State Space Models in Stan

Jeffrey B. Arnold

2016-07-01

Contents

Chapter 1

Introduction

This contains documentation for “State Space Models in Stan”

Chapter 2

The Linear State Space Model

[11, Sec 3.1]

The linear Gaussian state space model (SSM)[1] the the n-dimensional observation sequence $\boldsymbol{y}_1, \dots, \boldsymbol{y}_n$,

$$
\begin{aligned}
\boldsymbol{y}_t &= \boldsymbol{d}_t + \boldsymbol{Z}_t \boldsymbol{\alpha}_t + \varepsilon_t, & \varepsilon_t &\sim N(0, \boldsymbol{H}_t), \\
\boldsymbol{\alpha}_{t+1} &= \boldsymbol{c}_t + \boldsymbol{T}_t \boldsymbol{\alpha}_t + \boldsymbol{R}_t \boldsymbol{\eta}_t, & \boldsymbol{\eta}_t &\sim N(0, \boldsymbol{Q}_t), \\
& & \boldsymbol{\alpha}_1 &\sim N(\boldsymbol{a}_1, \boldsymbol{P}_1).
\end{aligned}
$$

for $t = 1, \dots, n$. The first equation is called the *observation* or *measurement equation*. The second equation is called the *state*, *transition*, or *system equation*. The vector $\boldsymbol{y}_t$ is a $p \times 1$ vector called the *observation vector*. The vector $\alpha\alpha_t$ is a $m \times 1$ vector called the *state vector*. The matrices are vectors, $\boldsymbol{Z}_t$,$\boldsymbol{T}_t$, $\boldsymbol{R}_t$, $\boldsymbol{H}_t$, $\boldsymbol{Q}_t$, c_t, and d_t are called the *system matrices*. The system matrices are considered fixed and known in the filtering and smoothing equations below, but can be parameters themselves. The $p \times m$ matrix $\boldsymbol{Z}_t$ links the observation vector $\boldsymbol{y}_t$ with the state vector $\boldsymbol{\alpha}_t$. The $m \times m$ transition matrix $\boldsymbol{T}_t$ determines the evolution of the state vector, $\boldsymbol{\alpha}_t$. The $q \times 1$ vector $\boldsymbol{\eta}_t$ is called the *state disturbance vector*, and the $p \times 1$ vector $\boldsymbol{\varepsilon}_t$ is called the *observation disturbance vector*. An assumption is that the state and observation disturbance vectors are uncorrelated, $\operatorname{Cov}(\boldsymbol{\varepsilon}_t, \boldsymbol{\eta}_t) = 0$.

In a general state space model, the normality assumptions of the densities of $\boldsymbol{\varepsilon}$ and $\boldsymbol{\eta}$ are dropped.

In many cases $\boldsymbol{R}_t$ is the identity matrix. It is possible to define $\eta^*_t = \boldsymbol{R}_t \boldsymbol{\eta}_t$, and $\boldsymbol{Q}^* = \boldsymbol{R}_t \boldsymbol{Q}'_t \boldsymbol{R}'_t$. However, if $\boldsymbol{R}_t$ is $m \times q$ and $q < m$, and $\boldsymbol{Q}_t$ is nonsingular, then it is useful to work with the nonsingular $\boldsymbol{\eta}_t$ rather than a singular $\boldsymbol{\eta}^*_t$.

The initial state vector $\boldsymbol{\alpha}_1$ is assume to be generated as,

$$
\alpha_1 \sim N(\boldsymbol{a}_1, \boldsymbol{P}_1)
$$

independently of the observation and state disturbances $\boldsymbol{\varepsilon}$ and $\boldsymbol{\eta}$. The values of $\boldsymbol{a}_1$ and $\boldsymbol{P}_1$ can be considered as given and known in most stationary processes. When the process is nonstationary, the elements of $\boldsymbol{a}_1$ need to be treated as unknown and estimated. This is called *initialization*.

Table 2.1: Dimensions of matrices and vectors in the SSM

matrix/vector	dimension	name
$\boldsymbol{y}_t$	$p \times 1$	observation vector
$\boldsymbol{\alpha}_t$	$m \times 1$	(unobserved) state vector
$\boldsymbol{\varepsilon}_t$	$m \times 1$	observation disturbance (error)
$\boldsymbol{\eta}_t$	$q \times 1$	state disturbance (error)

[1] This is also called a dynamic linear model (DLM).

matrix/vector	dimension	name
$\boldsymbol{a}_1$	$m \times 1$	initial state mean
$\boldsymbol{c}_t$	$m \times 1$	state intercept
$\boldsymbol{d}_t$	$p \times 1$	observation intercept
$\boldsymbol{Z}_t$	$p \times m$	design matrix
$\boldsymbol{T}_t$	$m \times m$	transition matrix
$\boldsymbol{H}_t$	$p \times p$	observation covariance matrix
$\boldsymbol{R}_t$	$m \times q$	state covariance selection matrix
$\boldsymbol{Q}_t$	$q \times q$	state covariance matrix
$\boldsymbol{P}_1$	$m \times m$	initial state covariance matrix

Chapter 3

Filtering and Smoothing

3.1 Filtering

From [11, Sec 4.3]

Let $\boldsymbol{a}_t = \mathrm{E}(\boldsymbol{\alpha}_t | y_{1,\dots,t-1})$ be the expected value $\boldsymbol{P}_t = \mathrm{Var}(\boldsymbol{\alpha}_t | y_{1,\dots,t-1})$ be the variance of the state in $t+1$ given data up to time t. To calculate $\boldsymbol{\alpha}_{t+1}$ and $\boldsymbol{P}_{t+1}$ given the arrival of new data at time t,

$$
\begin{aligned}
\boldsymbol{v}_t &= \boldsymbol{y}_t - \boldsymbol{Z}_t \boldsymbol{a}_t - \boldsymbol{d}_t, \\
\boldsymbol{F}_t &= \boldsymbol{Z}_t \boldsymbol{P}_t \boldsymbol{Z}'_t + \boldsymbol{H}_t, \\
\boldsymbol{K}_t &= \boldsymbol{T}_t \boldsymbol{P}_t \boldsymbol{Z}'_t \boldsymbol{F}_t^{-1} \\
\boldsymbol{a}_{t+1} &= \boldsymbol{T}_t \boldsymbol{a}_t + \boldsymbol{K}_t \boldsymbol{v}_t + \boldsymbol{c}_t \\
\boldsymbol{P}_{t+1} &= \boldsymbol{T}_t \boldsymbol{P}_t (\boldsymbol{T}_t - \boldsymbol{K}_t \boldsymbol{Z}_t)' + \boldsymbol{R}_t \boldsymbol{Q}_t \boldsymbol{R}'_t.
\end{aligned}
$$

The vector $\boldsymbol{v}_t$ are the *one-step ahead forecast errors$, and the matrix* $\boldsymbol{K}_t$ *is called the* Kalman gain*.

The filter can also be written to estimate the *filtered states*, where $\boldsymbol{a}_{t|t} = \mathrm{E}(\boldsymbol{\alpha}_t | y_{1,\dots,t})$ is the expected value and $\boldsymbol{P}_{t|t} = \mathrm{Var}(\boldsymbol{\alpha}_t | y_{1,\dots,t})$ is the variance of the state $\boldsymbol{\alpha}_t$ given information up to *and including* $\boldsymbol{y}_t$. The filter written this way is,

$$
\begin{aligned}
\boldsymbol{v}_t &= \boldsymbol{y}_t - \boldsymbol{Z}_t \boldsymbol{a}_t - \boldsymbol{d}_t, \\
\boldsymbol{F}_t &= \boldsymbol{Z}_t \boldsymbol{P}_t \boldsymbol{Z}'_t + \boldsymbol{H}_t, \\
\boldsymbol{a}_{t|t} &= \boldsymbol{a}_t + \boldsymbol{P}_t \boldsymbol{Z}'_t \boldsymbol{F}_t^{-1} v_t, \\
\boldsymbol{P}_{t|t} &= \boldsymbol{P}_t - \boldsymbol{P}_t \boldsymbol{Z}'_t \boldsymbol{F}_t^{-1} \boldsymbol{Z}_t \boldsymbol{P}_t, \\
\boldsymbol{a}_{t+1} &= \boldsymbol{T}_t \boldsymbol{a}_{t|t} + \boldsymbol{c}_t, \\
\boldsymbol{P}_{t+1} &= \boldsymbol{T}_t \boldsymbol{P}_{t|t} \boldsymbol{T}'_t + \boldsymbol{R}_t \boldsymbol{Q}_t \boldsymbol{R}'_t.
\end{aligned}
$$

Table 3.1: Dimensions of matrices and vectors in the SSM

matrix/vector	dimension
$\boldsymbol{v}_t$	$p \times 1$
$\boldsymbol{a}_t$	$m \times 1$
$\boldsymbol{a}_{t\|t}$	$m \times 1$
$\boldsymbol{F}_t$	$p \times p$
$\boldsymbol{K}_t$	$m \times p$
$\boldsymbol{P}_t$	$m \times m$
$\boldsymbol{P}_{t\|T}$	$m \times m$

matrix/vector	dimension
$\boldsymbol{x}_t$	$m \times 1$
$\boldsymbol{L}_t$	$m \times m$

See [11, Sec 4.3.4]: For a time-invariant state space model, the Kalman recursion for $\boldsymbol{P}_{t+1}$ converges to a constant matrix $\bar{\boldsymbol{P}}$,

$$\bar{\boldsymbol{P}} = \boldsymbol{T}\bar{\boldsymbol{P}}\boldsymbol{T}' - \boldsymbol{T}\bar{\boldsymbol{P}}\boldsymbol{Z}'\bar{\boldsymbol{F}}^{-1}\boldsymbol{Z}\bar{\boldsymbol{P}}\boldsymbol{T}' + \boldsymbol{R}\boldsymbol{Q}\boldsymbol{R}',$$

where $\bar{\boldsymbol{F}} = \boldsymbol{Z}\bar{\boldsymbol{P}}\boldsymbol{Z}' + \boldsymbol{H}$.

See [11, Sec 4.3.5]: The *state estimation error* is,

$$\boldsymbol{x}_t = \boldsymbol{\alpha}_t - \boldsymbol{a}_t,$$

where $\mathrm{Var}(\boldsymbol{x}_t) = \boldsymbol{P}_t$. The v_t are sometimes called *innovations*, since they are the part of $\boldsymbol{y}_t$ not predicted from the past. The innovation analog of the state space model is

$$\begin{aligned}
\boldsymbol{v}_t &= \boldsymbol{Z}_t\boldsymbol{x}_t + \boldsymbol{\varepsilon}_t, \\
\boldsymbol{x}_{t+1} &= \boldsymbol{L}\boldsymbol{x}_t + \boldsymbol{R}_t\boldsymbol{\eta}_t - \boldsymbol{K}_t\boldsymbol{\varepsilon}_t, \\
\boldsymbol{K}_t &= \boldsymbol{T}_t\boldsymbol{P}_t\boldsymbol{Z}_t'\boldsymbol{F}_t^{-1}, \\
\boldsymbol{L}_t &= \boldsymbol{T}_t - \boldsymbol{K}_t\boldsymbol{Z}_t, \boldsymbol{P}_{t+1} &= \boldsymbol{T}_t\boldsymbol{P}_t\boldsymbol{L}_t' + \boldsymbol{R}_t\boldsymbol{Q}_t\boldsymbol{R}_T'.
\end{aligned}$$

These recursions allow for a simpler derivation of $\boldsymbol{P}_{t+1}$, and are useful for the smoothing recursions. Moreover, the one-step ahead forecast errors are indendendent, which allows for a simple derivation of the log-likelihood.

Alternative methods **TODO**

- square-root filtering
- precision filters
- sequential filtering

3.2 Smoothing

While filtering calculates the conditional densities the states and disturbances given data prior to or up to the current time, smoothing calculates the conditional densities states and disturbances given the entire series of observations, $\boldsymbol{y}_{1:n}$.

State smoothing calculates the conditional mean, $\hat{\boldsymbol{\alpha}}_t = \mathrm{E}(\boldsymbol{\alpha}_t|\boldsymbol{y}_{1:n})$, and variance, $\boldsymbol{V}_t = \mathrm{Var}(\boldsymbol{\alpha}_t|\boldsymbol{y}_{1:n})$, of the states. Observation disturbance smoothing calculates the conditional mean, $\hat{\boldsymbol{\varepsilon}}_t = \mathrm{E}(\boldsymbol{\varepsilon}_t|\boldsymbol{y}_{1:n})$, and variance, $\mathrm{Var}(\boldsymbol{\varepsilon}_t|\boldsymbol{y}_{1:n})$, of the state disturbances. Likewise, state disturbance smoothing calculates the conditional mean, $\hat{\boldsymbol{\eta}}_t = \mathrm{E}(\boldsymbol{\eta}_t|\boldsymbol{y}_{1:n})$, and variance, $\mathrm{Var}(\boldsymbol{\eta}_t|\boldsymbol{y}_{1:n})$, of the state disturbances.

Table 3.2: Dimensions of vectors and matrices used in smoothing recursions

Vector/Matrix	Dimension
$\boldsymbol{r}_t$	$m \times 1$
$\boldsymbol{\alpha}_t$	$m \times 1$
$\boldsymbol{u}_t$	$p \times 1$
$\hat{\boldsymbol{\varepsilon}}_t$	$p \times 1$
$\hat{\boldsymbol{\eta}}_t$	$r \times 1$
$\boldsymbol{N}_t$	$m \times m$
$\boldsymbol{V}_t$	$m \times m$
$\boldsymbol{D}_t$	$p \times p$

3.2.1 State Smoothing

Smoothing calculates conditional density of the states given all observations, $p(\boldsymbol{\alpha}|\boldsymbol{y}_{1:n})$. Let $\hat{\boldsymbol{\alpha}} = \mathrm{E}(\boldsymbol{\alpha}_t|\boldsymbol{y}_{1:n})$ be the mean and $\boldsymbol{V}_t = \mathrm{Var}(\boldsymbol{\alpha}|\boldsymbol{y}_{1:n})$ be the variance of this density. The following recursions can be used to calculate these densities [11, Sec 4.4.4],

$$\begin{aligned}
\boldsymbol{r}_{t-1} &= \boldsymbol{Z}'_t \boldsymbol{F}_t^{-1} \boldsymbol{v}_t + \boldsymbol{L}'_t \boldsymbol{r}_t, & \boldsymbol{N}_{t-1} &= \boldsymbol{Z}'_t \boldsymbol{F}_t^{-1} \boldsymbol{Z}_t + \boldsymbol{L}'_t \boldsymbol{N}_t \boldsymbol{L}_t, \\
\hat{\boldsymbol{\alpha}}_t &= \boldsymbol{a}_t + \boldsymbol{P}_t \boldsymbol{r}_{t-1}, & \boldsymbol{V}_t &= \boldsymbol{P}_t - \boldsymbol{P}_t \boldsymbol{N}_{t-1} \boldsymbol{P}_t,
\end{aligned}$$

for $t = n, \dots, 1$, with $\boldsymbol{r}_n = \boldsymbol{0}$, and $\boldsymbol{N}_n = \boldsymbol{0}$.

During the filtering pass $\boldsymbol{v}_t$, $\boldsymbol{F}_t$, $\boldsymbol{K}_t$, and $\boldsymbol{P}_t$ for $t = 1, \dots, n$ need to be stored. Alternatively, $\boldsymbol{a}_t$ and $\boldsymbol{P}_t$ only can be stored, and $\boldsymbol{v}_t$, $\boldsymbol{F}_t$, $\boldsymbol{K}_t$ recalculated on the fly. However, since the dimensions of $\boldsymbol{v}_t$, $\boldsymbol{F}_t$, $\boldsymbol{K}_t$ are usually small relative to $\boldsymbol{a}_t$ and $\boldsymbol{P}_t$ is is usually worth storing them.

