bssm: Bayesian Inference of Non-linear and Non-Gaussian State Space Models in R

Jouni Helske[*]
Matti Vihola[†]

January 31, 2017

Introduction

State space models (SSM) are latent variable models which are commonly applied in analysing time series data due to their flexible and general framework (cf. J. Durbin and Koopman 2012). For `R` (R Core Team 2016), there is large number of packages available for state space modelling, especially for the two special cases. First special case is linear-Gaussian SSM (LGSSM) where both the observation and state densities are Gaussian with linear relationships with the states. Another special case is SSM with discrete state space, which are sometimes called hidden Markov models (HMM). We do not consider HMMs in this paper. What is special about these two class of models is that the marginal likelihood function, and the conditional state distributions (conditioned on the observations) of these models are analytically tractable, making inference relatively straightforward. See for example (Petris and Petrone 2011, Tusell (2011), J. Helske (2017), S. Helske and Helske (2017)) for review of some of the `R` packages dealing with these type of models. The `R` package `bssm` is designed for Bayesian inference of general state space models with non-Gaussian and/or non-linear observational and state equations. The package aims to provide easy-to-use and efficient functions for fully Bayesian inference of common time series models such basic structural time series model (BSM) (Harvey 1989) with exogenous covariates, simple stochastic volatility models, and discretized diffusion models, making it straighforward and efficient to make predictions and other inference in a Bayesian setting.

When extending the state space modelling to non-linear or non-Gaussian models, some difficulties arise. As the model densities are no longer analytically tractable, computing the latent state distributions, as well as hyperparameter estimation of the model becomes more difficult. One general option is to use Markov chain Monte Carlo (MCMC) methods targeting the full joint posterior of hyperparameters and the latent states, for example by Gibbs sampling or Hamiltonian Monte Carlo. Unfortunately the joint posterior can be very high dimensional and due to the strong autocorrelation structures of the state densities, the efficiency of such methods can be relatively poor. Another asymptotically exact approach is based on the pseudo-marginal particle MCMC approach (Andrieu, Doucet, and Holenstein 2010), where the likelihood function and the state distributions are estimated using sequential Monte Carlo (SMC) i.e. particle filter (PF). However, methods can also be computationally demanding, and optimal tuning of such algorithms can be cumbersome. Yet another option is to resort to approximative methods such extended and unscented Kalman filters, as well as more general Laplace approximation provided for example by (Lindgren and Rue 2015).

The motivation behind the `bssm` package is in (Vihola, Helske, and Franks 2017) which suggests a new computationally efficient, parallelisable approach for Bayesian inference of state space models. The core idea is to use fast approximative MCMC targeting the approximate marginal posterior of the hyperparameters, which is then used in importance sampling type weighting phase which provides asymptotically exact samples from the joint posterior of hyperparameters and the hidden states. In addition to this the two-stage procedure, standard pseudo-marginal MCMC and so called delayed acceptance pseudo-marginal MCMC are also supported.

We will first introduce the basic state space modelling framework used in `bssm`, and the relevant algorithms. We then give illustrations how to use `bssm` in practice.

[*]Linköping University, Department of Science and Technology, Sweden, University of Jyväskylä, Department of Mathematics and Statistics, Finland

[†]University of Jyväskylä, Department of Mathematics and Statistics, Finland

State space models with linear-Gaussian dynamics

Denote a sequence of observations $(y_1, \ldots, y_T)$ as $y_{1:T}$, and sequence of latent state variables $(\alpha_1, \ldots, \alpha_T)$ as $\alpha_{1:T}$. Note that in general both the observations and the states can be multivariate, but currently the main algorithms of `bssm` support only univariate observations. A general state space model consists of two parts: observation level densities $g_t(y_t|\alpha_t)$ and latent state transition densities $\mu_t(\alpha_{t+1}|\alpha_t)$. We first focus on the case where the state transitions are linear-Gaussian:

$$\alpha_{t+1} = c_t + T_t \alpha_t + R_t \eta_t,$$

where c_t is known input vector (often omitted), and T_t and R_t are a system matrices which can depend on unknown parameters. Also, $\eta_t \sim N(0, I_k)$ and $\alpha_1 \sim N(a_1, P_1)$ independently of each other. For observation level density g_t, the `bssm` package currently supports basic stochastic volatility model and general exponential family state space models.

For exponential family models, the observation equation has a general form

$$g_t(y_t|d_t + Z_t\alpha_t, x'_t\beta, \phi, u_t),$$

where d_t is a again known input, x_t contains the exogenous covariate values at time t, with β corresponding to the regression coefficients. Parameter ϕ and the known vector u_t are distribution specific and can be omitted in some cases. Currently, following observational level distributions are supported:

- Gaussian distribution: $g_t(y_t|Z_t\alpha_t, x'_t\beta) = x'_t\beta + Z_t\alpha_t + H_t\epsilon_t$ with $\epsilon_t \sim N(0, 1)$.
- Poisson distribution: $g_t(y_t|Z_t\alpha_t, x'_t\beta, u_t) = \text{Poisson}(u_t \exp(x'_t\beta + Z_t\alpha_t))$, where u_t is the known exposure at time t.
- Binomial distribution: $g_t(y_t|Z_t\alpha_t, x'_t\beta, u_t) = \text{binomial}(u_t, \exp(x'_t\beta + Z_t\alpha_t)/(1 + \exp(x'_t\beta + Z_t\alpha_t)))$, where u_t is the size and $\exp(x_t\beta + Z_t\alpha_t)/(1 + \exp(x'_t\beta + Z_t\alpha_t))$ is the probability of the success.
- Negative binomial distribution: $g_t(y_t|Z_t\alpha_t, x'_t\beta, \phi, u_t) = \text{negative binomial}(\exp(x'_t\beta + Z_t\alpha_t), \phi, u_t)$, where $u_t \exp(x'_t\beta + Z_t\alpha_t)$ is the expected value and ϕ is the dispersion parameter (u_t is again exposure term).

For stochastic volatility model, there are two possible parameterizations available. In general for we have

$$y_t = x'_t\beta + \sigma \exp(\alpha_t/2)\epsilon_t, \quad \epsilon_t \sim N(0, 1),$$

and

$$\alpha_{t+1} = \mu + \rho(\alpha_t - \mu) + \sigma_\eta \eta_t,$$

with $\alpha_1 \sim N(\mu, \sigma^2_\eta/(1 - \rho^2))$. For identifiability purposes we must either choose $\sigma = 1$ or $\mu = 0$. Although analytically identical, the parameterization with μ is often preferable in terms of computational efficiency.

Typically some of the model components such as β, T_t or R_t depend on unknown parameter vector θ, so $g_t(y_t|\alpha_t)$ and $\mu_t(\alpha_{t+1}|\alpha_t)$ depend implicitly on θ. Our goal is to perform Bayesian inference of the joint posterior of $\alpha_{1:T}$ and θ.

MCMC for Gaussian state space models

Given the prior $p(\theta)$, the joint posterior of θ and $\alpha_{1:T}$ is given as

$$p(\alpha_{1:T}, \theta|y_{1:T}) \propto p(\theta)p(\alpha_{1:T}, y_{1:T}|\theta) = p(\theta)p(y|\theta)p(\alpha_{1:T}|y_{1:T}, \theta)$$

where $p(y_{1:T}|\theta)$ is the marginal likelihood, and $p(\alpha_{1:T}|y_{1:T}, \theta)$ is often referred as a smoothing distribution. However, instead of targeting this joint posterior, it is typically more efficient to target the marginal posterior

$p(\theta|y)$, and then given the sample $\{\theta^i\}_{i=1}^n$ from this marginal posterior, simulate states $\alpha^i_{1:T}$ from the smoothing distribution $p(\alpha_{1:T}|y_{1:T}, \theta^i)$ for $i = 1 \ldots, n$.

For Gaussian models given the parameters θ, the marginal likelihood $p(y_{1:T}|\theta)$ can be computed using the well known Kalman filter recursions, and there are several algorithms for simulating the states $\alpha_{1:T}$ from the smoothing distribution $p(\alpha_{1:T}|y_{1:T})$ (see for example J. Durbin and Koopman (2012)). Therefore we can straightforwardly apply standard MCMC algoritms. In `bssm`, we use an adaptive random walk Metropolis algorithm based on RAM (Vihola 2012) where we fix the target acceptance rate beforehand. There RAM algorithm is provided by the `ramcmc` package (J. Helske 2016). The complete adaptive MCMC algorithm of `bssm` for Gaussian models is as follows.

Given the target acceptance rate a^* (e.g. 0.234) and $\gamma \in (0, 1]$ (the default 2/3 works well in practice), at iteration i:

1. Compute the proposal $\theta' = \theta^{i-1} + S_{i-1}u^i$, where u_i is simulated from the standard d-dimensional Gaussian distribution and S_{i-1} is a lower diagonal matrix with positive diagonal elements.
2. Accept the proposal with probability $a^i := \min\{1, \frac{p(\theta')p(y_{1:T}|\theta')}{p(\theta^{i-1})p(y_{1:T}|\theta^{i-1})\}}$.
3. If the proposal θ' is accepted, set $\theta^i = \theta'$ and simulate a realization (or multiple realizations) of the states $\alpha_{1:T}$ from $p(\alpha_{1:T}|y_{1:T}, \theta^i)$ using the simulation smoothing algorithm by J. Durbin and Koopman (2002). Otherwise, set $\theta^i = \theta^{i-1}$ and $\alpha^i_{1:T} = \alpha^{i-1}_{1:T}$.
4. Compute (using Cholesky update or downdate algorithm) the Cholesky factor matrix S^i satisfying the equation
$$S_i S_i^T = S_{i-1}\left(I + \min\{1, di^{-\gamma}\}(a^i - a^*)\frac{u_i u_i^T}{\|u_i\|^2}\right)S_{i-1}^T.$$

If the interest is in the posterior means and variances of the states, we can replace the simulation smoothing in step 3 with standard fixed interval smoothing which gives the smoothed estimates (expected values and variances) of the states given the data and the model parameters. From these, the posterior means and variances of the states can be computed straightforwardly.

