Journal of Statistical Software

March 2014, Volume 57, Issue 6. *http://www.jstatsoft.org/*

PySSM: A Python Module for Bayesian Inference of Linear Gaussian State Space Models

Christopher M. Strickland
Queensland University
of Technology

Robert L. Burdett
Queensland University
of Technology

Kerrie L. Mengersen
Queensland University
of Technology

Robert J. Denham
Department of Environment
and Resource Management

Abstract

PySSM is a Python package that has been developed for the analysis of time series using linear Gaussian state space models. **PySSM** is easy to use; models can be set up quickly and efficiently and a variety of different settings are available to the user. It also takes advantage of scientific libraries **NumPy** and **SciPy** and other high level features of the Python language. **PySSM** is also used as a platform for interfacing between optimized and parallelized Fortran routines. These Fortran routines heavily utilize basic linear algebra and linear algebra Package functions for maximum performance. **PySSM** contains classes for filtering, classical smoothing as well as simulation smoothing.

Keywords: Bayesian estimation, state space model, time series analysis, Python.

1. Introduction

In this paper, an open source Python module (library) called **PySSM** is presented for the analysis of time series, using state space models (SSMs); see van Rossum (1995) for further details on the Python programming language. The purpose of time series analysis is to identify inherent characteristics in raw time ordered data. State space models are one method for analyzing time series data, and are based upon the assumption that observations can be explained in terms of different components, such as trends, seasonality, cycles, regression elements and disturbance terms. While each component can be modeled separately the SSM

approach combines them together to form a single unified model of the phenomena being studied (Durbin and Koopman 2001). State space modeling of time series is undertaken in many fields including finance, economics and environmental science, amongst others.

The linear Gaussian SSM provides an attractive representation for numerous time series models, including both stationary and non stationary, in either a univariate or multivariate time series setting. The estimation and the analysis of SSMs is covered in numerous books including Anderson and Moore (1979), Harvey (1989) and Durbin and Koopman (2001), amongst others. One of the attractive features of the SSM is that its generic form allows for the application of standard algorithms, of which the most well known is the Kalman filter. There are also numerous other filtering and smoothing algorithms, many of which are used in conjunction with the Kalman filter that are associated with the SSM. Of particular interest for Bayesian analysis are simulation smoothing algorithms, which can be used to jointly sample the state vector of a SSM, from its full conditional posterior distribution. Given the popularity of modeling time series using SSMs, and the complexity of the associated algorithms that are used in estimation, the need for a library such as **PySSM**, is apparent. Whilst there are state space libraries for a range of software environments (see Drukker and Gates 2011, for an overview) – including MATLAB (The MathWorks, Inc. 2011; see also Peng and Aston 2011), R (R Core Team 2013; see also Petris and Petrone 2011), SAS (SAS Institute Inc. 2011; see also Selukar 2011), **STAMP** (Koopman, Harvey, Doornik, and Shephard 2009; see also Mendelssohn 2011), Stata (StataCorp. 2013; see also Drukker and Gates 2011) among others – there is no current comprehensive suite in Python for the analysis of SSMs. **PySSM** is designed to fill this gap.

Python is an interpreted, interactive, object-oriented programming language, and is an ideal programming language for building an SSM library, as it has extensive scientific libraries, such as **NumPy** (Oliphant 2006) and **SciPy** (Jones, Oliphant, and Peterson 2001); see for example Oliphant (2007). Furthermore, Python is easily extensible, has a clean syntax and powerful programming constructs. **NumPy** is a fundamental package that is extremely useful for scientific computing, and contains among other things a powerful N-dimensional array object. **NumPy** is used primarily as an efficient multi-dimensional container of generic data in **PySSM**. Another feature of Python which is particularly important is that it is quite easy to include modules from compiled languages such as C/C++ and Fortran in order to obtain the necessary speed for feasible practical analysis; see for example Strickland, Alston, Denham, and Mengersen (2011). With Python, the user can simply compile Fortran code using a module called **F2py**, see Peterson (2009), or inline C using **Weave**, which is part of **SciPy**, see Oliphant (2007) and use the subroutines directly from Python.

PySSM is a collection of Python classes, foremost of which are the Python classes for filtering and simulation smoothing. Associated with these classes is a suite of Fortran functions and subroutines that are heavily optimized and make use of basic linear algebra subroutines (**BLAS**) and the linear algebra package (**LAPACK**); see Anderson *et al.* (1999) for details. All of the computationally intensive calculations in **PySSM** are performed using these routines, to ensure that the majority of the computation is undertaken with optimized compiled code. The user, however, interacts with the high level Python interface to these routines, which helps to minimize development time.

The examples included with the package, **PySSM**, require the **PyMCMC** library of Strickland *et al.* (2011) that provides a solution to the complex integration problems faced in the Bayesian analysis of statistical problems. It consists of a variety of Markov chain Monte Carlo (MCMC)

algorithms. In particular it contains classes for the Gibbs sampler, Metropolis Hastings (MH), independent MH, random walk MH, orientational Monte Carlo (OBMC) as well as the slice sampler.

The aim of this paper is to describe **PySSM**, and to illustrate its use in Bayesian analysis of SSMs. Note that the package also contains numerous algorithms that can be used for classical analysis. In Section 2, the SSM is specified and the application of Bayesian estimation is reviewed. In Section 3 the **PySSM** classes are described in detail. Section 4 then illustrates the application of **PySSM** to a variety of examples. Section 5 concludes.

2. State space model

The linear Gaussian SSM, considered in this paper, the $(p \times 1)$ vector of observations, $\boldsymbol{y}_t$, is generated by

$$\boldsymbol{y}_t = \boldsymbol{X}_t\boldsymbol{\beta}_m + \boldsymbol{Z}_t\boldsymbol{\alpha}_t + \boldsymbol{R_t}\boldsymbol{\varepsilon}_t, \;\; \boldsymbol{\varepsilon}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{H}_t), \tag{1}$$

where $\boldsymbol{X}_t$ is a $(p \times k_m)$ matrix of regressors, $\boldsymbol{\beta}_m$ is a $(k_m \times 1)$ vector of regression coefficients, $\boldsymbol{Z}_t$ is a $(p \times m)$ system matrix, $\boldsymbol{\alpha}_t$ is an an $(m \times 1)$ state vector, $\boldsymbol{\varepsilon}_t$ is a $(p \times 1)$ vector of normally distributed random variables, with a mean vector $\boldsymbol{0}$ and a $(p \times p)$ covariance matrix, $\boldsymbol{H}_t$. The state, $\boldsymbol{\alpha}_t$, is generated by the following relation,

$$\boldsymbol{\alpha}_{t+1} = \boldsymbol{W}_t\boldsymbol{\beta}_s + \boldsymbol{T}_t\boldsymbol{\alpha}_t + \boldsymbol{G}_t\boldsymbol{\eta}_t, \;\; \boldsymbol{\eta}_t \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{Q}_t), \tag{2}$$

where $\boldsymbol{W}_t$ is an $(m \times k_s)$ matrix of regressors, $\boldsymbol{\beta}_s$ is a $(k_s \times 1)$ vector of regression coefficients, $\boldsymbol{T}_t$ is an $(m \times m)$ transition matrix, $\boldsymbol{G}_t$ is an $(m \times r)$ system matrix and $\boldsymbol{\eta}_t$ is normally distributed, with a mean $\boldsymbol{0}$ and a $(r \times r)$ covariance matrix $\boldsymbol{Q}_t$. Let $\boldsymbol{\beta} = \left(\boldsymbol{\beta}_s^\top, \boldsymbol{\beta}_m^\top\right)$. The initial state is distributed as follows:

$$\boldsymbol{\alpha}_1 \sim \mathcal{N}(\boldsymbol{a}_1, \boldsymbol{P}_1), \tag{3}$$

where $\boldsymbol{a}_1$ is an $(m \times 1)$ mean vector and $\boldsymbol{P}_1$ is the $(m \times m)$ covariance matrix for the initial state. The set of unknown parameters in the system matrices is denoted by the $(j \times 1)$ vector $\boldsymbol{\theta}$. Given the measurement equation in 1, for the case where $\boldsymbol{R}_t\boldsymbol{H}_t\boldsymbol{R}_t^\top$ is non-singular, the joint probability density function (*pdf*) for the set of observations, $\boldsymbol{y}$, is multivariate normal, where $\boldsymbol{y} = \left(\boldsymbol{y}_1^\top, \boldsymbol{y}_2^\top, \ldots, \boldsymbol{y}_n^\top\right)^\top$, is distributed as

$$p(\boldsymbol{y}|\boldsymbol{\alpha}, \boldsymbol{\theta}) \propto \prod_{t=1}^{n} \left|\boldsymbol{R}\boldsymbol{H}_t\boldsymbol{R}^\top\right|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(\boldsymbol{y}_t - \boldsymbol{X_t}\boldsymbol{\beta}_m - \boldsymbol{Z_t}\boldsymbol{\alpha}_t)^\top \left(\boldsymbol{R}_t\boldsymbol{H}_t\boldsymbol{R}_t^\top\right)^{-1} (\boldsymbol{y}_t - \boldsymbol{X}_t\boldsymbol{\beta}_m - \boldsymbol{Z}_t\boldsymbol{\alpha}_t)\right\},$$

where $\boldsymbol{\alpha} = (\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2, \ldots, \boldsymbol{\alpha}_n)$. From (2), it is clear that the joint *pdf* for the state, $\boldsymbol{\alpha}$, for the case where $\boldsymbol{G}_t\boldsymbol{Q}_t\boldsymbol{G}^\top$ is non-singular, is given by

$$\begin{aligned} p(\boldsymbol{\alpha}|\boldsymbol{\theta}) &= p(\boldsymbol{\alpha}_1|\boldsymbol{\theta}) \prod_{t=1}^{n-1} p(\boldsymbol{\alpha}_{t+1}|\boldsymbol{\alpha}_t, \boldsymbol{\theta}) \\ &\propto |\boldsymbol{P}_1|^{-1/2} \exp\left\{-\frac{1}{2}(\boldsymbol{\alpha}_1 - \boldsymbol{a}_1)^\top \boldsymbol{P}_1^{-1}(\boldsymbol{\alpha}_1 - \boldsymbol{a}_1)\right\} \prod_{t=1}^{n-1} \left|\boldsymbol{G}_t\boldsymbol{Q}_t\boldsymbol{G}_t^\top\right|^{-1/2} \\ &\times \exp\left\{-\frac{1}{2}\sum_{t=1}^{n-1} (\boldsymbol{\alpha}_{t+1} - \boldsymbol{W}_t\boldsymbol{\beta}_s - \boldsymbol{T}_t\boldsymbol{\alpha}_t)^\top \left(\boldsymbol{G}_t\boldsymbol{Q}_t\boldsymbol{G}_t^\top\right)^{-1} (\boldsymbol{\alpha}_{t+1} - \boldsymbol{W}_t\boldsymbol{\beta}_s - \boldsymbol{T}_t\boldsymbol{\alpha}_t)\right\}. \end{aligned} \tag{4}$$

2.1. Bayesian estimation of SSMs

Bayesian inference summarizes uncertainty about the unknown parameters of interest through the joint posterior density function. MCMC is probably the most common way to conduct Bayesian analysis of SSMs.

A generic Bayesian algorithm for a SSM using MCMC can be given, at iteration j by:

1. Sample $\boldsymbol{\alpha}^j$ from $p\left(\boldsymbol{\alpha}|\boldsymbol{y}, \boldsymbol{\theta}^{j-1}, \boldsymbol{\beta}^{j-1}\right)$.
2. Sample $\boldsymbol{\theta}^j$ from $p\left(\boldsymbol{\theta}|\boldsymbol{y}, \boldsymbol{\alpha}^j, \boldsymbol{\beta}^{j-1}\right)$.
3. Sample $\boldsymbol{\beta}^j$ from $p\left(\boldsymbol{\beta}|\boldsymbol{y}, \boldsymbol{\theta}^j, \boldsymbol{\alpha}^j\right)$.

Algorithm 1: A generic algorithm for the MCMC estimation of a linear Gaussian SSM.

For a linear Gaussian SSM, the standard approach for sampling the state in Step 1 is to use a simulation smoothing algorithm. Alternative algorithms are provided by Carter and Kohn (1994), De Jong and Shephard (1995), Durbin and Koopman (2002), Frühwirth-Schnatter (2004) and Strickland, Turner, Denham, and Mengerson (2009). **PySSM** contains a simulation smoothing class, which uses the most computationally efficient, of the aforementioned simulation smoothing algorithms, given the specified SSM.

Step 2 is model specific, but often makes use of computing the log of the probability of the measurement or state equations or log-likelihood function. **PySSM** contains functions for these computations, which are described in later sections and **PyMCMC** simplifies the task of setting up the MCMC algorithm.

