State Space Models Manual

Jyh-Ying Peng[1], John A. D. Aston[2]

[1]Institute of Information Science, Academia Sinica*

[2] Department of Statistics, University of Warwick

July 1, 2011

*Address for Correspondence:
Institute of Information Science,
Academia Sinica
128 Academia Road, Sec 2
Taipei 115, Taiwan
email: jypeng@iis.sinica.edu.tw

Contents

1 Introduction

State Space Models (SSM) is a MATLAB toolbox for doing time series analysis using general state space models and the Kalman filter [1]. The goal of this software package is to provide users with an intuitive, convenient and efficient way to do general time series modeling within the state space framework. Specifically, it seeks to provide users with easy construction and combination of arbitrary models without having to explicitly define every component of the model, and to maximize transparency in their data analysis usage so no special consideration is needed for any individual model. This is achieved through the unification of all state space models and their extension to non-Gaussian and nonlinear special cases. The user creation of custom models is also implemented to be as general, flexible and efficient as possible. Thus, there are often multiple ways of defining a single model and choices as to the parametrization versus initialization and to how the model update functions are implemented. Stock model components are also provided to ease user extension to existing predefined models. Functions that implement standard algorithms such as the Kalman filter and state smoother, log likelihood calculation and parameter estimation will work seamlessly across all models, including any user defined custom models.

These features are implemented through object-oriented programming primitives provided by MATLAB and classes in the toolbox are defined to conform to MATLAB conventions whenever possible. The result is a highly integrated toolbox with support for general state space models and standard state space algorithms, complemented by the built-in matrix computation and graphic plotting capabilities of MATLAB.

2 How to Cite State Space Models Toolbox

Please cite the following paper if you use this software in your research:
Jyh-Ying Peng and John A.D. Aston. The State Space Models toolbox for MATLAB. *Journal of Statistical Software*, 41(6):1-26, May 2011. Software available at `http://sourceforge.net/projects/ssmodels`.

3 The State Space Model Equations

This section presents a summary of the basic definition of models supported by SSM. Currently SSM implements the Kalman filter and related algorithms for model and state estimation, hence non-Gaussian or nonlinear models need to be approximated by linear Gaussian models prior to or during estimation. However the approximation is done automatically and seamlessly by the respective routines, even for user-defined non-Gaussian or nonlinear models.

The following notation for various sequences will be used throughout the paper:

y_t	$p \times 1$	Observation sequence
ε_t	$p \times 1$	Observation disturbance (Unobserved)
α_t	$m \times 1$	State sequence (Unobserved)
η_t	$r \times 1$	State disturbance (Unobserved)

3.1 Linear Gaussian models

SSM supports linear Gaussian state space model in the form

$$\begin{aligned} y_t &= Z_t\alpha_t + \varepsilon_t, & \varepsilon_t &\sim \mathcal{N}(0, H_t), \\ \alpha_{t+1} &= c_t + T_t\alpha_t + R_t\eta_t, & \eta_t &\sim \mathcal{N}(0, Q_t), \\ \alpha_1 &\sim \mathcal{N}(a_1, P_1), & t &= 1, \ldots, n. \end{aligned} \tag{1}$$

Thus the matrices $Z_t, c_t, T_t, R_t, H_t, Q_t, a_1, P_1$ are required to define a linear Gaussian state space model. The matrix Z_t is the state to observation linear transformation, for univariate models it is a row vector. The matrix c_t is the same size as the state vector, and is the "constant" in the state update equation, although it can be dynamic or dependent on model parameters. The square matrix T_t defines the time evolution of states. The matrix R_t transforms general disturbance into state space, and exists to allow for more varieties of models. H_t and Q_t are Gaussian variance matrices governing the disturbances, and a_1 and P_1 are the initial conditions. The matrices and their dimensions are listed:

Z_t	$p \times m$	State to observation transform matrix
c_t	$m \times 1$	State update constant
T_t	$m \times m$	State update transform matrix
R_t	$m \times r$	State disturbance transform matrix
H_t	$p \times p$	Observation disturbance variance
Q_t	$r \times r$	State disturbance variance
a_1	$m \times 1$	Initial state mean
P_1	$m \times m$	Initial state variance

