Adaptive Algorithms

13008341

13/07/2020

Adaptive Algorithms

In this document, we implement and test the following algorithms

- Adaptive Metropolis From Haario et al. (2001)
- Componentwise AM + Componentwise adaptive scaling
- Global AM + Componentwise adaptive scaling
- Localised N-SRWM
- Principal components Metropolis update

The structure of these are taken from 'A Tutorial on Adaptive MCMC' by Andrieu and Moulines

Pseudocode Descriptions

Initialise X_0, μ_0, and Σ_0;
At iteration $i+1$ given X_i, μ_i, and Σ_i;
1. Sample $X_{i+1} \sim P^{SRWM}_{\mu_i,\Sigma_i}(X_i,)$;
2. Update $\mu_{i+1} = \mu_i + \gamma_{i+1}(X_{i+1} - \mu_i)$;
$\Sigma_{i+1} = \Sigma_i + \gamma_{i+1}((X_{i+1} - \mu_i)(X_{i+1} - \mu_i)^T - \Sigma_i)$;

Algorithm 1: Adaptive Metropolis

Initialise X_0, μ_0, Σ_0, and λ_0^1,...,λ_0^n;
At iteration $i+1$ given X_i, μ_i, Σ_i, and λ_i^1,...,λ_i^n;
1. Choose a component $k \sim U\{1, ..., n\}$;
2. Sample $Y_{i+1} \sim X_i + e_k N(0, \lambda_i^k (\Sigma_i)_{k,k})$;
3. Accept with probability $\alpha = min\{1, \frac{\pi(Y_{i+1})}{\pi(X_i)}\}$ otherwise stay put.;
4. Update;
i) $\mu_{i+1} = \mu_i + \gamma_{i+1}(X_{i+1} - \mu_i)$;
ii) $\Sigma_{i+1} = \Sigma_i + \gamma_{i+1}((X_{i+1} - \mu_i)(X_{i+1} - \mu_i)^T - \Sigma_i)$;
iii) $log(\lambda_{i+1}^k) = log(\lambda_i^k) + \gamma_{i+1}(\alpha - \alpha_*)$;
iv) $\lambda_{i+1}^j = \lambda_i^j$ for $j \neq k$

Algorithm 2: Componentwise AM + Componentwise adaptive scaling

where n is the dimension of the state, e_k is a vector of dimension n with a 1 in the kth position and zeros

everywhere else. $(M)_{i,j}$ is the (i,j) - th element of a matrix M, and α_* is the optimal acceptance rate.

Initialise X_0, μ_0, Σ_0, and λ_0^1,...,λ_0^n;
At iteration $i+1$ given X_i, μ_i, Σ_i, and λ_i^1,...,λ_i^n;

1. Sample $Z_{i+1} \sim N(0, \Lambda_i^{\frac{1}{2}} \Sigma_i \Lambda_i^{\frac{1}{2}})$ where $\Lambda_i = diag(\lambda_i^1, ..., \lambda_i^n)$;
2. Set $X_{i+1} = X_i + Z_{i+1}$ with probability $min\{1, \frac{\pi(X_{i+1})}{\pi(X_i)}\}$ otherwise stay put.;
3. **for** *k in 1,...,n* **do**
   $\mu_{i+1}^k = \mu_i^k + \gamma_{i+1}(X_{i+1} - \mu_i^k)$;
   $\Sigma_{i+1}^k = \Sigma_i^k + \gamma_{i+1}((X_{i+1} - \mu_i^k)(X_{i+1} - \mu_i^k)^T - \Sigma_i^k)$;
   $log(\lambda_{i+1}^k) = log(\lambda_i^k) + \gamma_{i+1}(min\{1, \frac{\pi(X_i + Z_{i+1}^T e_k)}{\pi(X_i)}\} - \alpha_*)$;

end

Algorithm 3: Global AM + Componentwise adaptive scaling

Initialise X_0, $\mu_0^{1:s}$, $\Sigma_0^{1:s}$, $\lambda_0^{1:s}$, and $w_0^{1:s}$.;
At iteration $i+1$ given X_i, $\mu_i^{1:s}$, $\Sigma_i^{1:s}$, and $\lambda_i^{1:s}$;

1. Sample $Z_{i+1} \sim \hat{q}_{\theta_i}(Z = k|X_i)$ and $Y_{i+1} \sim N(X_i, \lambda_i^{Z_{i+1}} \Sigma_i^{Z_{i+1}})$;
2. Set $X_{i+1} = Y_{i+1}$ with probability $\alpha_k = min\{1, \frac{\pi(Y_{i+1})\hat{q}_{\theta_i}(k|Y_{i+1})}{\pi(X_i)\hat{q}_{\theta_i}(k|X_i)}\}$ otherwise stay put;
3. **for** *k in 1,...,s* **do**
   $\mu_{i+1}^k = \mu_i^k + \gamma_{i+1}\hat{q}_{\theta_i}(Z_{i+1} = k|X_i)(X_{i+1} - \mu_i^k)$;
   $\Sigma_{i+1}^k = \Sigma_i^k + \gamma_{i+1}\hat{q}_{\theta_i}(Z_{i+1} = k|X_i)((X_{i+1} - \mu_i^k)(X_{i+1} - \mu_i^k)^T - \Sigma_i^k)$;
   $w_{i+1}^k = w_i^k + \gamma_{i+1}(\hat{q}_{\theta_i}(Z_{i+1} = k|X_i) - w_i^k)$ $log(\lambda_{i+1}^k) = log(\lambda_i^k) + \gamma_{i+1}\mathbb{I}\{Z_{i+1} = k\}(\alpha_k - \alpha_*)$;

end

Algorithm 4: Localised RWM

where

$$\hat{q}_{\theta_i}(Z_{i+1} = k|x) := \frac{w_i^k N(x; \mu_i^k, \Sigma_i^k)}{\hat{q}_{\theta_i}(x)}$$

and

$$\hat{q}_{\theta_i}(x) := \sum_{k=1}^{s} w_i^k N(x; \mu_i^k, \Sigma_i^k)$$

with $N(x; \mu, \Sigma)$ the density of a normal distn. at x

W is a matrix whose columns ($w(l)$ denotes the lth column) are what we estimate to be the first m eigenvectors of Σ_π the covariance matrix of the target distribution. These columns have corresponding eigenvalues $\rho(l)$, and l_i is a vector of scale factors.

At iteration $i+1$ given X_i, (l_i, ρ_i, W_i);
1. Sample an update direction $l \sim (d(1), ..., d(m))$;
2. Sample $Z_{i+1} \sim N(0, l_i(l)\rho_i(l))$ and set $Y_{i+1} = X_{i+1} + Z_{i+1}w(l)$;
3. Set $X_{i+1} = Y_{i+1}$ with probability $min\{1, \frac{\pi(Y_{i+1})}{\pi(X_i)}\}$, stay put otherwise;
4. Update (l_i, ρ_i, W_i) to $(l_{i+1}, \rho_{i+1}, W_{i+1})$ in light of X_{i+1}

Algorithm 5: Principle Components Metropolis update

In all algo.s there are the following parts:

