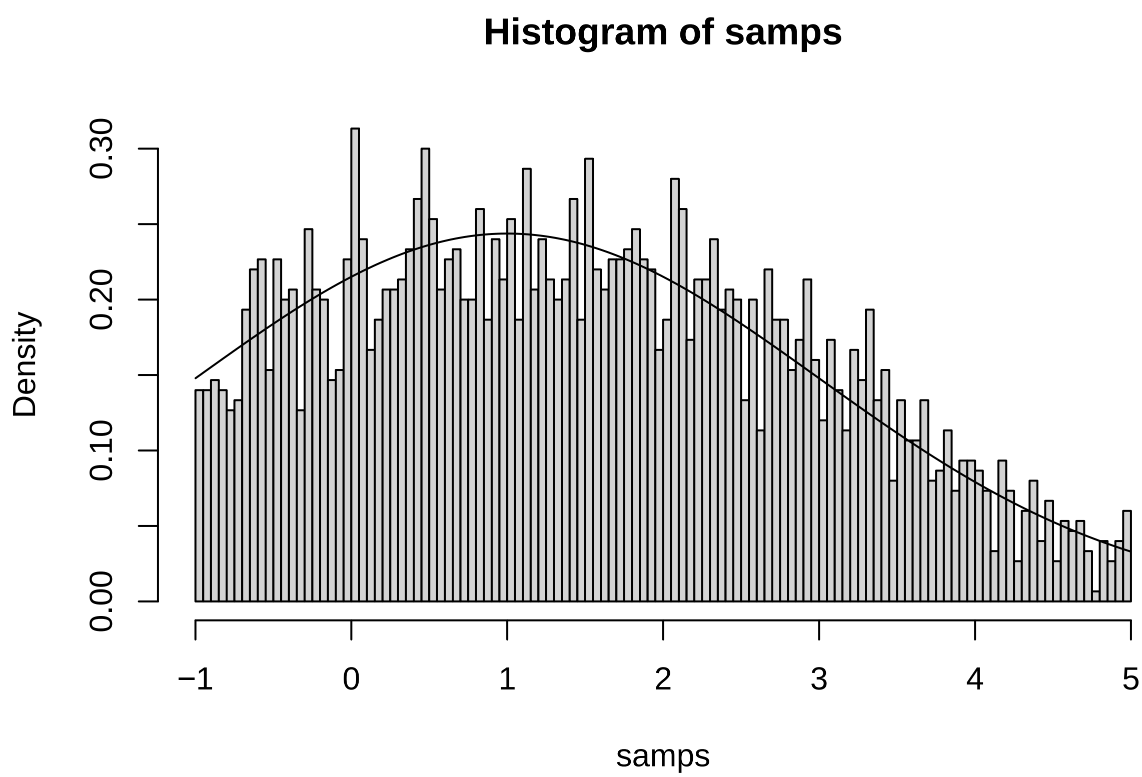
```r
# Truncated Normal Distribution

mu = -3
std = 3

lower = -5
upper = 1200

samps = ars(dnorm, 10000,  c(lower, upper), mean=mu, sd=std)

hist(samps, breaks=100, freq=FALSE)
plt_range = ((lower*10):(upper*10))/10
lines(plt_range, dnorm(plt_range, mean=mu, sd=std) / (pnorm(upper, mean=mu, sd=std)
                                                      - pnorm(lower, mean=mu, sd=std)))
```



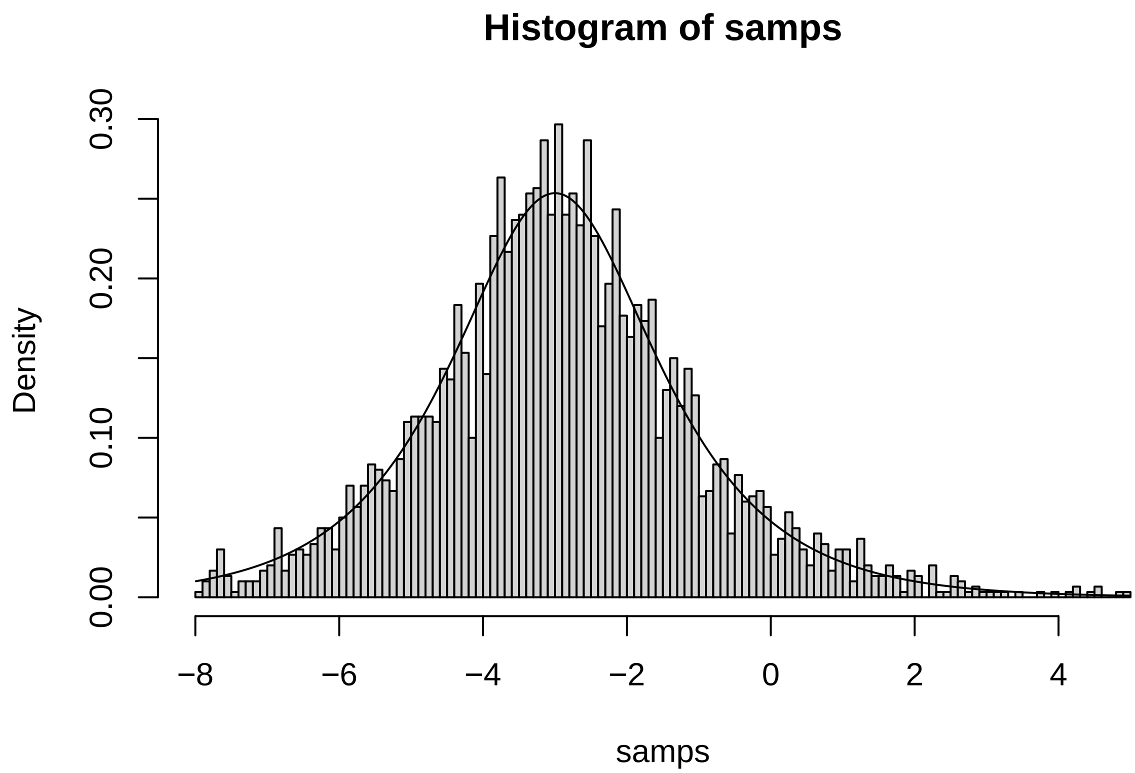
```r
# Truncated Normal Distribution

mu = -3
std = 2

lower = 3
upper = 15

samps = ars(dnorm, 3000, c(lower, upper), mean=mu, sd=std)

hist(samps, breaks=100, freq=FALSE)
plt_range = seq(min(samps) , max(samps) , length.out=1000)
lines(plt_range, dnorm(plt_range, mean=mu, sd=std) / (pnorm(max(samps)) - pnorm(min(samps))))
```



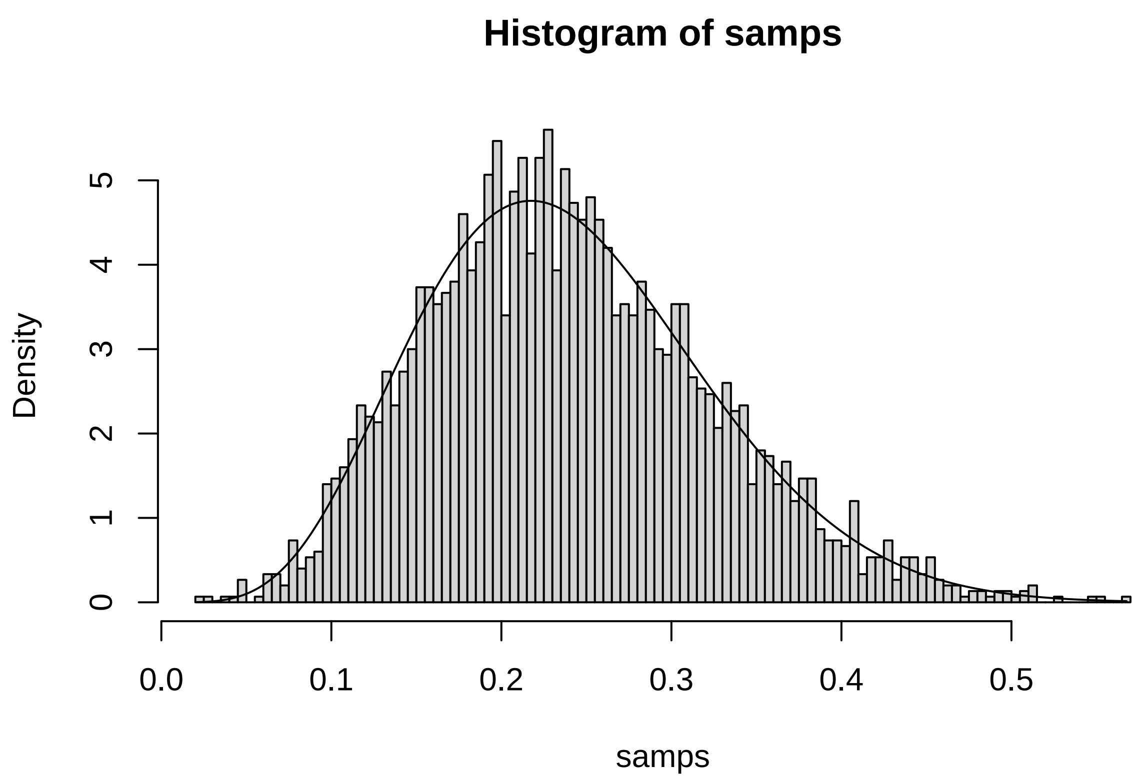
```r
# Truncated Normal Distribution

mu = 1
std = 2

lower = -1
upper = 5

samps = ars(dnorm, 3000, c(lower, upper), num_iter_allowed=1e7, mean=mu, sd=std)

