Computational efficiency comparison of MCMC algorithms for non-Gaussian state space models

Jouni Helske

13 June 2017

Introduction

The `R` (Team 2017) package `stannis` ("Stan 'n' IS") provides a small case study for Bayesian inference of structural time series (Harvey 1989) with Poisson distributed observations with different Markov chain Monte Carlo algorithms. Our model belongs to a class of exponential family state space models. These models contain a Gaussian latent process $\alpha_1 \ldots, \alpha_n$ evolving in time, which is only undirectly observed through observing $y_1, \ldots, y_n$. Consider for example a random walk model

$$y_t \sim Poisson(\exp(x_t)), \alpha_{t+1} = \alpha_t + \eta_t, \eta_t \sim N(0, \sigma^2_\eta)$$

with prior $\alpha_1 \sim N(a_1, P_1)$ and unknown σ^2_η. Performing computationally efficient inference of these type of models using `Stan` (Stan Development Team 2016b) can be difficult in practice due to large parameter space (here $n+1$ parameters corresponding to the unknown states α and standard deviation σ^2_η) with strong correlation structures. On the other hand, in case of Gaussian observations, the marginal likelihood $p(y)$ is tractable via the Kalman filter, so a simple Metropolis-type Markov chain Monte Carlo (MCMC) targeting the hyperparameters θ (here σ^2_η) can be easily constructed. Then, using the samples $\{\theta^{(i)}\}_{i=1}^N$, we can obtain samples from the joint posterior (α, θ) by sampling the states $\alpha^{(i)}$ given y and $\theta^{(i)}$ using efficient simulation smoother (Durbin and Koopman 2002) algorithms for $i = 1, \ldots, N$. This approach reduces the scale of the MCMC problem significantly as the dimension of θ is typically much smaller than n.

Unfortunately, the marginal likelihood $p(y)$ is intractable in case of non-Gaussian observations. In this case, one option is to use so called pseudo-marginal MCMC, or more specifically particle MCMC approach (Andrieu, Doucet, and Holenstein 2010), where we replace $p(y)$, with its unbiased estimate, which can be found using some variant particle filter. However, these methods are often computationally demanding unless access to efficient particle filter algorithm (PF) is available. For state space models with linear-Gaussian state dynamics and suitable observation densities such as those belonging to exponential family, we suggest using the ψ-PF (Vihola, Helske, and Franks 2016) which takes account readily available Gaussian approximations of the original model. With ψ-PF we can typically keep the number of particles low which is a key feature in making the particle MCMC methods computationally efficient.

Vihola, Helske, and Franks (2016) introduces an alternative approach based on importance sampling type correction (IS-MCMC), where first an MCMC run targeting approximate marginal posterior is performed, and then an straighforwardly parallelisable importance sampling type scheme is used to correct for the bias. The MCMC algorithms used in Vihola, Helske, and Franks (2016) were based on robust adaptive random walk Metropolis algorithm (Vihola 2012), implemented in the `bssm` package (Helske and Vihola 2017). This vignette provides short non-exhaustive simulation study of the computational efficiency of several Metropolis-type MCMC algorithms as in Vihola, Helske, and Franks (2016), NUTS-MCMC of `Stan` (Hoffman and Gelman 2014), and hybrid NUTS-IS-MCMC provided by `stannis` package.

The core pieces of the `bssm` package are written in `C++` using `Armadillo` (Sanderson and Curtin 2016) linear algebra library, interfaced with `Rcpp` (Eddelbuettel and François 2011) and `RcppArmadillo` (Eddelbuettel and Sanderson 2014) packages, whereas `Stan` was used via the `rstan` package (Stan Development Team 2016a). The `stannis` package contains some partially modified parts of `bssm` for the particle filtering task, where as the NUTS-sampler of `Stan`is used via calls to functions of `rstan`.