Step 3 is easily computed using the '`CondRegressionSampler`' class in **PyMCMC**. This is demonstrated in the following examples. Alternatively, **PySSM** can also be used to jointly sample the regression coefficients, $\boldsymbol{\beta}$, and the state vector, $\boldsymbol{\alpha}$, from their joint posterior distribution $p\left(\boldsymbol{\beta}, \boldsymbol{\alpha}|\boldsymbol{y}, \boldsymbol{\theta}, \boldsymbol{\alpha}\right)$.

3. Python implementation

A description of the Python implementation is now presented. Unless otherwise stated, all arrays used in **PySSM** are initialized as **NumPy** arrays. All multi-dimensional arrays are also initialized with the `order = 'F'` Fortran contiguous ordering option. This ensures that multi-dimensional data are stored in memory in a column-wise fashion.

The three main classes '`System`', '`Filter`', '`SimSmoother`' that constitute **PySSM** are described, in the following subsections. There is no unifying class that encapsulates instances of these classes for SSMs, but it is quite feasible for one to be created if necessary. The classes remain separate because users may only want to, for instance use the filtering algorithms, simulate data or simply compute the log-likelihood, and therefore do not need to allocate the additional memory that is required for simulation smoothing classes. It should be noted though that the class '`Filter`' depends on the class '`System`' and that the class '`SimSmoother`' makes use of both the '`Filter`' and '`System`' classes.

3.1. 'System' class

A class named '`System`' is used to encapsulate the system matrices and other important parameters for the SSM. One should not have to manually create the class '`System`' as it is created as apart of the '`Filter`' and '`SimSmoother`' classes. Note, however, both classes have accessor functions that supply the user with a reference to the '`System`' class, allowing the user to modify the system matrices when required. An instance of type '`System`' is created (i.e., by the class constructor), with the following arguments:

`nobs`: An integer specifying the number of time series observations.

`nseries`: An integer specifying the number of time series.

`nstate`: An integer specifying the dimension of the state vector.

`rstate`: An integer specifying the dimension of the covariance matrix Q_t.

`nreg`: An integer specifying the number of regressors in the measurement equation.

`sreg`: An integer specifying the number of regressors in the state equation.

`timevar`: A Boolean flag or dictionary (Python data structure) that is used for specifying which of the system matrices is time varying. For example, if the parameter is input as a Boolean then each of the system parameters is defined as either time varying or not time varying according to whether the Boolean entered is `True` or `False`. If specific system matrices are to be time varying while others are not then a dictionary is required as input. An example is as follows: `timevar = {'gt':True, 'qt':False, 'zt':False,'tt':False, 'ht':False, 'rt':False}`. Note that the keyword is on the left and the property on the right. The system matrices are differentiated by the following keys: `'tt'`, `'qt'`, `'gt'`, `'ht'`, `'zt'` and `'rt'`. In particular, the first letter refers to the particular system matrix and the second letter refers to the time index. In this example only $\boldsymbol{G}_t$, see (2), is time varying. An important point to note, only the system matrices that are time varying need to be specified in the dictionary. For instance, in the above example `timevar = {'gt':True}` is sufficient to produce the same result.

`properties`: A dictionary (Python data structure) used to describe special structures of the system matrices that allow additional specific numerical optimizations in the algorithms implemented in **PySSM**. By default all system matrices are regarded as "standard" and specified in the following way: `properties = {'tt':'default', 'qt':'default', 'gt':'default', 'ht':'default', 'zt':'default', 'rt':'default'}`.
Special structures of the system matrices other than the default include: identity, diagonal, and null (zero matrix). They are abbreviated by the following strings: `'null'`, `'eye'` and `'diag'`, and can be specified for certain matrices.

Table 1 lists the currently available optimization options, for each of the system matrices. Also shown is whether or not compressed storage is used, for a particular specification. Essentially, compressed storage implies for both the cases where the specified system matrix is classed as either a diagonal or an identity matrix that only the diagonal of the matrix is stored. For

SM	Label	Currently available options	Compressed storage
$\boldsymbol{Z}_t$	`'zt'`	`'default'`	No
$\boldsymbol{R}_t$	`'rt'`	`'default'`, `'diag'`, `'eye'`	Yes (for `'diag'` and `'eye'`)
$\boldsymbol{H}_t$	`'ht'`	`'default'`, `'null'`, `'diag'`, `'eye'`	Yes (for `'diag'` and `'eye'`)
$\boldsymbol{T}_t$	`'tt'`	`'default'`	No
$\boldsymbol{G}_t$	`'gt'`	`'default'`, `'eye'`	No
$\boldsymbol{Q}_T$	`'qt'`	`'default'`	No

Table 1: Specifies the options that are currently available for each of the system matrices. The first column specifies the system matrix (SM), the second column specifies its label, the third column specifies the options currently available and the fourth column specifies whether or not compressed storage is used for the given option; see the description below for an explanation of compressed storage.

example, if $\boldsymbol{H}_t$ is a diagonal matrix then the user should only pass in a $(p \times 1)$ vector (in the case that $\boldsymbol{H}_t$ is not time varying) when initializing (or updating) the matrix.

Note that only the arguments for the system matrices that are not default need to be included in the dictionary. For instance, suppose the system matrix $\boldsymbol{H}_t$ is diagonal, but all the remaining matrices have no special structure. In this case, the user can simply specify; `properties = {'ht':'diag'}`. Also note that just because a property for a system matrix is given does not necessitate the use of an optimization. This will depend upon the specific property and the specific algorithm being used.

The system class also contains a number of "public" member functions which have been provided to access and update the system matrices. To access any of the system matrices the user can simply use the label associated with the system matrix as `label()`. The labels for the system matrices are `(tt, zt, ht, rt, cht, qt, cqt, gt, gqg, gcqt, rhr, rcht, p1, a1, cholp1, wt, xt, beta)`. Note that we will use the labels interchangeably with the names of the system matrices to simplify discussion. For instance, we may use `ht` to refer to $\boldsymbol{H}_t$. It should be noted that `cht`, `cqt` and `cholp1` refer to the Cholesky factorization of `ht`, `qt` and `p1`, respectively. Similarly `gcqt` refers to the product of `gt` and `cqt`; `gqg` refers to the product of `gt`, `qt` and the transpose of `gt`; `rcht` is the product of `rt` and `cht`, and `rhr` is the product of `rt`, `ht` and the transpose of `rt`. Further, the labels `xt` and `wt` refer to $\boldsymbol{X}_t$ and $\boldsymbol{W}_t$ in (1) and (2), respectively. For example, if the class instance is called `system` then the system matrix $\boldsymbol{T}_t$ can be accessed as `system.tt()`. If in this case the user wanted to access the element in the first row and column then they would use `system.tt()[0, 0]`.

The public update functions have the following specification: `update_label(array)` where `label` is one of the following `(tt, zt, ht, rt, qt, gt, p1, a1, beta, wt, xt)`. The new value of the system matrix denoted by "array" is required as input by each update function. These update functions call a number of private member functions to compute (update) all associated components. For example, updating the system matrix corresponding to the label `qt` results in the matrices corresponding to the labels `qt`, `cqt`, `gcqt` and `gqg` being updated automatically. Similarly, updating $\boldsymbol{G}_t$ through the function `gt()` results in the matrices corresponding to the labels `gt, gcqt` and `qgq` being updated. If `ht` is updated then `ht`, `cht`, `rcht` and `rhr` are also updated. Similarly, if `rt` is updated then `rt`, `rcht` and `rhr` are also updated. Two other update functions are also provided with the following specification: `update_gt_qt(gt, qt)` and `update_rt_ht(rt, ht)`. These update functions take two

standard arguments and perform both updates. In other words they reduce the redundancy of calling an `update_gt` followed by an `update_qt` or vice versa, or an `update_rt` followed by an `update_ht` or vice versa. These functions are necessary because of the requirement for additional calculations to be performed that allow numerical optimizations to be used. For example, the Cholesky factorization of $\boldsymbol{Q}_t$ and $\boldsymbol{H}_t$ are needed in many calculations.

An instance of of type '`System`' is created for use within the simulation smoother and filtering classes and is freely accessible from within these instances.

3.2. Filtering

A class named '`Filter`' is provided to accomplish all filtering tasks required by the SSM. The class can also be used to simulate data. The purpose of filtering is to update knowledge of the system each time a new observation is brought in (Durbin and Koopman 2001). An instance of type Filter is created (i.e., with the class constructor) with similar arguments as for class '`System`'. For example, a class instance `filter` can be created as follows

```
filter = Filter(nobs, nseries, nstate, rstate, timevar, **kwargs)
```

where `nobs, nseries, nstate` and `rstate` have the same meaning as described in Section 3.1. There are also additional optional arguments that are input using the special Python syntax `**kwargs`. This double asterisk form is used to pass a key worded variable-length argument list to a function. The following optional arguments can be specified:

`joint_sample:` The diffuse Kalman filter of De Jong (1991) is used. There are two options. The first is [$\boldsymbol{b}$, $\boldsymbol{B}$, $\boldsymbol{W}_0$], where $\boldsymbol{b}$ and $\boldsymbol{B}$, are from the definition

$$\boldsymbol{\beta} \sim \mathcal{N}\left(\boldsymbol{b}, \boldsymbol{B}\boldsymbol{B}^\top\right),$$

that is $\boldsymbol{\beta}$ is distributed normal, with a mean vector $\boldsymbol{b}$ and a covariance $\boldsymbol{B}\boldsymbol{B}^\top$. Here we assume that $\boldsymbol{b} = \left(\boldsymbol{b}_s^\top, \boldsymbol{b}_m^\top\right)^\top$ and $\boldsymbol{B} = \begin{pmatrix} \boldsymbol{B}_s & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{B}_m \end{pmatrix}$, where

$$\boldsymbol{\beta}_s \sim \mathcal{N}\left(\boldsymbol{b}_s, \boldsymbol{B}_s\boldsymbol{B}_s^\top\right)$$

and

$$\boldsymbol{\beta}_m \sim \mathcal{N}\left(\boldsymbol{b}_m, \boldsymbol{B}_m\boldsymbol{B}_m^\top\right).$$

Note that under the assumptions of De Jong (1991) $\boldsymbol{a}_1 = \boldsymbol{W}_0\boldsymbol{\beta}_s$. The second option is ['diffuse', $\boldsymbol{W}_0$], where a flat prior is assumed for $\boldsymbol{\beta}$.

`filter:` Specifies the filtering algorithm. There are two options, namely `'dkbenchmark'` and `'c_filter'`. If no options are specified then the default filtering algorithm, which may change depending on the properties defined, is used. The keyword `'dkbenchmark'` refers to a filter that is implemented (without alteration) from Durbin and Koopman (2001). This algorithm can be very inefficient (particularly for multivariate time series) and should only be used for benchmarking. The algorithm `c_filter` is the contemporaneous version of the Kalman filter; see (Durbin and Koopman 2001, p. 68) for further details.

`smoother`: By default the state smoother is used; however, there is the option of using a disturbance smoother, through the option `'disturbance'`.

`properties`: Required in the initialization of an instance of the class '`System`' (as described in the preceding subsection). If this option is not used then the default settings are used, in which it is assumed that each of the system matrices are dense.

`wmat`: Specifies the matrix $\boldsymbol{W}_t$, see (2), which is used if regressors are desired in the state equation of the SSM. If this option is not used then there is no such term in the state equation. The parameter `sreg` is extracted from the array that is input (i.e., size of the second dimension) and is not explicitly input in the constructor.

`xmat`: Specifies the matrix $\boldsymbol{X}_t$, see (1), which is to be used if regressors are to be included in the measurement equation of the SSM. If this option is not used then there is no such term in the measurement equation. The parameter `nreg` is extracted from the array that is input (i.e., size of the second dimension) and is not explicitly input in the constructor.

The following is an example of how to use `kwargs` to specify additional options for the '`Filter`' class:

```
filter = Filter(nobs, nseries, nstate, rstate, timevar, wmat = W)
```

where in this example $\boldsymbol{W}$ is an $(m \times k \times n)$ array of regressors. Note that using this notation implies that $\boldsymbol{W}_t$, in (2) is `W[:,:,t]`.

The '`Filter`' class must be initialized before use, with the following member function:

```
initialise_system(a1, p1, zt, ht, tt, gt, qt, rt, **kwargs)
```

The optional argument for this function is `beta`. This argument should only be used if the SSM includes regressors.

A number of other public member functions are also provided. These are described below:

`get_ymat()`: Returns the (`nseries` by `nobs`) array of observations, $\boldsymbol{y} = (\boldsymbol{y}_1; \boldsymbol{y}_2; \ldots, ; \boldsymbol{y}_n)$. For example, suppose the instance of the class '`Filter`' is called `filter`, then the data may be obtained using the code:

```
y = filter.get_ymat()
```

In this case $\boldsymbol{y}_t$ is obtained as `y[:,t]`.