- Initial distributions of parameters + state
- Sampling step
- Update step

Therefore, for each of the final two parts, we should write a bespoke function for each type of adaptive algorithm.

```
adapt <- function(init,
                  sample,
                  update,
                  logpi,
                  logpi_args,
                  seed = 2000,
                  learn_in,
                  nits = 5000,
                  gamma,
                  stop_after,
                  m = 2) {
  # Constructs a chain using an adaptive MCMC algorithm.
  # init: initial state and parameter values
  # sample: function for the sample step
  # update: function for the update step
  # logpi: logarithm of the posterior density
  # logpi_args: list of named arguments for the logpi function
  # seed: hold constant for replicatable results
  # learn_in: period in which only sample step is applied
  # nits: number of iterations to run for (including the learn in)
  # gamma: a function which returns the next value of gamma
  # stop_after: number of iterations to stop adapting after
  # m: the number of eigenvectors estimated in the PCA update

  # Timing
  start_time <- Sys.time()

  # set the seed
  set.seed(seed)

  # initial values
  x_curr <- init[['X']]
  p_curr <- init[['params']]
  gamma_curr <- init[['gamma']]

  # get the dimension, and number of weights (for localised RWM)
  dim <- length(x_curr)
  if (sample == 'Local_RWM_sample') {
    k_max <- nrow(p_curr[['mus']])
  }

  # initialise the acceptance rate, the chain, and the current
  # value of logpi
  accepted <- 0
  x_store <- matrix(nrow = nits, ncol = dim)
  arg_name <- formalArgs(logpi)[1]
  logpi_args[[arg_name]] <- x_curr

  logpi_curr <- do.call(logpi, logpi_args)

  # make a place to store the mus, sigmas, and 'acceptedness'
  if (sample == 'Local_RWM_sample') {
    mus <- matrix(nrow = nits * k_max, ncol = dim)
```

```
  sigmas <- matrix(nrow = nits * dim * k_max, ncol = dim)
} else {
  mus <- matrix(nrow = nits, ncol = dim)
  sigmas <- matrix(nrow = nits * dim, ncol = dim)
}
as <- matrix(nrow = nits, ncol = 1)

# make a place to store the eigenvectors if using a PCA update
if (sample == 'PCA_sample') {
  e_vectors <- matrix(nrow = nits * m, ncol = dim)
} else {
  e_vectors <- matrix(nrow = 0, ncol = 1)
}
# make a place to store the weights if using local RWM
if (sample == 'Local_RWM_sample') {
  weights <- matrix(nrow = nits, ncol = k_max)
} else {
  weights <- matrix(nrow = 0, ncol = 1)
}
# make a place to store the lambdas if using CAM-CAS
if (sample == 'CAM_CAS_sample') {
  lambdas <- matrix(nrow = nits, ncol = length(p_curr[['lambdas']]))
} else if (sample == 'AM_GAS_sample') {
  lambdas <- matrix(nrow = nits, ncol = 1)
} else {
  lambdas <- matrix(nrow = 0, ncol = 1)
}

for (i in 1:nits) {
  # sample step
  if (sample == "AM_sample") {
    samp <- AM_sample(x_curr = x_curr,
                      logpi = logpi,
                      logpi_args = logpi_args,
                      logpi_curr = logpi_curr,
                      p_curr = p_curr)
  } else if (sample == "CAM_CAS_sample") {
      samp <- CAM_CAS_sample(x_curr = x_curr,
                             logpi = logpi,
                             logpi_args = logpi_args,
                             logpi_curr = logpi_curr,
                             p_curr = p_curr)

      p_curr[['loga']] <- samp[['loga']]
      p_curr[['k']] <- samp[['k']]

  } else if (sample == "GAM_CAS_sample") {
      samp <- GAM_CAS_sample(x_curr = x_curr,
                             logpi = logpi,
                             logpi_args = logpi_args,
                             logpi_curr = logpi_curr,
                             p_curr = p_curr)
```

```
    p_curr[['old_x']] <- samp[['old_x']]
    p_curr[['z']] <- samp[['z']]
    p_curr[['old_logpi']] <- samp[['old_logpi']]

} else if (sample == "Local_RWM_sample") {
    samp <- Local_RWM_sample(x_curr = x_curr,
                             logpi = logpi,
                             logpi_args = logpi_args,
                             logpi_curr,
                             p_curr)

    p_curr[['z']] <- samp[['z']]
    p_curr[['densities']] <- samp[['densities']]
    p_curr[['loga']] <- samp[['loga']]

} else if (sample == 'PCA_sample') {
    samp <- PCA_sample(x_curr,
                       logpi,
                       logpi_args,
                       logpi_curr,
                       p_curr)
    p_curr[['direction_number']] <- samp[['direction_number']]
    p_curr[['loga']] <- samp[['loga']]
} else if (sample == "AM_GAS_sample") {
  # AM_GAS
  samp <- AM_GAS_sample(x_curr,
                        logpi,
                        logpi_args,
                        logpi_curr,
                        p_curr)
  p_curr[['loga']] <- samp[['loga']]
} else {
  # LPM_RWM
  samp <- LPM_RWM_sample(x_curr = x_curr,
                     logpi = logpi,
                     logpi_args = logpi_args,
                     logpi_curr = logpi_curr,
                     p_curr = p_curr)
}

x_temp <- samp[['x']]
logpi_curr <- samp[['logpi']]

accepted <- (accepted / i) * (i - 1 + as.numeric(x_temp == x_curr))
x_curr <- x_temp

# store the state, the acceptedness, and the parameters used to sample
# the state
x_store[i, ] <- x_curr
if (sample == 'Local_RWM_sample') {
  # for local RWM there are k_max means and k_max sigmas etc.
  # you don't actually need the loop, since p_curr[['mus']] is a matrix
  for (s in 1:k_max) {
```

```
      mus[k_max * (i - 1) + s, ] <- p_curr[['mus']][s, ]
    }
    sigmas[(dim * k_max * (i - 1) + 1):(dim * k_max * i), ] <- p_curr[['sigmas']]
  } else {
    mus[i, ] <- p_curr[['mu']]
    as[i, 1] <- matrix(as.numeric(x_temp == x_curr), nrow = 1, ncol = 1)

    sigmas[(dim * (i - 1) + 1):(i * dim), ] <- p_curr[['sigma']]
  }

  # store the eigenvectors if using the PCA update
  if (sample == 'PCA_sample') {
    e_vectors[(m * (i - 1) + 1):(m * i), ] <- p_curr[['e_vectors']]
  }
  # store the weights if using local RWM
  if (sample == 'Local_RWM_sample') {
    weights[i, ] <- p_curr[['weights']]
  }
  # store the lambdas if using CAM_CAS
  if (sample == 'CAM_CAS_sample') {
    lambdas[i, ] <- p_curr[['lambdas']]
  }
  # store the scale if using AM_GAS
  if (sample == 'AM_GAS_sample') {
    lambdas[i, ] <- p_curr[['scale']]
  }

  if (i > learn_in) {
    # adaptive step
    gamma_curr <- gamma(current = gamma_curr,
                        iteration = i)
    if (update == "AM_update") {
      if (i == learn_in + 1 && learn_in != 0) {
      # use empirical covar. and mean from initial sample:
      p_curr[['mu']] <- vector(mode = "numeric", length = dim)
      for (j in 1:dim) {
        p_curr[['mu']][j] <- mean(x_store[, j], na.rm = TRUE)
      }

      # scale sigma appropriately and perturb slightly to avoid singularity:
      scale <- 2.4 ^ 2 / dim
      epsilon <- 0.001
      p_curr[['sigma']] <- scale * (cov(x_store[1:i, ]) + epsilon * diag(dim))
      }
      p_curr <- AM_update(x_curr = x_curr,
                          p_curr = p_curr,
                          gamma_curr = gamma_curr)
    } else if (update == "CAM_CAS_update") {
      if (i == learn_in + 1 && learn_in != 0) {
      # use empirical covar. and mean from initial sample:
      p_curr[['mu']] <- vector(mode = "numeric", length = dim)
      for (j in 1:dim) {
        p_curr[['mu']][j] <- mean(x_store[, j], na.rm = TRUE)
```

```
    }

    # scale sigma appropriately and perturb slightly to avoid singularity:
    scale <- 2.4 ^ 2 / dim
    epsilon <- 0.001
    p_curr[['sigma']] <- scale * (cov(x_store[1:i, ]) + epsilon * diag(dim))
    }
    p_curr <- CAM_CAS_update(x_curr = x_curr,
                             p_curr = p_curr,
                             loga_curr = p_curr[['loga']],
                             optimal_a = 0.234,
                             k_curr = p_curr[['k']],
                             gamma_curr = gamma_curr)
  } else if (update == "GAM_CAS_update") {
    if (i == learn_in + 1 && learn_in != 0) {
    # use empirical covar. and mean from initial sample:
    p_curr[['mu']] <- vector(mode = "numeric", length = dim)
    for (j in 1:dim) {
      p_curr[['mu']][j] <- mean(x_store[, j], na.rm = TRUE)
    }

    # scale sigma appropriately and perturb slightly to avoid singularity:
    scale <- 2.4 ^ 2 / dim
    epsilon <- 0.001
    p_curr[['sigma']] <- scale * (cov(x_store[1:i, ]) + epsilon * diag(dim))
    }
    p_curr <- GAM_CAS_update(x_curr = x_curr,
                             p_curr = p_curr,
                             optimal_a = 0.234,
                             gamma_curr = gamma_curr,
                             logpi = logpi,
                             logpi_args = logpi_args)
  } else if (update == "Local_RWM_update") {
    p_curr <- Local_RWM_update(x_curr = x_curr,
                               p_curr = p_curr,
                               optimal_a = 0.234,
                               gamma_curr = gamma_curr)
  } else if (update == 'PCA_update') {
    if (i == learn_in + 1 && learn_in != 0) {
      # use empirical covar. and mean from initial sample:
      p_curr[['mu']] <- vector(mode = "numeric", length = dim)
      for (j in 1:dim) {
        p_curr[['mu']][j] <- mean(x_store[, j], na.rm = TRUE)
      }

      # scale sigma appropriately and perturb slightly to avoid singularity:
      scale <- 2.4 ^ 2 / dim
      epsilon <- 0.001
      p_curr[['sigma']] <- scale * (cov(x_store[1:i, ]) + epsilon * diag(dim))
    }
    p_curr <- PCA_update(x_curr = x_curr,
                         p_curr = p_curr,
                         optimal_a = 0.234,
```

```
                            gamma_curr = gamma_curr)
      } else if (update == "AM_GAS_update") {
        # AM_GAS_update
        if (i == learn_in + 1 && learn_in != 0) {
          # use empirical covar. and mean from initial sample:
          p_curr[['mu']] <- vector(mode = "numeric", length = dim)
          for (j in 1:dim) {
            p_curr[['mu']][j] <- mean(x_store[, j], na.rm = TRUE)
          }

          # scale sigma appropriately and perturb slightly to avoid singularity:
          scale <- 2.4 ^ 2 / dim
          epsilon <- 0.001
          p_curr[['sigma']] <- scale * (cov(x_store[1:i, ]) + epsilon * diag(dim))
        }
        p_curr <- AM_GAS_update(x_curr = x_curr,
                            p_curr = p_curr,
                            optimal_a = 0.234,
                            gamma_curr = gamma_curr)
      }
    }
  }
  # Timing
  end_time <- Sys.time()
  return(list(x_store = x_store,
              mus = mus,
              as = as,
              sigmas = sigmas,
              e_vectors = e_vectors,
              weights = weights,
              lambdas = lambdas,
              time = end_time - start_time))
}
```

```
LPM_RWM_sample <- function(x_curr,
                       logpi,
                       logpi_args,
                       logpi_curr,
                       p_curr) {
  sigma <- p_curr[['sigma']]
  x_prop <- rmvnorm(n = 1,
                    mean = x_curr,
                    sigma = sigma)
  # tau has to be in [0, 1]
  if (x_prop[n * d + 1] < 0) {
    # reflect into the [0, 1] interval by the non-integral amount (i.e. if tau is
    # -9.4, push it into the interval by 0.4)
    x_prop[n * d + 1] <- - x_prop[n * d + 1] - floor(- x_prop[n * d + 1])
  } else if (x_prop[n * d + 1] > 1) {
    # similarly in this case
    x_prop[n * d + 1] <- 1 - (x_prop[n * d + 1] - floor(x_prop[n * d + 1]))
  }
  # gamma2 has to be +ve
  if (x_prop[n * d + 2] < 0) {
```

```
    # reflect about the origin
    x_prop[n * d + 2] <- - x_prop[n * d + 2]
  }

  loga_list <- loga(logpi = logpi,
                    logpi_args = logpi_args,
                    x_prop = x_prop,
                    logpi_curr = logpi_curr)

  loga <- loga_list[['loga']]
  logpi_prop <- loga_list[['logpi_prop']]

  u <- runif(1)

  if (log(u) < loga) {
    return(list(x = x_prop, logpi = logpi_prop))
  } else {
    return(list(x = x_curr, logpi = logpi_curr))
  }
}
```

```
AM_sample <- function(x_curr,
                      logpi,
                      logpi_args,
                      logpi_curr,
                      p_curr) {
  sigma <- p_curr[['sigma']]
  x_prop <- rmvnorm(n = 1,
                    mean = x_curr,
                    sigma = sigma)
  # Haario et al 2001 says the mean is the current point. So then why update mu?
  # ^ So you can update sigma.

  # # work out the density of the the proposed point:
  # arg_name <- formalArgs(logpi)[1]
  # logpi_args[[arg_name]] <- x_prop
  #
  # logpi_prop <- do.call(logpi, logpi_args)
  #
  # # accept-reject
  # loga <- logpi_prop - logpi_curr

  loga_list <- loga(logpi = logpi,
                    logpi_args = logpi_args,
                    x_prop = x_prop,
                    logpi_curr = logpi_curr)

  loga <- loga_list[['loga']]
  logpi_prop <- loga_list[['logpi_prop']]

  u <- runif(1)

  if (log(u) < loga) {
    return(list(x = x_prop, logpi = logpi_prop))
```

```
  } else {
    return(list(x = x_curr, logpi = logpi_curr))
  }
}
```

```
AM_update <- function(x_curr,
                      p_curr,
                      gamma_curr) {
  mu_new <- p_curr[['mu']] + gamma_curr * (x_curr - p_curr[['mu']])

  sig_step <- outer_prod(x_curr - p_curr[['mu']], x_curr - p_curr[['mu']])
  sig_new <- p_curr[['sigma']] + gamma_curr * (sig_step - p_curr[['sigma']])

  return(list(mu = mu_new, sigma = sig_new))
}
```

```
CAM_CAS_sample <- function(x_curr,
                           logpi,
                           logpi_args,
                           logpi_curr,
                           p_curr) {
  # Componentwise AM + Componentwise adaptive scaling sampler
  k <- floor(runif(n = 1, min = 1, max = length(x_curr) + 1))

  x_prop <- x_curr

  sigma <- p_curr[['lambdas']][k] * p_curr[['sigma']][k, k]
  x_prop[k] <- x_prop[k] + rnorm(n = 1,
                                 sd = sqrt(sigma))


  # # work out the density of the the proposed point:
  # arg_name <- formalArgs(logpi)[1]
  # logpi_args[[arg_name]] <- x_prop
  #
  # logpi_prop <- do.call(logpi, logpi_args)
  #
  # # accept-reject
  # loga <- logpi_prop - logpi_curr

  loga_list <- loga(logpi = logpi,
                    logpi_args = logpi_args,
                    x_prop = x_prop,
                    logpi_curr = logpi_curr)

  loga <- loga_list[['loga']]
  logpi_prop <- loga_list[['logpi_prop']]

  u <- runif(1)

  if (log(u) < loga) {
    return(list(x = x_prop, logpi = logpi_prop, loga = loga, k = k))
  } else {
    return(list(x = x_curr, logpi = logpi_curr, loga = loga, k = k))
```

```
  }
}
```

```
CAM_CAS_update <- function(x_curr,
                           p_curr,
                           loga_curr,
                           optimal_a,
                           k_curr,
                           gamma_curr) {
  lambda_new <- exp(log(p_curr[['lambdas']][k_curr]) +
                                      gamma_curr * (min(exp(loga_curr), 1) - optimal_a))
  lambdas_new <- p_curr[['lambdas']]
  lambdas_new[k_curr] <- lambda_new

  mu_new <- p_curr[['mu']] + gamma_curr * (x_curr - p_curr[['mu']])

  sig_step <- outer_prod(x_curr - p_curr[['mu']], x_curr - p_curr[['mu']])
  sig_new <- p_curr[['sigma']] + gamma_curr * (sig_step - p_curr[['sigma']])

  return(list(mu = mu_new, sigma = sig_new, lambdas = lambdas_new))
}
```

```
GAM_CAS_sample <- function(x_curr,
                           logpi,
                           logpi_args,
                           logpi_curr,
                           p_curr) {
  reduced_lambda <- diag(sqrt(p_curr[['lambdas']]))
  sigma <- reduced_lambda %*% p_curr[['sigma']] %*% reduced_lambda
  z <- rmvnorm(n = 1,
               sigma = sigma)

  x_prop <- x_curr + z

  # # work out the density of the the proposed point:
  # arg_name <- formalArgs(logpi)[1]
  # logpi_args[[arg_name]] <- x_prop
  #
  # logpi_prop <- do.call(logpi, logpi_args)
  #
  # # accept-reject
  # loga <- logpi_prop - logpi_curr
  loga_list <- loga(logpi = logpi,
               logpi_args = logpi_args,
               x_prop = x_prop,
               logpi_curr = logpi_curr)

  loga <- loga_list[['loga']]
  logpi_prop <- loga_list[['logpi_prop']]

  u <- runif(1)

  if (log(u) < loga) {
    return(list(x = x_prop,
```

```
                logpi = logpi_prop,
                old_x = x_curr,
                z = z,
                old_logpi = logpi_curr))
  } else {
    return(list(x = x_curr,
                logpi = logpi_curr,
                old_x = x_curr,
                z = z,
                old_logpi = logpi_curr))
  }
}
```

```
GAM_CAS_update <- function(x_curr,
                           p_curr,
                           optimal_a,
                           gamma_curr,
                           logpi,
                           logpi_args) {
  new_lambdas <- vector(mode = "numeric", length = length(x_curr))
  for (k in 1:length(x_curr)) {
    component_update <- p_curr[['old_x']]
    component_update[k] <- component_update[k] + p_curr[['z']][k]

    # work out the density of the proposed componentwise update:
    arg_name <- formalArgs(logpi)[1]
    logpi_args[[arg_name]] <- component_update

    logpi_prop <- do.call(logpi, logpi_args)

    # and the acceptance prob.
    acceptance <- min(exp(logpi_prop - p_curr[['old_logpi']]), 1)

    # now update:
    new_lambdas[k] <- exp(log(p_curr[['lambdas']][k]) +
                          gamma_curr * (acceptance - optimal_a))
  }
  new_mu <- p_curr[['mu']] + gamma_curr * (x_curr - p_curr[['mu']])

  sig_step <- outer_prod(x_curr - p_curr[['mu']], x_curr - p_curr[['mu']])
  sig_new <- p_curr[['sigma']] + gamma_curr * (sig_step - p_curr[['sigma']])

  return(list(mu = new_mu,
              sigma = sig_new,
              lambdas = new_lambdas,
              old_x = p_curr[['old_x']],
              z = p_curr[['z']],
              old_logpi = p_curr[['old_logpi']]))
}
```

```
Local_RWM_sample <- function(x_curr,
                             logpi,
                             logpi_args,
                             logpi_curr,
```

```
                            p_curr) {
# Create the distribution of Z
weights <- p_curr[['weights']]
mus <- p_curr[['mus']]
sigmas <- p_curr[['sigmas']]
lambdas <- p_curr[['lambdas']]

dim <- length(x_curr)

# vector to hold the normal densities
norms <- vector(mode = "numeric", length = length(weights))
for (i in 1:length(weights)) {
  density <- dmvnorm(x = x_curr,
                     mean = mus[i, ],
                     sigma = sigmas[(dim * i - dim + 1):(dim * i), ])

  norms[i] <- density
}

z_weights <- norms * weights
marginal <- sum(z_weights)
z_weights_normd <- z_weights / marginal

z <- sample_int_R(n = length(z_weights),
                  size = 1,
                  prob = z_weights_normd)

# Keep the density of the particular z to use in the update step
density_z <- z_weights_normd[z]

x_prop <- rmvnorm(n = 1,
                  mean = mus[z, ],
                  sigma = sigmas[(dim * z - dim + 1):(dim * z), ])

# work out the conditional density of the proposed point:
arg_name <- formalArgs(logpi)[1]
logpi_args[[arg_name]] <- x_prop

logpi_prop <- do.call(logpi, logpi_args)

# need to do the same with the proposed point as we did with the current
# point above to work out the conditional ratio (or the log difference)
# (this is stupid, just make a function and call it twice, duh)

# Is it that stupid? Making a function and only calling it twice doesn't
# save that many lines of code, and the code is more expository as it is.

# vector to hold the normal densities
norms_y <- vector(mode = "numeric", length = length(weights))
for (i in 1:length(weights)) {
  density <- dmvnorm(x = x_prop,
                     mean = mus[i, ],
                     sigma = sigmas[(dim * i - dim + 1):(dim * i), ])
```

```
  norms_y[i] <- density
}

z_weights_y <- norms_y * weights
marginal_y <- sum(z_weights_y)
z_weights_normd_y <- z_weights_y / marginal_y

z_y <- sample_int_R(n = length(z_weights_y),
                    size = 1,
                    prob = z_weights_normd_y)

# Keep the density of the particular z_y to use in the update step
density_z_y <- z_weights_normd_y[z_y]

# accept-reject using modified detail balance condition
loga <- logpi_prop - logpi_curr + log(density_z_y) - log(density)
u <- runif(1)

# need to calculate all the densities given the new point and return
# them in the list
if (log(u) < loga) {
  new_point <- x_prop

  # vector to hold the normal densities
  new_norms <- vector(mode = "numeric", length = length(weights))
  for (i in 1:length(weights)) {
    density <- dmvnorm(x = new_point,
                       mean = mus[i, ],
                       sigma = sigmas[(dim * i - dim + 1):(dim * i), ])

    new_norms[i] <- density
  }

  new_z_weights <- new_norms * weights
  new_marginal <- sum(new_z_weights)
  new_z_weights_normd <- new_z_weights / new_marginal

  return(list(x = new_point,
              logpi = logpi_prop,
              densities = new_z_weights_normd,
              z = z,
              loga = loga))
} else {
  new_point <- x_curr

  # vector to hold the normal densities
  new_norms <- vector(mode = "numeric", length = length(weights))
  for (i in 1:length(weights)) {
    density <- dmvnorm(x = new_point,
                       mean = mus[i, ],
                       sigma = sigmas[(dim * i - dim + 1):(dim * i), ])

    new_norms[i] <- density
```

```
    }

    new_z_weights <- new_norms * weights
    new_marginal <- sum(new_z_weights)
    new_z_weights_normd <- new_z_weights / new_marginal
    return(list(x = new_point,
                logpi = logpi_curr,
                densities = new_z_weights_normd,
                z = z,
                loga = loga))
  }
}
```

```
Local_RWM_update <- function(x_curr,
                             p_curr,
                             optimal_a,
                             gamma_curr) {
  dim <- length(x_curr)
  densities <- p_curr[['densities']]
  z <- p_curr[['z']]

  # Create placeholder variables to store the next iterations
  mus <- matrix(nrow = nrow(p_curr[['mus']]), ncol = dim)
  sigmas <- matrix(nrow = dim * nrow(p_curr[['mus']]), ncol = dim)
  weights <- rep(NA, nrow(p_curr[['mus']]))
  lambdas <- p_curr[['lambdas']]
  alphas <- p_curr[['alphas']]

  # now adapt for all k
  for (k in 1:length(weights)) {
    kth_density <- densities[k]
    mus[k, ] <- p_curr[['mus']][k, ] + gamma_curr * kth_density * (x_curr - p_curr[['mus']][k, ])

    sig_step <- outer_prod(x_curr - p_curr[['mus']][k, ], x_curr - p_curr[['mus']][k, ])
    sig_new <- p_curr[['sigmas']][(dim * (k - 1) + 1):(dim * k), ] + gamma_curr * kth_density * (sig_st

    sigmas[(dim * (k - 1) + 1):(dim * k), ] <- sig_new

    weights[k] <- p_curr[['weights']][k] + gamma_curr * (kth_density - p_curr[['weights']][k])

    if (k == z) {
      acc <- min(exp(p_curr[['loga']]), 1)

      alphas[k] <- alphas[k] + gamma_curr * (acc - optimal_a)
      lambdas[k] <- lambdas[k] + gamma_curr * (acc - lambdas[k])
    }
  }
  # renormalize weights
  weights <- weights / sum(weights)
  return(list(mus = mus,
              sigmas = sigmas,
              weights = weights,
              lambdas = lambdas,
              alphas = alphas,
```

```
              z = z,
              densities = densities))
}
```

```
PCA_sample <- function(x_curr,
                       logpi,
                       logpi_args,
                       logpi_curr,
                       p_curr) {
  sigma <- p_curr[['sigma']]
  scales <- p_curr[['scales']]
  e_vectors <- p_curr[['e_vectors']]
  e_values <- p_curr[['e_values']]
  m <- p_curr[['m']]

  direction_number <- sample(m, 1)
  direction <- e_vectors[direction_number, ]
  scale <- scales[direction_number]
  e_value <- e_values[direction_number]

  z <- rnorm(n = 1,
             mean = 0,
             sd = sqrt(e_value * scale))
  x_prop <- x_curr + z * direction

  # # work out the density of the the proposed point:
  # arg_name <- formalArgs(logpi)[1]
  # logpi_args[[arg_name]] <- x_prop
  #
  # logpi_prop <- do.call(logpi, logpi_args)
  #
  # # accept-reject
  # loga <- logpi_prop - logpi_curr

  loga_list <- loga(logpi = logpi,
                    logpi_args = logpi_args,
                    x_prop = x_prop,
                    logpi_curr = logpi_curr)

  loga <- loga_list[['loga']]
  logpi_prop <- loga_list[['logpi_prop']]

  u <- runif(1)

  if (log(u) < loga) {
    return(list(x = x_prop,
                logpi = logpi_prop,
                loga = loga,
                direction_number = direction_number))
  } else {
    return(list(x = x_curr,
                logpi = logpi_curr,
                loga = loga,
                direction_number = direction_number))
```

```
  }
}
```

```
PCA_update <- function(x_curr,
                       p_curr,
                       optimal_a,
                       gamma_curr) {
  scales <- p_curr[['scales']]
  mu <- p_curr[['mu']]
  sigma <- p_curr[['sigma']]
  direction_number <- p_curr[['direction_number']]
  a <- min(exp(p_curr[['loga']]), 1)
  m <- p_curr[['m']]

  new_scale <- exp(log(scales[direction_number]) + gamma_curr * (a - optimal_a))
  scales[direction_number] <- new_scale

  mu_new <- mu + gamma_curr * (x_curr - mu)

  sig_step <- outer_prod(x_curr - mu, x_curr - mu)
  sig_new <- sigma + gamma_curr * (sig_step - sigma)

  # Now do the PCA on sig_new. This requires calculating the eigenvectors
  # and values of sig_new. Not entirely sure which algo. to use for this
  # so we'll use the in built R function for the time being.

  e_vector_value <- eigen(sig_new)
  e_vectors <- e_vector_value$vectors[1:m, ]
  e_values <- e_vector_value$values[1:m]

  return(list(scales = scales,
              sigma = sig_new,
              mu = mu_new,
              e_vectors = e_vectors,
              e_values = e_values,
              m = m))
}
```

```
AM_GAS_sample <- function(x_curr,
                          logpi,
                          logpi_args,
                          logpi_curr,
                          p_curr) {
  scale <- p_curr[['scale']]
  sigma <- p_curr[['sigma']]

  x_prop <- rmvnorm(1, mean = x_curr, sigma = scale * sigma)

  loga_list <- loga(logpi = logpi,
                logpi_args = logpi_args,
                x_prop = x_prop,
                logpi_curr = logpi_curr)
  loga <- loga_list[['loga']]
  logpi_prop <- loga_list[['logpi_prop']]
```

```
  u <- runif(1)

  if (log(u) < loga) {
    return(list(x = x_prop,
                logpi = logpi_prop,
                loga = loga))
  } else {
    return(list(x = x_curr,
                logpi = logpi_curr,
                loga = loga))
  }
}
```

```
AM_GAS_update <- function(x_curr,
                          p_curr,
                          optimal_a,
                          gamma_curr) {
  mu_new <- p_curr[['mu']] + gamma_curr * (x_curr - p_curr[['mu']])

  sig_step <- outer_prod(x_curr - p_curr[['mu']], x_curr - p_curr[['mu']])
  sig_new <- p_curr[['sigma']] + gamma_curr * (sig_step - p_curr[['sigma']])

  scale_new <- exp(log(p_curr[['scale']]) + gamma_curr * (min(1, exp(p_curr[['loga']])) - optimal_a))

  return(list(mu = mu_new,
              sigma = sig_new,
              scale = scale_new))
}
```

```
AM_logpi <- function(X,
                     mu,
                     sigma) {
  sigma_inverse <- solve(sigma)

  right_vec <- sigma_inverse %*% as.vector(X - mu)
  return(-0.5 * sum((X - mu) * t(right_vec)))
}

# AM_logpi <- function(X,
#                      dyads,
#                      A) {
#   # the state is a vector of length d * n + 2 consisting of a concatenation
#   # of the columns of Z and tau and gamma ^ 2
#   z_curr <- matrix(nrow = n, ncol = d)
#   for (j in 1:d) {
#     z_curr[, j] <- X[(n * (j - 1) + 1):(n * j)]
#   }
#
#   tau_curr <- X[n * d + 1]
#   gamma2_curr <- X[n * d + 2]
#
#   # make the laplacian
#   degree <- vector(length = n)
#
```

```
#   for (i in 1:n) {
#     degree[i] <- sum(A[i, ])
#   }
#
#   Laplacian <- diag(degree) - A
#
#   # make the new gaussmat based on updated gamma2
#   gaussmat <- Laplacian + (gamma2_curr ^ -1) * priorprecision
#
#   return(ld_gaussian(z_curr, gaussmat) + ld_nogaussian_categorical(z_curr, dyads, gamma2_curr, tau_cu
# }
```

```
gamma <- function(current,
                  iteration) {
  return(0.001)
}
```

```
outer_prod <- function(x, y) {
  out <- matrix(nrow = length(x), ncol = length(y))
  for (i in 1:length(x)) {
    for (j in 1:length(y)) {
      out[i, j] <- x[i] * y[j]
    }
  }

  return(out)
}
```

```
flat_plot <- function(x_store) {
  # plots the first two coordinates against each other of an MCMC
  # run
  x <- x_store[, 1]
  y <- x_store[, 2]

  plot(x, y)
}
```

```
tracer <- function(x_store,
                   alpha,
                   title,
                   xlab,
                   ylab) {
  # traceplots of the first two dimensions of an MCMC run
  df <- as.data.frame(x_store)
  df <- cbind(df, iter = 1:nrow(df))
  melted_df <- melt(df, id = "iter")

  p <- ggplot(melted_df, aes(x = iter,
                             y = value,
                             colour = as.factor(variable))) +
    geom_line() +
    ggtitle(title) +
    theme(plot.title = element_text(size = 0),
          axis.title.x = element_text(size = 10),
          axis.text = element_text(size = 10)) +
```

```
    scale_colour_manual(values = c("green", "red"),
                        labels = c('x', 'y'),
                        guide = guide_legend(title = " ",
                                             title.theme = element_text(size = 0),
                                             label.theme = element_text(size = 10))) +
    xlab(xlab) +
    ylab(ylab)

  grid.arrange(p, ncol = 1, nrow = 1)
}
```

```
contour <- function(x_store) {
  # Shows a contour + scatter plot of the first two dimensions of
  # an MCMC run
  df <- as.data.frame(x_store[, 1:2])

  m <- ggplot(df, aes(x = V1, y = V2)) +
    geom_point() +
    geom_density_2d_filled(alpha = 0.5, show.legend = FALSE) +
    xlim(c(-45, 45)) +
    ylim(c(-45, 45))

  grid.arrange(m, ncol = 1, nrow = 1)
}
```

```
loga <- function(logpi,
                 logpi_args,
                 x_prop,
                 logpi_curr) {
  # Returns a list of the log(acceptance probability) and the
  # log(density) of the proposed point.

  # work out the density of the the proposed point:
  arg_name <- formalArgs(logpi)[1]
  logpi_args[[arg_name]] <- x_prop

  # logpi_prop <- AM_logpi(logpi_args[[3]], logpi_args[[2]], logpi_args[[1]])
  logpi_prop <- do.call(logpi, logpi_args)

  # accept-reject
  loga <- logpi_prop - logpi_curr

  return(list(loga = loga,
              logpi_prop = logpi_prop))
}
```

```
esjd <- function(x_store) {
  n <- nrow(x_store)
  esjd_matrix <- (1 / (n - 1)) * (x_store[2:n, ] - x_store[1:(n - 1), ]) ^ 2
  # vector to store componentwise ESJD
  ret_vec <- vector(length = ncol(x_store))
  ret_vec <- colMeans(esjd_matrix)
  return(ret_vec)
}
```

```r
animate_sigma <- function(sigmas,
                          target_sigma,
                          heat = FALSE) {
  # Animates the evolution of the 95% contour line of the covariance
  # matrices adapting towards the target_sigma. heat = TRUE means the
  # ellipses will be colour coded as to their inhomogeneity factor
  # as expounded on in Rosenthal 2008 (Optimal Proposal Distributions
  # and Adaptive MCMC). This would make the animation considerably slower.
  dim <- nrow(target_sigma)

  nits <- nrow(sigmas) / dim

  # we want to plot the target ellipse so we know what the proposal ellipses
  # are aiming for
  target_ellipse <- ellipse(x = target_sigma)

  max_x <- max(target_ellipse[, 1])
  max_y <- max(target_ellipse[, 2])
  min_x <- min(target_ellipse[, 1])
  min_y <- min(target_ellipse[, 2])

  plot.new()
  plot.window(xlim = c(min_x, max_x),
              ylim = c(min_y, max_y))

  lines(target_ellipse, col = "black")

  for (i in 1:nits) {
    current_sig <- sigmas[(dim * (i - 1) + 1):(dim * i), ]
    ellipse <- ellipse(x = current_sig)

    if (heat) {
      eigen_vals <- eigen(x = sigmas %*% solve(target_sigma))$values
      inhomogeneity <- dim * sum(eigen_vals) * (sum(sqrt(eigen_vals))) ^ (-2)
    } else {
      plot.new()
      plot.window(xlim = c(min_x, max_x),
                  ylim = c(min_y, max_y))

     lines(target_ellipse, col = "black")
     lines(ellipse, col = "red")
     Sys.sleep(0.05)
    }
  }
}

animate_sigmas <- function(sigmas,
                           target_sigma,
                           mus,
                           target_mu,
                           chain) {
  # Animates the evolution of the 95% contour line of the covariance
  # matrices adapting towards the target_sigma.
```

```
  dim <- ncol(sigmas)
  ellipse <- ellipse(x = sigmas[1:dim, ],
                     centre = mus[1, ],
                     npoints = 15)

  df <- as.data.frame(ellipse)

  # time column signifies the iteration, mask column signifies which
  # aspect of the plot we are plotting (i.e. covar., chain, weights etc.)
  df <- cbind(df, time = rep(1, nrow(ellipse)))
  df <- cbind(df, mask = rep(1, nrow(ellipse)))

  nits <- nrow(sigmas) / dim

  # # Set the amount of seconds to animate, and the fps
  # seconds <- 5
  # fps <- 10

  for (i in 2:nits) {
    # create a new dataframe to append to the bottom of the larger df.
    new_ellipse <- ellipse(x = sigmas[(dim * (i - 1) + 1):(dim * i), ],
                           npoints = 15,
                           centre = mus[i, ])
    df_temp <- as.data.frame(new_ellipse)
    df_temp <- cbind(df_temp,
                     time = rep(i, nrow(new_ellipse)),
                     mask = rep(1, nrow(new_ellipse)))

    # make a new row for the chain:
    new_row <- c(x = chain[i, 1], y = chain[i, 2], time = i, mask = 2)
    df_temp <- rbind(df_temp, new_row)

    df <- rbind(df, df_temp)
  }
  # add the target covariance, so we know what is being aimed for
  print(df)
  target_df <- as.data.frame(ellipse(target_sigma, centre = target_mu))

  ggplot(df, aes(x = x, y = y, colour = as.factor(mask), shape = as.factor(mask))) +
    geom_point() +
    scale_shape_manual(values = c(19, 3)) +
    scale_colour_manual(values = c("darkseagreen3", "darkgoldenrod")) +
    geom_point(data = target_df, colour = "violetred2", shape = 14) +
    transition_time(time) +
    labs(title = 'Iteration: {frame_time}')
}

animate_sigmas_CAM_CAS <- function(sigmas,
                           target_sigma,
                           mus,
                           target_mu,
                           chain,
                           lambdas) {
```

```r
# Animates the evolution of the 95% contour line of the covariance
# matrices adapting towards the target_sigma.
dim <- ncol(sigmas)
ellipse <- ellipse(x = sigmas[1:dim, ],
                   centre = mus[1, ],
                   npoints = 15)

df <- as.data.frame(ellipse)

# time column signifies the iteration, mask column signifies which
# aspect of the plot we are plotting (i.e. covar., chain, weights etc.)
df <- cbind(df, time = rep(1, nrow(ellipse)))
df <- cbind(df, mask = rep(1, nrow(ellipse)))
df <- cbind(df, lambda1 = rep(1, nrow(ellipse)))
df <- cbind(df, lambda2 = rep(1, nrow(ellipse)))

nits <- nrow(sigmas) / dim

# # Set the amount of seconds to animate, and the fps
# seconds <- 5
# fps <- 10

for (i in 2:nits) {
  # create a new dataframe to append to the bottom of the larger df.
  new_ellipse <- ellipse(x = sigmas[(dim * (i - 1) + 1):(dim * i), ],
                         npoints = 15,
                         centre = mus[i, ])
  df_temp <- as.data.frame(new_ellipse)
  df_temp <- cbind(df_temp,
                   time = rep(i, nrow(new_ellipse)),
                   mask = rep(1, nrow(new_ellipse)))

  # make a new row for the chain:
  new_row <- c(x = chain[i, 1], y = chain[i, 2], time = i, mask = 2)
  df_temp <- rbind(df_temp, new_row)

  # attach the lambda columns
  df_temp <- cbind(df_temp,
                   lambda1 = rep(lambdas[i, 1], nrow(df_temp)),
                   lambda2 = rep(lambdas[i, 2], nrow(df_temp)))

  df <- rbind(df, df_temp)
}
print(df)
# add the target covariance, so we know what is being aimed for

target_df <- as.data.frame(ellipse(target_sigma, centre = target_mu))

ggplot(df, aes(x = x, y = y)) +
  geom_point(aes(shape = as.factor(mask), colour = as.factor(mask))) +
  scale_shape_manual(values = c(19, 3)) +
  scale_colour_manual(values = c("darkseagreen3", "darkgoldenrod")) +
  annotate(geom = "text",
```

```
            x = 20,
            y = 20,
            label = paste("*lambda1 = ", lambda1,
                          "*lambda2 = ", lambda2),
            parse = TRUE) +
    geom_point(data = target_df, colour = "violetred2", shape = 14) +
    transition_time(time) +
    labs(title = 'Iteration: {frame_time}')
}

animate_sigmas_local_RWM <- function(sigmas,
                                     target_sigma,
                                     k_max,
                                     colours,
                                     fps = 10,
                                     duration = 7,
                                     mus,
                                     target_mu,
                                     weights,
                                     chain) {
  # Animates the evolution of the 95% contour line of the covariance
  # matrices adapting towards the target_sigma. Since the adaptation
  # scheme is local RWM, this animates each sigma per component of the
  # mixture being fit.

  # sigmas: matrix of covariance matrices stacked on top of one another
  # target_sigma: the covariance matrix to be adapted to
  # k_max: number of components in the mixture
  # colours: a list of names of colours, one per component
  # fps: how many frames per second and
  # duration: for how many seconds
  # mus: location of each component
  # target_mu: location of the target
  # weights: weights in the mixture approximating the target
  # chain: the Markov Chain

  dim <- ncol(sigmas)
  nits <- nrow(sigmas) / (dim * k_max)
  frames <- fps * duration
  period <- floor(nits / frames)

  # initialise an empty df
  df <- data.frame(matrix(ncol = 5,
                          nrow = 0,
                          dimnames = list(NULL, c('x',
                                                  'y',
                                                  'component',
                                                  'time',
                                                  'weight'))))

  for (i in 1:nits) {
    if (i %% period == 1) {
      for (s in 1:k_max) {
```

```
        top_index <- dim * k_max * (i - 1) + dim * (s - 1) + 1
        bottom_index <- dim * k_max * (i - 1) + dim * s
        mu <- mus[(k_max * (i - 1) + s), ]
        current_weight <- weights[i, s]

        new_ellipse <- ellipse(x = sigmas[top_index:bottom_index, ],
                               npoints = 15,
                               centre = mu)
        df_temp <- as.data.frame(new_ellipse)
        df_temp <- cbind(df_temp,
                         component = rep(s, nrow(new_ellipse)),
                         time = rep(i, nrow(new_ellipse)),
                         weight = rep(current_weight, nrow(new_ellipse)))

        df <- rbind(df, df_temp)
      }
      new_row <- c(chain[i, 1], chain[i, 2], k_max + 1, i, 0.2)
      df <- rbind(df, new_row)
    }
  }
  # add the target covariance, so we know what is being aimed for
  target_df <- as.data.frame(ellipse(x = target_sigma,
                                     centre = target_mu))

  ggplot(df, aes(x = x,
                 y = y,
                 colour = as.factor(component))) +
    theme(panel.grid.major = element_blank(),
          panel.grid.minor = element_blank(),
          panel.background = element_blank(),
          legend.position = "none",
          axis.line=element_blank(),
          axis.text.x=element_blank(),
          axis.text.y=element_blank(),
          axis.ticks=element_blank(),
          axis.title.x=element_blank(),
          axis.title.y=element_blank()) +
    geom_point(shape = 15,
               aes(size = weight)) +
    scale_colour_manual(values = colours,
                        guide = guide_legend(title = "Components")) +
    geom_point(data = target_df, colour = "yellow", shape = 14) +
    transition_time(time) +
    labs(title = 'Iteration: {frame_time}')
}

animate_PCA <- function(sigmas,
                        target_sigma,
                        e_vectors,
                        m,
                        mus,
                        target_mu,
                        rescale,
```

```
                        chain) {
  # Animates adaptive MCs that use the PCA update. Here
  # the e_vectors are plotted with the evolving covariance
  # matrix. Rescale rescales the eigenvectors so they are visible
  dim <- ncol(sigmas)
  nits <- nrow(sigmas) / dim

  # initialise an empty df:
  df <- data.frame(matrix(ncol = 4,
                          nrow = 0,
                          dimnames = list(NULL, c('x', 'y', 'e_vec', 'time'))))

  for (i in 1:nits) {
    # create a new dataframe to append to the bottom of the larger df.
    new_ellipse <- ellipse(x = sigmas[(dim * (i - 1) + 1):(dim * i), ],
                           npoints = 15,
                           centre = mus[i, ])
    df_temp <- as.data.frame(new_ellipse)
    df_temp <- cbind(df_temp,
                     e_vec = rep(1, nrow(new_ellipse)),
                     time = rep(i, nrow(new_ellipse)))

    for (j in 1:m) {
      new_row <- c(rescale * e_vectors[(m * (i - 1) + j), 1],
                   rescale * e_vectors[(m * (i - 1) + j), 2],
                   2,
                   i)
      df_temp <- rbind(df_temp, new_row)
    }
    new_row <- c(x = chain[i, 1], y = chain[i, 2], e_vec = 3, time = i)
    df_temp <- rbind(df_temp, new_row)

    df <- rbind(df, df_temp)
  }
  # add the target covariance, so we know what is being aimed for
  target_df <- as.data.frame(ellipse(x = target_sigma, centre = target_mu))

  ggplot(df, aes(x = x, y = y)) +
    geom_point(aes(shape = as.factor(e_vec), colour = as.factor(e_vec))) +
    scale_shape_manual(values = c(19, 3, 7)) +
    scale_colour_manual(values = c("darkseagreen3", "darkgoldenrod", 'tomato')) +
    geom_point(data = target_df, colour = "red", shape = 14) +
    transition_time(time) +
    labs(title = 'Iteration: {frame_time}')
}
```

```
nine_plot <- function(obj,
                      scheme,
                      frames,
                      target_sigma,
                      target_mu,
                      k_max = 5,
                      nrow,
                      ncol,
```

```
                    directions = matrix(rep(1, 4),
                                        ncol = 2,
                                        nrow = 2),
                    rescale = 20,
                    output_png = FALSE) {
# A function that plots nine snapshots of an adaptive scheme's Markov Chain
# Monte Carlo Run, using contour plots amongst other things.
# obj: the MCMC list, complete with adaptive parameters
# scheme: the adaptive scheme
# frames: which iterations should have snapshots taken of them
# target_sigma: the covariance of the target
# target_mu: the mean of the target
# k_max: the number of components in the mixture if using Local RWM
# nrow: no. of rows in the final grid of plots
# ncol: no. of cols in the final grid of plots
# directions: a matrix with 2 cols showing the default sampling directions in
# CAM_CAS
# rescale: factor to rescale any arrows
# output_png: whether to output the grobs as pngs

x_store <- obj[['x_store']]
sigmas_full <- obj[['sigmas']]
mus_full <- obj[['mus']]

nits <- nrow(x_store)
dim <- ncol(x_store)
frame_no <- length(frames)

# if the frames object is missing, take nine snapshots of equal distance, starting
# at the first iteration
if (missing(frames)) {
  frames <- floor(nits / frame_no) * 1:frame_no - (floor(nits / frame_no) - 1)
}

# make a list for the grobs
grobs <- list()

# every scheme updates mu and sigma, and has a state, target sigma,
# and target mu so take snapshots of those in any case:
# Take the snapshots:
states <- x_store[frames, ]

sigmas <- list()
for (i in 1:frame_no) {
  sigmas[[i]] <- sigmas_full[(dim * (frames[i] - 1) + 1):(dim * frames[i]), ]
}

mus <- mus_full[frames, ]

# iterate through the frames, taking a snapshot per frame and storing it
# in the grob list
for (i in 1:frame_no) {
  target_points <- as.data.frame(rmvnorm(1000,
```

```
                                        mean = target_mu,
                                        sigma = target_sigma))
  points <- as.data.frame(rmvnorm(1000,
                                  mean = mus[i, ],
                                  sigma = sigmas[[i]]))

  p <- ggplot(target_points, aes(x = V1, y = V2, alpha = 1)) +
  theme(panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(),
        panel.background = element_blank(),
        legend.position = "none",
        axis.line=element_blank(),
        axis.text.x=element_blank(),
        axis.text.y=element_blank(),
        axis.ticks=element_blank(),
        axis.title.x=element_blank(),
        axis.title.y=element_blank()) +
  geom_density_2d(aes(alpha = 1, colour = "red")) +
  geom_density_2d(data = points,
                  aes(x = V1, y = V2, alpha = 1, colour = "blue")) +
  geom_point(data = as.data.frame(t(mus[i, ])),
             aes(x = V1, y = V2, colour = "red", alpha = 1, size = 1.5),
             inherit.aes = FALSE) +
  geom_point(data = as.data.frame(t(states[i, ])),
             aes(x = V1, y = V2, colour = "green", alpha = 1, size = 1.5),
             inherit.aes = FALSE) +
  xlab("") +
  ylab("") +
  ggtitle(paste("iteration ", frames[i])) +
  xlim(-45, 45) +
  ylim(-45, 45)
  # Now we have the grob, add it to the list:
  grobs[[i]] <- p
}

if (scheme == "PCA") {
  eigen_vecs_full <- obj[['e_vectors']]
  segments <- list()

  for (i in 1:frame_no) {
    eigen_vecs <- eigen_vecs_full[(2 * (frames[i] - 1) + 1):(2 * frames[i]), ]

    # normalise the eigenvectors and set them to be a pre specified length
    eigen_vecs[1, ] <- eigen_vecs[1, ] / sqrt(sum(eigen_vecs[1, ] ^ 2))
    eigen_vecs[2, ] <- eigen_vecs[2, ] / sqrt(sum(eigen_vecs[2, ] ^ 2))

    eigen_vecs <- rescale * eigen_vecs

    mu <- mus[i, ]

    eigen_df_1 <- data.frame(x1 = mu[1],
                             y1 = mu[2],
                             x2 = mu[1] + eigen_vecs[1, 1],
```

```
                              y2 = mu[2] + eigen_vecs[1, 2])
    eigen_df_2 <- data.frame(x1 = mu[1],
                             y1 = mu[2],
                             x2 = mu[1] + eigen_vecs[2, 1],
                             y2 = mu[2] + eigen_vecs[2, 2])
    seg_1 <- geom_segment(aes(x = x1,
                  y = y1,
                  xend = x2,
                  yend = y2,
                  colour = "segment"),
              data = eigen_df_1,
              arrow = arrow(length = unit(0.2,"cm")),
              size = 1)
    seg_2 <- geom_segment(aes(x = x1,
                              y = y1,
                              xend = x2,
                              yend = y2,
                              colour = "segment"),
                          data = eigen_df_2,
                          arrow = arrow(length = unit(0.2,"cm")),
                          size = 1)

    grobs[[i]] <- grobs[[i]] + seg_1 + seg_2
  }
  grid.arrange(grobs = grobs, nrow = nrow, ncol = ncol)
} else if (scheme == "CAM_CAS") {
  lambdas_full <- obj[['lambdas']]
  segments <- list()

  for (i in 1:frame_no) {
    # get the scales for this frame
    lambdas <- lambdas_full[frames[frame_no], ]
    mu <- mus[i, ]
    for (r in 1:nrow(directions)) {
      # normalise and set to a length according to their scale, and rescale
      # if muddled in the final plot
      dir <- directions[r, ] / sqrt(sum(directions[r, ] ^ 2))
      dir <- rescale * lambdas[r] * dir

      dir_df <- data.frame(x1 = mu[1],
                           y1 = mu[2],
                           x2 = mu[1] + dir[1],
                           y2 = mu[2] + dir[2])
      seg <- geom_segment(aes(x = x1,
                              y = y1,
                              xend = x2,
                              yend = y2,
                              colour = "segment"),
                          data = dir_df,
                          arrow = arrow(length = unit(0.2,"cm")),
                          size = 1)
      # add to the right grob
      grobs[[i]] <- grobs[[i]] + seg
```