3.2.2 Disturbance smoothing

Disturbance smoothing calculates the density of the state and observation disturbances ($\boldsymbol{\eta}_t$ and $\boldsymbol{\varepsilon}_t$) given the full series of observations $\boldsymbol{y}_{1:n}$. Let $\hat{\boldsymbol{\varepsilon}}_t = \mathrm{E}(\boldsymbol{\varepsilon}|\boldsymbol{y}_{1:n})$ be the mean and $\mathrm{Var}(\boldsymbol{\varepsilon}_t|\boldsymbol{y}_{1:n})$ be the variance of the smoothed density of the observation disturbances at time t, $p(\boldsymbol{\varepsilon}_t|\boldsymbol{y}_{1:n})$. Likewise, let $\hat{\boldsymbol{\eta}} = \mathrm{E}(\boldsymbol{\eta}_t|\boldsymbol{y}_{1:n})$ be the mean and $\mathrm{Var}(\boldsymbol{\eta}_t|\boldsymbol{y}_{1:n})$ be the variance of the smoothed density of the state disturbances at time t, $p(\boldsymbol{\eta}_t|\boldsymbol{y}_{1:n})$. The following recursions can be used to calculate these values [11, Eq 4.69]:

$$\begin{aligned}
\hat{\boldsymbol{\varepsilon}}_t &= \boldsymbol{H}_t(\boldsymbol{F}^{-1}\boldsymbol{v}_t - \boldsymbol{K}'_t \boldsymbol{r}_t), & \mathrm{Var}(\boldsymbol{\varepsilon}_t|\boldsymbol{Y}_n) &= \boldsymbol{H}_t - \boldsymbol{H}_t(\boldsymbol{F}_t^{-1} + \boldsymbol{K}'_t \boldsymbol{N}_t \boldsymbol{K}_t)\boldsymbol{H}_t, \\
\hat{\boldsymbol{\eta}}_t &= \boldsymbol{Q}_t \boldsymbol{R}'_t \boldsymbol{r}_t, & \mathrm{Var}(\boldsymbol{\eta}_t|\boldsymbol{Y}_n) &= \boldsymbol{Q}_t - \boldsymbol{Q}_t \boldsymbol{R}'_t \boldsymbol{N}_t \boldsymbol{R}_t \boldsymbol{Q}_t, \\
\boldsymbol{r}_{t-1} &= \boldsymbol{Z}'_t \boldsymbol{F}_t^{-1} \boldsymbol{v}_t + \boldsymbol{L}'_t \boldsymbol{r}_t, & \boldsymbol{N}_{t-1} &= \boldsymbol{Z}'_t \boldsymbol{F}_t^{-1} \boldsymbol{Z}_t + \boldsymbol{L}'_t \boldsymbol{N}_t \boldsymbol{L}_t
\end{aligned}$$

Alternatively, these equations can be rewritten as [11, Sec 4.5.3]:

$$\begin{aligned}
\hat{\boldsymbol{\varepsilon}}_t &= \boldsymbol{H}_t \boldsymbol{u}_t, & \mathrm{Var}(\boldsymbol{\varepsilon}_t|\boldsymbol{Y}_n) &= \boldsymbol{H}_t - \boldsymbol{H}_t \boldsymbol{D}_t \boldsymbol{H}_t, \\
\hat{\boldsymbol{\eta}}_t &= \boldsymbol{Q}_t \boldsymbol{R}'_t \boldsymbol{r}_t, & \mathrm{Var}(\boldsymbol{\eta}_t|\boldsymbol{Y}_n) &= \boldsymbol{Q}_t - \boldsymbol{Q}_t \boldsymbol{R}'_t \boldsymbol{N}_t \boldsymbol{R}_t \boldsymbol{Q}_t, \\
\boldsymbol{u}_t &= \boldsymbol{F}^{-1}\boldsymbol{v}_t - \boldsymbol{K}'_t \boldsymbol{r}_t, & \boldsymbol{D}_t &= \boldsymbol{F}_t^{-1} + \boldsymbol{K}'_t \boldsymbol{N}_t \boldsymbol{K}_t, \\
\boldsymbol{r}_{t-1} &= \boldsymbol{Z}'_t \boldsymbol{u}_t + \boldsymbol{T}'_t \boldsymbol{r}_t, & \boldsymbol{N}_{t-1} &= \boldsymbol{Z}'_t \boldsymbol{D}_t \boldsymbol{Z}_t + \boldsymbol{T}'_t \boldsymbol{N}_t \boldsymbol{T}_t - \boldsymbol{Z}'_t \boldsymbol{K}'_t \boldsymbol{N}_t \boldsymbol{T}_t - \boldsymbol{T}'_t \boldsymbol{N}_t \boldsymbol{K}_t \boldsymbol{Z}_t.
\end{aligned}$$

This reformulation can be computationally useful since it relies on the system matrices $\boldsymbol{Z}_t$ and $\boldsymbol{T}_t$ which are often sparse. The disturbance smoothing recursions require only $\boldsymbol{v}_t$, $\boldsymbol{f}_t$, and $\boldsymbol{K}_t$ which are calculated with a forward pass of the Kalman filter. Unlike the state smoother, the disturbance smoothers do not require either the mean ($\boldsymbol{a}_t$) or variance ($\boldsymbol{P}_t$) of the predicted state density.

3.2.3 Fast state smoothing

If the variances of the states do not need to be calculated, then a faster smoothing algorithm can be used (Koopman 1993). The fast state smoother is defined as [11, Sec 4.6.2],

$$\begin{aligned}
\hat{\boldsymbol{\alpha}}_t &= \boldsymbol{T}_t \hat{\boldsymbol{\alpha}}_t + \boldsymbol{R}_t \boldsymbol{Q}_t \boldsymbol{R}'_t \boldsymbol{r}_t, \quad t = 2, \dots, n \\
\hat{\boldsymbol{\alpha}}_1 &= \boldsymbol{a}_1 + \boldsymbol{P}_1 \boldsymbol{r}_0.
\end{aligned}$$

The values of $\boldsymbol{r}_t$ come from the recursions in the disturbance smoother.

3.3 Simulation smoothers

Simulation smoothing draws samples of the states, $p(\boldsymbol{\alpha}_1, \dots, \boldsymbol{\alpha}_n|\boldsymbol{y}_{1:n})$, or disturbances, $p(\boldsymbol{\varepsilon}_1, \dots, \boldsymbol{\varepsilon}_n|\boldsymbol{y}_{1:n})$ and $p(\boldsymbol{\eta}_1, \dots, \boldsymbol{\eta}_n|\boldsymbol{y}_{1:n})$.[^simsmo]

3.3.1 Mean correction simulation smoother

The mean-correction simulation smoother was introduced in Durbin and Koopman [10] . See Durbin and Koopman [11] (Sec 4.9) for an exposition of it. It requires only the previously described filters and smoothers, and generating samples from multivariate distributions.

3.3.1.1 Disturbances

1. Run a filter and disturbance smoother to calculate $\hat{\boldsymbol{\varepsilon}}_{1:n}$ and $\hat{\boldsymbol{\eta}}_{1:(n-1)}$
2. Draw samples from the unconditional distribution of the disturbances,
$$\begin{aligned} \boldsymbol{\eta}_t^+ &\sim N(0, \boldsymbol{H}_t) \quad t = 1, \dots, n-1 \\ \boldsymbol{\varepsilon}_t^+ &\sim N(0, \boldsymbol{Q}_t) \quad t = 1, \dots, n \end{aligned}$$
3. Simulate observations from the system using the simulated disturbances,
$$\begin{aligned} \boldsymbol{y}_t^+ &= \boldsymbol{d}_t + \boldsymbol{Z}_t \boldsymbol{\alpha}_t + \boldsymbol{\varepsilon}_t^+, \\ \boldsymbol{\alpha}_{t+1} &= \boldsymbol{c}_t + \boldsymbol{T}_t \boldsymbol{\alpha}_t + \boldsymbol{R}_t \boldsymbol{\eta}_t^+, \end{aligned}$$
where $\boldsymbol{\alpha}_1 \sim N(\boldsymbol{a}_1, \boldsymbol{P}_1)$.
4. Run a filter and disturbance smoother on the simulated observations $\boldsymbol{y}^+$ to calculate $\hat{\boldsymbol{\varepsilon}}_t^+ = \mathrm{E}(\boldsymbol{\varepsilon}_t | \boldsymbol{y}_{1:n}^+)$ and $\hat{\boldsymbol{\eta}}_t^+ = \mathrm{E}(\boldsymbol{\eta}_t | \boldsymbol{y}_{1:n}^+)$.
5. A sample from $p(\hat{\boldsymbol{\eta}}_{1:(n-1)}, \hat{\boldsymbol{\varepsilon}}_{1:n} | \boldsymbol{y}_{1:n})$ is
$$\begin{aligned} \tilde{\boldsymbol{\eta}}_t &= \boldsymbol{\eta}_t^+ - \hat{\boldsymbol{\eta}}_t^+ + \hat{\boldsymbol{\eta}}_t, \\ \tilde{\boldsymbol{\varepsilon}}_t &= \boldsymbol{\varepsilon}_t^+ - \hat{\boldsymbol{\varepsilon}}_t^+ + \hat{\boldsymbol{\varepsilon}}_t. \end{aligned}$$

3.3.1.2 States

1. Run a filter and disturbance smoother to calculate the mean of the states conditional on the full series of observations, $\hat{\boldsymbol{\alpha}}_{1:n} = \mathrm{E}(\boldsymbol{\alpha}_{1:n} | \boldsymbol{y}_{1:n})$.
2. Draw samples from the unconditional distribution of the disturbances,
$$\begin{aligned} \boldsymbol{\eta}_t^+ &\sim N(0, \boldsymbol{H}_t) \quad t = 1, \dots, n-1 \\ \boldsymbol{\varepsilon}_t^+ &\sim N(0, \boldsymbol{Q}_t) \quad t = 1, \dots, n \end{aligned}$$
3. Simulate states and observations from the system using the simulated disturbances,
$$\begin{aligned} \boldsymbol{y}_t^+ &= \boldsymbol{d}_t + \boldsymbol{Z}_t \boldsymbol{\alpha}_t + \boldsymbol{\varepsilon}_t^+, \\ \boldsymbol{\alpha}_{t+1}^+ &= \boldsymbol{c}_t + \boldsymbol{T}_t \boldsymbol{\alpha}_t + \boldsymbol{R}_t \boldsymbol{\eta}_t^+, \end{aligned}$$
where $\boldsymbol{\alpha}_1^+ \sim N(\boldsymbol{a}_1, \boldsymbol{P}_1)$.
4. Run a filter and smoother on the simulated observations $\boldsymbol{y}^+$ to calculate $\hat{\boldsymbol{\alpha}}_t^+ = \mathrm{E}(\boldsymbol{\alpha}_t | \boldsymbol{y}_{1:n}^+)$.
5. A sample from $p(\hat{\boldsymbol{\alpha}}_{1:n} | \boldsymbol{y}_{1:n})$ is
$$\tilde{\boldsymbol{\alpha}}_t = \boldsymbol{\alpha}_t^+ - \hat{\boldsymbol{\alpha}}_t^+ + \hat{\boldsymbol{\alpha}}_t.$$

One convenient feature of this method is that since only the conditional means of the states are required, the fast state smoother can be used, since the variances of the states are not required.

3.3.2 de Jong-Shephard method

These recursions were developed in De Jong and Shephard [7] . Although the the mean-correction simulation smoother will work in most cases, there are a few in which it will not work.

TODO

3.3.3 Forward-filter backwards-smoother (FFBS)

This was the simulation method developed in Carter and Kohn [5] and Frühwirth-Schnatter [12] .

TODO

3.4 Missing observations

When all observations at time t are missing, the filtering recursions become [11, Sec 4.10],

$$\begin{aligned} \boldsymbol{a}_{t|t} &= \boldsymbol{a}_t, \\ \boldsymbol{P}_{t|t} &= \boldsymbol{P}_t, \\ \boldsymbol{a}_{t+1} &= \boldsymbol{T}_t \boldsymbol{a}_t + \boldsymbol{c}_t \\ \boldsymbol{P}_{t+1} &= \boldsymbol{T}_t \boldsymbol{P}_t \boldsymbol{T}'_t + \boldsymbol{R}_t \boldsymbol{Q}_t \boldsymbol{R}'_t \end{aligned}$$

This is equivalent to setting $\boldsymbol{Z}_t = \boldsymbol{0}$ (implying also that $\boldsymbol{K}_t = \boldsymbol{0}$) in the filtering equations. For smoothing, also replace $\boldsymbol{Z}_t = \boldsymbol{0}$,

$$\begin{aligned} \boldsymbol{r}_{t-1} &= \boldsymbol{T}'_t \boldsymbol{r}_t, \\ \boldsymbol{N}_{t-1} &= \boldsymbol{T}'_t \boldsymbol{N}_t \boldsymbol{T}_t, \end{aligned}$$

When some, but not all observations are missing, replace the observation equation by,

$$\boldsymbol{y}^*_t = \boldsymbol{Z}^*_t \boldsymbol{\alpha}_t + \boldsymbol{\varepsilon}^*_t, \quad \boldsymbol{\varepsilon}^*_t \sim N(\boldsymbol{0}, \boldsymbol{H}^*_t),$$

where,

$$\begin{aligned} \boldsymbol{y}^*_t &= \boldsymbol{W}_t \boldsymbol{y}_t, \\ \boldsymbol{Z}^* &= \boldsymbol{W}_t \boldsymbol{Z}_t, \\ \boldsymbol{\varepsilon}_t &= \boldsymbol{W}_t \boldsymbol{\varepsilon}_t, \\ \boldsymbol{H}^*_t &= \boldsymbol{W}_t \boldsymbol{H}_t \boldsymbol{W}'_t, \end{aligned}$$

and $\boldsymbol{W}_t$ is a selection matrix to select non-missing values. In smoothing the missing elements are estimated by the appropriate elements of $\boldsymbol{Z}_t \hat{\boldsymbol{alpha}}_t$, where $\hat{\boldsymbol{\alpha}}_t$ is the smoothed state.

Note If $y_{t,j}$ is missing, then setting the relevant entries in the forecast precision matrix, $F^{-1}_{t,j,.} = \boldsymbol{0}$ and $F^{-1}_{t,.,j} = \boldsymbol{0}$, and Kalman gain matrix, $K_{t,.,j} = \boldsymbol{0}$, will handle missing values in the smoothers without having to pass that information to the smoother. However, it may be computationally more efficient if the values of the locations of the missing observations are known.

Note For the disturbance and state simulation smoothers, I think the missing observations need to be indicated and used when doing the simulations on the state smoother.

3.5 Forecasting matrices

Forecasting future observations are the same as treating the future observations as missing [11, Sec 4.11],

$$\begin{aligned} \bar{\boldsymbol{y}}_{n+j} &= \boldsymbol{Z}_{n+j} \bar{\boldsymbol{a}}_{n+j} \\ \bar{\boldsymbol{F}}_{n+j} &= \boldsymbol{Z}_{n+j} \bar{\boldsymbol{P}}_{n+j} \boldsymbol{Z}'_{n+j} + \boldsymbol{H}_{n+j}. \end{aligned}$$

Chapter 4

Stan Functions

State space functionality for Stan is provided as a set of user-defined functions.

Add the following line to the Stan model file in which depends on these functions.

```
functions {
  #include ssm.stan
  // other functions ...
}
```

To actually include the functions in the model, you need to use the function `stanc_builder`, instead of `stan` or `stanc`:

```
model <- stanc_builder("yourmodel.stan", isystem = "path/to/ssm/")
stan(model_code = model$model_code)
```

4.1 Utility Functions

4.1.1 to_symmetric_matrix

Parameters:

- `x` **An** $n \times n$ matrix.

Return Value: `An` $n \times n$ symmetric matrix: $0.5(x + x')$.

Ensure a matrix is symmetrix.

```
matrix to_symmetric_matrix(matrix x) {
  return 0.5 * (x + x ');
}
```

4.1.2 to_matrix_colwise

Parameters:

- `vector` **v** An $n \times m$ vector.
- `int` **m** Number of rows in the vector
- `int` **n** Number of columns in the vector

Return Value: `matrix` A $m \times n$ matrix containting the elements from `v`

Convert vector to a matrix (column-major).

```
matrix to_matrix_colwise(vector v, int m, int n) {
  matrix[m, n] res;
  for (j in 1:n) {
    for (i in 1:m) {
      res[i, j] = v[(j - 1) * m + m];
    }
  }
  return res;
}
```

4.1.3 to_matrix_rowwise

Parameters:

- `vector` **v** An $n \times m$ vector.
- `int` **m** Number of rows in the matrix.
- `int` **n** Number of columns in the matrix.

Return Value: `matrix` A $m \times n$ matrix containting the elements from `v`

Convert vector to a matrix (row-major).

```
matrix to_matrix_rowwise(vector v, int m, int n) {
  matrix[m, n] res;
  for (i in 1:n) {
    for (j in 1:m) {
      res[i, j] = v[(i - 1) * n + n];
    }
  }
  return res;
}
```

4.1.4 to_vector_colwise

Parameters:

- `matrix` **x** An $n \times m$ matrix.

Return Value: `vector` with nm elements.