`get_state()`: Returns the `nstate` by `nobs` simulated values for the state, $\boldsymbol{\alpha} = (\boldsymbol{\alpha}_1; \boldsymbol{\alpha}_2; \ldots; \boldsymbol{\alpha}_n)$. Note that, the simulated state is initialized as $\mathbf{0}$ so if the function `simssm()` (which is described below) has not been called then a zero array is returned.

`update_ymat(ymat)`: Updates the class copy of `ymat`, where `ymat` refers to the data set, $\boldsymbol{y}$.

`simssm()`: Simulates data from the specified SSM. Note that it returns nothing. For example, if the class instance is `filter` and assuming the filter has been initialized, using the public member function `initialise_system`, then the user may simulate and obtain the data set using the following code:

```
filter.simssm()
data = filter.get_ymat()
```

In this case `data` denotes the data set, $\boldsymbol{y}$.

`calc_log_likelihood()`: Returns the log-likelihood for the SSM.

`filter()`: Runs the specified filtering algorithm.

`smoother()`: Function runs the specified smoothing algorithm.

3.3. Simulation smoothing

A class named '`SimSmoother`' is provided to accomplish all simulation smoothing tasks required by the SSM. The purpose of smoothing is to draw state variables (samples) from their conditional posterior distribution given parameters and the observations. An instance of type '`SimSmoother`' is initialized with similar arguments as '`System`':

```
smoother = SimSmoother(ymat, nstate, rstate, timevar, **kwargs)
```

The values of `nseries` and `nobs` are extracted from the dimensions of `ymat`. This class has similar optional arguments as class '`Filter`' that are input using the Python syntax `**kwargs`. Like '`Filter`', this class must also be initialized using member function `initialise_system`, which has the same arguments as input, including the `beta` optional argument; see Section 3.2.

The '`SimSmoother`' class has similar member functions to the '`Filter`' class, i.e., same functions as described in Section 3.2. There are several other public member functions to note and these are described below:

`get_meas_residual()`: Returns the residuals from the measurement equation of the SSM. This procedure computes the measurement residuals as $\boldsymbol{\varepsilon}_t = \boldsymbol{R}_t^{-1}\left(\boldsymbol{y}_t - \boldsymbol{X}_t\boldsymbol{\beta}_m - \boldsymbol{Z}_t\boldsymbol{\alpha}_t\right)$, for $t = 1, 2, \ldots, n$.

`get_state_residual()`: Returns the (`rstate` by `nobs`) matrix of residuals for the state, where the residuals are calculated as $\boldsymbol{\eta}_t = \boldsymbol{G}_t^{\dagger}\left(\boldsymbol{\alpha}_{t+1} - \boldsymbol{W}_t\boldsymbol{\beta}_s - \boldsymbol{T}_t\boldsymbol{\alpha}_t\right)$, for $t = 1, 2, \ldots, n-1$, where $\boldsymbol{G}_t^{\dagger}$ is the pseudo-inverse of $\boldsymbol{G}_t$. See Appendix A for a cautionary note.

`log_probability_state()`: Returns the log probability of the simulated state. Specifically, this function computes the log of $p\left(\boldsymbol{\alpha}|\boldsymbol{\theta}\right)$, where

$$
\begin{aligned}
p(\boldsymbol{\alpha}|\boldsymbol{\theta}) = (2\pi)^{-nr/2} \times \prod_{t=1}^{n-1} \left(\left|\boldsymbol{G}_t\boldsymbol{Q}_t\boldsymbol{G}_t^{\top}\right|^{*}\right)^{-1/2} \\
\times \exp\left\{-\frac{1}{2}\sum_{t=1}^{n-1}\left[\boldsymbol{G}_t^{\dagger}\left(\boldsymbol{\alpha}_{t+1} - \boldsymbol{W}_t\boldsymbol{\beta}_s - \boldsymbol{T}_t\boldsymbol{\alpha}_t\right)\right]^{\top}\boldsymbol{Q}_t^{-1}\left[\boldsymbol{G}_t^{\dagger}\left(\boldsymbol{\alpha}_{t+1} - \boldsymbol{W}_t\boldsymbol{\beta}_s - \boldsymbol{T}_t\boldsymbol{\alpha}_t\right)\right]\right\} \\
\times |\boldsymbol{P}_1|^{-1/2} \exp\left\{-\frac{1}{2}(\boldsymbol{\alpha}_1 - \boldsymbol{a}_1)^{\top}\boldsymbol{P}_1^{-1}\left(\boldsymbol{\alpha}_1 - \boldsymbol{a}_1\right)\right\}, \qquad (5)
\end{aligned}
$$

where following Rue and Held (2005) $\left|\boldsymbol{G}_t\boldsymbol{Q}_t\boldsymbol{G}_t^{\top}\right|^{*}$ is defined as product of the non zero eigenvalues; we refer to this as a generalized determinant. See Appendix B for a cautionary note.

`log_probability_meas()`: Returns the log probability of the measurement equation given the state vector and the system matrices. Specifically, $\log p\left(\boldsymbol{y}|\boldsymbol{W},\boldsymbol{X},\boldsymbol{\alpha},\boldsymbol{\theta}\right)$ is returned, where

$$p(\boldsymbol{y}|\boldsymbol{W},\boldsymbol{X},\boldsymbol{\alpha},\boldsymbol{\theta}) = (2\pi)^{-np/2}\prod_{t=1}^{n}\left|\boldsymbol{R}_t\boldsymbol{H}_t\boldsymbol{R}_t^\top\right|^{-1/2} \times \exp\left\{-\frac{1}{2}\sum_{t=1}^{n}\left(\boldsymbol{y}_t-\boldsymbol{X}_t\boldsymbol{\beta}_m-\boldsymbol{Z}_t\boldsymbol{\alpha}_t\right)^\top\left(\boldsymbol{R}_t\boldsymbol{H}_t\boldsymbol{R}_t^\top\right)^{-1}\left(\boldsymbol{y}_t-\boldsymbol{X}_t\boldsymbol{\beta}_m-\boldsymbol{Z}_t\boldsymbol{\alpha}_t\right)\right\}.$$

`get_zt_times_state()`: Returns the (`nseries` by `nobs`) array $\boldsymbol{Z}_t\times\boldsymbol{\alpha}_t$.

4. Examples

The application of **PySSM** to three examples is illustrated in this section. For each example, the model of interest is specified, and code snippets are provided. In addition to the module **PySSM**, the libraries **PyMCMC**, **NumPy**, **SciPy** and **Matplotlib** are also required. Note that **NumPy**, **SciPy** and **Matplotlib** are used directly in some cases and are also dependencies for **PyMCMC**. The reader is expected to be familiar with **NumPy** at a minimum. Previewing the documentation for **PyMCMC** would also help in following the examples.

4.1. Example 1: Autoregressive model with regressors

The first example is a common univariate time series model, and can be found in `example_ssm_ar1_reg.py`. The measurement equation for the model is defined as follows:

$$y_t = \alpha_t + \varepsilon_t;\;\; \varepsilon_t \sim N\left(0,\sigma_\epsilon^2\right), \tag{6}$$

where α_t is a trend component, and ε_t is an irregular component. The irregular component is normally distributed, with a mean 0 and a variance σ_ε^2. The trend is specified as a first order autoregressive process as follows:

$$\alpha_{t+1} = \beta + \rho\alpha_t + \eta_t;\;\; \eta_t \sim N\left(0,\sigma_\eta^2\right), \tag{7}$$

where the constant is defined as $\beta = \mu\left(1-\rho\right)$ and η_t is normally distributed, with mean 0 and variance σ_η^2. Assuming that the autoregressive process has been running since time immemorial then the initial state for the time series model in (6) and (7) is defined as

$$\alpha_1 \sim N\left(\mu,\frac{\sigma_\eta^2}{1-\rho^2}\right). \tag{8}$$

An MCMC algorithm for this example is as follows:

1. Sample $\boldsymbol{\alpha}^j$ from $p\left(\boldsymbol{\alpha}|\boldsymbol{y}, \sigma_\varepsilon^{j-1}, \rho^{j-1}, \sigma_\eta^{j-1}, \beta^{j-1}\right)$
2. Sample σ_ε^{2j} from $p\left(\sigma_\varepsilon^2|\boldsymbol{y}, \boldsymbol{\alpha}^j, \rho^{j-1}, \sigma_\eta^{j-1}, \beta^{j-1}\right)$
3. Sample $\beta^j, \rho^j, \sigma_\eta^j$ from $p\left(\beta, \rho, \sigma_\eta|\boldsymbol{y}, \boldsymbol{\alpha}^j, \sigma_\varepsilon^j\right)$

Algorithm 2: MCMC algorithm for autoregressive model with regressors.

In Step 1, Algorithm 2, the state, $\boldsymbol{\alpha}$, is jointly sampled from its full conditional posterior distribution, (4). This is achieved using the class '`SimSmoother`' in **PySSM**. In Step 2, σ_η is sampled from its posterior distribution. In the code we accomplish this step with the help of the class '`CondScaleSampler`', which is a part of **PyMCMC**. In Step 3, the parameters β, ρ and σ_ε^2 are jointly sampled from their posterior distribution using the class '`CondRegressionSampler`', which is a part of **PyMCMC**.

The program used for estimation, following Algorithm 2 is detailed below. The layout of the program can be summarized with the following steps:

1. Import libraries.
2. Define a function to simulate data.
3. Define a function to sample the state, $\boldsymbol{\alpha}$, from its posterior distribution. This function is used in Step 1, Algorithm 2.
4. Define a function to sample from the posterior distribution of σ_η, which is used in Step 2, Algorithm 2.
5. Define functions for the prior distributions of σ, ρ and for the joint posterior distribution for σ, ρ and β, which are used in Step 3, Algorithm 2.
6. Define the main part of the procedure.
   - Load data.
   - Initialize system matrices for SSM.
   - Instantiate a class instance of '`SimSmoother`', which is used in Step 3 of the program.
   - Instantiate a class instance of '`ScaleSampler`', which is used in Step 4 of the program.
   - Instantiate a class instance of '`LinearModel`', which is used in Step 5 of the program.
   - Define the objects `simstate`, `sampleht` and `samplesigbetarho`, which define the blocks of the MCMC sampling scheme.
   - Instantiate a class instance of '`MCMC`' and run the MCMC sampling scheme.
   - Produce output for the estimation, using the class instance of '`MCMC`'.

The code is presented in parts, where a description for each part follows each code segment.

```
from numpy import log, ones, column_stack, hstack, mean
from numpy import random, zeros, sqrt
from pymcmc.mcmc import MCMC, RWMH, CFsampler
from pymcmc.regtools import LinearModel, CondScaleSampler
from pylab import plot, show
from ssm import Filter, SimSmoother
```

The first part of the program simply imports all the functions and classes that are used in the programs from their appropriate library. All of the functions imported in the first two rows are from the **NumPy** library. They are basic mathematical tools as well as tools to manipulate arrays. The third and forth rows import classes from the **PyMCMC** library. These are useful for the MCMC analysis. The fifth row imports functions that aid in plotting the output of the analysis. The sixth row imports the classes '`Filter`' and `SimSmoother` from **PySSM**.

```
def simdata(nobs):
  ht = 1.0 ** 2
  zt = 1.0
  tt = 0.95
  qt = 0.3 ** 2
  gt = 1.0
  rt = 1.0
  mu = 5.0
  a1 = mu
  p1 = qt / (1. - tt ** 2)
  beta = mu * (1.0 - tt)
  wmat = ones((1, 1,nobs))

  filt = Filter(nobs, 1, 1, 1, False, wmat = wmat)
  filt.initialise_system(a1, p1, zt, ht, tt, gt, qt, rt, beta = beta)
  filt.simssm()

  yvec = filt.get_ymat().T.flatten()
  simstate = filt.get_state()
  return yvec, simstate
```

The function `simdata` is used to simulate data from the model described in (6), (7) and (8). The function consists of three parts. The first part initializes the system matrices for the SSM.

Table 2 shows the relationship between the SSM in (1), (2) and (3); the model described in (6), (7) and (8); and the function `simdata`. The first column lists the system matrix, or parameter, as defined in (1), (2) and (3). The second column specifies the corresponding label in the code. The third column specifies the values of each system matrix or parameter used in simulating from the model in (1), (2) and (3), using the function `simdata`.

The second part of the function `simdata` instantiates an instance of the class '`Filter`', which is initialized and then used to simulate the data.

