SSpace Manual

User guide

Author: Diego J. Pedregal & Marco A. Villegas

30 September 2020

Contents

1 SSpace Overview

SSpace is a `MATLAB` toolbox ([1]) providing a number of routines designed for a general analysis of State Space systems (SS henceforth) combining both flexibility and simplicity, enhancing the power and versatility of SS modeling, all together in a easy-to-use framework. It is built up mostly following the methods shown in [2], but there are also additional bits taken from many other sources, mainly [3], that all together provide **SSpace** with a particular flavour. Further readings dealing with this toolbox and **SSpace** for practitioners may be seen in [4, 5].

In a broad sense, **SSpace** provides the user with the most advanced and up-to-date features available in the SS framework. The flexibility and easiness of use is reflected in the fact that **SSpace** was designed keeping in mind the final user and the usability of the toolbox, by selecting easy-to-remember function names, and more importantly, by allowing a direct correspondence between the analytical expression of models (e.g., written on paper) and the corresponding definition in `MATLAB` code. In addition, users are also provided with a set of model-templates for approaching many standard models with maximum simplicity. A full system of help is included individually for each function and eight step-by-step demos are included that demonstrate the use of the toolbox with standard well-known examples and others much less standard.

Usually default options for modelling are in place in a way such that a few commands with a few options would produce sensible results, but advanced users may take advantage of the possibility of configuring the toolbox at their own convenience.

1.1 Linear models in SSpace

The linear Gaussian models in **SSpace** is shown in equation (1).

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

In this equation 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 $S_t = Cov(\eta_t, \epsilon_t)$ 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 system matrices with appropriate dimensions, i.e.,

$$\begin{array}{cccccc} T_t\text{:} & n \times n; & \Gamma_t\text{:} & n \times 1; & R_t\text{:} & n \times r; \\ Z_t\text{:} & m \times n; & D_t\text{:} & m \times 1; & C_t & m \times h. \end{array} \tag{2}$$

One interesting feature is that the toolbox is flexible enough to allow the terms Γ_t and D_t be defined in a way such 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 dimensions $k \times 1$. Beware that general functions $f(\bullet)$ and $g(\bullet)$ include as particular cases a linear function $\Gamma_t = \gamma_t u_t$ and $D_t = d_t u_t$, with γ_t and d_t of dimensions $n \times k$ and $m \times k$, respectively.

1.2 Non-Gaussian models in SSpace

The non-Gaussian SS set up is shown in equation (3).

$$\begin{array}{rl} \alpha_{t+1} = T_t\alpha_t + \Gamma_t + R_t\eta_t, & \eta_t \sim N(0, Q_t) \\ y_t \sim p(y_t \mid \theta_t) + D_t, & \\ \theta_t = Z_t\alpha_t & t = 1, 2, \ldots, N \end{array} \tag{3}$$

Here θ_t is known as the signal. With this representation it is possible to deal with three types of models [2]:

* Exponential family distribution, where $p(y_t \mid \theta_t) = exp[y_t'\theta_t - b_t(\theta_t) + c_t(y_t)], -\infty < \theta_t < \infty$. * Stochastic Volatility models, i.e., $y_t = exp(\frac{1}{2}\theta_t)\epsilon_t + d_t$. * Observations generated by the relation $y_t = \theta_t + \epsilon_t, \epsilon_t \sim p(\epsilon_t)$, with $p(\bullet)$ being a non-Gaussian distribution.

1.3 Non-linear models in SSpace

Finally, the non-linear models are of the type shown in equation (4).

$$\begin{array}{rl} \alpha_{t+1} = T_t(\alpha_t) + \gamma_t + R_t(\alpha_t)\eta_t, & \eta_t \sim N(0, Q_t(\alpha_t)) \\ y_t = Z_t(\alpha_t) + d_t + C_t(\alpha_t)\epsilon_t, & \epsilon_t \sim N(0, H_t(\alpha_t)) \\ & t = 1, 2, \ldots, N \end{array} \tag{4}$$

Functions $T_t(\alpha_t)$ and $Z_t(\alpha_t)$ with first derivatives provide non-linear transformations of the state vector into vectors of sizes $n \times 1$ and $m \times 1$, respectively. The rest of 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 manual refer to [2] and [3].

2 SSpace general workflow

2.1 List of usable functions in SSpace

Table 1 shows the core functions necessary to carry out a comprehensive time series analysis. Beware that all the names start with 'SS' in order to make them easy to remember or to search for. From all this 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 model setting (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 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 that implements the model, either exact or diffuse or ad-hoc initialization of recursive algorithms, initial conditions for parameter estimation, fixed parameters that would be frozen in estimation, the cost function to optimize, exact or numerical score (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.

`SS`	Run all functions
`SSmodel`	Set up the model
`SSestim`	Parameter estimation
`SSvalidate`	Prints estimation results and diagnostics
`SSfilter`	(Extended) Kalman Filter
`SSsmooth`	(Extended) Fixed Interval Smoother
`SSdisturb`	Disturbance Smoother
`SSdemo`	Run **SSpace** demos 1 to 8

Table 1: Main functions of the **SSpace** Library

Once the model is set up, `SSestim` performs parameter estimation by the method previously selected in `SSmodel`. `SSvalidate` produces a table with the estimation results and diagnostics. `SSfilter` produce the innovations and filtered estimates of states and covariance matrices with additional output. If smoothed output is prefered, it is produced by the `SSsmooth` function. Disturbance errors (and smoothed output) may be computed by `SSdisturb`. Finally, there are eight step-by-step demos ready, that may be run with the `SSdemo` function.

All the functions in Table 1 run on a 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, shown in Table 2. Beware that all the template names start with 'Sample' in order to make them easy to remember or to search for. `SampleSS` set up any linear and gaussian models with or without inputs and any sort of non-standard feature. The rest of linear models are self-explanatory and are restricted to standard models. There are also

some templates for non-Gaussian models and for general non-linear models. 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.

Linear and Gaussian models	
`SampleSS`	General SS template
`SampleARIMA`	ARIMA models with eXogenous variables
`SampleVARMAX`	VARMAX models
`SampleDHR`	Dynamic Harmonic Regression
`SampleDHRt`	Dynamic Harmonic Regression with irregular sampling
`SampleBSM`	Basic Structural Model
`SampleSH`	BSM with heteroskedastic dummy seasonality
`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`	Sochastic volatility models
General non-linear models	
`SampleNL`	General non-linear models
Other templates	
`SampleAGG`	Models with time aggregation
`SampleCAT`	Concatenation of State Space systems
`SampleNEST`	Nesting in inputs State Space systems

Table 2: Available templates for the **SSpace** toolbox

The rest of 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, this is rather useful, because variances coming out of filtering and the rest of functions are three-dimensional, ii) `constrain` settles constraints among parameters in the models, iii) `varmatrix` is a function usefull to constrain covariance matrices to be semi positive definite in multivariate models or just positive in scalar cases, iv) `normalize` standardizes any data, v) `vdiff`, `vconv`, `vfilter` and `vroots` produce different operations useful for vector time series, namely differencing, convolutional product, filter and roots of multivariate polynomials and vector time series, respectively, and vi) `optimizer` is an editable script that allows tunning the optimizer tolerances and even to change the optimizer itself.

2.2 General usage of SSpace

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

1. Write the model in a paper or specify the model in SS form.

`confband`	Confidence intervals
`constrain`	Constraints of parameters
`varmatrix`	Semi definite positive covariance matrices
`dhrMatrix`	Builds matrix of time varying periods to use in `SampleDHRt`
`normalize`	Variable normalization (standarization)
`vdiff`	Differentiation of vector time series
`vconv`	Convolution of vector polynomials
`vfilter`	Filter of vector time series with vector polynomial
`vroots`	Roots of vector polynomial
`optimizer`	Optimizer options

Table 3: Auxiliar functions for the **SSpace** toolbox

2. Translate model to familiar code by using `MATLAB` templates.

3. Set up model (`SSmodel`).

4. Estimate unknown parameters (`SSestim`).

5. Check appropriateness of model (`SSvalidate`).

6. Determine optimal estimates of states, their covariance matrices, innovations, forecasts, etc. (`SSfilter`, `SSsmooth` or `SSdisturb`).

Function `SS` runs all steps from 3. to the end.

