Journal of Statistical Software

November 2018, Volume 87, Issue 5. *doi: 10.18637/jss.v087.i05*

SSpace: A Toolbox for State Space Modeling

Marco A. Villegas
Universidad de Castilla-La Mancha

Diego J. Pedregal
Universidad de Castilla-La Mancha

Abstract

SSpace is a MATLAB toolbox for state space modeling. State space modeling is in itself a powerful and flexible framework for dynamic system modeling, and **SSpace** is conceived in a way that tries to maximize this flexibility. One of the most salient features is that users implement their models by coding a MATLAB function. In this way, users have complete flexibility when specifying the systems, have absolute control on parameterizations, constraints among parameters, etc. Besides, the toolbox allows for some ways to implement either non-standard models or standard models with non-standard extensions, like heteroskedasticity, time-varying parameters, arbitrary nonlinear relations with inputs, transfer functions without the need of using explicitly the state space form, etc. The toolbox may be used on the basis of scratch state space systems, but is supplied with a number of templates for standard widespread models. A full help system and documentation are provided. The way the toolbox is built allows for extensions in many ways. In order to fuel such extensions and discussions an online forum has been launched.

Keywords: state space models, unobserved components, ARIMA, exponential smoothing, Kalman filter, MATLAB.

1. Introduction

SSpace is a MATLAB toolbox (The MathWorks Inc. 2017) that provides a number of routines designed for a general analysis of state space systems. It combines both flexibility and simplicity, and at the same time it enhances the power and versatility of state space modeling in a user-friendly environment. The toolbox possesses very distinct properties compared to other state space pieces of software, but at the same time takes advantage of methods and algorithms from other sources, mainly Taylor, Pedregal, Young, and Tych (2007) and Durbin and Koopman (2012). The combination of all these factors gives **SSpace** a particular flavor.

Popularity of high level programming languages like MATLAB has brought the availability of many free packages with an incredibly wide range of applications in many research areas.

State space (SS) routines are not an exception, and therefore good packages are so widespread nowadays, either as open source or paid versions, that it is impossible to quote even a portion of them. Without any intention of being exhaustive, we offer here a list of the most popular packages within the academic community. A partial review is available in Volume 41 of the *Journal of Statistical Software* (Commandeur, Koopman, and Ooms 2011). There are several toolboxes written in MATLAB, like toolboxes supplied with the core program, but also **CAPTAIN** (Taylor *et al.* 2007), **SSM** (Peng and Aston 2011), **SSMMATLAB** (Gómez 2015) and **E4** (Casals, Garcia-Hiernaux, Jerez, Sotoca, and Trindade 2016). One piece of software widely known is **SSfPack** (Koopman, Shephard, and Doornik 2008). Some others are written either in R (R Core Team 2018; Petris and Petrone 2011; e.g., package **KFAS**; Helske 2017), **RATS** (Doan 2011), **gretl** (Lucchetti 2011), etc. Also, some other, menu-driven programs that incorporate SS routines with less flexible programming capabilities are **STAMP** (Koopman, Harvey, Doornik, and Shephard 2009), Eviews (IHS Inc. 2010; Van den Bossche 2011), SAS (SAS Institute Inc. 2013; Selukar 2011), Stata (StataCorp 2017; Drukker and Gates 2011), etc. **ECOTOOL** is a complementary MATLAB toolbox written by the same authors for the identification and estimation of dynamical systems (Pedregal and Trapero 2012).

In a broad sense, **SSpace** provides the user with the most advanced and up-to-date features available in the state space framework, sharing some of these properties with some packages mentioned above and competing with them. However, some other features are specific to this toolbox and will not be found in any of the alternatives.

Regarding statistical issues, the main features of **SSpace** are:

1. Full multivariate linear and nonlinear Gaussian models, and univariate non-Gaussian models are implementable. In any of these possibilities nonlinear, time-varying or transfer function relations with inputs are possible.

2. The framework is very general in the equations formulation and in the sense that all system matrices are potentially time-varying or state-dependent.

3. Kalman filter, fixed interval smoothing and disturbance smoothing are implemented with exact, diffuse or ad-hoc initialization. Exact initialization in nonlinear models is also possible following Koopman and Lee (2009).

4. Steady state detection of linear invariant systems.

5. Use of exact scores in maximum likelihood estimation, when possible. Use of numerical gradients is always possible, sometimes compulsory.

6. Other estimation procedures apart from maximum likelihood are implemented. At the moment the toolbox offers concentrated likelihood and minimization of combinations of several steps ahead forecast errors. More cost functions may be added in the future.

7. Kalman filtering and smoothing of multivariate systems are based on univariate treatment of multivariate systems, see Durbin and Koopman (2012, pages 155–160).

8. A family of models not used in any of the alternative packages is included, namely the dynamic harmonic regression. To the best knowledge of the authors, this is the first time that a multivariate seemingly unrelated dynamic harmonic regression is used and implemented. Such model is an extension of the univariate dynamic harmonic regression counterpart, see Young, Pedregal, and Tych (1999) and the worked example 3 below.

On the operative side, **SSpace** is rather flexible and user-friendly because:

- The toolbox is user-oriented in the sense that a big effort has been done on the developer side with the aim of simplifying usage to the final users. As a consequence, a comprehensive time series analysis may be performed with full control over models, parameters and specifications, by using just a few number of functions that follow a simple and fixed calling-standard that is easy to remember (see the section with the worked examples below). One example of this simplicity is that just one function (namely `SSfilter`) is used for filtering any kind of model, regardless of whether it is linear, non-Gaussian or nonlinear, i.e., the toolbox detects the type of model and applies the appropriate algorithms in each case automatically without any specific intervention of the user.
- The toolbox is composed of less than thirty functions with names that have been carefully selected following nemotechnic rules, so that the user may remember them or easily look for their names. There are three groups of functions:
  - Core functions (named `SS*`) which set up the models and perform the basic operations of filtering, smoothing, forecasting, validating, etc.
  - Template functions (`Sample*`) which help the user to set up the model either in terms of the crude state space system matrices or in terms of its specific nature.
  - Other helpful functions for easy handling of models, matrices, etc. The main ones are for constraining parameters, building semi-definite covariance matrices, differencing time series, building forecast confidence bands, etc.
- Another key point is that models are directly specified by the user in a user-coded function written in plain MATLAB syntax. This approach has at least the following advantages:
  - Once the model is specified in the user-coded function, the same syntax applies to all sort of models, regardless of linearity, Gaussianity, estimation method, etc. i.e., the same functions with the same syntax are used for estimation, filtering, smoothing, etc. Internally the operation of the toolbox may be rather different in each case, but such complexity does not require any intervention of the user.
  - The user-coded function is written following a particular template, that is basically a list of empty values for all the system matrices. The toolbox offers particular templates for a wide range of standard models. These templates may be extended or substituted by users. Such functions and templates may be extended in many ways. Typically, for complex models the model function has to be extended with the aid of additional inputs to the MATLAB function in order to define the model in SS form.
  - The user has full control over his models in a fairly straightforward manner, e.g., different specific parameterizations of the same model are possible, parameter constraints of any kind may be imposed, non-standard features of the models may be added (like adding heteroskedasticity, time-varying parameters, nonlinear exogenous relations, ...), etc.
- Parameter estimation is carried out by the standard `fminunc` function from the **Optimization** toolbox in MATLAB. By editing the script `optimizer.m` the user may tune the optimization settings and/or change even the optimizer itself.

As a summary, the toolbox is highly configurable and could be extended in several ways, a reason for which users are encouraged to push their own contributions to the repository https://bitbucket.org/predilab/sspace-matlab/. Usually default options for modeling are in place in such a way that a few commands with a few options would produce sensible results, but advanced users may take advantage of the possibility to configure the toolbox at their own convenience. For example, they could select different sets of initial parameter values to start estimation in order to find the global optimum, build alternative cost functions to the log-likelihood for parameter estimation based on the output of the Kalman filter, try out different optimization algorithms to search for optimal parameters, add templates for models not yet implemented or suggest different versions to the existing ones, build canned functions for standard model estimation, etc.

The rest of the paper is organized as follows: The next section shows the models implemented in analytical form; Section 3 provides a broad vision of the functions included in the toolbox and its main usage on implementing a full time series analysis through a simple example; Section 4 illustrates the usage of **SSpace** with several examples for different scenarios and complexity levels; and Section 5 extracts the main conclusions. Code listings and examples are considerably compressed for space reasons, but may be consulted in extended form in the **SSpace** demos.