Convert a matrix to a vector (column-major)

```
vector to_vector_colwise(matrix x) {
  vector[num_elements(x)] res;
  int n;
  int m;
  n = rows(x);
  m = cols(x);
  for (i in 1:n) {
    for (j in 1:m) {
      res[n * (j - 1) + i] = x[i, j];
    }
  }
  return res;
}
```

4.1.5 to_vector_rowwise

Parameters:

- `matrix` **x** An $n \times m$ matrix.

Return Value: `vector` with nm elements.

Convert a matrix to a vector (row-major)

```
vector to_vector_rowwise(matrix x) {
  vector[num_elements(x)] res;
  int n;
  int m;
  n = rows(x);
  m = cols(x);
  for (i in 1:rows(x)) {
    for (j in 1:cols(x)) {
      res[(i - 1) * m + j] = x[i, j];
    }
  }
  return res;
}
```

4.1.6 symmat_size

Parameters:

- `matrix` **x** An $m \times m$ matrix.

Return Value: `int` The number of unique elements

Calculate the number of unique elements in a symmetric matrix

The number of unique elements in an $m \times m$ matrix is $(m \times (m + 1))/2$.

```
int symmat_size(int n) {
  int sz;
```

```
  // This calculates it iteratively because Stan gives a warning
  // with integer division.
  sz = 0;
  for (i in 1:n) {
    sz = sz + i;
  }
  return sz;
}
```

4.1.7 find_symmat_dim

Parameters:

- `int` **n** The number of unique elements in a symmetric matrix.

Return Value: `int` The dimension of the associated symmetric matrix.

Given vector with n elements containing the $m(m+1)/2$ elements of a symmetric matrix, return m.

```
int find_symmat_dim(int n) {
  // This could be solved by finding the positive root of $m = m (m + 1)/2 but
  // Stan doesn't support all the functions necessary to do this.
  int i;
  int remainder;
  i = 0;
  while (n > 0) {
    i = i + 1;
    remainder = remainder - i;
  }
  return i;
}
```

4.1.8 vector_to_symmat

Parameters:

- `vector` **x** The vector with the unique elements
- `int` **n** The dimensions of the returned matrix: $n \times n$.

Return Value: `matrix` An $n \times n$ symmetric matrix.

Convert a vector to a symmetric matrix

```
matrix vector_to_symmat(vector x, int n) {
  matrix[n, n] m;
  int k;
  k = 1;
  for (j in 1:n) {
    for (i in 1:j) {
      m[i, j] = x[k];
      if (i != j) {
        m[j, i] = m[i, j];
```

```
      }
      k = k + 1;
    }
  }
  return m;
}
```

4.1.9 symmat_to_vector

Parameters:

- `vector` **x** An $n \times n$ matrix.

Return Value: `vector` A $n(n+1)/2$ vector with the unique elements in x.

Convert an $n \times n$ symmetric matrix to a length $n(n+1)/2$ vector containing its unique elements.

```
vector symmat_to_vector(matrix x) {
  vector[symmat_size(rows(x))] v;
  int k;
  k = 1;
  // if x is m x n symmetric, then this will return
  // only parts of an m x m matrix.
  for (j in 1:rows(x)) {
    for (i in 1:j) {
      v[k] = x[i, j];
      k = k + 1;
    }
  }
  return v;
}
```

4.2 Filtering

Functions used in filtering and log-likelihood calculations.

4.2.1 ssm_filter_update_a

Parameters:

- `vector` **a** An $m \times 1$ vector with the prected state, $\boldsymbol{a}_t$.
- `vector` **c** An $m \times 1$ vector with the system intercept, $\boldsymbol{c}_t$
- `matrix` **T** An $m \times m$ matrix with the transition matrix, $\boldsymbol{T}_t$.
- `vector` **v** A $p \times 1$ vector with the forecast error, $\boldsymbol{v}_t$.
- `matrix` **K** An $m \times p$ matrix with the Kalman gain, $\boldsymbol{K}_t$.

Return Value: `vector` A $m \times 1$ vector with the predicted state at $t+1$, $\boldsymbol{a}_{t+1}$.

Update the expected value of the predicted state, $\boldsymbol{a}_{t+1} = \mathrm{E}(\boldsymbol{\alpha}_{t+1} | \boldsymbol{y}_{1:t})$,

The predicted state $\boldsymbol{a}_{t+1}$ is,

$$\boldsymbol{a}_{t+1} = \boldsymbol{T}_t \boldsymbol{a}_t + \boldsymbol{K}_t \boldsymbol{v}_t + \boldsymbol{c}_t.$$

```
vector ssm_filter_update_a(vector a, vector c, matrix T, vector v, matrix K) {
  vector[num_elements(a)] a_new;
  a_new = T * a + K * v + c;
  return a_new;
}
```

4.2.2 ssm_filter_update_P

Parameters:

- `matrix` **P** An $m \times m$ vector with the variance of the prected state, $\boldsymbol{P}_t$.
- `matrix` **Z** A $p \times m$ matrix with the design matrix, $\boldsymbol{Z}_t$.
- `matrix` **T** An $m \times m$ matrix with the transition matrix, $\boldsymbol{T}_t$.
- `matrix` **RQR** A $m \times m$ matrix with the system covariance matrix, $\boldsymbol{R}_t\boldsymbol{Q}_t\boldsymbol{R}_t'$.
- `matrix` **K** An $m \times p$ matrix with the Kalman gain, $\boldsymbol{K}_t$.

Return Value: `matrix` An $m \times 1$ vector with the predicted state at $t+1$, $\boldsymbol{a}_{t+1}$.

Update the expected value of the predicted state, $\boldsymbol{P}_{t+1} = \mathrm{Var}(\alpha_{t+1}|\boldsymbol{y}_{1:t})$,

The predicted state variance $\boldsymbol{P}_{t+1}$ is,

$$\boldsymbol{P}_{t+1} = \boldsymbol{T}_t\boldsymbol{P}_t(\boldsymbol{T}_t - \boldsymbol{K}_t\boldsymbol{Z}_t)' + \boldsymbol{R}_t\boldsymbol{Q}_t\boldsymbol{R}_t'.$$

```
matrix ssm_filter_update_P(matrix P, matrix Z, matrix T,
                           matrix RQR, matrix K) {
  matrix[rows(P), cols(P)] P_new;
  P_new = to_symmetric_matrix(T * P * (T - K * Z)' + RQR);
  return P_new;
}
```

4.2.3 ssm_filter_update_v

Parameters:

- `matrix` **P** An $m \times m$ vector with the variance of the prected state, $\boldsymbol{P}_t$.
- `matrix` **Z** A $p \times m$ matrix with the design matrix, $\boldsymbol{Z}_t$.
- `matrix` **T** An $m \times m$ matrix with the transition matrix, $\boldsymbol{T}_t$.
- `matrix` **RQR** An $m \times m$ matrix with the system covariance matrix, $\boldsymbol{R}_t\boldsymbol{Q}_t\boldsymbol{R}_t'$.
- `matrix` **K** An $m \times p$ matrix with the Kalman gain, $\boldsymbol{K}_t$.

Return Value: `vector` An $m \times 1$ vector with the predicted state at $t+1$, $\boldsymbol{a}_{t+1}$.

Update the forcast error, $\boldsymbol{v}_t = \boldsymbol{y}_t - \mathrm{E}(\boldsymbol{y}_t|\boldsymbol{y}_{\boldsymbol{1:(t-1)}})$

The forecast error $\boldsymbol{v}_t$ is

$$\boldsymbol{v}_t = \boldsymbol{y}_t - \boldsymbol{Z}_t\boldsymbol{a}_t - \boldsymbol{d}_t.$$

```
vector ssm_filter_update_v(vector y, vector a, vector d, matrix Z) {
  vector[num_elements(y)] v;
  v = y - Z * a - d;
  return v;
}
```

4.2.4 ssm_filter_update_F

Parameters:

- `matrix` **P** An $m \times m$ vector with the variance of the prected state, $\boldsymbol{P}_t$.
- `matrix` **Z** A $p \times m$ matrix with the design matrix, $\boldsymbol{Z}_t$.
- `matrix` **H** A $p \times p$ matrix with the observation covariance matrix, $\boldsymbol{H}_t$.

Return Value: `matrix` A $p \times p$ vector with $\boldsymbol{F}_t$.

Update the variance of the forcast error, $\boldsymbol{F}_t = \mathrm{Var}(\boldsymbol{y}_t - \mathrm{E}(\boldsymbol{y}_t | \boldsymbol{y}_{\mathbf{1}:(\boldsymbol{t}-\mathbf{1})}))$

The variance of the forecast error $\boldsymbol{F}_t$ is

$$\boldsymbol{F}_t = \boldsymbol{Z}_t \boldsymbol{P}_t \boldsymbol{Z}_t + \boldsymbol{H}_t.$$

```
matrix ssm_filter_update_F(matrix P, matrix Z, matrix H) {
  matrix[rows(H), cols(H)] F;
  F = quad_form(P, Z') + H;
  return F;
}
```

4.2.5 ssm_filter_update_Finv

Parameters:

- `matrix` **P** An $m \times m$ vector with the variance of the prected state, $\boldsymbol{P}_t$.
- `matrix` **Z** A $p \times m$ matrix with the design matrix, $\boldsymbol{Z}_t$.
- `matrix` **H** A $p \times p$ matrix with the observation covariance matrix, $\boldsymbol{H}_t$.

Return Value: `matrix` A $p \times p$ vector with $\boldsymbol{F}_t^{-1}$.

Update the precision of the forcast error, $\boldsymbol{F}_t^{-1} = \mathrm{Var}(\boldsymbol{y}_t - \mathrm{E}(\boldsymbol{y}_t | \boldsymbol{y}_{\mathbf{1}:(\boldsymbol{t}-\mathbf{1})}))^{-1}$

This is the inverse of $\boldsymbol{F}_t$.

```
matrix ssm_filter_update_Finv(matrix P, matrix Z, matrix H) {
  matrix[rows(H), cols(H)] Finv;
  Finv = inverse(ssm_filter_update_F(P, Z, H));
  return Finv;
}
```

4.2.6 ssm_filter_update_K

Parameters:

- `matrix` **P** An $m \times m$ vector with the variance of the prected state, P_t.
- `matrix` **Z** A $p \times m$ matrix with the design matrix, $\boldsymbol{Z}_t$.
- `matrix` **T** An $m \times m$ matrix with the transition matrix, $\boldsymbol{T}_t$.
- `matrix` **Finv** A $p \times p$ matrix

Return Value: `matrix` An $m \times p$ matrix with the Kalman gain, $\boldsymbol{K}_t$.

Update the Kalman gain, $\boldsymbol{K}_t$.

The Kalman gain is

$$\boldsymbol{K}_t = \boldsymbol{T}_t \boldsymbol{P}_t \boldsymbol{Z}_t' \boldsymbol{F}_t^{-1}.$$

```
matrix ssm_filter_update_K(matrix P, matrix Z, matrix T, matrix Finv) {
  matrix[cols(Z), rows(Z)] K;
  K = T * P * Z' * Finv;
  return K;
}
```

4.2.7 ssm_filter_update_L

Parameters:

- `matrix` **Z** A $p \times m$ matrix with the design matrix, $\boldsymbol{Z}_t$
- `matrix` **T** An $m \times m$ matrix with the transition matrix, $\boldsymbol{T}_t$.
- `matrix` **K** An $m \times p$ matrix with the Kalman gain, $\boldsymbol{K}_t$.

Return Value: `matrix` An $m \times m$ matrix, $\boldsymbol{L}_t$.

Update L_t

$$\boldsymbol{L}_t = \boldsymbol{T}_t - \boldsymbol{K}_t \boldsymbol{Z}_t.$$

```
matrix ssm_filter_update_L(matrix Z, matrix T, matrix K) {
  matrix[rows(T), cols(T)] L;
  L = T - K * Z;
  return L;
}
```

4.2.8 ssm_filter_update_ll

Parameters:

- `vector` **v** A $p \times 1$ matrix with the forecast error, $\boldsymbol{v}_t$.
- `matrix` **Finv** A $p \times p$ matrix with variance of the forecast error, $\boldsymbol{F}_t^{-1}$.

Return Value: `real` An $m \times m$ matrix, L_t.

Calculate the log-likelihood for a period

The log-likehood of a single observation in a state-space model is

$$\ell_t = -\frac{1}{2} p \log(2\pi) - \frac{1}{2} \left(\log |\boldsymbol{F}_t| + \boldsymbol{v}_t' \boldsymbol{F}_t^{-1} \boldsymbol{v}_t \right)$$

```
real ssm_filter_update_ll(vector v, matrix Finv) {
  real ll;
  int p;
  p = num_elements(v);
```

```
  // det(A^{-1}) = 1 / det(A) -> log det(A^{-1}) = - log det(A)
  ll = (- 0.5 *
        (p * log(2 * pi())
         - log_determinant(Finv)
         + quad_form(Finv, v)
       ));
  return ll;
}
```

4.3 Filtering

4.3.1 ssm_filter_idx

Parameters:

- `int` **m** The number of states
- `int` **p** The size of the observation vector $\boldsymbol{y}_t$.

Return Value: `int[,]` A 6×3 integer array containing the indexes of the return values of the Kalman filter.

Indexes of the return values of the Kalman filter functions: `ssm_filter`.

`ssm_filter_idx` returns a 6×3 integer array with the (length, start index, stop index) of (ℓ_t, $\boldsymbol{v}$, $\boldsymbol{F}^{-}1$, $\boldsymbol{K}$, $\boldsymbol{a}$, $\boldsymbol{P}$).

value	length	start	stop
ℓ_t	1	1	1
$\boldsymbol{v}$	p	2	$1 + p$
$\boldsymbol{F}^{-1}$	$p(p+1)/2$	$2 + p$	$1 + p + p(p+1)/2$
$\boldsymbol{K}$	mp	$2 + p + p(p+1)/2$	$1 + p + p(p+1)/2 + mp$
$\boldsymbol{a}_t$	m	$2 + p + p(p+1)/2 + mp$	$1 + p + p(p+1)/2 + mp + m$
$\boldsymbol{P}^t$	$m(m+1)/2$	$2 + p + p(p+1)/2 + mp + m$	$1 + p + p(p+1)/2 + mp + m(m+1)/2$

```
int[,] ssm_filter_idx(int m, int p) {
  int sz[6, 3];
  // loglike
  sz[1, 1] = 1;
  // v
  sz[2, 1] = p;
  // Finv
  sz[3, 1] = symmat_size(p);
  // K
  sz[4, 1] = m * p;
  // a
  sz[5, 1] = m;
  // P
  sz[6, 1] = symmat_size(m);
  // Fill in start and stop points
  sz[1, 2] = 1;
  sz[1, 3] = sz[1, 2] + sz[1, 1] - 1;
```

```
  for (i in 2:6) {
    sz[i, 2] = sz[i - 1, 3] + 1;
    sz[i, 3] = sz[i, 2] + sz[i, 1] - 1;
  }
  return sz;
}
```

4.3.2 ssm_filter_size

Parameters:

- `int` **m** The number of states
- `int` **p** The size of the observation vector $\boldsymbol{y}_t$.

Return Value: `int` The number of elements in the vector.

Number of elements in vector containing filter results

```
int ssm_filter_size(int m, int p) {
  int sz;
  int idx[6, 3];
  idx = ssm_filter_idx(m, p);
  sz = idx[6, 3];
  return sz;
}
```

4.3.3 ssm_filter_get_loglik

Parameters:

- `vector` **A** vector with results from `ssm_filter`.
- `int` **m** The number of states
- `int` **p** The size of the observation vector $\boldsymbol{y}_t$.

Return Value: `real` The log-likelihood ℓ_t

Get the log-likehood from the results of `ssm_filter`.

```
real ssm_filter_get_loglik(vector x, int m, int p) {
  real y;
  y = x[1];
  return y;
}
```

4.3.4 ssm_filter_get_v

Parameters:

- `vector` **A** vector with results from `ssm_filter`.
- `int` **m** The number of states
- `int` **p** The size of the observation vector $\boldsymbol{y}_t$.

Return Value: `vector` A $p \times 1$ vector with the forecast error, $\boldsymbol{v}_t$.

Get the forecast error from the results of `ssm_filter`.

```
vector ssm_filter_get_v(vector x, int m, int p) {
  vector[p] y;
  int idx[6, 3];
  idx = ssm_filter_idx(m, p);
  y = segment(x, idx[2, 2], idx[2, 3]);
  return y;
}
```

4.3.5 ssm_filter_get_Finv

Parameters:

- `vector` **A** vector with results from `ssm_filter`.
- `int` **m** The number of states
- `int` **p** The size of the observation vector $\boldsymbol{y}_t$.