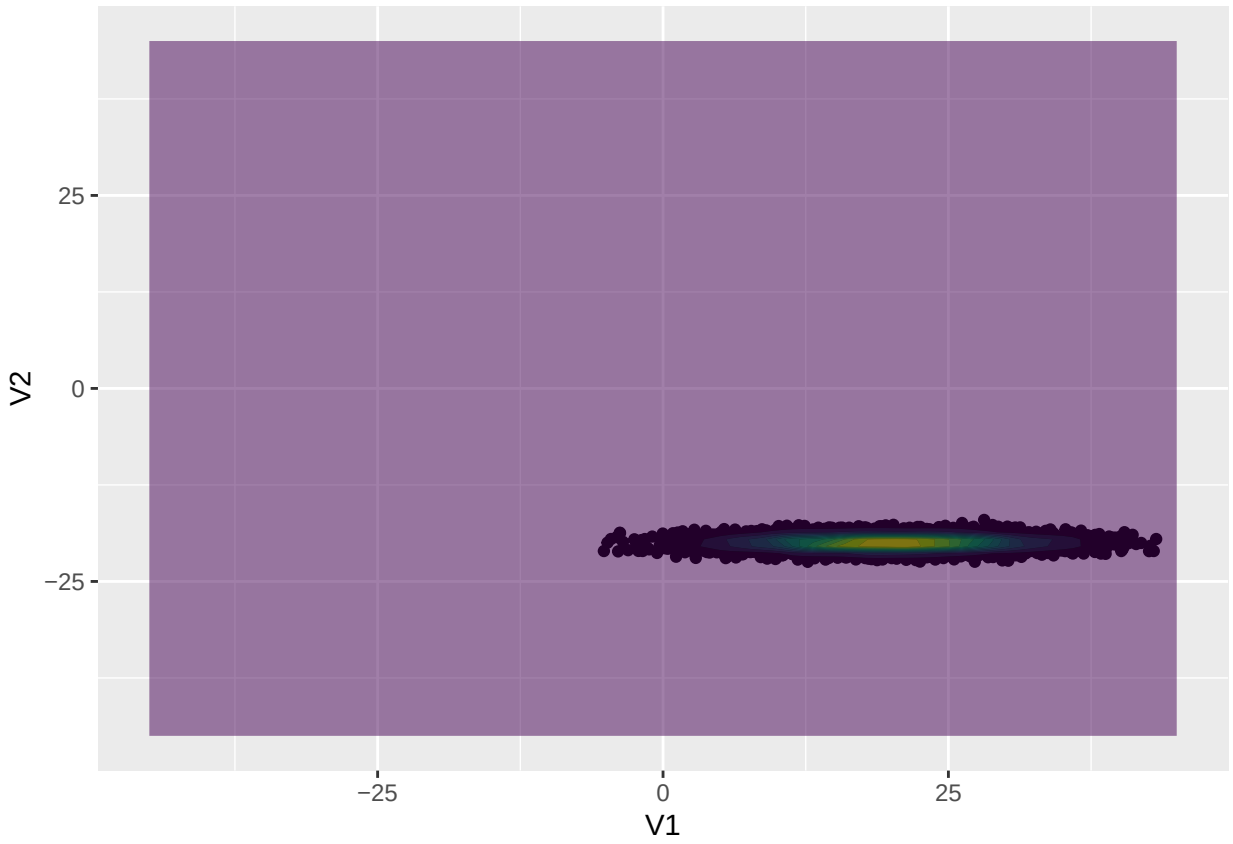
```
      }
    }
  }

  # output the pngs
  if (output_png) {
    for (i in 1:frame_no) {
      name <- paste("output", i, ".png", sep = "")

      ggsave(filename = name,
             plot = grobs[[i]],
             device = "png")
    }
  } else {
    grid.arrange(grobs = grobs, nrow = nrow, ncol = ncol)
  }
}
```

We start with the following target distribution:

```
target_data <- rmvnorm(n = 10000,
                mean = c(20, -20),
                sigma = 0.5 * matrix(c(100, 0, 0, 1),
                                     nrow = 2,
                                     ncol = 2))

contour(target_data)
```

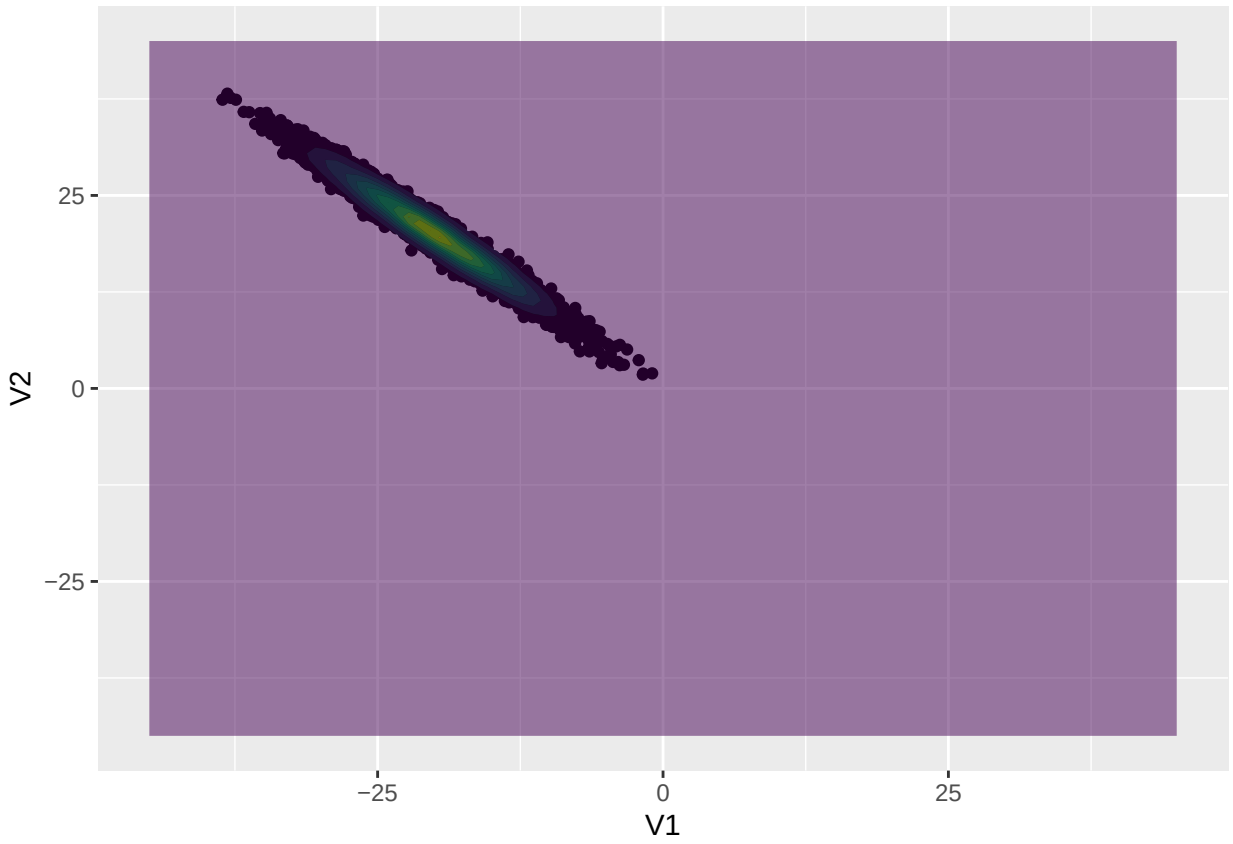
```
## Warning: Removed 2 rows containing non-finite values (stat_density2d_filled).
```

```
## Warning: Removed 2 rows containing missing values (geom_point).
```



with initial proposal distribution:

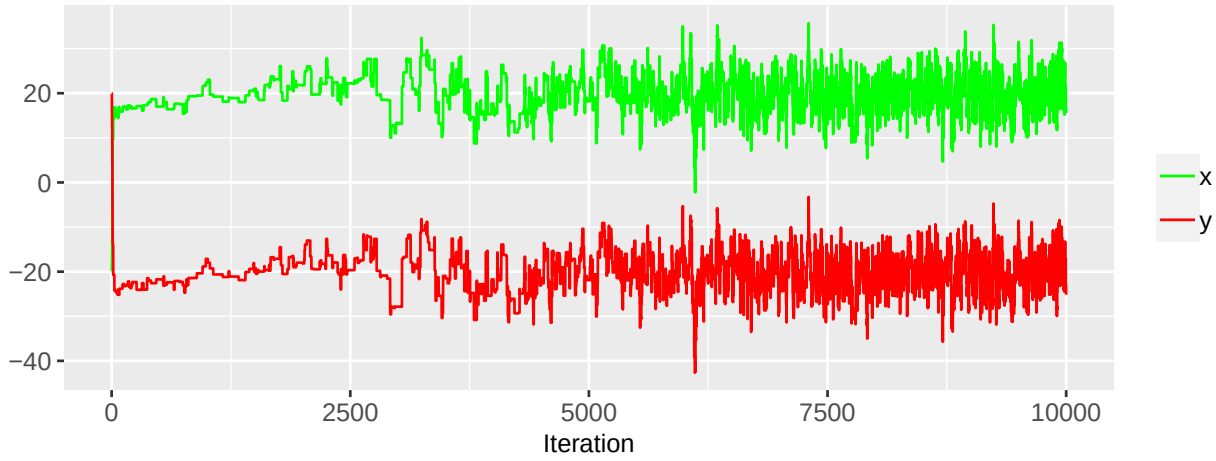
```
proposal_data <- rmvnorm(n = 10000,
                 mean = c(-20, 20),
                 sigma = 0.25 * matrix(c(101, -99, -99, 101),
                                       nrow = 2,
                                       ncol = 2))

contour(proposal_data)
```



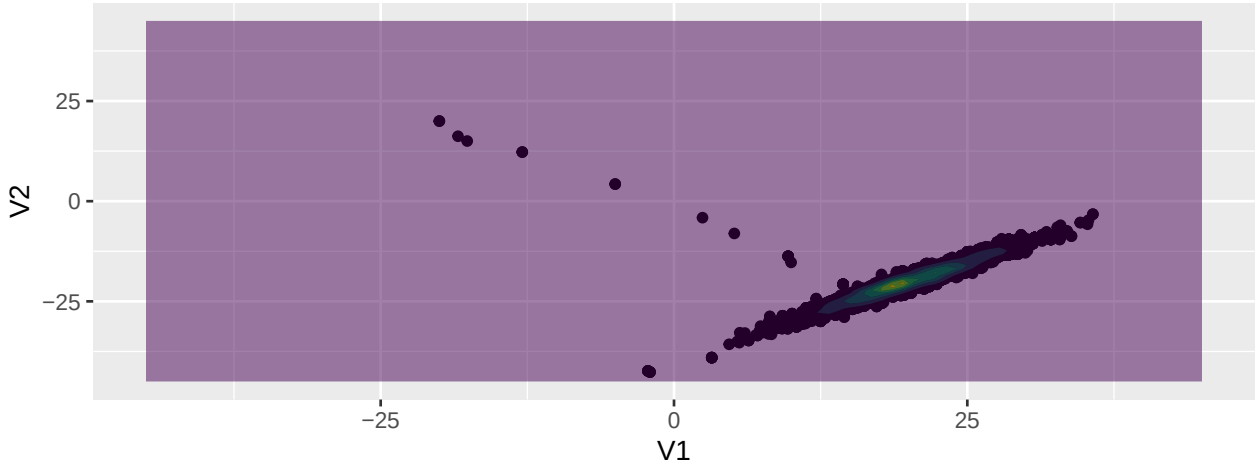
Adaptive Metropolis

```
init <- list(X = c(-20, 20),
             params = list(mu = c(-20, 20),
                           sigma = 0.25 * matrix(c(101, -99, -99, 101),
                                                 nrow = 2,
                                                 ncol = 2)),
             gamma = gamma)
sample <- "AM_sample"
update <- "AM_update"
logpi <- AM_logpi
logpi_args <- list(mu = c(20, -20),
                   sigma = 0.25 * matrix(c(101, 99, 99, 101),
                                         nrow = 2,
                                         ncol = 2))
seed <- 2000
learn_in <- 0
nits <- 10000
gamma <- gamma
#stop_after <- 5000

first_run <- adapt(init = init,
                   sample = sample,
                   update = update,
                   logpi = logpi,
                   logpi_args = logpi_args,
```

```
                  learn_in = learn_in,
                  nits = nits,
                  gamma = gamma,
                  seed = seed)
tracer(first_run[['x_store']],
       1,
       " ",
       "Iteration",
       " ")
```




```
contour(first_run[['x_store']])
```




```
print('MCSE is (batch size 1000)')
```

```
## [1] "MCSE is (batch size 1000)"
```

```
print(batchSE(mcmc(first_run[['x_store']]), 1000))
```

```
## [1] 0.3738884 0.3324629
```

```
print('Effective n is')
```

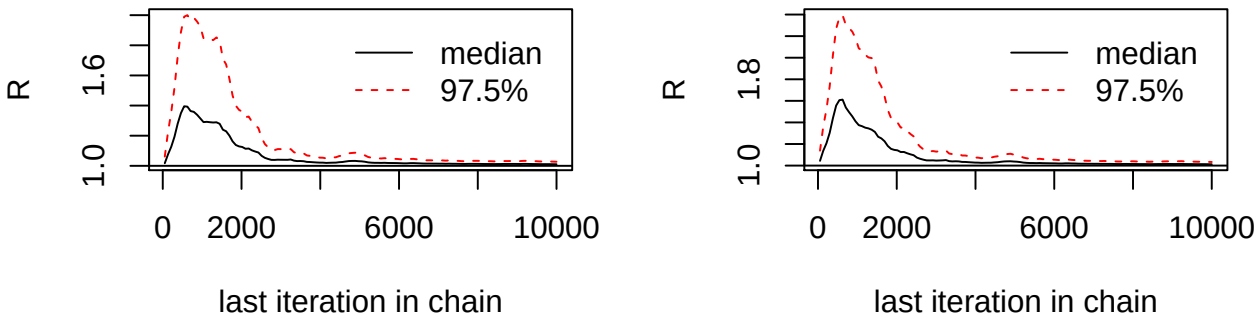
```
## [1] "Effective n is"
```

```
print(effectiveSize(mcmc(first_run[['x_store']])))
```

```
##     var1     var2
```

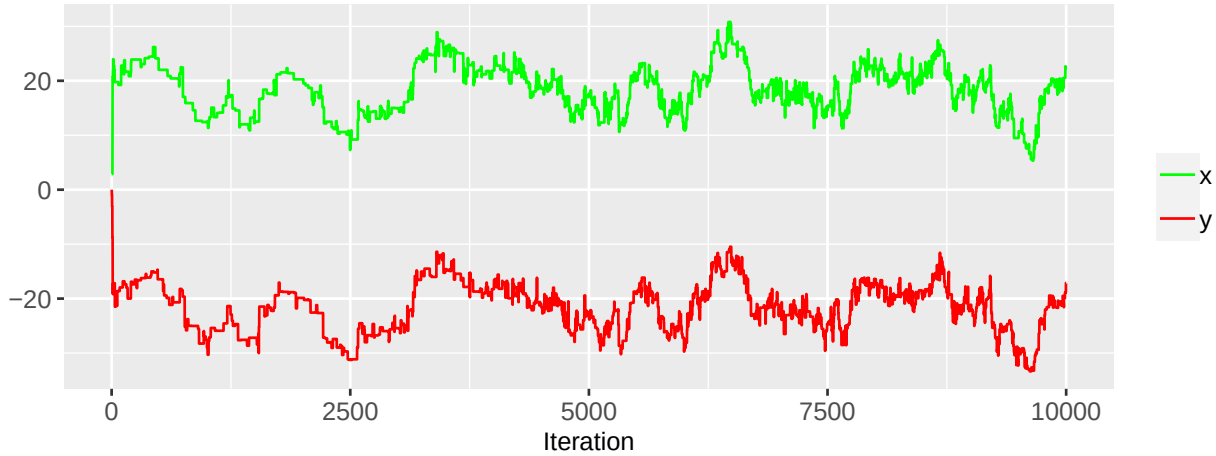
```
## 233.3638 360.5279
```

```
# make a new chain to plot the Gelman-Rubin diagnostic

first_run_alt2 <- adapt(init = init,
                    sample = sample,
                    update = update,
                    logpi = logpi,
                    logpi_args = logpi_args,
                    learn_in = learn_in,
                    nits = nits,
                    gamma = gamma,
                    seed = seed + 100)
first_run_alt3 <- adapt(init = init,
                    sample = sample,
                    update = update,
                    logpi = logpi,
                    logpi_args = logpi_args,
                    learn_in = learn_in,
                    nits = nits,
                    gamma = gamma,
                    seed = seed + 200)
first_run_alt4 <- adapt(init = init,
                    sample = sample,
                    update = update,
                    logpi = logpi,
                    logpi_args = logpi_args,
                    learn_in = learn_in,
                    nits = nits,
                    gamma = gamma,
                    seed = seed + 300)
gelman.plot(x = mcmc.list(mcmc(first_run[['x_store']]),
                          mcmc(first_run_alt2[['x_store']]),
                          mcmc(first_run_alt3[['x_store']]),
                          mcmc(first_run_alt4[['x_store']])),
            bin.width = 100,
            max.bins = 120,
            autoburnin = FALSE,
            ylab = 'R')
```

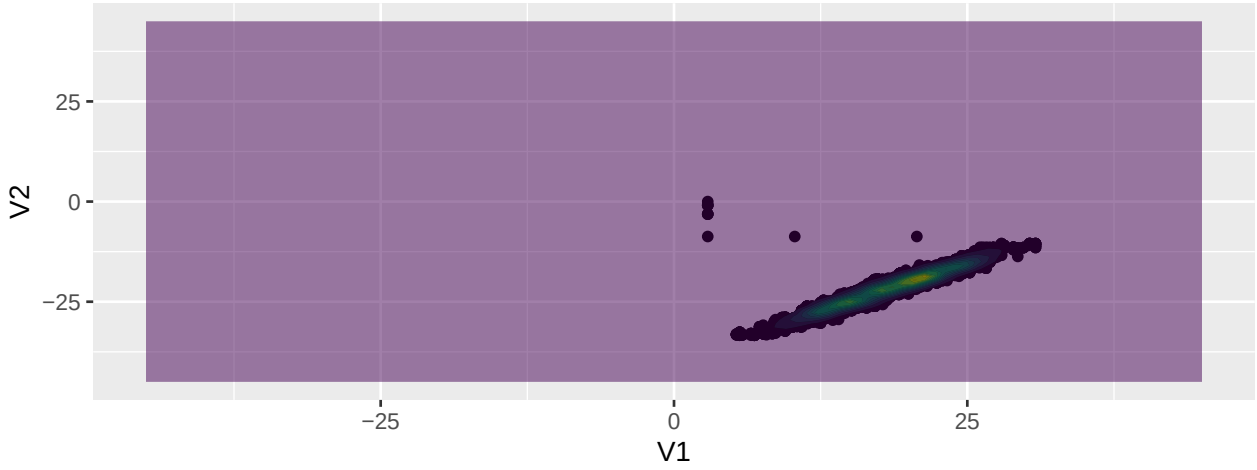


Componentwise Adaptive Metropolis, Componentwise Adaptive Scaling

```
init <- list(X = c(0, 0),
             params = list(mu = c(-20, 20),
                           sigma = 0.25 * matrix(c(101, -99, -99, 101),
                                                 nrow = 2,
                                                 ncol = 2),
                           lambdas = rep(1, 2)),
             gamma = gamma)
sample <- "CAM_CAS_sample"
update <- "CAM_CAS_update"
logpi <- AM_logpi
logpi_args <- list(mu = c(20, -20),
                   sigma = 0.25 * matrix(c(101, 99, 99, 101),
                                         nrow = 2,
                                         ncol = 2))
seed <- 200
learn_in <- 0
nits <- 10000
gamma <- gamma
#stop_after <- 5000

second_run <- adapt(init = init,
                    sample = sample,
                    update = update,
                    logpi = logpi,
                    logpi_args = logpi_args,
                    learn_in = learn_in,
                    nits = nits,
                    gamma = gamma)
tracer(second_run[['x_store']],
       1,
       " ",
       "Iteration",
       " ")
```



```
contour(second_run[['x_store']])
```




```
print('MCSE is (batch size 1000)')
```

```
## [1] "MCSE is (batch size 1000)"
```

```
print(batchSE(mcmc(second_run[['x_store']]), 1000))
```

```
## [1] 0.9240957 0.9144963
```

```
print('Effective n is')
```

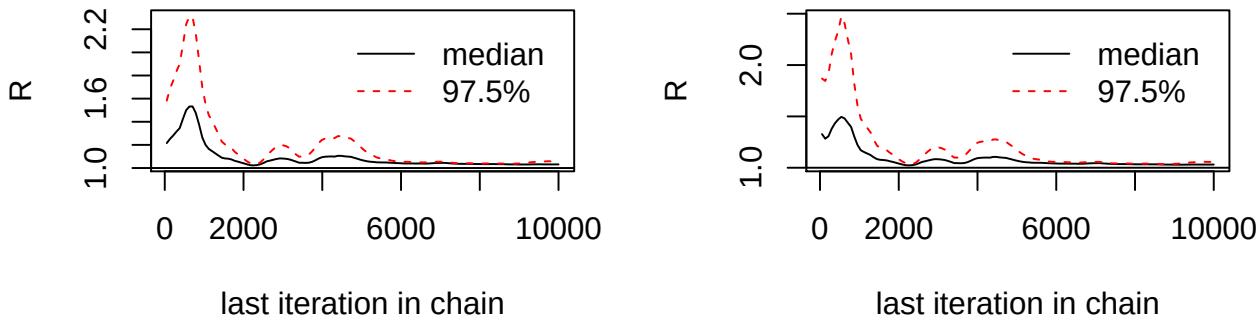
```
## [1] "Effective n is"
```

```
print(effectiveSize(mcmc(second_run[['x_store']])))
```

```
##     var1     var2
## 20.07965 19.17127
```

```
second_run_alt2 <- adapt(init = init,
                  sample = sample,
                  update = update,
                  logpi = logpi,
                  logpi_args = logpi_args,
                  learn_in = learn_in,
                  nits = nits,
                  gamma = gamma,
                  seed = seed + 100)
second_run_alt3 <- adapt(init = init,
                  sample = sample,
                  update = update,
                  logpi = logpi,
                  logpi_args = logpi_args,
                  learn_in = learn_in,
                  nits = nits,
                  gamma = gamma,
                  seed = seed + 200)
second_run_alt4 <- adapt(init = init,
                  sample = sample,
                  update = update,
                  logpi = logpi,
                  logpi_args = logpi_args,
```

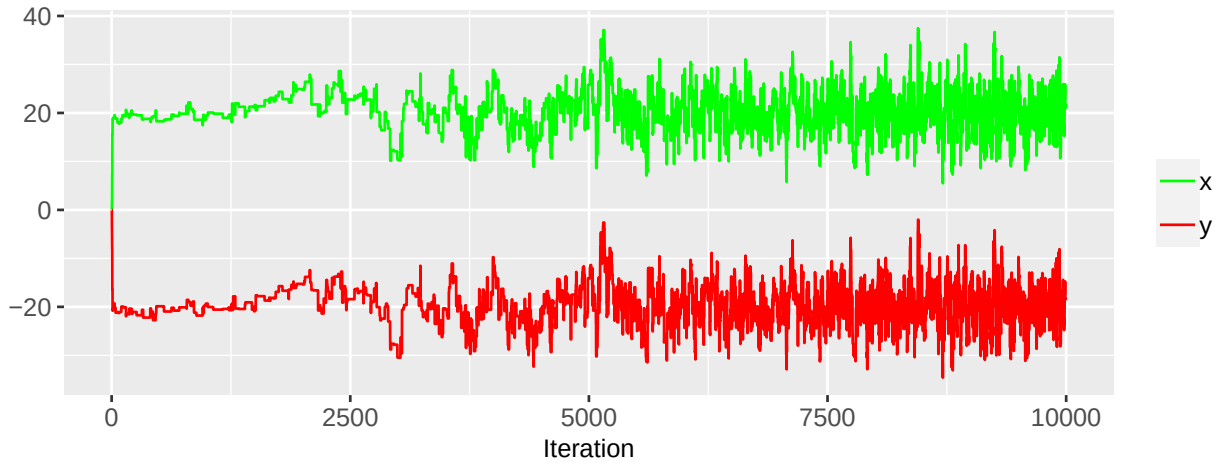
```
                  learn_in = learn_in,
                  nits = nits,
                  gamma = gamma,
                  seed = seed + 300)
gelman.plot(x = mcmc.list(mcmc(second_run[['x_store']]),
                          mcmc(second_run_alt2[['x_store']]),
                          mcmc(second_run_alt3[['x_store']]),
                          mcmc(second_run_alt4[['x_store']])),
            bin.width = 100,
            max.bins = 120,
            autoburnin = FALSE,
            ylab = 'R')
```



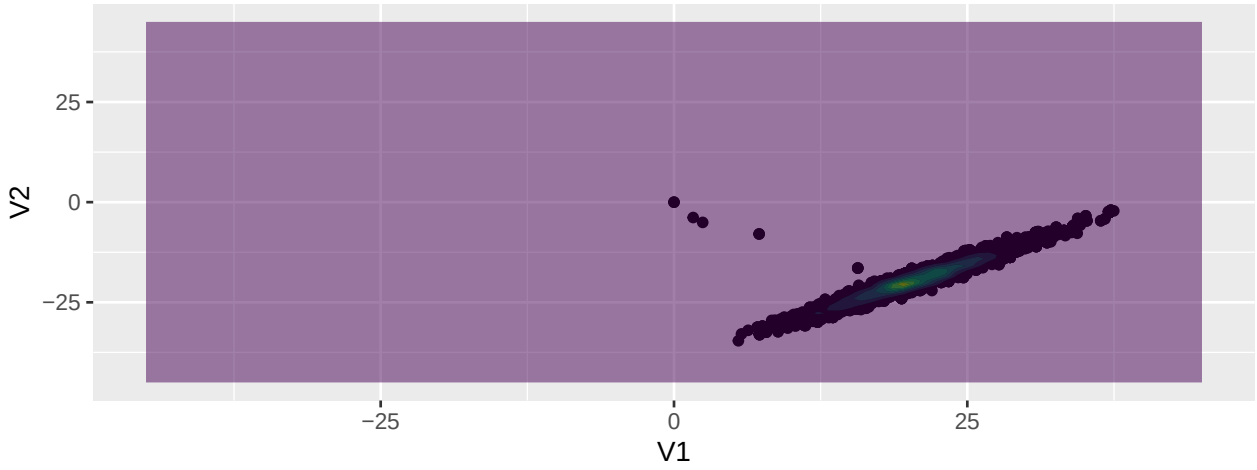
Global Adaptive Metropolis, Componentwise Adaptive Scaling

```
init <- list(X = c(0, 0),
             params = list(mu = c(-20, 20),
                           sigma = 0.25 * matrix(c(101, -99, -99, 101),
                                                 nrow = 2,
                                                 ncol = 2),
                           lambdas = rep(1, 2)),
             gamma = gamma)
sample <- "GAM_CAS_sample"
update <- "GAM_CAS_update"
logpi <- AM_logpi
logpi_args <- list(mu = c(20, -20),
                   sigma = 0.25 * matrix(c(101, 99, 99, 101),
                                         nrow = 2,
                                         ncol = 2))
seed <- 200
learn_in <- 0
nits <- 10000
gamma <- gamma
#stop_after <- 5000

third_run <- adapt(init = init,
                   sample = sample,
                   update = update,
                   logpi = logpi,
                   logpi_args = logpi_args,
                   learn_in = learn_in,
                   nits = nits,
```

```
                  gamma = gamma)
tracer(third_run[['x_store']],
       1,
       " ",
       "Iteration",
       " ")
```




```
contour(third_run[['x_store']])
```




```
print('MCSE is (batch size 1000)')
```

```
## [1] "MCSE is (batch size 1000)"
```

```
print(batchSE(mcmc(third_run[['x_store']]), 1000))
```

```
## [1] 0.2808341 0.2602979
```

