Rejection sampling

Computational Statistics

Adam Gorm Hoffmann

Simulate from unnormalized density q

- Find unnormalized density p and $\alpha > 0$ such that $\alpha q \leq p$
- Draw values $U \sim \mathrm{Unif}(0, 1)$ and $Y \sim p$.
  - If $U > \alpha q(Y)/p(Y)$ reject Y and try again.
  - Otherwise, take Y as a simulated value from q.

Specific problem

$$
\begin{aligned}
q(y) &= \prod_{i=1}^{100} \exp(y z_i x_i - \exp(y x_i)), \quad y \geq 0, \\
p(y) &= \exp(-y^2/2).
\end{aligned}
$$

Due to the size of q, I will work on log-scale.

`log_p_over_q(uniroot(ddy_log, c(0, 2), tol = 1e-20))`
(where `ddy_log` calculates $\frac{d}{dy} \log \frac{p(y)}{q(y)}$) gives

$$
\alpha_{log} = \inf_{y \geq 0} \log \frac{p(y)}{q(y)} \approx 92.38.
$$

Simple loop implementation

```r
sum_xz <- sum(dat$x * dat$z)

log_q_over_p_single <- function(y)
    y^2 / 2 + y * sum_xz - sum(exp(y * x))

not_rejected_log_single <- function(y,
                              alpha_log = alpha_log_gauss,
                              log_q_over_p = log_q_over_p_single)
    log(runif(1)) <= alpha_log + log_q_over_p(y)

r_abs_norm <- function(n)
    abs(rnorm(n))

sim_loop <- function(n,
                     r_proposal = r_abs_norm,
                     not_rejected = not_rejected_log_single)
{
    y <- numeric(n)
    count <- 1
    while (count <= n) {
        y_proposal <- r_proposal(1)
        if (not_rejected(y_proposal)) {
            y[count] <- y_proposal
            count <- count + 1
        }
    }
    y
}
```

Profiling: 10^6 simulations.

```
log_q_over_p_single <- function(y)                                   260
{
    sum_exp_yx <- sum(exp(y * x))                                    14350
    y^2 / 2 + y * sum_xz - sum_exp_yx                                1680
}

not_rejected_log_single <- function(y,                               160
                                    alpha_log = alpha_log_gauss,
                                    log_q_over_p = log_q_over_p_single)
{
    u <- runif(1)                                                    17320
    log_q_over_p_y <- log_q_over_p(y)                                17400
    log(u) <= alpha_log + log_q_over_p_y                             1540
}

r_abs_norm <- function(n)                                            170
{
    normal_sim <- rnorm(n)                                           18650
    abs(normal_sim)                                                  1380
}

sim_loop <- function(n,
                     r_proposal = r_abs_norm,
                     not_rejected = not_rejected_log_single)
{
    y <- numeric(n)
    count <- 1
    while (count <= n) {                                             330
        y_proposal <- r_proposal(1)                                  21860
        if (not_rejected(y_proposal)) {                              39980
            y[count] <- y_proposal                                   140
            count <- count + 1                                       10
        }
    }
    y
}
```

Bottlenecks

- Simulating uniform variables and proposals are bottlenecks.
- Idea: Simulate more values at once.

Simulate several proposals at once

```
log_q_over_p_more <- Vectorize(log_q_over_p_single)

not_rejected_more <- function(y,
                              alpha_log = alpha_log_gauss,
                              log_q_over_p = log_q_over_p_more)
    log(runif(length(y))) <= alpha_log + log_q_over_p(y)

## Tries to simulate n values by estimating the probability of rejection based
## on the total number of accepted and rejected proposals so far.
sim_approx <- function(n,
                       total_accepted,
                       total_rejected,
                       r_proposal = r_abs_norm,
                       not_rejected = not_rejected_more)
{
    if (total_accepted == 0)
        n_proposal <- n
    else
        n_proposal <- n * (total_accepted + total_rejected) / total_accepted
    y <- r_proposal(n_proposal)
    accepted <- y[not_rejected(y)]
    n_accepted <- length(accepted)
    list(
        accepted = accepted,
        n_accepted = n_accepted,
        n_rejected = n_proposal - n_accepted

    )
}
```