Non-Gaussian models

For non-linear/non-Gaussian models, the marginal likelihood $p(y_{1:T}|\theta)$ is typically not available in closed form. Thus we need to resort to simulation methods, which leads to pseudo-marginal MCMC algorithm (Lin, Liu, and Sloan 2000, Beaumont (2003), Andrieu and Roberts (2009)). The observational densities of our non-linear/non-Gaussian models are all twice differentiable, so we can straightforwardly use the Laplace approximation based on (J. Durbin and Koopman 2000). This gives us an approximating Gaussian model which has the same mode of $p(\alpha_{1:T}|y_{1:T}, \theta)$ as the original model. Often this approximating Gaussian model works well as such, and thus we can use it in MCMC scheme directly, which results in an approximate Bayesian inference. We can also use the approximating model together with importance sampling or particle filtering, which produces exact Bayesian inference on $p(\alpha_{1:T}, \theta|y_{1:T})$.

We will illustrate our approach using simple importance sampling. We can factor the likelihood of the non-Gaussian model as (J. Durbin and Koopman 2012)

$$\begin{aligned} p(y_{1:T}|\theta) &= \int g(\alpha_{1:T}, y_{1:T}|\theta)\mathrm{d}\alpha \\ &= g(y_{1:T}|\theta)E_g\left[\frac{g(y_{1:T}|\alpha_{1:T}, \theta)}{\tilde{g}(y_{1:T}|\alpha_{1:T}, \theta)}\right], \end{aligned}$$

where $\tilde{g}(y_{1:T}|\theta)$ is the likelihood of the Gaussian approximating model and the expectation is taken with respect to the Gaussian density $g(\alpha|y,\theta)$. Equivalently we can write

$$\begin{aligned}
\log p(y_{1:T}|\theta) &= \log g(y_{1:T}|\theta) + \log E_g\left[\frac{g(y_{1:T}|\alpha_{1:T},\theta)}{\tilde{g}(y_{1:T}|\alpha_{1:T},\theta)}\right] \\
&= \log g(y_{1:T}|\theta) + \log \frac{g(y_{1:T}|\hat{\alpha}_{1:T},\theta)}{\tilde{g}(y_{1:T}|\hat{\alpha}_{1:T},\theta)} + \log E_g\left[\frac{g(y_{1:T}|\alpha,\theta)/g(y_{1:T}|\hat{a}lpha_{1:T},\theta)}{\tilde{g}(y_{1:T}|\alpha_{1:T},\theta)/\tilde{g}(y_{1:T}|\hat{\alpha}_{1:T},\theta)}\right] \\
&= \log g(y|\theta) + \log \hat{w} + \log E_g w^* \\
&\approx \log g(y|\theta) + \log \hat{w} + \log \frac{1}{N}\sum_{j=1}^{N} w_j^*,
\end{aligned}$$

where $\hat{\alpha}_{1:T}$ is the conditional mode estimate obtained from the approximating Gaussian model. For approximating inference, we simply omit the term $\log \frac{1}{N}\sum_{j=1}^{N} w_j^*$.

In principle, when using the exact Bayesian inference we should simulate multiple realizations of the states $\alpha_{1:T}$ in each iteration of MCMC in order to compute $\log \frac{1}{N}\sum_{j=1}^{N} w_j^*$. Fortunately, we can use so called delayed acceptance (DA) approach (Christen and Fox 2005; Banterle et al. 2015) which speeds up the computation considerably. Instead of single acceptance step we use two-stage approach as follows.

1. Make initial acceptance of the given proposal θ' with probability $\min\left\{1, \frac{p(y_{1:T}|\theta')\hat{w}'}{p(y_{1:T}|\theta^{i-1})\hat{w}^{i-1}}\right\}$.
2. If accepted, perform the importance sampling of the states $\alpha_{1:T}$ and make the delayed acceptance with probability $\min\{1, \sum_{j=1}^{N} w_j^{*,'} / \sum_{j=1}^{N} w_j^{*,i-1}\}$.
3. If the delayed acceptance is successful, set $\theta^i = \theta'$ and sample one (or multiple) realization of the previously simulated states with weights $w_j^i, j = 1,\ldots,N$ (with replacement in case of multiple samples are stored). Otherwise, set $\theta^i = \theta^{i-1}$ and similarly for the states.

If our approximation is good, then most of the times when we accept in the first stage we also accept in seconds stage, and thus we often need to simulate the states only for each accepted state. Compared to standard pseudo-marginal approach where we need to simulate the states for each proposal, DA can provide substantial computational benefits.

However, the simple importance approach does not scale well with the data, leading to large variance in importance weights. Thus it is more efficient to use particle filtering based simulation methods for the marginal likelihood estimation and state simulation. Although `bssm` supports standard bootstrap particle filter (Gordon, Salmond, and Smith 1993), we recommend using more efficient ψ-auxiliary particle filter (Vihola, Helske, and Franks 2017) which makes use of our approximating Gaussian model. With ψ-APF, we typically need only a very few particles (say 10) for relatively accurate likelihood estimate, which again speeds up the computations.

In addition to standard pseudo-marginal MCMC or its DA variant, `bssm` also supports the importance sampling type correction method presented in Vihola, Helske, and Franks (2017). Here the MCMC algorithm targets the approximate marginal posterior of θ, and the correction to actual target joint posterior is made in offline fashion using SMC. Essentially it has all the same ingredients as DA algorithm described above, but by splitting work into to separate tasks, we get additional computational benefits over DA as we need to run the particle filter only for each accepted value of the Markov chain (after burnin), and the weight computations are straightforwardly parallelisable. fOr effciency comparisons between IS-weighting and DA, see Vihola, Helske, and Franks (2017) and Franks and Vihola (2017).

For all MCMC algorithms, `bssm` uses so-called jump chain representation of the Markov chain $X_1,\ldots,X_n$, where we only store each accepted X_k and the number of steps we stayed on the same state. So for example if $X_{1:n} = (1,2,2,1,1,1)$, we present such chain as $\tilde{X} = (1,2,1)$, $N = (1,2,3)$. This approach reduces the storage space, and makes it more efficient to use importance sampling type correction algorithms. One drawback of this approach is that the results from the MCMC runs correspond to weighted samples from the target posterior, so some of the commonly used postprocessing tools need to be adjusted. Of course in case of other methods than IS-weighting, the simplest option is to just expand the samples using the stored counts N instead.

Non-linear model state densities

In case the state equation is non-linear, the standard particle MCMC approach using bootstrap filter can still be used. For delayed acceptance and IS-weighting approaches, as well as ψ-APF, new approximation techniques are needed. For this tasks, the `bssm` supports approximations by extended Kalman filter (EKF). Same approach is also applicable to the case with non-linear Gaussian observation densities. Due to the more general form of these models, definition of these models with `bssm` is slightly more complex. The general non-linear Gaussian model in the `bssm` has following form:

$$y_t = Z(t, \alpha_t, \theta) + H(t, \alpha_t, \theta)\epsilon_t, \alpha_{t+1} = T(t, \alpha_t, \theta) + R(t, \alpha_t, \theta)\eta_t, \alpha_1 \sim N(a_1(\theta), P_1(\theta)),$$

with $t = 1, \ldots, n$, ϵ_t $N(0, \mathrm{I}_p)$, and η $N(0, \mathrm{I}_k)$. Here vector θ contains the unknown model parameters. Functions $T(\cdot)$, $H(\cdot)$, $T(\cdot)$, $R(\cdot)$,$a_1(\cdot)$, $P_1(\cdot)$, as well as functions defining the Jacobians of $Z(\cdot)$ and $T(\cdot)$ needed by the EKF and the prior distribution for θ must be defined by user as a external pointers to `C++` functions. All of these functions can also depend on some known parameters, defined as `known_params` (vector) and `known_tv_params` (matrix with n columns) arguments to `nlg_ssm` function. Note that while using the Laplace approximation as intermediate step is typically always more efficient that standard BSF based particle MCMC, the EKF approximation can be very unstable, and thus the the methods using EKF in the approximation phase can have poor performance in some cases.

Time-discretised diffusion models

The `bssm` package also supports models where the state equation is defined as a continuous time diffusion model of form

$$\mathrm{d}\alpha_t = \mu(t, \alpha_t, \theta)\mathrm{d}t + \sigma(t, \alpha_t, \theta)\mathrm{d}B_t, \quad t \geq 0,$$

where B_t is a (vector valued) Brownian motion and where μ and σ are vector and matrix valued functions, with the univariate observation density $p(y_k|\alpha_k)$ defined at integer times $k = 1\ldots, n$. As these transition densities are generally unavailable for non-linear diffusions, we use Millstein time-discretisation scheme for approximate simulation with bootstrap particle filter. Fine discretisation mesh gives less bias than the coarser one, with increased computational complexity. The DA and IS approaches can be used to speed up the inference by using coarse discretisation in the first stage, and then using more fine mesh in the second stage. For comparison of DA and IS approaches in case of geometric Brownian motion model, see again Vihola, Helske, and Franks (2017). Like non-linear Gaussian models, this model is also defined by small C++ snippets.