3.2 Non-Gaussian models

SSM supports non-Gaussian state space models in the form

$$\begin{array}{rcll} y_t & \sim & p(y_t|\theta_t), & \theta_t = Z_t\alpha_t, \\ \alpha_{t+1} & = & c_t + T_t\alpha_t + R_t\eta_t, & \eta_t \sim Q_t = p(\eta_t), \\ \alpha_1 & \sim & \mathcal{N}(a_1, P_1), & t = 1, \ldots, n. \end{array} \tag{2}$$

The sequence θ_t is the signal and Q_t is a non-Gaussian distribution (e.g. heavy-tailed distribution). The non-Gaussian observation disturbance can take two forms: an exponential family distribution

$$H_t = p(y_t|\theta_t) = \exp\left[y_t^T\theta_t - b_t(\theta_t) + c_t(y_t)\right], \ -\infty < \theta_t < \infty, \tag{3}$$

or a non-Gaussian additive noise

$$y_t = \theta_t + \varepsilon_t, \ \varepsilon_t \sim H_t = p(\varepsilon_t). \tag{4}$$

With model combination it is also possible for H_t and Q_t to be a combination of Gaussian distributions (represented by variance matrices) and various non-Gaussian distributions.

3.3 Nonlinear models

SSM supports nonlinear state space models in the form

$$\begin{array}{rcll} y_t & = & Z_t(\alpha_t) + \varepsilon_t, & \varepsilon_t \sim \mathcal{N}(0, H_t), \\ \alpha_{t+1} & = & c_t + T_t(\alpha_t) + R_t\eta_t, & \eta_t \sim \mathcal{N}(0, Q_t), \\ \alpha_1 & \sim & \mathcal{N}(a_1, P_1), & t = 1, \ldots, n. \end{array} \tag{5}$$

Z_t and T_t are functions that map $m \times 1$ vectors to $p \times 1$ and $m \times 1$ vectors respectively. With model combination it is also possible for Z_t and T_t to be a combination of linear functions (matrices) and nonlinear functions.

4 The State Space Algorithms

This section summarizes the core algorithms implemented in SSM, starting with the Kalman filter. Normally the initial conditions a_1 and P_1 should be given for these algorithms, but the exact diffuse initialization implemented permits

elements of either to be unknown, and derived implicitly from the first few observations. The unknown elements in a_1 are set to 0, and the corresponding diagonal entry in P_1 is set to ∞, this represents an improper prior which reflects the lack of knowledge about that particular element *a priori*. See [1] for details. To keep variables finite in the algorithms we define another initial condition $P_{\infty,1}$ with elements

$$(P_{\infty,1})_{i,j} = \begin{cases} 0, & \text{if } (P_1)_{i,j} < \infty, \\ 1, & \text{if } (P_1)_{i,j} = \infty, \end{cases} \tag{6}$$

and set infinite elements of P_1 to 0. All the state space algorithms supports exact diffuse initialization.

For non-Gaussian and nonlinear models linear Gaussian approximations are performed by linearization of the loglikelihood equation and model matrices respectively, and the Kalman filter algorithms can then be used on the resulting approximation models.

4.1 Kalman filter

Given the observation y_t, the model and initial conditions $a_1, P_1, P_{\infty,1}$, Kalman filter outputs $a_t = \mathrm{E}(\alpha_t|y_{1:t-1})$ and $P_t = \mathrm{Var}(\alpha_t|y_{1:t-1})$, the one step ahead state prediction and prediction variance. Theoretically the diffuse state variance $P_{\infty,t}$ will only be non-zero for the first couple of time points (usually the number of diffuse elements, but may be longer depending on the structure of Z_t), after which exact diffuse initialization is completed and normal Kalman filter is applied. Here n denotes the length of observation data, and d denotes the number of time points where the state variance remains diffuse.