In the rest of this chapter, a local level model (or random walk plus noise model) is used to illustrate how to implement all these steps applied to the Nile river data used in [2], actually `demo1` of **SSpace** (it may be accessed by editing and running `demo1` directly or by running `SSdemo(1)` at the `MATLAB` prompt). 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 (5), being B the back-shift operator (i.e., $B^k y_t = y_{t-k}$).

$$y_t = \frac{\eta_t}{(1-B)} + \epsilon_t; \quad VAR(\eta_t) = Q \quad VAR(\epsilon_t) = H \tag{5}$$

2.2.1 Specify model (step 1)

One SS representation of (5) is (6).

$$\begin{array}{rll} \text{State Equation:} & \alpha_{t+1} = \alpha_t + \eta_t & \eta_t \sim N(0, Q) \\ \text{Observation Equation:} & y_t = \alpha_t + \epsilon_t & \epsilon_t \sim N(0, H) \end{array} \tag{6}$$

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 (5) it is easy to 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.

2.2.2 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 matrice 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;
```

Beware 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 `Q=varmatrix(p(1))`.

2.2.3 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(nile, 'model', @example1);
```

It is assumed that `nile` is a vector variable already available in `MATLAB` containing the Nile data. Basically with this code the user is telling that he wants to apply the local level model written in `example1` to the data in `nile`.

This is the most important step because at this stage is when the user defines the posterior performance of the library. In essence, if the validation of the model says the model is not correct, then the user has to come back to `example1` and redifine the model, or to `SSmodel` and change the model options.

There are many options available to set up the model (see Chapter 4), that are passed on to `SSmodel` using duplets, but most of them have default values assigned. For example, it is possible to select the input and output data, the model filename, additional inputs to the user-coded function that implements the model, initial values for parameters, fixed values for parameters, diffuse or exact initializations of algorithms, objective cost function and analytical or exact score in Maximum Likelihood estimation, etc.

2.2.4 Parameter estimation (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. Parameters are stored in `sys` output structure. If 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);
```

2.2.5 Use the model (step 5)

A final step consists of estimating the filtered and/or smoothed output together with the disturbances of the model, for further validation tests. These operations constitute the basis for forecasting, signal extraction, interpolation, etc. When only the filtered output is required the coding is

```
sys= SSfilter(sys);
```

If smoothed estimates without disturbances is preferred then the code is

```
sys= SSsmooth(sys);
```

The full output is produced by

```
sys= SSdisturb(sys);
```

Results are stored in `sys` output structure. It stores parameters with covariance matrix, optimal states and covariances, fitted output values and covariances, forecasted values and covariances, innovations, disturbances estimates with covariances (see Chapter 4). Further statistical diagnostics are advisable.

3 SSpace examples

The examples shown in this chapter are shown as illustrations of the flexibility and power of the toolbox. Code listings are truncated in order to save space, this is especially important in the case of the templates shown, because all of them are supplied with an abundant help.

3.1 Local level model

Putting 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, the resulting code is:

```
% load data and set up time variable
load nile
t= (1 : size(nile, 1))';
y= [nile; nan(10, 1)];
y(61:70)= nan;
t= (1 : size(y, 1))';
% Build \textbf{SSpace} model
sys= SSmodel(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') % More results
```

This listing may be simplified by using function SS replacing the calls to SSmodel, SSestim, SSvalidate and SSsmooth, i.e.,

```
...
t= (1 : size(y, 1))';
% Build \textbf{SSpace} model, estimate, show table and smooth
sys= SS(y, 'model', @example1);
% Plotting fitted values with 90% confidence bands
...
```

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. Part of the output of the SSvalidate call is:

```
-----------------------------------------------------
Linear Gaussian model: demo_nile.
Objective function: llik
System colapsed at observation 1 of 100.
Exact gradient used.
-----------------------------------------------------
         Param    S.E.    T-test  P-value |Gradient|
-----------------------------------------------------
```

```
 p(1)    3.1404   0.3593    8.7406    0.0000    0.000000
 p(2)    4.2084   0.0855   49.2258    0.0000    0.000006
-------------------------------------------------------
                AIC: 12.906
                SBC: 12.9938
                HQC: 12.9398
    Log-likelihood: -571.3177
      Corrected R2: 0.2669
Residual Variances: 21919.3612
-------------------------------------------------------
```

The resulting plots are shown in Figure 1.

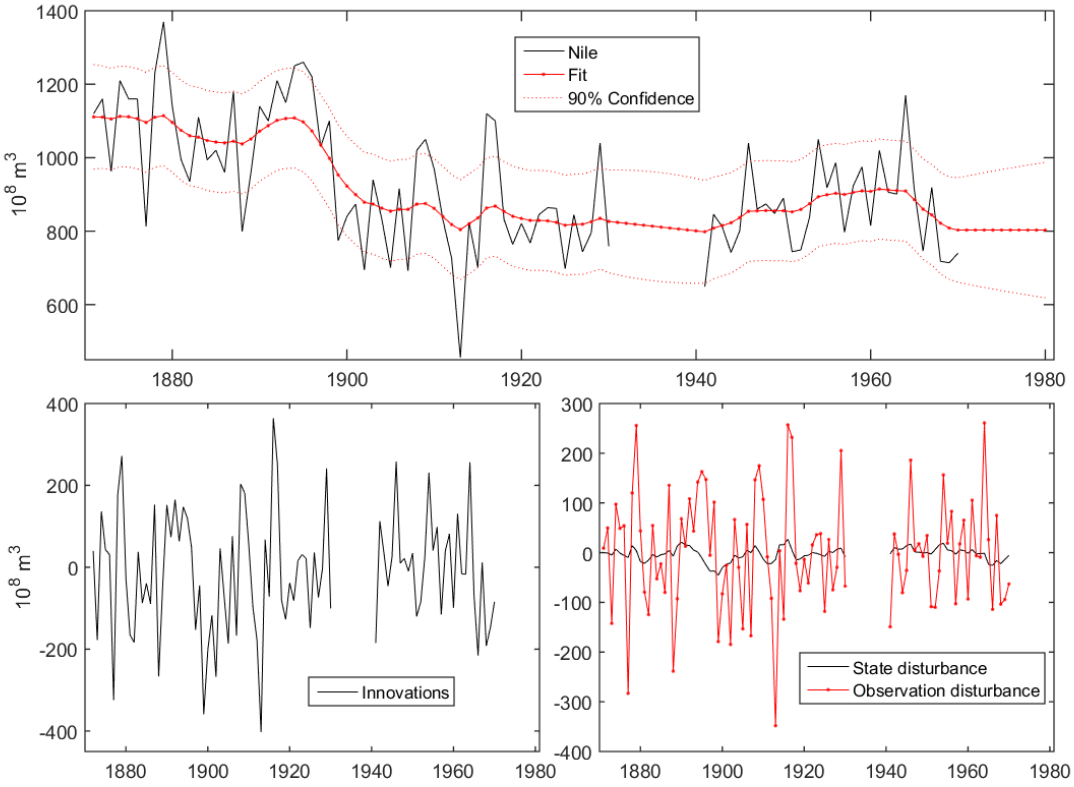

Figure 1: Fit, innovations and disturbances of `example1`.

Exactly the same results may be obtained if the model is estimated by Concentrated Maximum Likelihood with two slight changes: i) one of the variances in the user model has to be concentrated out of the likelihood by setting it to 1 (e.g., `model.Q= 1`); ii) the call to `SSmodel` changes to `sys= SSmodel(nile, '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'` is valid). 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 tested in several ways by changing the user function, either by including a D_t 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.

3.1.1 Case 1

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

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

Beware that matrix D_t is time varying, but it is not defined as a three dimensional matrix, as is the general convention in **SSpace**. This is an exception that affects also to Γ_t and have been considered very convenient from the user point of view, since handling three dimensional matrices in `MATLAB` is much more cumbersome than two dimensional matrices. Nevertheless, three dimensional matrices would work exactly in the same way. The call for estimating the model by Concentrated Maximum Likelihood is `sys= SSmodel(nile, 'model', @example2, 'OBJ_FUNCTION_NAME', @llikc, 'p0', [-1; 1000])`. An interesting point of this example is that a D_t matrix is used, but not 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.

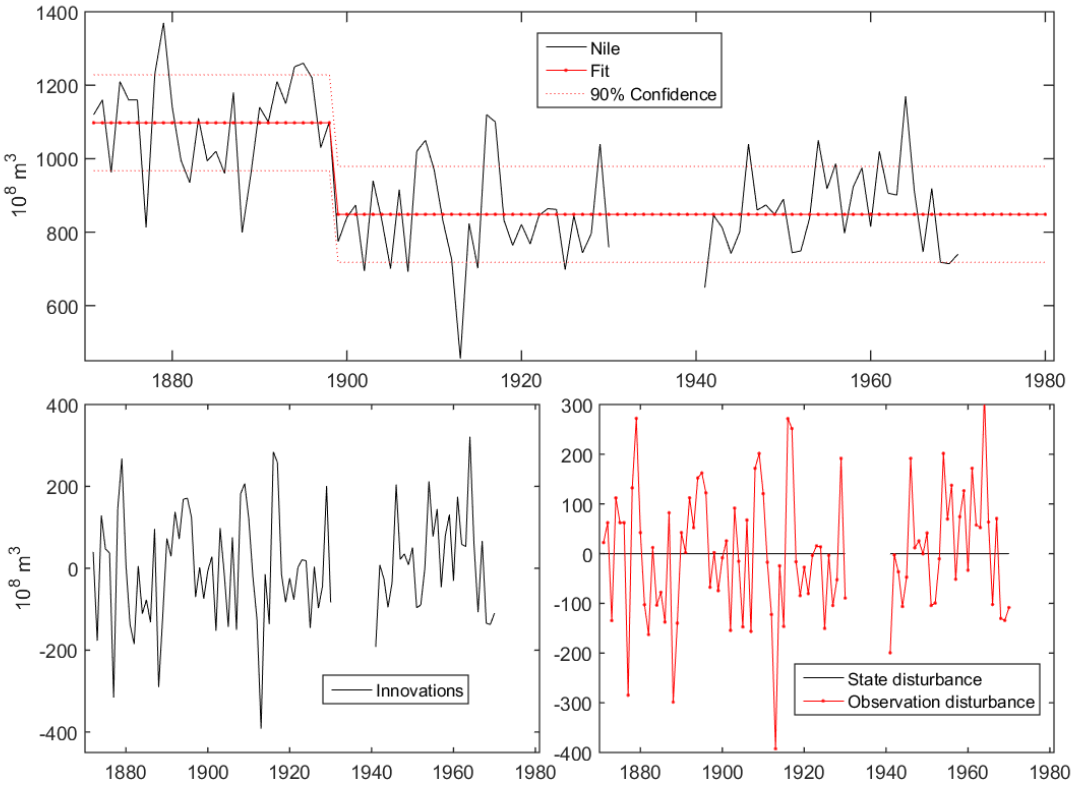

Figure 2: Fit, innovations and disturbances of `example2`.

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.

3.1.2 Case 2

A different way more formal from a statistical point of view to do the same is by defining one dummy variables 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(nile, 'u', u, 'model', @example3, 'OBJ', @llikc);
% or
sys= SS(nile, 'u', u, 'model', @example3, 'OBJ', @llikc);
```