2. Models implemented in SSpace

SSpace supports linear Gaussian models, non-Gaussian models and nonlinear models.

The linear Gaussian version is shown in Equation 1.

$$\begin{array}{rl} \alpha_{t+1} = T_t\alpha_t + \Gamma_t + R_t\eta_t, & \eta_t \sim N(0, Q_t), \\ y_t = Z_t\alpha_t + D_t + C_t\epsilon_t, & \epsilon_t \sim N(0, H_t), \\ \alpha_1 \sim N(a_1, P_1), & t = 1, 2, \ldots, N. \end{array} \tag{1}$$

In these equations α_t is the state vector of length n; y_t are the $m \times 1$ vector of output data; η_t and ϵ_t are the state and observational vectors of zero mean Gaussian noises, with dimensions $r \times 1$ and $h \times 1$, respectively; both noises are allowed to be correlated by a system matrix $S_t = \mathsf{COV}(\eta_t, \epsilon_t)$ of dimension $r \times h$; α_1 is the initial state with mean a_1 and covariance P_1 and independent of all disturbances and observations involved. The remaining elements in (1) are the rest of the system matrices with appropriate dimensions, i.e.,

$$\begin{array}{llllll} T_t\text{:} & n \times n; & \Gamma_t\text{:} & n \times 1; & R_t\text{:} & n \times r; \\ Z_t\text{:} & m \times n; & D_t\text{:} & m \times 1; & C_t\text{:} & m \times h. \end{array}$$

One interesting feature (see Section 4.1 for an example) is that the toolbox is flexible enough to allow the terms Γ_t and D_t to be defined in such a way that k exogenous input variables appear explicitly, i.e., $\Gamma_t = f(\gamma_t, u_t)$ and $D_t = g(d_t, u_t)$, with u_t of dimension $k \times 1$. Beware that general functions $f(\bullet)$ and $g(\bullet)$ include as particular cases time-varying linear functions $\Gamma_t = \gamma_t u_t$ and $D_t = d_t u_t$, with γ_t and d_t of dimensions $n \times k$ and $m \times k$, respectively.

Apart from this, the formulation in (1) is also rather general. In particular, data sets may be multivariate, all system matrices are time-varying and noises in state and observation equations may be correlated. The system is actually so general that some readers would immediately detect some redundancies and some terms that are not strictly necessary in

most applications. However, we have preferred to set up the model in the most general possible way, such that any potential user acquainted with SS methodology does not have to change his particular mindset, so that the effort to translate the system into **SSpace** is kept to a minimum.

The non-Gaussian SS setup is shown in Equation 2:

$$\begin{aligned} \alpha_{t+1} &= T_t\alpha_t + \Gamma_t + R_t\eta_t, \quad \eta_t \sim N(0, Q_t), \\ y_t &\sim p(y_t \mid \theta_t) + D_t, \\ \theta_t &= Z_t\alpha_t, \qquad\qquad t = 1, 2, \ldots, N. \end{aligned} \tag{2}$$

Here θ_t is known as the *signal*. With this representation it is possible to deal with three types of models (see Durbin and Koopman 2012):

1. Exponential family distribution, where $p(y_t \mid \theta_t) = \exp[y_t^\top \theta_t - b_t(\theta_t) + c_t(y_t)], -\infty < \theta_t < \infty$.
2. Stochastic volatility models, i.e., $y_t = \exp(\frac{1}{2}\theta_t)\epsilon_t + D_t$.
3. Observations generated by the relation $y_t = \theta_t + \epsilon_t, \epsilon_t \sim p(\epsilon_t)$, with $p(\bullet)$ being a distribution of the exponential family.

Finally, the nonlinear models are of the type shown in Equation 3.

$$\begin{aligned} \alpha_{t+1} &= T_t(\alpha_t) + \Gamma_t + R_t(\alpha_t)\eta_t, \quad \eta_t \sim N(0, Q_t(\alpha_t)), \\ y_t &= Z_t(\alpha_t) + D_t + C_t(\alpha_t)\epsilon_t, \quad \epsilon_t \sim N(0, H_t(\alpha_t)) \quad t = 1, 2, \ldots, N. \end{aligned} \tag{3}$$

Functions $T_t(\alpha_t)$ and $Z_t(\alpha_t)$ with first derivatives provide nonlinear transformations of the state vector into vectors of size $n \times 1$ and $m \times 1$, respectively. The rest of the system matrices may also depend on the state vector and $S_t = 0$.

Given this general framework, (extended) Kalman filtering, state and disturbance smoothing provide the basis for optimal state estimation, parameter estimation, signal extraction, forecasting, etc. For all the algorithmic issues not explicitly commented in this paper refer to Young *et al.* (1999), Taylor *et al.* (2007) and Durbin and Koopman (2012).

Table 2 shows some of the main features of most common software packages. The top block corresponds to toolboxes written in MATLAB, the rest are developed in other environments. The table highlights several facts: (i) exact initialization is present in most packages, (ii) nonlinear and non-Gaussian models are less common than expected and (iii) most packages use maximum likelihood estimation as the only estimation method.

Moreover, there are some unique properties of **SSpace**, as far as the authors are aware of, like the implementation of multivariate dynamic harmonic regression models, the implementation of systems by direct specification of the system matrices in a MATLAB function, system estimation via the minimization of forecast errors several steps ahead, the possibility of systems implementing arbitrary nonlinear (and possibly multivariate) relations among inputs and outputs, the possibility of multiple-input-multiple-output transfer functions, and the chance to select an arbitrary minimizer algorithm to estimate the models (with `fminunc` as a default).

The flexibility of **SSpace** is so big that, being this its most powerful feature, at the same time it may be a problem for some users. Specifying a model from scratch in **SSpace** takes some

	EI	+ML	UTMM	NLNG	DA
SSpace	✓	✓	✓	✓	✓
E4	✓	✓			
SSM	✓			✓	✓
SSMMATLAB	✓				
SSfPack	✓		✓	✓	✓
STAMP	✓		✓		✓
R **KFAS**	✓		✓	✓	✓
Stata	✓		✓		
Eviews					
gretl			✓		✓
SAS	✓				

Table 1: Options available in most common state space software packages. The options are exact initialization (EI), other estimation methods apart from ML (+ML), univariate treatment of multivariate models (UTMM), nonlinear and non-Gaussian models (NLNG) and availability of state disturbance algorithms (DA).

time and a bit of familiarization with the toolbox. To solve these problems two solutions are implemented: (i) when specifying the model, **SSpace** performs internally a full set of coherency tests to ensure that the model is correctly specified and issues error or warning messages with the specific problems that guide the user towards the solution, and (ii) templates for most common models are provided so that the user does not have to bear in mind the state space form of any of them. This is actually a part of **SSpace** that will grow as more templates appear in the future.

3. SSpace overview

Table 3 shows the core functions necessary to carry out a comprehensive time series analysis. Among all these the most important to understand is `SSmodel`. It creates a structure with the user inputs and all the outputs, that will be empty at the time of creation (for a full description of inputs and outputs type `help SSmodel` at the MATLAB prompt). This structure will be the input to the rest of the functions that have to handle the system, like estimation, filtering, etc., and it also may be the output to such functions, in a way that it is completed little by little with each additional operation. With `SSmodel` the user specifies the input and output data, the model to use, additional inputs to the user-coded function necessary to implement the model, either exact or diffuse or ad-hoc initialization of recursive algorithms, initial parameters for parameter estimation, fixed parameters that would be frozen in estimation, the cost function to optimize, exact or numerical scores (if possible) in maximum likelihood estimation, etc. To sum up, it sets up the models up to the smallest detail, controlling the posterior performance of the rest of functions.

Once the model is created, `SSestim` performs parameter estimation by the method previously selected in `SSmodel`. `SSvalidate` produces a table with the estimation results and diagnostics. `SSfilter` produces the innovations and filtered estimates of states and covariance matrices with additional output. If smoothed output is preferred, it is produced by the `SSsmooth` function. Disturbance errors (and smoothed output) may be computed by `SSdisturb`. Finally,

`SSmodel`	Creates **SSpace** model object or adds properties to an existing one.
`SSestim`	Estimation of a **SSpace** model.
`SSvalidate`	Validation of a **SSpace** model.
`SSfilter`	Optimal Kalman filtering of a **SSpace** model.
`SSsmooth`	Optimal fixed interval smoothing of a **SSpace** model.
`SSdisturb`	Optimal disturbance smoother.
`SSdemo`	Runs **SSpace** demos 1 to 8.