Package functionality

Main functions of `bssm` is written in `C++`, with help of `Rcpp` and `RcppArmadillo` packages. On the `R`side, package uses S3 methods in order to provide relatively unified workflow independent of the type of the model one is working with. The model building functions such as `ng_bsm` and `svm` are used to construct the actual state models which can be then passed to other methods, such as `logLik` and `run_mcmc` which compute the log-likelihood value and run MCMC algorithm respectively. We will now briefly describe the main functions and methods of `bssm`, for more detailed descriptions of different function arguments and return values, see the corresponding documentation in `R`.

Model building functions

For linear-Gaussian models, `bssm` offers functions `bsm` for basic univariate structural time series models (BSM), `ar1` for univariate, possibly noisy AR(1) process, as well as general `lgg_ssm` for arbitrary (multivariate) LGSSMs, defined via pointers to user supplied `C++` functions. There is also a function `gssm` for general univariate LGSSM where user defines the system matrices directly as R objects, which can be more convinient in many cases. However, defining the priors for this model is somewhat more restricted, whereas for `lgg_smm`, arbitrary prior definitions can be used. As an example, consider a Gaussian local linear trend model of form

$$
\begin{aligned}
y_t &= \mu_t + \epsilon_t, \\
\mu_{t+1} &= \mu_t + \nu_t + \eta_t, \\
\nu_{t+1} &= \nu_t + \xi_t,
\end{aligned}
$$

with zero-mean Gaussian noise terms $\epsilon_t, \eta_t, \xi_t$ with unknown standard deviations. This model can be built with `bsm` function as

```r
library("bssm")
data("nhtemp", package = "datasets")
prior <- halfnormal(1, 10)
bsm_model <- bsm(y = nhtemp, sd_y = prior, sd_level = prior,
sd_slope = prior)
```

Here we use helper function `halfnormal` which defines half-Normal prior distribution for the standard deviation parameters, with first argument defining the initial value of the parameter, and second defines the scale parameter of the half-Normal distribution. Other prior options are `normal` and `uniform`.

Same model could also by built with `lgg_ssm`, by defining the small `C++` snippets and passing the corresponding external pointers to `lgg_ssm`. For the BSM model, we can use following `cpp` file:

```cpp
// A template for building a general linear-Gaussian state space model
// Here we define an univariate local linear trend model which could be
// constructed also with bsm function.

#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::interfaces(r, cpp)]]

// theta:
// theta(0) = standard deviation sigma_y
// theta(1) = standard deviation sigma_level
// theta(2) = standard deviation sigma_slope
//
// Function for the prior mean of alpha_1
// [[Rcpp::export]]
arma::vec a1_fn(const arma::vec& theta, const arma::vec& known_params) {
  return arma::vec(2, arma::fill::zeros);
}
// Function for the prior variance of alpha_1
// [[Rcpp::export]]
arma::mat P1_fn(const arma::vec& theta, const arma::vec& known_params) {

  arma::mat P1(2, 2, arma::fill::zeros);
  P1(0, 0) = 1000;
  P1(1, 1) = 1000;
  return P1;
}

// Function for the Cholesky of the observational level covariance matrix
// [[Rcpp::export]]
arma::mat H_fn(const unsigned int t, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {
  // note no transformations, needs to check for positivity in prior
  // we could also use exp(theta) here and work with the corresponding prior
  arma::mat H(1,1);
  H(0, 0) = theta(0);
  return H;
}

// Function for the Cholesky of state level covariance matrix
// [[Rcpp::export]]
arma::mat R_fn(const unsigned int t, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {
  arma::mat R(2, 2, arma::fill::zeros);
  R(0, 0) = theta(1);
  R(1, 1) = theta(2);
  return R;
}


// Z function
// [[Rcpp::export]]
arma::mat Z_fn(const unsigned int t, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {
  arma::mat Z(1, 2, arma::fill::zeros);
  Z(0, 0) = 1.0;
  return Z;
}

// T function
// [[Rcpp::export]]
arma::mat T_fn(const unsigned int t, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {
  arma::mat T(2, 2, arma::fill::ones);
  T(1, 0) = 0.0;
  return T;
}

// input to state equation
// [[Rcpp::export]]
arma::vec C_fn(const unsigned int t, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {
  return arma::vec(2, arma::fill::zeros);
}
// input to observation equation
// [[Rcpp::export]]
arma::vec D_fn(const unsigned int t, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {
  return arma::vec(1, arma::fill::zeros);
}

// # log-prior pdf for theta
// [[Rcpp::export]]
double log_prior_pdf(const arma::vec& theta) {
  double log_pdf = -std::numeric_limits<double>::infinity();
  if (arma::all(theta >= 0)) {
   log_pdf = R::dnorm(theta(0), 0, 10, 1) +
     R::dnorm(theta(1), 0, 10, 1) +
     R::dnorm(theta(2), 0, 10, 1);
  }

  return log_pdf;
}


// Create pointers, no need to touch this if
// you don't alter the function names above
// [[Rcpp::export]]
Rcpp::List create_xptrs() {

  // typedef for a pointer returning matrices Z, H, T, and R
  typedef arma::mat (*lmat_fnPtr)(const unsigned int t, const arma::vec& theta,
    const arma::vec& known_params, const arma::mat& known_tv_params);
  // typedef for a pointer of linear function of lgg-model equation returning vectors D and C
  typedef arma::vec (*lvec_fnPtr)(const unsigned int t, const arma::vec& theta,
    const arma::vec& known_params, const arma::mat& known_tv_params);

  // typedef for a pointer returning vector a1
  typedef arma::vec (*a1_fnPtr)(const arma::vec& theta, const arma::vec& known_params);
  // typedef for a pointer returning matrix P1
  typedef arma::mat (*P1_fnPtr)(const arma::vec& theta, const arma::vec& known_params);
  // typedef for a pointer of log-prior function
  typedef double (*prior_fnPtr)(const arma::vec&);

  return Rcpp::List::create(
    Rcpp::Named("a1_fn") = Rcpp::XPtr<a1_fnPtr>(new a1_fnPtr(&a1_fn)),
    Rcpp::Named("P1_fn") = Rcpp::XPtr<P1_fnPtr>(new P1_fnPtr(&P1_fn)),
    Rcpp::Named("Z_fn") = Rcpp::XPtr<lmat_fnPtr>(new lmat_fnPtr(&Z_fn)),
    Rcpp::Named("H_fn") = Rcpp::XPtr<lmat_fnPtr>(new lmat_fnPtr(&H_fn)),
    Rcpp::Named("T_fn") = Rcpp::XPtr<lmat_fnPtr>(new lmat_fnPtr(&T_fn)),
    Rcpp::Named("R_fn") = Rcpp::XPtr<lmat_fnPtr>(new lmat_fnPtr(&R_fn)),
    Rcpp::Named("D_fn") = Rcpp::XPtr<lvec_fnPtr>(new lvec_fnPtr(&D_fn)),
    Rcpp::Named("C_fn") = Rcpp::XPtr<lvec_fnPtr>(new lvec_fnPtr(&C_fn)),
    Rcpp::Named("log_prior_pdf") =
      Rcpp::XPtr<prior_fnPtr>(new prior_fnPtr(&log_prior_pdf)));
}
```

Note that most of this code is general and can be modified to specific models accordingly by changing the function bodies. Word of caution when using these `C++` snippets: Due to the use of pointers, users must recompile and construct the model objects in each `R` session, i.e. saving and later loading the model object is not sufficient. Also, using the `sourceCpp` function from the `Rcpp` package repeatedly for the recompilation of the external pointers from the same file can cause `R` crash unexpectedly, if the corresponding cache directory is not changed between compilations[1]. Changing the cache directory (`cacheDir` argument of `sourceCpp`) between subsequent compilations is therefore recommended.

We can then compile the file and construct the model as

[1] 1

```
Rcpp::sourceCpp("lgg_ssm_template.cpp", rebuild = TRUE, cleanupCacheDir = TRUE)
```

```
## Warning in normalizePath(path.expand(path), winslash, mustWork):
## path[1]="C:/Users/jouhe21/AppData/Local/Temp/RtmpywYOGg/Rbuild24b4648b4be7/
## bssm/vignettes/../inst/include": The system cannot find the file specified
```

```
pntrs <- create_xptrs()
bsm_model2 <- lgg_ssm(y = nhtemp, Z = pntrs$Z, H = pntrs$H, T = pntrs$T, R = pntrs$R,
a1 = pntrs$a1, P1 = pntrs$P1, state_intercept = pntrs$C, obs_intercept = pntrs$D,
log_prior_pdf = pntrs$log_prior_pdf, theta = rep(1, 3),
n_states = 2, state_names = c("level", "slope"), time_varying = rep(FALSE, 6))
```

Compared to `bsm`, we need to manually define the actual number of states, and optionally their names. We can also use argument `time_varying` for defining whether or not some of the system matrices vary in time or not which can lead to some computational savings

And we seem to get identical results as expected:

```
logLik(bsm_model)
```

```
## [1] -127.639
```

```
logLik(bsm_model2)
```

```
## [1] -127.639
```

For non-Gaussian models, function `ng_bsm` can be used for constructing an BSM model where the observations are assumed to be distributed according to Poisson, binomial or negative binomial distribution. The syntax is nearly identical as in case of `bsm`, but we now define also the distribution via argument `distribution`, and depending on the model, we can also define paramters `u` and `phi`. For Poisson and negative binomial models, the known parameter `u` corresponds to the offset term, whereas in case of binomial model `u` defines the number of trials. For negative binomial model, argument `phi` defines the dispersion term, which can be given as a fixed value, or as a prior function. For same observational densities, a model where the state equation follows a first order autoregressive process can be defined using the function `ng_ar1`. Finally, a stochastic volatility model can be defined using a function `svm`, and an arbirary linear-Gaussian state model with Poisson, binomial or negative binomial distributed observations can be defined with `ngssm`.