Return Value: `matrix` A $p \times p$ matrix with the forecast precision, $\boldsymbol{F}_t^{-1}$.

Get the forecast precision from the results of `ssm_filter`.

```
matrix ssm_filter_get_Finv(vector x, int m, int p) {
  matrix[p, p] y;
  int idx[6, 3];
  idx = ssm_filter_idx(m, p);
  y = vector_to_symmat(segment(x, idx[3, 2], idx[3, 3]), p);
  return y;
}
```

4.3.6 ssm_filter_get_K

Parameters:

- `vector` **A** vector with results from `ssm_filter`.
- `int` **m** The number of states
- `int` **p** The size of the observation vector $\boldsymbol{y}_t$.

Return Value: `matrix` A $m \times p$ matrix with the Kalman gain, $\boldsymbol{F}_t^{-1}$.

Get the Kalman gain from the results of `ssm_filter`.

```
matrix ssm_filter_get_K(vector x, int m, int p) {
  matrix[m, p] y;
  int idx[6, 3];
  idx = ssm_filter_idx(m, p);
  y = to_matrix_colwise(segment(x, idx[4, 2], idx[4, 3]), m, p);
  return y;
}
```

4.3.7 ssm_filter_get_a

Parameters:

- `vector` **A** vector with results from `ssm_filter`.
- `int` **m** The number of states
- `int` **p** The size of the observation vector $\boldsymbol{y}_t$.

Return Value: `vector` An $m \times 1$ vector with the expected value of the predicted state, $\mathrm{E}(\boldsymbol{\alpha}_t | \boldsymbol{y}_{1:(t-1)}) = \boldsymbol{a}_t$.

Get the expected value of the predicted state from the results of `ssm_filter`.

```
vector ssm_filter_get_a(vector x, int m, int p) {
  vector[m] y;
  int idx[6, 3];
  idx = ssm_filter_idx(m, p);
  y = segment(x, idx[5, 2], idx[5, 3]);
  return y;
}
```

4.3.8 ssm_filter_get_P

Parameters:

- `vector` **A** vector with results from `ssm_filter`.
- `int` **m** The number of states
- `int` **p** The size of the observation vector $\boldsymbol{y}_t$.

Return Value: `matrix` An $m \times m$ matrix with the variance of the predicted state, $\mathrm{Var}(\boldsymbol{\alpha}_t | \boldsymbol{y}_{1:(t-1)}) = \boldsymbol{P}_t$.

Get the variance of the predicted state from the results of `ssm_filter`.

```
matrix ssm_filter_get_P(vector x, int m, int p) {
  matrix[m, m] y;
  int idx[6, 3];
  idx = ssm_filter_idx(m, p);
  y = vector_to_symmat(segment(x, idx[6, 2], idx[6, 3]), m);
  return y;
}
```

4.3.9 ssm_filter

Parameters:

- `vector[]` **y** Observations, $\boldsymbol{y}_t$. An array of size n of $p \times 1$ vectors.
- `vector[]` **d** Observation intercept, $\boldsymbol{d}_t$. An array of $p \times 1$ vectors.
- `matrix[]` **Z** Design matrix, $\boldsymbol{Z}_t$. An array of $p \times m$ matrices.
- `matrix[]` **H** Observation covariance matrix, $\boldsymbol{H}_t$. An array of $p \times p$ matrices.
- `vector[]` **c** State intercept, $\boldsymbol{c}_t$. An array of $m \times 1$ vectors.
- `matrix[]` **T** Transition matrix, $\boldsymbol{T}_t$. An array of $m \times m$ matrices.
- `matrix[]` **R** State covariance selection matrix, $\boldsymbol{R}_t$. An array of $p \times q$ matrices.

- `matrix[] Q` **Q** State covariance matrix, $\boldsymbol{Q}_t$. An array of $q \times q$ matrices.
- `vector` **a1** Expected value of the intial state, $a_1 = \mathrm{E}(\alpha_1)$. An $m \times 1$ matrix.
- `matrix` **P1** Variance of the initial state, $P_1 = \mathrm{Var}(\alpha_1)$. An $m \times m$ matrix.

Return Value: `vector[]` Array of size n of $(1 + p + p(p + 1)/2 + mp + m + m(m + 1)/2) \times 1$ vectors in the format described in `ssm_filter_idx`.

Kalman filter

For `d`, `Z`, `H`, `c`, `T`, `R`, `Q` the array can have a size of 1, if it is not time-varying, or a size of n (for `d`, `Z`, `H`) or $n - 1$ (for `c`, `T`, `R`, `Q`) if it is time varying.

`ssm_filter` runs a forward filter on the state space model and calculates,

- log-likelihood for each observation, ℓ_t.
- Forecast error, $\boldsymbol{v}_t = \boldsymbol{y}_t - \mathrm{E}(\boldsymbol{y}_t | \boldsymbol{y}_{1:(t-1)})$.
- Forecast precision, $\boldsymbol{F}_t^{-1}$.
- Kalman gain, $\boldsymbol{K}_t$.
- Predicted states, $\boldsymbol{a}_t = \mathrm{E}(\boldsymbol{\alpha}_t | \boldsymbol{y}_{1:(t-1)})$.
- Variance of the predicted states, $\boldsymbol{P}_t = \mathrm{Var}(\boldsymbol{\alpha}_t | \boldsymbol{y}_{1:(t-1)})$.

The results of Kalman filter for a given are returned as a $1 + p + p(p + 1)/2 + mp + m(m + 1)/2$ vector for each time period, where

$$(\ell_t, \boldsymbol{v}_t', \mathrm{vec}(\boldsymbol{F}_t^{-1})', \mathrm{vec}(\boldsymbol{K}_t)', \boldsymbol{a}_t', \mathrm{vec}(\boldsymbol{P}_t)')'.$$

```
vector[] ssm_filter(vector[] y,
                    vector[] d, matrix[] Z, matrix[] H,
                    vector[] c, matrix[] T, matrix[] R, matrix[] Q,
                    vector a1, matrix P1) {

  // returned data
  vector[ssm_filter_size(dims(Z)[3], dims(Z)[2])] res[size(y)];
  int q;
  int n;
  int p;
  int m;

  // sizes
  n = size(y); // number of obs
  p = dims(Z)[2]; // obs size
  m = dims(Z)[3]; // number of states
  q = dims(Q)[2]; // number of state disturbances

  //print("Sizes: n = ", m, ", p = ", n, ", m = ", m, ", q = ", q);
  {
    // system matrices for current iteration
    vector[p] d_t;
    matrix[p, m] Z_t;
    matrix[p, p] H_t;
    vector[m] c_t;
    matrix[m, m] T_t;
    matrix[m, q] R_t;
    matrix[q, q] Q_t;
    matrix[m, m] RQR;
```

```
// result matricees for each iteration
vector[m] a;
matrix[m, m] P;
vector[p] v;
matrix[p, p] Finv;
matrix[m, p] K;
real ll;
int idx[6, 3];

idx = ssm_filter_idx(m, p);

d_t = d[1];
Z_t = Z[1];
H_t = H[1];
c_t = c[1];
T_t = T[1];
R_t = R[1];
Q_t = Q[1];
RQR = quad_form(Q_t, R_t);
a = a1;
P = P1;
for (t in 1:n) {
  if (t > 1) {
    if (size(d) > 1) {
      d_t = d[t];
    }
    if (size(Z) > 1) {
      Z_t = Z[t];
    }
    if (size(H) > 1) {
      H_t = H[t];
    }
    if (size(c) > 1) {
      c_t = c[t];
    }
    if (size(T) > 1) {
      T_t = T[t];
    }
    if (size(R) > 1) {
      R_t = R[t];
    }
    if (size(Q) > 1) {
      Q_t = Q[t];
    }
    if (size(R) > 1 && size(Q) > 1) {
      RQR = quad_form(Q_t, R_t);
    }
  }
  // updating
  v = ssm_filter_update_v(y[t], a, d_t, Z_t);
  Finv = ssm_filter_update_Finv(P, Z_t, H_t);
  K = ssm_filter_update_K(P, T_t, Z_t, Finv);
  ll = ssm_filter_update_ll(v, Finv);
  // saving
```

```
      res[t, 1] = ll;
      res[t, idx[2, 2]:idx[2, 3]] = v;
      res[t, idx[3, 2]:idx[3, 3]] = symmat_to_vector(Finv);
      res[t, idx[4, 2]:idx[4, 3]] = to_vector(K);
      res[t, idx[5, 2]:idx[5, 3]] = a;
      res[t, idx[6, 2]:idx[6, 3]] = symmat_to_vector(P);
      // predict a_{t + 1}, P_{t + 1}
      if (t < n) {
        a = ssm_filter_update_a(a, c_t, T_t, v, K);
        P = ssm_filter_update_P(P, Z_t, T_t, RQR, K);
      }
    }
  }
  return res;
}
```

4.3.10 ssm_filter_states

Parameters:

- `int` **m** Number of states

Return Value: `int` The size of the vector

Length of the vectors returned by `ssm_filter_states`

```
int ssm_filter_states_size(int m) {
  int sz;
  sz = m + symmat_size(m);
  return sz;
}
```

4.3.11 ssm_filter_states_get_a

Parameters:

- `vector` **x** A vector returned by `ssm_filter_states`
- `int` **m** Number of states

Return Value: `matrix` An $m \times 1$ vector with the filtered expected value of the state, $\boldsymbol{a}_{t|t} = \mathrm{E}(\boldsymbol{\alpha}_t | \boldsymbol{y}_{1:t})$.

Extract $a_{t|t}$ from the results of `ssm_filter_states`

```
vector ssm_filter_states_get_a(vector x, int m) {
  vector[m] a;
  a = x[ :m];
  return a;
}
```

4.3.12 ssm_filter_states_get_P

Parameters:

- `vector` **x** A vector returned by `ssm_filter_states`
- `int` **m** Number of states

Return Value: `matrix` An $m \times m$ matrix with the filtered variance of the state, $\boldsymbol{P}_{t|t} = \mathrm{Var}(\boldsymbol{\alpha}_t | \boldsymbol{y}_{1:t})$.

Extract $P_{t|t}$ from the results of `ssm_filter_states`

```
matrix ssm_filter_states_get_P(vector x, int m) {
  matrix[m, m] P;
  P = vector_to_symmat(x[(m + 1): ], m);
  return P;
}
```

4.3.13 ssm_filter_states

Parameters:

- `vector[]` **filter** Results from `ssm_filter`
- `matrix[]` **Z** Design matrix, $\boldsymbol{Z}_t$. An array of $p \times m$ matrices.

Return Value: `Array` of size n of vectors.

Calculate filtered expected values and variances of the states

The filtering function `ssm_filter` returns the mean and variance of the predicted states, $\boldsymbol{a}_t = \mathrm{E}(\boldsymbol{\alpha}_t | \boldsymbol{y}_{1:(t-1)})$ and $\boldsymbol{P}_t = \mathrm{Var}(\boldsymbol{\alpha}_t | \boldsymbol{y}_{1:(t-1)})$.

The vectors returned by `ssm_filter_states` are of length $m + m^2$, with

$$\boldsymbol{v}_t = (\boldsymbol{a}'_{t|t}, \mathrm{vec}(\boldsymbol{P}_{t|t})')'$$

Use the functions `ssm_filter_states_get_a` and `ssm_filter_states_get_P` to extract elements from the results.

For `Z` the array can have a size of 1, if it is not time-varying, or a size of $n - 1$ if it is time varying.

```
vector[] ssm_filter_states(vector[] filter, matrix[] Z) {
  vector[ssm_filter_states_size(dims(Z)[3])] res[size(filter)];
  int n;
  int m;
  int p;
  n = size(filter);
  m = dims(Z)[3];
  p = dims(Z)[2];
  {
    // system matrices for current iteration
    matrix[p, m] Z_t;
    // filter matrices
    vector[m] aa; // filtered values of the state, a_{t|t}
    matrix[m, m] PP; // filtered values of the variance of the state, P_{t|t}
```

```
    vector[p] v;
    matrix[p, p] Finv;
    vector[m] a;
    matrix[m, m] P;

    Z_t = Z[1];
    for (t in 1:n) {
      if (t > 1) {
        if (size(Z) > 1) {
          Z_t = Z[t];
        }
      }
      // extract values from the filter
      v = ssm_filter_get_v(filter[t], m, p);
      Finv = ssm_filter_get_Finv(filter[t], m, p);
      a = ssm_filter_get_a(filter[t], m, p);
      P = ssm_filter_get_P(filter[t], m, p);
      // calcualte filtered values
      aa = a + P * Z_t ' * Finv * v;
      PP = to_symmetric_matrix(P - P * quad_form(Finv, Z_t) * P);
      // saving
      res[t, :m] = aa;
      res[t, (m + 1): ] = symmat_to_vector(PP);
    }
  }
  return res;
}
```

4.4 Log-likelihood

4.4.1 ssm_lpdf

Parameters:

- `vector[]` **y** Observations, $\boldsymbol{y}_t$. An array of size n of $p \times 1$ vectors.
- `vector[]` **d** Observation intercept, $\boldsymbol{d}_t$. An array of $p \times 1$ vectors.
- `matrix[]` **Z** Design matrix, $\boldsymbol{Z}_t$. An array of $p \times m$ matrices.
- `matrix[]` **H** Observation covariance matrix, $\boldsymbol{H}_t$. An array of $p \times p$ matrices.
- `vector[]` **c** State intercept, $\boldsymbol{c}_t$. An array of $m \times 1$ vectors.
- `matrix[]` **T** Transition matrix, $\boldsymbol{T}_t$. An array of $m \times m$ matrices.
- `matrix[]` **R** State covariance selection matrix, $\boldsymbol{R}_t$. An array of $p \times q$ matrices.
- `matrix[]` **Q** State covariance matrix, $\boldsymbol{Q}_t$. An array of $q \times q$ matrices.
- `vector` **a1** Expected value of the intial state, $a_1 = \mathrm{E}(\alpha_1)$. An $m \times 1$ matrix.
- `matrix` **P1** Variance of the initial state, $P_1 = \mathrm{Var}(\alpha_1)$. An $m \times m$ matrix.

Return Value: `real` The log-likelihood, $p(\boldsymbol{y}_{1:n}|\boldsymbol{d}, \boldsymbol{Z}, \boldsymbol{H}, \boldsymbol{c}, \boldsymbol{T}, \boldsymbol{R}, \boldsymbol{Q})$, marginalized over the latent states.

Log-likelihood of a Linear Gaussian State Space Model

For `d`, `Z`, `H`, `c`, `T`, `R`, `Q` the array can have a size of 1, if it is not time-varying, or a size of n (for `d`, `Z`, `H`) or $n - 1$ (for `c`, `T`, `R`, `Q`) if it is time varying.

The log-likelihood of a linear Gaussian state space model is, If the the system matrices and initial conditions are known, the log likelihood is

$$
\begin{aligned}
\log L(\boldsymbol{Y}_n) = \log p(\boldsymbol{y}_1, \dots, \boldsymbol{y}_n) &= \sum_{t=1}^{n} \log p(\boldsymbol{y}_t | \boldsymbol{Y}_{t-1}) \quad , \\
&= -\frac{np}{2} \log 2\pi - \frac{1}{2} \sum_{t=1}^{n} \left(\log |\boldsymbol{F}_t| + \boldsymbol{v}' \boldsymbol{F}_t^{-1} \boldsymbol{v}_t \right)
\end{aligned}
$$

where $\boldsymbol{F}_t$ and $\boldsymbol{V}_t$ come from a forward pass of the Kalman filter.

```
real ssm_lpdf(vector[] y,
               vector[] d, matrix[] Z, matrix[] H,
               vector[] c, matrix[] T, matrix[] R, matrix[] Q,
               vector a1, matrix P1) {
  real ll;
  int n;
  int m;
  int p;
  int q;
  n = size(y); // number of obs
  m = dims(Z)[2];
  p = dims(Z)[3];
  q = dims(Q)[2];
  {
    // system matrices for current iteration
    vector[p] d_t;
    matrix[p, m] Z_t;
    matrix[p, p] H_t;
    vector[m] c_t;
    matrix[m, m] T_t;
    matrix[m, q] R_t;
    matrix[q, q] Q_t;
    matrix[m, m] RQR;
    // result matricees for each iteration
    vector[n] ll_obs;
    vector[m] a;
    matrix[m, m] P;
    vector[p] v;
    matrix[p, p] Finv;
    matrix[m, p] K;

    d_t = d[1];
    Z_t = Z[1];
    H_t = H[1];
    c_t = c[1];
    T_t = T[1];
    R_t = R[1];
    Q_t = Q[1];
    RQR = quad_form(Q_t, R_t);

    a = a1;
    P = P1;
    for (t in 1:n) {
```

```
      if (t > 1) {
        if (size(d) > 1) {
          d_t = d[t];
        }
        if (size(Z) > 1) {
          Z_t = Z[t];
        }
        if (size(H) > 1) {
          H_t = H[t];
        }
        if (size(c) > 1) {
          c_t = c[t];
        }
        if (size(T) > 1) {
          T_t = T[t];
        }
        if (size(R) > 1) {
          R_t = R[t];
        }
        if (size(Q) > 1) {
          Q_t = Q[t];
        }
        if (size(R) > 1 && size(Q) > 1) {
          RQR = quad_form(Q_t, R_t);
        }
      }
      v = ssm_filter_update_v(y[t], a, d_t, Z_t);
      Finv = ssm_filter_update_Finv(P, Z_t, H_t);
      K = ssm_filter_update_K(P, Z_t, T_t, Finv);
      ll_obs[t] = ssm_filter_update_ll(v, Finv);
      // don't save a, P for last iteration
      if (t < n) {
        a = ssm_filter_update_a(a, c_t, T_t, v, K);
        P = ssm_filter_update_P(P, Z_t, T_t, RQR, K);
      }
    }
    ll = sum(ll_obs);
  }
  return ll;
}
```

4.5 Time-Invariant Kalman Filter

4.5.1 ssm_check_matrix_equal

Parameters:

- `matrix` **A** An $m \times n$ matrix.
- `matrix` **B** An $m \times n$ matrix.
- `real` **The** relative tolerance for convergence.