Table 2: Main functions of the **SSpace** library.

Linear and Gaussian models	
`SampleSS`	General SS template.
`SampleARIMA`	ARIMA models with exogenous variables.
`SampleBSM`	Basic structural model.
`SampleDHR`	Dynamic harmonic regression.
`SampleDLR`	Dynamic linear regression.
`SampleES`	Exponential smoothing with exogenous variables.
Non-Gaussian models	
`SampleNONGAUSS`	General non-Gaussian models.
`SampleEXP`	Non-Gaussian exponential family models.
`SampleSV`	Stochastic volatility models.
Nonlinear models	
`SampleNL`	General nonlinear models.
Other templates	
`SampleAGG`	Models with time aggregation.
`SampleCAT`	Concatenation of state space systems.
`SampleNEST`	Nesting in inputs state space systems.

Table 3: Available templates for the **SSpace** toolbox.

there are eight step-by-step demos ready, that may be run with the `SSdemo` function.

All the functions in Table 3 run on the model previously coded by the user in MATLAB language. In order to make the communication between the user and **SSpace** efficient and easy, a number of templates have been created and listed in Table 3. `SampleSS` sets up any linear and Gaussian models with or without inputs and any sort of non-standard feature. The rest of the linear models are self-explanatory and are intended for the creation of well-known models. There are also some templates for non-Gaussian models and for general nonlinear models (`SamplenNL`). Additional templates help the user to build models with time aggregation, concatenate systems or nest systems in inputs. In all cases, time-varying system matrices are three dimensional, being time the third dimension.

The rest of the functions in Table 3 are very useful to set up models in SS form: (i) `confband` builds confidence bands of filtered or smoothed outputs in a comfortable way, (ii) `constrain` settles constraints among parameters in the models, (iii) `varmatrix` is a function useful to constrain covariance matrices to be semi-positive-definite in multivariate models or just positive in scalar cases, (iv) `normalize` standardizes any set of time series with a time-varying covariance structure, (v) `vdiff` produces differencing of vector time series, and (vi) `optimizer`

`confband`	Forecasts confidence intervals.
`constrain`	Free constraints of parameters.
`varmatrix`	Semi-definite-positive covariance matrices.
`normalize`	Variable normalization (standardization).
`vdiff`	Differentiation of vector time series.
`optimizer`	Optimizer options.

Table 4: Auxiliary functions for the **SSpace** library.

is an editable script that allows tuning the optimizer tolerances and even to change the optimizer itself.

The analysis with **SSpace** consists of following the next steps, that mimic closely the steps any researcher ought to follow in any SS analysis.

1. Write the model on paper or specify the model in SS form.
2. Translate model to MATLAB code using the templates supplied.
3. Set up model with `SSmodel`.
4. Estimate unknown parameters with `SSestim`.
5. Check appropriateness of model with `SSvalidate`.
6. Determine optimal estimates of states, their covariance matrices, innovations, etc. Any or several of `SSfilter`, `SSsmooth` or `SSdisturb` may be used.

In the next subsections, a local level (or random walk plus noise) model is used to illustrate how to implement all these steps applied to the Nile river data used in Durbin and Koopman (2012), specifically `demo1` of **SSpace**. The data consists of the flow volume of the Nile river at Aswan from 1871 to 1970. The local level model is given in Equation 4, being B the backshift operator.

$$y_t = \frac{\eta_t}{(1-B)} + \epsilon_t; \quad \mathsf{VAR}(\eta_t) = Q; \quad \mathsf{VAR}(\epsilon_t) = H. \tag{4}$$

3.1. Specify model (step 1)

One SS representation of (4) is (5):

$$\begin{aligned} \text{State equation:} \quad & \alpha_{t+1} = \alpha_t + \eta_t, \quad \eta_t \sim N(0, Q), \\ \text{Observation equation:} \quad & y_t = \alpha_t + \epsilon_t, \qquad \epsilon_t \sim N(0, H). \end{aligned} \tag{5}$$

It may be seen immediately that the local level is one of the simplest models that can be specified in SS form. Comparing it with the general form (1) we see that for this case all system variables are scalar and time invariant, i.e., $T_t = R_t = Z_t = C_t = 1$, $Q_t = Q$, $H_t = H$ and Γ_t and D_t do not exist because the model has no inputs.

3.2. Code the model (step 2)

The best way to deal with the previous model is to edit the `SampleSS` template, rename it to, say, `example1`, and fill in all the matrix values accordingly. The aspect of `SampleSS` is

shown below, with the system matrix names easily identifiable. The template is offered in this way, nothing should be removed, but anything could be added in order to define the system matrices.

```
function model = SampleSS(p)
  model.T = [];
  model.Gam = [];
  model.R = [];
  model.Z = [];
  model.D = [];
  model.C = [];
  model.Q = [];
  model.H = [];
  model.S = [];
```

The following is the adaptation of such template to the local level model.

```
function model = example1(p)
  model.T = 1;
  model.Gam = [];
  model.R = 1;
  model.Z = 1;
  model.D = [];
  model.C = 1;
  model.Q = 10.^p(1);
  model.H = 10.^p(2);
  model.S = 0;
```

Note that the input argument `p` to both functions is a vector of parameters, in this case just the scalars Q and H. By default, both state and observation noises are considered independent. Furthermore, the first element in the vector `p` has been assigned to the matrix Q, while the second is assigned to H. Since both must be positive values, the system matrices are defined as powers of 10. An alternative and equivalent definition is `model.Q = varmatrix(p(1))`.

3.3. Setting up the model (step 3)

Now the user has to communicate with **SSpace** and build a model to use later on. This is done with the `SSmodel` function. In this case the MATLAB code is simply `sys = SSmodel('y', nile, 'model', @example1)`. It is assumed that `nile` is a vector variable containing the Nile data and basically with this line of code the user is telling that he wants to apply the local level model written in `example1` to the data in the MATLAB variable `nile`.

This is the most important step because at this stage the user defines the posterior performance of the toolbox. In essence, if the validation of the model is not correct, then the user has to come back to `SSmodel` and change either the model, options, etc. There are many options available to set up the model, that are passed on to `SSmodel` using duplets (see all possibilities in the **SSpace** documentation).

3.4. Estimate parameters (step 4)

Having defined the model in the previous step, the rest is straightforward. In particular, the estimation is done by `sys = SSestim(sys)`.

No additional inputs to this functions are necessary, since everything has been set up in the previous step via the `SSmodel` function, in particular the estimation method that will be used, i.e., exact maximum likelihood by default. Parameters are stored in the `sys` output structure. If the estimation converges to a well defined optimum, then estimation results, with standard errors of parameters, information criteria, etc. may be shown by means of the `SSvalidate` function with the syntax `sys = SSvalidate(sys)`.

3.5. Use the model (step 5)

A final step consists of estimating the filtered and/or smoothed output together with the disturbances of the model, and further validation tests. These operations constitute the basis for forecasting, signal extraction, interpolation, etc. When only the filtered output is required the call should be `sys = SSfilter(sys)`; if smoothed estimates without disturbances are preferred then the call should be `sys = SSsmooth(sys)`; whereas the full computation and output are produced by the call `sys = SSdisturb(sys)`.

Results in all these brief examples are stored always in the `sys` output structure, though at any point different structures may be used. It stores parameters with covariance matrix, optimal states and covariances, fitted output values and covariances, forecasted values and covariances, innovations, disturbances estimates with covariances. Further statistical diagnostics are advisable.

4. Worked examples

The examples shown in this section are presented as illustrations of the flexibility and power of the toolbox, with no pretension of showing any scientific result or improvement over other researchers' analysis. Code listings are truncated in order to save space. This is especially important in the case of the templates shown, for which the extended versions are also included in the software provided with an abundant help. Identification and validation issues are not treated in the examples to keep them short.