For models where the state equation is now longer linear-Gaussian, we can again use our pointer-based interface. General non-linear Gaussian model can be defined with the function `nlg_ssm`, with similar fashion as in case of `lgg_ssm`. A template for `nlg_ssm` can be found in the Appendix.

As a relatively new feature, `bssm` now supports also discretely observed diffusion models where the state process is assumed to be continous stochastic process These can be constructed using the `sde_ssm` function, which takes pointers to C++ functions defining the drift, diffusion, the derivative of the diffusion function, and the log-densities of the observations and the prior. As an example, let us consider an Ornstein–Uhlenbeck process

$$\mathrm{d}\alpha_t = \rho(\nu - \alpha_t)\mathrm{d}t + \sigma \mathrm{d}B_t,$$

with parameters $\theta = (\phi, \nu, \sigma) = (0.5, 2, 1)$ and the initial condition $\alpha_0 = 1$. For observation density, we use Poisson distribution with parameter $\exp(\alpha_k)$. We first simulate a trajectory $x_0, \ldots, x_n$ using the `sde.sim` function from the `sde` package (Iacus 2016) and use that for the simulation of observations y:

```
set.seed(1)
suppressMessages(library("sde"))
```

```
## Warning: package 'sde' was built under R version 3.4.4

## Warning: package 'fda' was built under R version 3.4.4

## Warning: package 'zoo' was built under R version 3.4.4
```

```
x <- sde.sim(t0 = 0, T = 100, X0 = 1, N = 100,
  drift = expression(0.5 * (2 - x)),
  sigma = expression(1),
  sigma.x = expression(0))
y <- rpois(100, exp(x[-1]))
```

We then compile and build the model as in the case of `lgg_ssm` model:

```
Rcpp::sourceCpp("sde_ssm_template.cpp", rebuild = TRUE, cleanupCacheDir = TRUE)
```

```
## Warning in normalizePath(path.expand(path), winslash, mustWork):
## path[1]="C:/Users/jouhe21/AppData/Local/Temp/RtmpywYOGg/Rbuild24b4648b4be7/
## bssm/vignettes/../inst/include": The system cannot find the file specified
```

```
pntrs <- create_xptrs()
sde_model <- sde_ssm(y, pntrs$drift, pntrs$diffusion,
  pntrs$ddiffusion, pntrs$obs_density, pntrs$prior, c(0.5, 2, 1), 1, FALSE)
```

Filtering and smoothing

Filtering refers to estimating the conditional densities of the hidden states at time t, given the observations up to that point. For linear-Gaussian models, these densities can be efficiently computed using the Kalman filter recursions. The `bssm` has a method `kfilter` for this task. For models defined with the `ngssm` ,`ng_bsm`, `ng_ar1`, and `svm` functions, `kfilter` will first construct an approximating Gaussian model for which the Kalman filter is then used. For details of this approximation, see James Durbin and Koopman (1997) and Vihola, Helske, and Franks (2017). For non-linear models defined by `nlg_ssm` it is possible to perform filtering using extended Kalman filter (EKF) with the function `ekf`, or unscented Kalman filter with the function `ukf`. It is also possible to use iterated EKF (IEKF) by changing the argument `iekf_iter` of the `ekf` function. Compared to EKF, in IEKF the observation equation is linearized iteratively within each time step.

While Kalman filter solves the filtering problem exactly in case of linear-Gaussian models, EKF, UKF, and the filtering based on the approximating Gaussian models produce only approximate, possibly biased filtering estimates for general models. This problem can be solved by the use of particle filters (PF). These sequential Monte Carlo methods are computationally more expensive, but can in principle deal with almost arbitrary state space models. The `bssm` supports general bootstrap particle filter (BSF) for all model classes of the `bssm`. For `ngssm` ,`ng_bsm`, `ng_ar1`, and `svm` models we recommend the particle filter called ψ-APF (Vihola, Helske, and Franks 2017) which makes use of the previously mentioned approximating Gaussian model in order to produce more efficient filter. It is also available for `nlg_ssm` models but in case of severe non-linearities, it is not necessarily best option.

Compared to filtering problem, in smoothing problems we are interested in the conditional densities of the hidden states at certain time point t given all the observations $y_1, \ldots, y_t, \ldots, y_n$. Again for linear-Gaussian models we can use so called Kalman smoothing recursions, where as in case of more general models we can rely on approximating methods, or smoothing algorithms based on the output of particle filters. Currently only filter-smoother approach (Kitagawa 1996) for particle smoothing is fully supported.

Markov chain Monte Carlo

The main purpose of the `bssm` is to allow efficient MCMC-based inference for various state space models. For this task, a method `run_mcmc` can be used. The function takes large number of arguments, depending on the model class, but for many of these, default values are provided. For linear-Gaussian models, we only need to supply the number of iterations. Here we define a random walk model with a drift and stochastic seasonal component for UK gas consumption dataset and use 40 000 MCMC iteration where first half is discarded by default as a burn-in (burn-in phase is also used for the adaptation of the proposal distribution):

```r
prior <- halfnormal(0.1, 1)
UKgas_model <- bsm(log10(UKgas), sd_y = prior, sd_level = prior,
  sd_slope = prior, sd_seasonal =  prior)

mcmc_bsm <- run_mcmc(UKgas_model, n_iter = 4e4)
mcmc_bsm
```

```
##
## Call:
## run_mcmc.bsm(object = UKgas_model, n_iter = 40000)
##
## Iterations = 20001:40000
## Thinning interval = 1
## Length of the final jump chain = 4820
##
## Acceptance rate after the burn-in period:  0.24095
##
## Summary for theta:
##
##                    Mean           SD           SE
## sd_y        0.016277908 0.0058299179 1.791884e-04
## sd_level    0.005103946 0.0032997156 9.627454e-05
## sd_slope    0.001220829 0.0005275716 1.475450e-05
## sd_seasonal 0.026181135 0.0037166520 1.082403e-04
##
## Effective sample sizes for theta:
##
##                  ESS
## sd_y        1058.535
## sd_level    1174.709
## sd_slope    1278.538
## sd_seasonal 1179.033
##
## Summary for alpha_109:
##
##                    Mean         SD           SE
## level       2.845139098 0.01722130 3.369275e-04
## slope       0.009829496 0.00389792 7.794318e-05
## seasonal_1  0.268115967 0.03542028 6.909860e-04
## seasonal_2  0.061533709 0.01774275 3.647596e-04
## seasonal_3 -0.295414064 0.01517585 3.242937e-04
##
## Effective sample sizes for alpha_109:
##
##                 ESS
## level      2612.514
## slope      2500.976
## seasonal_1 2627.638
## seasonal_2 2366.075
## seasonal_3 2189.925
##
## Run time:
##    user  system elapsed
##    6.45    0.04    6.61
```

Note that all MCMC algorithms of `bssm` output also state forecasts for the timepoint $n + 1$, the summary statistics of this state is also shown in the output above.

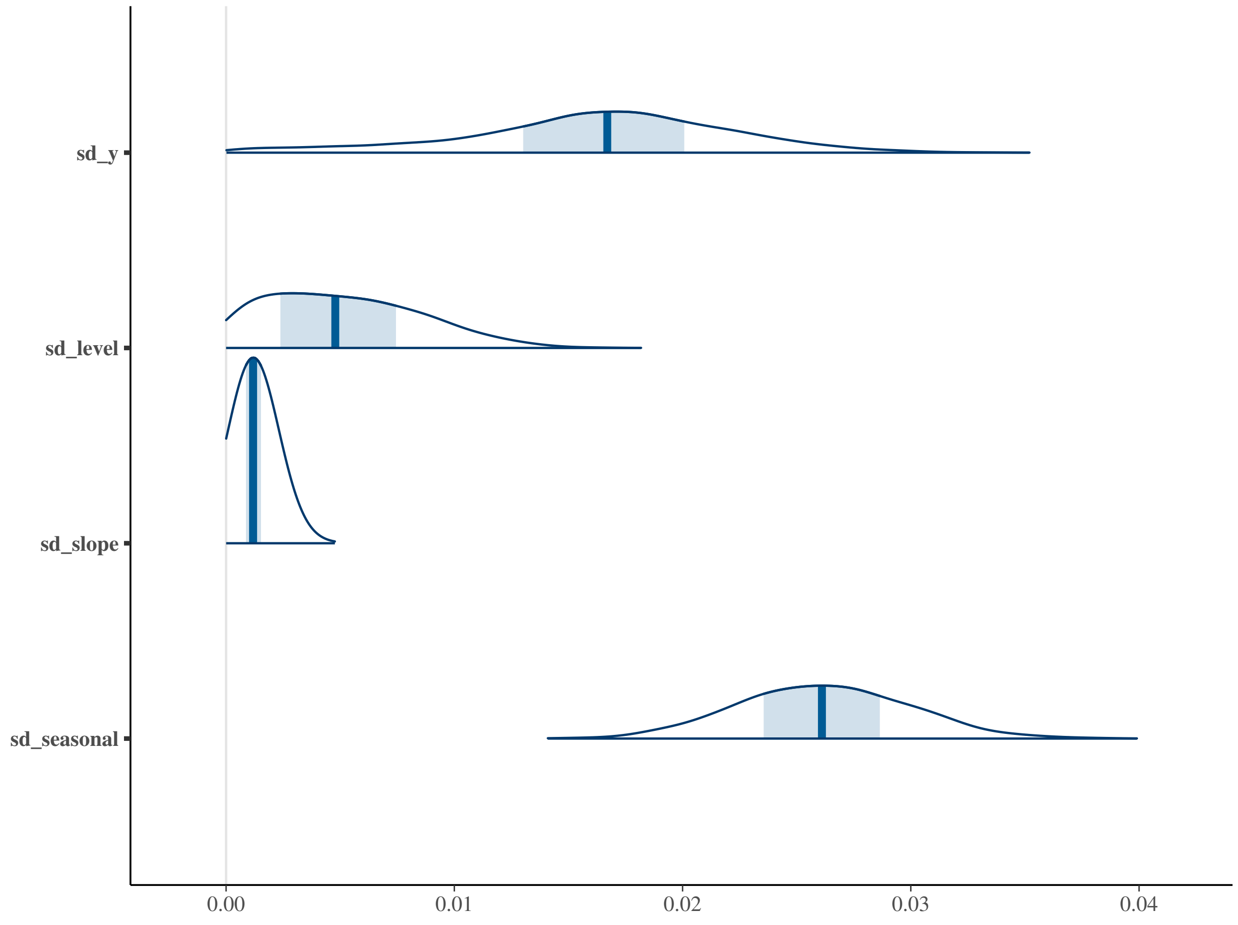
For plotting purposes, we'll use `bayesplot` package for the figures after expanding our jump chain representation. The function `expand_sample` expands the jump chain representation to typical Markov chain, and returns an object of class `mcmc` of the `coda` package (Plummer et al. 2006) (thus the plotting and diagnostic methods of `coda` can also be used).

```
theta <- expand_sample(mcmc_bsm, "theta")
suppressMessages(library("bayesplot"))
```