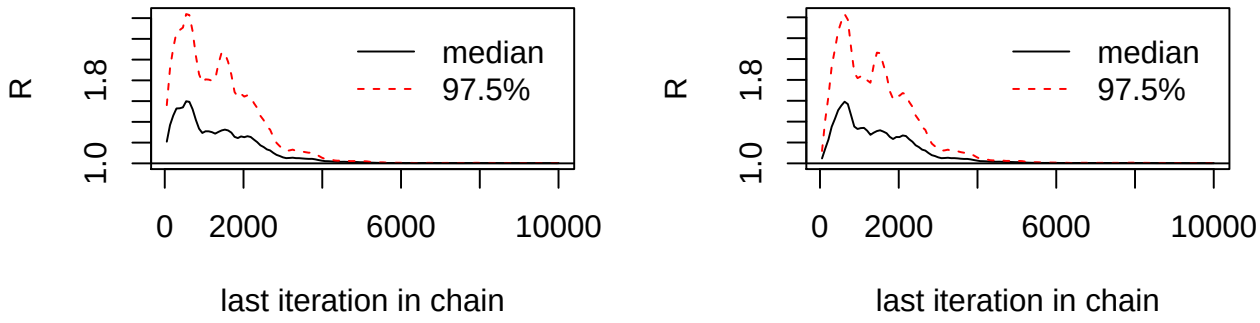
```
print('Effective n is')
```

```
## [1] "Effective n is"
```

```
print(effectiveSize(mcmc(third_run[['x_store']])))
```

```
##     var1     var2
## 224.9630 228.3772
```

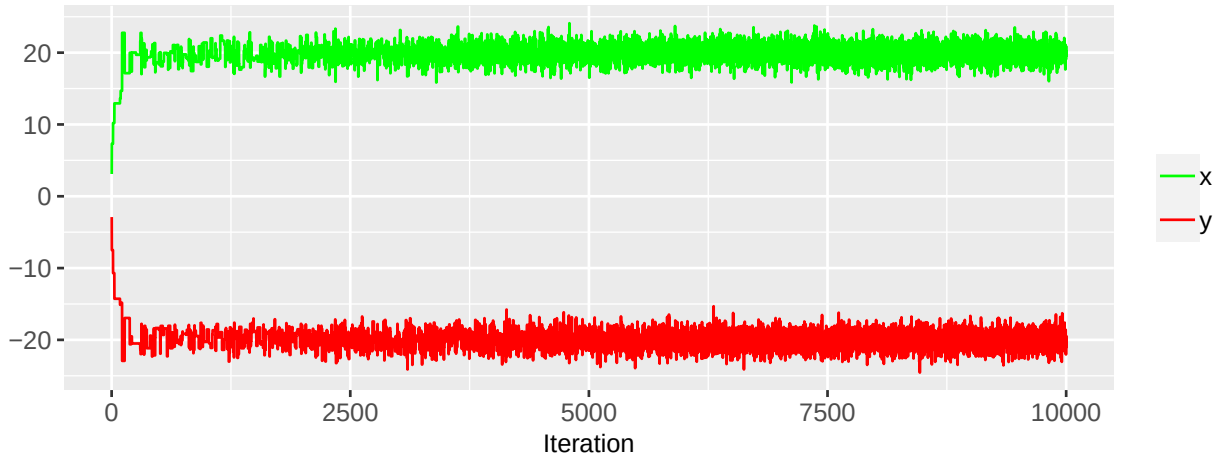
```
third_run_alt2 <- adapt(init = init,
                        sample = sample,
                        update = update,
                        logpi = logpi,
                        logpi_args = logpi_args,
                        learn_in = learn_in,
                        nits = nits,
                        gamma = gamma,
                        seed = seed + 100)
third_run_alt3 <- adapt(init = init,
                        sample = sample,
                        update = update,
                        logpi = logpi,
                        logpi_args = logpi_args,
                        learn_in = learn_in,
                        nits = nits,
                        gamma = gamma,
                        seed = seed + 200)
third_run_alt4 <- adapt(init = init,
                        sample = sample,
                        update = update,
                        logpi = logpi,
                        logpi_args = logpi_args,
                        learn_in = learn_in,
                        nits = nits,
                        gamma = gamma,
                        seed = seed + 300)
gelman.plot(x = mcmc.list(mcmc(third_run[['x_store']]),
                          mcmc(third_run_alt2[['x_store']]),
                          mcmc(third_run_alt3[['x_store']]),
                          mcmc(third_run_alt4[['x_store']])),
            bin.width = 100,
            max.bins = 120,
            autoburnin = FALSE,
            ylab = 'R')
```



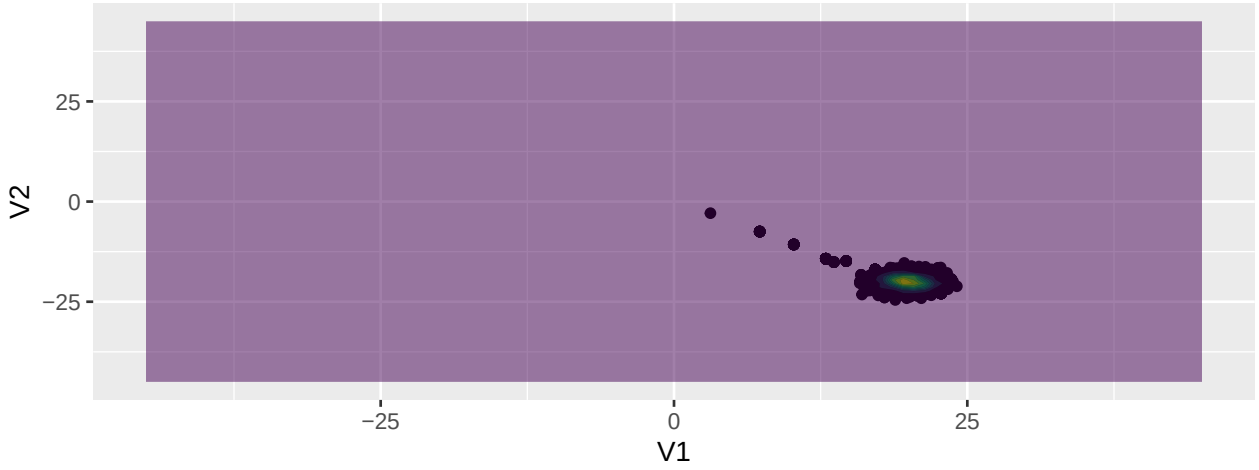
Localised RWM

```
init_sigma <- 0.25 * matrix(c(101, -99, -99, 101),
                            nrow = 2,
                            ncol = 2)
init_sigmas <- matrix(nrow = 8, ncol = 2)
```

```r
for (i in 0:3) {
  init_sigmas[(2 * i + 1):(2 * i + 2), ] <- init_sigma
}

mus <- matrix(c(-20, 20, -20, 0, 20, 20, -20, 0), nrow = 4, ncol = 2)

init <- list(X = c(0, 0),
             params = list(mus = mus,
                           sigmas = init_sigmas,
                           weights = rep(1 / 4, 4),
                           lambdas = rep(1, 4)),
             gamma = gamma)

sample <- "Local_RWM_sample"
update <- "Local_RWM_update"
logpi <- AM_logpi
logpi_args <- list(mu = c(20, -20),
                   sigma = 0.25 * matrix(c(101, 99, 99, 101),
                                         nrow = 2,
                                         ncol = 2))
seed <- 200
learn_in <- 0
nits <- 10000
gamma <- gamma
#stop_after <- 5000

fourth_run <- adapt(init = init,
                    sample = sample,
                    update = update,
                    logpi = logpi,
                    logpi_args = logpi_args,
                    learn_in = learn_in,
                    nits = nits,
                    gamma = gamma)
tracer(fourth_run[['x_store']],
       1,
       " ",
       "Iteration",
       " ")
```



```
contour(fourth_run[['x_store']])
```




```
print('MCSE is (batch size 1000)')
```

```
## [1] "MCSE is (batch size 1000)"
```

```
print(batchSE(mcmc(fourth_run[['x_store']]), 1000))
```

```
## [1] 0.1218369 0.1115902
```

```
print('Effective n is')
```

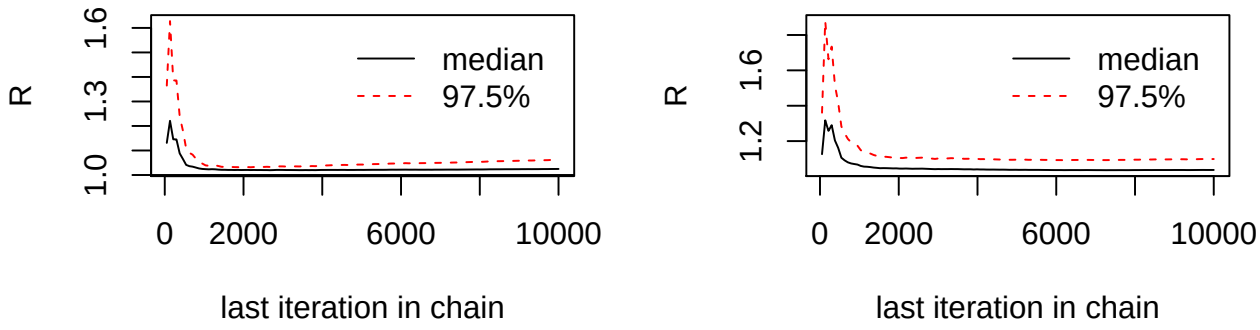
```
## [1] "Effective n is"
```

```
print(effectiveSize(mcmc(fourth_run[['x_store']])))
```

```
##     var1     var2
## 423.0591 367.0825
```

```
fourth_run_alt2 <- adapt(init = init,
                  sample = sample,
                  update = update,
                  logpi = logpi,
                  logpi_args = logpi_args,
                  learn_in = learn_in,
                  nits = nits,
                  gamma = gamma,
                  seed = seed + 100)
fourth_run_alt3 <- adapt(init = init,
                  sample = sample,
                  update = update,
                  logpi = logpi,
                  logpi_args = logpi_args,
                  learn_in = learn_in,
                  nits = nits,
                  gamma = gamma,
                  seed = seed + 200)
fourth_run_alt4 <- adapt(init = init,
                  sample = sample,
                  update = update,
                  logpi = logpi,
                  logpi_args = logpi_args,
```

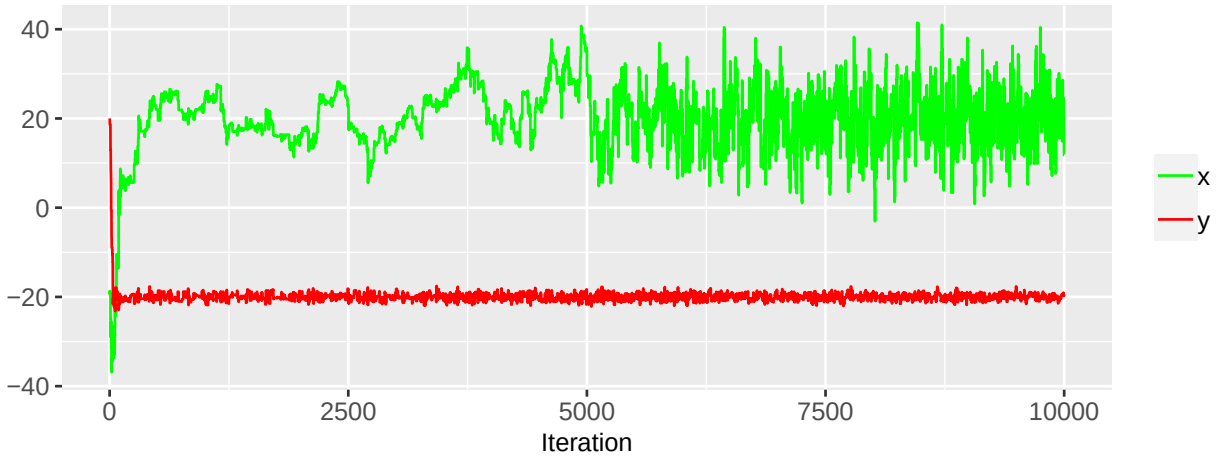
```
                  learn_in = learn_in,
                  nits = nits,
                  gamma = gamma,
                  seed = seed + 300)
gelman.plot(x = mcmc.list(mcmc(fourth_run[['x_store']]),
                          mcmc(fourth_run_alt2[['x_store']]),
                          mcmc(fourth_run_alt3[['x_store']]),
                          mcmc(fourth_run_alt4[['x_store']])),
            bin.width = 100,
            max.bins = 120,
            autoburnin = FALSE,
            ylab = 'R')
```



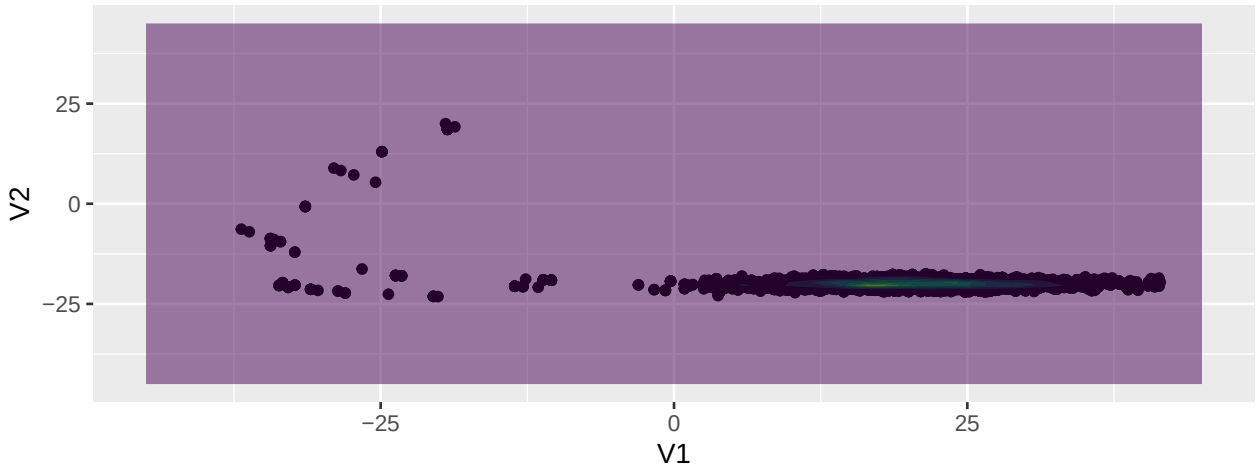
Principal components Metropolis Update

```
# m is the number of eigenvectors we want to explore along

m <- 2
X <- c(-20, 20)
dim <- length(X)
init <- list(X = X,
             params = list(mu = c(-20, 20),
                           sigma = 0.25 * matrix(c(101, -99, -99, 101),
                                                 nrow = dim,
                                                 ncol = dim),
                           m = m,
                           scales = rep(1, m),
                           e_vectors = diag(dim)[1:m, ],
                           e_values = rep(1, m)),
             gamma = gamma)

sample <- "PCA_sample"
update <- "PCA_update"
logpi <- AM_logpi

# Lets change the target distn. s.t. the eigen vectors of the target
# covar. are pi / 4 away from those of the initial proposal.
logpi_args <- list(mu = c(20, -20),
                   sigma =  0.5 * matrix(c(100, 0, 0, 1),
                                         nrow = 2,
                                         ncol = 2))
```

```
seed <- 2
learn_in <- 0
nits <- 10000
gamma <- gamma
#stop_after <- 5000

fifth_run <- adapt(init = init,
                   sample = sample,
                   update = update,
                   logpi = logpi,
                   logpi_args = logpi_args,
                   learn_in = learn_in,
                   nits = nits,
                   gamma = gamma,
                   m = m,
                   seed = seed)
tracer(fifth_run[['x_store']],
       1,
       " ",
       "Iteration",
       " ")
```




```
contour(fifth_run[['x_store']])
```



```
print('MCSE is (batch size 1000)')
```

```
## [1] "MCSE is (batch size 1000)"
```

```
print(batchSE(mcmc(fifth_run[['x_store']]), 1000))
```

```
## [1] 0.9359042 0.0654743
```

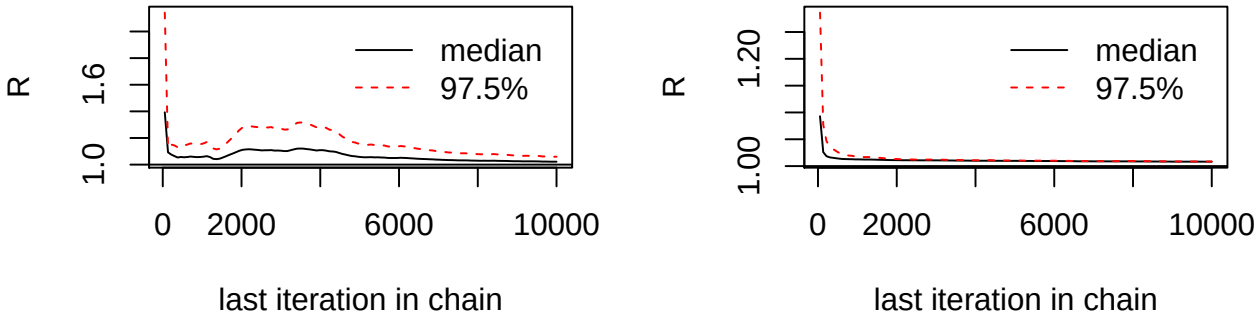
```
print('Effective n is')
```

```
## [1] "Effective n is"
```

```
print(effectiveSize(mcmc(fifth_run[['x_store']])))
```

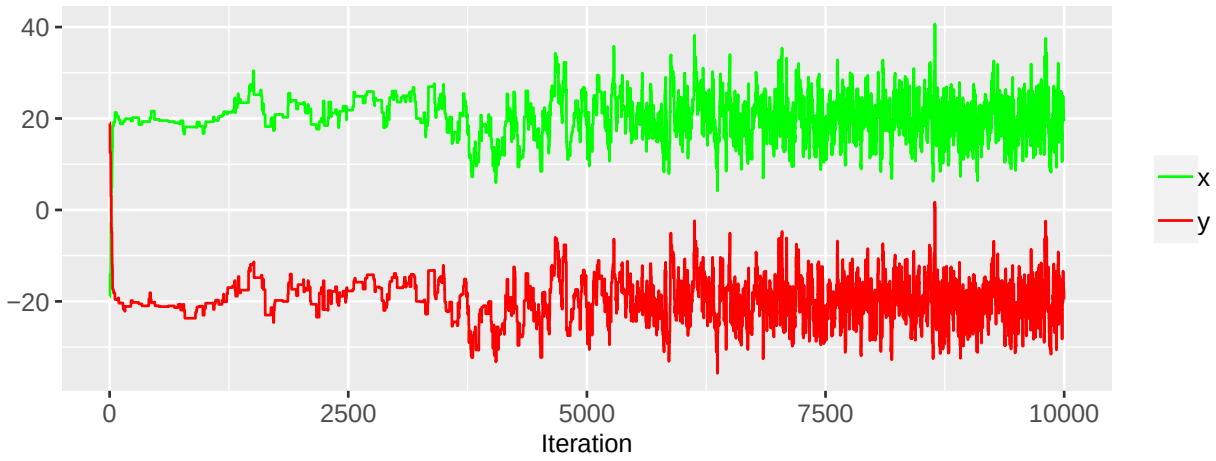
```
##      var1      var2
##  82.51232 445.89477
```

```
fifth_run_alt2 <- adapt(init = init,
                  sample = sample,
                  update = update,
                  logpi = logpi,
                  logpi_args = logpi_args,
                  learn_in = learn_in,
                  nits = nits,
                  gamma = gamma,
                  seed = seed + 100)
fifth_run_alt3 <- adapt(init = init,
                  sample = sample,
                  update = update,
                  logpi = logpi,
                  logpi_args = logpi_args,
                  learn_in = learn_in,
                  nits = nits,
                  gamma = gamma,
                  seed = seed + 200)
fifth_run_alt4 <- adapt(init = init,
                  sample = sample,
                  update = update,
                  logpi = logpi,
                  logpi_args = logpi_args,
                  learn_in = learn_in,
                  nits = nits,
                  gamma = gamma,
                  seed = seed + 300)
gelman.plot(x = mcmc.list(mcmc(fifth_run[['x_store']]),
                          mcmc(fifth_run_alt2[['x_store']]),
                          mcmc(fifth_run_alt3[['x_store']]),
                          mcmc(fifth_run_alt4[['x_store']])),
            bin.width = 100,
            max.bins = 120,
            autoburnin = FALSE,
            ylab = 'R')
```

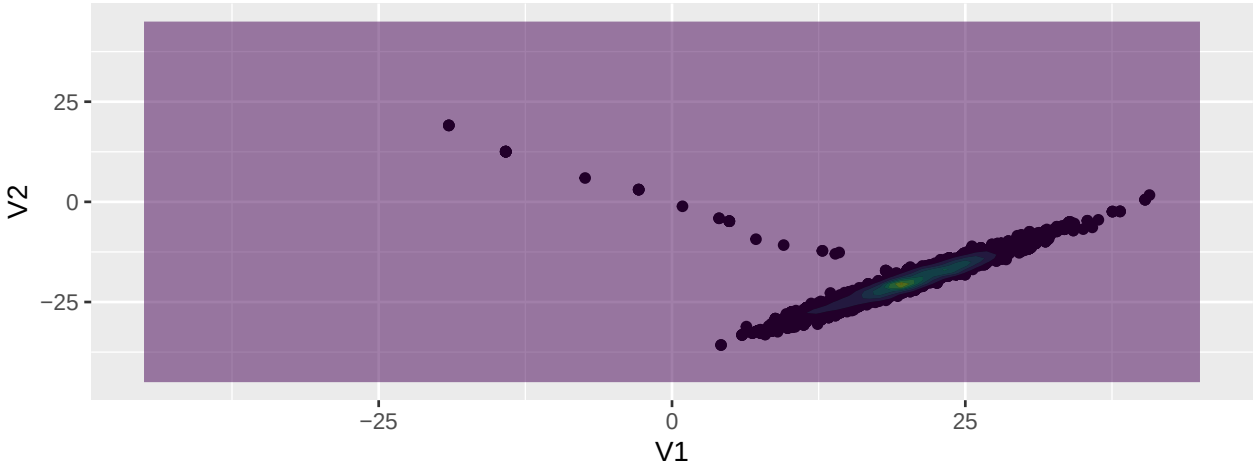


Adaptive Metropolis, Global Adaptive Scaling

```
init <- list(X = c(-20, 20),
             params = list(mu = c(-20, 20),
                           sigma = 0.25 * matrix(c(101, -99, -99, 101),
                                                 nrow = 2,
                                                 ncol = 2),
                           scale = 1),
             gamma = gamma)
sample <- "AM_GAS_sample"
update <- "AM_GAS_update"
logpi <- AM_logpi
logpi_args <- list(mu = c(20, -20),
                   sigma = 0.25 * matrix(c(101, 99, 99, 101),
                                         nrow = 2,
                                         ncol = 2))
seed <- 2008
learn_in <- 0
nits <- 10000
gamma <- gamma
#stop_after <- 5000

sixth_run <- adapt(init = init,
                   sample = sample,
                   update = update,
                   logpi = logpi,
                   logpi_args = logpi_args,
                   learn_in = learn_in,
                   nits = nits,
                   gamma = gamma,
                   seed = seed)
tracer(sixth_run[['x_store']],
       1,
       " ",
       "Iteration",
       " ")
```




```
contour(sixth_run[['x_store']])
```




```
print('MCSE is (batch size 1000)')
```

```
## [1] "MCSE is (batch size 1000)"
```

```
print(batchSE(mcmc(sixth_run[['x_store']]), 1000))
```

```
## [1] 0.3525315 0.2959582
```

```
print('Effective n is')
```