4.1. Example 1: Local level

The next listing puts together the MATLAB code shown up to now for the local level model applied to the Nile river data, with some additions for plotting outputs.

```
% Load data and set up time variable with NaN's
% for interpolation and forecasting
load nile
y = [nile; nan(10, 1)];
y(61:70) = nan;
t = (1 : size(y, 1))';
% Build SSpace model
sys = SSmodel('y', y, 'model', @example1);
% Estimate model by exact ML
sys = SSestim(sys);
```

```
% Model table output etc.
sys = SSvalidate(sys);
% Smoothing
sys = SSsmooth(sys);
% Plotting fitted values with 90% confidence bands
plot(t, y, 'k', t, sys.yfit, 'r.-', t, confband(sys.yfit, sys.F, 1), 'r:')
% Plotting innovations
plot(sys.v)
% Estimating and plotting disturbances
sys = SSdisturb(sys);
plot(t, sys.eta, 'k', t, sys.eps, 'r')
```

Some missing observations have been arbitrarily added in the middle of the data and at the end to show how interpolation and forecasting are automatically done. The output of the `SSvalidate` call is:

```
-----------------------------------------------------
Linear Gaussian model: example1.
Objective function: llik
System collapsed at observation 1 of 100.
Exact gradient used.
-----------------------------------------------------
         Param    S.E.   T-test  P-value |Gradient|
-----------------------------------------------------
 p(1)   3.1404  0.3595   8.7362   0.0000   0.000008
 p(2)   4.2084  0.0855  49.2198   0.0000   0.000051
-----------------------------------------------------
               AIC: 12.906
               SBC: 12.9938
               HQC: 12.9398
    Log-likelihood: -571.3177
      Corrected R2: 0.2669
Residual Variances: 21919.3612
-----------------------------------------------------
Summary Statistics:
-----------------------------------------------------
                             1
 Missing data points    10.0000
 Q( 1)                   1.1031
 Q( 4)                   4.1847
 Q( 8)                   7.6771
 Q(12)                  18.9641
 H(33)                   0.6020
 P-value                 0.1501
 Bera-Jarque             0.0209
 P-value                 0.9896
-----------------------------------------------------
```

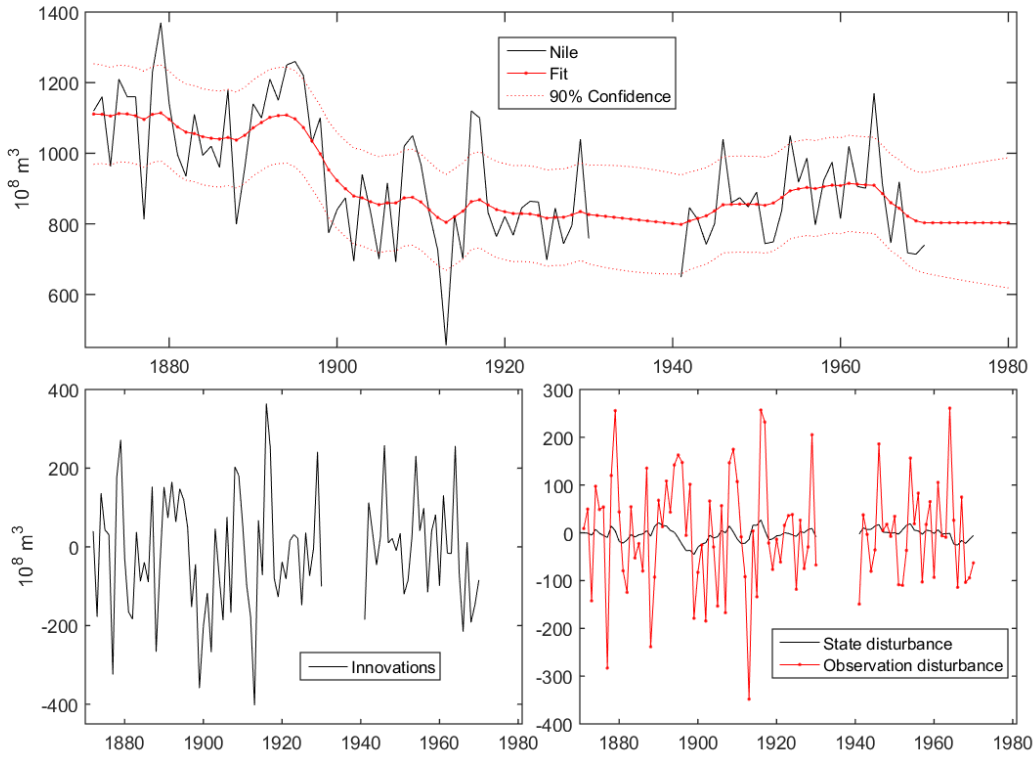

Figure 1: Fit, innovations and disturbances of `example1.m`.

This table shows abundant information about convergence of estimation, significance of parameters, information criteria, and diagnostic statistics on the innovations (autocorrelation, heteroskedasticity and Gaussianity). The resulting plots are shown in Figure 1.

Exactly the same results may be obtained if the model is estimated by concentrated maximum likelihood. Two slight changes have to be made to the previous code: (i) one of the variances in the user model has to be concentrated out of the likelihood by setting it to 1 (say, `model.Q = 1;` in `example1.m`); (ii) the call to `SSmodel` changed to `sys = SSmodel('y', y, 'model', @example1, 'OBJ_FUNCTION_NAME', @llikc)`. Beware that only the three initial characters are necessary in the string inputs arguments (i.e., `'OBJ'`, instead of `'OBJ_FUNCTION_NAME'`). See a more detailed description by running `SSdemo(1)`.

One of the challenges for this time series, is to evaluate whether the construction of the Aswan dam in 1899 (observation 29) led to a significant decline in the river flow. This may be tested in several ways by changing the user function, either by including a D_t in Equation 1 directly (Case 1 below), or by using a dummy variable as input to the SS system (Cases 2 and 3). It is worthy passing through these three cases to realize the flexibility of **SSpace** when specifying models.

Case 1

In this case the user function for concentrated maximum likelihood would be:

```
function model = example2(p)
  model.T = 1;
```

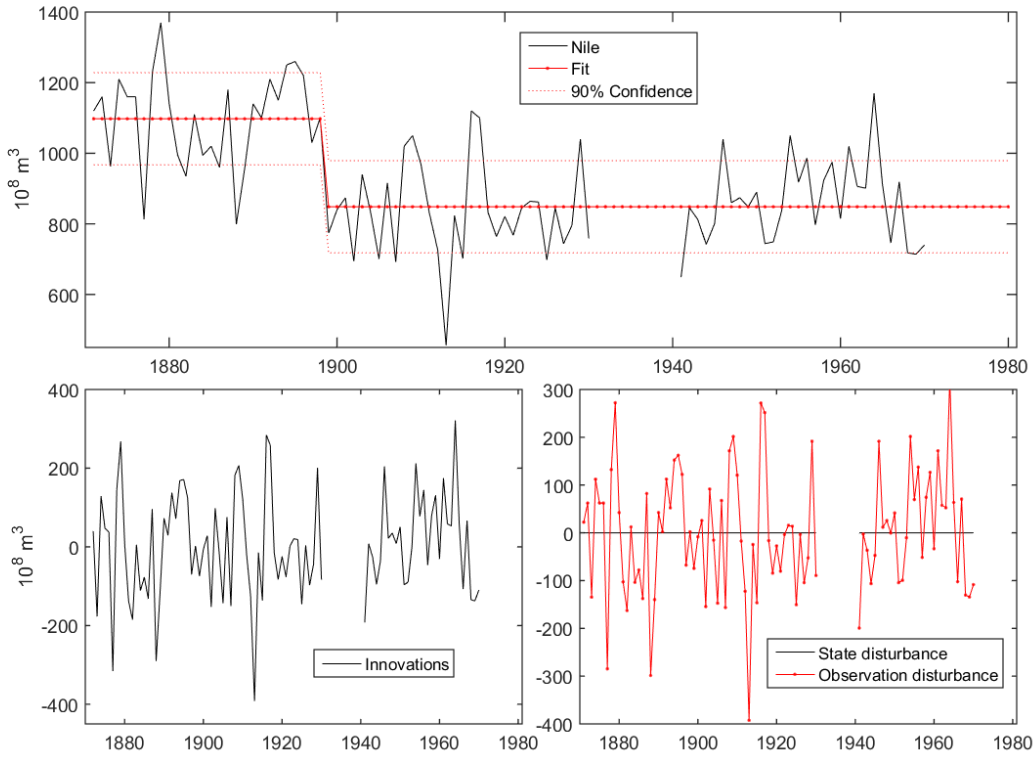

Figure 2: Fit, innovations and disturbances of `example2.m`.

```
model.Gam = [];
model.R = 1;
model.Z = 1;
model.D = [repmat(p(2), 1, 28) repmat(p(3), 1, 82)];
model.C = 1;
model.Q = 1;
model.H = 10.^p(1);
model.S = 0;
```

Keep in mind that matrix D_t is time-varying, but it is not defined as a three dimensional matrix, as it is the general convention in **SSpace**. This is an exception that affects also Γ_t and has been considered very convenient from the user point of view, since handling three dimensional matrices in MATLAB is much more cumbersome than two dimensional. Nevertheless, orthodox three dimensional matrices would work exactly in the same way. The call for estimating the model by concentrated maximum likelihood is `sys = SSmodel('y', y, 'model', @example2, 'OBJ_FUNCTION_NAME', @llikc, 'p0', [-1; 1000])`. An interesting point of this example is that a D_t matrix is used, but no input data is supplied and there is no need to specify a Γ_t matrix. Here, initial parameters for the numerical search are added via the duplet `{'p0', [-1; 1000]}`. The rest of the code is identical to the previous example. The results are shown in Figure 2.