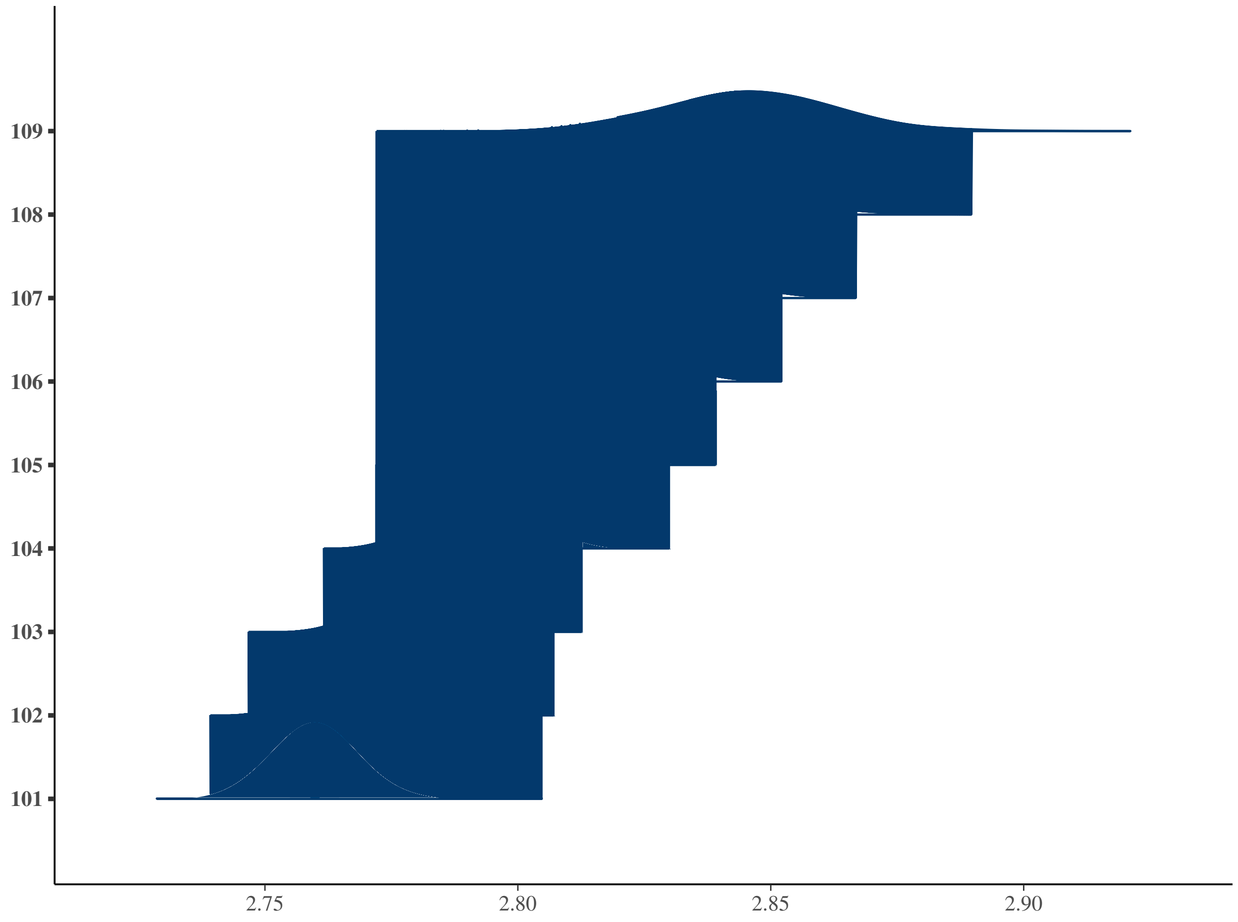
```
## Warning: package 'bayesplot' was built under R version 3.4.4
```

```
mcmc_areas(theta, bw = 0.001)
```

```
## Warning: package 'bindrcpp' was built under R version 3.4.4
```



```
level <- expand_sample(mcmc_bsm, "alpha",  times = 101:109, states = 1)
mcmc_areas(level, bw = 0.005)
```



Using the `summary` method we can obtain the posterior estimates of the trend:

```r
sumr <- summary(mcmc_bsm)
level <- sumr$states$Mean[, 1]
lwr <- level - 1.96 * sumr$states$SD[, 1]
upr <- level + 1.96 * sumr$states$SD[, 1]
ts.plot(UKgas_model$y, cbind(level, lwr, upr), col = c(1, 2, 2, 2), lty = c(1, 1, 2, 2))
```

For prediction intervals, we first build a model for the future time points, and then use the previously obtained posterior samples for the prediction:

```r
future_model <- UKgas_model
future_model$y <- ts(rep(NA, 24), start = end(UKgas_model$y) + c(0, 1), frequency = 4)
pred <- predict(mcmc_bsm, future_model, probs = c(0.025, 0.1, 0.9, 0.975))
ts.plot(log10(UKgas), pred$mean, pred$intervals[, -3],
  col = c(1, 2, c(3, 4, 4, 3)), lty = c(1, 1, rep(2, 4)))
```

Now same with ggplot2:

```r
require("ggplot2")
```

```
## Loading required package: ggplot2
```

```r
level_fit <- ts(colMeans(expand_sample(mcmc_bsm, "alpha")$level), start = start(UKgas_model$y),
  frequency = 4)
autoplot(pred, y = UKgas_model$y, fit = level_fit, interval_color = "red", alpha_fill = 0.2)
```

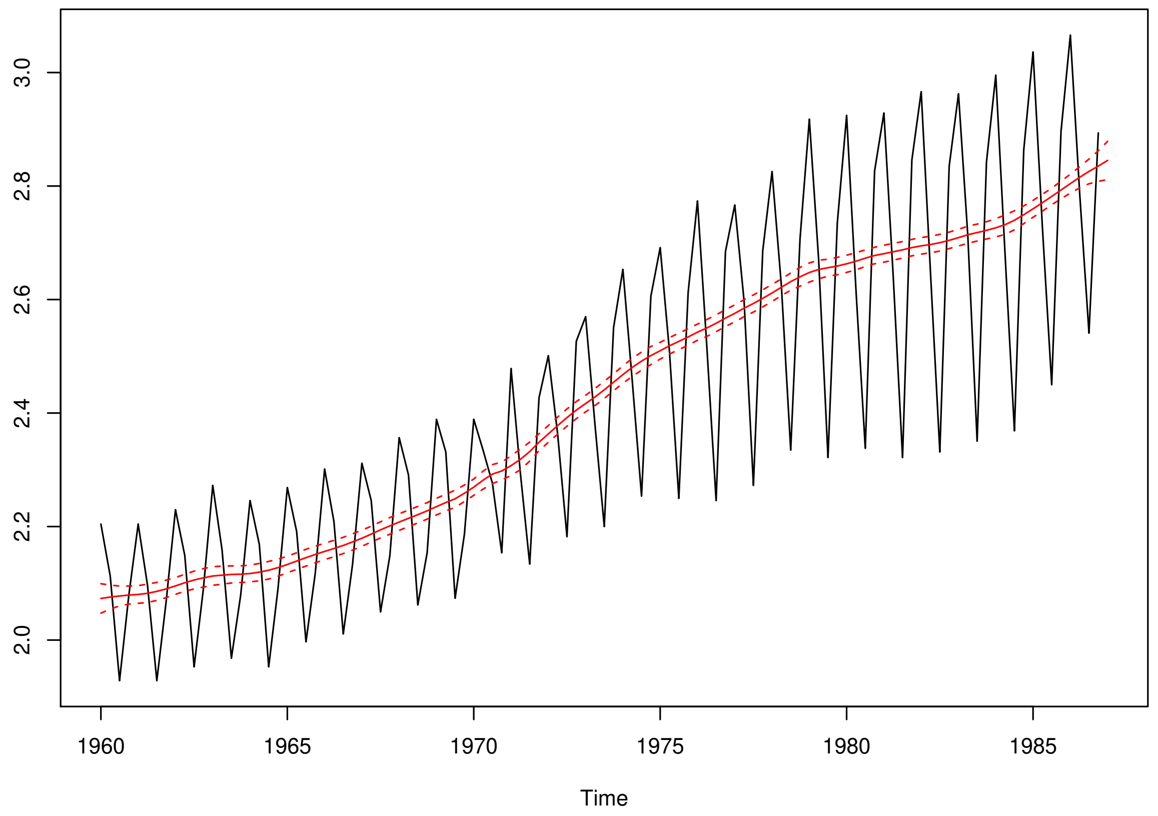


Figure 1: Smoothed trend component with 95% intervals.