The normal Kalman filter recursion (for $t > d$) are as follows:

$$\begin{aligned} v_t &= y_t - Z_t a_t, \\ F_t &= Z_t P_t Z_t^T + H_t, \\ K_t &= T_t P_t Z_t^T F_t^{-1}, \\ L_t &= T_t - K_t Z_t, \\ a_{t+1} &= c_t + T_t a_t + K_t v_t, \\ P_{t+1} &= T_t P_t L_t^T + R_t Q_t R_t^T. \end{aligned}$$

The exact diffuse initialized Kalman filter recursion (for $t \leq d$) is divided into two cases, when $F_{\infty,t} = Z_t P_{\infty,t} Z_t^T$ is nonsingular:

$$\begin{aligned} v_t &= y_t - Z_t a_t, \\ F_t^{(1)} &= (Z_t P_{\infty,t} Z_t^T)^{-1}, \\ F_t^{(2)} &= -F_t^{(1)} (Z_t P_t Z_t^T + H_t) F_t^{(1)}, \\ K_t &= T_t P_{\infty,t} Z_t^T F_t^{(1)}, \\ K_t^{(1)} &= T_t (P_t Z_t^T F_t^{(1)} + P_{\infty,t} Z_t^T F_t^{(2)}), \\ L_t &= T_t - K_t Z_t, \\ L_t^{(1)} &= -K_t^{(1)} Z_t, \\ a_{t+1} &= c_t + T_t a_t + K_t v_t, \\ P_{t+1} &= T_t P_{\infty,t} L_t^{(1)T} + T_t P_t L_t^T + R_t Q_t R_t^T, \\ P_{\infty,t+1} &= T_t P_{\infty,t} L_t^T, \end{aligned}$$

and when $F_{\infty,t} = 0$:

$$\begin{aligned} v_t &= y_t - Z_t a_t, \\ F_t &= Z_t P_t Z_t^T + H_t, \\ K_t &= T_t P_t Z_t^T F_t^{-1}, \\ L_t &= T_t - K_t Z_t, \\ a_{t+1} &= c_t + T_t a_t + K_t v_t, \\ P_{t+1} &= T_t P_t L_t^T + R_t Q_t R_t^T, \\ P_{\infty,t+1} &= T_t P_{\infty,t} T_t^T. \end{aligned}$$

4.2 State smoother

Given the Kalman filter outputs, the state smoother outputs $\hat{\alpha}_t = \mathrm{E}(\alpha_t | y_{1:n})$ and $V_t = \mathrm{Var}(\alpha_t | y_{1:n})$, the smoothed state mean and variance given the whole observation data, by backwards recursion.

To start the backwards recursion, first set $r_n = 0$ and $N_n = 0$, where r_t and N_t are the same size as a_t and P_t respectively. The normal backwards recursion (for $t > d$) is then

$$\begin{aligned} r_{t-1} &= Z_t^T F_t^{-1} v_t + L_t^T r_t, \\ N_{t-1} &= Z_t^T F_t^{-1} Z_t + L_t^T N_t L_t, \\ \hat{\alpha}_t &= a_t + P_t r_{t-1}, \\ V_t &= P_t - P_t N_{t-1} P_t. \end{aligned}$$

For the exact diffuse initialized portion ($t \leq d$) set additional diffuse variables $r_d^{(1)} = 0$ and $N_d^{(1)} = N_d^{(2)} = 0$ corresponding to r_d and N_d respectively, the backwards recursion is

$$\begin{aligned} r_{t-1}^{(1)} &= Z_t^T F_t^{(1)} v_t + L_t^T r_t^{(1)} + L_t^{(1)T} r_t, \\ r_{t-1} &= L_t^T r_t, \\ N_{t-1}^{(2)} &= Z_t^T F_t^{(2)} Z_t + L_t^T N_t^{(2)} L_t + L_t^T N_t^{(1)} L_t^{(1)} \\ &\quad + L_t^{(1)T} N_t^{(1)} L_t + L_t^{(1)T} N_t L_t^{(1)}, \\ N_{t-1}^{(1)} &= Z_t^T F_t^{(1)} Z_t + L_t^T N_t^{(1)} L_t + L_t^{(1)T} N_t L_t, \\ N_{t-1} &= L_t^T N_t L_t, \\ \hat{\alpha}_t &= a_t + P_t r_{t-1} + P_{\infty,t} r_{t-1}^{(1)}, \\ V_t &= P_t - P_t N_{t-1} P_t - (P_{\infty,t} N_{t-1}^{(1)} P_t)^T \\ &\quad - P_{\infty,t} N_{t-1}^{(1)} P_t - P_{\infty,t} N_{t-1}^{(2)} P_{\infty,t}, \end{aligned}$$