It is important to note that the estimation is such that there is no additional information in the series apart from the jump in the volume due to the Aswan dam. By comparison with Figure 1, it is easy to check this, since the innovations and output are essentially the same.

Case 2

An alternative way to do the same, which is more formal from a statistical point of view, consists of defining one dummy variable as a step, taking zeros up to observation 28 and ones afterwards, i.e., `u = [zeros(28, 1); ones(82, 1)]`. In this case, the user function is:

```
function model = example3(p)
  model.T = 1;
  model.Gam = [];
  model.R = 1;
  model.Z = 1;
  model.D = p(2);
  model.C = 1;
  model.Q = 1;
  model.H = 10.^p(1);
  model.S = 0;
```

The difference with the previous option is that matrix `model.D` is just the second element of the parameter vector `p`, i.e., a coefficient that will multiply the input dummy variable u_t. Now the call to `SSmodel` should be:

```
sys = SSmodel('y', y, 'u', u, 'model', @example3, 'OBJ', @llikc);
```

Here, the duplet `{'u', u}` tells **SSpace** which is the input variable to use. The estimation of the coefficient measuring the jump is negative and significant.

Case 3

A final case, still worth mentioning, consists of including the dummy input variable into the user function as an additional input argument, but now the SS system is considered as a system without inputs:

```
function model = example4(p, u)
  model.T = 1;
  model.Gam = [];
  model.R = 1;
  model.Z = 1;
  model.D = p(2)*u;
  model.C = 1;
  model.Q = 1;
  model.H = 10.^p(1);
  model.S = 0;
```

The call now should be:

```
sys = SSmodel('y', y, 'model', @example4, 'OBJ', @llikc, 'user_inputs', {u});
```

Additional inputs to the user function may be passed on to the model definition with the help of duplets `{'user_inputs', {u}}`. Though it does not make sense in this case, it is

worth noting that the dependence to the inputs may be modeled by a nonlinear function, e.g., just by defining `model.D = p(2)*(u(2, :).^(p(3)))` or any other specification. This is the main advantages of specifying models by writing a function. More examples may be checked in `demo5`.

4.2. Example 2: Univariate models

In this second example the air-passenger data taken from Box, Jenkins, Reinsel, and Ljung (2015) is analyzed with a number of different univariate possibilities.

Case 1: Basic structural model (BSM)

A BSM *à la* Harvey (Harvey 1989) may be used with the template `SampleBSM` (it is also possible to use `SampleSS` and test the SS-form from scratch). In such a template, separate definitions are in place for the trend and the harmonics and input variables, if any. Have a look at the function `demo_airpasbsm.m` that is one of the cases implemented in `demo2`.

The part of the template related to the trends is:

```
m = ;
I = eye(m);
O = zeros(m);
TT = [I I;O I];
ZT = [I O];
RT = [I O; O I];
QT = [];
```

where the variable `m` is the number of output variables (1 for univariate) and matrices `I` and `O` are pre-defined as a unity matrix and a block of zeros. The model is specified directly by the state space form of the local linear trend type (LLT), i.e.,

$$\begin{pmatrix} \alpha_{1,t+1} \\ \alpha_{2,t+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha_{1,t} \\ \alpha_{2,t} \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \eta_{1,t} \\ \eta_{2,t+1} \end{pmatrix}$$

A LLT is fully specified by filling in the missing variables in the previous listing, i.e., `m = 1`, and `QT = [10.^p(1) 0; 0 10.^p(2)]`. Other possibilities are implemented by small variations to this option, one interesting case is an integrated random walk (IRW) or smooth trend with one single noise by setting `RT = [O; I]; QT = 10.^p(1);` (Young *et al.* 1999).

The periodic components (either seasonal or cyclical) in `SampleBSM` may be coded as either trigonometric or dummy seasonality. For the trigonometric case the empty code in the template is:

```
Period = [];
Rho = [];
Qs = repmat( , , );
```

This portion of the code filled in for a monthly time series converts to:

```
Period = [12 6 4 3 2.4 2];
Rho = [1 1 1 1 1 1];
Qs = repmat(varmatrix(p(3)), 1, 6);
```

The argument `Period` is used to provide the periods for the seasonal and/or cyclical, `Rho` is the damping factor of each harmonic sinusoid, i.e., values between zero and one and values of unity recommended for seasonal harmonics. The variances for each harmonic are introduced via the `Qs` matrix (all variances equal for all the harmonics in the example). If all variances have to be different the code would be `Qs = varmatrix(p(3:8)')`. The function `varmatrix` provides an alternative way of building semi-positive-definite covariance matrices.

The rest of the template is:

```
H = varmatrix(p(4));
D = [];
```

where `H` is the variance of the observed noise and matrix `D` is included to deal with input variables (none in this case).

Once the BSM model is set up, the code to run is as follows:

```
load airpas;
y = log(airpas);
sys = SSmodel('y', y, 'model', @demo_airpasbsm);
sys = SSestim(sys);
sys = SSsmooth(sys);
T = sys.a(1, :)';
I = y-sys.yfit';
S = y-I-T;
```

Here the trend, seasonal and irregular components are estimated by combinations of the estimated states and stored in `T`, `S` and `I` matrices, respectively.

Case 2: ARIMA

The ARIMA model is easily implemented by using the `SampleARIMA` template used as an example in `demo3`, check `demo_airpasarima`. Assuming that an $ARIMA(0,1,1) \times (0,1,1)_{12}$ is to be estimated, the relevant part of this template is:

```
Sigma = varmatrix(p(3));
Diffy = conv([1 -1], [1 zeros(1, 11) -1])';
ARpoly = 1;
MApoly = conv([1 p(1)], [1 zeros(1, 11) p(2)])';
```

where `Diffy` is the differencing polynomial in the backshift operator B such that $B^j y_t = y_{t-j}$, `ARpoly` is the AR polynomial and `MApoly` is the MA polynomial. Check that the differencing order is the convolution (i.e., multiplication) of a regular and seasonal difference operators, i.e., $\Delta\Delta^{12} = (1-B)(1-B^{12})$. The MA polynomial of the model is $(1+\theta_1 B)(1+\Theta_1 B^{12})$, with `p(1)`$= \theta_1$ and `p(2)`$= \Theta_1$. All polynomials are coded as column vectors of the corresponding coefficients in ascending order of powers of the backshift operator, as it is common in other MATLAB toolboxes. Though it does not make sense in this example, constraints among parameters would be very simple to impose, e.g., if the constraint $\theta_1 = \Theta_1$ is needed, the MA polynomial may be coded as `MApoly = conv([1 p(1)], [1 zeros(1, 11) p(1)])'`.

Case 3: Exponential smoothing

A final illustration for this data is an exponential smoothing model *à la* Hyndman, Koehler, Ord, and Snyder (2008), where the template is `SampleES`, check `demo_airpasES` used in `demo3`. The template filled in for this example is:

```
ModelType = 'AAA12';
Phi = [];
Alpha = constrain(p(1), [0 2]);
Beta =  constrain(p(2), [0 4-2*Alpha]);
AlphaS = constrain(p(3), [0 2]);
D = [];
Sigma = varmatrix(p(4));
```

The argument `ModelType` deals with the type of model, the first letter stands for the type of level ('N' for none, 'A' for additive and 'D' for damped with damping factor `Phi`), the second letter is the slope type ('N' for none or 'A' for additive), and the third letter stands for the type of seasonal (again either 'N' or 'A'), and the numbers after those letters is the period of the seasonal component. The rest of the code sets up the α, β and γ parameters affecting the level, slope and seasonal components, that are supposed to be estimated within certain values, see Hyndman *et al.* (2008). `Sigma` stands for the variance of the unique noise present in the model. Input variables may be included by introducing a `D` matrix as a function of the `p` vector of parameters.