... wrapped up.

```
sim_smart <- function(n,
                      r_proposal = r_abs_norm,
                      not_rejected = not_rejected_more)
{
    y <- numeric(n)
    total_accepted <- 0
    total_rejected <- 0
    while (total_accepted < n) {
        sim <- sim_approx(n,
                          total_accepted,
                          total_rejected,
                          r_proposal,
                          not_rejected)
        if (sim$n_accepted > 0) {
            y[(1 + total_accepted):(total_accepted + sim$n_accepted)] <-
                sim$accepted
            total_accepted <- total_accepted + sim$n_accepted
        }
        total_rejected <- total_rejected + sim$n_rejected
    }
    y[1:n]
}
```

Profiling: 10^6 simulations.

Only `sim_approx` takes time (of course).

```
sim_smart <- function(n,
                      r_proposal = r_abs_norm,
                      not_rejected = not_rejected_more)
{
    y <- numeric(n)
    total_accepted <- 0
    total_rejected <- 0
    while (total_accepted < n) {
        sim <- sim_approx(n,                                              30070
                          total_accepted,
                          total_rejected,
                          r_proposal,
                          not_rejected)
        if (sim$n_accepted > 0) {
            y[(1 + total_accepted):(total_accepted + sim$n_accepted)] <-     10
                sim$accepted
            total_accepted <- total_accepted + sim$n_accepted
        }
        total_rejected <- total_rejected + sim$n_rejected
    }
    y[1:n]                                                                 10
}
```

Profiling: `r_proposal` no longer a bottleneck!

```
sim_approx <- function(n,
                       total_accepted,
                       total_rejected,
                       r_proposal = r_abs_norm,
                       not_rejected = not_rejected_more)
{
    if (total_accepted == 0)
        n_proposal <- n
    else
        n_proposal <- n * (total_accepted + total_rejected) / total_accepted
    y <- r_proposal(n_proposal)                                          560
    not_rejected_y <- not_rejected(y)                                  29480
    accepted <- y[not_rejected_y]                                         20
    n_accepted <- length(accepted)
    list(
        accepted = accepted,
        n_accepted = n_accepted,
        n_rejected = n_proposal - n_accepted

    )
}
```

Profiling: `runif` no longer a bottleneck!

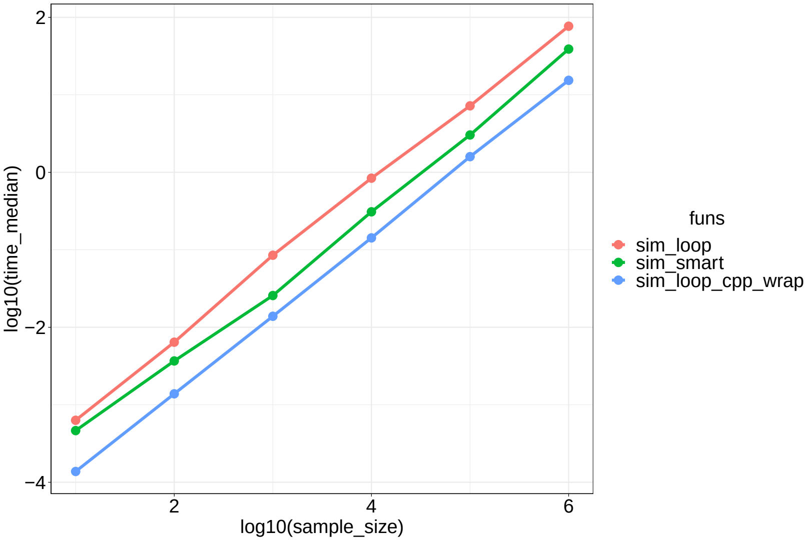
```
not_rejected_more <- function(y,
                              alpha_log = alpha_log_gauss,
                              log_q_over_p = log_q_over_p_more)
{
    unif_sim <- runif(length(y))                          250
    log_q_over_p_y <- log_q_over_p(y)                   28280
    log(unif_sim) <= alpha_log + log_q_over_p_y           960
}
```

- Only bottleneck now is `log_q_over_p`.
- Idea: Do loop implementation in Rcpp.