if $F_{\infty,t}$ is nonsingular, and

$$\begin{aligned} r_{t-1}^{(1)} &= T_t^T r_t^{(1)}, \\ r_{t-1} &= Z_t^T F_t^{-1} v_t + L_t^T r_t, \\ N_{t-1}^{(2)} &= T_t^T N_t^{(2)} T_t, \\ N_{t-1}^{(1)} &= T_t^T N_t^{(1)} L_t, \\ N_{t-1} &= Z_t^T F_t^{-1} Z_t + L_t^T N_t L_t, \\ \hat{\alpha}_t &= a_t + P_t r_{t-1} + P_{\infty,t} r_{t-1}^{(1)}, \end{aligned}$$

$$\begin{aligned} V_t &= P_t - P_t N_{t-1} P_t - (P_{\infty,t} N_{t-1}^{(1)} P_t)^T \\ &\quad - P_{\infty,t} N_{t-1}^{(1)} P_t - P_{\infty,t} N_{t-1}^{(2)} P_{\infty,t}, \end{aligned}$$

if $F_{\infty,t} = 0$.

4.3 Disturbance smoother

Given the Kalman filter outputs, the disturbance smoother outputs $\hat{\varepsilon}_t = \mathrm{E}(\varepsilon_t|y_{1:n})$, $\mathrm{Var}(\varepsilon_t|y_{1:n})$, $\hat{\eta}_t = \mathrm{E}(\eta_t|y_{1:n})$ and $\mathrm{Var}(\eta_t|y_{1:n})$. Calculate the sequence r_t and N_t by backwards recursion as before, the disturbance can then be estimated by

$$\begin{aligned} \hat{\eta}_t &= Q_t R_t^T r_t, \\ \mathrm{Var}(\eta_t|y_{1:n}) &= Q_t - Q_t R_t^T N_t R_t Q_t, \\ \hat{\varepsilon}_t &= H_t(F_t^{-1} v_t - K_t^T r_t), \\ \mathrm{Var}(\varepsilon_t|y_{1:n}) &= H_t - H_t(F_t^{-1} + K_t^T N_t K_t) H_t, \end{aligned}$$

if $t > d$ or $F_{\infty,t} = 0$, and

$$\begin{aligned} \hat{\eta}_t &= Q_t R_t^T r_t, \\ \mathrm{Var}(\eta_t|y_{1:n}) &= Q_t - Q_t R_t^T N_t R_t Q_t, \\ \hat{\varepsilon}_t &= -H_t K_t^T r_t, \\ \mathrm{Var}(\varepsilon_t|y_{1:n}) &= H_t - H_t K_t^T N_t K_t H_t, \end{aligned}$$

if $t \leq d$ and $F_{\infty,t}$ is nonsingular.

4.4 Simulation smoother

The simulation smoother randomly generates an observation sequence $\tilde{y}_t$ and the underlying state and disturbance sequences $\tilde{\alpha}_t$, $\tilde{\epsilon}_t$ and $\tilde{\eta}_t$ conditional on the model and observation data y_t. The algorithm is as follows (each step is performed for all time points t):

1. Draw random samples of the state and observation disturbances:

$$\begin{aligned} \varepsilon_t^+ &\sim \mathcal{N}(0, H_t), \\ \eta_t^+ &\sim \mathcal{N}(0, Q_t). \end{aligned}$$