4.3. Example 3: Multivariate dynamic harmonic regression (DHR)

The models presented so far are univariate and may be easily extended to multivariate versions. In this example we present a new model that has not yet been used in its multivariate version, namely the multivariate dynamic harmonic regression. It is built in the spirit of seemingly unrelated equations models, i.e., the relation among the endogenous variables is not explicit in the equations, but are modeled exclusively via non-diagonal covariance matrices of the different noises. The specific formulation is given in Equation 6.

$$\begin{aligned} y_t &= T_t + C_t + S_t + f(u_t) + \epsilon_t \\ C_t + S_t &= \textstyle\sum_{i=1}^{k} [A_{it} \cos(\omega_i t) + B_{it} \sin(\omega_i t)] \end{aligned} \tag{6}$$

Obviously y_t is multivariate in this case; T_t, C_t, S_t are a set of trends, cycles and seasonals, respectively; $f(u_t)$ models linear or nonlinear relations to inputs; ϵ_t is a white noise variable with full covariance matrix H_t; ω_i $(i = 1, 2, \ldots, k)$ are the frequencies for the periodic components that typically include the seasonal fundamental frequency and all the harmonics down to π but also may include cyclical frequencies; and A_{it} and B_{it} are diagonal matrices of time-varying parameters. Usually random walk parameters may suffice, but other options may be preferred in specific applications. In essence the DHR model is effectively a multivariate Fourier analysis in particular frequencies with time-varying parameters. The univariate version of these models estimated in the frequency domain where explored by Young *et al.* (1999) and Taylor *et al.* (2007).

These type of models may be easily implemented in **SSpace** with the help of the template `SampleDHR`. This template resembles `SampleBSM` of the previous example, with a user function

that needs an additional input argument that have to be taken into account in the `SSmodel` call, i.e., the size in time of the sample. In the case of a trivariate model of quarterly data is implemented in `demo_energydhr` used in `demo4`. A compacted version of such template is

```
function model = demo_energydhr(p, N)
% Trend
m = 3;
I = eye(m); O = zeros(m);
TT = [I I;O I];
ZT = [I O];
RT = [I O; O I];
QT = blkdiag(varmatrix(p(1:6)), varmatrix(p(7:12)));
% Seasonal/cyclical DHR components
Periods = repmat([4 2], 3, 1);
Rho = ones(3, 2);
Qs = repmat(varmatrix(p(13:18)), 1, 2);
% Covariance matrix of irregular component (observed noise)
H = varmatrix(p(19:24));
```

There are a total of 24 unknown parameters, all of them located at covariances matrices. It is important to note here the use of the `varmatrix` function, used to convert any set of arbitrary parameters into a semi-positive covariance matrix. Beware that, since covariance matrices are symmetrical, a 3×3 matrix have 9 elements, but only 6 of them are different. This function allows also to impose rank and other constraints to build homogeneous models, etc. (see the **SSpace** documentation). It is also easy to check that there are only two harmonics and both are estimated with the same covariance matrices, check the `Qs` matrix.

Such model may be run on the energy data of Harvey (1989), with the following code:

```
load energy
y = log(energy);
p0 = repmat([-2 -2 -2 0 0 0]', 4, 1);
sys = SSmodel('y', y, 'model', @demo_energydhr, 'p0', p0, ...
  'user_inputs', length(y));
sys = SSestim(sys);
sys = SSsmooth(sys);
Trend = sys.a(1:3, :)';
Irregular = y-sys.yfit';
Seasonal = y-Trend-Irregular;
```

It is important to initialize the searching algorithm with appropriate diagonal covariance matrices, this is achieved by the setting of `p0` above. The call to `SSmodel` is done with the required initial parameters and the number of time samples.

Figure 3 shows some output of this DHR model.

4.4. Example 4: Non-Gaussian

The number of van driver casualties in the UK, see Durbin and Koopman (2012) and Figure 4, is a case where a Poisson distribution is justified, because numbers are small and the units

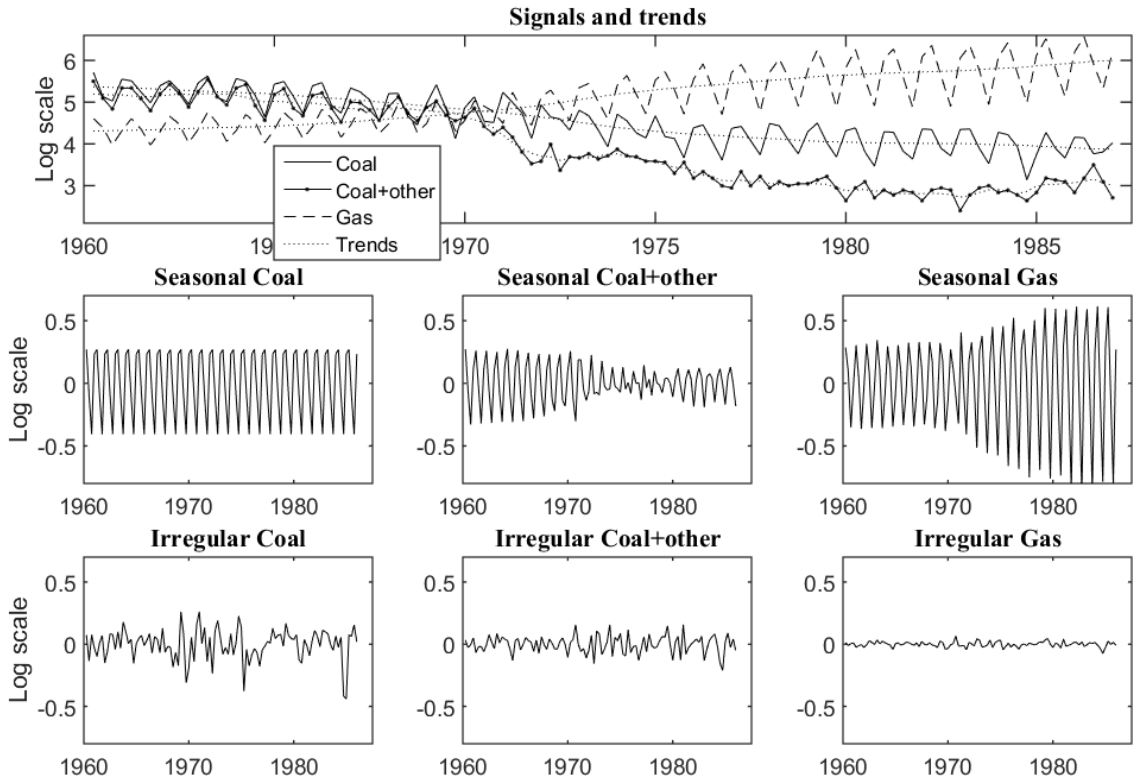

Figure 3: Unobserved components of the DHR example.

are integers. Neither the Gaussian assumption is strictly correct nor does the logarithmic transformation manage to produce sensible results. The model used is a BSM with a trend, dummy seasonal, irregular and an exogenous effect due to the enforcement of the seat belt law, but the distribution of the observations is a Poisson, see Equation 7.

$$\begin{aligned} \alpha_{t+1} &= T_t \alpha_t + \Gamma_t + R_t \eta_t \\ p(y_t \mid \theta_t) &= \exp\{\theta_t y_t - \exp(\theta_t) - \log y_t!\} + bI_t \end{aligned} \tag{7}$$

In order to set up this model in **SSpace** we have to create two user functions, one with the linear model, i.e., the BSM with a dummy seasonal (note that there is no noise in the observation equation) by using the `SampleBSM` template and a second one to change the observation equation into a Poisson model with the help of `SampleEXP` template (other distributions available are Binary, Binomial(n), Negative Binomial and Exponential). Both functions may be checked in `demo_van_poisson`, used in `demo7`. The linear model is (function `demo_van` in file `demo_vanl.m`):

```
function model = demo_van(p, H)
% Trend
TT = 1;
ZT = 1;
RT = 1;
QT = varmatrix(p(1));
% Dummy seasonal
```