Here, the duplet `'u', u` tells **SSpace** which the input variable is.

3.1.3 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(nile, 'model', @example4, 'OBJ', @llikc, 'user_inputs', u);
% or
sys= SS(nile, 'model', @example4, 'OBJ', @llikc, 'user_inputs', u);
```

Though it does not make sense in this case, it is worth noting that the dependence to the input may be modeled by a non-linear function, e.g.,, just by defining `model.D= p(2)*(u(2, :).^ (p(3)))` or any other specification. This is the main advantage of specifying models by writing a function. More examples may be checked in `demo5`.

3.2 Univariate models

In this second example the airpassenger data taken from [6] is analysed with a number of different univariate possibilities.

3.2.1 Case 1: Basic Structural Model

Lets try a BSM 'a la' Harvey [7] by using the template `SampleBSM`(it is also possible to use `SampleSS` and write 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. Check `demo_airpasbsm` that is part of the cases implemented in `demo2`.

The part of the template related to the trends, conveniently filled in, for the data is:

```
m= 1;
I= eye(m); O= zeros(m);
TT = [I I;O I];
ZT = [I O];
RT = [I O; O I];
QT = [10^p(1) 0; 0 10.^p(2)];
```

where the variable `m` is the number of output variables (1 for univariate) and matrices `I` and `O` are defined as a unity matrix and a block of zeros, respectively. The model is specified directly by the SS form of the Local Level Trend type, i.e.,

$$\begin{pmatrix} \alpha_{1,t+1} \\ \alpha_{2,t+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha_{1,t} \\ \alpha_{2,t} \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \eta_{1,t} \\ \eta_{2,t} \end{pmatrix} \tag{7}$$

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= [0; I]; QT= 10.^ p(1);` [8].

The periodic components (either seasonal or cyclical) in `SampleBSM` using the trigonometric seasonality are specified by (dummy seasonality is simpler and also possible).

```
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, though one values are 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)')`, etc.

The rest of the template is obvious:

```
H = 10.^p(4);
D= [];
```

where `H` is the variance of the observed noise and matrix `D` is included to deal with input variables.

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

```
sys= SSmodel(y, 'model', @airpasbsm, 'p0', -3);
sys= SSestim(sys);
sys= SSsmooth(sys);
% or
sys= SS(y, 'model', @airpasbsm, 'p0', -3);

T= sys.a(1, :)';
S= sum(sys.a(3:2:end, :))';
I= y-T-S;
```

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

3.2.2 Case 2: ARIMA

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

```
Sigma = exp(2*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`, `ARpoly` and `MApoly` are the difference, AR and MA polynomials, repectively. 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 $\theta_1 =$ `p(1)` and $\Theta_1 =$ `p(2)`. Constraints may be easily imposed, for example by choosing $\Theta_1 = 1/\theta_1 = 1/$ `p(1)`.

3.2.3 Case 3: Exponential Smoothing

A final illustration for this data is an Exponential Smoothing model [9], which template is `SampleES`. Check `demo_airpasES` used in `demo3`. The template filled in for this example is:

```
ModelType= 'AA12';
SigmaLevel= varmatrix(p(1));
SigmaSlope= varmatrix(p(2));
Damping= 1;
SigmaSeasonal= varmatrix(p(3));
SigmaObserved= varmatrix(p(4));
D= [];
```

The argument `ModelType` deals with the type of model, the first letter stands for the type of trend ('N' for none, 'A' for additive and 'D' for damped), the second letter is the type of seasonal ('N' for none and 'A' for additive), and the numbers after those letters is the period of the seasonal component. The rest of arguments are the variances for all the components plus the damping factor for the trends. Input variables may be included by introducing a `D` matrix as a function of the `p` vector of parameters.

3.3 Multivariate Dynamic Harmonic Regression (DHR)

The models in the previous examples may be extended to multivariate versions. A DHR model is an unobserved components model similar to the BSM, but with the cosine and sine terms appearing explicitly in the observation equations, it is effectively a Fourier analysis with time varying parameters (see [8] for univariate DHR models). These type of models may be easily implemented with the help of the template `SampleDHR` or done from scratch with the `SampleSS` template. In essence the template is very similar to the BSM of the previous example, but the user function needs an additional input argument that have to be taken into account in the `SSmodel` call, i.e., the number of time samples. In the case of a trivariate model of quarterly data is implemented in `demo_energydhr` used in `demo4`, a compact version of such template is

```
function model= multiDHR(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. It is important to note here the use of the `varmatrix` function, used to convert any set of arbitrary parameters into a semi positive definite covariance matrix. Beware that, since covariance matrices are symmetrical, in a trivariate model have 3×3 elements, but only the lower triangle

are different elements (6 values). This function allows also to impose rank and other constraints to build homogeneus models (see Chapter 4). It is also easy to check that there are only two harmonics and both are estimated with the same covariance matrices, check `Qs` matrix.

Such model may be run on the energy data of [7], with the following code:

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

It is important to initialise 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.