2. Draw random samples of the initial state:

$$\alpha_1^+ \sim \mathcal{N}(a_1, P_1).$$

3. Generate state sequences α_t^+ and observation sequences y_t^+ from the random samples.

4. Use disturbance smoothing to obtain

$$\begin{aligned} \hat{\varepsilon}_t &= \mathrm{E}(\varepsilon_t|y_{1:n}), \\ \hat{\eta}_t &= \mathrm{E}(\eta_t|y_{1:n}), \\ \hat{\varepsilon}_t^+ &= \mathrm{E}(\varepsilon_t^+|y_{1:n}^+), \\ \hat{\eta}_t^+ &= \mathrm{E}(\eta_t^+|y_{1:n}^+). \end{aligned}$$

5. Use state smoothing to obtain

$$\begin{aligned}\hat{\alpha}_t &= \mathrm{E}(\alpha_t|y_{1:n}),\\ \hat{\alpha}_t^+ &= \mathrm{E}(\alpha_t^+|y_{1:n}^+).\end{aligned}$$

6. Calculate

$$\begin{aligned}\tilde{\alpha}_t &= \hat{\alpha}_t - \hat{\alpha}_t^+ + \alpha_t^+,\\ \tilde{\varepsilon}_t &= \hat{\varepsilon}_t - \hat{\varepsilon}_t^+ + \varepsilon_t^+,\\ \tilde{\eta}_t &= \hat{\eta}_t - \hat{\eta}_t^+ + \eta_t^+.\end{aligned}$$

7. Generate $\tilde{y}_t$ from the output.

5 Getting Started

The easiest and most frequent way to start using SSM is by constructing predefined models, as opposed to creating a model from scratch. This section presents some examples of simple time series analysis using predefined models, the complete list of available predefined models can be found in Appendix A.

5.1 Preliminaries

Some parts of SSM is programmed in c for maximum efficiency, before using SSM, go to the subfolder `csrc` and execute the script `mexall` to compile and distribute all needed c mex functions.

5.2 Model construction

To create an instance of a predefined model, call the functions `ssm_*` where the wildcard is the short name for the model, with arguments as necessary. For example:

- `model = ssm_llm` creates a local level model.
- `model = ssm_arma(p, q)` creates an ARMA(p, q) model.

The resulting variable `model` is a SSMODEL object and can be displayed just like any other MATLAB variables. To set or change the model parameters, use `model.param`, which is a row vector that behaves like a MATLAB matrix except its size cannot be changed. The initial conditions usually defaults to exact diffuse initialization, where `model.a1` is zero, and `model.P1` is ∞ on the diagonals, but can likewise be changed. Models can be combined by horizontal concatenation, where only the observation disturbance model of the first one will be retained. The class SSMODEL will be presented in detail in later sections.

5.3 Model and state estimation

With the model created, estimation can be performed. SSM expects the data `y` to be a matrix of dimensions $p \times n$, where n is the data size (or time duration). The model parameters are estimated by maximum likelihood, the SSMODEL class method `estimate` performs the estimation. For example:

- `model1 = estimate(y, model0)` estimates the model and stores the result in `model1`, where the parameter values of `model0` is used as initial value.

- `[model1 logL] = estimate(y, model0, psi0, [], optname1, optvalue1, optname2, optvalue2, ...)` estimates the model with `psi0` as the initial parameters using option settings specified with option value pairs, and returns the resulting model and loglikelihood.

After the model is estimated, state estimation can be performed, this is done by the SSMODEL class method `kalman` and `statesmo`, which is the Kalman filter and state smoother respectively.

- `[a P] = kalman(y, model)` applies the Kalman filter on `y` and returns the one-step-ahead state prediction and variance.
- `[alphahat V] = statesmo(y, model)` applies the state smoother on `y` and returns the expected state mean and variance.

The filtered and smoothed state sequences `a` and `alphahat` are `m` $\times$ `n+1` and `m` $\times$ `n` matrices respectively, and the filtered and smoothed state variance sequences `P` and `V` are `m` $\times$ `m` $\times$ `n+1` and `m` $\times$ `m` $\times$ `n` matrices respectively, except if `m` $= 1$, in which case they are squeezed and transposed.