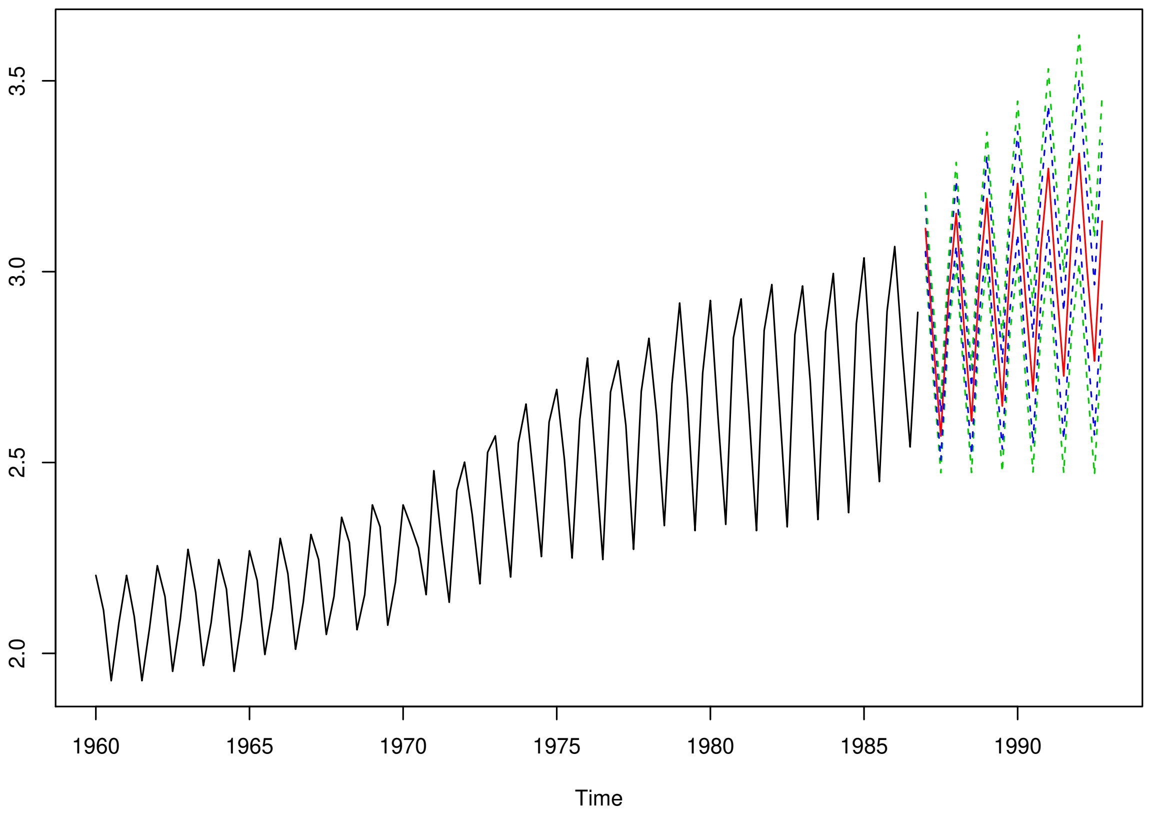


Figure 2: Mean predictions and prediction intervals.

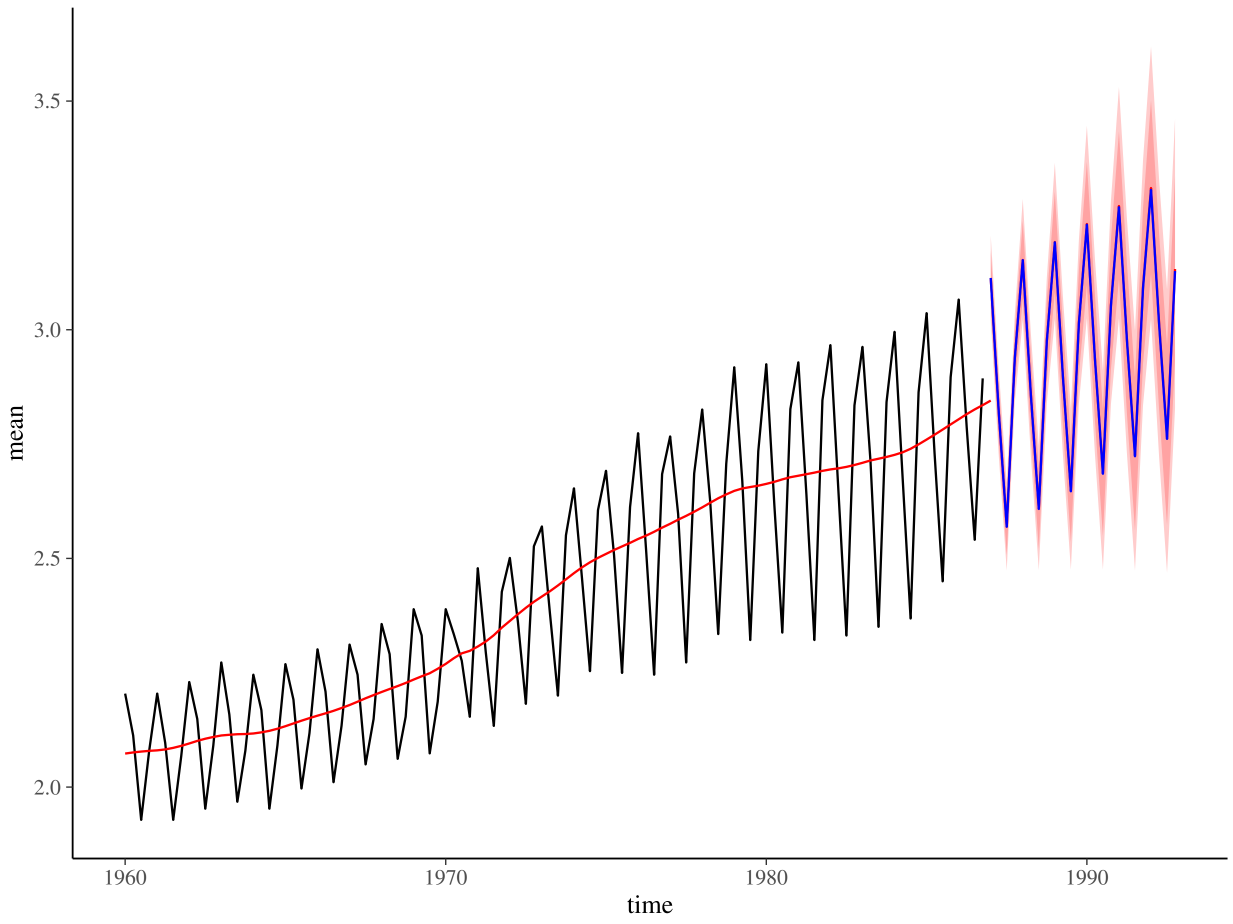


Figure 3: Prediction plots with ggplot2.

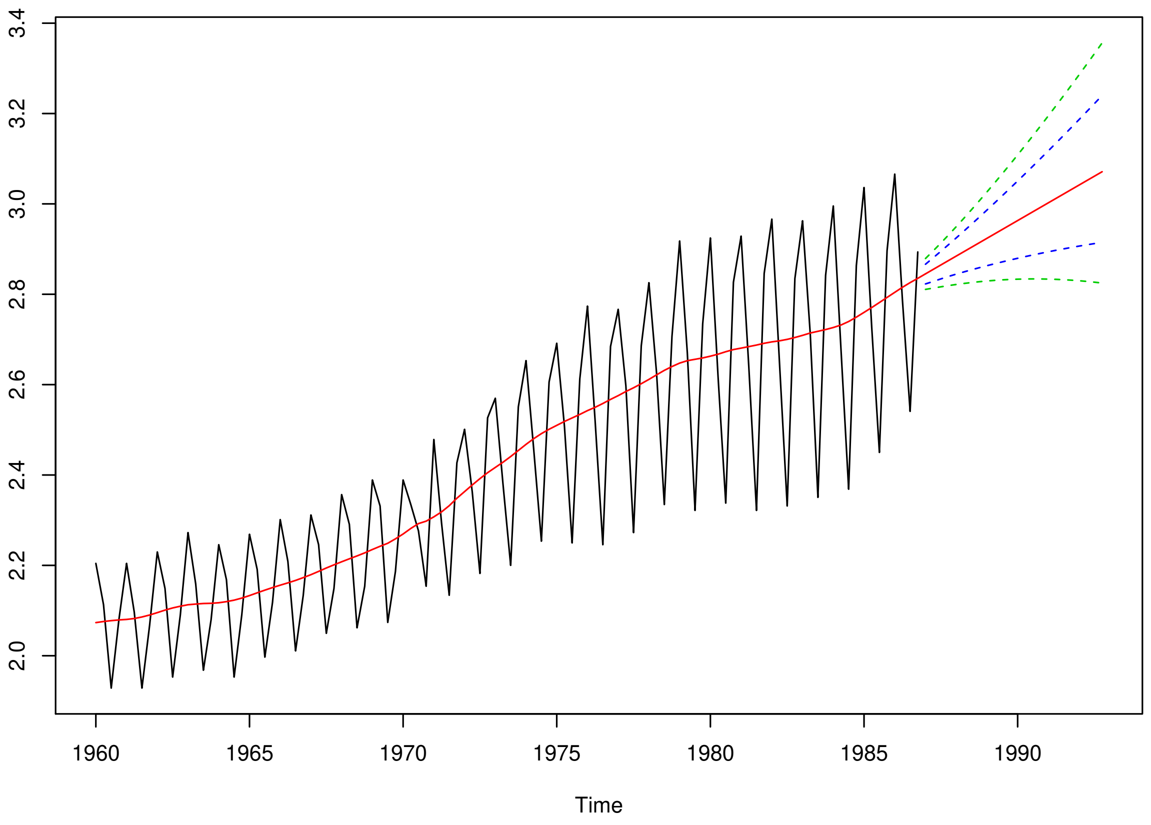


Figure 4: State prediction.