Model and data

In our experiment, we use a local linear trend model with Poisson observations, defined as

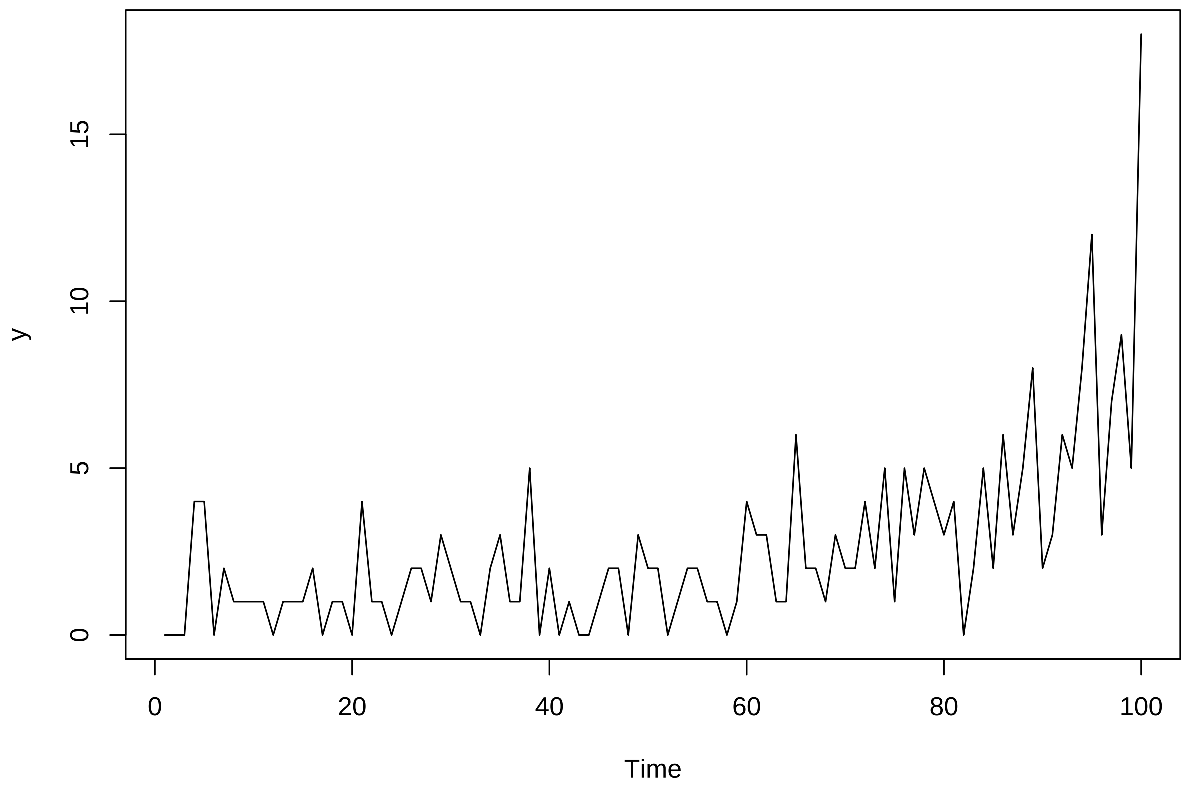
$$
\begin{aligned}
y_t &\sim \text{Poisson}(\exp(\mu_t)) \\
\mu_{t+1} &= \mu_t + \nu_t + \sigma_\eta \eta_t, \qquad & \eta_t \sim N(0,1) \\
\mu_{t+1} &= \nu_t + \sigma_\xi \xi_t, \qquad & \xi_t \sim N(0,1).
\end{aligned}
$$

We simulated times series of length $n = 100$ with $\sigma_\eta = \sigma_\xi = 0.01$, and $\mu_1 = \nu_1 = 0$.

```r
set.seed(123)
n <- 100

slope <- cumsum(c(0, rnorm(n - 1, sd = 0.01)))
level <- cumsum(slope + c(0, rnorm(n - 1, sd = 0.01)))
y <- rpois(n, exp(level))

ts.plot(y)
```



bssm

We can run different MCMC algorithms with `bssm` by first building the model:

```r
model <- ng_bsm(y, P1 = diag(c(10, 0.1)),
  sd_level = halfnormal(0.01, 1),
  sd_slope = halfnormal(0.01, 0.1), distribution = "poisson")
```

We define half-Normal priors (i.e. zero-mean Gaussian distribution folded at zero) for the standard deviation parameters. The first argument to `halfnormal` defines the initial value, whereas the second defines the standard deviation of half-Normal distribution. For illustrative purposes the initial values correspond to the true values.