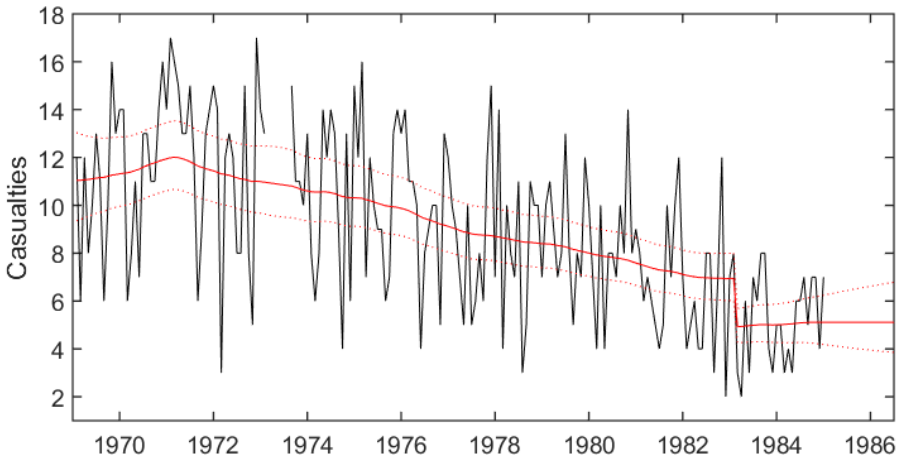

Figure 4: Data and trend of non-Gaussian example.

```
Period = 12;
Qs = 0;
% Linear term
D = p(2);
```

As it may be seen, the model is just a random walk trend with a dummy seasonal component and a dummy effect dealing with the step change due to the seat belt law enforcement.

The second user model is rather simple, it consists of a call to the linear model above and just telling **SSpace** that the observations follow a Poisson distribution. This function has the peculiarity that it necessarily has to have two input arguments, while the user may add as many as the model requires for its definition:

```
function model = demo_van_poisson(p, H)
% Call to linear model
model = demo_van(p, H);
% Distribution of observations
model.dist = Poisson;
```

Then, the following code would produce the analysis for the data with some artificial missing values in the middle and some at the end to produce the forecasts:

```
load Van
y = Van(:, 1);
u = Van(:, 2);
y(50:55)= nan;
y = [y; nan(20, 1)];
u = [u; ones(20, 1)];
sys = SSmodel('y', y, 'u', u, 'model', @demo_van_poisson);
sys = SSestim(sys);
sys = SSvalidate(sys);
sys = SSsmooth(sys);
```

Figure 4 shows the trend with the jump due to the seat-belt law effect and twice the standard error confidence bands.

4.5. Example 5: Nonlinear

This example illustrates the use of the extended Kalman filter with exact initialization applied to the monthly visits abroad by UK residents from January 1980 to December 2006 following Koopman and Lee (2009) and Durbin and Koopman (2012). This example was used to test for the convenience of the log transform so widely use in econometrics. As a matter of fact, it is shown that the log transform does not fix the heteroskedasticity problem, and, consequently a model with an interaction between the trend and the seasonal component is proposed instead while the rest of hypothesis are applied, see Equation 8.

$$y_t = \text{Trend}_t + \text{Cycle}_t + \exp\{b\text{Trend}_t\}\text{Seasonal}_t + \epsilon_t \tag{8}$$

Comparing this equation with the general nonlinear system (3) we see that there is only one nonlinear term, namely $T_t(\alpha_t)$ that is the Equation 3 without the noise. Setting up the model now is much more difficult because the partial derivatives of $T_t(\alpha_t)$ with respect to the vector state α_t ought to be calculated explicitly. For convenience a linear BSM model is implemented in a separate model using the `SampleBSM` template (check `demo_uk` in `demo8`):

```
function model = demo_uk(p)
TT = [1 1;0 1];
ZT = [1 0];
RT = [0; 1];
QT = varmatrix(p(1));
% Trigonommetric seasonal model
Period = [constrain(p(3), [18 800]) 12 6 4 3 2.4 2];
Rho = [constrain(p(4), [0.5 1]) 1 1 1 1 1 1];
Qs = [varmatrix(p(5)) repmat(varmatrix(p(6)), 1, 6)];
% Observed noise variance
H = varmatrix(p(2));
```

This model has some singularities with respect to previous listings in this paper. Firstly, an integrated random walk or smooth trend is chosen that depends on just one parameter (`p(1)`). Secondly, a cycle is introduced into the model by adding one element to all the arguments in the function having to do with the seasonal component. Thirdly, one interesting point is that the period of such a cycle is estimated as a constrained value between 18 and 800 months (one and a half year and 67 years, see the use of `constrain` in `Period`). Fourthly, the cycle is modulated by a damping factor estimated as a value between 0.5 and 1 (again with the help of the `constrain` function). Finally, separate variance values for the cycle and the seasonal components are estimated.

The nonlinear model is now built with the template `SampleNL` (check `model_uknle`):

```
function model = demo_uknle(p, at, ctrl)
model1 = demo_uk(p(1:6));
% Defining linear matrices
if ctrl< 2
```

```
    model.T = model1.T;
    model.Gam = [];
    model.R = model1.R;
    model.Z = [];
    model.D = [];
    model.C = 1;
    model.Q = model1.Q;
    model.H = model1.H;
    model.p = p;
    model.S = [];
end
% Defining nonlinear matrices in State Equation
if any(ctrl== [2 0])
    % Code defining derivative of matrix T(a(t)) (ns x Nsigma)
    model.dTa = [];
    % Code defining matrix T(a(t)) (ns x Nsigma)
    model.Ta = [];
    % Code defining matrix R(a(t)) (ns x neps x (1 or n))
    model.Ra = [];
    % Code defining matrix Q(a(t)) (neps x neps x (1 or n))
    model.Qa = [];
end
% Defining nonlinear matrices in Observation Equation
if any(ctrl== [3 0])
    b = p(7);
    expfun = exp(b*at(1));
    S = sum(at(5:2:15));
    % Derivative of matrix Z(a(t)) (m x Nsigma)
    model.dZa = [1+b*expfun*S 0 1 0 expfun 0 expfun 0 expfun ...
                            0 expfun 0 expfun 0 expfun];
    % Code defining matrix Z(a(t)) (m x Nsigma)
    model.Za = at(1)+at(3)+expfun*S;
    % Code defining matrix C(a(t)) (Ny x Neps x (1 or n))
    model.Ca = [];
    % Code defining matrix H(a(t)) (Ny x Ny x (1 or n))
    model.Ha = [];
end
```

This template has three compulsory input arguments, i.e., the parameter vector `p`, the current state vector `at`, and a variable that controls the execution (`ctrl`). As a first step, the function calls the linear model `demo_uk` in order to set up all the matrices in the state equation. Then the system matrices are redefined in three blocks, (i) linear matrices, (ii) nonlinear or state-dependent matrices in the state equation, (iii) nonlinear or state-dependent matrices in the observation equation. Only one definition for each matrix in any of the blocks is permitted, in order to avoid errors. Beware that both $Z_t(\alpha_t)$ (in `model.Za`) and $\frac{\partial Z_t(\alpha_t)}{\partial \alpha_t}$ (`model.dZa`) have to be correctly specified.

Figure 5 shows the estimated components.