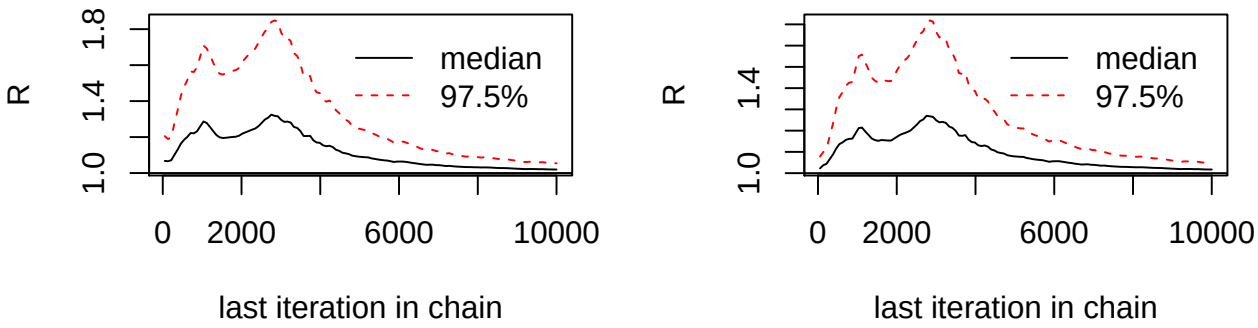
```
## [1] "Effective n is"
```

```
print(effectiveSize(mcmc(sixth_run[['x_store']])))
```

```
##     var1     var2
## 288.0814 291.6536
```

```
# make a new chain to plot the Gelman-Rubin diagnostic

sixth_run_alt2 <- adapt(init = init,
                  sample = sample,
                  update = update,
                  logpi = logpi,
                  logpi_args = logpi_args,
                  learn_in = learn_in,
```

```
                    nits = nits,
                    gamma = gamma,
                    seed = seed + 100)
sixth_run_alt3 <- adapt(init = init,
                    sample = sample,
                    update = update,
                    logpi = logpi,
                    logpi_args = logpi_args,
                    learn_in = learn_in,
                    nits = nits,
                    gamma = gamma,
                    seed = seed + 200)
sixth_run_alt4 <- adapt(init = init,
                    sample = sample,
                    update = update,
                    logpi = logpi,
                    logpi_args = logpi_args,
                    learn_in = learn_in,
                    nits = nits,
                    gamma = gamma,
                    seed = seed + 300)
gelman.plot(x = mcmc.list(mcmc(sixth_run[['x_store']]),
                          mcmc(sixth_run_alt2[['x_store']]),
                          mcmc(sixth_run_alt3[['x_store']]),
                          mcmc(sixth_run_alt4[['x_store']])),
            bin.width = 100,
            max.bins = 120,
            autoburnin = FALSE,
            ylab = 'R')
```



Testing on a more complex target

Let M be the 10×10 matrix s.t. $(M)_{i,j} = ij/100$. M is symmetric $\implies$ M^2 is positive definite, so we can let M^2 be the target covariance. The top left entry is 1.0384. The mean of the square of the first coordinate of our MCMC runs should be 1.0384. Let's have a look at the target distn.

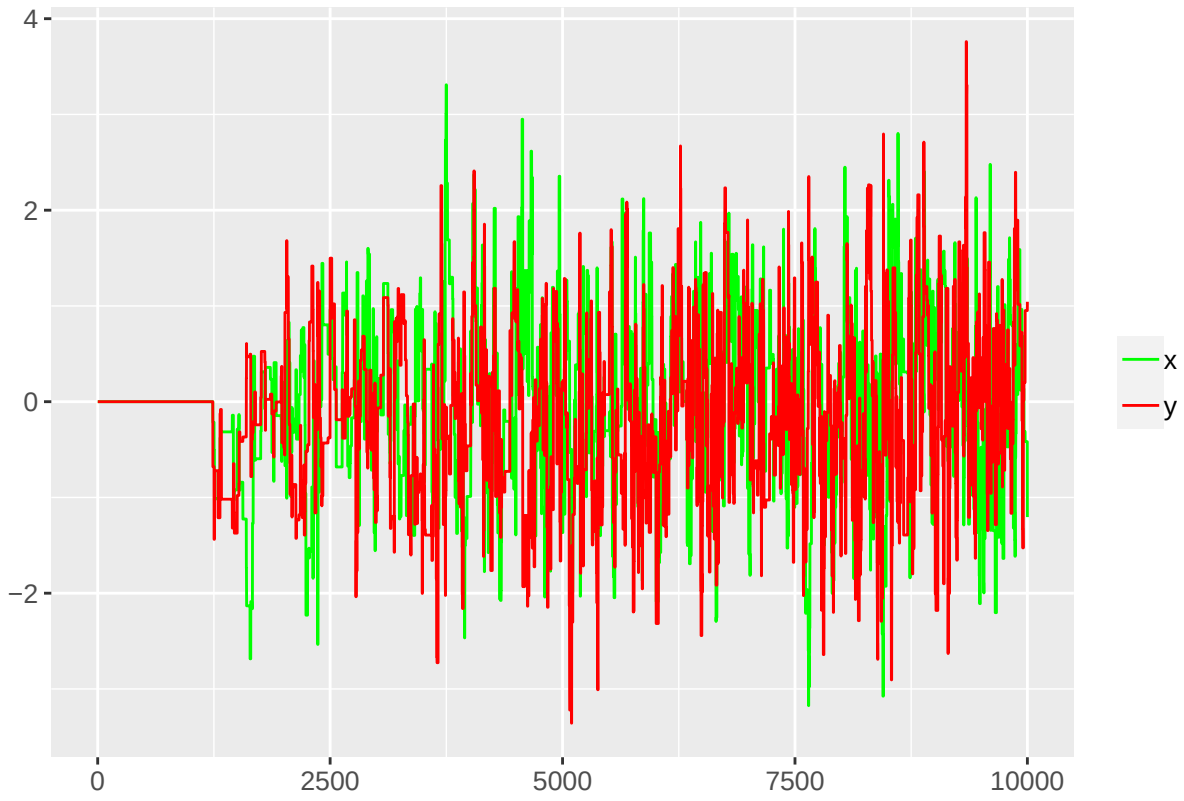
```
dimension <- 10
M <- matrix(nrow = dimension, ncol = dimension)

for (i in 1:nrow(M)) {
  for (j in 1:ncol(M)) {
    if (i == j) {
      M[i, j] <- 1
```

```
    } else {
      M[i, j] <- (i * j) / (nrow(M) ^ 2)
    }
  }
}

M_sq <- M %*% M

print(M_sq[1, 1])
```

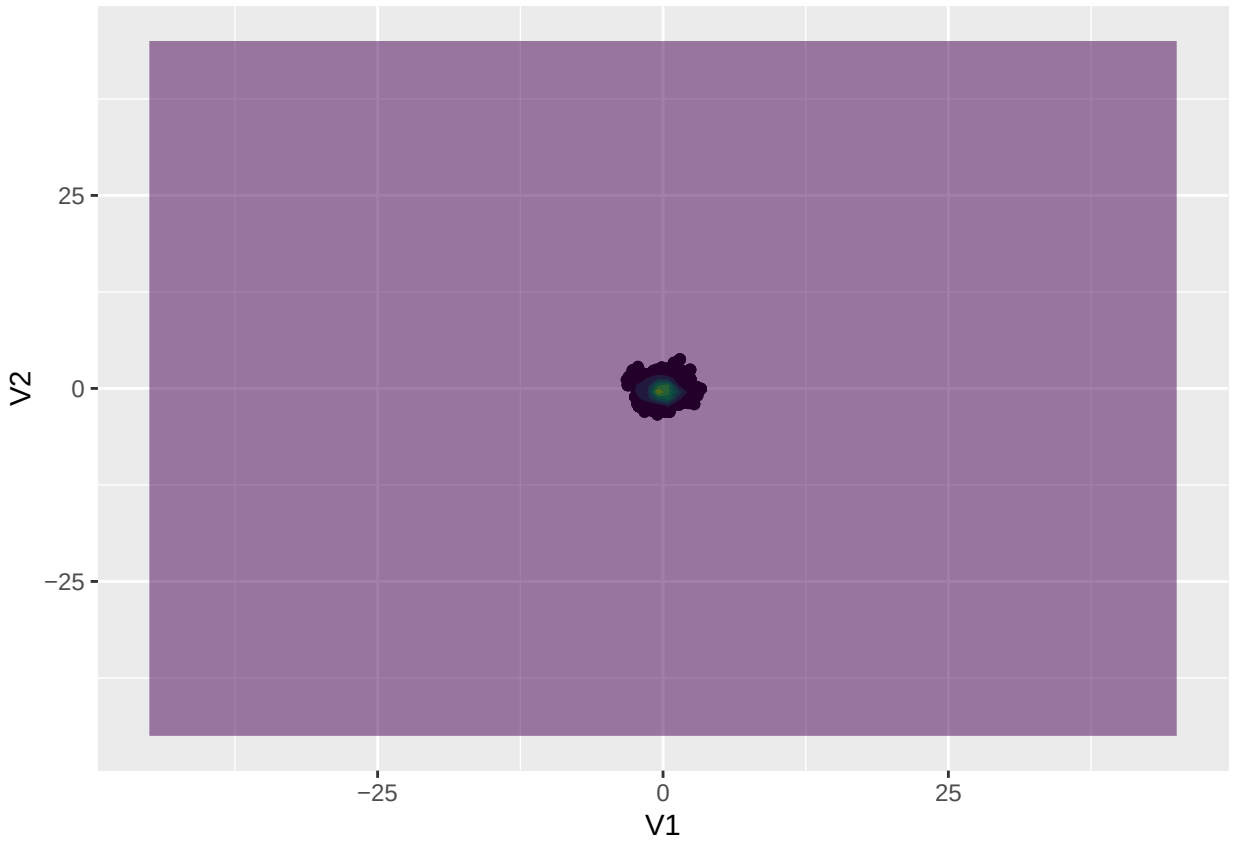
```
## [1] 1.0384
```

Adaptive Metropolis

```
init <- list(X = rep(0, dimension),
             params = list(mu = rep(0, dimension), sigma = diag(dimension)),
             gamma = gamma)

sample <- "AM_sample"
update <- "AM_update"
logpi <- AM_logpi
logpi_args <- list(mu = rep(0, dimension),
                   sigma = M_sq)
seed <- 2000
learn_in <- 0
nits <- 10000
gamma <- gamma
#stop_after <- 5000

sixth_run <- adapt(init = init,
                   sample = sample,
                   update = update,
                   logpi = logpi,
                   logpi_args = logpi_args,
                   learn_in = learn_in,
                   nits = nits,
                   gamma = gamma)
tracer(sixth_run[['x_store']][, 1:2],
       0.7,
       " ",
       " ",
       " ")
```




```
contour(sixth_run[['x_store']][, 1:2])
```




```
square_mean <- mean(sixth_run[['x_store']][, 1] ^ 2)
RMSE <- sqrt(mean((sixth_run[['x_store']][, 1] ^ 2 - 1.0384) ^ 2))
print('Mean is')
```

```
## [1] "Mean is"
```

```
print(square_mean)
```

```
## [1] 0.7478544
```

```
print('RMSE is')
```

```
## [1] "RMSE is"
```

```
print(RMSE)
```

```
## [1] 1.237605
```

```
print('MCSE is (batch size 1000)')
```

```
## [1] "MCSE is (batch size 1000)"
```

```
print(batchSE(mcmc(sixth_run[['x_store']]), 1000))
```

```
##  [1] 0.05338846 0.05686962 0.06825212 0.07962844 0.04216855 0.05063807
##  [7] 0.08836326 0.06678148 0.07695982 0.07975548
```

```
print('Effective n is')
```

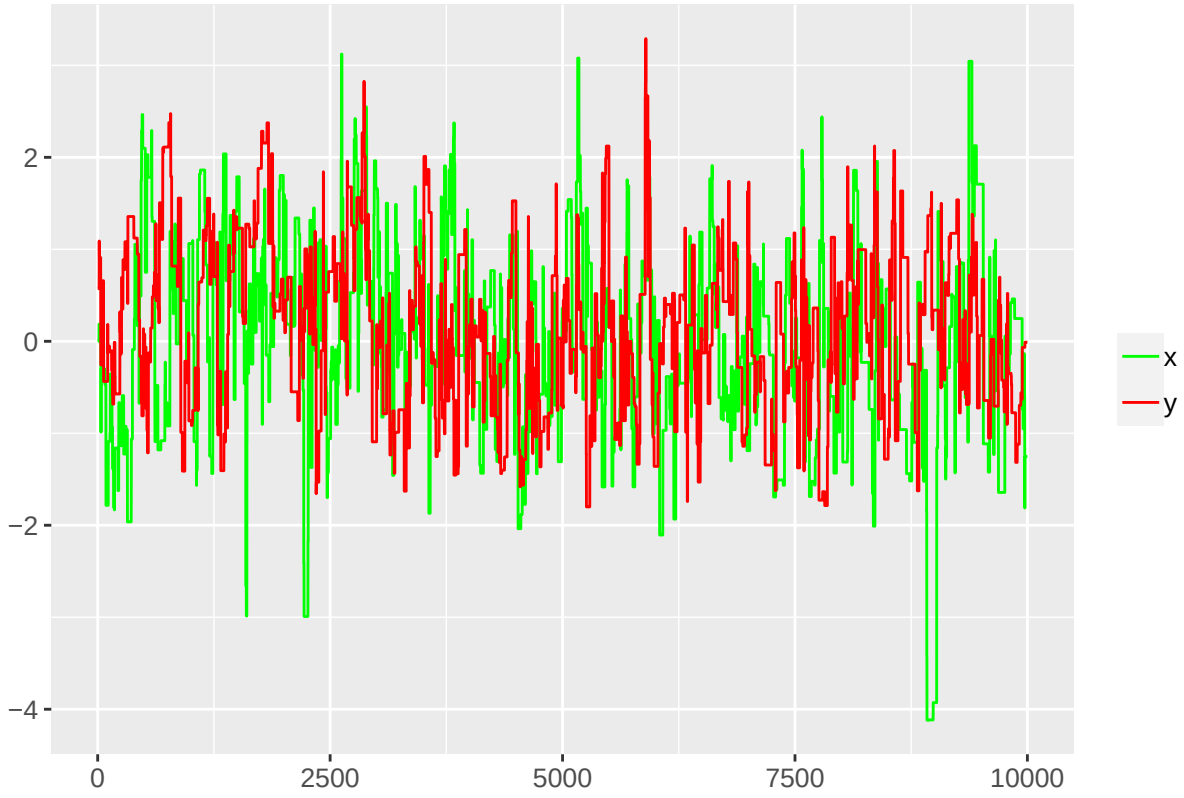
```
## [1] "Effective n is"
```

```
print(effectiveSize(mcmc(sixth_run[['x_store']])))
```

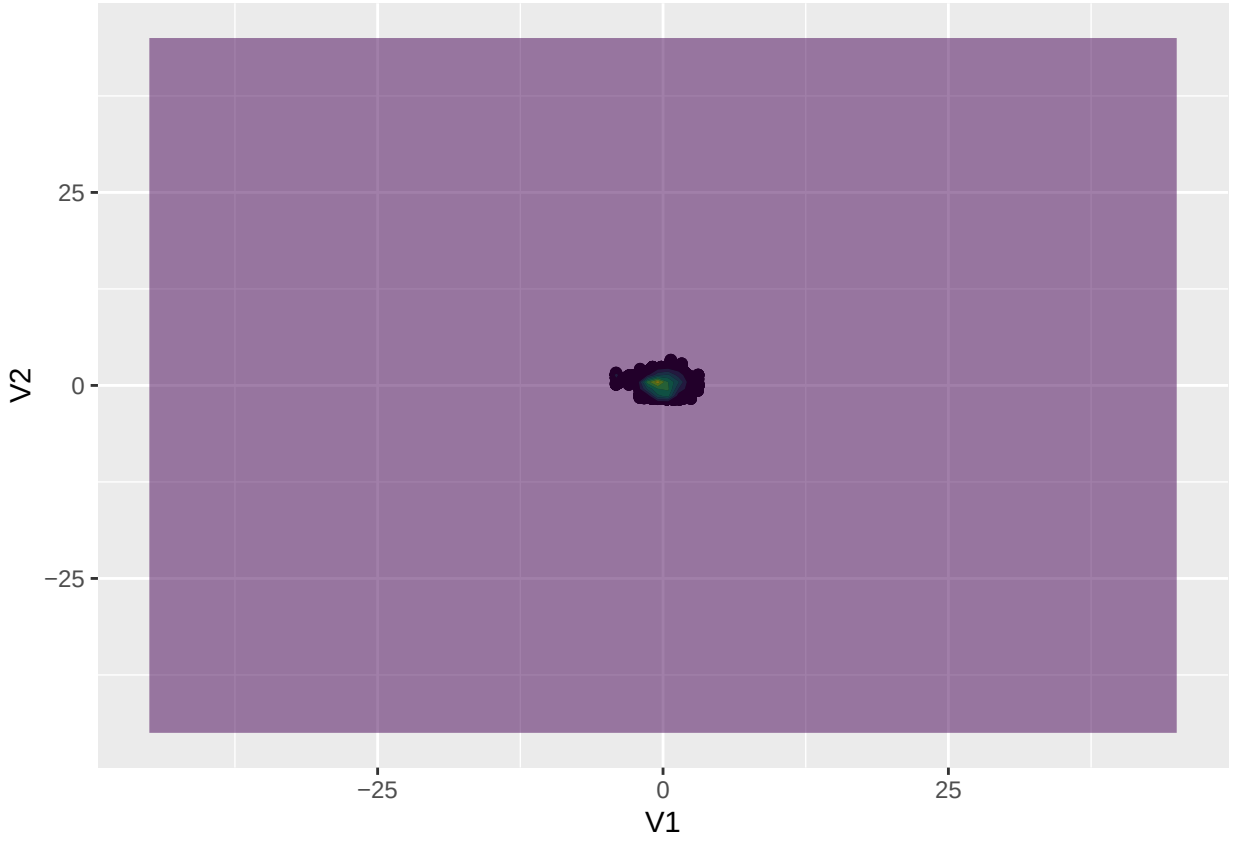
```
##     var1     var2     var3     var4     var5     var6     var7     var8
## 252.2459 276.1294 264.8671 271.7838 274.7623 292.7814 268.0645 284.4696
##     var9    var10
## 286.2500 283.7839
```

Componentwise Adaptive Metropolis, Componentwise adaptive scaling

```
init <- list(X = rep(0, dimension),
             params = list(mu = rep(0, dimension),
                           sigma = diag(dimension),
                           lambdas = rep(1, dimension)),
             gamma = gamma)
sample <- "CAM_CAS_sample"
update <- "CAM_CAS_update"
logpi <- AM_logpi
logpi_args <- list(mu = rep(0, dimension),
                   sigma = M_sq)
seed <- 200
learn_in <- 100
nits <- 10000
gamma <- gamma
#stop_after <- 5000

seventh_run <- adapt(init = init,
                  sample = sample,
                  update = update,
                  logpi = logpi,
                  logpi_args = logpi_args,
                  learn_in = learn_in,
                  nits = nits,
                  gamma = gamma)
tracer(seventh_run[['x_store']][, 1:2],
       0.7,
       " ",
       " ",
       " ")
```




```
contour(seventh_run[['x_store']][, 1:2])
```




```
square_mean <- mean(seventh_run[['x_store']][, 1] ^ 2)
RMSE <- sqrt(mean((seventh_run[['x_store']][, 1] ^ 2 - 1.0384) ^ 2))
print('Mean is')
```

```
## [1] "Mean is"
```

```
print(square_mean)
```

```
## [1] 1.19226
```

```
print('RMSE is')
```

```
## [1] "RMSE is"
```

```
print(RMSE)
```

```
## [1] 2.119186
```

```
print('MCSE is (batch size 1000)')
```

```
## [1] "MCSE is (batch size 1000)"
```

```
print(batchSE(mcmc(seventh_run[['x_store']]), 1000))
```

```
##  [1] 0.1139961 0.1098893 0.1032527 0.1130509 0.1091964 0.1317705 0.1563191
##  [8] 0.1451582 0.1767961 0.1811703
```

```
print('Effective n is')
```