Function `run_mcmc` performs the Markov chain Monte Carlo, where the actual algorithm depends on several arguments:

```r
da_mcmc <- run_mcmc(model, n_iter = 2000, nsim_states = 10)
```

The default option for non-Gaussian non-linear state space models is the delayed acceptance (DA) (Christen and Fox 2005, Banterle et al. (2015)) pseudo-marginal algorithm using ψ-PF with `nsim_states` particles. This is natural option in DA setting as we use the same Gaussian approximation in the initial acceptance step of DA as in ψ-PF. ψ-PF is also used in `stannis`.

Here we use `n_iter = 2000` MCMC iterations (by default, first half is discarded as burnin), which is typically too little for reliable inference but illustrates the workflow. The print method of the `run_mcmc` output gives basic information about the MCMC run:

```r
da_mcmc
```

```
##
## Call:
## run_mcmc.ng_bsm(object = model, n_iter = 2000, nsim_states = 10)
##
## Iterations = 1001:2000
## Thinning interval = 1
## Length of the final jump chain = 277
##
## Acceptance rate after the burn-in period:  0.276
##
## Summary for theta:
##
##                 Mean          SD        SE-IS        SE-AR           SE
## sd_level 0.069095255 0.047767847 0.0041240466 0.0052173081 0.0066504183
## sd_slope 0.008245962 0.006021337 0.0004595262 0.0006341608 0.0007831502
##
## Effective sample sizes for theta:
##
##            ESS-IS   ESS-AR
## sd_level 134.2282 83.82593
## sd_slope 117.8115 90.15443
##
## Summary for alpha_100:
##
##             Mean         SD       SE-IS       SE-AR          SE
## level 2.32783204 0.18901993 0.014977954 0.015005063 0.021201203
## slope 0.06335016 0.03776722 0.002824202 0.002829314 0.003997641
##
## Effective sample sizes for alpha:
##
##         ESS-IS   ESS-AR
## level 129.0596 158.6863
## slope 115.7391 178.1837
##
## Run time:
##    user  system elapsed
##   1.188   0.004   1.215
```

Stan

For `Stan`, users typically need to write their own model using Stan language (see Appendix A), which is then automatically compiled to C++ when calling `stan` or `sampling` functions of `rstan`. But in this case the `stannis` package contains our models as a precombiled object, which we can access as `stannis:::stanmodels$llt_poisson`. First we define our data as a list `stan_data` and similarly for initial values `stan_inits` (this is optional, and not all initial values need to be provided), where the latter is a a list of lists (one for each chain). We then call the function `sampling`, again and print some summary statistics using appropriate printing method.

```r
stan_data <- list(n = n, y = y, a1 = c(0, 0), P1 = diag(c(10, 0.1)),
  sd_prior_means = rep(0, 2), sd_prior_sds = c(1, 0.1))
stan_inits <- list(list(theta = c(0.01, 0.01),
  slope_std = rep(0, n), level_std = log(y + 0.1)))

stan_mcmc <- sampling(stannis:::stanmodels$llt_poisson, iter = 500,
  init = stan_inits, data = stan_data, refresh = 0, chains = 1)
```

```
## trying deprecated constructor; please alert package maintainer
```

```
##
## Gradient evaluation took 5.6e-05 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 0.56 seconds.
## Adjust your expectations accordingly!
##
##
##
##  Elapsed Time: 6.7959 seconds (Warm-up)
##                2.14045 seconds (Sampling)
##                8.93635 seconds (Total)
```

```
## The following numerical problems occurred the indicated number of times on chain 1
```

```
##
## Exception thrown at line -1: poisson_log_lpmf: Log rate parameter[100] is -nan, but must not be nan!
```

```
## When a numerical problem occurs, the Hamiltonian proposal gets rejected.
```

```
## See http://mc-stan.org/misc/warnings.html#exception-hamiltonian-proposal-rejected
```

```
## If the number in the 'count' column is small, there is no need to ask about this message on stan-use
```

```
## Warning: There were 19 divergent transitions after warmup. Increasing adapt_delta above 0.8 may help
## http://mc-stan.org/misc/warnings.html#divergent-transitions-after-warmup
```

```
## Warning: Examine the pairs() plot to diagnose sampling problems
```

```r
print(stan_mcmc, pars = "theta")
```

```
## Inference for Stan model: llt_poisson.
## 1 chains, each with iter=500; warmup=250; thin=1;
## post-warmup draws per chain=250, total post-warmup draws=250.
```

```
##
##          mean se_mean   sd 2.5%  25%  50%  75% 97.5% n_eff Rhat
## theta[1] 0.07       0 0.04    0 0.03 0.06 0.09  0.16   110 1.03
## theta[2] 0.01       0 0.00    0 0.00 0.01 0.01  0.02   113 1.01
##
## Samples were drawn using NUTS(diag_e) at Tue Aug  8 17:38:17 2017.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

stannis

With `stannis`, priors for σ_η and $\sigma_x i$ are defined via vectors of length two, where the first value is the mean and second the standard deviation of the Gaussian prior distribution. The output of `stannis` function is currently somewhat limited compared to other packages, as `stannis` is specifically built for studying the potential computational benefits of combining the HMC algorithms of `Stan` and the importance sampling correction approach, without putting too much effort into actual usability.