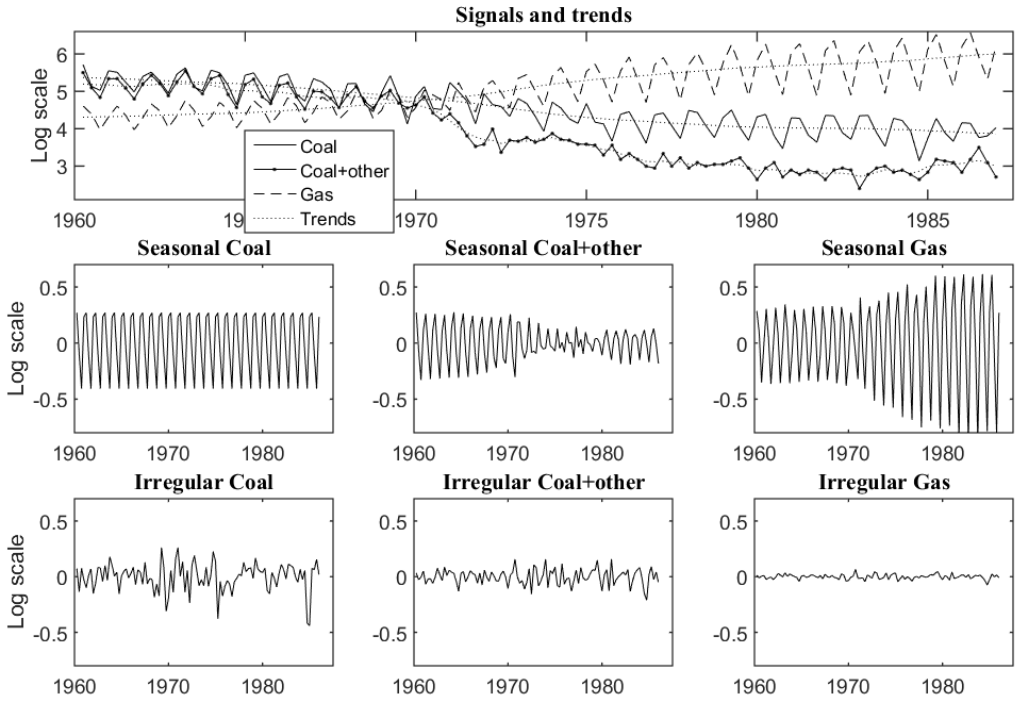

Figure 3: Unobserved components of `multiDHR`.

3.4 Non-Gaussian models

The number of van drivers casualties in the UK (see [2] and Figure 4) is a case where a Poisson distribution is justified, because numbers are small and the units are integers, neither the Gaussian assumption is strictly correct nor the logarithmic transformation produces 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 (8)).

$$\begin{aligned} \alpha_{t+1} &= T_t\alpha_t + \Gamma_t + R_t\eta_t \\ p(y_t \mid \theta_t) &= \exp\{\theta_t' y_t - \exp(\theta_t) - \log y_t!\} + bI_t \end{aligned} \tag{8}$$

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 (beware 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 = exp(p(1));
    % Dummy seasonal
    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 necessarily has to have two input arguments, while the user may add as many as the model require 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:

```
y(50:55)= nan;
y= [y; nan(20, 1)]; u= [u; ones(20, 1)];
sys= SSmodel(y, 'u', u, 'model', @demo_van_poisson);
sys= SSestim(sys);
sys= SSvalidate(sys);
sys= SSsmooth(sys);
% or just
sys= SS(y, 'u', u, 'model', @demo_van_poisson);
```

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

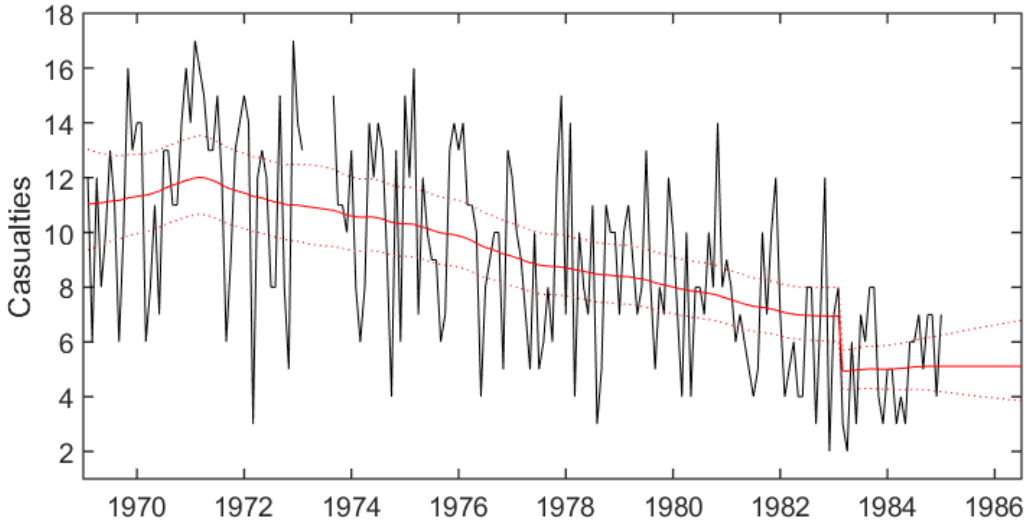

Figure 4: Data and trend of `demo_van_poisson`.

3.5 Non-linear models

This example illustrates the use of the Extended Kalman Filter with exact initialisation applied to the monthly visits abroad by UK residents from January 1980 to December 2006 following [10] and [2]. 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 heteroscedasticity problem, and, consequently a model with an interaction between the trend and the seasonal component is proposed instead with the rest of hypotesis aplying, see equation (9).

$$y_t = \text{Trend}_t + \text{Cycle}_t + \exp\{b\text{Trend}_t\}\text{Seasonal}_t + \epsilon_t \tag{9}$$

Comparing this equation with the general non-linear system in Chapter 1, we see that there is only one non-linear term, namely $T_t(\alpha_t)$ that is the equation (9) 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. For convenience a linear BSM model is implemented in a separate user 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 codings in this chapter. Firstly, an Integrated Random Walk or smooth trend is chosen depending 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 cycle is estimated as a constrained value between 18 and 800 months. Fourthly, the cycle is modulated by a damping factor estimated as a value between 0.5 and 1. Finally, separate variance values for the cycle and the seasonal components are estimated.

The non-linear 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) non-linear or state-dependent matrices in state equation, iii) non-linear or state-dependent matrices in observation equation. Only one definition for each matrix in any of the blocks are 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.

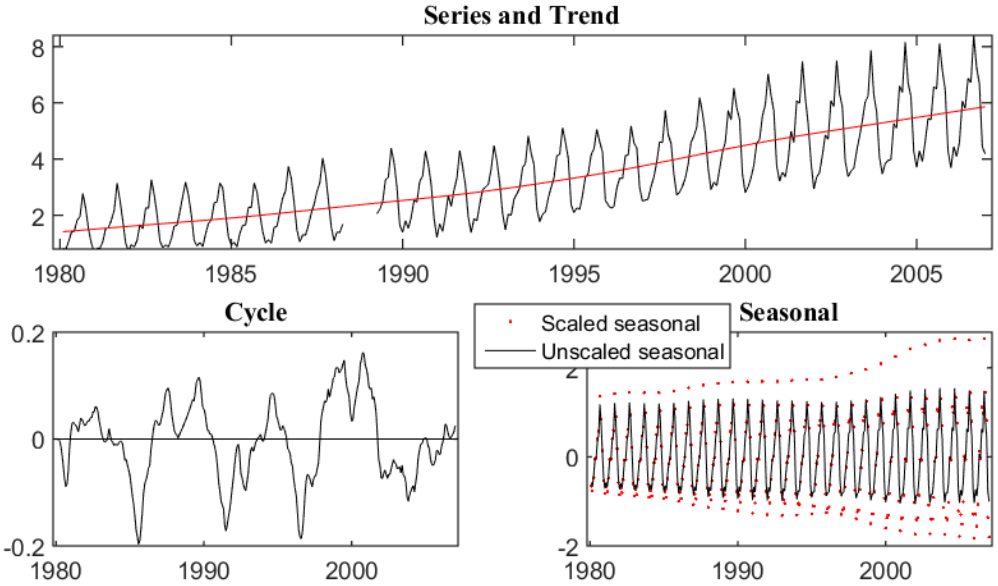

Figure 5: Series and trend, cycle and raw seasonal and scaled seasonal ($\exp(b\ \text{Trend}_t)\ \text{Seasonal}_t$) in `demo_uknle`.

3.6 Further issues

More templates and more demos are available in **SSpace**, but are left out of this manual by a matter of space. They may be checked by running carefuly the 8 demos included in the toolbox. Some of the most relevant are linear, non-linear and time varying regressions (see `SampleDLR` and `demo5`), concatenation of SS 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 `demo2`), time aggregation problems (see `SampleAGG` and `demo6`), and nesting in inputs models (see `SampleNEST` and `demo5`).