Rcpp version of simple loop implementation

```cpp
double
log_q_over_p_cpp(double y, const NumericVector& x, double sum_xz)
{
  unsigned long long n = x.length();
  double result = y * y / 2 + y * sum_xz;

  for (unsigned long long i = 0; i < n; i++)
    result -= exp(y * x[i]);
  return result;
}

// [[Rcpp::export]]
NumericVector
sim_loop_cpp(unsigned long long n, double alpha_log,
             const NumericVector& x, double sum_xz)
{
  NumericVector y(n);
  double y_norm;
  unsigned long long count = 0;

  while (count < n) {
    y_norm = R::rnorm(0,1);
    if (log(R::runif(0,1)) <= alpha_log + log_q_over_p_cpp(y_norm, x, sum_xz))
      y[count++] = y_norm;
  }
  return y;
}
```

Benchmarking: order as expected



Adaptive envelopes

- Assume log-concave density
- log-affine envelope

$$f(y) \leq \exp\left(\sum_{i=1}^{m}(a_i y + b_i)1_{I_i}(y)\right)$$

where

$$a_i = (\log f(t_i))', \quad b_i = \log f(t_i) - a_i t_i.$$

- Knots $t_1 < \ldots < t_m$ and grid $I_i = (z_i, z_{i+1}]$ with $z_{i+1} = \frac{b_{i+1} - b_i}{a_i - a_{i+1}}$ such that $a_i z_{i+1} + b_i = a_{i+1} z_{i+1} + b_{i+1}$ (so the envelope changes log-affine function where they intersect).
- More knots $\Rightarrow$ fewer rejections, but more computations per proposal.
- Fewer knots $\Rightarrow$ more rejections, but fewer computations per proposal.

Simulating from adaptive envelope

With

$$F_i(y) = \int_{z_i}^{y} e^{a_i z + b_i} dz = \frac{1}{a_i} e^{b_i} (e^{a_i y} - e^{a_i z_i})$$

and $Q_i = \sum_{k=1}^{i-1} F_i(z_i)$ for $i = 1, \ldots, m+1$, and $c = Q_{m+1}$, the piecewise log-affine adaptive envelope density leads to the CDF

$$F(y) = \frac{Q_i + F_i(y)}{c}, \quad \text{for } y \in I_i.$$

Hence, we can simulate from F by drawing a uniform $q \in (0, 1)$ and solving

$$F_i(y) = cq - Q_i, \quad \text{where } Q_i < cq \leq Q_{i+1},$$

that is,

$$y = \frac{1}{a_i} \log \left(a_i e^{-b_i} (cq - Q_i) + e^{a_i z_i} \right).$$

Where to place the knots t?

- Choose reasonable end-knots.
  - Plot unnormalized density q and eyeball reasonable values.
  - Or: use other envelope to simulate initial values and find, e.g., 10% and 90% empirical quantiles.
- Place knots equidistantly between the two end points.
- Or: place knots at equidistant empirical quantiles between 10% and 90%.