```
stannis_mcmc <- stannis(y, iter = 500, level = c(0, 1), slope = c(0, 0.1),
  refresh = 0, a1 = c(0, 0), P1 = diag(c(10, 0.1)),
  stan_inits = list(list(theta = c(0.01, 0.01))))
```

```
## trying deprecated constructor; please alert package maintainer
```

```
##
## Gradient evaluation took 0.004404 seconds
## 1000 transitions using 10 leapfrog steps per transition would take 44.04 seconds.
## Adjust your expectations accordingly!
##
##
##
##  Elapsed Time: 7.71973 seconds (Warm-up)
##                7.37972 seconds (Sampling)
##                15.0994 seconds (Total)
```

```
## The following numerical problems occurred the indicated number of times on chain 1
```

```
##
## Exception thrown at line -1: Exception thrown at line -1: Maximum number of iterations for approxima
```

```
## When a numerical problem occurs, the Hamiltonian proposal gets rejected.
```

```
## See http://mc-stan.org/misc/warnings.html#exception-hamiltonian-proposal-rejected
```

```
## If the number in the 'count' column is small, there is no need to ask about this message on stan-use
```

```
stannis_mcmc$mean_theta
```

```
##             [,1]
## [1,] 0.077557707
## [2,] 0.007196784
```

```
stannis_mcmc$ess_theta
```

```
## [1] 191.4950 158.2623
```

```
stannis_mcmc$stan_time
```

```
## [1] 15.09945
```

```
stannis_mcmc$correction_time
```

```
## elapsed
##   0.578
```

Computational efficiency experiment

We will use the same local linear trend model and the data as in previous section. As prior distributions we used $\mu_1 \sim N(0, 10^2)$, $\nu_1 \sim N(0, 0.1^2)$, $\sigma_\eta \sim N(0, 1^2)$, and $\sigma_\xi \sim N(0, 0.1^2)$, with latter two truncated to positive real axis. We compare following algorithms:

- HMC: HMC using NUTS algorithm (from `Stan`)
- PM: Pseudo-marginal MCMC (`bssm`)
- DA: Delayed acceptance PM (`bssm`)
- IS-RWM: IS type correction of MCMC with RWM (`bssm`)
- IS-HMC: IS type correction of MCMC with HMC (`stannis`)

For particle filter approaches (all but HMC), we used ψ-PF with 10 particles.

As a measure of efficiency, we use inverse relative efficiency (IRE), defined as

$$IRE = 100\frac{\bar{T}}{N}\sum_{i=1}^{N}(\hat{\theta}_i - \theta)^2,$$

where $\bar{T}$ is the average running time, $\hat{\theta}_i$ is the estimate from ith run, and θ is the estimate based on 1,000,000 iterations of pseudo-marginal MCMC.

We ran all algorithms $N = 500$ times with $20,000 + 20,000$ iterations, where first half was discarded as a burnin. For standard HMC, default parameters governing the sampler's behaviour caused severe amount of divergent transitions. By changing the `adapt_delta` parameter (corresponding to the target average proposal acceptance probability in the adaptation phase) from the default 0.8 to 0.99, and the maximum treedepth (argument `max_treedepth`) from 10 to 15, we were able to reduce the number of divergent transitions, but there were still on avarage 78 divergent transitions per 20,000 iterations (sd 91).