4 SSpace Reference

4.1 Core functions

The core functions are those which performs the usual operations in SS modelling. In order to be remembered or to be found easily their names always start by 'SS'. Most of the functions keep a very simple sintax (with the exception of `SSmodel` and `SS`) and in a normal time series analysis are used one after another in a fixed order. Such SS operations are setting up the model (function `SSmodel`), estimation (`SSestim`), filtering (`SSfilter`), state or disturbance smoothing (`SSsmooth` or `SSdisturb`) and validating (`SSvalidate`). Function `SS` runs all of them with one single call. Finally, `SSdemo` runs each one of the eight step-by-step demos that accompany the toolbox.

4.1.1 SSmodel

Usage

```
sys = SSmodel(y, 'property1', value1, 'property2', value2, ...)
```

Description

Creates **SSpace** model object. It may also add properties to an existing one that should be entered as the first input. The system created with `SSmodel` is the inputs to the rest of core functions, and is a `MATLAB` structure with the property names listed below. The inputs, marked with an asterisk (*) below, are set up by the user with `SSmodel`, the outputs are built later on with the rest of core functions. Only the leading characters of each property name, enough to identify uniquely the item, are necessary.

Input

Each of the `'property#'` is any of the following values. Mind that all these inputs are strings, except the first one that is compulsory and is the output data. Only the initial trailing characters enough to identify the label are necessary.

`y`:	(*) Matrix of output data. Put missing data at the end to produce forecasts.
`'u'`:	(*) Matrix of input data (if any).
`'model'`:	(*) Handle to model function.
`'user_inputs'`:	(*) Cell of addional parameters for model function.
`'p0'`:	(*) Vector of initial parameters for search.
`'p'`:	(*) Parameters vector (either estimated or fixed by the user).
`'a1'`:	(*) Initial state (exact if empty or `NaN`(all or part)).
`'P1'`:	(*) Initial diagonal of state covariance matrix.(`NaN` indicate calculation of a particular initial state by the toolbox and `Inf` indicate diffuse initialisation).
`'OBJ_FUNCTION_NAME'`:	(*) Handle to objective function.
`'gradient'`:	(*) Analytical gradient on/off for estimation.
`'Nsimul'`:	(*) Number of simulation for non-gaussian models.

Output

The output is a structure with the following fields (it also include the 'inputs' fields):

`table`	Table output after estimation.
`obj_value`	Optimal value of objective function.
`llik`	Log likelihood value.
`covp`	Covariance of parameter estimates.
`mat`	System matrices.
`a`	State estimates.
`P`	State (co)variance estimates.

`yfit`	Fitted values.
`F`	Fitted values (co)variance.
`yfor`	Forecasted values.
`Ffor`	Forecasted (co)variance.
`v`	Innovations.
`Fv`	Innovations (co)variance.
`eta`	State disturbance estimates.
`eps`	Observation disturbance estimates.
`Veta`	Covariance of state disturbances.
`Veps`	Covariance of observation disturbances.

The parameter `OBJ_FUNCTION_NAME` should contain one of these options:

`@llik`:	Log Likelihood (default).
`@llikc`:	Concentrated Log Likelihood.
{`@nsteps, j:N`}:	Sum of `j` to `N` steps ahead minimal error (`j > 1`).

4.1.2 SS

Usage

```
sys = SS(y, 'property1', value1, 'property2', value2, ...)
```

Description

Sets up a **SSpace** object and runs all relevant function to perform a full time series analysis. The syntax, inputs, etc. are the same as `SSmodel`.

Input

See inputs to `SSmodel`.

Output

See outputs to `SSmodel`.

4.1.3 SSestim

Usage

```
sys = SSestim(sys)
```

Description

Estimation of unkwnown parameters in SS systems. It fills in the fields `.p`, `.obj_value`, `.llik` and `.mat` of a **SSpace** system (see `SSmodel`).

Input

`sys`:	State space object created with SSmodel.

Output

`sys`:	Sys object with estimation output.

4.1.4 SSfilter

Usage

```
sys = SSfilter(sys)
```

Description

Optimal Kalman filtering estimation of states with their innovations. Produces the optimal filtered estimates of states, fitted values, innovations and their covariance matrices, i.e., $\hat{a}_{t|t}$ (field `a` of **SSpace** system, see `SSmodel`), $\hat{P}_{t|t}$ (`P`), $\hat{y}_{t|t}$ (`yfit`), $cov(\hat{y}_{t|t})$ (`F`), $\hat{v}_t$ (`v`) and $cov(\hat{v}_t)$ (`Fv`). Fields `yfor` and `Ffor` are filled in only when forecasts are required, indicated by `NaN` values at the end of the output data `y`.

Input

`sys:`	State space object created with SSmodel.

Output

`sys:`	Sys object with estimation output.

4.1.5 SSvalidate

Usage

```
sys = SSvalidate(sys)
```

Description

Validation of estimated SS system. Prints out a table with diagnosis information. It produces the same output than `SSfilter`, and also the table output in `table`.

Input

`sys`:	State space object created with SSmodel.

Output

`sys`:	Sys object with estimation output.

4.1.6 SSsmooth

Usage

```
sys = SSsmooth(sys)
```

Description

Optimal state smoothing estimation of states with their covariance and innovations. Produces the optimal smoothed estimates of states, fitted values, innovations and teir covariance matrices, i.e., $\hat{a}_{t|N}$ (field `a` of **SSpace** system, see `SSmodel`), $\hat{P}_{t|N}$ (`P`), $\hat{y}_{t|N}$ (`yfit`), $cov(\hat{y}_{t|N})$ (`F`), $\hat{v}_t$ (`v`) and $cov(\hat{v}_t)$ (`Fv`). Fields `yfor` and `Ffor` are filled in only when forecasts are required, indicated by missing values at the end of the output data `y`.

Input

`sys:`	State space object created with SSmodel.

Output

`sys:`	Sys object with estimation output.

4.1.7 SSdisturb

Usage

```
sys = SSdisturb(sys)
```

Description

Optimal disturbance smoother estimation with their covariance and innovations. It also runs the smoothing algorithm, then the output is the same as `SSestim`, with the addition of disturbance errors and their covariance matrices, i.e., $\hat{\eta}_{t|N}$ (field `eta`), $cov(\hat{\eta}_{t|N})$ (field `Veta`), $\hat{\epsilon}_{t|N}$ (`eps`) and $cov(\hat{\epsilon}_{t|N})$ (field `Veps`).

Input

`sys:`	State space object created with SSmodel.

Output

`sys:`	Sys object with estimation output.

4.1.8 SSdemo

Usage

```
SSdemo(number)
```

Description

SSpace demos. The available demos are the following:

1. Overview and tutorial.
2. Univariate Unobserved Components models.
3. ARIMA and Exponential Smoothing models.
4. Multivariate Unobserved Components.
5. Regression.
6. Time aggregation.
7. Non-Gaussian models .
8. Non-linear application.

Input

`number`:	a number between 1 and 8 to select the demo number.

4.2 Auxiliary functions

The auxiliary functions are general purpose functions aiming at facilitating the use of **SSpace**. Most of them are used at the specification of the model, in the function by which the user defines de model (see `SSmodel`, field `model`), but some are useful either at the beginning or at the end of the modelling procedure, as additional validation functions.

4.2.1 confband

Usage

```
bands = confband(fit, P, const)
```

Description

The inputs to this function are usually the output of functions `SSfilter`, `SSestim` or `SSdisturb`, stored on the fields of a **SSpace** system created previously with `SSmodel`.

Input

`fit`:	filtered or smoothed states, fitted values, forecasts, etc. (`a`, `yfit`, `yfor`, etc., see `SSmodel`).
`P`:	Covariance matrix of fit (`P`, `F`, `Ffor`, etc., see `SSmodel`).
`const`:	Constant for confidence band (2 for 95% confidence).

Output

`bands`:	Upper and lower confidence bands, i.e., `fit`$\pm$ const* $\sqrt{(P_t)}$.

4.2.2 constrain

Usage

```
P = constrain(p, limits, unconstrain)
```

Description

Transform any set of unbound values `p` in constrained `P` values between given limits. It also reverses the use, i.e., it calculates the unbound values to which bound values correspond to.

Input

`p`:	Vector of n unconstrained values.
`limits`:	Matrix 1×2 or $n \times 2$ of minimum and maximum limits.
`unconstrain`:	If 1 reverse the way this function works, i.e., `p` would be a set of parameters within the limits.