hist(samps, breaks=100, freq=FALSE)
plt_range = seq(min(samps) , max(samps) , length.out=1000)
lines(plt_range, dnorm(plt_range, mean=mu, sd=std) / (pnorm(max(samps), mean=mu, sd=std)
                                                       - pnorm(min(samps), mean=mu, sd=std)))
```



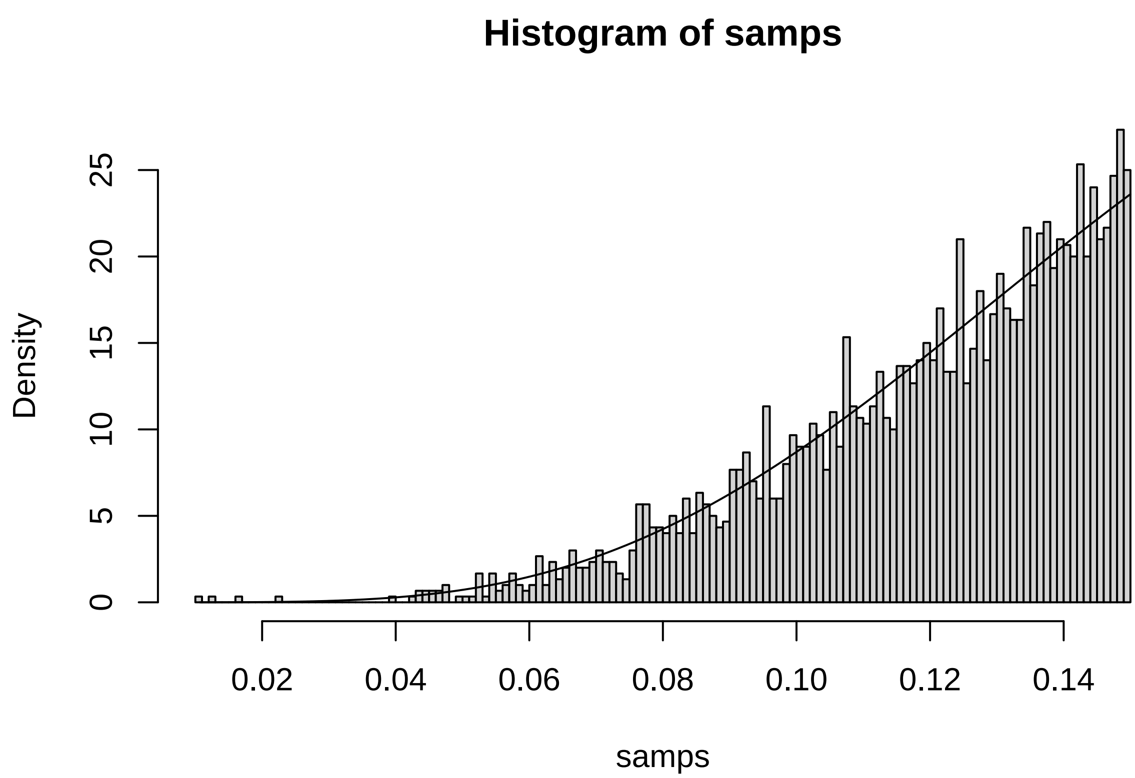
```r
# The Hyperbolic Secant Distribution

dsech <- Vectorize(function(x,mu,sigma,log = FALSE){
  logden <-  -log(2) - log(sigma) - log( cosh( 0.5*pi*(x-mu)/sigma ) )
  val <- ifelse(log, logden, exp(logden))
  return(val)
})

psech <- Vectorize(function(x,mu,sigma,log.p = FALSE){
  logcdf <-  log(2) - log(pi) + log( atan( exp( 0.5*pi*(x-mu)/sigma ) ) )
  val <- ifelse(log.p, logcdf, exp(logcdf))
  return(val)
})


mean = -3
std = 2

lower = -8
upper = 5

samps = ars(dsech, x_domain=c(lower, upper), n=3000, mu=mean, sigma=std)

hist(samps, breaks=100, freq=FALSE)
plt_range = seq(min(samps) , max(samps) , length.out=1000)
lines(plt_range, dsech(plt_range, mu=mean, sigma=std) / (psech(max(samps), mu=mean, sigma=std)