```
library("dplyr")

results_stan <- readRDS("stan_llt_iter4e4.rda")
results_stannis <- readRDS("stannis_llt_iter4e4.rda")
results_is <- readRDS("is_llt_iter4e4.rda")
results_da <- readRDS("da_llt_iter4e4.rda")
results_pm <- readRDS("pm_llt_iter4e4.rda")

results <-rbind(results_stan, results_stannis, results_is, results_da, results_pm)
grouped <- group_by(results, method)
IRE <- function(x, time, variable) {
  mean((x - reference[variable])^2) * mean(time)
}
reference <- readRDS("reference_llt.rda")
```

The following table summarises the results for σ_{eta}:

```
sumr <- summarise(grouped, mean = mean(theta_1),
  SE = sd(theta_1),
  IRE = 100 * IRE(theta_1, time, "theta_1"),
  "time (s)" = mean(time)) %>% arrange(IRE)
print.data.frame(sumr, digits = 2)
```

```
##    method  mean      SE    IRE time (s)
## 1     isc 0.073 0.00157 0.0042       17
## 2      da 0.073 0.00160 0.0061       24
## 3      pm 0.073 0.00157 0.0095       39
## 4 stannis 0.073 0.00057 0.0313      927
## 5    Stan 0.073 0.00064 0.0486     1166
```

We see that given the equal amount of MCMC iterations, the standard deviation of $E(\sigma_\eta|y)$ over 100 replications with `Stan` is over two times smaller than the Metropolis-based algorithms. However, the computation time of `Stan` is also almost two orders of magnitude higher, and thus in terms of IRE, `Stan` performs poorly compared to standard pseudo-marginal MCMC algorithm of `bssm`. As expected, DA approach provides improvements over PM by decreasing the computation time with only negligible increase in SE. Here the IS approach gives small additional boost compared to DA. The hybrid method combining the IS-correction with HMC slightly decreases the computation time and the standard errors obtained with standard HMC approach. Still, the resulting IRE is clearly higher than with random-walk based algorithms.

For state variables we see similar differences between the algorithms:

```
sumr <- summarise(grouped, mean = mean(level_1),
  SE = sd(level_1),
  IRE = 100 * IRE(level_1, time, "level_1"),
  "time (s)" = mean(time)) %>% arrange(IRE)
print.data.frame(sumr, digits = 2)
```

```
##    method  mean     SE   IRE time (s)
## 1     isc 0.021 0.0074 0.096       17
## 2      da 0.022 0.0076 0.140       24
## 3      pm 0.022 0.0074 0.218       39
## 4 stannis 0.022 0.0027 0.755      927
## 5    Stan 0.022 0.0028 1.023     1166
```

```
sumr <- summarise(grouped, mean = mean(slope_1),
  SE = sd(slope_1),
  IRE = 100 * IRE(slope_1, time, "slope_1"),
  "time (s)" = mean(time)) %>% arrange(IRE)
print.data.frame(sumr, digits = 2)
```

```
##    method  mean      SE     IRE time (s)
## 1     isc 0.011 0.00068 0.00079       17
## 2      da 0.011 0.00070 0.00121       24
## 3      pm 0.011 0.00063 0.00156       39
## 4 stannis 0.011 0.00027 0.00737      927
## 5    Stan 0.011 0.00037 0.01712     1166
```

```
sumr <- summarise(grouped, mean = mean(level_n),
  SE = sd(level_n),
  IRE = 100 * IRE(level_n, time, "level_n"),
  "time (s)" = mean(time)) %>% arrange(IRE)
print.data.frame(sumr, digits = 2)
```

```
##    method mean     SE   IRE time (s)
## 1     isc  2.3 0.0045 0.034       17
## 2      da  2.3 0.0046 0.052       24
## 3      pm  2.3 0.0047 0.087       39
## 4 stannis  2.3 0.0015 0.218      927
## 5    Stan  2.3 0.0018 0.366     1166
```

```
sumr <- summarise(grouped, mean = mean(slope_n),
  SE = sd(slope_n),
  IRE = 100 * IRE(slope_n, time, "slope_n"),
  "time (s)" = mean(time)) %>% arrange(IRE)
print.data.frame(sumr, digits = 2)
```