6 SSM Classes

SSM is composed of various MATLAB classes each of which represents specific parts of a general state space model. The class `SSMODEL` represents a state space model, and is the most important class of SSM. It embeds the classes `SSMAT`, `SSDIST`, `SSFUNC` and `SSPARAM`, all designed to facilitate the construction and usage of `SSMODEL` for general data analysis. Consult Appendix B for complete details about these classes.

6.1 Class SSMAT

The class state space matrix (`SSMAT`) forms the basic components of a state space model, they can represent linear transformations that map vectors to vectors (the Z_t, T_t and R_t matrices), or variance matrices of Gaussian disturbances (the H_t and Q_t matrices).

The first data member of class `SSMAT` is `mat`, which is a matrix that represents the stationary part of the state space matrix, and is often the only data member that is needed to represent a state space matrix. For dynamic state space matrices, the additional data members `dmmask` and `dvec` are needed. `dmmask` is a logical matrix the same size as `mat`, and specifies elements of the state space matrix that varies with time, these elements arranged in a column in column-wise order forms the columns of the matrix `dvec`, thus `nnz(dmmask)` is equal to the number of rows in `dvec`, and the number of columns of `dvec` is the number of time points specified for the state space matrix. This is designed so that the code

```
mat(dmmask) = dvec(:, t);
```

gets the value of the state space matrix at time point `t`.

Now these are the constant part of the state space matrix, in that they do not depend on model parameters. The logical matrix `mmask` specifies which elements of the stationary `mat` is dependent on model parameters (variable), while the logical column vector `dvmask` specifies the variable elements of `dvec`. Functions that update the state space matrix take the model parameters as input argument, and should output results so that the following code correctly updates the state space matrix:

```
mat(mmask) = func1(psi);
dvec(dvmask, :) = func2(psi);
```

where `func1` update the stationary part, and `func2` updates the dynamic part. The class `SSMAT` is designed to represent the "structure" and current value of a state space matrix, thus the update functions are not part of the class and will be described in detail in later sections.

Objects of class `SSMAT` behaves like a matrix to a certain extent, calling `size` returns the size of the stationary part, concatenations `horzcat`, `vertcat` and `blkdiag` can be used to combine `SSMAT` with each other and ordinary matrices, and same for `plus`. Data members `mat` and `dvec` can be read and modified provided the structure of the state space matrix stays the same, parenthesis indexing can also be used.

6.2 Class SSDIST

The class state space distributions (`SSDIST`) is a child class of `SSMAT`, and represents non-Gaussian distributions for the observation and state disturbances (H_t and Q_t). Since disturbances are vectors, it's possible to have both elements with Gaussian distributions, and elements with different non-Gaussian distributions. Also in order to use the Kalman filter, non-Gaussian distributions are approximated by Gaussian noise with dynamic variance. Thus the class `SSDIST` needs to keep track of multiple non-Gaussian distributions and also their corresponding disturbance elements, so that the individual Gaussian approximations can be concatenated in the correct order.

As non-Gaussian distributions are approximated by dynamic Gaussian distribution in SSM, its representation need only consist of data required in the approximation process, namely the functions that take disturbance samples as input and outputs the dynamic Gaussian variance. Also a function that outputs the probability of the disturbance samples is needed for loglikelihood calculation. These functions are stored in two data members `matf` and `logpf`, both cell arrays of function handles. A logical matrix `diagmask` is needed to keep track of which elements each non-Gaussian distribution governs, where each column of `diagmask` corresponds to each distribution. Elements not specified in any columns are therefore Gaussian. Lastly a logical vector `dmask` of the same length as `matf` and `logpf` specifies which distributions are variable, for example, t-distributions can be variable if the variance and degree of freedom are model parameters. The update function `func` for `SSDIST` objects should be defined to take parameter values as input arguments and output a cell matrix of function handles such that

```
distf = func(psi);
[matf{dmask}] = distf{:, 1};
[logpf{dmask}] = distf{:, 2};
```

correctly updates the non-Gaussian distributions.

In SSM most non-Gaussian distributions such as exponential family distributions and some heavy-tailed noise distributions are predefined and can be constructed similarly to predefined models, these can then be inserted in stock components to allow for non-Gaussian disturbance generalizations.