We can also obtain predictions in terms of individual components of the state vector:

```
pred_state <- predict(mcmc_bsm, future_model, probs = c(0.025, 0.1, 0.9, 0.975), type = "state")
ts.plot(log10(UKgas), level_fit, pred_state$mean[,"level"], pred_state$intervals$level[, -3],
  col = c(1, 2, 2, c(3, 4, 4, 3)), lty = c(1, 1, 1, rep(2, 4)))
```

Acknowledgements

This work has been supported by the Academy of Finland research grants 284513 and 312605.

References

Appendix

Template for general linear-Gaussian model

```
// A template for building a general linear-Gaussian state space model
// Here we define an univariate local linear trend model which could be
// constructed also with bsm function.

#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::interfaces(r, cpp)]]

// theta:
// theta(0) = standard deviation sigma_y
// theta(1) = standard deviation sigma_level
// theta(2) = standard deviation sigma_slope
//
// Function for the prior mean of alpha_1
// [[Rcpp::export]]
arma::vec a1_fn(const arma::vec& theta, const arma::vec& known_params) {
  return arma::vec(2, arma::fill::zeros);
}
// Function for the prior variance of alpha_1
// [[Rcpp::export]]
arma::mat P1_fn(const arma::vec& theta, const arma::vec& known_params) {

  arma::mat P1(2, 2, arma::fill::zeros);
  P1(0, 0) = 1000;
  P1(1, 1) = 1000;
  return P1;
}

// Function for the Cholesky of the observational level covariance matrix
// [[Rcpp::export]]
arma::mat H_fn(const unsigned int t, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {
  // note no transformations, needs to check for positivity in prior
  // we could also use exp(theta) here and work with the corresponding prior
  arma::mat H(1,1);
  H(0, 0) = theta(0);
  return H;
}

// Function for the Cholesky of state level covariance matrix
// [[Rcpp::export]]
arma::mat R_fn(const unsigned int t, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {
  arma::mat R(2, 2, arma::fill::zeros);
  R(0, 0) = theta(1);
  R(1, 1) = theta(2);
  return R;
}


// Z function
// [[Rcpp::export]]
arma::mat Z_fn(const unsigned int t, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {
  arma::mat Z(1, 2, arma::fill::zeros);
  Z(0, 0) = 1.0;
  return Z;
}

// T function
// [[Rcpp::export]]
arma::mat T_fn(const unsigned int t, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {
  arma::mat T(2, 2, arma::fill::ones);
  T(1, 0) = 0.0;
  return T;
}

// input to state equation
// [[Rcpp::export]]
arma::vec C_fn(const unsigned int t, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {
  return arma::vec(2, arma::fill::zeros);
}
// input to observation equation
// [[Rcpp::export]]
arma::vec D_fn(const unsigned int t, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {
  return arma::vec(1, arma::fill::zeros);
}

// # log-prior pdf for theta
// [[Rcpp::export]]
double log_prior_pdf(const arma::vec& theta) {

  double log_pdf = -std::numeric_limits<double>::infinity();
  if (arma::all(theta >= 0)) {
   log_pdf = R::dnorm(theta(0), 0, 10, 1) +
     R::dnorm(theta(1), 0, 10, 1) +
     R::dnorm(theta(2), 0, 10, 1);
  }

  return log_pdf;
}


// Create pointers, no need to touch this if
// you don't alter the function names above
// [[Rcpp::export]]
Rcpp::List create_xptrs() {

  // typedef for a pointer returning matrices Z, H, T, and R
  typedef arma::mat (*lmat_fnPtr)(const unsigned int t, const arma::vec& theta,
    const arma::vec& known_params, const arma::mat& known_tv_params);
  // typedef for a pointer of linear function of lgg-model equation returning vectors D and C
  typedef arma::vec (*lvec_fnPtr)(const unsigned int t, const arma::vec& theta,
    const arma::vec& known_params, const arma::mat& known_tv_params);

  // typedef for a pointer returning vector a1
  typedef arma::vec (*a1_fnPtr)(const arma::vec& theta, const arma::vec& known_params);
  // typedef for a pointer returning matrix P1
  typedef arma::mat (*P1_fnPtr)(const arma::vec& theta, const arma::vec& known_params);
  // typedef for a pointer of log-prior function
  typedef double (*prior_fnPtr)(const arma::vec&);

  return Rcpp::List::create(
    Rcpp::Named("a1_fn") = Rcpp::XPtr<a1_fnPtr>(new a1_fnPtr(&a1_fn)),
    Rcpp::Named("P1_fn") = Rcpp::XPtr<P1_fnPtr>(new P1_fnPtr(&P1_fn)),
    Rcpp::Named("Z_fn") = Rcpp::XPtr<lmat_fnPtr>(new lmat_fnPtr(&Z_fn)),
    Rcpp::Named("H_fn") = Rcpp::XPtr<lmat_fnPtr>(new lmat_fnPtr(&H_fn)),
    Rcpp::Named("T_fn") = Rcpp::XPtr<lmat_fnPtr>(new lmat_fnPtr(&T_fn)),
    Rcpp::Named("R_fn") = Rcpp::XPtr<lmat_fnPtr>(new lmat_fnPtr(&R_fn)),
    Rcpp::Named("D_fn") = Rcpp::XPtr<lvec_fnPtr>(new lvec_fnPtr(&D_fn)),
    Rcpp::Named("C_fn") = Rcpp::XPtr<lvec_fnPtr>(new lvec_fnPtr(&C_fn)),
    Rcpp::Named("log_prior_pdf") =
      Rcpp::XPtr<prior_fnPtr>(new prior_fnPtr(&log_prior_pdf)));
}
```

Template for non-linear Gaussian model

```
// A template for building a general non-linear Gaussian state space model
// Here we define an univariate growth model (see vignette growth_model)

#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::interfaces(r, cpp)]]

// Function for the prior mean of alpha_1
// [[Rcpp::export]]
arma::vec a1_fn(const arma::vec& theta, const arma::vec& known_params) {

  arma::vec a1(2);
  a1(0) = known_params(2);
  a1(1) = known_params(3);
  return a1;
}
// Function for the prior covariance matrix of alpha_1
// [[Rcpp::export]]
arma::mat P1_fn(const arma::vec& theta, const arma::vec& known_params) {

  arma::mat P1(2, 2, arma::fill::zeros);
  P1(0,0) = known_params(4);
  P1(1,1) = known_params(5);
  return P1;
}

// Function for the Cholesky of observational level covariance matrix
// [[Rcpp::export]]
```

```cpp
arma::mat H_fn(const unsigned int t, const arma::vec& alpha, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {
  arma::mat H(1,1);
  H(0, 0) = theta(0);
  return H;
}

// Function for the Cholesky of state level covariance matrix
// [[Rcpp::export]]
arma::mat R_fn(const unsigned int t, const arma::vec& alpha, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {
  arma::mat R(2, 2, arma::fill::zeros);
  R(0, 0) = theta(1);
  R(1, 1) = theta(2);
  return R;
}


// Z function
// [[Rcpp::export]]
arma::vec Z_fn(const unsigned int t, const arma::vec& alpha, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {
  arma::vec tmp(1);
  tmp(0) = alpha(1);
  return tmp;
}
// Jacobian of Z function
// [[Rcpp::export]]
arma::mat Z_gn(const unsigned int t, const arma::vec& alpha, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {
  arma::mat Z_gn(1, 2);
  Z_gn(0, 0) = 0.0;
  Z_gn(0, 1) = 1.0;
  return Z_gn;
}

// T function
// [[Rcpp::export]]
arma::vec T_fn(const unsigned int t, const arma::vec& alpha, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {

  double dT = known_params(0);
  double k = known_params(1);

  arma::vec alpha_new(2);
  alpha_new(0) = alpha(0);
  alpha_new(1) = k * alpha(1) * exp(alpha(0) * dT) /
    (k + alpha(1) * (exp(alpha(0) * dT) -1));

  return alpha_new;
}

// Jacobian of T function
// [[Rcpp::export]]
arma::mat T_gn(const unsigned int t, const arma::vec& alpha, const arma::vec& theta,
  const arma::vec& known_params, const arma::mat& known_tv_params) {

  double dT = known_params(0);
  double k = known_params(1);

  double tmp = exp(alpha(0) * dT) /
    std::pow(k + alpha(1) * (exp(alpha(0) * dT) - 1), 2);

  arma::mat Tg(2, 2);
  Tg(0, 0) = 1.0;
  Tg(0, 1) = 0;
  Tg(1, 0) = k * alpha(1) * dT * (k - alpha(1)) * tmp;
  Tg(1, 1) = k * k * tmp;

  return Tg;
}

// # log-prior pdf for theta
// [[Rcpp::export]]
double log_prior_pdf(const arma::vec& theta) {

  double log_pdf;
  if(arma::any(theta < 0)) {
     log_pdf = -std::numeric_limits<double>::infinity();
   } else {
    // weakly informative priors.
    // Note that negative values are handled above
    log_pdf = R::dnorm(theta(0), 0, 10, 1) + R::dnorm(theta(1), 0, 10, 1) +
      R::dnorm(theta(2), 0, 10, 1);
  }
  return log_pdf;
}

// Create pointers, no need to touch this if
// you don't alter the function names above
// [[Rcpp::export]]
Rcpp::List create_xptrs() {

  // typedef for a pointer of nonlinear function of model equation returning vec (T, Z)
  typedef arma::vec (*nvec_fnPtr)(const unsigned int t, const arma::vec& alpha,
    const arma::vec& theta, const arma::vec& known_params, const arma::mat& known_tv_params);
  // typedef for a pointer of nonlinear function returning mat (Tg, Zg, H, R)
  typedef arma::mat (*nmat_fnPtr)(const unsigned int t, const arma::vec& alpha,
    const arma::vec& theta, const arma::vec& known_params, const arma::mat& known_tv_params);

  // typedef for a pointer returning a1
  typedef arma::vec (*a1_fnPtr)(const arma::vec& theta, const arma::vec& known_params);
  // typedef for a pointer returning P1
  typedef arma::mat (*P1_fnPtr)(const arma::vec& theta, const arma::vec& known_params);
  // typedef for a pointer of log-prior function
  typedef double (*prior_fnPtr)(const arma::vec&);
  return Rcpp::List::create(
    Rcpp::Named("a1_fn") = Rcpp::XPtr<a1_fnPtr>(new a1_fnPtr(&a1_fn)),
    Rcpp::Named("P1_fn") = Rcpp::XPtr<P1_fnPtr>(new P1_fnPtr(&P1_fn)),
    Rcpp::Named("Z_fn") = Rcpp::XPtr<nvec_fnPtr>(new nvec_fnPtr(&Z_fn)),
    Rcpp::Named("H_fn") = Rcpp::XPtr<nmat_fnPtr>(new nmat_fnPtr(&H_fn)),
    Rcpp::Named("T_fn") = Rcpp::XPtr<nvec_fnPtr>(new nvec_fnPtr(&T_fn)),
    Rcpp::Named("R_fn") = Rcpp::XPtr<nmat_fnPtr>(new nmat_fnPtr(&R_fn)),
    Rcpp::Named("Z_gn") = Rcpp::XPtr<nmat_fnPtr>(new nmat_fnPtr(&Z_gn)),
    Rcpp::Named("T_gn") = Rcpp::XPtr<nmat_fnPtr>(new nmat_fnPtr(&T_gn)),
    Rcpp::Named("log_prior_pdf") =
      Rcpp::XPtr<prior_fnPtr>(new prior_fnPtr(&log_prior_pdf)));

}
```