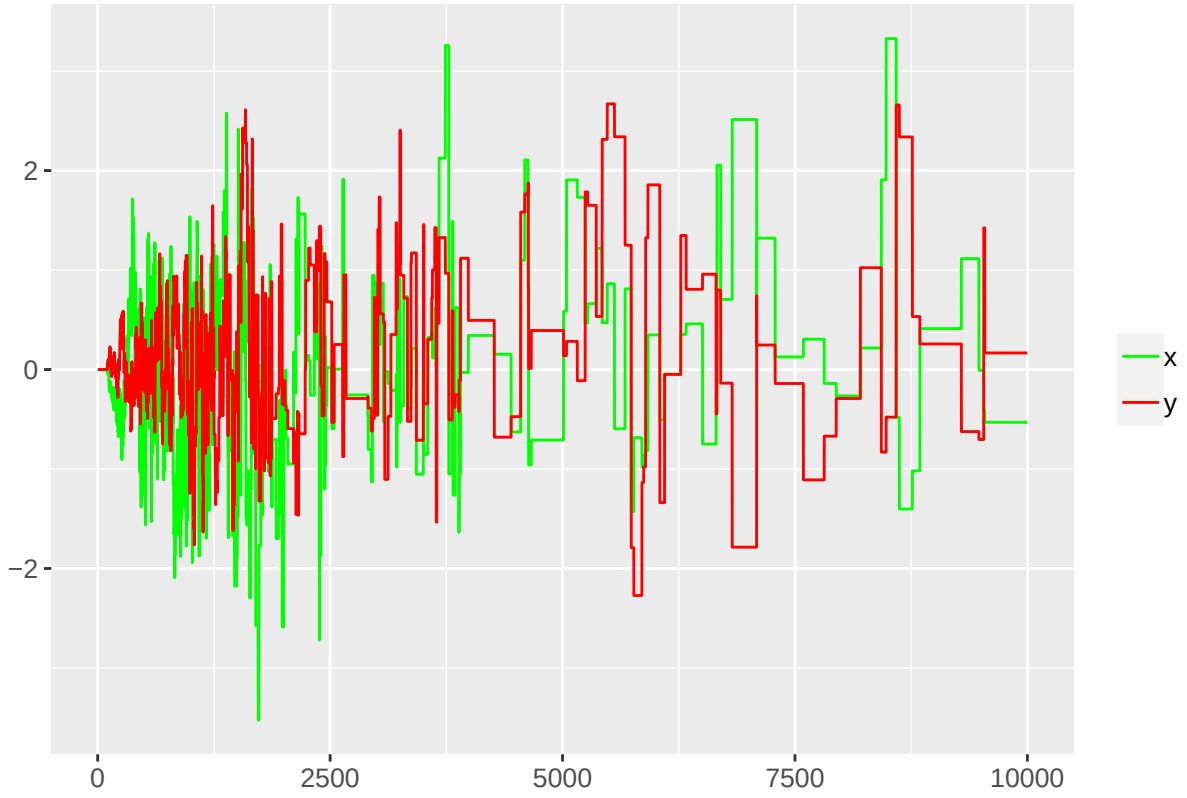
```
## [1] "Effective n is"
```

```
print(effectiveSize(mcmc(seventh_run[['x_store']])))
```

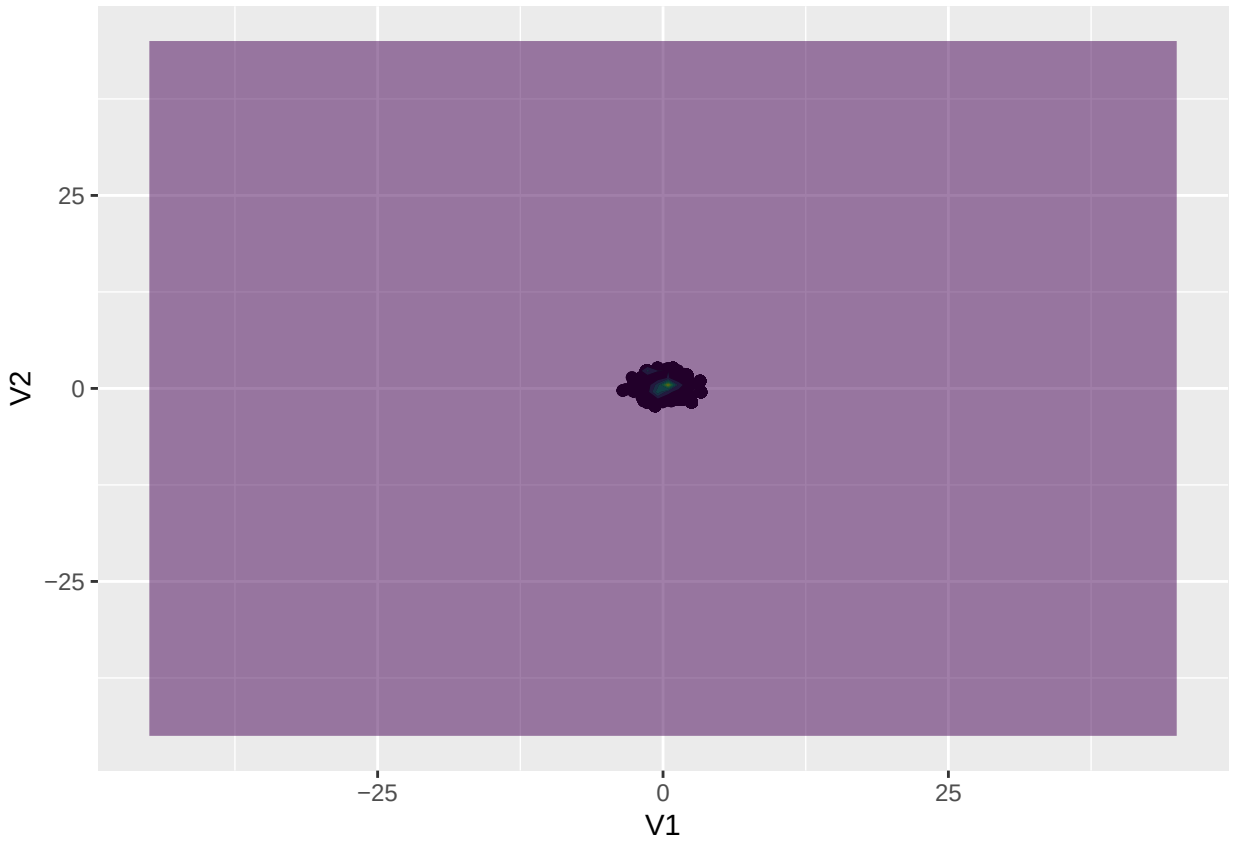
```
##       var1       var2       var3       var4       var5       var6       var7
## 112.758017 110.782376 111.080908 105.461613 101.395618  81.153723  32.735968
##       var8       var9      var10
##  30.788250   8.875951   2.933179
```

Global Adaptive Metropolis, Componentwise Adaptive Scaling

```
init <- list(X = rep(0, dimension),
             params = list(mu = rep(0, dimension),
                           sigma  = diag(dimension),
                           lambdas = rep(1, dimension)),
             gamma = gamma)
sample <- "GAM_CAS_sample"
update <- "GAM_CAS_update"
logpi <- AM_logpi
logpi_args <- list(mu = rep(0, dimension),
                   sigma = M_sq)
seed <- 200
learn_in <- 100
nits <- 10000
gamma <- gamma
#stop_after <- 5000

eighth_run <- adapt(init = init,
                    sample = sample,
                    update = update,
                    logpi = logpi,
                    logpi_args = logpi_args,
                    learn_in = learn_in,
                    nits = nits,
                    gamma = gamma)
tracer(eighth_run[['x_store']][, 1:2],
       0.7,
       " ",
       " ",
       " ")
```




```
contour(eighth_run[['x_store']])
```




```
square_mean <- mean(eighth_run[['x_store']][, 1] ^ 2)
RMSE <- sqrt(mean((eighth_run[['x_store']][, 1] ^ 2 - 1.0384) ^ 2))
print('Mean is')
```

```
## [1] "Mean is"
```

```
print(square_mean)
```

```
## [1] 0.977031
```

```
print('RMSE is')
```

```
## [1] "RMSE is"
```

```
print(RMSE)
```

```
## [1] 1.85486
```

```
print('MCSE is (batch size 1000)')
```

```
## [1] "MCSE is (batch size 1000)"
```

```
print(batchSE(mcmc(eighth_run[['x_store']]), 1000))
```

```
##  [1] 0.1025311 0.1161782 0.1758728 0.1248175 0.1359840 0.1388455 0.1270784
##  [8] 0.1626771 0.1678146 0.1695148
```

```
print('Effective n is')
```

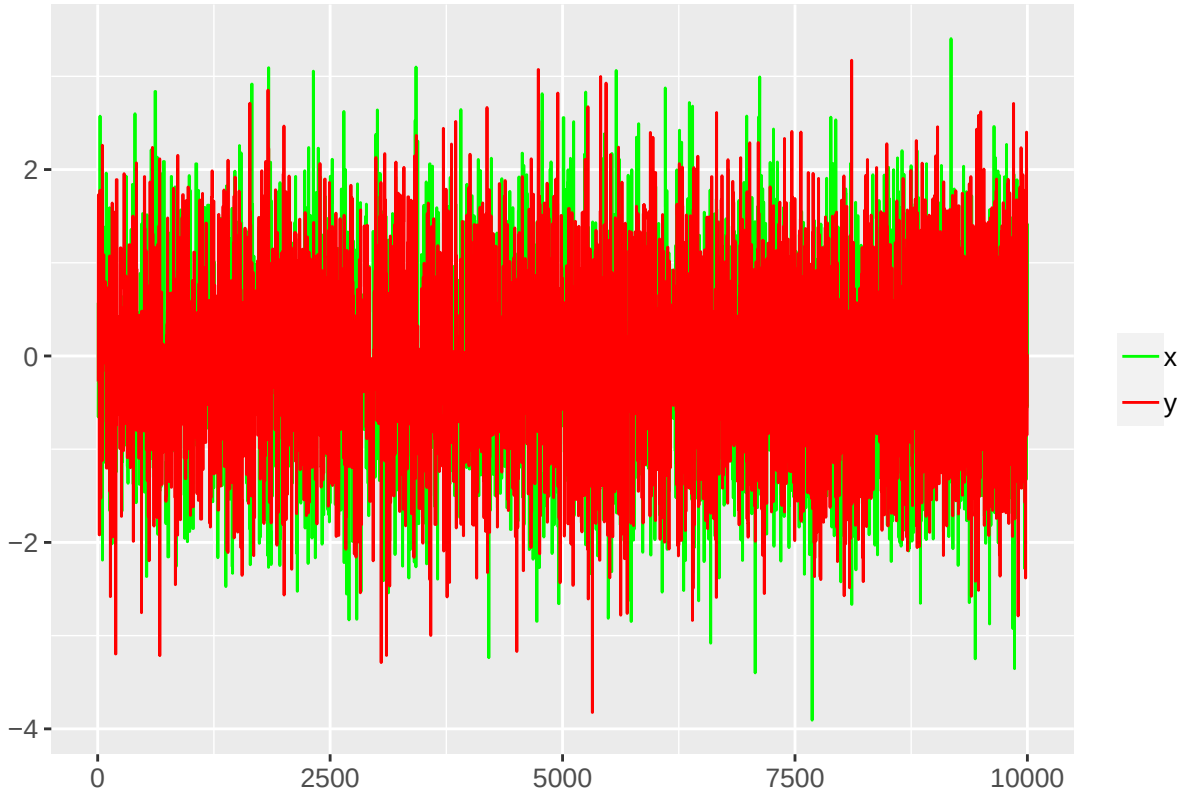
```
## [1] "Effective n is"
```

```r
print(effectiveSize(mcmc(eighth_run[['x_store']])))
```

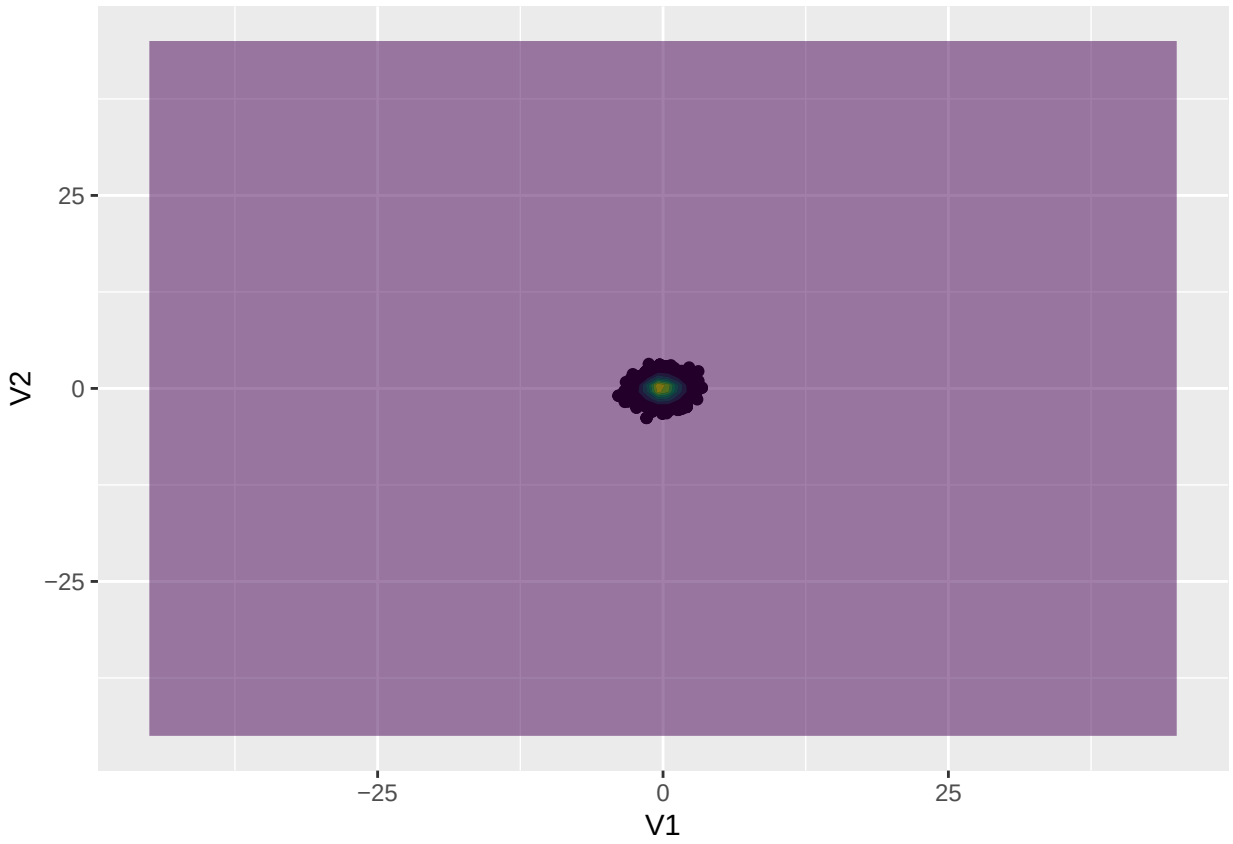
```
##     var1     var2     var3     var4     var5     var6     var7     var8
## 58.60983 56.26457 72.82165 95.02066 46.68450 40.38853 61.83924 35.92807
##     var9    var10
## 28.59511 29.31403
```

Localised RWM

```r
k_max <- 5

init_sigma <- diag(dimension)
init_sigmas <- matrix(nrow = dimension * k_max, ncol = dimension)
for (i in 0:(k_max - 1)) {
  init_sigmas[(dimension * i + 1):(dimension * (i + 1)), ] <- init_sigma
}


init <- list(X = rep(0, dimension),
             params = list(mus = matrix(rep(0, dimension * k_max),
                                        nrow = k_max,
                                        ncol = dimension),
                           sigmas = init_sigmas,
                           weights = rep(1 / k_max, k_max),
                           lambdas = rep(1, k_max)),
             gamma = gamma)

sample <- "Local_RWM_sample"
update <- "Local_RWM_update"
logpi <- AM_logpi
logpi_args <- list(mu = rep(0, dimension),
                   sigma = M_sq)
seed <- 200
learn_in <- 100
nits <- 10000
gamma <- gamma
#stop_after <- 5000

ninth_run <- adapt(init = init,
                   sample = sample,
                   update = update,
                   logpi = logpi,
                   logpi_args = logpi_args,
                   learn_in = learn_in,
                   nits = nits,
                   gamma = gamma)
tracer(ninth_run[['x_store']][, 1:2],
       0.7,
       " ",
       " ",
       " ")
```




```
contour(ninth_run[['x_store']])
```




```
square_mean <- mean(ninth_run[['x_store']][, 1] ^ 2)
RMSE <- sqrt(mean((ninth_run[['x_store']][, 1] ^ 2 - 1.0384) ^ 2))
print('Mean is')
```

```
## [1] "Mean is"
```

```
print(square_mean)
```

```
## [1] 0.9043993
```

```
print('RMSE is')
```

```
## [1] "RMSE is"
```

```
print(RMSE)
```

```
## [1] 1.313579
```

```
print('MCSE is (batch size 1000)')
```

```
## [1] "MCSE is (batch size 1000)"
```

```
print(batchSE(mcmc(ninth_run[['x_store']]), 1000))
```

```
##  [1] 0.01334973 0.01907973 0.01851550 0.01659112 0.01151745 0.01318639
##  [7] 0.01674913 0.01740897 0.01503908 0.01400031
```

```
print('Effective n is')
```

```
## [1] "Effective n is"
```

```r
print(effectiveSize(mcmc(ninth_run[['x_store']])))
```

```
##     var1     var2     var3     var4     var5     var6     var7     var8
## 2927.647 2897.667 2718.784 2779.756 2817.123 2976.107 3049.309 2673.480
##     var9    var10
## 2834.559 2721.607
```