6.3 Class SSFUNC

The class state space functions (`SSFUNC`) is another child class of `SSMAT`, and represents nonlinear functions that transform vectors to vectors (Z_t, T_t and R_t). As with non-Gaussian distribution it is possible to "concatenate" nonlinear functions corresponding to different elements of the state vector and destination vector. The dimension of input and output vector and which elements they corresponds to will have to be kept track of for each nonlinear function, and their linear approximations are concatenated using these information.

Both the function itself and its derivative are needed to calculate the linear approximation, these are stored in data members `f` and `df`, both cell arrays of function handles. `horzmask` and `vertmask` are both logical matrices where each column specifies the elements of the input and output vector for each function, respectively. In this

way the dimensions of the approximating matrix for the `i`th function can be determined as `nnz(vertmask(:, i)) × nnz(horzmask(:, i))`. The logical vector `fmask` specifies which functions are variable with respect to model parameters. The update function `func` for `SSFUNC` objects should be defined to take parameter values as input arguments and output a cell matrix of function handles such that

```
funcf = func(psi);
[f{fmask}] = funcf{:, 1};
[df{fmask}] = funcf{:, 2};
```

correctly updates the nonlinear functions.

6.4 Class SSPARAM

The class state space parameters (`SSPARAM`) stores and organizes the model parameters, each model parameter has a name, value, transform and possible constraints. Transforms are necessary because most model parameters have a restricted domain, but most numerical optimization routines generally operate over the whole real line. For example a variance parameter σ^2 are suppose to be positive, so if σ^2 is optimized directly there's the possibility of error, but by defining $\psi = \log(\sigma^2)$ and optimizing ψ instead, this problem is avoided. Model parameter transforms are not restricted to single parameters, sometimes a group of parameters need to be transformed together, the same goes for constraints. Hence the class `SSPARAM` also keeps track of parameter grouping.

Like `SSMAT`, `SSPARAM` objects can be treated like a row vector of parameters to a certain extent. Specifically, it is possible to horizontally concatenate `SSPARAM` objects as a means of combining parameter groups. Other operations on `SSPARAM` objects include getting and setting the parameter values and transformed values. The untransformed values are the original parameter values meaningful to the model defined, and will be referred to as `param` in code examples; the transformed values are meant only as input to optimization routines and have no direct interpretation in terms of the model, they are referred to as `psi` in code examples, but the user will rarely need to use them directly if predefined models are used.

6.5 Class SSMODEL

The class state space model (`SSMODEL`) represents state space models by embedding `SSMAT`, `SSDIST`, `SSFUNC` and `SSPARAM` objects and also keeping track of update functions, parameter masks and general model component information. The basic constituent elements of a model is described earlier as `Z`, `T`, `R`, `H`, `Q`, `c`, `a1` and `P1`. `Z`, `T` and `R` are vector transformations, the first two can be `SSFUNC` or `SSMAT` objects, but the last one can only be a `SSMAT` object. `H` and `Q` govern the disturbances and can be `SSDIST` or `SSMAT` objects. `c`, `a1` and `P1` can only be `SSMAT` objects. These form the "constant" part of the model specification, and is the only part needed in Kalman filter and related state estimation algorithms.

The second part of `SSMODEL` concerns the update functions and model parameters, general specifications for the model estimation process. `func` and `grad` are cell arrays of update functions and their gradient for the basic model elements respectively. It is possible for a single function to update multiple parts of the model and these information are stored in the data member `A`, a `length(func) × 18` adjacency matrix. Each row of `A` represents one update function, and each column represents one updatable element. The `18` updatable model elements in order are

```
H Z T R Q c a1 P1 Hd Zd Td Rd Qd cd Hng Qng Znl Tnl
```

where the `d` suffix means dynamic, `ng` means non-Gaussian and `nl` means nonlinear. It is therefore possible for a function that updates `Z` to output `[vec dsubvec funcf]` updating the stationary part of `Z`, dynamic part of `Z` and the nonlinear functions in `Z` respectively. The row in `A` for this function would be

```
H Z T R Q c a1 P1 Hd Zd Td Rd Qd cd Hng Qng Znl Tnl
0 1 0 0 0 0  0  0  0  1  0  0  0  0   0   0   1   0
```

Any function can update any combination of these 18 elements, the only restriction is that the output should follow the order set in `A` with possible omissions.