Output

`P`:	Vector of values between given limits. With `unconstrain = 1P` is the unconstrain parameters corresponding to `p`.

Examples

`constrain(0, [0 100])` is `50`.
`constrain(50, [0 100], 1)` is `0`.
`constrain(-10, [0 100])` is `0.25`.
`constrain(10, [0 100])` is `99.75`.
`constrain([0; 0], [0 20; -1 10])` is `[10 4.5]'`.

4.2.3 normalize

Usage

```
z = normalize(y, P)
```

Description

Normalize a vector of variables `y` with time varying covariance matrix `P`, it is assumed that variables has zero mean.

Input

`y`:	Output data.
`P`:	Three-dimensional matrix of covariance matrices, typically the output of `SSfilter` or any other core function.

Output

`z`:	Normalized variables, variables y transformed with identity covariance matrix.

4.2.4 varmatrix

Usage

```
P = varmatrix(p, m, r)
```

Description

Builds a semidefinite positive covariance matrix from a vector of arbitrary values. Dimension of input `p` should be the distinct elements in a simmetrical covariance matrix.

This function may be used in different ways:

- With just one input: i) if `p` is a column vector `P` will be a semidefinite positive matrix, ii) if `p` is a row vector `P` is a row of variances taking any element in `p` as a scalar variance.
- With two inputs, `m` should be a squared matrix of the same dimension than the desired covariance matrix with zeros and ones. Zero positions imply zero covariances in the output `P`.
- With three inputs `m` is the dimension of the desired covariance matrix and `r` is the rank of such matrix.

Input

`p`:	Vector of any values ($np \times 1$) or semidefinite positive matrix ($m \times m$).
`m`:	Dimension of desired covariance matrix or zero and ones matrix containing zero constraints.
`r`:	Desired rank of output covariance matrix.

Output

`P`:	Semidefinite positive matrix of dimension m with rank r when argument `p` is a vector, or vector of parameters when input `p` is a covariance matrix.

Examples

```
varmatrix((1:10)'/100).
varmatrix((1:5)'/100, 3, 2).
varmatrix((1:5)'/100, [1 1 0; 1 1 1; 0 1 1]).
```

4.2.5 dhrMatrix

Usage

```
X = dhrMatrix(n, P, harmonics)
```

Description

Builds matrix of time varying periods to use in `SampleDHRt`

Input

`n`:	Number of observations to generate.
`P`:	Time varying vector of periods 1 (fundamental period).
`harmonics`:	Vector of harmonics required.

Output

`X`:	Regressors for DHR model with time varying periods.

4.2.6 vdiff

Usage

```
z = vdiff(y, order, s)
```

Description

Differenciation of a vector of variables.

Given some time series $Y = \{y_1, y_2, ..., y_T\}$, the 'B' is the backshift operator such that $B^i y_t = y_{t-i}$.

The `vdiff` function computes the difference series in a wide variety of forms:

`z=vdiff(Y, 1)` $\rightarrow (1 - B)Y$
`z=vdiff(Y, 2)` $\rightarrow (1 - B)^2 Y$
`z=vdiff(Y, [1 2], [1 12])` $\rightarrow (1 - B)(1 - B^{12})^2 Y$

In the case of a multivariable series ($m \times N$ matrix), `zdif` accepts different parameters for each individual variable. Let P and Q be two univariate time series of length N.

```
z=vdiff([P; Q], [1 2; 1 3], [1 12; 1 6])
```
$\rightarrow [(1 - B)(1 - B^{12})^2 p; (1 - B)(1 - B^6)^3 Q]$

```
z=vdiff([P; Q], [1 2], [1 12])
```
$\rightarrow [(1 - B)(1 - B^{12})^2 p; (1 - B)(1 - B^{12})^2 Q]$

Input

`y`:	Vector time series data.
`order`:	Difference orders for multiplicative polynomials ($1 \times any$, default is 1).
`s`:	Seasonal parameter for each polynomial ($1 x any$, default is 1).

Output

`z`:	Differenced data.

4.2.7 vconv

Usage

```
C = vconv(A, B, m)
```

Description

Convolution of vector polynomials.

Input

`A`:	Vector polynomial of block coefficients m x cA (*) (m*cA x cA).
`B`:	Vector polynomial of block coefficients cA x cB (*) (cA*cB x cB).
`m`:	Rows of each block of coefficients in A.

Output

`C`:	Vector polynomial result of the multiplication.

Examples

```
vconv([1 0 -0.8 0.3 -0.6 0.1; 0 1 0.2 0.4 0 0.3]', [1 0
0.3 0.2;0 1 -0.3 -0.2]')
```

4.2.8 vfilter

Usage

```
z = vfilter(B, A, u)
```

Description

Filter vector of time series with vector polynomials

Input

A:	Vector polynomial of block coefficients m x cA (*) (m*cA x cA).
B:	Vector polynomial of block coefficients cA x cB (*) (cA*cB x cB).
u:	Vector time series data (m x T or T x m).

Output

z:	Filtered data (m x ¡T)

Examples

```
z= randn(500, 2);   vfilter([1 0 -0.8 0.3 -0.6 0.1; 0 1 0.2
0.4 0 0.3]', [1 0 0.3 0.2;0 1 -0.3 -0.2]', z)
```

4.2.9 vroots

Usage

```
r = vroots(A)
```

Description

Roots of vector polynomial

Input

`A`: Vector polynomial of block coefficients m x cA (*) (m*cA x cA)

Output

`r`: Roots of polynomial.

Examples

```
vroots([1 0 -0.8 0.3 -0.6 0.1; 0 1 0.2 0.4 0 0.3])
```

4.3 Optimizer

This is not a function, but a simple script, intended for advanced users. By editing the script the user may change two basic inputs:

1. The way the optimization routine works, by changing any of the options at the top by the standard `MATLAB` function `optimset`. One typical and recommended change would be to switch the `'Display'` option so that the optimization routine shows intermediate or final reports on the estimation. That would be essential to know whether the optimization converged to a local optimum, etc. Other typical changes would be the tolerances in the gradient or the function reduction, in case the user detects that the optimization terminates too soon or it spends too much time without real improvement. Any other options may be changed, see the help of `optimset`.

2. The optimization routine itself. This means using any other `MATLAB` routine to do the optimization, instead of `fminunc`, that is the default option. Independently of the routine used, the inputs to that function should be provided exactly at is written for the `fminunc` function in `optimset`, because such is the order in which the routines to evaluate the cost functions are built. The optimization routine should also return a column vector `p`, a scalar `FUN_OPTIMUM` and a vector `GRAD` containing the optimized values for the parameters, the value of the cost function at the optimum and vector gradient, respectively.

4.4 Template functions

The list of template functions below are intended to set up models in SS form. All of them start with the letters 'Sample' in order to be remembered or searched for easily. The models created with these templates enter into a **SSpace** system created with `SSmodel` function through the field `model`, that should contain a handle to it.

There are a number of rules that apply to all of them, some others are template-specific:

1. Nothing should be removed from the template, but anything may be added to it in order to specify the system matrices. In particular, additional inputs may be added to the templates, that should be supplied to the system in `SSmodel` with the duplet `user_inputs` and a cell containing such inputs.

2. The templates come with different inputs, being the first one compulsory. Its name is `p` and represents a vector of unknown parameters to be estimated. This name may be changed and used accordingly.

3. All matrices have to be of appropriate dimensions. The user should check them carefuly, but **SSpace** peforms exhaustive checks to make sure of it and issues specific messages in case of errors.

4. Time varying matrices may be defined using three dimensional matrices, being time the third dimension.

5. There are two exception to the previous rule, the D_t and Γ_t matrices. They manage the output-input relations but may be handled independently, for example, one of them may be full and the other empty. Assuming m is the number of output variables, k the number of inputs, ns the number of states, and n the time lenght, D_t would be correctly set if it is any matrix of dimension $m \times 1$, $m \times n$, $m \times k$ or $m \times k \times n$. The difference of the two first options is that no input to the system has to be passed to the `SSmodel` function, while it is compulsory in two latter cases. Correct dimensions for Γ_t are $ns \times 1$, $ns \times n$, $ns \times k$ or $ns \times k \times n$.

6. Each template has its own idiosincracy and should be checked carefully before using it.

4.4.1 SampleSS

Usage

```
model = SampleSS(p)
```

Description

Template for general linear SS systems.