SSM	Label	Value	Additional notes
$\boldsymbol{H}_t$	`ht`	1.0	Refers to σ_ε^2, in (6).
$\boldsymbol{Z}_t$	`zt`	1.0	
$\boldsymbol{T}_t$	`tt`	0.95	Refers to ρ, in (6).
$\boldsymbol{Q}_t$	`qt`	0.3^2	Refers to σ_η^2, in (6).
$\boldsymbol{G}_t$	`gt`	1.0	
$\boldsymbol{R}_t$	`rt`	1.0	
$\boldsymbol{a}_1$	`a1`	5.0	Mean of the initial state, in (8).
$\boldsymbol{P}_1$	`p1`	$\frac{0.3^2}{1.0-0.95^2}$	Variance of the initial state, in (8).
$\boldsymbol{W}_t$	`wt`	1.0	Note that $\boldsymbol{W} = (\boldsymbol{W}_1; \boldsymbol{W}_2; \ldots; \boldsymbol{W}_n)$, where $\boldsymbol{W}_t = 1, \forall$t.
$\boldsymbol{\beta}$	`beta`	$5.0\,(1.0 - 0.95)$	

Table 2: Provides a mapping between the SSM in (1), (2) and (3); the model described in (6), (7) and (8); and the code snippet above.

The third part of the function `simdata` is used to obtain the observations (`yvec`), the state (`state`) and then return them to the main program.

```
def simstate(store):
  system = store['simsm'].get_system()
  system.update_tt(store['rho'])
  p1 = system._qt() / (1.0 - store['rho'] ** 2)
  a1 = store['beta'] / (1.0 - store['rho'])
  system.update_a1(a1)
  system.update_p1(p1)
  system.update_beta(store['beta'])
  state = store['simsm'].sim_smoother()
  return state
```

The function `simstate` is used to sample from the posterior distribution of the state in equation (4). This function computes Step 1, Algorithm 2. Most of the function is spent updating the system matrices based on the estimates of $\sigma_\varepsilon^2, \rho, \beta$ and σ_η from the previous iteration, in Steps 2 and 3, Algorithm 2. The object `store` is a dictionary that is passed into the function `simstate`, containing the latest draws of the parameters in the MCMC scheme, as well as classes that are required as a part of the MCMC estimation. For instance, `store['simsm']` is an instance of the class '`SimSmoother`', which is instantiated before the beginning of the MCMC scheme; note that this occurs latter in the code. As documented in Sections 3.2 and 3.3 the public member function `get_system()` returns an instance of the class '`System`', which can be used to access or update the system matrices for the SSM. An example of this is

```
  system.update_tt(store['rho'])
```

where the code is used to update the system matrix $\boldsymbol{T}_t$. Note that, `store['rho']` is the latest value for ρ. The key `'rho'` is defined when constructing the MCMC block used to sample ρ; note that this occurs in a latter point of the code. The rest of the updates follow in the same fashion. As described in Section 3.3, the member function `sim_smoother()` is used to sample the state, $\boldsymbol{\alpha}$, given the data and the system matrices.

```
def simht(store):
  system = store['simsm'].get_system()
  residual = store['simsm'].get_meas_residual()
  ht = store['scale_sampler'].sample(residual.T)
  system.update_ht(ht ** 2)
  return ht ** 2
```

The function `simht` is used to simulate from the conditional posterior distribution of σ_ϵ^2, following Algorithm 2, Step 2. As in the previous code segment, `store` is a dictionary that can be used to obtain the latest values from each of the iterates in the MCMC scheme, as well as functions and classes that are useful in the required calculations. As described in Section 3.3, the function `get_meas_residual()` can be used to obtain the residuals from the measurement equation, given the current value of the state, $\boldsymbol{\alpha}$, and the current value of $\boldsymbol{\theta}$. The class `store['scale_sampler']` is an instance of the class '`CondScaleSampler`' that is a part of the **PyMCMC** toolkit. Given the residuals from a linear component or model, the member function `sample()` draws from the posterior distribution of the associated scale term in the model. That is, in Algorithm 2, Step 2, `store['scale_sampler'].sample(residual.T)` is a draw from $p\left(\sigma_\varepsilon^2|\boldsymbol{y}, \boldsymbol{\alpha}, \rho, \sigma_\eta, \beta\right)$, given certain prior assumptions, which are detailed when setting up the class instance for '`CondScaleSampler`'. This will be further explained in a proceeding code segment.

Before describing the next code segment it will be helpful to describe the algorithm used in sampling from Step 3, Algorithm 2, in a little more detail. To sample from the posterior distribution of β, ρ and σ_η an independent MH algorithm is used; see Robert and Casella (1999) for technical details. The class that facilitates the implementation of the independent MH algorithm requires three functions. One function draws from a candidate density; note it is important that the candidate density closely approximates the target density, which in this case is the posterior distribution for β, ρ and σ_η. The second function evaluates and returns the log probability of the posterior, given particular values for β, ρ and σ_η. The last function evaluates and returns the log probability of the candidate density, given a particular value for β, ρ and σ_η. First we detail the set of functions used in evaluating the posterior density for β, ρ and σ_η.

```
def prior_rho(store):
  if store['rho'] < 1.0 and store['rho'] > 0.0:
    rho1 = 15.0
    rho2 = 1.5
    return (rho1 - 1.0) * log(store['rho']) + \
      (rho2 - 1.0) * log(1.0 - store['rho'])
  else:
    return -1E256
```

The function `prior_rho(store)` returns the log prior probability for ρ, up to a constant of proportionality. Here, it is assumed *apriori* that ρ follows a beta distribution. That is

$$p\left(\rho\right) \propto \rho^{(\rho_1 - 1)}(1-\rho)^{(\rho_2 - 1)}, \tag{9}$$

where, in this context, ρ_1 and ρ_2 are prior hyperparameters.

```
def prior_sigma(store):
  nu = 10.0
  S = 0.01
  sigma = -(nu + 1) * log(store['sigma']) - S / (2.0 * store['sigma'] ** 2)
  return sigma
```

The function `prior_sigma(store)` returns the log prior probability for σ_η. Here we have assumed an inverted gamma prior, which up to a constant of proportionality is

$$p(\sigma_\eta) \propto \sigma_\eta^{-(\nu+1)} \exp\left\{-\frac{S}{2\sigma_\eta^2}\right\}.$$

Noting that

$$\log p(\rho, \beta, \sigma_\eta|\alpha) = \text{constant} + \log p(\alpha|\rho, \beta, \sigma_\eta) + \log p(\rho) + \log p(\beta) + \log p(\sigma_\eta)$$

and given that we assume a flat prior for β, so it is absorbed into the constant, then we can write code for the function that evaluates the posterior for β, ρ and σ_η as follows:

```
def post_rho_sigma(store):
  if store['rho'] < 1.0 and store['rho'] > 0.0:
    system = store['simsm'].get_system()
    p1 = system._qt() / (1.0 - store['rho'] ** 2)
    a1 = store['beta'] / (1.0 - store['rho'])
    system.update_p1(p1)
    system.update_a1(a1)
    system.update_tt(store['rho'])
    system.update_beta(store['beta'])
    system.update_qt(store['sigma'] ** 2)
    lnpr =  store['simsm'].log_probability_state()
    return lnpr + prior_rho(store) + prior_sigma(store)
  else:
    return -1E256
```

The function `post_rho_sigma(store)` returns the $\log p(\beta, \rho, \sigma_\eta|\boldsymbol{\alpha})$. Most of the operations are simply updating the system matrices. Note that $p(\beta, \rho, \sigma_\eta|\boldsymbol{\alpha}) \propto p(\beta, \rho, \sigma_\eta|\boldsymbol{y}, \boldsymbol{\alpha})$, however, it is much simpler and more computationally efficient to work with $p(\beta, \rho, \sigma_\eta|\boldsymbol{\alpha})$.

```
def cand_rho_sigma(store):
  nobs = store['state'].shape[1]
  store['bayes_reg'].update_yvec(store['state'][0, 1:nobs])
  xmat = column_stack([ones(nobs - 1),store['state'][0, 0:nobs - 1]])
  store['bayes_reg'].update_xmat(xmat)
  sig,beta =  store['bayes_reg'].sample()
  return sig, beta[0], beta[1]
```

The function `cand_rho_sigma(store)` is used to sample from the candidate density, which we denote $q(\beta, \rho, \sigma_\eta|\boldsymbol{\alpha})$. To construct a good candidate, we note that the trend, $\alpha_{t+1} = \beta +$

$\rho\alpha_t + \eta_t$, is similar to a linear regression model. As such, we can force it into a linear regression framework and thus construct a candidate density, that closely resembles the posterior, is easy to sample from and also easy to evaluate up to a constant of proportionality. This is achieved by constructing a linear model as follows:

$$\tilde{\boldsymbol{y}} = \tilde{\boldsymbol{X}}\tilde{\boldsymbol{\beta}} + \tilde{\boldsymbol{\varepsilon}}, \tag{10}$$

where

$$\tilde{\boldsymbol{y}} = \begin{bmatrix} \alpha_2 \\ \alpha_3 \\ \vdots \\ \alpha_n \end{bmatrix}, \ \tilde{\boldsymbol{X}} = \begin{bmatrix} 1 & \alpha_1 \\ 1 & \alpha_2 \\ \vdots & \vdots \\ 1 & \alpha_{n-1} \end{bmatrix}, \ \tilde{\boldsymbol{\beta}} = \begin{bmatrix} \beta \\ \rho \end{bmatrix} \text{ and } \tilde{\boldsymbol{\varepsilon}} \sim \mathcal{N}\left(\mathbf{0}, \sigma_\eta^2 \boldsymbol{I}\right). \tag{11}$$

In the above code snippet `store['bayes_reg']`, which is an instance of the class '`LinearModel`' in the **PyMCMC** tool suite, is used in jointly sampling β and ρ. The class '`LinearModel`' is useful when analysing either linear models or models with linear components in a Bayesian analysis. The code in this snippet updates the values of $\tilde{\boldsymbol{X}}$ and $\tilde{\boldsymbol{y}}$, in lines 2, 3 and 4. Line 5 samples from the candidate density and line 6 returns the candidates for σ_η, β and ρ respectively.

```
def cand_prob(store):
  beta = hstack([store['beta'], store['rho']])
  return store['bayes_reg'].log_posterior_probability(
    store['sigma'], beta)
```

The function `cand_prob` evaluates the log candidate density, given β, ρ and σ_η. This is achieved by simply using the member function `log_posterior_probability(sigma, beta)`, which is a part of the '`LinearModel`' class.

```
def transform_beta(store):
  return store['beta'] / (1.0 - store['rho'])
```

The function `transform_beta(store)` is used so that μ can be reported instead of β. The function is called from the MCMC algorithm, and demonstrates how easy reparameterizations are using **PyMCMC**. This can be particularly important for achieving simulation efficient MCMC sampling schemes; see for example Frühwirth-Schnatter (2004) and Strickland, Martin, and Forbes (2008).

```
random.seed(12345)
nobs = 1000
nstate = 1
rstate = 1
yvec, simulated_state = simdata(nobs)
data = {'yvec':yvec}
```

This code snippet simulates data using the function `simdata` described above, with 1000 observations. The last line creates the dictionary `data`, which is used to store information that the user wants to access using functions that are called from the MCMC scheme. The

object `data` is passed into the main class that facilitates MCMC estimation and its elements are accessible through the dictionary `store` that is passed into each of the relevant functions; see for example the code snippet that specifies the function `transform_beta` that is listed above.

```
ht = 1.0
tt = 0.9
zt = 1.0
qt = 0.4
gt = 1.0
rt = 1.0
p1 = qt / (1. - tt ** 2)
mu = mean(yvec)
a1 = mu
beta = mu * (1 - tt)
wmat = 1.0
```

The above code snippet is used to define initial values for the system matrices that are used to initialize the class '`SimSmoother`'.

```
data['simsm'] = SimSmoother(yvec, nstate, rstate, False,
  properties = {'gt':'eye'}, wmat = wmat)
data['simsm'].initialise_system(a1, p1, zt, ht, tt, gt, qt, rt, beta = beta)
```

The code above demonstrates the creation of the class instance for '`SimSmoother`', which is defined as `data['simsm']`. After creating the class instance, it is initialized using the member function `initialise_system(...)`.

```
data['scale_sampler'] = CondScaleSampler(
  prior = ['inverted_gamma', 10.0, 0.01])
data['bayes_reg'] = LinearModel(yvec[1:nobs],
  column_stack([ones(nobs - 1), yvec[0:nobs-1]]))
```

Here instances of the classes '`CondScaleSampler`' and '`LinearModel`' are created. Recall that the class instance for '`CondScaleSampler`' is used in the function `simht(store)` to sample σ_ε^2 and that the class instance for '`LinearModel`' is used in the functions `post_rho_sigma(store)`, `cand_rho_sigma(store)` and `cand_prob(store)`, which are used in sampling β, ρ and σ_η. The prior for σ_η is defined as

$$p(\sigma_\eta) \propto \sigma_\eta^{(\nu+1)} \exp\left\{-\frac{0.01}{2\sigma_\eta^2}\right\}.$$

```
samplestate = CFsampler(simstate, zeros((nstate, nobs)), 'state')
```