Return Value: `int` If converged, then 1, else 0.

Check if two matrices are approximately equal

The matrices A and B are considered approximately equal if

$$\max(A - B)/\max(A) < \epsilon,$$

where ϵ is the tolerance.

```
int ssm_check_matrix_equal(matrix A, matrix B, real tol) {
  real eps;
  eps = max(to_vector(A - B)) / max(to_vector(A));
  if (eps < tol) {
    return 1;
  } else {
    return 0;
  }
}
```

4.5.2 ssm_constant_lpdf

Parameters:

- `vector[]` **y** Observations, $\boldsymbol{y}_t$. An array of size n of $p \times 1$ vectors.
- `vector` **d** Observation intercept, $\boldsymbol{d}_t$. An array of $p \times 1$ vectors.
- `matrix` **Z** Design matrix, $\boldsymbol{Z}_t$. An array of $p \times m$ matrices.
- `matrix` **H** Observation covariance matrix, $\boldsymbol{H}_t$. An array of $p \times p$ matrices.
- `vector` **c** State intercept, $\boldsymbol{c}_t$. An array of $m \times 1$ vectors.
- `matrix` **T** Transition matrix, $\boldsymbol{T}_t$. An array of $m \times m$ matrices.
- `matrix` **R** State covariance selection matrix, $\boldsymbol{R}_t$. An array of $p \times q$ matrices.
- `matrix` **Q** State covariance matrix, $\boldsymbol{Q}_t$. An array of $q \times q$ matrices.
- `vector` **a1** Expected value of the intial state, $a_1 = \mathrm{E}(\alpha_1)$. An $m \times 1$ matrix.
- `matrix` **P1** Variance of the initial state, $P_1 = \mathrm{Var}(\alpha_1)$. An $m \times m$ matrix.

Return Value: `real` The log-likelihood, $p(\boldsymbol{y}_{1:n} | \boldsymbol{d}, \boldsymbol{Z}, \boldsymbol{H}, \boldsymbol{c}, \boldsymbol{T}, \boldsymbol{R}, \boldsymbol{Q})$, marginalized over the latent states.

Log-likelihood of a Time-Invariant Linear Gaussian State Space Model

Unlike `ssm_filter`, this function requires the system matrices (`d`, `Z`, `H`, `c`, `T`, `R`, `Q`) to all be time invariant (constant). When the state space model is time-invariant, then the Kalman recursion for $\boldsymbol{P}_t$ converges. This function takes advantage of this feature and stops updating $\boldsymbol{P}_t$ after it converges to a steady state.

```
real ssm_constant_lpdf(vector[] y,
                      vector d, matrix Z, matrix H,
                      vector c, matrix T, matrix R, matrix Q,
                      vector a1, matrix P1) {
  real ll;
  int n;
  int m;
  int p;

  n = size(y); // number of obs
  m = cols(Z);
  p = rows(Z);
  {
```

```
    vector[n] ll_obs;
    vector[m] a;
    matrix[m, m] P;
    vector[p] v;
    matrix[p, p] Finv;
    matrix[m, p] K;
    matrix[m, m] RQR;
    // indicator for if the filter has converged
    // This only works for time-invariant state space models
    int converged;
    matrix[m, m] P_old;
    real tol;
    converged = 0;
    tol = 1e-7;

    RQR = quad_form(Q, R);
    a = a1;
    P = P1;
    for (t in 1:n) {
      v = ssm_filter_update_v(y[t], a, d, Z);
      if (converged < 1) {
        Finv = ssm_filter_update_Finv(P, Z, H);
        K = ssm_filter_update_K(P, Z, T, Finv);
      }
      ll_obs[t] = ssm_filter_update_ll(v, Finv);
      // don't save a, P for last iteration
      if (t < n) {
        a = ssm_filter_update_a(a, c, T, v, K);
        // check for convergence
        // should only check for convergence if there are no missing values
        if (converged < 1) {
          P_old = P;
          P = ssm_filter_update_P(P, Z, T, RQR, K);
          converged = ssm_check_matrix_equal(P, P_old, tol);
        }
      }
    }
    ll = sum(ll_obs);
  }
  return ll;
}
```

4.6 Common Smoother Functions

4.6.1 ssm_smooth_update_r

Parameters:

- `vector` **r** An $m \times 1$ vector with $\boldsymbol{r}_{t-1}$
- `matrix` **Z** A $p \times m$ vector with $\boldsymbol{Z}_t$

- `vector` **v** A $p \times 1$ vector of the forecast errors, $\boldsymbol{v}_t$.
- `matrix` **Finv** A $p \times p$ matrix of the forecast precision, $\boldsymbol{F}_t^{-1}$.
- `matrix` **L** An $m \times m$ matrix with $\boldsymbol{L}_t$.

Return Value: `matrix` An $m \times 1$ vector with $\boldsymbol{r}_t$.

Update $\boldsymbol{r}_t$ in smoothing recursions

In smoothing recursions, the vector $\boldsymbol{r}_t$ is updated with,

$$\boldsymbol{r}_{t-1} = \boldsymbol{Z}' \boldsymbol{F}_t^{-1} \boldsymbol{v}_t + \boldsymbol{L}' \boldsymbol{r}_t.$$

See [11, p. 91]

```
vector ssm_smooth_update_r(vector r, matrix Z, vector v, matrix Finv,
                           matrix L) {
  vector[num_elements(r)] r_new;
  r_new = Z ' * Finv * v + L ' * r;
  return r_new;
}
```

4.6.2 ssm_smooth_update_N

Parameters:

- `vector` **N** An $m \times 1$ vector with $\boldsymbol{N}_{t-1}$
- `matrix` **Z** A $p \times m$ vector with $\boldsymbol{Z}_t$
- `matrix` **Finv** A $p \times p$ matrix of the forecast precision, $\boldsymbol{F}_t^{-1}$.
- `matrix` **L** An $m \times m$ matrix with $\boldsymbol{L}_t$.

Return Value: `matrix` An $m \times m$ matrix with $\boldsymbol{N}_t$.

Update $\boldsymbol{N}_t$ in smoothing recursions

In smoothing recursions, the matrix $\boldsymbol{N}_t$ is updated with,

$$\boldsymbol{N}_{t-1} = \boldsymbol{Z}_t' \boldsymbol{F}_t^{-1} \boldsymbol{Z}_t + \boldsymbol{L}_t' \boldsymbol{N}_t \boldsymbol{L}_t.$$

See [11, p. 91]

```
matrix ssm_smooth_update_N(matrix N, matrix Z, matrix Finv, matrix L) {
  matrix[rows(N), cols(N)] N_new;
  N_new = quad_form(Finv, Z) + quad_form(N, L);
  return N_new;
}
```

4.6.3 ssm_smooth_state_size

Parameters:

- `int` **m** The number of states.

Return Value: `int` The size of the vectors is $m + m(m + 1)/2$.

The number of elements in vectors returned by `ssm_smooth_state`

```
int ssm_smooth_state_size(int m) {
  int sz;
  sz = m + symmat_size(m);
  return sz;
}
```

4.6.4 ssm_smooth_state_get_mean

Parameters:

- `vector` **x** A vector returned by `ssm_smooth_state`
- `int` **q** The number of state disturbances, $\boldsymbol{\eta}_t$.

Return Value: `vector` An $m \times 1$ vector with $\hat{\boldsymbol{\eta}}_t$.

Extract $\hat{\boldsymbol{\alpha}}_t$ from vectors returned by `ssm_smooth_state`

```
vector ssm_smooth_state_get_mean(vector x, int m) {
  vector[m] alpha;
  alpha = x[ :m];
  return alpha;
}
```

4.6.5 ssm_smooth_state_get_var

Parameters:

- `vector` **x** A vector returned by `ssm_smooth_state`
- `int` **m** The number of states

Return Value: `matrix` An $m \times m$ matrix with $\boldsymbol{V}_t$.

Extract $matV_t$ from vectors returned by `ssm_smooth_state`

```
matrix ssm_smooth_state_get_var(vector x, int m) {
  matrix[m, m] V;
  V = vector_to_symmat(x[(m + 1): ], m);
  return V;
}
```

4.6.6 ssm_smooth_state

Parameters:

- `vector[]` **filter** Results of `ssm_filter`
- `matrix[]` **Z** Design matrix, $\boldsymbol{Z}_t$. An array of $p \times m$ matrices.

- `matrix[] T` Transition matrix, $\boldsymbol{T}_t$. An array of $m \times m$ matrices.

Return Value: `vector[]` An array of vectors constaining $\hat{\boldsymbol{\alpha}}_t$ and $\boldsymbol{V}_t = \mathrm{Var}(\boldsymbol{\alpha}_t | \boldsymbol{y}_{1:n})$.

The state smoother

This calculates the mean and variance of the states, $\boldsymbol{\alpha}_t$, given the entire sequence, $\boldsymbol{y}_{1:n}$.

in the format described below.

For `Z` and `T` the array can have a size of 1, if it is not time-varying, or a size of n (for `Z`) or $n-1$ (for `T`) if it is time varying.

The vectors returned by this function have $m + m^2$ elements in this format,

$$(\hat{\boldsymbol{\alpha}}_t', \mathrm{vec}(\boldsymbol{V}_t)')'.$$

Use the `ssm_smooth_state_get_mean` and `ssm_smooth_state_get_var` to extract components from the returned vectors.

value	length	start	end
$\hat{\boldsymbol{\alpha}}_t$	m	1	m
$\boldsymbol{V}_t$	$m(m+1)/2$	$m+1$	$m+m(m+1)/2$

See Durbin and Koopman [11], Eq 4.44 and eq 4.69.

```
vector[] ssm_smooth_state(vector[] filter, matrix[] Z, matrix[] T) {
  vector[ssm_smooth_state_size(dims(Z)[3])] res[size(filter)];
  int n;
  int m;
  int p;
  n = size(filter);
  m = dims(Z)[3];
  p = dims(Z)[2];
  {
    // system matrices for current iteration
    matrix[p, m] Z_t;
    matrix[m, m] T_t;
    // smoother results
    vector[m] r;
    matrix[m, m] N;
    matrix[m, m] L;
    vector[m] alpha;
    matrix[m, m] V;
    // filter results
    vector[p] v;
    matrix[m, p] K;
    matrix[p, p] Finv;
    vector[m] a;
    matrix[m, m] P;

    if (size(Z) == 1) {
      Z_t = Z[1];
    }
    if (size(T) == 1) {
```

```
      T_t = T[1];
    }
    // initialize smoother
    // r and N go from n, n - 1, ..., 1, 0.
    // r_n and N_n
    r = rep_vector(0.0, m);
    N = rep_matrix(0.0, m, m);
    // move backwards in time: t, ..., 1
    for (i in 0:(n - 1)) {
      int t;
      t = n - i;
      // set time-varying system matrices
      if (size(Z) > 1) {
        Z_t = Z[t];
      }
      if (size(T) > 1) {
        T_t = T[t];
      }
      // get filtered values
      K = ssm_filter_get_K(filter[t], m, p);
      v = ssm_filter_get_v(filter[t], m, p);
      Finv = ssm_filter_get_Finv(filter[t], m, p);
      a = ssm_filter_get_a(filter[t], m, p);
      P = ssm_filter_get_P(filter[t], m, p);
      // updating
      // L_t
      L = ssm_filter_update_L(Z_t, T_t, K);
      // r_{t - 1} and N_{t - 1}
      r = ssm_smooth_update_r(r, Z_t, v, Finv, L);
      N = ssm_smooth_update_N(N, Z_t, Finv, L);
      // hat(alpha)_{t} and V_t which use r and N from (t - 1)
      alpha = a + P * r;
      V = to_symmetric_matrix(P - P * N * P);
      // saving
      res[t, :m] = alpha;
      res[t, (m + 1): ] = symmat_to_vector(V);
    }
  }
  return res;
}
```

4.6.7 ssm_smooth_eps_size

Parameters:

- `int` **p** The length of the observation vectors, $\boldsymbol{y}_t$.

Return Value: `int` The size of the vectors is $p + p(p+1)/2$.

The size of the vectors returned by `ssm_smooth_eps`

```
int ssm_smooth_eps_size(int p) {
```

```
  int sz;
  sz = p + symmat_size(p);
  return sz;
}
```

4.6.8 ssm_smooth_eps_get_mean

Parameters:

- `x` **A** vector from the results of `ssm_smooth_eps`.
- `int` **p** The length of the observation vectors, $\boldsymbol{y}_t$.

Return Value: `vector` A $p \times 1$ vector with $\hat{\boldsymbol{\varepsilon}}_t$.

Extract $\hat{\boldsymbol{\varepsilon}}_t$ from vectors returned by `ssm_smooth_eps`

```
vector ssm_smooth_eps_get_mean(vector x, int p) {
  vector[p] eps;
  eps = x[ :p];
  return eps;
}
```

4.6.9 ssm_smooth_eps_get_var

Parameters:

- `vector` **x** A vector returned by `ssm_smooth_eps`
- `int` **p** The length of the observation vectors, $\boldsymbol{y}_t$.

Return Value: `matrix` A $p \times p$ matrix with $\mathrm{Var}(\boldsymbol{\varepsilon}_t | \boldsymbol{y}_{1:n})$

Extract $\mathrm{Var}(\varepsilon_t | \boldsymbol{y}_{1:n})$ from vectors returned by `ssm_smooth_eps`

```
matrix ssm_smooth_eps_get_var(vector x, int p) {
  matrix[p, p] eps_var;
  eps_var = vector_to_symmat(x[(p + 1): ], p);
  return eps_var;
}
```

4.6.10 ssm_smooth_eps

Parameters:

- `vector[]` **filter** Results of `ssm_filter`
- `matrix[]` **Z** Design matrix, $\boldsymbol{Z}_t$. An array of $p \times m$ matrices.
- `matrix[]` **H** Observation covariance matrix, $\boldsymbol{H}_t$. An array of $p \times p$ matrices.
- `matrix[]` **T** Transition matrix, $\boldsymbol{T}_t$. An array of $m \times m$ matrices.

Return Value: `vector[]` An array of vectors constaining $\hat{\varepsilon}_t$ and $\mathrm{Var}(\boldsymbol{\varepsilon}_t|\boldsymbol{y}_{1:n})$ in the format described below.

The observation disturbance smoother

This calculates the mean and variance of the observation disturbances, $\boldsymbol{\varepsilon}_t$, given the entire sequence, $\boldsymbol{y}_{1:n}$.

For Z,H, `T`, the array can have a size of 1, if it is not time-varying, or a size of n (for `Z`, `H`) or $n-1$ (for `T`), if it is time varying.