I use five knots at empirical quantiles of 10^6 simulated values.

```r
simulations <- sim_loop_cpp_wrap(10^6)
knots <- quantile(simulations, seq(0.1, 0.9, length.out = 5))
```

Implementation: highest level.

```
sim_adapt_more <- function(n, t)
{
    a <- dlogf(t)
    b <- logf(t) - a * t
    z <- get_z(a, b)
    Q_and_c <- get_Q_and_c(a, b, z)
    replicate(n, get_accepted_proposal(a, b, Q_and_c$c, z, Q_and_c$Q))
}

get_accepted_proposal <- function(a, b, c, z, Q)
{
    reject <- TRUE
    while(reject) {
        cu <- c * runif(1)
        i <- findInterval(cu, Q)
        proposal <- log(a[i] * exp(-b[i]) * (cu - Q[i]) +
                        exp(a[i] * z[i])) / a[i]
        reject <- log(runif(1)) > logf(proposal) - a[i] * proposal - b[i]
    }
    proposal
}
```

Implementation: z, Q, and c.

```
get_z <- function(a, b)
{
    N <- length(a)
    c(
        0,
        (b[2:N] - b[1:(N-1)]) / (a[1:(N-1)] - a[2:N]),
        1
    )
}

get_Q_and_c <- function(a, b, z)
{
    N <- length(a)
    Qc <- cumsum(c(0,
                   exp(b[1:N]) * (exp(a[1:N] * z[2:(N+1)])
                       - exp(a[1:N] * z[1:N])) / a[1:N]))
    list(
        Q = Qc[1:N],
        c = Qc[N+1]
    )
}
```

Profiling 10^6 simulations

- `logf` is the largest bottleneck.

```
get_accepted_proposal <- function(a, b, c, z, Q)                              20
{
    reject <- TRUE                                                            140
      while(reject) {                                                         160
          cu <- c * runif(1)                                                 3890
          i <- findInterval(cu, Q)                                           4090
          proposal <- log(a[i] * exp(-b[i]) * (cu - Q[i]) + exp(a[i] * z[i])) / a[i]   5020
          unif_random <- runif(1)                                            2200
          logf_proposal <- logf(proposal)                                   38520
          reject <- log(unif_random) > logf_proposal - a[i] * proposal - b[i]   2460
      }
    proposal                                                                   40
}
```

Not much to do about it in R:

```
logf <- function(y)
    y * sum_xz - sum(exp(y * x))
```

Rcpp implementation of adaptive envelope

```
// [[Rcpp::export]]
NumericVector
sim_adapt_more_cpp(unsigned long long n, const NumericVector& t,
                   const NumericVector& x, double sum_xz)
{
  unsigned long long n_t = t.length();
  double *a = get_a(t, n_t, x, sum_xz);
  double *b = get_b(a, t, n_t, x, sum_xz);
  double *z = get_z(a, b, n_t);
  double *Q_and_c = get_Q_and_c(a, b, z, n_t);
  double c = *(Q_and_c + n_t);
  NumericVector simulations(n);
  for (unsigned long long i = 0; i < n; i++)
    simulations[i] = get_accepted_proposal(a, b, c, z, Q_and_c,
                                           n_t, x, sum_xz);
  return simulations;
}
```

Rcpp implementation of adaptive envelope

```
double
get_accepted_proposal(double *a, double *b, double c,
                      double *z, double *Q, unsigned long long n_ab,
                      const NumericVector& x, double sum_xz)
{
  double cu, proposal;
  unsigned long long i;

  do {
    cu = c * R::runif(0, 1);
    i = find_interval(cu, Q, n_ab);
    proposal = log(*(a + i) * exp(-*(b + i)) * (cu - *(Q + i))
                  + exp(*(a + i) * *(z + i))) / *(a + i);
  } while (log(R::runif(0, 1)) >
          logf(proposal, x, sum_xz) - *(a + i) * proposal - *(b + i));

  return proposal;
}
```

Rcpp implementation of adaptive envelope

```
double
logf(double y, const NumericVector& x, double sum_xz)
{
  unsigned long long n = x.length();
  double result = y * sum_xz;

  for (unsigned long long i = 0; i < n; i++)
    result -= exp(y * x[i]);
  return result;
}

double
dlogf(double y, const NumericVector& x, double sum_xz)
{
  unsigned long long n = x.length();
  double result = sum_xz;

  for (unsigned long long i = 0; i < n; i++)
    result -= x[i] * exp(y * x[i]);
  return result;
}
```