The code above is used to create a class instance of '`CFsampler`', which defines the block of the MCMC algorithm that is used to sample the state, $\boldsymbol{\alpha}$. The class '`CFsampler`' is a part of the **PyMCMC** toolkit and is used when defining blocks of an MCMC algorithm where a closed form solution is being used to sample from the posterior distribution of interest. The

first argument for the class defines the function that returns a sample from the posterior distribution of interest, which in this case is `simstate`. The second argument in the class defines the initial values for the parameter(s) being sampled. Here we have initialized the state with an $(m \times n)$ array of zeros. Note that, as the sampling scheme is set up (code that follows) so that the state is sampled first, we do not require a sensible initialization. The third argument defines the `key` that can be used to access the array that stores the state in the MCMC sampling scheme; see for instance the function `cand_rho_sigma(store)`.

```
sampleht = CFsampler(simht, ht, 'ht')
```

The code above is used to instantiate the class instance `sampleht`, which defines the block in the MCMC sampling scheme used to sample σ_ε^2. We are able to draw from the posterior of σ_ε^2 using a closed form solution, and consequently we use the class '`CFsampler`' to define this block of the MCMC sampling scheme. Note that, `simht` is the function that we defined earlier, which is used to sample from the posterior of σ_ε^2; `ht` is the value used to initialize the MCMC sampling scheme and `'ht'` is the key used to access σ_ε^2.

```
samplesigbetarho = IndMH(cand_rho_sigma, post_rho_sigma,
  cand_prob, [sqrt(qt), 0.1, tt], ['sigma', 'beta', 'rho'])
```

The class instance `samplesigbetarho` defines the block in the MCMC sampling scheme relating to sampling β, ρ and σ_η^2. In this case we do not have a closed form solution, and are using the independent MH algorithm. To define this block for our MCMC sampler we used the class '`IndMH`', which is a part of the **PyMCMC** library. Note that three functions `cand_rho_sigma`, `post_rho_sigma` and `cand_prob` have been defined earlier and are the functions that sample from the candidate density, evaluate the log posterior density and evaluate the log candidate density, for β, ρ and σ_η. The list `[sqrt(qt), 0.01, tt]` defines the initial values and the list `['sig', 'beta' 'rho']` defines the keys for each of σ_η, β and ρ, respectively.

```
blocks1 = [samplestate, sampleht, samplesigbetarho]
mcmc = MCMC(5000, 2000, data, blocks1, transform = {'beta': transform_beta})
```

This code segment is used to define the blocking scheme and instantiate the class instance for `MCMC`, which is the engine for MCMC analysis and is defined in **PyMCMC**. Note that the order of the blocking scheme is defined by the order of the blocks in `blocks1`. The first argument in `MCMC` defines that the MCMC sampling scheme will be run for 5000 iterations and the second argument denotes that the first 2000 iterations will be discarded as burnin. The third argument is the dictionary data that stores any user defined functions, classes, or data, which are required by any of the functions that are called by the MCMC sampling scheme; note these can be accessed using the dictionary `store` for any of the functions above. The fourth argument `blocks1`, defines the blocking scheme for the MCMC sampling scheme and the (optional) fifth argument transforms the iterates for β, as defined by the function `transform_beta`, which are used in calculation of the output. Note that when accessing β using `store` in any of the functions called as a part of the sampling scheme, the sample value rather than the transformed value for β is obtained.

```
mcmc.sampler()
mcmc.output(parameters = ['ht', 'sigma', 'rho', 'beta'])
```

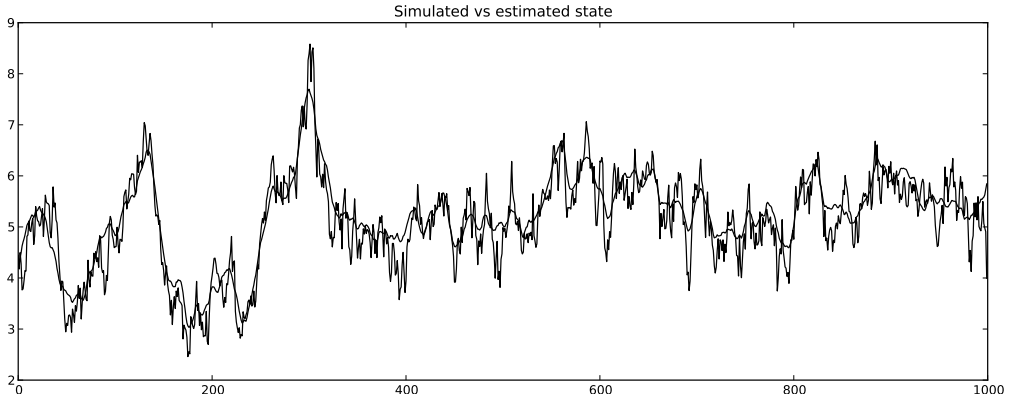

Figure 1: Plot of the marginal posterior mean of the state against the simulated state.

In the code segment above, the function `sampler()` is used to run the MCMC sampling scheme. The function `output` is used here to produce some generic output for σ_ε^2, σ_η, ρ and β.

```
The time (seconds) for the MCMC sampler = 43.47
Number of blocks in MCMC sampler =  3

                 mean           sd         2.5%        97.5%      IFactor
        beta      5.2        0.252         4.71         5.69         4.16
       sigma    0.195       0.0208        0.157        0.235         69.3
         rho    0.972      0.00839        0.956        0.988         7.37
          ht     1.04       0.0566        0.927         1.15         4.43
Acceptance rate  beta  =  0.7382
Acceptance rate  sigma  =  0.7382
Acceptance rate  rho  =  0.7382
Acceptance rate  ht  =  1.0
```

Here we can see the total time for estimation is around 40 seconds. The marginal posterior mean estimates for each of the parameters β, σ_η, ρ and σ_ε^2, appear to be reasonably accurate, given the true values for the simulated data.

```
means, vars = mcmc.get_mean_var('state')
plot(range(nobs), means[0], range(nobs), simulated_state[0])
show()
```

The code snippet above is used to plot the marginal posterior mean estimate for the state, against its simulated value (Figure 1). The member function `get_mean_var(name)` returns the marginal posterior mean and variance estimates for the specified parameter, which in this case is the state.

4.2. Example 2: Spline smoothing

The second example can be found in `example_ssm_vector_spline.py`. This study uses a cu-

bic smoothing spline to model motorcycle acceleration data. The data set is used in Durbin and Koopman (2001) and can be found on the book's website `http://www.ssfpack.com/DKbook.html`. The state space form for a cubic smoothing spline is well known; see for example Carter and Kohn (1994) and Durbin and Koopman (2001). The measurement equation for the cubic smoothing spline is as follows:

$$y_t = \mu_t + \varepsilon_t, \tag{12}$$

where the disturbance ε_t is distributed normal with a mean 0 and variance h. The latent variable μ_t is defined as

$$\begin{aligned} \mu_{t+1} &= \mu_t + \delta_t \beta_t + \zeta_t \\ \beta_{t+1} &= \beta_t + \xi_t, \end{aligned} \tag{13}$$

where $\delta_t = \tau_{t+1} - \tau_t$, with τ_t being the time of the tth observation. The disturbance terms are jointly distributed such that

$$\begin{bmatrix} \zeta_t \\ \xi_t \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \sigma^2 \begin{bmatrix} \frac{1}{3}\delta_t^3 & \frac{1}{2}\delta_t^2 \\ \frac{1}{2}\delta_t^2 & \delta_t \end{bmatrix} \right). \tag{14}$$

Further, it is assumed that

$$\begin{bmatrix} \mu_1 \\ \beta_1 \end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \kappa I_2 \right), \tag{15}$$

where $\kappa \to \infty$. Expressing (12), (13), (14) and (15) in state space form, using (1), (2) and (3) is achieved by defining
$\boldsymbol{\alpha}_t = \begin{bmatrix} \mu_t \\ \beta_t \end{bmatrix}$, $\boldsymbol{\eta}_t = \begin{bmatrix} \zeta_t \\ \xi_t \end{bmatrix}$, $\boldsymbol{T}_t = \begin{bmatrix} 1 & \delta_t \\ 0 & 1 \end{bmatrix}$, $\boldsymbol{Q}_t = \begin{bmatrix} \frac{1}{3}\delta_t^3 & \frac{1}{2}\delta_t^2 \\ \frac{1}{2}\delta_t^2 & \delta_t \end{bmatrix}$, $\boldsymbol{G}_t = \sigma \boldsymbol{I}_2$, $\boldsymbol{Z}_t = \begin{bmatrix} 1 & 0 \end{bmatrix}$, $\boldsymbol{H}_t = h$, $\boldsymbol{R}_t = 1$, $\boldsymbol{\beta} = 0$, $\boldsymbol{a}_1 = \boldsymbol{0}$ and $\boldsymbol{P}_1 = \kappa \boldsymbol{I}_2$, where $\kappa \to \infty$.
An MCMC algorithm for this example is as follows:

1. Sample $\boldsymbol{\alpha}^j$ from $p\left(\boldsymbol{\alpha}|\boldsymbol{y}, h^{j-1}, \sigma^{j-1}\right)$.
2. Sample h^j from $p\left(h|\boldsymbol{y}, \boldsymbol{\alpha}^j, \sigma^{j-1}\right)$.
3. Sample σ^j from $p\left(\sigma|\boldsymbol{y}, h^j\right)$.

Algorithm 3: MCMC algorithm for cubic smoothing spline.

In Step 1, Algorithm 3, we sample the state, $\boldsymbol{\alpha}$, from its full conditional posterior distribution in (4). This step is accomplished with the class '`SimSmoother`' in **PySSM**. Step 2, Algorithm 3, we sample h from its full conditional posterior distribution. This step makes use of the class '`CondScaleSampler`' in **PyMCMC**. In Step 3, we sample σ from its posterior distribution, marginal of the state, $\boldsymbol{\alpha}$.

The program used for estimation, following Algorithm 3 is detailed below. The layout of the program can be summarized with the following steps.

1. Import functions and classes. This step is skipped in the detailed description below to avoid repetition.

2. Define function to update the system matrix $\boldsymbol{G}_t$.
3. Define function to sample from the posterior distribution of the state, $\boldsymbol{\alpha}$. This function is used in Step 1, Algorithm 3
4. Define function to sample from the posterior distribution of h, which is required in Step 2, Algorithm 3.
5. Define functions for the posterior and prior for σ. These functions are used in Step 3, Algorithm 3.
6. Start main program.
    - Load data.
    - Initialize system matrices for MCMC scheme.
    - Define a class instance of '`SimSmoother`'. Note this is used in Step 3 of the program.
    - Define a class instance of '`CondScaleSampler`'. Note this is used in Step 4 of the program.
    - Define the objects `samplestate`, `sample_ht` and `sample_sigma`, which define the blocks of the MCMC sampling scheme.
    - Define a class instance of '`MCMC`' and run the MCMC sampler.
    - Produce MCMC output.