Data Viz

```r
# mat1 <- as.data.frame(matrix(rnorm(12), nrow = 6, ncol = 2))
# mat2 <- as.data.frame(matrix(rnorm(12), nrow = 6, ncol = 2))
#
# p1 <- ggplot(mat1, aes(x = V1, y = V2)) +
#   geom_point(colour = "red")
# p2 <- ggplot(mat2, aes(x = V1, y = V2)) +
#   geom_point(colour = "blue")
#
# plotlist <- list()
#
# plotlist[['1']] <- p1
# plotlist[['2']] <- p2
#
# # test how value/reference works with list elements:
# plotlist[['1']] <- plotlist[['1']] + geom_line()
# grid.arrange(grobs = plotlist, ncol = 2, nrow = 1)
# # changes the object within the list (i.e. reference)
#
# # so you can pass a list of 'grobs' to grid.arrange. nice
#
# # make sure the axes are the same
# ggsave('test1.png',
#        plot = plotlist[['1']],
#        device = "png")
# ggsave('test2.png',
#        plot = plotlist[['2']],
#        device = "png")
# # the above could be implemented within the 'adapt' function

# sigma <- 0.5 * matrix(c(101, 99, 99, 101), nrow = 2, ncol = 2)
# ellipse_df <- as.data.frame(ellipse(sigma, npoints = 50))
# ellipse_df <- cbind(ellipse_df, e_vec = rep(FALSE, nrow(ellipse_df)))
#
# ellipse_df <- rbind(ellipse_df, c(1, 0, TRUE))
# ellipse_df <- rbind(ellipse_df, c(0, 1, TRUE))
#
# ggplot(ellipse_df, aes(x = x,
#                        y = y)) +
#   geom_point(aes(colour = as.factor(e_vec),
#                  shape = as.factor(e_vec))) +
#   scale_shape_manual(values = c(19, 3))

# animate_sigmas_local_RWM(sigmas = fourth_run[['sigmas']],
#                          target_sigma = 0.25 * matrix(c(101, 99, 99, 101),
```

```
#                                                       nrow = 2,
#                                                       ncol = 2),
#                           target_mu = c(20, -20),
#                           k_max = 4,
#                           colours = c("darkgoldenrod1",
#                                       "midnightblue",
#                                       "darkolivegreen4",
#                                       "darkorchid1",
#                                       "tomato"),
#                           mus = fourth_run[['mus']],
#                           weights = fourth_run[['weights']],
#                           chain = fourth_run[['x_store']])

# test_df <- data.frame(x = c(1, 2, 3, 4, 5, 6, 7, 8),
#                       y = c(6, 8, 2, 4, 3, 8, 1, 1),
#                       component = c(1, 1, 2, 2, 3, 3, 4, 4),
#                       weight = c(0.2, 0.2, 0.1, 0.1, 0.3, 0.3, 0.2, 0.2))
#
# ggplot(test_df, aes(x = x, y = y, colour = as.factor(component))) +
#   geom_point(shape = 15,
#              aes(size = weight)) +
#   scale_colour_manual(values = c("darkgoldenrod1",
#                                  "midnightblue",
#                                  "darkolivegreen4",
#                                  "darkorchid1"))

# animate_PCA(sigmas = fifth_run[['sigmas']],
#             target_sigma = 0.5 * matrix(c(100, 0, 0, 1), nrow = 2, ncol = 2),
#             target_mu = c(20, -20),
#             chain = fifth_run[['x_store']],
#             m = 2,
#             e_vectors = fifth_run[['e_vectors']],
#             mus = fifth_run[['mus']],
#             rescale = 20)

# animate_sigmas_CAM_CAS(sigmas = second_run[['sigmas']],
#                        target_sigma = 0.25 * matrix(c(101, 99, 99, 101),
#                                                     nrow = 2,
#                                                     ncol = 2),
#                        target_mu = c(20, -20),
#                        mus = second_run[['mus']],
#                        chain = second_run[['x_store']],
#                        lambdas = second_run[['lambdas']])

# target_sigma <- 0.25 * matrix(c(101, 99, 99, 101),
#                               nrow = 2,
#                               ncol = 2)
# target_mu <- c(20, -20)
# sigma <- 0.25 * matrix(c(101, -99, -99, 101),
#                        nrow = 2,
#                        ncol = 2)
# mu <- c(-20, 20)
# state <- c(0, 0)
```

```
# # plot the sigmas as contour plots, the mus as points, the
# # state as a point
# target_points <- as.data.frame(rmvnorm(1000,
#                          mean = target_mu,
#                          sigma = target_sigma))
# points <- as.data.frame(rmvnorm(1000,
#                   mean = mu,
#                   sigma = sigma))
#
# eigen_vecs <- matrix(c(1, 2, 3, 4), nrow = 2, ncol = 2)
# rescale <- 2
# eigen_vecs <- rescale * eigen_vecs
#
# eigen_df_1 <- data.frame(x1 = mu[1],
#                          y1 = mu[2],
#                          x2 = mu[1] + eigen_vecs[1, 1],
#                          y2 = mu[2] + eigen_vecs[1, 2])
# eigen_df_2 <- data.frame(x1 = mu[1],
#                          y1 = mu[2],
#                          x2 = mu[1] + eigen_vecs[2, 1],
#                          y2 = mu[2] + eigen_vecs[2, 2])
#
# p <- ggplot(target_points, aes(x = V1, y = V2, alpha = 1)) +
#   theme(panel.grid.major = element_blank(),
#         panel.grid.minor = element_blank(),
#         panel.background = element_blank(),
#         legend.position = "none",
#         axis.line=element_blank(),
#         axis.text.x=element_blank(),
#         axis.text.y=element_blank(),
#         axis.ticks=element_blank(),
#         axis.title.x=element_blank(),
#         axis.title.y=element_blank()) +
#   geom_density_2d(aes(alpha = 1, colour = "red")) +
#   geom_density_2d(data = points,
#                   aes(x = V1, y = V2, alpha = 1, colour = "blue")) +
#   geom_point(data = as.data.frame(t(mu)),
#              aes(x = V1, y = V2, colour = "red", alpha = 1, size = 1.5),
#              inherit.aes = FALSE) +
#   geom_point(data = as.data.frame(t(state)),
#              aes(x = V1, y = V2, colour = "green", alpha = 1, size = 1.5),
#              inherit.aes = FALSE) +
#   xlab("") +
#   ylab("")
#
# # Below proves you can create standalone ggproto objects then add to the plot if
# # necessary.
# seg <- geom_segment(aes(x = x1,
#                     y = y1,
#                     xend = x2,
#                     yend = y2,
#                     colour = "segment"),
#                data = eigen_df_1,
```

```
#                 arrow = arrow(length = unit(0.5,"cm")),
#                 size = 1)
# seg1 <-  geom_segment(aes(x = x1,
#                     y = y1,
#                     xend = x2,
#                     yend = y2,
#                     colour = "segment"),
#                 data = eigen_df_2,
#                 arrow = arrow(length = unit(0.5,"cm")),
#                 size = 1)
#
# p <- p + seg + seg1
#
# grid.arrange(p, nrow = 1, ncol = 1)

# target_sigma <- 0.5 * matrix(c(100, 0, 0, 1),
#                              nrow = 2,
#                              ncol = 2)
# target_mu <- c(20, -20)
# nine_plot(obj = fifth_run,
#           target_sigma = target_sigma,
#           target_mu = target_mu,
#           scheme = "PCA",
#           frames = c(2,
#                      100,
#                      910,
#                      1617,
#                      2324,
#                      3334,
#                      5051,
#                      6162,
#                      7000,
#                      8000,
#                      10000),
#           ncol = 3,
#           nrow = 4)

# for (i in 1:10000) {
#   off_dia_1 <- first_run[['sigmas']][2 * (i - 1) + 1, 2]
#   off_dia_2 <- first_run[['sigmas']][2 * i, 1]
#   if (off_dia_1 != off_dia_2) {
#     print("wth")
#   }
# }

# animate_sigmas(sigmas = sixth_run[['sigmas']],
#             target_sigma = 0.25 * matrix(c(101, 99, 99, 101), nrow = 2, ncol = 2),
#             target_mu = c(20, -20),
#             chain = sixth_run[['x_store']],
#             mus = fifth_run[['mus']])

# target_sigma <- 0.25 * matrix(c(101, 99, 99, 1),
#                              nrow = 2,
```

```
#                               ncol = 2)
# target_mu <- c(20, -20)
# directions <- matrix(c(1, 0, 0, 1), ncol = 2, nrow = 2)
# frames <- 100 * 1:100 - 99
# nine_plot(obj = second_run,
#           target_sigma = target_sigma,
#           target_mu = target_mu,
#           scheme = "CAM_CAS",
#           frames = frames,
#           ncol = 10,
#           nrow = 14,
#           directions = directions,
#           rescale = 25,
#           output_png = TRUE)

# target_sigma <- 0.5 * matrix(c(100, 0, 0, 1),
#                               nrow = 2,
#                               ncol = 2)
# target_mu <- c(20, -20)
# frames <- 100 * 1:100 - 99
# nine_plot(obj = fifth_run,
#           target_sigma = target_sigma,
#           target_mu = target_mu,
#           scheme = "PCA",
#           frames = frames,
#           ncol = 11,
#           nrow = 10,
#           output_png = TRUE)

# target_sigma <- 0.25 * matrix(c(101, 99, 99, 1),
#                               nrow = 2,
#                               ncol = 2)
# target_mu <- c(20, -20)
# frames <- 100 * 1:100 - 99
# nine_plot(obj = first_run,
#           target_sigma = target_sigma,
#           target_mu = target_mu,
#           scheme = "AM",
#           frames = frames,
#           ncol = 11,
#           nrow = 10,
#           output_png = TRUE)

# # plot the difference in scales to show potential for ergodicity even tho gamma
# # returns a constant
# difference_scale1 <- second_run$lambdas[2:10000, 1] - second_run$lambdas[1:9999, 1]
# difference_scale2 <- second_run$lambdas[2:10000, 2] - second_run$lambdas[1:9999, 2]
#
# df_scale1 <- data.frame(difference = difference_scale1, scale = "1", iteration = 1:9999)
# df_scale2 <- data.frame(difference = difference_scale2, scale = "2", iteration = 1:9999)
#
# df_big <- rbind(df_scale1, df_scale2)
# p <- ggplot(df_big, aes(x = iteration, y = difference, colour = scale)) +
```

```
#   geom_line()
# p

# difference_mu <- first_run$mus[2:10000] - first_run$mus[1:9999]
# df <- data.frame(difference = difference_mu, iteration = 1:9999)
#
# p <- ggplot(df, aes(x = iteration, y = difference, colour = "dummy")) +
#   geom_line() +
#     theme(legend.position = "none")
# p
# ideal_difference <- 0.001 * (first_run$x_store[, 1] - 20)
# noise <- 0.001 * (20 - first_run$mus[, 1])
#
# df_ideal <- data.frame(difference = ideal_difference, iteration = 1:10000, type = "Optimal Difference
# df_noise <- data.frame(difference = noise, iteration = 1:10000, type = "Noise")
#
# df <- rbind(df_ideal, df_noise)
# p <- ggplot(df, aes(x = iteration, y = difference, colour = type)) +
#   geom_line()
# p
# save(first_run, file = "decomp_example.RData")
```

RWM in Gaussian LPMs

Log densities and gradients

```
# log densities and gradients
ld_all_categorical <- function(z,
                               dyads,
                               gamma2,
                               tau,
                               gaussmat) {
  return(ld_gaussian(z, gaussmat) + ld_nogaussian_categorical(z, dyads, gamma2, tau))
}

ld_gaussian <- function(z,
                        gaussmat) {
  # the gaussian part of the targetted density
  zzt <- z %*% t(z)
  n <- nrow(z)
  res <- 0
  for (i in 1:n) {
    res <- res + sum(zzt[i, ] * gaussmat[, i])
  }
  res <- res - res / 2
  return(res)
}

ld_nogaussian_categorical <- function(z,
                                      dyads,
                                      gamma2,
                                      tau) {
```

```
  # the non-gaussian part of the targetted density
  # save time by pre-computing the squares (they assume they use each at least once)
  squares <- vector(length = nrow(z))
  for (i in 1:nrow(z)) {
    squares[i] <- sum(z[i, ] ^ 2)
  }

  ndyads <- nrow(dyads)

  result <- 0
  for (i in 1:ndyads) {
    result <- result + log(1 - tau[dyads[i, 3] + 1] * exp(- (0.5 / gamma2) * (squares[dyads[i, 1]] + squ
  }
  return(result)
}

gradient_all_categorical <- function(z_, # the latent positions
                                     dyads, # the dyads without edges to consider
                                     gamma2, # the variance of the link function
                                     tau, # the probability of an edge at distance 0
                                     gaussmat) {
  nnodes <- nrow(z_)
  d <- ncol(z_)
  ret <- matrix(nrow = nnodes, ncol = d)

  ret <- gradient_nogaussian_categorical(z_, dyads, gamma2, tau)

  # add a gaussian part
  ret <- ret + gaussmat %*% z_

  return(ret)
}

gradient_nogaussian_categorical <- function(z, # the latent positions
                                            dyads, # the dyads without edges to consider
                                            gamma2, # variance of the link fn.
                                            tau # prob of an edge at dist. 0
) {
  nnodes <- nrow(z)
  d <- ncol(z)
  ndyads <- nrow(dyads)
  ret <- matrix(0, nrow = nnodes, ncol = d)
  squares <- vector(length = nnodes)
  for (i in 1:nnodes) {
    squares[i] <- sum(z[i, ] * z[i, ])
  }

  for (i in 1:ndyads) {
    front <- -1 / (gamma2 * (exp((0.5 / gamma2) * (squares[dyads[i, 1]] + squares[dyads[i, 2]] - 2 * su
    for (j in 1:d) {
      entry <- front * (z[dyads[i, 1], j] - z[dyads[i, 2], j])
      ret[dyads[i, 1], j] <- ret[dyads[i, 1], j] + entry
      ret[dyads[i, 2], j] <- ret[dyads[i, 2], j] - entry
```

```
    }
  }
  return(ret)
}
```

Initializing

```
# initializing
createA <- function(Alabels, tau, priorprecision, gamma2, d = 2){
  sigma2inv = gamma2
  ztrue <- t(mvtnorm::rmvnorm(d, sigma = 1/sigma2inv * solve(priorprecision)))
  nnodes <- nrow(Alabels)
  A <- matrix(0, nrow = nnodes, ncol = nnodes)
  diag(A) <- 0
  for(i in 1:(nnodes - 1)){
    for(j in (i + 1):nnodes){
      if(Alabels[i, j] != -1){
        A[i, j] <- stats::rbinom(1, 1, tau[1 + Alabels[i,j]] * exp(-0.5 * t(ztrue[i, ] - ztrue[j, ]) %*%
        A[j, i] <- A[i, j]
      }
    }
  }
  return(list(A = A, z = ztrue, zraw = ztrue * sqrt(gamma2)))
}

MLE_initialize <- function(A,
                           Alabels,
                           prior,
                           z_init,
                           tau,
                           gamma2,
                           eps = 0.001,
                           niterations = 100) {
  sigma2inv <- gamma2
  ntaus <- length(tau) # no. of unique categories
  n <- nrow(z_init)
  d <- ncol(z_init)
  z <- z_init

  # make the laplacian
  degree <- vector(length = n)

  for (i in 1:n) {
    degree[i] <- sum(A[i, ])
  }

  Laplacian <- diag(degree) - A

  # create stuff needed to work with dyads
  ret <- getdyadsmat(Alabels, A, ntaus)
  dyads <- ret$dyads # holds the endpoints for each non-edge dyad and its category
  ndyads <- nrow(dyads) # no. of dyads
```

```
  gaussmat <- Laplacian + sigma2inv * prior
  for (i in 1:niterations) {
    z <- z - eps * gradient_all_categorical(z, dyads, 1, tau, gaussmat)
    ll2 <- ld_all_categorical(z, dyads, 1, tau, gaussmat)
  }
  return(z)
}

getdyadsmat <- function(Alabels,
                        A,
                        ntaus) {
  # Obtains the necessary quantities related to the dyads from A and Alabels.
  # note -1 in Alabels indicates a missing value
  inds <- which(Alabels > -1)
  ndyads <- (length(inds) - sum(A) - nrow(A)) / 2 # no. of known non-edge dyads
  dyads <- matrix(nrow = ndyads, ncol = 3) # matrix with each known non-edge dyad and category
                                           # as rows
  nedges_categories <- as.vector(rep(0, ntaus)) # vector with how many known edges in each category
  ndyadsincat <- vector(length = length(ntaus)) # vector with how many known non-edges in each
                                                # category
  dyadids <- vector() # vector with blocks of non edge ids (i.e. positions in dyads) for each
                      # category

  k <- 1
  for (i in 2:nrow(Alabels)) {
    for (j in 1:(i - 1)) {
      if (Alabels[i, j] >= 0) {
        if (A[i, j] == 1) {
          nedges_categories[Alabels[i, j] + 1] <- nedges_categories[Alabels[i, j] + 1] + 1
        } else {
          dyads[k, 1] <- i
          dyads[k, 2] <- j
          dyads[k, 3] <- Alabels[i, j]
          k <- k + 1
        }
      }
    }
  }
  # filling the above objects
  # i ranges from 0 to ntaus - 1 because we are iterating over categories
  # of dyads NOT over indices.
  for (i in 0:(ntaus - 1)) {
    ids <- which(dyads[, 3] == i) # find dyads in category i
    ndyadsincat[i + 1] <- length(ids) # count how many there are
    dyadids <- c(dyadids, ids) # store them
  }

  return(list('dyads' = dyads,
              'dyadids' = dyadids,
              'ndyadsincat' = ndyadsincat,
              'nedges_categories' = nedges_categories))
}
```

MCMC run

```
n <- 50
d <- 2
Alabels <- matrix(0, nrow = n, ncol = n)
tau <- c(0.2)
gamma2 <- 0.2
# these are the priors for the columns of A only NOT the whole state
priorprecision <- diag(n)
nruns <- 1000
sample_tau <- TRUE
sample_gamma2 <- TRUE
graphseed <- 137
mcmcseed <- 300

set.seed(graphseed)
creation <- createA(Alabels, tau, priorprecision, gamma2, d = 2)
A <- creation$A
set.seed(mcmcseed)

tau_init = rep(0.5, 1 + max(Alabels))
gamma2_init = 1
alpha_param = rep(1, 1 + max(Alabels))
beta_param = rep(1, 1 + max(Alabels))
a_param = 1
b_param = 1

# The following is just a re-naming so that the user accessed function
# reflects the notation they used in the paper.
sigma2inv_init = gamma2_init
sample_sigma2inv = sample_gamma2
beta1 = alpha_param
beta2 = beta_param
sigparam1 = a_param
sigparam2 = b_param

z_init <- matrix(rnorm(nrow(A) * d), nrow = nrow(A), ncol = d)

z_init <- MLE_initialize(A,
                         Alabels,
                         priorprecision,
                         z_init,
                         tau_init,
                         sigma2inv_init,
                         eps = 0.005,
                         niterations = 300)

# get the graph theory stuff
ntaus <- length(tau)
ret <- getdyadsmat(Alabels, A, ntaus)
dyads <- ret$dyads # holds the endpoints for each non-edge dyad and its category

# make the laplacian
degree <- vector(length = n)
```

```
for (i in 1:n) {
  degree[i] <- sum(A[i, ])
}

Laplacian <- diag(degree) - A
# update the Laplacian and inverse using new precision (sigma2inv)
gaussmat <- Laplacian + sigma2inv_init * priorprecision

# the state is a vector of length d * n + 2 consisting of a concatenation
# of the columns of Z and tau and gamma ^ 2
X <- vector(length = n * d + 2)
for (j in 1:d) {
  X[(n * (j - 1) + 1):(n * j)] <- z_init[, j]
}

X[n * d + 1] <- tau_init
X[n * d + 2] <- gamma2_init
# set the prior covar. of the state
priorcovariance <- diag(n * d + 2)

init <- list(X = X,
             params = list(mu = rep(0, n * d + 2),
                           sigma = priorcovariance,
                           guassmat = gaussmat,
                           dyads = dyads),
             gamma = gamma)


sample <- "LPM_RWM_sample"
# update only gaussmat in light of the updated gamma2
update <- "LPM_RWM_update"
logpi <- AM_logpi
logpi_args <- list(A = A,
                   dyads = dyads)
seed <- mcmcseed
learn_in <- 0
nits <- 10000
gamma <- gamma
#stop_after <- 5000

time1 <- Sys.time()
# need to change the logpi function
# first_run_LPM <- adapt(init = init,
#                    sample = sample,
#                    update = update,
#                    logpi = logpi,
#                    logpi_args = logpi_args,
#                    learn_in = learn_in,
#                    nits = nits,
#                    gamma = gamma,
#                    seed = seed)
# runtime <- as.numeric(difftime(Sys.time(), time1, units = "secs"))
#
```

```
# tracer(first_run[['x_store']],
#        1,
#        " ",
#        "Iteration",
#        " ")
# contour(first_run[['x_store']])
#
# print('MCSE is (batch size 1000)')
# print(batchSE(mcmc(first_run[['x_store']]), 1000))
# print('Effective n is')
# print(effectiveSize(mcmc(first_run[['x_store']])))
# # make a new chain to plot the Gelman-Rubin diagnostic
#
# first_run_LPM_alt2 <- adapt(init = init,
#                   sample = sample,
#                   update = update,
#                   logpi = logpi,
#                   logpi_args = logpi_args,
#                   learn_in = learn_in,
#                   nits = nits,
#                   gamma = gamma,
#                   seed = seed + 100)
# first_run_LPM_alt3 <- adapt(init = init,
#                   sample = sample,
#                   update = update,
#                   logpi = logpi,
#                   logpi_args = logpi_args,
#                   learn_in = learn_in,
#                   nits = nits,
#                   gamma = gamma,
#                   seed = seed + 200)
# first_run_LPM_alt4 <- adapt(init = init,
#                   sample = sample,
#                   update = update,
#                   logpi = logpi,
#                   logpi_args = logpi_args,
#                   learn_in = learn_in,
#                   nits = nits,
#                   gamma = gamma,
#                   seed = seed + 300)
# gelman.plot(x = mcmc.list(mcmc(first_run_LPM[['x_store']]),
#                           mcmc(first_run_LPM_alt2[['x_store']]),
#                           mcmc(first_run_LPM_alt3[['x_store']]),
#                           mcmc(first_run_LPM_alt4[['x_store']])),
#             bin.width = 100,
#             max.bins = 120,
#             autoburnin = FALSE,
#             ylab = 'R')
```

Adaptive Metropolis, with a non-increasing gamma for the latter half of the run

```
init <- list(X = c(-20, 20),
             params = list(mu = c(-20, 20),
                           sigma = 0.25 * matrix(c(101, -99, -99, 101),
                                                 nrow = 2,
                                                 ncol = 2)),
             gamma = gamma)
sample <- "AM_sample"
update <- "AM_update"
logpi <- AM_logpi
logpi_args <- list(mu = c(20, -20),
                   sigma = 0.25 * matrix(c(101, 99, 99, 101),
                                         nrow = 2,
                                         ncol = 2))
seed <- 2000
learn_in <- 0
nits <- 10000
gamma <- gamma
#stop_after <- 5000

# first_run_a <- adapt(init = init,
#                      sample = sample,
#                      update = update,
#                      logpi = logpi,
#                      logpi_args = logpi_args,
#                      learn_in = learn_in,
#                      nits = nits,
#                      gamma = gamma,
#                      seed = seed)
# tracer(first_run_a[['x_store']],
#        1,
#        " ",
#        "Iteration",
#        " ")
# contour(first_run_a[['x_store']])
#
# print('MCSE is (batch size 1000)')
# print(batchSE(mcmc(first_run_a[['x_store']]), 1000))
# print('Effective n is')
# print(effectiveSize(mcmc(first_run_a[['x_store']])))
# # make a new chain to plot the Gelman-Rubin diagnostic
#
# first_run_alt2_a <- adapt(init = init,
#                      sample = sample,
#                      update = update,
#                      logpi = logpi,
#                      logpi_args = logpi_args,
#                      learn_in = learn_in,
#                      nits = nits,
#                      gamma = gamma,
#                      seed = seed + 100)
```

```
# first_run_alt3_a <- adapt(init = init,
#                   sample = sample,
#                   update = update,
#                   logpi = logpi,
#                   logpi_args = logpi_args,
#                   learn_in = learn_in,
#                   nits = nits,
#                   gamma = gamma,
#                   seed = seed + 200)
# first_run_alt4_a <- adapt(init = init,
#                   sample = sample,
#                   update = update,
#                   logpi = logpi,
#                   logpi_args = logpi_args,
#                   learn_in = learn_in,
#                   nits = nits,
#                   gamma = gamma,
#                   seed = seed + 300)
# gelman.plot(x = mcmc.list(mcmc(first_run_a[['x_store']]),
#                           mcmc(first_run_alt2_a[['x_store']]),
#                           mcmc(first_run_alt3_a[['x_store']]),
#                           mcmc(first_run_alt4_a[['x_store']])),
#             bin.width = 100,
#             max.bins = 120,
#             autoburnin = FALSE,
#             ylab = 'R')
#
# ideal_difference <- 0.001 * (first_run_a$x_store[, 1] - 20)
# noise <- 0.001 * (20 - first_run_a$mus[, 1])
#
# df_ideal <- data.frame(difference = ideal_difference, iteration = 1:10000, type = "Optimal Difference
# df_noise <- data.frame(difference = noise, iteration = 1:10000, type = "Noise")
#
# df <- rbind(df_ideal, df_noise)
# p <- ggplot(df, aes(x = iteration, y = difference, colour = type)) +
#   geom_line()
# p
```