Rcpp implementation of adaptive envelope

```
double *
get_a(const NumericVector& t, unsigned long long n_t, const NumericVector& x, double sum_xz)
{
  double *a = (double *) malloc(n_t * sizeof(double));

  for (unsigned long long i = 0; i < n_t; i++)
    *(a + i) = dlogf(t[i], x, sum_xz);
  return a;
}

double *
get_b(double *a, const NumericVector& t, unsigned long long n_t,
      const NumericVector& x, double sum_xz)
{
  double *b = (double *) malloc(n_t * sizeof(double));
  for (unsigned long long i = 0; i < n_t; i++)
    *(b + i) = logf(t[i], x, sum_xz) - *(a + i) * t[i];
  return b;
}

double *
get_z(double *a, double *b, unsigned long long n_ab)
{
  double *z = (double *) malloc((n_ab + 1) * sizeof(double));

  *z = 0;
  for (unsigned long long i = 1; i < n_ab; i++)
    *(z + i) = (*(b + i) - *(b + i - 1)) / (*(a + i - 1) - *(a + i));
  *(z + n_ab) = 1;
  return z;
}
```

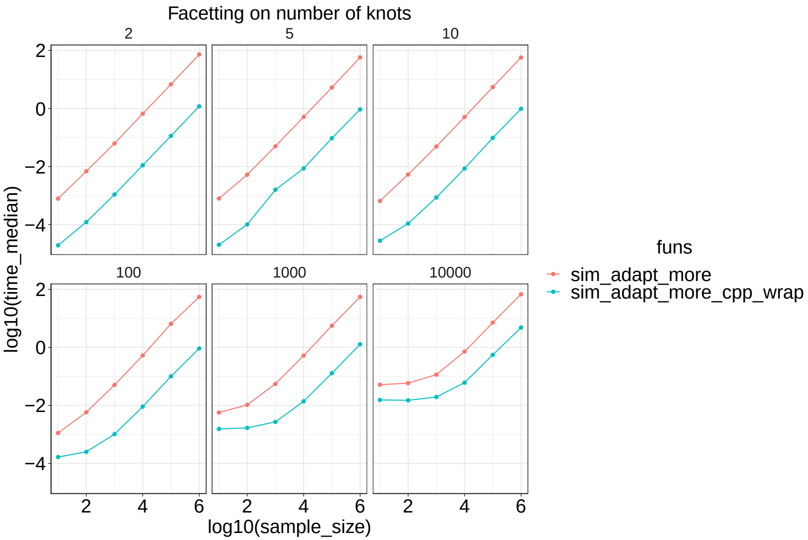
Rcpp implementation of adaptive envelope

```
/* Replaces contents of array with the cumulative sum */
void
replace_with_cumsum(double *array, unsigned long long array_length)
{
  for (unsigned long long i = 1; i < array_length; i++)
    *(array + i) += *(array + i - 1);
}

/* Returns pointer to array of length n_ab where element 0 is Q_0 = 0,
   element 1 is Q_1, and so on, until element n_ab which is Q_m = c */
double *
get_Q_and_c(double *a, double *b, double *z, unsigned long long n_ab)
{
  double *Q_and_c = (double *) malloc((n_ab + 1) * sizeof(double));

  *Q_and_c = 0;
  for (unsigned long long i = 0; i < n_ab; i++)
    *(Q_and_c + i + 1) = exp(*(b + i))
      * (exp(*(a + i) * *(z + i + 1))
         - exp(*(a + i) * *(z + i))) / *(a + i);
  replace_with_cumsum(Q_and_c, n_ab + 1);
  return Q_and_c;
}
```