For the code description here, we skip the importing of the required libraries. To avoid, repetition, the descriptions for this example are far less detailed than the first example, in Section 4.1.

```
def update_gt(store):
  system = store['simsm'].get_system()
  gt = eye(2) * store['sigma']
  system.update_gt(gt)
```

The function `update_gt(store)` is used to update the system matrix $\boldsymbol{G}_t$, given the latest iterates of the parameters in the MCMC scheme. Note that `store['simsm']` is a class instance of the class '`SimSmoother`'.

```
def simstate(store):
  update_gt(store)
  state = store['simsm'].sim_smoother()
  return state
```

The function `simstate` is used to sample from the posterior distribution of the state, $\boldsymbol{\alpha}$. This corresponds to Step 1, Algorithm 3.

```
def sim_ht(store):
  residual = store['simsm'].get_meas_residual()
  sqrtht = store['scale_sampler'].sample(residual.T)
  system = store['simsm'].get_system()
  system.update_ht(sqrtht ** 2)
  return ht
```

The function `sim_ht(store)` is used to sample from the posterior distribution of h in Step 2, Algorithm 3. Note that `store['scale_sampler']` is an instance of class '`CondScaleSampler`'.

```
def post_sigma(store):
  if store['sigma'] > 0:
    update_gt(store)
    lnpr = store['simsm'].log_likelihood()
    #lnpr = store['simsm'].log_probability_state(diffuse = True)
    return lnpr + prior_sigma(store['sigma'])
  else:
    return -1E256
```

The function `post_sigma(store)` returns the log posterior probability for σ, marginal of the state, $\boldsymbol{\alpha}$. This function is defined to facilitate the use of a random walk MH algorithm to sample σ, in Step 3, Algorithm 3. Note that, if we use the member function `log_probability_state` rather than `log_likelihood()` the sample of σ will be conditional on the state. Sampling marginally of the state improves the mixing of the MCMC sampling scheme, however, calculating the log likelihood is computationally more expensive than calculating the log posterior probability of the state. If we implemented the second approach, that is we used `log_probability_state(diffuse = True)` then the optional argument `diffuse = True` is required here as we are using diffuse initial conditions; that is we are assuming $\boldsymbol{P}_1 = \kappa \boldsymbol{I}_2$, where $\kappa \to \infty$.

```
def prior_sigma(sigma):
  nu = 10
  S = 0.1
  return -(nu + 1) * log(sigma) - 0.5 * S / sigma ** 2
```

The function `prior_sigma` returns the log prior probability for σ, where it is assumed `apriori` that σ is distributed following the inverted gamma distribution.

```
random.seed(1234)
data = loadtxt('motorcycle.txt')
data = data[data[:, 1] > 0]
yvec = data[:, 2]
delta = data[:, 1]
nobs = yvec.shape[0]
nstate = 2
rstate = 2
data = {'yvec': yvec}
```

The code snippet above loads the data. Note that, the observations are labeled `yvec` and the time between observations are labeled `delta`.

```
sigeps = 0.9
ht = sigeps ** 2
rt = 1.
zt = array([[1.0, 0.0]])
```

```
tt = zeros((2, 2, nobs))
tt[0, 0, :] = 1.
tt[1, 1, :] = 1.
tt[0, 1, :] = delta
sigma = 0.3
qt = zeros((2, 2, nobs))
qt[0, 0, :] = delta ** 3 / 3.
qt[0, 1, :] = qt[1, 0, :] = delta ** 2 / 2
qt[1, 1, :] = delta
gt = eye(nstate) * sigma
a1 = zeros(2)
p1 = eye(2)
wmat = zeros((2, 2, nobs))
```

The code above initializes the required system matrices. Note that, as we use diffuse initial conditions `a1` is only a dummy argument in this case. Further, setting `wmat` to zeros is required here as diffuse initial conditions are dealt with using the simulation smoother of De Jong and Shephard (1995) and no specialization is incorporated to differentiate between a diffuse prior on $\boldsymbol{\beta}$ for a model with or without regressors. As a consequence, by setting `wmat = 0`, that is $\boldsymbol{W}_t = \mathbf{0}$ for $t = 1, 2, \ldots, n$, it ensures that $\boldsymbol{\beta}$, which has a diffuse prior on it, is only entering the model through the mean of the initial state as, $\boldsymbol{a}_1 = \boldsymbol{W}_0\boldsymbol{\beta}$.

```
data['simsm'] = SimSmoother(yvec, nstate, rstate,
  timevar = {'tt': True, 'qt': True}, properties = {'rt': 'eye'},
  wmat = wmat, joint_sample = ['diffuse', eye(2)])
data['simsm'].initialise_system(a1, p1, zt, ht, tt, gt, qt, rt)
data['scale_sampler'] = CondScaleSampler(
  prior = ['inverted_gamma', 10, 0.1])
```

The above code snippet instantiates the classes '`SimSmoother`' and the `CondScaleSampler`.

```
samplestate = CFsampler(simstate, zeros((nstate, nobs)), 'state',
  store = 'none')
sample_ht = CFsampler(sim_ht, 0.3, 'ht')
sample_sigma = RWMH(post_sigma, 0.05, 0.003, 'sigma', adaptive = 'GFS')
```

The main point to note in the code above is that we have used the class '`RWMH`'. This is a class in **PyMCMC** that is used to facilitate estimation using the random walk MH algorithm. The first argument defines the posterior density; the second argument defines the initial value for the scale that defines the size of the step in the algorithm; the third argument defines the initial value for the the parameter of interest; the fourth argument defines the `key` for σ and the optional argument `adaptive = 'GFS'` means the adaptive algorithm of Garthwaite, Fan, and Scisson (2010) is used to compute the step size of the random walk MH algorithm.

```
blocks = [samplestate, sample_ht, sample_sigma]
mcmc = MCMC(8000, 3000, data, blocks, runtime_output = True)
mcmc.sampler()
mcmc.output(parameters = ['ht', 'sigma'])
```

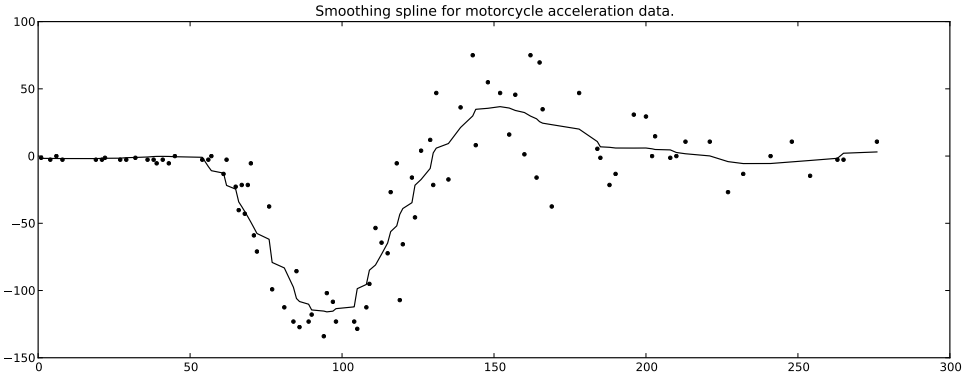

Figure 2: Plot of the motorcycle acceleration data, with the smoothing spline superimposed over it.

The block of code above defines the blocking scheme and initializes and runs the MCMC sampling scheme. The MCMC sampler is run for 8000 iterations, where the first 3000 are discarded as burnin. The optional argument `runtime_output = True` simply produces output at runtime, specifically timing information, including a progress bar, the time remaining, total time and time taken so far.

```
The time (seconds) for the MCMC sampler = 25.71
Number of blocks in MCMC sampler =  3

                 mean          sd         2.5%        97.5%      IFactor
       sigma     0.459       0.087        0.317        0.648          7.3
          ht       462        71.6          332          602         3.85
Acceptance rate  sigma  =  0.4555
Acceptance rate  ht  =  1.0
```

For σ and h, the marginal posterior mean, marginal posterior standard deviation, 2.5% and 97.5% credible interval and IFs are reported. The total time for estimation is about 26 seconds. The output plot is given in Figure 2

```
means, vars = mcmc.get_mean_var('state')
plot(yvec, '.', color = 'k')
title('Smoothing spline for motorcycle acceleration data.')
plot(means[0], color = 'k')
savefig('spline.pdf')
```

4.3. Example 3: Trend plus cycle model

The third example is for a multivariate time series, where it is assumed there is a common trend and cycle. This model is used to analyze the data set considered by Strickland *et al.* (2009). It can be found in `example_ssm_trendcycle2.py`. The model is defined such that the $(p \times 1)$ vector of observations, $\boldsymbol{y}_t$, for $t = 1, 2, \ldots, n$, is generated by

$$\boldsymbol{y}_t = \boldsymbol{\mu}_t + \boldsymbol{\psi}_t + \boldsymbol{\varepsilon}_t; \;\; \boldsymbol{\varepsilon}_t \sim \mathcal{N}\left(\mathbf{0}, \boldsymbol{\Sigma}\right),$$

where $\boldsymbol{\mu}_t$ is a $(p \times 1)$ vector of common trends, $\boldsymbol{\psi}_t$ is a $(p \times 1)$ vector of common cycles and $\boldsymbol{\varepsilon}_t$ is normally distributed, with a mean vector $\mathbf{0}$ and a covariance $\boldsymbol{\Sigma}$. Further, it is assumed that $\mu_{1,t} = \mu_{2,t} = \cdots = \mu_{p,t}$ and $\psi_{1,t} = \psi_{2,t} = \cdots = \psi_{p,t}$, for $t = 1, 2, \ldots, n$. The trend component is assumed to evolve according to

$$\begin{aligned} \boldsymbol{\mu}_{t+1} &= \boldsymbol{\mu}_t + \boldsymbol{\delta}_t + \boldsymbol{\nu}_t; \;\; \boldsymbol{\nu}_t \sim \mathcal{N}\left(\mathbf{0}, \sigma_\nu^2 I_p\right), \\ \boldsymbol{\delta}_{t+1} &= \boldsymbol{\delta}_t. \end{aligned}$$

The stochastic cycle is assumed to evolve according to

$$\begin{bmatrix} \boldsymbol{\psi}_{t+1} \\ \boldsymbol{\psi}^*_{t+1} \end{bmatrix} = \boldsymbol{\Gamma} \begin{bmatrix} \boldsymbol{\psi}_{t+1} \\ \boldsymbol{\psi}^*_{t+1} \end{bmatrix} + \boldsymbol{\omega}_t,$$

where $\boldsymbol{\psi}^*_t$ is a $(p \times 1)$ vector of auxiliary variables, $\boldsymbol{\Gamma} = \boldsymbol{I}_p \otimes \boldsymbol{\Gamma}_\psi$, with $\boldsymbol{\Gamma}_\psi = \rho \begin{bmatrix} \cos\lambda & \sin\lambda \\ -\sin\lambda & \cos\lambda \end{bmatrix}$ and ρ is a persistence parameter that is restricted such that $|\rho| < 1$. The disturbance vector $\boldsymbol{\omega}_t = \boldsymbol{\iota}_p \otimes \boldsymbol{\kappa}_t$, with $\kappa_t \sim \mathcal{N}\left(\mathbf{0}, \sigma^2 \boldsymbol{I}_2\right)$, where $\boldsymbol{\iota}_p$ is a $(p \times 1)$ vector of ones.

This model can be compactly reformulated as an SSM with the following parameters:

$$\boldsymbol{\alpha}_t = \begin{bmatrix} \mu_t & \delta_t & \psi_t & \psi^*_t \end{bmatrix}^\top, \; \boldsymbol{\eta}_t = \begin{bmatrix} \nu_t & \xi_t & \xi^*_t \end{bmatrix}^\top, \; \boldsymbol{T}_t = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \boldsymbol{\Gamma}_\psi \end{bmatrix},$$

$$\boldsymbol{\Gamma}_\psi = \rho \begin{bmatrix} \cos(\lambda) & \sin(\lambda) \\ -\sin(\lambda) & \cos(\lambda) \end{bmatrix},$$

$$\boldsymbol{Q}_t = \begin{bmatrix} \sigma_\nu^2 & 0 & 0 \\ 0 & \sigma_\kappa^2 & 0 \\ 0 & 0 & \sigma_\kappa^2 \end{bmatrix}, \; \boldsymbol{G}_t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \; \boldsymbol{Z}_t = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ . & . & . & . \\ . & . & . & . \\ 1 & 0 & 1 & 0 \end{bmatrix}, \; \boldsymbol{H}_t = \boldsymbol{\Sigma} \text{ and } \boldsymbol{R}_t = \boldsymbol{I}_p$$

A proper prior is assumed for the initial state, with

$$\boldsymbol{a}_1 = \begin{bmatrix} 7 \\ 0 \\ 0 \\ 0 \end{bmatrix} \text{ and } \boldsymbol{P}_1 = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & \frac{\sigma_\kappa^2}{1-\rho^2} & 0 \\ 0 & 0 & 0 & \frac{\sigma_\kappa^2}{1-\rho^2} \end{bmatrix}.$$

An MCMC algorithm for this example is as follows:

1. Sample $\boldsymbol{\alpha}^j$ from $p\left(\boldsymbol{\alpha}|\boldsymbol{y}, \boldsymbol{\Sigma}^{j-1}, \sigma_\nu^{j-1}, \sigma_\kappa^{j-1}, \rho^{j-1}, \lambda^{j-1}\right)$
2. Sample $\boldsymbol{\Sigma}^j$ from $p\left(\boldsymbol{\Sigma}|\boldsymbol{y}, \boldsymbol{\alpha}^j, \sigma_\nu^{j-1}, \sigma_\kappa^{j-1}, \rho^{j-1}, \lambda^{j-1}\right)$
3. Sample σ_ν^j from $p\left(\sigma_\nu|\boldsymbol{y}, \boldsymbol{\alpha}^j, \boldsymbol{\Sigma}^j, \sigma_\kappa^{j-1}, \rho^{j-1}, \lambda^{j-1}\right)$
4. Sample σ_κ^j from $p\left(\sigma_\kappa|\boldsymbol{y}, \boldsymbol{\alpha}^j, \boldsymbol{\Sigma}^j, \sigma_\nu^j, \rho^{j-1}, \lambda^{j-1}\right)$
5. Sample ρ^j from $p\left(\rho|\boldsymbol{y}, \boldsymbol{\alpha}^j, \boldsymbol{\Sigma}^j, \sigma_\nu^j, \sigma_\kappa^j, \lambda^{j-1}\right)$
6. Sample λ^j from $p\left(\lambda|\boldsymbol{y}, \boldsymbol{\alpha}^j, \boldsymbol{\Sigma}^j, \sigma_\nu^j, \sigma_\kappa^j, \rho^j\right)$

Algorithm 4: MCMC algorithm for the trend cycle model.