Template for SDE model

```
// A template for building a univariate discretely observed diffusion model
// Here we define a latent Ornsteinâ€"Uhlenbeck process with Poisson observations
// d\alpha_t = \rho (\nu - \alpha_t) dt + \sigma dB_t, t>=0
// y_k ~ Poisson(exp(\alpha_k)), k = 1,...,n

#include <RcppArmadillo.h>
// [[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::interfaces(r, cpp)]]

// x: state
// theta: vector of parameters

// theta(0) = rho
// theta(1) = nu
// theta(2) = sigma

// Drift function
// [[Rcpp::export]]
double drift(const double x, const arma::vec& theta) {
  return theta(0) * (theta(1) - x);
}
// diffusion function
// [[Rcpp::export]]
double diffusion(const double x, const arma::vec& theta) {
  return theta(2);
}
// Derivative of the diffusion function
// [[Rcpp::export]]
double ddiffusion(const double x, const arma::vec& theta) {
  return 0.0;
}

// log-density of the prior
// [[Rcpp::export]]
double log_prior_pdf(const arma::vec& theta) {
  double log_pdf;
  if(theta(0) <= 0.0 || theta(2) <= 0.0) {
    log_pdf = -std::numeric_limits<double>::infinity();
  } else {
    // weakly informative priors.
    // Note that negative values are handled above
    log_pdf = R::dnorm(theta(0), 0, 10, 1) + R::dnorm(theta(1), 0, 10, 1) +
      R::dnorm(theta(2), 0, 10, 1);
  }
  return log_pdf;
}

// log-density of observations
// [[Rcpp::export]]
arma::vec log_obs_density(const double y,
  const arma::vec& alpha, const arma::vec& theta) {

  arma::vec log_pdf(alpha.n_elem);
  for (unsigned int i = 0; i < alpha.n_elem; i++) {
    log_pdf(i) = R::dpois(y, exp(alpha(i)), 1);
  }
  return log_pdf;
}


// Function which returns the pointers to above functions (no need to modify)

// [[Rcpp::export]]
Rcpp::List create_xptrs() {
  // typedef for a pointer of drift/volatility function
  typedef double (*funcPtr)(const double x, const arma::vec& theta);
  // typedef for log_prior_pdf
  typedef double (*prior_funcPtr)(const arma::vec& theta);
  // typedef for log_obs_density
  typedef arma::vec (*obs_funcPtr)(const double y,
    const arma::vec& alpha, const arma::vec& theta);

  return Rcpp::List::create(
    Rcpp::Named("drift") = Rcpp::XPtr<funcPtr>(new funcPtr(&drift)),
    Rcpp::Named("diffusion") = Rcpp::XPtr<funcPtr>(new funcPtr(&diffusion)),
    Rcpp::Named("ddiffusion") = Rcpp::XPtr<funcPtr>(new funcPtr(&ddiffusion)),
    Rcpp::Named("prior") = Rcpp::XPtr<prior_funcPtr>(new prior_funcPtr(&log_prior_pdf)),
    Rcpp::Named("obs_density") = Rcpp::XPtr<obs_funcPtr>(new obs_funcPtr(&log_obs_density)));
}
```

Andrieu, Christophe, and Gareth O. Roberts. 2009. "The Pseudo-Marginal Approach for Efficient Monte Carlo Computations." *Annstat* 37 (2): 697–725.

Andrieu, Christophe, Arnaud Doucet, and Roman Holenstein. 2010. "Particle Markov Chain Monte Carlo Methods." *J. R. Stat. Soc. Ser. B Stat. Methodol.* 72 (3): 269–342.

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.

Beaumont, Mark A. 2003. "Estimation of Population Growth or Decline in Genetically Monitored Populations."

Genetics 164: 1139–60.

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. 2000. “Time Series Analysis of Non-Gaussian Observations Based on State Space Models from Both Classical and Bayesian Perspectives.” *Journal of Royal Statistical Society B* 62: 3–56.

———. 2002. “A Simple and Efficient Simulation Smoother for State Space Time Series Analysis.” *Biometrika* 89: 603–15.

———. 2012. *Time Series Analysis by State Space Methods*. 2nd ed. New York: Oxford University Press.

Durbin, James, and Siem Jan Koopman. 1997. “Monte Carlo Maximum Likelihood Estimation for Non-Gaussian State Space Models.” *Biometrika* 84 (3): 669–84. doi:10.1093/biomet/84.3.669.

Franks, Jordan, and Matti Vihola. 2017. “Importance Sampling and Delayed Acceptance via a Peskun Type Ordering.” Preprint arXiv:1706.09873.

Gordon, Neil J., D. J. Salmond, and A. F. M. Smith. 1993. “Novel Approach to Nonlinear/Non-Gaussian Bayesian State Estimation.” *IEE Proceedings-F* 140 (2): 107–13.

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

Helske, Jouni. 2016. *ramcmc: Robust Adaptive Metropolis Algorithm*. http://github.com/helske/ramcmc.

———. 2017. “KFAS: Exponential Family State Space Models in R.” *Journal of Statistical Software* 78 (10): 1–39. doi:10.18637/jss.v078.i10.

Helske, Satu, and Jouni Helske. 2017. “Mixture Hidden Markov Models for Sequence Data: The seqHMM Package in R.” *Submitted to Journal of Statistical Software*. http://arxiv.org/abs/1704.00543.

Iacus, Stefano Maria. 2016. *Sde: Simulation and Inference for Stochastic Differential Equations*. https://CRAN.R-project.org/package=sde.

Kitagawa, Genshiro. 1996. “Monte Carlo Filter and Smoother for Non-Gaussian Nonlinear State Space Models.” *J-Cgs* 5 (1): 1–25.

Lin, L., K.F. Liu, and J. Sloan. 2000. “A Noisy Monte Carlo Algorithm.” *Physical Review D* 61.

Lindgren, Finn, and Håvard Rue. 2015. “Bayesian Spatial Modelling with R-INLA.” *Journal of Statistical Software* 63 (19): 1–25. doi:10.18637/jss.v063.i19.

Petris, Giovanni, and Sonia Petrone. 2011. “State Space Models in R.” *Journal of Statistical Software* 41 (4): 1–25. doi:10.18637/jss.v041.i04.

Plummer, Martyn, Nicky Best, Kate Cowles, and Karen Vines. 2006. “CODA: Convergence Diagnosis and Output Analysis for Mcmc.” *R News* 6 (1): 7–11. http://CRAN.R-project.org/doc/Rnews/.

R Core Team. 2016. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

Tusell, Fernando. 2011. “Kalman Filtering in R.” *Journal of Statistical Software* 39 (2): 1–27. doi:10.18637/jss.v039.i02.

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.

Vihola, Matti, Jouni Helske, and Jordan Franks. 2017. “Importance Sampling Type Estimators Based on Approximate Marginal Markov Chain Monte Carlo.” Preprint arXiv:1609.02541v3.