Model definition:

$$\begin{array}{cc} \alpha_{t+1} = T_t\alpha_t + \Gamma_t + R_t\eta_t, & \eta_t \sim N(0, Q_t) \\ y_t = Z_t\alpha_t + D_t + C_t\epsilon_t, & \epsilon_t \sim N(0, H_t) \\ \alpha_1 \sim N(a_1, P_1) & t = 1, 2, \ldots, N \end{array} \tag{10}$$

Variables to change in this template:

`model.T`:	T_t.
`model.Gam`:	Γ_t.
`model.R`:	R_t.
`model.Z`:	Z_t.
`model.D`:	D_t.
`model.C`:	C_t.
`model.Q`:	Q_t.
`model.H`:	H_t.
`model.S`:	S_t.

4.4.2 SampleARIMA

Usage

```
model = SampleARIMA(p)
```

Description

Template for univariate ARIMA(p, d, q) models with inputs

Model definition:

$$\Delta_d(B)y_t = f(u_t) + \Theta_q(B)/\Phi_p(B)a_t \tag{11}$$

where:

B is the lag operator.
y_t if the output variable.
u_t is a set of k input variables.
a_t is white noise.
$\Delta_d(B) = (1 - B)^d$.
$\Phi_p(B) = (1 + \phi_1 B + \ldots \phi_p B^p)$.
$\Theta_q(B) = (1 + \theta_1 B + \ldots \theta_q B^q)$.
$f(u_t)$ is the relation of inputs to outputs.

The function $f(\bullet)$ may be linear, non linear regression, time varying regression, multiple transfer function, etc. Some of the options should be implemented with the help of the `catsystem` function.

Any of the polynomials may be defined as a product of several other polynomials with the help of the `conv` function, in this way, it is straightforward to build $ARIMA(p, d, q) \times (P, D, Q)_s$ models.

Variables to change in this template:

`Sigma`:	variance of a_t, always positive (`varmatrix` may be used).
`DIFFpoly`:	column vector of coefficients in $\Delta_d(B)$ polynomial. Use `conv` function to produce an arbitrary number of differences.
`ARpoly`:	column vector of AR coefficients in $\Phi_p(B)$ polynomial.
`MApoly`:	column vector of MA coefficients in $\Theta_q(B)$ polynomial.
`D`:	`D` matrix in the general SS system.

4.4.3 SampleBSM

Usage

```
model = SampleBSM(p)
```

Description

Template for Multivariate Basic Structural Model with dummy or trigonommetric seasonality 'a la' [7].

Model definition:

$$y_t = T_t + C_t + S_t + f(u_t) + I_t \tag{12}$$

y_t:	vector of m output variables.
u_t:	vector of k input variables.
T_t:	trend components.
C_t:	cyclical components (either trigonommetric or dummy).
S_t:	seasonal components (either trigonommetric or dummy).
$f(u_t)$:	relation of inputs to outputs. It may be linear, non linear regression, time varying regression, multiple transfer function, etc. Some of the options should be implemented with the help of the `catsystem` function.
I_t:	irregular components.

Variables to change in this template:

Regarding trend:

`m`:	number of outputs.
`QT`:	covariance matrix of trend noises (use `varmatrix` to ensure positive semidefiniteness).

Regarding trigonommetric seasonality or cycles:

`Periods`:	$m \times Nhar$ matrix containing the periods of the cyclical and seasonal components, where $Nhar$ is the number of harmonics. For multivariate systems, a `NaN` indicates that such periodic component is not present in a time series.
`Rho`:	damping factors affecting each periodic component (same dimensions as `Periods`).
`Qs`:	covariance matrix of η disturbances for the harmonics. `Qs= [Q(1) Q(2) ... Q(Nhar)]`, where each `Q(i)` is a squared covariance matrix (use `varmatrix` to set it up).

Regarding dummy seasonality:

`Periods`: fundamental period of seasonality, i.e., the number of samples per year.
`Qs`: covariance matrix of η_t disturbances.

Regarding irregular component:

`H`: Covariance matrix of irregular component.

Regarding inputs relations:

`D`: `D` matrix in the general SS system.

4.4.4 SampleDHR

Usage

```
model = SampleDHR(p)
```

Description

Template for Multivariate Dynamic Harmonic Regression 'a la' [8].

Model definition

$$y_t = T_t + C_t + S_t + f(u_t) + I_t \tag{13}$$

Everything is equal to `SampleBSM`, with the only exception that this function needs an extra input that should be passed on to the system when building the model with `SSmodel` in field `user_inputs`. Such input is `N` that is the time length of the data.

4.4.5 SampleDHRt

Usage

```
model = SampleDHR(p, dhrX)
```

Description

Template for Dynamic Harmonic Regression with time varying periods.

Model definition

$$y_t = T_t + C_t + S_t + f(u_t) + I_t \tag{14}$$

Everything is equal to `SampleDHR`, with the only exception that this function needs an extra input that should be passed on to the system when building the model with `SSmodel` in field `user_inputs`. Such input is `dhrX` that is the time varying periods and harmonics, for which `dhrMatrix` may be used.

4.4.6 SampleSH

Usage

```
model = SampleSH(p, N)
```

Description

Same as `SampleDHR` but with a dummy seasonal model with different variances for each dummy variable 'a la' [11].

Model definition

$$y_t = T_t + C_t + S_t + f(u_t) + I_t \tag{15}$$

Everything is equal to `SampleBSM`, with the only exception that this function needs an extra input that should be passed on to the system when building the model with `SSmodel` in field `user_inputs`. Such input is `N` that is the time length of the data.

4.4.7 SampleDLR

Usage

```
model = SampleDLR(p)
```

Description

Template for Dynamic Linear Regression modelling (also SURE systems).

Model definition:

$$y_t = A_t u_t + a_t \tag{16}$$

y_t: set of m output variables.
u_t: set of k input variables.
A_t: block of $m \times k$ time varying parameters relating inputs and outputs.
a_t: m white noises with full covariance matrix.

Variables to change in this template:

`D`: matrix $m \times k$. Ones in this matrix indicates where a parameter should be placed, i.e., which input (column in `D` or A_t) affect which output (row in `D` or A_t). E.g., if `D(3, 2)= 1` one time varying parameter is being allocated between input 2 and output 3. Parameters will be the states in the system and are allocated by rows, i.e., first states are those affecting the first output, etc.

`Q`: diagonal variance matrix for the $k \times m$ parameters. Use zeros for constant parameters.

`h`: switch between Random Walk type of parameters (`h=1`) or Integrated Random Walk (`h=2`).

`H`: covariance matrix of irregular component (observed noise).

4.4.8 SampleES

Usage

```
model = SampleES(p)
```

Description

Template for univariate Exponential Smoothing models 'a la' [9].

Model definition:

(1) Damped Trend model with seasonal and inputs:

$$\begin{array}{rllll} y_t = & l_{t-1} + & \phi b_{t-1} + & s_{t-m} + & f(u_t) \quad + a_t \\ l_t = & l_{t-1} + & \phi b_{t-1} & & \quad + \alpha a_t \\ b_t = & & \phi b_{t-1} & & \quad + \beta a_t \\ s_t = & s_{t-m} & & & \quad + \alpha_s a_t \end{array} \tag{17}$$

Standard constraints apply (other are possible):

$$\begin{array}{rrl} 0 < & \alpha & < 2 \\ 0 < & \beta & < 4 - 2\alpha \\ 0 <= & \phi & < 1 \\ 0 < & \alpha_s & < 2 \end{array} \tag{18}$$

(2) Damped Local level model with seasonal and inputs:

$$\begin{array}{lll} y_t = \phi l_{t-1} & + s_{t-m} + f(u_t) & + a_t \\ l_t = \phi l_{t-1} & + & + \alpha a_t \\ s_t = s_{t-m} & & + \alpha_s a_t \end{array} \tag{19}$$

Standard constraints apply (other are possible):

$$\begin{array}{rrl} \phi - 1 < & \alpha & < \phi + 1 \\ -1 < & \phi & < 1 \\ 0 < & \alpha_s & < 2 \end{array} \tag{20}$$

y_t:	output variable.
u_t:	set of k input variables.
l_t:	level for y_t variable.
b_t:	slope for y_t variable.
s_t:	seasonal for y_t variable.
$f(u_t)$:	relation of inputs to outputs. It may be linear, non linear regression, time varying regression, multiple transfer function, etc. Some of the options should be implemented with the help of the `catsystem` function.
a_t:	white noise or irregular.
α:	discounting factor for level.
β:	discounting factor for slope.
α_s:	discounting factor for seasonal.