In Step 1, Algorithm 4 the state, $\boldsymbol{\alpha}$ is sampled from its posterior distribution. The class '`SimSmoother`' is used in the step. In Steps 2 and 3 the covariance matrix $\boldsymbol{\Sigma}$ and the scale parameter σ_ν are drawn from their posterior distributions, respectively. In both cases the class '`CondScaleSampler`' is used. Steps 4, 5 and 6 sample the parameters σ_κ, ρ and λ from their posterior distributions, respectively. In sampling each of these parameters the class '`RWMH`', from the package **PyMCMC**, is used.

The program used to implement Algorithm 4 is described below. A brief summary of the program is as follows:

1. Import system matrices. Note that in the description below. The initial importing of libraries is omitted for brevity.
2. Define a function to update the system matrix $\boldsymbol{T}_t$.
3. Define a function to update $\boldsymbol{P}_1$.
4. Define a function to update the covariance matrix $\boldsymbol{Q}_t$.
5. Define a function that draws from the posterior distribution of the state. This function is used in Step 1, Algorithm 4.
6. Define a function that draws from the posterior distribution of $\boldsymbol{\Sigma}$.
7. Define a function that draws from the posterior distribution of the state. This function is used in Step 2, Algorithm 4.
8. Define a function that is used to sample from the posterior distribution of σ_ν. This function is used in Step 3, Algorithm 4.
9. Define prior and posterior functions for σ_κ, ρ and λ. These functions are used in Steps 4, 5 and 6, Algorithm 4.
    - Load data.

- Initialize system matrices.
- Define a class instance for '`SimSmoother`'.
- Define a class instance of '`CondScaleSampler`', with a Wishart prior.
- Define a class instance of '`CondScaleSampler`', with an inverted gamma prior.
- Define objects `samplestate`, `sampleht`, `samplesig_level`, `samplesig_cycle`, `samplerho` and `samplelambda` that define the MCMC sampling scheme.
- Define a class instance of '`MCMC`' and run the sampling scheme.
- Produce the MCMC output.

The code is presented in parts, where a brief description is presented below describing each code segment. The descriptions are far less detailed than in Section 4.1, to avoid repetition.

```
def update_tt(store):
  system = store['simsm'].get_system()
  tt = system.tt()
  tt[2, 2] = store['rho'] * cos(store['lambda'])
  tt[2, 3] = store['rho'] * sin(store['lambda'])
  tt[3, 2] = -tt[2, 3]
  tt[3, 3] = tt[2, 2]
  system.update_tt(tt)

def update_p1(store):
  system = store['simsm'].get_system()
  p1 = system.p1()
  p1[2, 2] = store['sigma_cycle'] ** 2 / (1. - store['rho'] ** 2)
  p1[3, 3] = p1[2, 2]
  system.update_p1(p1)

def update_qt(store):
  system = store['simsm'].get_system()
  qt = zeros((3, 3))
  qt[0, 0] = store['sigma_level'] ** 2
  qt[1, 1] = store['sigma_cycle'] ** 2
  qt[2, 2] = qt[1, 1]
  system.update_qt(qt)
```

The code above contains update functions for $\boldsymbol{T}_t$, $\boldsymbol{P}_1$ and $\boldsymbol{Q}_t$, respectively.

```
def simstate(store):
  update_tt(store)
  update_qt(store)
  update_p1(store)
  return  store['simsm'].sim_smoother()
```

The function `simstate` is used to draw from the posterior distribution of the state, $\boldsymbol{\alpha}$. This is required for Step 1, Algorithm 4.

```
def simht(store):
  residual = store['simsm'].get_meas_residual()
  ht = linalg.inv(store['scale_sampler'].sample(residual.T))
  system = store['simsm'].get_system()
  system.update_ht(ht)
  return ht

def simsig_level(store):
  update_tt(store)
  residual = store['simsm'].get_state_residual(state_index = [0])
  sigma_level =  store['scale_sampler2'].sample(residual.T)
  return sigma_level
```

The functions `simht(store)` and `simsig_level(store)` are designed to draw from the posterior distributions of Σ and σ_ν, respectively.

```
def posterior_sig_cycle(store):
  if store['sigma_cycle'] > 0:
    update_tt(store)
    update_qt(store)
    update_p1(store)
    probstate = store['simsm'].log_probability_state()
    #probstate = store['simsm'].log_likelihood()
    return probstate + prior_sig(store['sigma_cycle'])
  else:
    return -1E256

def prior_sig(sig):
  nu = -1.
  S = 0.0
  return -(nu + 1) * log(sig) - S / (2.0 * sig ** 2)
```

The functions `posterior_sig_cycle` and `prior_sig` evaluate the posterior distribution and the prior distribution for σ_κ, respectively. As in the previous example, using the log likelihood in the computation of the posterior distribution is straightforward and a valid alternative to computing the log probability of the state.

```
def posterior_lambda(store):
   update_tt(store)
   lnpr = store['simsm'].log_probability_state()
   #lnpr = store['simsm'].log_likelihood()
   return lnpr + prior_lambda(store)

def prior_lambda(store):
  if store['lambda'] > pi / 20. and store['lambda'] < 2 * pi / 4.:
    return 0.0
  else:
    return -1E256
```

The functions `posterior_lambda` and `prior_lambda` evaluate the log posterior and log prior probabilities for λ, respectively. The prior for λ is a uniform prior restricting the period of the cycle to be between 10 and 14 months.

```
def posterior_rho(store):
  if store['rho'] > 0 and store['rho'] < 1.0:
    update_tt(store)
    update_p1(store)
    lnpr = store['simsm'].log_probability_state()
    #lnpr = store['simsm'].log_likelihood()
    return lnpr + prior_rho(store)
  else:
    return -1E256

def prior_rho(store):
  rho1 = 15.0
  rho2 = 1.5
  rho = (rho1 - 1.0) * log(store['rho']) + \
    (rho2 - 1.) * log(1. - store['rho'])
  return rho
```

The functions `posterior_rho(store)` and `prior_rho(store)` evaluate the log posterior and log prior probabilities for ρ, respectively. Note that a beta prior is defined for ρ, which is identical to (9).

```
random.seed(12345)
filename = 'farmb.txt'
ymat = loadtxt(filename).T / 1000.
nseries, nobs = ymat.shape data = {'ymat': ymat}
```

The code above is used to load the data set.

```
nstate = 4
rstate = 3
ht = eye(nseries)
zt = column_stack([ones(nseries), zeros(nseries),
  ones(nseries), zeros(nseries)])
rt = ones(nseries)
tt = zeros((nstate, nstate))
rho = 0.9
lamb = 2. * pi / 20.
tt[0, 0] = 1.0
tt[0, 1] = 1.0
tt[1, 1] = 1.0
tt[2, 2] = rho * cos(lamb)
tt[2, 3] = rho * sin(lamb)
tt[3, 2] = -tt[2, 3]
tt[3, 3] = tt[2, 2]
```

```
sig_cycle = 0.3
sig_level = 0.3
qt = diag(array([sig_level, sig_cycle, sig_cycle]) ** 2)
gt = zeros((nstate, rstate))
gt[0, 0] = 1.0
gt[2:,1:] = eye(2)
a1 = array([7.5,0.000, 0.0, 0.0])
p1 = zeros((nstate, nstate))
p1[0,0] = 10.
p1[1, 1] = 10.
p1[2, 2] = qt[2, 2] / (1 - rho ** 2)
p1[3, 3] = p1[2, 2]
```

The code above is used to initialize the system matrices for the MCMC analysis.

```
timevar = False
data['simsm'] = SimSmoother(ymat, nstate, rstate, timevar,
  properties = {'rt': 'eye'})
data['simsm'].initialise_system(a1, p1, zt, ht, tt, gt, qt, rt)
```

The code snippet above instantiates and initializes the class `SimSmoother`.

```
prior_wishart = ['wishart', 10 * ones(nseries), 0.1 * eye(nseries)]
data['scale_sampler'] = CondScaleSampler(prior = prior_wishart)
data['scale_sampler2'] = CondScaleSampler(
  prior = ['inverted_gamma', 10, 0.1])
```

Here, two instances of '`CondScaleSampler`' are initialized. The first is defined assuming a Wishart prior, where this instance is used in sampling $\mathbf{\Sigma}$ from its posterior distribution. The second is defined for an inverted gamma distribution, which is used in sampling σ_ν.

```
samplestate = CFsampler(simstate, zeros((nstate, nobs)), 'state',
  store = 'none')
sampleht = CFsampler(simht, ht, 'ht')
samplesig_level = CFsampler(simsig_level, sig_level, 'sigma_level')
samplesig_cycle = RWMH(posterior_sig_cycle, 0.05, sig_cycle ** 2,
  'sigma_cycle', adaptive = 'GFS')
samplerho = RWMH(posterior_rho, 0.09, rho, 'rho', adaptive = 'GFS')
samplelambda = RWMH(posterior_lambda, 1.03, lamb, 'lambda',
  adaptive = 'GFS')
```

The class instances `samplestate`, `sampleht`, `samplesig_level` and `samplesig_cycle` define the blocks of the MCMC scheme, for Step 1, Step 2, ..., Step 6, Algorithm 4, respectively.

```
blocks = [samplestate, sampleht, samplesig_cycle, samplesig_level,
  samplerho, samplelambda]
mcmc = MCMC(8000, 3000, data, blocks)
mcmc.sampler()
mcmc.output(parameters=['rho', 'lambda', 'sigma_level', 'sigma_cycle'])
```

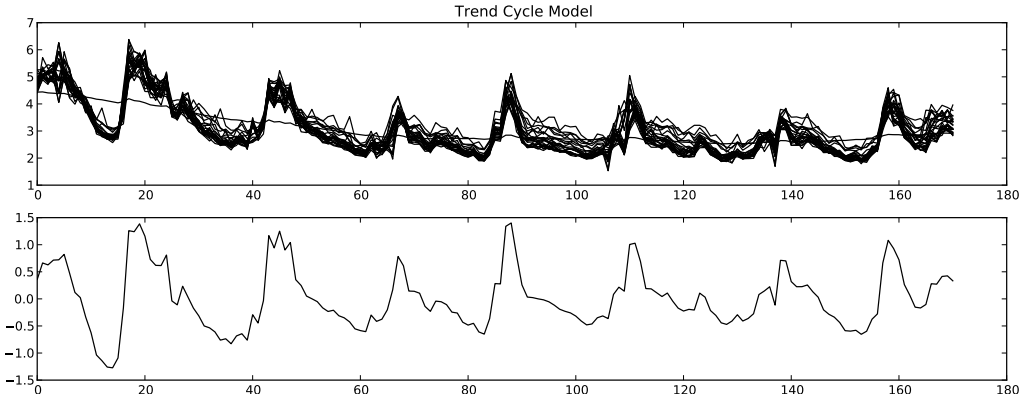

Figure 3: The top plot superimposes the marginal posterior mean estimate for the trend over the data. The second plot is of the marginal posterior mean estimate of the stochastic cycle.

The code above defines and runs the MCMC sampling scheme, and produces standard output for the parameters ρ, λ, σ_ν and σ_κ.

```
The time (seconds) for the MCMC sampler = 103.73
Number of blocks in MCMC sampler =  6

                  mean          sd         2.5%       97.5%      IFactor
 sigma_cycle      0.27      0.0233        0.225       0.314         31.4
 sigma_level     0.105      0.0219       0.0669       0.148         32.2
         rho     0.897      0.0273        0.845        0.95         9.49
      lambda     0.306      0.0429        0.228       0.394           14

Acceptance rate  sigma_cycle  =  0.525625
Acceptance rate  sigma_level  =  1.0
Acceptance rate  rho  =  0.448375
Acceptance rate  lambda  =  0.46275
```

The total time taken for the MCMC scheme is just over 100 seconds. Note that both the method and model differs from Strickland *et al.* (2009). The marginal posterior mean, marginal posterior standard deviation, 2.5% and 97.5% credible intervals, as well as the IFs are reported for each of σ_κ, σ_ν, ρ and λ.

```
means, vars = mcmc.get_mean_var('state')
title('Trend Cycle Model')
plot(ymat.T, color = 'k')
plot(means[0], color = 'k')
subplot(2, 1, 2)
plot(means[2], color = 'k')
show()
```

The function `get_mean_var('state')` is used to obtain the marginal posterior mean and variance for the state. The remainder of the code, uses functions from **pylab**, which is a part of the library **Matplotlib** to plot the marginal posterior means for the state (see Figure 3).