```
##    method mean      SE    IRE time (s)
## 1     isc 0.06 0.00078 0.0010       17
## 2      da 0.06 0.00079 0.0015       24
## 3      pm 0.06 0.00075 0.0022       39
## 4 stannis 0.06 0.00033 0.0099      927
## 5    Stan 0.06 0.00042 0.0210     1166
```

Discussion

In this vignette we compared computational aspects of different MCMC algorithms for Poisson state space model using a simple simulation experiment. We also experimented with new hybrid method combining the IS correction with HMC algorithm. Although theoretically valid, the benefits of this new method are not clear. Although it improved the basic approach with `Stan`, it was still found to be less efficient than the random-walk based approaches. In hindsight this is not suprising, as repeated runs of Kalman filter and smoother used in the Laplace approximation algorithm can be problematic for the automatic differention used in `Stan`. It should also be noted that the performance of `Stan` varied heavily between different state space models we initially tested, and some amount of tuning of the NUTS algorithm was needed in all cases. Also for some model parameterizations we were not able to get rid of divergence problems in NUTS, but it might be possible to obtain better performance from `Stan` using other HMC algoritms, or by defining the behaviour of the warmup phase more carefully.

Appendix A: Stan model for Poisson local linear trend model

```
data {
  int<lower=0> n;              // number of data points
  int<lower=0> y[n];           // time series
  vector[2] a1;                // prior mean for the initial state
  matrix[2, 2] P1;             // prior covariance for the initial state
  vector[2] sd_prior_means;    // prior means for the sd parameters
  vector[2] sd_prior_sds;      // prior sds for the sd parameters
}

parameters {
  real<lower=0> theta[2];      // sd parameters for level and slope
  // instead of working directly with true states level and slope
  // it is often suggested use standard normal variables in sampling
  // and reconstruct the true parameters in transformed parameters block
  // this should make sampling more efficient although coding the model
  // is less intuitive...
  vector[n] level_std;         // N(0, 1) level noise
  vector[n] slope_std;         // N(0, 1) slope noise
}

transformed parameters {
  vector[n] level;
  vector[n] slope;
  // construct the actual states
  // note that although P1 was allowed to have general form here
  // it is assumed that it is diagonal... laziness (and covers typical cases)
  level[1] = a1[1] + sqrt(P1[1,1]) * level_std[1];
  slope[1] = a1[2] + sqrt(P1[2,2]) * slope_std[1];
  for(t in 2:n) {
    level[t] = level[t-1] + slope[t-1] + theta[1] * level_std[t];
    slope[t] = slope[t-1] + theta[2] * slope_std[t];
  }
}

model {
  // priors for theta
  theta ~ normal(sd_prior_means, sd_prior_sds);
  // standardised noise terms
  level_std ~ normal(0, 1);
  slope_std ~ normal(0, 1);
  // Poisson likelihood
  y ~ poisson_log(level);
}
```

Appendix B: Codes for simulation experiment

```
library("devtools")
install_github("helske/bssm")
install_github("helske/stannis")
library("bssm")
library("coda")
library("rstan")
library("diagis")
library("stannis")
library("doParallel")
library("foreach")

ire_experiment_llt <- function(n_iter,
  nsim_states = 10, seed = sample(.Machine$integer.max, 1), method){

  # simulate the data (with fixed seed)
  set.seed(123)
  n <- 100

  slope <- cumsum(c(0, rnorm(n - 1, sd = 0.01)))
  level <- cumsum(slope + c(0, rnorm(n - 1, sd = 0.01)))
  y <- rpois(n, exp(level))


  results <- data.frame(method = method,
    time = 0, "theta_1" = 0, "theta_2" = 0,
    "level_1" = 0, "slope_1" = 0,
    "level_n" = 0, "slope_n" = 0,
    "divergent" = 0, "treedepth" = 0)
```