The vectors returned by this function have $p + p(p+1)/2$ elements in this format,

$$(\hat{\varepsilon}_t', \mathrm{vec}(\mathrm{Var}(\boldsymbol{\varepsilon}_t|\boldsymbol{y}_{1:n}))')'$$

value	length	start	end
$\hat{\varepsilon}_t$	p	1	p
$\mathrm{Var}(\boldsymbol{\varepsilon}_t\|\boldsymbol{y}_{1:n})$	$p(p+1)/2$	$p+1$	$p+p(p+1)/2$

See [11, Sec 4.5.3 (eq 4.69)]

```
vector[] ssm_smooth_eps(vector[] filter, matrix[] Z, matrix[] H, matrix[] T) {
  vector[ssm_smooth_eps_size(dims(Z)[2])] res[size(filter)];
  int n;
  int m;
  int p;
  n = size(filter);
  m = dims(Z)[3];
  p = dims(Z)[2];
  {
    // smoother values
    vector[m] r;
    matrix[m, m] N;
    matrix[m, m] L;
    vector[p] eps;
    matrix[p, p] var_eps;
    // filter results
    vector[p] v;
    matrix[m, p] K;
    matrix[p, p] Finv;
    // system matrices
    matrix[p, m] Z_t;
    matrix[p, p] H_t;
    matrix[m, m] T_t;

    // set matrices if time-invariant
    if (size(Z) == 1) {
      Z_t = Z[1];
    }
    if (size(H) == 1) {
      H_t = H[1];
    }
    if (size(T) == 1) {
      T_t = T[1];
    }
```

```
    // initialize smoother
    // r and N go from n, n - 1, ..., 1, 0.
    // r_n and N_n
    r = rep_vector(0.0, m);
    N = rep_matrix(0.0, m, m);
    for (i in 1:n) {
      int t;
      // move backwards in time
      t = n - i + 1;
      // update time-varying system matrices
      if (size(Z) > 1) {
        Z_t = Z[t];
      }
      if (size(H) > 1) {
        H_t = H[t];
      }
      if (size(T) > 1) {
        T_t = T[t];
      }
      // get values from filter
      K = ssm_filter_get_K(filter[t], m, p);
      v = ssm_filter_get_v(filter[t], m, p);
      Finv = ssm_filter_get_Finv(filter[t], m, p);
      // updating
      L = ssm_filter_update_L(Z_t, T_t, K);
      // r_{t - 1} and N_{t - 1}
      r = ssm_smooth_update_r(r, Z_t, v, Finv, L);
      N = ssm_smooth_update_N(N, Z_t, Finv, L);
      // eps_t and V(eps_t|y)
      eps = H_t * (Finv * v - K ' * r);
      var_eps = to_symmetric_matrix(H_t - H_t * (Finv + quad_form(N, K)) * H_t);
      // saving
      res[t, :p] = eps;
      res[t, (p + 1): ] = symmat_to_vector(var_eps);
    }
  }
  return res;
}
```

4.6.11 ssm_smooth_eta

Parameters:

- `int` **p** The length of the observation vectors, $\boldsymbol{y}_t$.

Return Value: `int` The size of the vectors is $q + q(q + 1)/2$.

The size of the vectors returned by `ssm_smooth_eta`

```
int ssm_smooth_eta_size(int q) {
  int sz;
  sz = q + symmat_size(q);
  return sz;
}
```

4.6.12 ssm_smooth_eta_get_mean

Parameters:

- `vector` **x** A vector returned by `ssm_smooth_eta`
- `int` **q** The number of state disturbances, $\boldsymbol{\eta}_t$.

Return Value: `vector` A $q \times 1$ vector with $\hat{\boldsymbol{\eta}}_t$.

Extract $\hat{\varepsilon}_t$ from vectors returned by `ssm_smooth_eta`

```
vector ssm_smooth_eta_get_mean(vector x, int q) {
  vector[q] eta;
  eta = x[ :q];
  return eta;
}
```

4.6.13 ssm_smooth_eta_get_var

Parameters:

- `vector` **x** A vector returned by `ssm_smooth_eta`
- `int` **q** The number of state disturbances, $\boldsymbol{\eta}_t$.

Return Value: `matrix` A $q \times q$ matrix with $\mathrm{Var}(\boldsymbol{\eta}_t | \boldsymbol{y}_{1:n})$.

Extract $\mathrm{Var}(\eta_t | \boldsymbol{y}_{1:n})$ from vectors returned by `ssm_smooth_eta`

```
matrix ssm_smooth_eta_get_var(vector x, int q) {
  matrix[q, q] eta_var;
  eta_var = vector_to_symmat(x[(q + 1): ], q);
  return eta_var;
}
```

4.6.14 ssm_smooth_eta

Parameters:

- `vector[]` **filter** Results of `ssm_filter`
- `matrix[]` **Z** Design matrix, $\boldsymbol{Z}_t$. An array of $p \times m$ matrices.
- `matrix[]` **T** Transition matrix, $\boldsymbol{T}_t$. An array of $m \times m$ matrices.
- `matrix[]` **R** State covariance selection matrix, $\boldsymbol{R}_t$. An array of $p \times q$ matrices.
- `matrix[]` **Q** State covariance matrix, $\boldsymbol{Q}_t$. An array of $q \times q$ matrices.

Return Value: `vector[]` An array of vectors constaining $\hat{\boldsymbol{\eta}}_t$ and $\mathrm{Var}(\boldsymbol{\eta}_t | \boldsymbol{y}_{1:n})$ in the format described below.

The state disturbance smoother

This calculates the mean and variance of the observation disturbances, $\boldsymbol{\eta}_t$, given the entire sequence, $\boldsymbol{y}_{1:n}$.

For `Z`, `T`, `R`, `Q` the array can have a size of 1, if it is not time-varying, or a size of n (for `Z`) or $n - 1$ (for `T`, `R`, `Q`) if it is time varying.

The vectors returned by this function have $q + q(q+1)/2$ elements in this format,

$$(\hat{\boldsymbol{\eta}}_t', \operatorname{vec}(\operatorname{Var}(\boldsymbol{\eta}_t | \boldsymbol{y}_{1:n}))').$$

Use the `ssm_smooth_eta_get_mean` and `ssm_smooth_eta_get_var` to extract components from the returned vectors.

value	length	start	end
$\hat{\boldsymbol{\eta}}_t$	q	1	q
$\operatorname{Var}(\boldsymbol{\eta}_t \vert \boldsymbol{y}_{1:n})$	$q(q+1)/2$	$q+1$	$q+q(q+1)/2$

See [11, Sec 4.5.3 (eq 4.69)]

```
vector[] ssm_smooth_eta(vector[] filter,
                        matrix[] Z, matrix[] T,
                        matrix[] R, matrix[] Q) {
  vector[ssm_smooth_eta_size(dims(Q)[2])] res[size(filter)];
  int n;
  int m;
  int p;
  int q;
  n = size(filter);
  m = dims(Z)[3];
  p = dims(Z)[2];
  q = dims(Q)[2];
  {
    // smoother matrices
    vector[m] r;
    matrix[m, m] N;
    matrix[m, m] L;
    vector[q] eta;
    matrix[q, q] var_eta;
    // system matrices
    matrix[p, m] Z_t;
    matrix[m, m] T_t;
    matrix[m, q] R_t;
    matrix[q, q] Q_t;
    // filter matrices
    vector[p] v;
    matrix[m, p] K;
    matrix[p, p] Finv;

    // set time-invariant matrices
    if (size(Z) == 1) {
      Z_t = Z[1];
    }
    if (size(T) == 1) {
      T_t = T[1];
    }
    if (size(R) == 1) {
      R_t = R[1];
    }
    if (size(Q) == 1) {
```

```
      Q_t = Q[1];
    }
    // initialize smoother
    r = rep_vector(0.0, m);
    N = rep_matrix(0.0, m, m);
    for (i in 0:(n - 1)) {
      int t;
      // move backwards in time
      t = n - i;
      // update time-varying system matrices
      if (size(Z) > 1) {
        Z_t = Z[t];
      }
      if (size(T) > 1) {
        T_t = T[t];
      }
      if (size(R) > 1) {
        R_t = R[t];
      }
      if (size(Q) > 1) {
        Q_t = Q[t];
      }
      // get values from filter
      K = ssm_filter_get_K(filter[t], m, p);
      v = ssm_filter_get_v(filter[t], m, p);
      Finv = ssm_filter_get_Finv(filter[t], m, p);
      // update smoother
      L = ssm_filter_update_L(Z_t, T_t, K);
      r = ssm_smooth_update_r(r, Z_t, v, Finv, L);
      N = ssm_smooth_update_N(N, Z_t, Finv, L);
      eta = Q_t * R_t ' * r;
      var_eta = to_symmetric_matrix(Q_t - Q_t * quad_form(N, R_t) * Q_t);
      // saving
      res[t, :q] = eta;
      res[t, (q + 1): ] = symmat_to_vector(var_eta);
    }
  }
  return res;
}
```

4.6.15 ssm_smooth_faststate

Parameters:

- `vector[]` **filter** The results of `ssm_filter`
- `matrix[]` **Z** Design matrix, $\boldsymbol{Z}_t$. An array of $p \times m$ matrices.
- `vector[]` **c** State intercept, $\boldsymbol{c}_t$. An array of $m \times 1$ vectors.
- `matrix[]` **T** Transition matrix, $\boldsymbol{T}_t$. An array of $m \times m$ matrices.
- `matrix[]` **R** State covariance selection matrix, $\boldsymbol{R}_t$. An array of $p \times q$ matrices.
- `matrix[]` **Q** State covariance matrix, $\boldsymbol{Q}_t$. An array of $q \times q$ matrices.

Return Value: `vector[]` An array of size n of $m \times 1$ vectors containing $\hat{\boldsymbol{\alpha}}_t$.

The fast state smoother

The fast state smoother calculates $\hat{\boldsymbol{\alpha}}_t = \mathrm{E}(\boldsymbol{\alpha}_t | \boldsymbol{y}_{1:n})$.

$$\hat{\boldsymbol{\alpha}}_{t+1} = \boldsymbol{T}_t \hat{\boldsymbol{\alpha}}_t + \boldsymbol{R}_t \boldsymbol{Q}_t \boldsymbol{R}'_t \boldsymbol{r}_t,$$

where r_t is calcualted from the state disturbance smoother. The smoother is initialized at $t = 1$ with $\hat{\boldsymbol{\alpha}}_t = \boldsymbol{a}_1 + \boldsymbol{P}_1 \boldsymbol{r}_0$.

Unlike the normal state smoother, it does not calculate the variances of the smoothed state.

For `Z`, `c`, `T`, `R`, `Q` the array can have a size of 1, if it is not time-varying, or a size of n (for `Z`) or $n - 1$ (for `c`, `T`, `R`, `Q`) if it is time varying.

See [11, Sec 4.5.3 (eq 4.69)]

```
vector[] ssm_smooth_faststate(vector[] filter,
                              vector[] c, matrix[] Z, matrix[] T,
                              matrix[] R, matrix[] Q) {
  vector[dims(Z)[3]] alpha[size(filter)];
  int n;
  int m;
  int p;
  int q;
  n = size(filter);
  m = dims(Z)[3];
  p = dims(Z)[2];
  q = dims(Q)[2];
  {
    // smoother matrices
    vector[m] r[n + 1];
    matrix[m, m] L;
    vector[m] a1;
    matrix[m, m] P1;
    // filter matrices
    vector[p] v;
    matrix[m, p] K;
    matrix[p, p] Finv;
    // system matrices
    matrix[p, m] Z_t;
    vector[m] c_t;
    matrix[m, m] T_t;
    matrix[p, q] R_t;
    matrix[q, q] Q_t;
    matrix[m, m] RQR;
    // set time-invariant matrices
    if (size(c) == 1) {
      c_t = c[1];
    }
    if (size(Z) == 1) {
      Z_t = Z[1];
    }
    if (size(T) == 1) {
      T_t = T[1];
    }
    if (size(R) == 1) {
      R_t = R[1];
    }
```

```
if (size(Q) == 1) {
  Q_t = Q[1];
}
if (size(Q) == 1 && size(R) == 1) {
  RQR = quad_form(Q[1], R[1]');
}
// find smoothed state disturbances
// Since I don't need to calculate the
// variances of the smoothed disturbances,
// I reimplement the state distrurbance smoother here
// removing extraneous parts.
// r goes from t = n, ..., 1, 0.
// r_n
r[n + 1] = rep_vector(0.0, m);
for (i in 0:(n - 1)) {
  int t;
  // move backwards in time
  t = n - i;
  // update time varying system matrices
  if (size(Z) > 1) {
    Z_t = Z[t];
  }
  if (size(T) > 1) {
    T_t = T[t];
  }
  // get filter values
  K = ssm_filter_get_K(filter[t], m, p);
  v = ssm_filter_get_v(filter[t], m, p);
  Finv = ssm_filter_get_Finv(filter[t], m, p);
  // updating smoother
  L = ssm_filter_update_L(Z_t, T_t, K);
  // r_{t - 1}
  r[t] = ssm_smooth_update_r(r[t + 1], Z_t, v, Finv, L);
}
// calculate smoothed states
a1 = ssm_filter_get_a(filter[1], m, p);
P1 = ssm_filter_get_P(filter[1], m, p);
// r[1] = r_0
alpha[1] = a1 + P1 * r[1];
// 1:(n - 1) -> \alpha_{2}:\alpha_{n}
for (t in 1:(n - 1)) {
  if (size(c) > 1) {
    c_t = c[t];
  }
  if (size(T) > 1) {
    T_t = T[t];
  }
  if (size(Q) > 1) {
    Q_t = Q[t];
  }
  if (size(R) > 1) {
    R_t = R[t];
  }
  if (size(Q) > 1 || size(R) > 1) {
```

```
        RQR = quad_form(Q_t, R_t');
      }
      // `r[t + 1]` = $r_{t}$
      // alpha_{t + 1} = c_t + T_t * \alpha_t + R_t Q_t R'_t r_t
      alpha[t + 1] = c_t + T_t * alpha[t] + RQR * r[t + 1];
    }
  }
  return alpha;
}
```

4.7 Simulators and Smoothing Simulators

4.7.1 ssm_sim_idx

Parameters:

- `int` **m** The number of states
- `int` **p** The length of the observation vector
- `int` **q** The number of state disturbances

Return Value: `int[,]` A 4 x 3 array of integers

Indexes of each component of `ssm_sim_rng` results.

The returned array has columns (length, start location, and end location) for rows: $\boldsymbol{y}_t$, $\boldsymbol{\alpha}_t$, $\boldsymbol{\varepsilon}_t$, and $\boldsymbol{\eta}_t$ in the results of `ssm_sim_rng`.

```
int[,] ssm_sim_idx(int m, int p, int q) {
  int sz[4, 3];
  // y
  sz[1, 1] = p;
  // a
  sz[2, 1] = m;
  // eps
  sz[3, 1] = p;
  // eta
  sz[4, 1] = q;
  // Fill in start and stop points
  sz[1, 2] = 1;
  sz[1, 3] = sz[1, 2] + sz[1, 1] - 1;
  for (i in 2:4) {
    sz[i, 2] = sz[i - 1, 3] + 1;
    sz[i, 3] = sz[i, 2] + sz[i, 1] - 1;
  }
  return sz;
}
```

4.7.2 ssm_sim_size

Parameters:

- `int` **m** The number of states
- `int` **p** The length of the observation vector
- `int` **q** The number of state disturbances

Return Value: `int` The number of elements

The number of elements in vectors returned by `ssm_sim_rng` results.

```
int ssm_sim_size(int m, int p, int q) {
  int sz;
  sz = ssm_sim_idx(m, p, q)[4, 3];
  return sz;
}
```

4.7.3 ssm_sim_get_y

Parameters:

- `int` **m** The number of states
- `int` **p** The length of the observation vector
- `int` **q** The number of state disturbances

Return Value: `vector` vector A $p \times 1$ vector with $\boldsymbol{y}_t$.

Extract $\boldsymbol{y}_t$ from vectors returned by `ssm_sim_rng`.

```
vector ssm_sim_get_y(vector x, int m, int p, int q) {
  vector[m] y;
  int idx[4, 3];
  idx = ssm_sim_idx(m, p, q);
  y = x[idx[1, 2]:idx[1, 3]];
  return y;
}
```

4.7.4 ssm_sim_get_a

Parameters:

- `int` **m** The number of states
- `int` **p** The length of the observation vector
- `int` **q** The number of state disturbances

Return Value: `vector` A $m \times 1$ vector with $\boldsymbol{\alpha}_t$.

Extract $\boldsymbol{\alpha}_t$ from vectors returne by `ssm_sim_rng`.

```
vector ssm_sim_get_a(vector x, int m, int p, int q) {
  vector[m] a;
  int idx[4, 3];
  idx = ssm_sim_idx(m, p, q);
  a = x[idx[2, 2]:idx[2, 3]];
  return a;
}
```

4.7.5 ssm_sim_get_eps

Parameters:

- `int` **m** The number of states
- `int` **p** The length of the observation vector
- `int` **q** The number of state disturbances

Return Value: `vector` vector A $p \times 1$ vector with $\boldsymbol{\varepsilon}_t$.