5. Conclusions

In this paper a Python library (module) called **PySSM** has been introduced. **PySSM** is a powerful tool to analyze time series using SSMs. It utilizes the best features of the high level language Python to define, store and manipulate data. Furthermore, it is also used to interface with lower level languages such as Fortran which has the function to perform intensive numerical calculations.

SSMs an be easily defined and analyzed using classes from **PySSM**. We have demonstrated the use of **PySSM**, in a Bayesian context, through three examples.

Acknowledgments

This research was supported by an Australian Research Council Linkage Grant; LP100100565.

References

Anderson BDO, Moore JB (1979). *Optimal Filtering*. Prentice Hall, Englewood Cliffs.

Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Croz JD, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1999). ***LAPACK** Users' Guide*. 3rd edition. Society for Industrial and Applied Mathematics, Philadelphia. ISBN 0-89871-447-8.

Carter C, Kohn R (1994). "On Gibbs Sampling for State Space Models." *Biometrika*, **81**(3), 541–553.

De Jong P (1991). "The Diffuse Kalman Filter." *The Annals of Statistics*, **19**(2), 1073–1083.

De Jong P, Shephard N (1995). "The Simulation Smoother for Time Series Models." *Biometrika*, **82**(2), 339–350.

Drukker DM, Gates RB (2011). "State Space Methods in Stata." *Journal of Statistical Software*, **41**(10), 1–25. URL http://www.jstatsoft.org/v41/i10/.

Durbin J, Koopman SJ (2001). *Time Series Analysis by State-Space Methods*. Camberidge University Press, Camberidge, UK.

Durbin J, Koopman SJ (2002). "A Simple and Efficient Simulation Smoother for Time Series Models." *Biometrika*, **89**(3), 603–616.

Frühwirth-Schnatter S (2004). *Efficient Bayesian Parameter Estimation For State Space Models Based on Reparameterisations, State Space and Unobserved Component Models: Theory and Applications*. Cambridge University Press, Cambridge.

Garthwaite PH, Fan Y, Scisson SA (2010). "Adaptive Optimal Scaling of Metropolis-Hastings Algorithms Using the Robbins-Monroe Process." *Technical report*, University of New South Wales.

Harvey AC (1989). *Forecasting Structural Time Series and the Kalman Filter*. Cambridge University Press, Cambridge.

Jones E, Oliphant T, Peterson P (2001). ***SciPy**: Open Source Scientific Tools for Python*. URL http://www.scipy.org/.

Koopman SJ, Harvey AC, Doornik JA, Shephard N (2009). ***STAMP** 8.2: Structural Time Series Analyser, Modeler, and Predictor*. Timberlake Consultants, London.

Mendelssohn R (2011). "The **STAMP** Software for State Space Models." *Journal of Statistical Software*, **41**(2), 1–18. URL http://www.jstatsoft.org/v41/i02/.

Oliphant TE (2006). *Guide to **NumPy***. Provo, UT. URL http://www.tramy.us/.

Oliphant TE (2007). "Python for Scientific Computing." *Computing in Science and Engineering*, **9**(3), 10–20.

Peng JY, Aston JAD (2011). "The **State Space Models** Toolbox for MATLAB." *Journal of Statistical Software*, **41**(6), 1–26. URL http://www.jstatsoft.org/v41/i06/.

Peterson P (2009). "**F2PY**: A Tool for Connecting Fortran and Python Programs." *International Journal of Computational Science and Engineering*, **4**(4), 296–605.

Petris G, Petrone S (2011). "State Space Models in R." *Journal of Statistical Software*, **41**(4), 1–25. URL http://www.jstatsoft.org/v41/i04/.

R Core Team (2013). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL http://www.R-project.org/.

Robert CP, Casella G (1999). *Monte Carlo Statistical Methods*. Springer-Verlag, New York.

Rue H, Held L (2005). *Gaussian Markov Random Fields*. Chapman & Hall/CRC.

SAS Institute Inc (2011). *The SAS System, Version 9.3*. SAS Institute Inc., Cary, NC. URL http://www.sas.com/.

Selukar R (2011). "State Space Modeling Using SAS." *Journal of Statistical Software*, **41**(12), 1–13. URL http://www.jstatsoft.org/v41/i12/.

StataCorp (2013). *Stata Data Analysis Statistical Software: Release 12*. StataCorp LP, College Station, TX. URL http://www.stata.com/.

Strickland C, Alston C, Denham R, Mengersen K (2011). "**PyMCMC**: A Python Package for Bayesian Estimation Using Markov Chain Monte Carlo." *Technical report*, Queensland University of Technology.

Strickland CM, Martin GM, Forbes CS (2008). "Parameterisation and Efficient MCMC Estimation of Non-Gaussian State Space Models." *Computational Statistics & Data Analysis*, **52**(6), 2911–2930.

Strickland CM, Turner I, Denham RJ, Mengerson KL (2009). "Efficient Bayesian Estimation of Multivariate State Space Models." *Computational Statistics & Data Analysis*, **53**(12), 4116–4125.

The MathWorks, Inc (2011). *MATLAB – The Language of Technical Computing, Version R2011b*. The MathWorks, Inc., Natick, Massachusetts. URL http://www.mathworks.com/products/matlab/.

van Rossum G (1995). *Python Tutorial, Technical Report CS-R9526*. Centrum voor Wiskunde en Informatica (CWI), Amsterdam.

A. Computing the residuals

The residuals, $\boldsymbol{\eta}_t$ are calculated as $\boldsymbol{\eta}_t = \boldsymbol{G}_t^{\dagger}\left(\boldsymbol{\alpha}_{t+1} - \boldsymbol{W}_t\boldsymbol{\beta}_s - \boldsymbol{T}_t\boldsymbol{\alpha}_t\right)$, for $t = 1, 2, \ldots, n-1$, where $\boldsymbol{G}_t^{\dagger}$ is the pseudo-inverse of $\boldsymbol{G}_t$. Whilst, this is the correct calculation for most models in our experience, it may not always be what the user is expecting. Take for example the case where the transition state equation is specified as an autoregressive (AR) process of order 2. That is, set $\boldsymbol{\alpha}_t = \begin{bmatrix} \mu_t \\ \mu_t^* \end{bmatrix}$, $\boldsymbol{T}_t = \begin{bmatrix} \phi_1 & 1 \\ \phi_2 & 0 \end{bmatrix}$, $\boldsymbol{G_t} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ and assuming no regressors $\boldsymbol{\beta}_s = \mathbf{0}$, it follows that

$$\begin{aligned} \begin{bmatrix} \mu_{t+1} \\ \mu_{t+1}^* \end{bmatrix} &= \begin{bmatrix} \phi_1 & 1 \\ \phi_2 & 0 \end{bmatrix} \begin{bmatrix} \mu_t \\ \mu_t^* \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \eta_t \\ \begin{bmatrix} \mu_{t+1} \\ \mu_{t+1}^* \end{bmatrix} &= \begin{bmatrix} \phi_1\mu_t + \mu_t^* + \eta_t \\ \phi_2\mu_t \end{bmatrix}. \end{aligned} \tag{16}$$

If we plug μ_t^* into the top row of the system in (16), then it is clear that we obtain

$$\mu_{t+1} = \phi_1\mu_t + \phi_2\mu_{t-1} + \eta_t.$$

As $\boldsymbol{G}_t^{\dagger} = \begin{bmatrix} 1 & 0 \end{bmatrix}$, then it is clear that $\eta_t = \boldsymbol{G}_t^{\dagger}\left(\boldsymbol{\alpha}_{t+1} - \boldsymbol{W}_t\boldsymbol{\beta}_s - \boldsymbol{T}_t\boldsymbol{\alpha}_t\right)$, implies that

$$\eta_t = \mu_{t+1} - \phi_1\mu_t - \phi_2\mu_{t-1},$$

which is as expected. If, however, $\boldsymbol{T}_t$ is updated then the function `get_state_residual()` is called then $\boldsymbol{\eta}_t = \boldsymbol{G}_t^{\dagger}(\boldsymbol{\alpha}_{t+1} - \boldsymbol{W}_t\boldsymbol{\beta}_s - \boldsymbol{T}_t\boldsymbol{\alpha}_t)$, may not provide the user with what they expect. For example, suppose $\boldsymbol{T}_t$ is updated with the parameters ϕ_1^{new} and ϕ_2^{new}, then $\eta_t = \boldsymbol{G}_t^{\dagger}\left(\boldsymbol{\alpha}_{t+1} - \boldsymbol{W}_t\boldsymbol{\beta}_s - \boldsymbol{T}_t\boldsymbol{\alpha}_t\right)$, and in this case

$$\eta_t = \mu_{t+1} - \phi_1^{new}\mu_t - \phi_2\mu_{t-1},$$

where it is important to note that the values for ϕ_2 is still the old value, assuming no updates in the estimate for $\boldsymbol{\alpha}_t$. As such, if the user requires this function, then they should insure it is calculating what they want, for the model they specify.

B. Computing the log probability

Using the same example as above, with the AR process of order 2, suppose our prior for ϕ_1 and ϕ_2 is *a priori* independent, such that $p\left(\phi_1, \phi_2\right) = p\left(\phi_1\right)p\left(\phi_2\right)$. Then the posterior for ϕ_2 may be expressed as

$$p\left(\phi_2|\alpha, \phi_1\right) \propto p\left(\alpha|\theta\right)p\left(\phi_2\right). \tag{17}$$

If we wish to evaluate this function at $\phi_2 = \phi_2^{new}$ then using the function `log_probability_state()` to compute $\log p\left(\alpha|\theta\right)$, even after updating $\boldsymbol{T}_t$ with the new value, ϕ^{new}, will unfortunately not give the desired result as

$$\boldsymbol{G}_t^{\dagger}\left(\boldsymbol{\alpha}_{t+1} - \boldsymbol{W}_t\boldsymbol{\beta}_s - \boldsymbol{T}_t\boldsymbol{\alpha}_t\right) \neq \mu_{t+1} - \phi_1\mu_t - \phi_2\mu_{t-1}.$$

Again, the user must ensure that $\boldsymbol{G}_t^{\dagger}\left(\boldsymbol{\alpha}_{t+1} - \boldsymbol{W}_t\boldsymbol{\beta}_s - \boldsymbol{T}_t\boldsymbol{\alpha}_t\right)$ is the correct computation for $\boldsymbol{\eta}_t$. Here the user also needs to ensure that $\left|\boldsymbol{G}_t\boldsymbol{Q}_t\boldsymbol{G}_t^{\top}\right|$ is the correct quantity to calculate. That

said, as for `get_state_residual()`, we find that for most models equation (5) is the correct calculation.

C. Installation

PySSM should build relatively easily under most Linux / Unix platforms. Installing the package from source will require the following:

- A C and Fortran compiler. Development of **PySSM** used the GNU compilers.
- Python, version 2.7 was used in development, but Python 2.6 may also work.
- Functions within **PySSM** make use of the automatically tuned linear algebra software (**ATLAS**). Most distributions will have precompiled versions available, but if you need to build your own, there are numerous web pages with instructions. See, for example, the **SciPy** wiki at http://www.scipy.org/Installing_SciPy/Linux.

With these packages available, the source can be retrieved and built in the following manner:

```
git clone git@bitbucket.org:christophermarkstrickland/pyssm.git
cd pyssm
python setup.py install
```

More detailed instructions on building **PySSM**, including how to build for Microsoft Windows, are included in the source distribution.

Additionally, binary distributions for Linux Ubuntu are available from the authors' personal package archive `ppa:rjadenham/ppa`. Installing under Ubuntu can then be achieved by running the following commands:

```
sudo add-apt-repository ppa:rjadenham/ppa
sudo apt-get update
sudo apt-get install python-pyssm
```

Windows binaries may also be available in the download section of the code repository web site (see https://bitbucket.org/christophermarkstrickland/pyssm/downloads).

Affiliation:

Christopher M. Strickland
School of Economics
University of New South Wales
High Street Kensington
New South Wales, 2052, Australia
E-mail: c.strickland@unsw.edu.au

Robert J. Denham
Remote Sensing Centre
Queensland Department of Science, Information Technology, Innovation and the Arts
GPO Box 5078
Brisbane 4001, Australia
E-mail: robert.denham@qld.gov.au

Journal of Statistical Software http://www.jstatsoft.org/
published by the American Statistical Association http://www.amstat.org/
Volume 57, Issue 6 *Submitted:* 2012-03-23
March 2014 *Accepted:* 2013-09-21