```r
set.seed(seed)

if(method == "stannis") {

  res <- stannis(y, iter = n_iter,
    level = c(0, 1), slope = c(0, 0.1), refresh = 0, a1 = c(0, 0), P1 = diag(c(10, 0.1)),
    stan_inits = list(list(theta = c(0.01, 0.01))))

  results[1, 2] <- res$stan_time + res$correction_time
  results[1, 3:4] <- res$mean_theta
  results[1, 5:6] <- weighted_mean(t(res$states[1,,]), res$weights)
  results[1, 7:8] <- weighted_mean(t(res$states[n,,]), res$weights)

  return(results)
}
if(method == "Stan") {
  stan_data <- list(n = n, y = y, a1 = c(0, 0), P1 = diag(c(10, 0.1)),
    sd_prior_means = rep(0, 2), sd_prior_sds = c(1, 0.1))
  stan_inits <- list(list(theta = c(0.01, 0.01),
    slope_std = rep(0, n), level_std = log(y + 0.1)))

  res <- sampling(stannis:::stanmodels$llt_poisson,
    control = list(adapt_delta = 0.99, max_treedepth = 15),
    data = stan_data, refresh = 0,
    iter = n_iter, chains = 1, cores = 1, init = stan_inits)
  results[1, 2] <- sum(get_elapsed_time(res))
  results[1, 3:4] <- summary(res, pars = "theta")$summary[, "mean"]
  results[1, 5] <- summary(res, pars = "level[1]")$summary[, "mean"]
  results[1, 6] <- summary(res, pars = "slope[1]")$summary[, "mean"]
  results[1, 7] <- summary(res, pars = paste0("level[",n,"]"))$summary[, "mean"]
  results[1, 8] <- summary(res, pars = paste0("slope[",n,"]"))$summary[, "mean"]
  diags <- get_sampler_params(res, inc_warmup = FALSE)[[1]]
  results[1, 9] <- sum(diags[, "divergent__"])
  results[1, 10] <- sum(diags[, "treedepth__"] >= 15)

  return(results)
}

model <- ng_bsm(y, P1 = diag(c(10, 0.1)),
  sd_level = halfnormal(0.01, 1),
  sd_slope = halfnormal(0.01, 0.1), distribution = "poisson")
if(method == "isc") {
  res <- run_mcmc(model, n_iter = n_iter, nsim_states = 10,
    method = "isc", n_threads = 1)
  results[1, 2] <- res$time[3]
  results[1, 3:4] <- weighted_mean(res$theta, res$weights * res$counts)
  results[1, 5:6] <- weighted_mean(t(res$alpha[1,,]), res$weights * res$counts)
  results[1, 7:8] <- weighted_mean(t(res$alpha[n,,]), res$weights * res$counts)

  return(results)
}
if(method == "da") {
  res <- run_mcmc(model, n_iter = n_iter, nsim_states = 10, method = "pm")
```

```
    results[1, 2] <- res$time[3]
    results[1, 3:4] <- weighted_mean(res$theta, res$counts)
    results[1, 5:6] <- weighted_mean(t(res$alpha[1,,]), res$counts)
    results[1, 7:8] <- weighted_mean(t(res$alpha[n,,]), res$counts)

    return(results)
  }

  res <- run_mcmc(model, n_iter = n_iter, nsim_states = 10, method = "pm",
    delayed_acceptance = FALSE)
  results[1, 2] <- res$time[3]
  results[1, 3:4] <- weighted_mean(res$theta, res$counts)
  results[1, 5:6] <- weighted_mean(t(res$alpha[1,,]), res$counts)
  results[1, 7:8] <- weighted_mean(t(res$alpha[n,,]), res$counts)
  results
}


cl<-makeCluster(16)
registerDoParallel(cl)

results <-
  foreach (i = 1:500, .combine=rbind, .packages = c("bssm", "diagis", "stannis", "rstan")) %dopar%
  ire_experiment_llt(n_iter = 4e4, seed = i, method = "Stan")
saveRDS(results, file = "stan_llt_iter4e4.rda")


results <-
  foreach (i = 1:500, .combine=rbind, .packages = c("bssm", "diagis", "stannis", "rstan")) %dopar%
  ire_experiment_llt(n_iter = 4e4, seed = i, method = "stannis")
saveRDS(results, file = "stannis_llt_iter4e4.rda")


results <-
  foreach (i = 1:500, .combine=rbind, .packages = c("bssm", "diagis", "stannis", "rstan")) %dopar%
  ire_experiment_llt(n_iter = 4e4, seed = i, method = "isc")
saveRDS(results, file = "is_llt_iter4e4.rda")

results <-
  foreach (i = 1:500, .combine=rbind, .packages = c("bssm", "diagis", "stannis", "rstan")) %dopar%
  ire_experiment_llt(n_iter = 4e4, seed = i, method = "da")
saveRDS(results, file = "da_llt_iter4e4.rda")

results <-
  foreach (i = 1:500, .combine=rbind, .packages = c("bssm", "diagis", "stannis", "rstan")) %dopar%
  ire_experiment_llt(n_iter = 4e4, seed = i, method = "pm")
saveRDS(results, file = "pm_llt_iter4e4.rda")



stopCluster(cl)