Variables to change in this template:

`ModelType`:	type of model required, from this eight types: `'ANN'`: Local level model, no seasonal. `'AAN'`: Local trend, no seasonal. `'DNN'`: Damped level, no seasonal. `'DAN'`: Damped trend, no seasonal. `'ANA#'`: Local level trend, additive seasonal of period #. `'AAA#'`: Local trend, additive seasonal of period #. `'DNA#'`: Damped level, additive seasonal of period #. `'DAA#'`: Damped trend, additive seasonal of period #.
`Phi`:	damping factor for damped level or damped trend models. For multivariate models `Phi` is normally diagonal. Empty for no damping factors. Use `constrain` to restric values between 0 and 1.
`Alpha`:	α.
`Beta`:	β. Empty for local level.
`AlphaS`:	α_s. Empty if no sesonal component is present.
`D`:	`D` matrix in the general SS system.
`Sigma`:	noise variance.

4.4.9 SampleNONGAUSS

Usage

```
model = SampleNONGAUSS(p, H)
```

Description

Template for models with non-gaussian observed noise 'a la' [2].

Model description:

$$\begin{aligned} \alpha_{t+1} &= T_t\alpha_t + \Gamma_t + R_t\eta_t, \quad \eta_t \sim N(0, Q_t) \\ y_t &= \theta_t + \epsilon_t, \epsilon_t \sim p(\epsilon_t), \\ \theta_t &= Z_t\alpha_t \qquad\qquad t = 1, 2, \ldots, N \end{aligned} \tag{21}$$

with $p(\bullet)$ being any distribution function from a list below. Beware that the `H` input to this function, i.e., the observation equation noise variance, is compulsory and should be passed on to the `SSsystem` in line 15 of the template. This model requires a linear model that is previously defined with an additional template. Actually it is run in line 15, i.e., `model= SSsystem(p, H, ...)`.

Variables to change in this template:

`model.dist`:	name from the following list: Binary Binomial(n) Exponential NegativeBinomial Poisson Tdist (T distribution)
`model.param`:	cell containing all the inputs necessary to run the chosen distribution function.
`Sigma2`:	variance of observed noise (use `varmatrix` to make it positive).

4.4.10 SampleEXP

Usage

```
model = SampleEXP(p, H)
```

Description

Template for models with observation equation from the exponential family of distribution 'a la' [2].

Model description:

$$\begin{array}{rll} \alpha_{t+1} = & T_t\alpha_t + \Gamma_t + R_t\eta_t, & \eta_t \sim N(0, Q_t) \\ p(y_t \mid \theta_t) = & exp[y_t'\theta_t - b_t(\theta_t) + c_t(y_t)], & -\infty < \theta_t < \infty \\ \theta_t = & Z_t\alpha_t & t = 1, 2, \ldots, N \end{array} \tag{22}$$

Beware that the `H` input to this function, i.e., the observation equation noise variance, is compulsory and should be passed on to the `SSsystem` in line 16 of the template. This model requires a linear model that is previously defined with an additional template. Actually it is run in line 16, i.e., `model= SSsystem(p, H, ...)`.

Variables to change in this template:

`model.dist`: any member of the exponential distribution functions from this list:
Binary
Binomial(n)
Exponential
NegativeBinomial
Poisson

4.4.11 SampleSV

Usage

```
model = SampleSV(p, H)
```

Description

Template for Stochastic Volatility models 'a la' [2].

Model description:

$$\begin{aligned} \alpha_{t+1} &= \phi\alpha_t + \Gamma_t + R_t\eta_t, \quad \eta_t \sim N(0, \sigma_\eta^2) \\ y_t &= exp(\tfrac{1}{2}\theta_t)\epsilon_t + D_t, \\ \theta_t &= \alpha_t \qquad\qquad t = 1, 2, \ldots, N \end{aligned} \tag{23}$$

Beware that the `H` input to this function, i.e., the observation equation noise variance, is compulsory.

Variables to change in this template:

`Sigma2`:	variance of observed noise (ϵ_t).
`AR1`:	AR(1) parameter (ϕ).
`Q`:	variance of AR(1) process noise (η_t).
`D`:	`D` matrix in standard SS system.

4.4.12 SampleNL

Usage

```
model = SampleNL(p, at, ctrl)
```

Description

Template for non-linear SS models.

Model description:

$$\begin{aligned} \alpha_{t+1} &= T_t(\alpha_t) + \Gamma_t + R_t(\alpha_t)\eta_t, \quad \eta_t \sim N(0, Q_t(\alpha_t)) \\ y_t &= Z_t(\alpha_t) + D_t + C_t(\alpha_t)\epsilon_t, \quad \epsilon_t \sim N(0, H_t(\alpha_t)) \\ & \qquad\qquad\qquad\qquad t = 1, 2, \ldots, N \end{aligned} \tag{24}$$

Beware that this template has two additional inpus that are necessary for the correct execution of the program. Some or even most of the system matrices may come from a standard linear models. Then this template may rely on a linear template, whose call may be introduced at the very beginning of the template.

Variables to change in this template:

Linear or state independent matrices are defined as usual, i.e.,:

`model.T`:	T_t.
`model.Gam`:	Γ_t.
`model.R`:	R_t.
`model.Z`:	Z_t.
`model.D`:	D_t.
`model.C`:	C_t.
`model.Q`:	Q_t.
`model.H`:	H_t.

However, for non-linear or state-dependent matrices (state at `t` is `at` variable in `MATLAB`):

`model.dTa`:	$\frac{\partial T(a_t)}{\partial a_t}$.
`model.Ta`:	$T(a_t)$.
`model.Ra`:	$R(a_t)$.
`model.Qa`:	$Q(a_t)$.
`model.dZa`:	$\frac{\partial Z(a_t)}{\partial a_t}$.
`model.Za`:	$Z(a_t)$.
`model.Ca`:	$C(a_t)$.
`model.Ha`:	$H(a_t)$.

4.4.13 SampleAGG

Usage

```
model = SampleAGG
```

Description

Template for models with temporal aggregation (flow variables). This template only needs to include a call to a linear time invariant SS system.

4.4.14 SampleCAT

Usage

```
model = SampleCAT(p)
```

Description

Template for models with temporal aggregation (flow variables). This template only needs to include several calls to all the systems that wants to be concatenated.

4.4.15 SampleNEST

Usage

```
model = SampleNEST(p)
```

Description

Template to help writing general models by nesting input and output SS systems. This template only needs to include a call to the input system and to the output system. Both are supposed to exist beforehand, probably created with other templates.

References

[1] The MathWorks Inc. *MATLAB - The Language of Technical Computing, Version R2015b*. Natick, Massachusetts. URL http://www.mathworks.com/products/matlab/, 2015.

[2] J Durbin and Siem Jan Koopman. *Time series analysis by state space methods*, volume 38. Oxford University Press, 2012.

[3] C James Taylor, Diego J Pedregal, Peter C Young, and Wlodek Tych. Environmental time series analysis and forecasting with the captain toolbox. *Environmental Modelling & Software*, 22(6):797–814, 2007.

[4] M.A. Villegas and D.J. Pedregal. Sspace: A toolbox for state space modeling. *Journal of Statistical Software*, 87:1–26, 2018.

[5] D.J. Pedregal. State space modeling for practitioners. *Foresight*, 54:21–25, 2019.

[6] George EP Box, Gwilym M Jenkins, Gregory C Reinsel, and Greta M Ljung. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[7] Andrew C Harvey. *Forecasting, structural time series models and the Kalman filter*. Cambridge university press, 1989.

[8] Peter C Young, Diego J Pedregal, and Wlodek Tych. Dynamic harmonic regression. *Journal of forecasting*, 18(6):369–394, 1999.

[9] Rob Hyndman, Anne B Koehler, J Keith Ord, and Ralph D Snyder. *Forecasting with exponential smoothing: the state space approach*. Springer Science & Business Media, 2008.

[10] Siem Jan Koopman and Kai Ming Lee. Seasonality with trend and cycle interactions in unobserved components models. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 58(4):427–448, 2009.

[11] T. Proietti. Seasonal heteroscedasticity and trends. *Journal of forecasting*, 17:1–17, 1998.