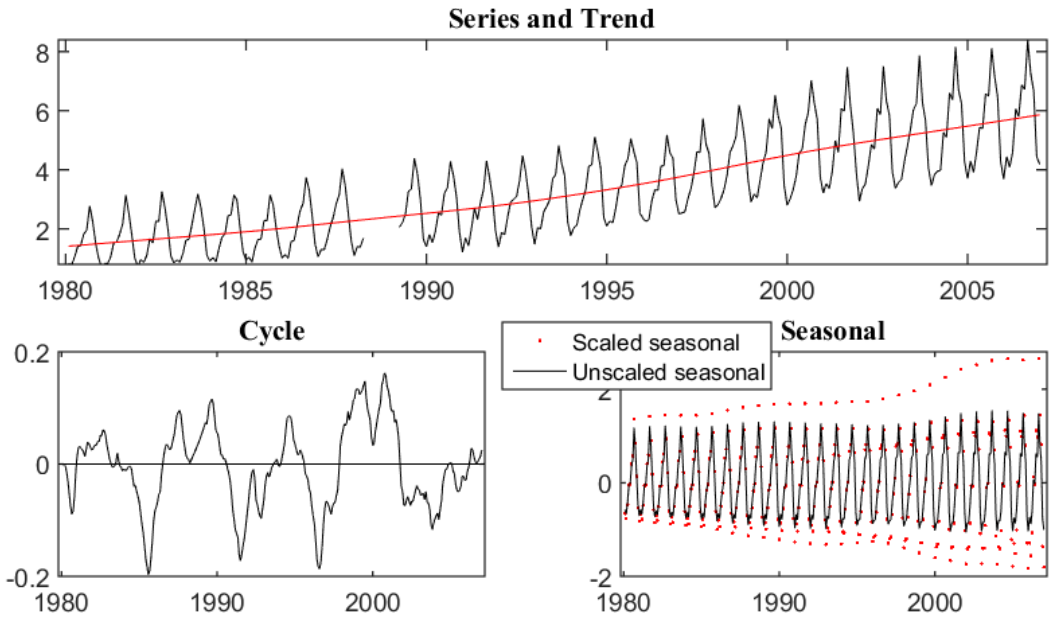

Figure 5: Series and trend, cycle and raw seasonal and scaled seasonal ($\exp\{b\text{Trend}_t\}\text{Seasonal}_t$) of Nonlinear example.

4.6. Further features

More templates and more demos are available in **SSpace**, but are left out of this paper for the sake of clarity and space. They may be checked by running carefully the 8 demos included in the toolbox. Some of the most relevant are linear, nonlinear and time-varying regressions (see `SampleDLR` and `demo5`), concatenation of state space systems with the `SampleCAT` template (a typical case would be a BSM model with time-varying parameters), estimating parameters of any model by minimizing functions of squared of several-steps-ahead forecast errors (see e.g., `demo2`), time aggregation problems (see `SampleAGG` and `demo6`), and nesting in inputs models (see `SampleNEST` and `demo5`).

5. Concluding remarks

This paper has presented **SSpace**, a new toolbox for the exploitation of state space models. It is intended for a wide audience, including professional practitioners, researchers and students, indeed anyone involved in the analysis of time series, forecasting or signal processing.

The library incorporates most modern algorithms and advances in the field of state space modeling, following mainly Taylor *et al.* (2007) and Durbin and Koopman (2012). The system is very general because it is possible to implement linear, non-Gaussian and nonlinear systems, all system matrices may vary over time and may be multivariate, several estimation methods are implemented, inputs to the system may be introduced explicitly, etc.

Other advantages of the library are that a few functions are necessary to carry out a comprehensive analysis of time series. Such functions are used systematically following a fixed pattern that simplifies the usage of the toolbox. However, one of the main feature that makes **SSpace** really flexible, powerful and transparent is that the user implements models directly

as a function written in MATLAB. This approach makes some cumbersome tasks truly simple and transparent, this is the case, say of trying different parameterizations of the same model or imposing parameter constraints.

The toolbox is supplied with templates for building general SS models from scratch in a completely free way, but is also accompanied by a number of templates useful for the implementation of a variety of common models.

In this paper, the capabilities of the toolbox have been demonstrated in action on several worked examples. These properties should make the toolbox particularly interesting for those in need of non-standard models, for which even many commercial alternatives may not provide the required flexibility.

Acknowledgments

We acknowledge the interesting comments of J.R. Trapero from UCLM and C.J. Taylor from Lancaster University, that substantially improved the manuscript.

This work was supported by the European Regional Development Fund and Spanish Government (MINECO/FEDER, UE) under the project with reference DPI2015-64133-R and by the Vicerrectorado de Investigación y Política Científica from UCLM by DOCM 31/07/2014 [2014/10340].

References

Box GEP, Jenkins GM, Reinsel GC, Ljung GM (2015). *Time Series Analysis: Forecasting and Control*. John Wiley & Sons.

Casals JM, Garcia-Hiernaux A, Jerez M, Sotoca S, Trindade AA (2016). *State-Space Methods for Time Series Analysis: Theory, Applications and Software*. Chapman & Hall/CRC.

Commandeur JJF, Koopman SJ, Ooms M (2011). "Statistical Software for State Space Methods." *Journal of Statistical Software*, **41**(1), 1–18. doi:10.18637/jss.v041.i01.

Doan T (2011). "State Space Methods in **RATS**." *Journal of Statistical Software*, **41**(9), 1–16. doi:10.18637/jss.v041.i09.

Drukker DM, Gates RB (2011). "State Space Methods in Stata." *Journal of Statistical Software*, **41**(10), 1–25. doi:10.18637/jss.v041.i10.

Durbin J, Koopman SJ (2012). *Time Series Analysis by State Space Methods*. 38. Oxford University Press.

Gómez V (2015). "**SSMMATLAB**: A Set of MATLAB Programs for the Statistical Analysis of State Space Models." *Journal of Statistical Software*, **66**(9), 1–37. doi:10.18637/jss.v066.i09.

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

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

Hyndman R, Koehler AB, Ord JK, Snyder RD (2008). *Forecasting with Exponential Smoothing: The State Space Approach*. Springer-Verlag. doi:10.1007/978-3-540-71918-2.

IHS Inc (2010). *EViews 7 for Windows*. Irvine. URL http://www.eviews.com/.

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

Koopman SJ, Lee KM (2009). "Seasonality with Trend and Cycle Interactions in Unobserved Components Models." *Journal of the Royal Statistical Society C*, **58**(4), 427–448. doi:10.1111/j.1467-9876.2009.00661.x.

Koopman SJ, Shephard N, Doornik JA (2008). *Statistical Algorithms for Models in State Space Form: **SSfPack** 3.0*. Timberlake Consultants.

Lucchetti R (2011). "State Space Methods in **gretl**." *Journal of Statistical Software*, **41**(11), 1–22. doi:10.18637/jss.v041.i11.

Pedregal DJ, Trapero JR (2012). "The Power of **ECOTOOL** MATLAB Toolbox." In *Industrial Engineering: Innovative Networks*, pp. 319–327. Springer-Verlag.

Peng JY, Aston JAD (2011). "The State Space Models Toolbox for MATLAB." *Journal of Statistical Software*, **41**(6), 1–26. doi:10.18637/jss.v041.i06.

Petris G, Petrone S (2011). "State Space Models in R." *Journal of Statistical Software*, **41**(4), 1–25. doi:10.18637/jss.v041.i04.

R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

SAS Institute Inc (2013). *The SAS System, Version 9.4*. SAS Institute Inc., Cary. URL http://www.sas.com/.

Selukar R (2011). "State Space Modeling Using SAS." *Journal of Statistical Software*, **41**(12), 1–13. doi:10.18637/jss.v041.i12.

StataCorp (2017). *Stata Statistical Software: Release 15*. StataCorp LLC, College Station. URL http://www.stata.com/.

Taylor CJ, Pedregal DJ, Young PC, Tych W (2007). "Environmental Time Series Analysis and Forecasting with the **Captain** Toolbox." *Environmental Modelling & Software*, **22**(6), 797–814. doi:10.1016/j.envsoft.2006.03.002.

The MathWorks Inc (2017). *MATLAB – The Language of Technical Computing, Version R2017a*. Natick. URL http://www.mathworks.com/products/matlab/.

Van den Bossche FAM (2011). "Fitting State Space Models with **EViews**." *Journal of Statistical Software*, **41**(8), 1–16. doi:10.18637/jss.v041.i08.

Young PC, Pedregal DJ, Tych W (1999). "Dynamic Harmonic Regression." *Journal of Forecasting*, **18**(6), 369–394. doi:10.1002/(sici)1099-131x(199911)18:6<369::aid-for748>3.0.co;2-k.

Affiliation:

Marco A. Villegas, Diego J. Pedregal
ETSI Industriales
Institute of Applied Mathematics in Science and Engineering (IMACI)
Universidad de Castilla-La Mancha
13071 Ciudad Real, Spain
Telephone: +34/(9)26/295430
E-mail: Marco.Villegas@uclm.es, Diego.Pedregal@uclm.es
URL: http://www.uclm.es/profesorado/diego/

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
November 2018, Volume 87, Issue 5 *Submitted:* 2016-04-28
doi:10.18637/jss.v087.i05 *Accepted:* 2017-06-24