## reference values
```

```r
set.seed(123)
n <- 100

slope <- cumsum(c(0, rnorm(n - 1, sd = 0.01)))
level <- cumsum(slope + c(0, rnorm(n - 1, sd = 0.01)))
y <- rpois(n, exp(level))

model <- ng_bsm(y, P1 = diag(c(10, 0.1)),
  sd_level = halfnormal(0.01, 1),
  sd_slope = halfnormal(0.01, 0.1), distribution = "poisson")
res <- run_mcmc(model, n_iter = 1.1e6, n_burnin = 1e5, delayed_acceptance = FALSE, nsim = 10)

theta <- weighted_mean(res$theta, res$counts)
level_1 <- weighted_mean(res$alpha[1,1,], res$counts)
level_n <- weighted_mean(res$alpha[n,1,], res$counts)
slope_1 <- weighted_mean(res$alpha[1,2,], res$counts)
slope_n <- weighted_mean(res$alpha[n,2,], res$counts)
reference_llt <- c(theta_1 = theta[1], theta_2 = theta[2],
  level_1 = level_1, level_n = level_n,
  slope_1 = slope_1, slope_n = slope_n)
saveRDS(reference_llt, file = "reference_llt.rda")
```

References

Andrieu, C., A. Doucet, and R. Holenstein. 2010. "Particle Markov Chain Monte Carlo Methods." *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 72 (3). Blackwell Publishing Ltd: 269–342. http://dx.doi.org/10.1111/j.1467-9868.2009.00736.x.

Banterle, M., C. Grazian, A. Lee, and C. P. Robert. 2015. "Accelerating Metropolis-Hastings algorithms by Delayed Acceptance." *ArXiv E-Prints*, March. http://arxiv.org/abs/1503.00996.

Christen, J. Andrés, and Colin Fox. 2005. "Markov Chain Monte Carlo Using an Approximation." *Journal of Computational and Graphical Statistics* 14 (4): 795–810. doi:10.1198/106186005X76983.

Durbin, J., and S. J. Koopman. 2002. "A Simple and Efficient Simulation Smoother for State Space Time Series Analysis." *Biometrika* 89: 603–15.

Eddelbuettel, Dirk, and Romain François. 2011. "Rcpp: Seamless R and C++ Integration." *Journal of Statistical Software* 40 (8): 1–18. http://www.jstatsoft.org/v40/i08/.

Eddelbuettel, Dirk, and Conrad Sanderson. 2014. "RcppArmadillo: Accelerating R with High-Performance C++ Linear Algebra." *Computational Statistics and Data Analysis* 71 (march): 1054–63. http://dx.doi.org/10.1016/j.csda.2013.02.005.

Harvey, A. C. 1989. *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press.

Helske, Jouni, and Matti Vihola. 2017. *bssm: Bayesian Inference of State Space Models*. http://github.com/helske/bssm.

Hoffman, M. D., and A. Gelman. 2014. "The No-U-Turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo." *Journal of Machine Learning Research* 15: 1593–1623.

Sanderson, Conrad, and Ryan Curtin. 2016. "Armadillo: A Template-Based C++ Library for Linear Algebra."

The Journal of Open Source Software 1 (2). The Open Journal. doi:10.21105/joss.00026.

Stan Development Team. 2016a. *RStan: The R Interface to Stan*. http://mc-stan.org/.

———. 2016b. *The Stan C++ Library*. http://mc-stan.org/.

Team, R Core. 2017. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. http://www.R-project.org/.

Vihola, M., J. Helske, and J. Franks. 2016. "Importance sampling type correction of Markov chain Monte Carlo and exact approximations." *ArXiv E-Prints*, December.

Vihola, Matti. 2012. "Robust Adaptive Metropolis Algorithm with Coerced Acceptance Rate." *Statistics and Computing* 22 (5): 997–1008. doi:10.1007/s11222-011-9269-5.