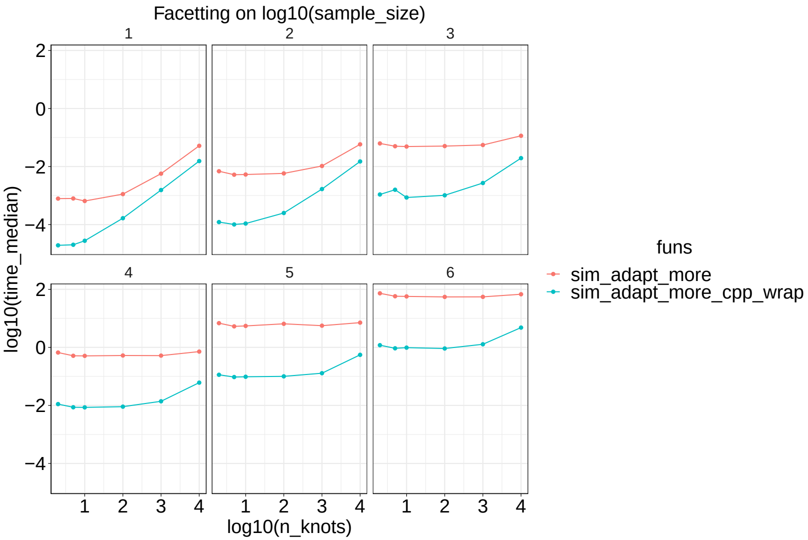
Rcpp implementation of adaptive envelope

```
unsigned long long
find_interval(double point, double *grid, unsigned long long n_grid)
{
  if (*(grid + n_grid - 1) < point)
    return n_grid - 1;

  for (unsigned long long i = 0; i < n_grid - 1; i++)
    if (*(grid + i) < point && point <= *(grid + i + 1))
      return i;

  return -1;                          /* Error code */
}
```

Benchmarking



Benchmarking



f is posterior in a Poisson regression with flat prior

If $Z_i \overset{\text{iid}}{\sim} \text{Poisson}(\lambda_i)$ where

$$\log \lambda_i = yx_i$$

then the likelihood is

$$\begin{aligned} p(z \mid x, y) &= \prod_{i=1}^{n} \frac{\exp(yx_i)^{z_i}}{(yx_i)!} \exp(-\exp(yx_i)) \\ &\propto \prod_{i=1}^{n} \exp(z_i yx_i - \exp(yx_i)) \end{aligned}$$

so with a flat prior $p(y) \propto 1$ the posterior for Y is

$$p(y \mid z, x) \propto p(z \mid x, y)p(y) \propto p(z \mid x, y).$$

Simulating from posterior with Stan

- Writing model in Stan:

```
data {
  int<lower = 0> N;
  int<lower = 0> z[N];
  real x[N];
}
parameters {
  real y;
}
model {
  z ~ poisson_log(y * x);
}
```

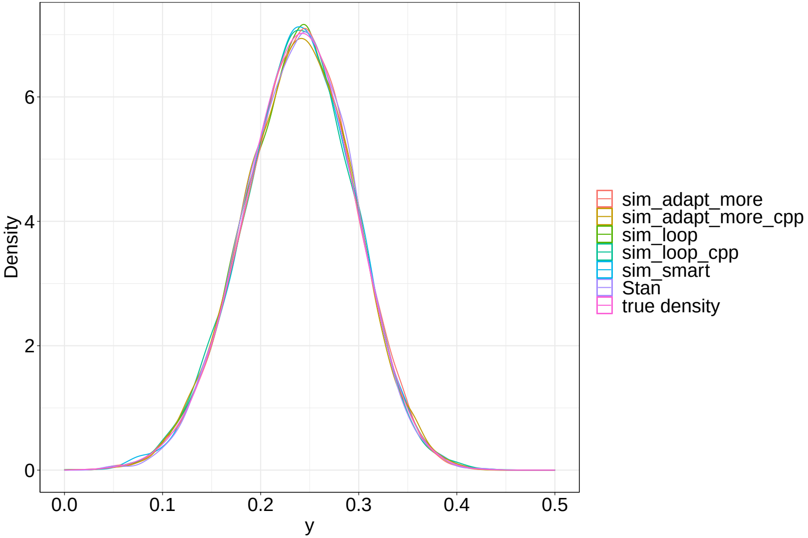
- Simulating from Posterior:

```
stan_sim <- rstan::stan(file = "model.stan",
                        data = list(N = nrow(dat),
                                    z = dat$z,
                                    x = dat$x),
                        chains = 4, cores = 4, iter = 1e4 * 2 / 4)
y_stan_sim <- rstan::extract(stan_sim)$y
```

Comparing simulated values to true density

Density estimates for 10^4 simulated values.