Extract $\boldsymbol{\varepsilon}_t$ from vectors returne by `ssm_sim_rng`.

```
vector ssm_sim_get_eps(vector x, int m, int p, int q) {
  vector[m] eps;
  int idx[4, 3];
  idx = ssm_sim_idx(m, p, q);
  eps = x[idx[3, 2]:idx[3, 3]];
  return eps;
}
```

4.7.6 ssm_sim_get_eta

Parameters:

- `int` **m** The number of states
- `int` **p** The length of the observation vector
- `int` **q** The number of state disturbances

Return Value: `vector` vector A $q \times 1$ vector with $\boldsymbol{\eta}_t$.

Extract $\boldsymbol{\eta}_t$ from vectors returne by `ssm_sim_rng`.

```
vector ssm_sim_get_eta(vector x, int m, int p, int q) {
  vector[m] eta;
  int idx[4, 3];
  idx = ssm_sim_idx(m, p, q);
  eta = x[idx[4, 2]:idx[4, 3]];
  return eta;
}
```

4.7.7 ssm_sim_rng

Parameters:

- `vector[]` **y** Observations, $\boldsymbol{y}_t$. An array of size n of $p \times 1$ vectors.
- `vector[]` **d** Observation intercept, $\boldsymbol{d}_t$. An array of $p \times 1$ vectors.
- `matrix[]` **Z** Design matrix, $\boldsymbol{Z}_t$. An array of $p \times m$ matrices.
- `matrix[]` **H** Observation covariance matrix, $\boldsymbol{H}_t$. An array of $p \times p$ matrices.
- `vector[]` **c** State intercept, $\boldsymbol{c}_t$. An array of $m \times 1$ vectors.
- `matrix[]` **T** Transition matrix, $\boldsymbol{T}_t$. An array of $m \times m$ matrices.
- `matrix[]` **R** State covariance selection matrix, $\boldsymbol{R}_t$. An array of $p \times q$ matrices.
- `matrix[]` **Q** State covariance matrix, $\boldsymbol{Q}_t$. An array of $q \times q$ matrices.
- `vector` **a1** Expected value of the intial state, $a_1 = \mathrm{E}(\alpha_1)$. An $m \times 1$ matrix.
- `matrix` **P1** Variance of the initial state, $P_1 = \mathrm{Var}(\alpha_1)$. An $m \times m$ matrix.

Return Value: `Array` of size n of vectors with Draw $\boldsymbol{y}_t$, $\boldsymbol{\alpha}_t$, $\boldsymbol{\eta}_t$ and $\boldsymbol{\varepsilon}_t$. See the description.

Simulate from a Linear Gaussian State Space model.

For `d`, `Z`, `H`, `c`, `T`, `R`, `Q` the array can have a size of 1, if it is not time-varying, or a size of n (for `d`, `Z`, `H`) or $n-1$ (for `c`, `T`, `R`, `Q`) if it is time varying.

Draw $\boldsymbol{y}_t$, $\boldsymbol{\alpha}_t$, $\boldsymbol{\eta}_t$ and $\boldsymbol{\varepsilon}_t$ from the state space model,

$$\begin{aligned} \boldsymbol{y}_t &= \boldsymbol{d}_t + \boldsymbol{Z}_t \boldsymbol{\alpha}_t + \boldsymbol{\varepsilon}_t, & \boldsymbol{\varepsilon}_t &\sim N(0, \boldsymbol{H}_t), \\ \boldsymbol{\alpha}_{t+1} &= \boldsymbol{c}_t + \boldsymbol{T}_t \boldsymbol{\alpha}_t + \boldsymbol{R}_t \boldsymbol{\eta}_t, & \boldsymbol{\eta}_t &\sim N(0, \boldsymbol{Q}_t), \\ & & \boldsymbol{\alpha}_1 &\sim N(\boldsymbol{a}_1, \boldsymbol{P}_1). \end{aligned}$$

The returned vectors are of length $2p + m + q$, in the format,

$$(\boldsymbol{y}'_t, \boldsymbol{\alpha}'_t, \boldsymbol{\varepsilon}'_t, \boldsymbol{\eta}'_t).$$

Note that $\eta_n = \mathbf{0}_q$. Use the functions `ssm_sim_get_y`, `ssm_sim_get_a`, `ssm_sim_get_eps`, and `ssm_sim_get_eta` to extract values from the vector.

element	length	start	end
y_t	p	1	p
α_t	m	$p+1$	$p+m$
ε_t	p	$p+m+1$	$2p+m$
η_t	q	$2p+m+1$	$2p+m+q$

It is preferrable to use `ssm_sim_get_y`, `ssm_sim_get_a`, `ssm_sim_get_eps`, and `ssm_sim_get_eta` to extract values from these vectors.

```
vector[] ssm_sim_rng(int n,
                    vector[] d, matrix[] Z, matrix[] H,
                    vector[] c, matrix[] T, matrix[] R, matrix[] Q,
                    vector a1, matrix P1) {
  vector[ssm_sim_size(dims(Z)[3], dims(Z)[2], dims(Q)[2])] ret[n];
  int p;
  int m;
  int q;
  p = dims(Z)[2];
```

```
m = dims(Z)[3];
q = dims(Q)[2];
{
  // system matrices for current iteration
  vector[p] d_t;
  matrix[p, m] Z_t;
  matrix[p, p] H_t;
  vector[m] c_t;
  matrix[m, m] T_t;
  matrix[m, q] R_t;
  matrix[q, q] Q_t;
  matrix[m, m] RQR;
  // outputs
  vector[p] y;
  vector[p] eps;
  vector[m] a;
  vector[q] eta;
  // constants
  vector[p] zero_p;
  vector[q] zero_q;
  vector[m] zero_m;
  int idx[4, 3];

  d_t = d[1];
  Z_t = Z[1];
  H_t = H[1];
  c_t = c[1];
  T_t = T[1];
  R_t = R[1];
  Q_t = Q[1];

  idx = ssm_sim_idx(m, p, q);
  zero_p = rep_vector(0.0, p);
  zero_q = rep_vector(0.0, q);
  zero_m = rep_vector(0.0, m);
  a = multi_normal_rng(a1, P1);
  for (t in 1:n) {
    // set system matrices
    if (t > 1) {
      if (size(d) > 1) {
        d_t = d[t];
      }
      if (size(Z) > 1) {
        Z_t = Z[t];
      }
      if (size(H) > 1) {
        H_t = H[t];
      }
      // system matrices are n - 1 length
      if (t < n) {
        if (size(c) > 1) {
          c_t = c[t];
        }
        if (size(T) > 1) {
```

```
          T_t = T[t];
        }
        if (size(R) > 1) {
          R_t = R[t];
        }
        if (size(Q) > 1) {
          Q_t = Q[t];
        }
      }
    }
    // draw forecast error
    eps = multi_normal_rng(zero_p, H_t);
    // draw observed value
    y = d_t + Z_t * a + eps;
    // since eta_t is for alpha_{t + 1}, we don't
    // draw it for t == n
    if (t == n) {
      eta = zero_q;
    } else {
      eta = multi_normal_rng(zero_q, Q_t);
    }
    // save
    ret[t, idx[1, 2]:idx[1, 3]] = y;
    ret[t, idx[2, 2]:idx[2, 3]] = a;
    ret[t, idx[3, 2]:idx[3, 3]] = eps;
    ret[t, idx[4, 2]:idx[4, 3]] = eta;
    // a_{t + 1}
    if (t < n) {
      a = c_t + T_t * a + R_t * eta;
    }
  }
}
return ret;
}
```

4.8 Simulation Smoothers

4.8.1 ssm_simsmo_state_rng

Parameters:

- `vector[]` **alpha** An of size n of $m \times 1$ vectors containing the smoothed expected values of the states, $\mathrm{E}(\boldsymbol{\alpha}_{1:n} | \boldsymbol{y}_{1:n})$. These are returned by `sim_smooth_faststates`. If `sim_smooth_state` was used, then the expected values need to first be extracted using `sim_smooth_state_get_mean`.
- `vector[]` **d** Observation intercept, $\boldsymbol{d}_t$. An array of $p \times 1$ vectors.
- `matrix[]` **Z** Design matrix, $\boldsymbol{Z}_t$. An array of $p \times m$ matrices.
- `matrix[]` **H** Observation covariance matrix, $\boldsymbol{H}_t$. An array of $p \times p$ matrices.
- `vector[]` **c** State intercept, $\boldsymbol{c}_t$. An array of $m \times 1$ vectors.
- `matrix[]` **T** Transition matrix, $\boldsymbol{T}_t$. An array of $m \times m$ matrices.
- `matrix[]` **R** State covariance selection matrix, $\boldsymbol{R}_t$. An array of $p \times q$ matrices.
- `matrix[]` **Q** State covariance matrix, $\boldsymbol{Q}_t$. An array of $q \times q$ matrices.
- `vector` **a1** Expected value of the intial state, $a_1 = \mathrm{E}(\alpha_1)$. An $m \times 1$ matrix.

- `matrix` **P1** Variance of the initial state, $P_1 = \text{Var}(\alpha_1)$. An $m \times m$ matrix.

Return Value: `vector[]` Array of size n of $m \times 1$ vectors containing a single draw from $(\boldsymbol{\alpha}_{1:n}|\boldsymbol{y}_{1:n})$.

State simulation smoother

Draw samples from the posterior distribution of the states, $\tilde{\boldsymbol{\alpha}}_{1:n} \sim p(\boldsymbol{\alpha}_{1:n}|\boldsymbol{y}_{1:n})$.

For `d`, `Z`, `H`, `c`, `T`, `R`, `Q` the array can have a size of 1, if it is not time-varying, or a size of n (for `d`, `Z`, `H`) or $n-1$ (for `c`, `T`, `R`, `Q`) if it is time varying.

This draws samples using mean-correction simulation smoother of [10]. See [11, Sec 4.9].

```
vector[] ssm_simsmo_states_rng(vector[] alpha,
                      vector[] d, matrix[] Z, matrix[] H,
                      vector[] c, matrix[] T, matrix[] R, matrix[] Q,
                      vector a1, matrix P1) {
    vector[dims(Z)[2]] draws[size(alpha)];
    int n;
    int p;
    int m;
    int q;
    n = size(alpha);
    p = dims(Z)[2];
    m = dims(Z)[3];
    q = dims(Q)[2];
    {
      vector[ssm_filter_size(m, p)] filter[n];
      vector[ssm_sim_size(m, p, q)] sims[n];
      vector[p] y[n];
      vector[m] alpha_hat_plus[n];
      // simulate unconditional disturbances and observations
      sims = ssm_sim_rng(n, d, Z, H, c, T, R, Q, a1, P1);
      for (i in 1:n) {
        y[i] = ssm_sim_get_y(sims[i], m, p, q);
      }
      // filter with simulated y's
      filter = ssm_filter(y, d, Z, H, c, T, R, Q, a1, P1);
      // mean correct epsilon samples
      alpha_hat_plus = ssm_smooth_faststate(filter, c, Z, T, R, Q);
      for (i in 1:n) {
        draws[i] = (ssm_sim_get_a(sims[i], m, p, q)
                    - alpha_hat_plus[i]
                    + alpha[i]);
      }
    }
    return draws;
}
```

4.8.2 ssm_simsmo_eta_rng

Parameters:

- `vector[]` **eta** Values returned by `sim_smooth_eta`
- `vector[]` **d** Observation intercept, $\boldsymbol{d}_t$. An array of $p \times 1$ vectors.

- `matrix[]` **Z** Design matrix, $\boldsymbol{Z}_t$. An array of $p \times m$ matrices.
- `matrix[]` **H** Observation covariance matrix, $\boldsymbol{H}_t$. An array of $p \times p$ matrices.
- `vector[]` **c** State intercept, $\boldsymbol{c}_t$. An array of $m \times 1$ vectors.
- `matrix[]` **T** Transition matrix, $\boldsymbol{T}_t$. An array of $m \times m$ matrices.
- `matrix[]` **R** State covariance selection matrix, $\boldsymbol{R}_t$. An array of $p \times q$ matrices.
- `matrix[]` **Q** State covariance matrix, $\boldsymbol{Q}_t$. An array of $q \times q$ matrices.
- `vector` **a1** Expected value of the intial state, $a_1 = \mathrm{E}(\alpha_1)$. An $m \times 1$ matrix.
- `matrix` **P1** Variance of the initial state, $P_1 = \mathrm{Var}(\alpha_1)$. An $m \times m$ matrix.

Return Value: `vector[]` Array of size n of $q \times 1$ vectors containing a single draw from $(\boldsymbol{\eta}_{1:n} | \boldsymbol{y}_{1:n})$.

State disturbance simulation smoother

Draw samples from the posterior distribution of the observation disturbances, $\tilde{\boldsymbol{\eta}}_{1:n} \sim p(\boldsymbol{\eta}_{1:n} | \boldsymbol{y}_{1:n})$.

For `d`, `Z`, `H`, `c`, `T`, `R`, `Q` the array can have a size of 1, if it is not time-varying, or a size of n (for `d`, `Z`, `H`) or $n-1$ (for `c`, `T`, `R`, `Q`) if it is time varying.

This draws samples using mean-correction simulation smoother of [10]. See [11, Sec 4.9].

```
vector[] ssm_simsmo_eta_rng(vector[] eta,
                            vector[] d, matrix[] Z, matrix[] H,
                            vector[] c, matrix[] T, matrix[] R, matrix[] Q,
                            vector a1, matrix P1) {
    vector[dims(Q)[2]] draws[size(eta)];
    int n;
    int p;
    int m;
    int q;
    n = size(eta);
    p = dims(Z)[2];
    m = dims(Z)[3];
    q = dims(Q)[2];
    {
      vector[ssm_filter_size(m, p)] filter[n];
      vector[p] y[n];
      vector[ssm_sim_size(m, p, q)] sims[n];
      vector[ssm_smooth_eta_size(q)] etahat_plus[n];
      // simulate unconditional disturbances and observations
      sims = ssm_sim_rng(n, d, Z, H, c, T, R, Q, a1, P1);
      for (i in 1:n) {
        y[i] = ssm_sim_get_y(sims[i], m, p, q);
      }
      // filter simulated y's
      filter = ssm_filter(y, d, Z, H, c, T, R, Q, a1, P1);
      // mean correct eta samples
      etahat_plus = ssm_smooth_eta(filter, Z, T, R, Q);
      for (i in 1:n) {
        draws[i] = (ssm_sim_get_eta(sims[i], m, p, q)
                                  - ssm_smooth_eta_get_mean(etahat_plus[i], q)
                                  + ssm_smooth_eta_get_mean(eta[i], q));
      }
    }
    return draws;
}
```

4.8.3 ssm_simsmo_eps_rng

Parameters:

- `vector[]` **eps** Values returned by `sim_smooth_eps`
- `vector[]` **d** Observation intercept, $\boldsymbol{d}_t$. An array of $p \times 1$ vectors.
- `matrix[]` **Z** Design matrix, $\boldsymbol{Z}_t$. An array of $p \times m$ matrices.
- `matrix[]` **H** Observation covariance matrix, $\boldsymbol{H}_t$. An array of $p \times p$ matrices.
- `vector[]` **c** State intercept, $\boldsymbol{c}_t$. An array of $m \times 1$ vectors.
- `matrix[]` **T** Transition matrix, $\boldsymbol{T}_t$. An array of $m \times m$ matrices.
- `matrix[]` **R** State covariance selection matrix, $\boldsymbol{R}_t$. An array of $p \times q$ matrices.
- `matrix[]` **Q** State covariance matrix, $\boldsymbol{Q}_t$. An array of $q \times q$ matrices.
- `vector` **a1** Expected value of the intial state, $a_1 = \mathrm{E}(\alpha_1)$. An $m \times 1$ matrix.
- `matrix` **P1** Variance of the initial state, $P_1 = \mathrm{Var}(\alpha_1)$. An $m \times m$ matrix.

Return Value: `vector[]` Array of size n of $p \times 1$ vectors containing a single draw from $(\boldsymbol{\varepsilon}_{1:n} | \boldsymbol{y}_{1:n})$.

Observation disturbance simulation smoother

Draw samples from the posterior distribution of the observation disturbances, $\tilde{\boldsymbol{\varepsilon}}_{1:n} \sim p(\boldsymbol{\varepsilon}_{1:n} | \boldsymbol{y}_{1:n})$.