                                                         - psech(min(samps), mu=mean, sigma=std)))
```


Histogram of samps
Density
0.00
0.10
0.20
0.30
−8
−6
−4
−2
0
2
4
samps

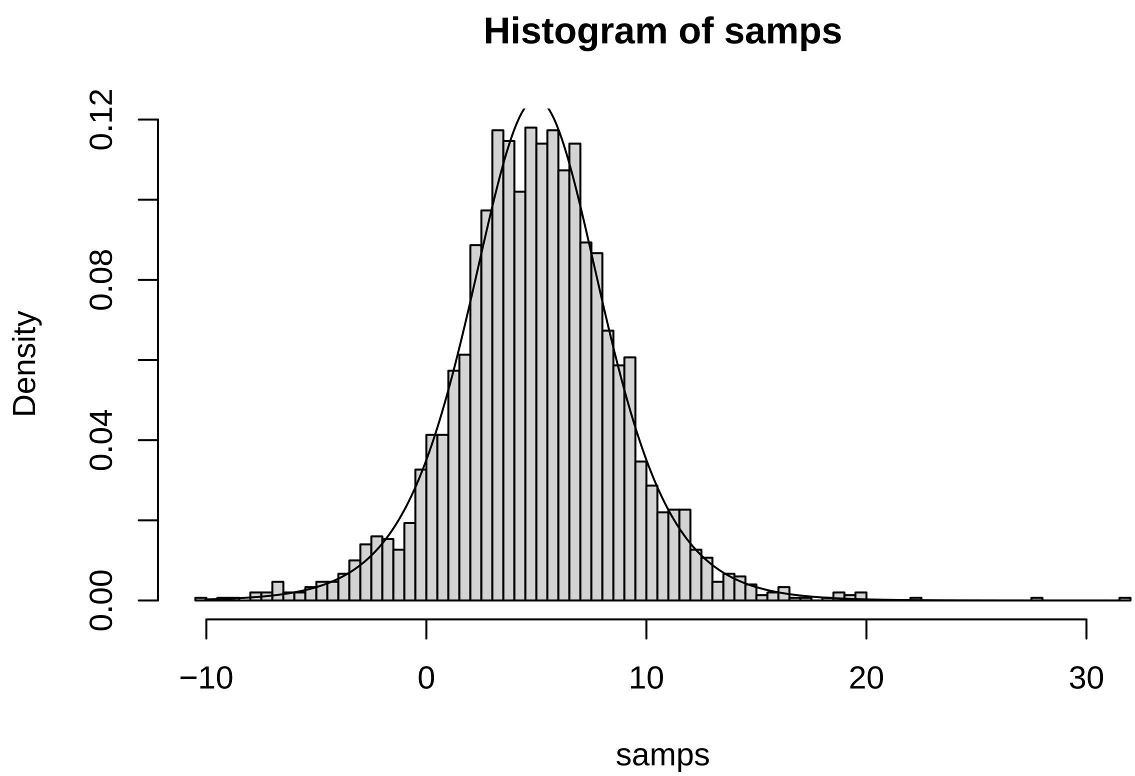
```r
# Beta distribution

shape1 = 6
shape2 = 19

samps = ars(dbeta, x_domain=c(0, 1), n=3000, shape1=shape1, shape2=shape2)

hist(samps, breaks=100, freq=FALSE)
plt_range = seq(min(samps) , max(samps) , length.out=1000)
lines(plt_range,
      dbeta(plt_range,
            shape1=shape1,
            shape2=shape2) / (pbeta(max(samps),
                                    shape1=shape1, shape2=shape2)
                              - pbeta(min(samps),
                                     shape1=shape1, shape2=shape2)))
```



```r
# Truncated Beta distribution

shape1 = 6
shape2 = 19

samps = ars(dbeta, 3000, c(0, 0.15), shape1=shape1, shape2=shape2)

hist(samps, breaks=100, freq=FALSE)
plt_range = seq(min(samps) , max(samps) , length.out=1000)
lines(plt_range,
      dbeta(plt_range,
            shape1=shape1,
            shape2=shape2) / (pbeta(max(samps),
                                    shape1=shape1, shape2=shape2)
                              - pbeta(min(samps),
                                      shape1=shape1, shape2=shape2)))
```



```r
# Logistic Distribution

location = 5
scale = 2

samps = ars(dlogis, x_domain=c(-Inf, Inf), n=3000, location = location, scale =scale)

hist(samps, breaks=100, freq=FALSE)
plt_range = seq(min(samps) , max(samps) , length.out=1000)
lines(plt_range,
      dlogis(plt_range,
            location = location,
            scale =scale) / (plogis(max(samps),
                                    location = location, scale =scale)
                              - plogis(min(samps),
                                    location = location, scale =scale)))
```



```r
# Uniform Distribution is not log-concave
# thus we expect ERROR here.

a = 10
b = 11
# samps = ars(target_density=dunif, x_domain=c(a, b), n=3000, min=a, max=b)
```