The true density is q normalized by its numerical integral.



What else could be done?

- Adaptive choice of knots for the log-affine envelope (each time a proposal is rejected, add a knot at the proposed value) (notes 3 slides from now).
- Explore rejection rates depending on number- and placement of knots (fun but not essential when our primary interest is runtime).

Other slides

- Use binary search instead of brute force search for large number of knots? (1 slide from now).
- Generic implementation for other densities. (2 slides from now).
- Check Stan convergence (4 slides from now)

Binary search instead of brute force

```c
unsigned long long
find_interval_binary_search(double point, double *grid,
                            unsigned long long n_grid)
{
  unsigned long long i_min = 0;
  unsigned long long i_max = n_grid - 1;
  unsigned long long i;

  if (*(grid + i_max) <= point)
    return i_max;

  while (i_max - i_min > 1) {
    i = floor((i_max + i_min) / 2);
    if (*(grid + i) == point)
      return i;
    else if (*(grid + i) > point)
      i_max = i;
    else
      i_min = i;
  }

  return i_min;
}
```

Slower than brute force for small grids (benchmarked).

Generic implementation for other densities

```
## The user has to supply logf.
## This only applies to get_accepted_proposal and sim_adapt_more.
get_accepted_proposal_generic <- function(a, b, c, z, Q, logf)
{
    reject <- TRUE
    while(reject) {
        cu <- c * runif(1)
        i <- findInterval(cu, Q)
        proposal <- log(a[i] * exp(-b[i]) * (cu - Q[i]) + exp(a[i] * z[i])) / a[i]
        reject <- log(runif(1)) > logf(proposal) - a[i] * proposal - b[i]
    }
    proposal
}

## dlogf is an optional argument.
## If it is NULL, then we use numerical differentiation.
sim_adapt_more_generic <- function(n, t, logf, dlogf = NULL)
{
    if (is.null(dlogf))
        a <- numDeriv::grad(logf, t, method = "simple")
    else
        a <- dlogf(t)
    b <- logf(t) - a * t
    z <- get_z(a, b)
    Q_and_c <- get_Q_and_c(a, b, z)
    replicate(n, get_accepted_proposal(a, b, Q_and_c$c, z, Q_and_c$Q, logf))
}
```

Adaptive knots in

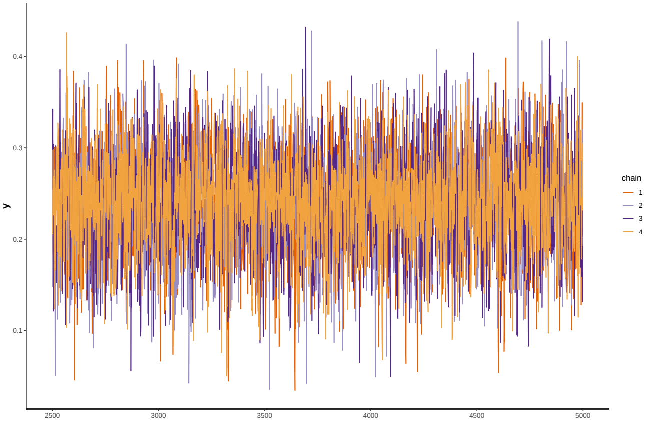
R

- Easy to add new knots in the middle of vector.

C++

- Linked list of knots should make it easy/cheap to add new knots adaptively.

Checking Stan



The trace plot shows good mixing and we get $\hat{R} \approx 1$, which indicates convergence of the chains.

The effective sample size for Y is 3930 (out of 10000 post-warmup draws in total).