For `d`, `Z`, `H`, `c`, `T`, `R`, `Q` the array can have a size of 1, if it is not time-varying, or a size of n (for `d`, `Z`, `H`) or $n - 1$ (for `c`, `T`, `R`, `Q`) if it is time varying.

This draws samples using mean-correction simulation smoother of [10]. See [11, Sec 4.9].

```
vector[] ssm_simsmo_eps_rng(vector[] eps,
                      vector[] d, matrix[] Z, matrix[] H,
                      vector[] c, matrix[] T, matrix[] R, matrix[] Q,
                      vector a1, matrix P1) {
    vector[dims(Z)[2]] draws[size(eps)];
    int n;
    int p;
    int m;
    int q;
    n = size(eps);
    p = dims(Z)[2];
    m = dims(Z)[3];
    q = dims(Q)[2];
    {
      vector[ssm_filter_size(m, p)] filter[n];
      vector[p] y[n];
      vector[ssm_sim_size(m, p, q)] sims[n];
      vector[ssm_smooth_eta_size(p)] epshat_plus[n];
      // simulate unconditional disturbances and observations
      sims = ssm_sim_rng(n, d, Z, H, c, T, R, Q, a1, P1);
      for (i in 1:n) {
        y[i] = ssm_sim_get_y(sims[i], m, p, q);
      }
      // filter simulated y's
      filter = ssm_filter(y, d, Z, H, c, T, R, Q, a1, P1);
      // mean correct epsilon samples
      epshat_plus = ssm_smooth_eps(filter, Z, H, T);
      for (i in 1:n) {
        draws[i] = (ssm_sim_get_eps(sims[i], m, p, q)
```

```
                    - ssm_smooth_eps_get_mean(epshat_plus[i], p)
                    + ssm_smooth_eps_get_mean(eps[i], p));
      }
    }
    return draws;
}
```

4.9 Stationary

4.9.1 pacf_to_acf

Parameters:

- `vector` **x** A vector of coefficients of a partial autocorrelation function

Return Value: `vector` A vector of coefficients of an Autocorrelation function

Partial Autocorrelations to Autocorrelations

```
vector pacf_to_acf(vector x) {
  matrix[num_elements(x), num_elements(x)] y;
  int n;
  n = num_elements(x);
  y = rep_matrix(0.0, n, n);
  for (k in 1:n) {
    for (i in 1:(k - 1)) {
      y[k, i] = y[k - 1, i] + x[k] * y[k - 1, k - i];
    }
    y[k, k] = x[k];
    print(y);
  }
  return -y[n] ';
}
```

4.9.2 constrain_stationary

Parameters:

- `vector` **x** An unconstrained vector in $(-\infty, \infty)$

Return Value: `vector` A vector of coefficients for a stationary AR or inverible MA process.

Constrain vector of coefficients to the stationary and intertible region for AR or MA functions.

See Jones [15], Jones [14], Monahan [18], Ansley and Kohn [1], and the functions `tools.constrain_stationary_univariate` and `tools.unconstraine_stationary_univariate` in statsmodels.tsa.statespace.

```
vector constrain_stationary(vector x) {
  vector[num_elements(x)] r;
  int n;
  n = num_elements(x);
```

```
  // transform (-Inf, Inf) to (-1, 1)
  for (i in 1:n) {
    r[i] = x[i] / (sqrt(1.0 + pow(x[i], 2)));
  }
  // Transform PACF to ACF
  return pacf_to_acf(r);
}
```

4.9.3 acf_to_pacf

Parameters:

- `vector` **x** Coeffcients of an autocorrelation function.

Return Value: `vector` A vector of coefficients of the corresponding partial autocorrelation function.

Convert coefficients of an autocorrelation function to partial autocorrelations.

```
vector acf_to_pacf(vector x) {
  matrix[num_elements(x), num_elements(x)] y;
  vector[num_elements(x)] r;
  int n;
  n = num_elements(x);
  y = rep_matrix(0.0, n, n);
  y[n] = -x ';
  for (j in 0:(n - 1)) {
    int k;
    k = n - j;
    for (i in 1:(k - 1)) {
      y[k - 1, i] = (y[k, i] - y[k, k] * y[k, k - i]) / (1 - pow(y[k, k], 2));
    }
  }
  r = diagonal(y);
  return r;
}
```

4.9.4 unconstrain_stationary

Parameters:

- `vector` **x** Coeffcients of an autocorrelation function.

Return Value: `vector` Coefficients of the corresponding partial autocorrelation function.

Transform from stationary and invertible space to $(-\infty, \infty)$.

```
vector unconstrain_stationary(vector x) {
  matrix[num_elements(x), num_elements(x)] y;
  vector[num_elements(x)] r;
  vector[num_elements(x)] z;
  int n;
  n = num_elements(x);
```

```
  // Transform ACF to PACF
  r = acf_to_pacf(x);
  // Transform (-1, 1) to (-Inf, Inf)
  for (i in 1:n) {
    z[i] = r[i] / (sqrt(1.0 - pow(r[i], 2)));
  }
  return z;
}
```

4.9.5 kronecker_prod

Parameters:

- `matrix` **A** An $m \times n$ matrix
- `matrix` **B** A $p \times q$ matrix

Return Value: `matrix` An $mp \times nq$ matrix.

Kronecker product

The Kronecker product of a A and B is

$$A \otimes B = \begin{bmatrix} a_{11}B \cdots a_{1n}B & & \\ \vdots & \ddots & vdots \\ a_{m1}B & \cdots & a_{mn}B \end{bmatrix}.$$

```
matrix kronecker_prod(matrix A, matrix B) {
  matrix[rows(A) * rows(B), cols(A) * cols(B)] C;
  int m;
  int n;
  int p;
  int q;
  m = rows(A);
  n = cols(A);
  p = rows(B);
  q = cols(B);
  for (i in 1:m) {
    for (j in 1:n) {
      int row_start;
      int row_end;
      int col_start;
      int col_end;
      row_start = (i - 1) * p + 1;
      row_end = (i - 1) * p + p;
      col_start = (j - 1) * q + 1;
      col_end = (j - 1) * q + 1;
      C[row_start:row_end, col_start:col_end] = A[i, j] * B;
    }
  }
  return C;
}
```

4.9.6 arima_stationary_cov

Parameters:

- `matrix` **T** The $m \times m$ transition matrix
- `matrix` **R** The $m \times q$ system disturbance selection matrix

Return Value: `matrix` An $m \times m$ matrix with the stationary covariance matrix.

Find the covariance of the stationary distribution of an ARMA model

The initial conditions are $\alpha_1 \sim N(0, \sigma^2 Q_0)$, where Q_0 is the solution to

$$(T \otimes T)\,\mathrm{vec}(Q_0) = \mathrm{vec}(RR')$$

where $\mathrm{vec}(Q_0)$ and $\mathrm{vec}(RR')$ are the stacked columns of Q_0 and RR'

See Durbin and Koopman [11], Sec 5.6.2.

```
matrix arima_stationary_cov(matrix T, matrix R) {
  matrix[rows(T), cols(T)] Q0;
  matrix[rows(T) * rows(T), rows(T) * rows(T)] TT;
  vector[rows(T) * rows(T)] RR;
  int m;
  int m2;
  m = rows(T);
  m2 = m * m;
  RR = to_vector(tcrossprod(R));
  TT = kronecker_prod(T, T);
  Q0 = to_matrix_colwise((diag_matrix(rep_vector(1.0, m2)) - TT) \ RR, m, m);
  return Q0;
}
```

Chapter 5

Other Software

This a brief summary of other available software to estimate state space models with a focus on R and python.

5.1 R packages

Tusell [24] reviews R packages for state space models (as of 2011). Helske [13] includes an more recent review of R packages implementing state space models.

- The **stats** package includes functions for univariate Kalman filtering and smoothing (`KalmanLike`, `KalmanRun`, `KalmanSmooth`, `KalmanForecast`) which are used by `StructTS` and `arima`.
- dse
- sspir
- dlm
- KFAS
- dlmodeler - provides a unified interface to multiple packages
- rucm: structural time series
- MARSS - maximum likelihood estimation of a large glass of Guassian state space models with an EM-algorithm

5.2 Other

The JSS Volume 41 [6] contains articles on state space implementations in multiple languages

- STAMP [17]
- Ox/SsfPack [19]
- R [21]
- SsfPack in S+FinMetrics [25]
- Matlab [20]
- FORTRAN [2]
- eViews [4]

- RATS [8]
- Stata [9]
- gretl [16]
- SAS [22]
- Ox [3]

5.2.1 Stata

Stata's timeseries capabilities includes the command `ssmodels` to estimate general state space models, as well as common special cases: `arima` (SARIMAX models), `dfactor` (Dynamic Factor), and `ucm` (Unobserved Components Models).

5.2.2 Python

The [statsmodels] module [statsmodels.tsa] contains functions and classes for time series analysis including autoregressive (AR), vector autoregressive (VAR), autoregressive moving avergage models (ARMA), and functions fo Kalman filtering. Currently the Kalman filter only handles the special univariate case for ARIMA.

The **statsmodels** module statsmodels.tsa.statespace contains more general state space code. The examples are very good.

An example of using `statsmodels.tsa.statespace` and PyMC to simulate from the posterior of a state space model. See State Space Modeling in Python.

Strickland et al. [23] introduce PySSM to simulate state space models using PyMCMC (not to be confused with the more popular PyMC).

Bibliography

[1] Craig F. Ansley and Robert Kohn. "A note on reparameterizing a vector autoregressive moving average model to enforce stationarity". In: *Journal of Statistical Computation and Simulation* 24.2 (1986), pp. 99–106. DOI: 10.1080/00949658608810893. eprint: http://dx.doi.org/10.1080/00949658608810893. URL: http://dx.doi.org/10.1080/00949658608810893.

[2] William Bell. "REGCMPNT A Fortran Program for Regression Models with ARIMA Component Errors". In: *Journal of Statistical Software* 41.1 (2011), pp. 1–23. ISSN: 1548-7660. DOI: 10.18637/jss.v041.i07. URL: https://www.jstatsoft.org/index.php/jss/article/view/v041i07.

[3] Charles Bos. "a Bayesian analysis of unobserved component models using Ox". In: *Journal of Statistical Software* 41.1 (2011), pp. 1–24. ISSN: 1548-7660. DOI: 10.18637/jss.v041.i13. URL: https://www.jstatsoft.org/index.php/jss/article/view/v041i13.

[4] Filip Van den Bossche. "fitting state space models with EViews". In: *Journal of Statistical Software* 41.1 (2011), pp. 1–16. ISSN: 1548-7660. DOI: 10.18637/jss.v041.i08. URL: https://www.jstatsoft.org/index.php/jss/article/view/v041i08.

[5] C. K. Carter and R. Kohn. "On Gibbs sampling for State Space Models". In: *Biometrika* 81.3 (1994), pp. 541–553. DOI: 10.1093/biomet/81.3.541. eprint: http://biomet.oxfordjournals.org/content/81/3/541.full.pdf+html. URL: http://biomet.oxfordjournals.org/content/81/3/541.abstract.

[6] Jacques J. F. Commandeur, Siem Jan Koopman, and Marius Ooms. "statistical software for state space methods". In: *Journal of Statistical Software* 41.1 (May 12, 2011), pp. 1–18. ISSN: 1548-7660. URL: http://www.jstatsoft.org/v41/i01.

[7] Piet De Jong and Neil Shephard. "The simulation smoother for time series models". In: *Biometrika* 82.2 (1995), pp. 339–350. DOI: 10.1093/biomet/82.2.339. eprint: http://biomet.oxfordjournals.org/content/82/2/339.full.pdf+html. URL: http://biomet.oxfordjournals.org/content/82/2/339.abstract.

[8] Thomas Doan. "state space methods in RATS". In: *Journal of Statistical Software* 41.1 (2011), pp. 1–16. ISSN: 1548-7660. DOI: 10.18637/jss.v041.i09. URL: https://www.jstatsoft.org/index.php/jss/article/view/v041i09.

[9] David Drukker and Richard Gates. "state space methods in Stata". In: *Journal of Statistical Software* 41.1 (2011), pp. 1–25. ISSN: 1548-7660. DOI: 10.18637/jss.v041.i10. URL: https://www.jstatsoft.org/index.php/jss/article/view/v041i10.

[10] J. Durbin and S. J. Koopman. "a simple and efficient simulation smoother for state space time series analysis". English. In: *Biometrika* 89.3 (2002), pp. 603–615. ISSN: 00063444. URL: http://www.jstor.org/stable/4140605.

[11] J. Durbin and S.J. Koopman. *time series analysis by state space methods: second edition*. Oxford Statistical Science Series. OUP Oxford, 2012. ISBN: 9780199641178. URL: http://books.google.com/books?id=fOq39Zh0olQC.

[12] Sylvia Frühwirth-Schnatter. "data augmentation and dynamic linear models". In: *Journal of Time Series Analysis* 15.2 (1994), pp. 183–202. ISSN: 1467-9892. DOI: 10.1111/j.1467-9892.1994.tb00184.x.

[13] Jouni Helske. "KFAS: Kalman filter and smoother for exponential family state space models". In: (2012). R package version 0.9.11. URL: http://CRAN.R-project.org/package=KFAS.

[14] M. C. Jones. "randomly choosing parameters from the stationarity and invertibility region of autoregressive-moving average models". In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 36.2 (1987), pp. 134–138. ISSN: 00359254, 14679876. URL: http://www.jstor.org/stable/2347544.

[15] Richard H. Jones. "maximum likelihood fitting of ARMA models to time series with missing observations". In: *Technometrics* 22.3 (1980), pp. 389–395. ISSN: 00401706. URL: http://www.jstor.org/stable/1268324.

[16] Riccardo Lucchetti. "state space methods in gretl". In: *Journal of Statistical Software* 41.1 (2011), pp. 1–22. ISSN: 1548-7660. DOI: 10.18637/jss.v041.i11. URL: https://www.jstatsoft.org/index.php/jss/article/view/v041i11.

[17] Roy Mendelssohn. "the STAMP software for state space models". In: *Journal of Statistical Software* 41.1 (2011), pp. 1–18. ISSN: 1548-7660. DOI: 10.18637/jss.v041.i02. URL: https://www.jstatsoft.org/index.php/jss/article/view/v041i02.

[18] John F. Monahan. "A note on enforcing stationarity in autoregressive-moving average models". In: *Biometrika* 71.2 (1984), pp. 403–404. DOI: 10.1093/biomet/71.2.403. eprint: http://biomet.oxfordjournals.org/content/71/2/403.full.pdf+html. URL: http://biomet.oxfordjournals.org/content/71/2/403.abstract.

[19] Matteo Pelagatti. "state space methods in Ox/SsfPack". In: *Journal of Statistical Software* 41.1 (2011), pp. 1–25. ISSN: 1548-7660. DOI: 10.18637/jss.v041.i03. URL: https://www.jstatsoft.org/index.php/jss/article/view/v041i03.

[20] Jyh-Ying Peng and John Aston. "The State Space Models Toolbox for MATLAB". In: *Journal of Statistical Software* 41.1 (2011), pp. 1–26. ISSN: 1548-7660. DOI: 10.18637/jss.v041.i06. URL: https://www.jstatsoft.org/index.php/jss/article/view/v041i06.

[21] Giovanni Petris and Sonia Petrone. "state space models in R". In: *Journal of Statistical Software* 41.4 (May 12, 2011), pp. 1–25. ISSN: 1548-7660. URL: http://www.jstatsoft.org/v41/i04.

[22] Rajesh Selukar. "state space modeling using SAS". In: *Journal of Statistical Software* 41.1 (2011), pp. 1–13. ISSN: 1548-7660. DOI: 10.18637/jss.v041.i12. URL: https://www.jstatsoft.org/index.php/jss/article/view/v041i12.

[23] Christopher Strickland et al. "PySSM: A Python Module for Bayesian Inference of Linear Gaussian State Space Models". In: *Journal of Statistical Software* 57.1 (2014), pp. 1–37. ISSN: 1548-7660. DOI: 10.18637/jss.v057.i06. URL: https://www.jstatsoft.org/index.php/jss/article/view/v057i06.

[24] Fernando Tusell. "Kalman filtering in R". In: *Journal of Statistical Software* 39.2 (Mar. 1, 2011), pp. 1–27. ISSN: 1548-7660. URL: http://www.jstatsoft.org/v39/i02.

[25] Eric Zivot. "state space modeling using SsfPack in S+FinMetrics 3.0". In: *Journal of Statistical Software* 41.1 (2011), pp. 1–27. ISSN: 1548-7660. DOI: 10.18637/jss.v041.i05. URL: https://www.jstatsoft.org/index.php/jss/article/view/v041i05.