On the other hand `A` also allows many functions to update a single model element, this is the main mechanism that enables smooth model combination. Suppose `T1` of model 1 is updated by `func1`, and `T2` of model 2 by `func2`, combination of models 1 and 2 requires block diagonal concatenation of `T1` and `T2`: `T = blkdiag(T1, T2)`. It is easy to see that the vertical concatenation of the output of `func1` and `func2` correctly updates the new matrix `T` as follows

```
vec1 = func1(psi);
vec2 = func2(psi);
T.mat(T.mmask) = [vec1; vec2];
```

This also holds for horizontal concatenation, but for vertical concatenation of more than one column this would fail, luckily this case never occurs when combining state space models, so can be safely ignored. The updates of dynamic, non-Gaussian and nonlinear elements can also be combined in this way, in fact, the whole scheme of the update functions are designed with combinable output in mind. For an example adjacency matrix `A`

```
        H Z T R Q c a1 P1 Hd Zd Td Rd Qd cd Hng Qng Znl Tnl
func1   1 0 1 0 1 0  0  0  0  0  0  0  0  0   0   0   1   0
func2   0 0 1 0 0 0  0  0  0  1  0  0  0  0   0   0   0   0
func3   0 1 1 0 1 0  0  0  0  1  0  0  0  0   0   0   1   0
```

model update would then proceed by first invoking the three update functions

```
[Hvec1 Tvec1 Qvec1 Zfuncf1] = func1(psi);
[Tvec2 Zdsubvec2] = func2(psi);
[Zvec3 Tvec3 Qvec3 Zdsubvec3 Zfuncf3] = func3(psi);
```

and then update the various elements

```
H.mat(H.mmask) = Hvec1;
Z.mat(Z.mmask) = Zvec3;
T.mat(T.mmask) = [Tvec1; Tvec2; Tvec3];
Q.mat(Q.mmask) = [Qvec1; Qvec3];
Z.dvec(Z.dvmask, :)  = [Zdsubvec2; Zdsubvec3];
Z.f(Z.fmask) = [Zfuncf1{:, 1}; Zfuncf3{:, 1}];
Z.df(Z.fmask) = [Zfuncf1{:, 2}; Zfuncf3{:, 2}];
```

Now to facilitate model construction and alteration, functions that need adjacency matrix `A` as input actually expects another form based on strings. Each of the 18 elements shown earlier is represented by the corresponding strings, `'H'`, `'Qng'`, `'Zd'`, ... etc., and each row of `A` is a single string in a cell array of strings, where each string is a concatenation of various elements in any order. This form (a cell array of strings) is called "adjacency string" and is used in all functions that require an adjacency matrix as input.

The remaining data members of `SSMODEL` are `psi` and `pmask`. `psi` is a `SSPARAM` object that stores the model parameters, and `pmask` is a cell array of logical row vectors that specifies which parameters are required by each update function. So each update function are in fact provided a subset of model parameters `func`i`(psi.value(pmask{`i`}))`. It is trivial to combine these two data members when combining models.

In summary the class SSMODEL can be divided into two parts, a "constant" part that corresponds to an instance of a theoretical state space model given all model parameters, which is the only part required to run Kalman filter and related state estimation routines, and a "variable" part that keeps track of the model parameters and how each parameter effects the values of various model elements, which is used (in addition to the "constant" part) in model estimation routines that produce parameter estimates. This division is the careful separation of the "structure" of a model and its "instantiation" given model parameters, and allows logical separation of various structural elements of the model (with the classes SSMAT, SSDIST and SSFUNC) to facilitate model combination and usage, without compromising the ability to efficiently define and combine complex model update mechanisms.

7 SSM Functions

There are several types of functions in SSM besides class methods, these are data analysis functions, stock element functions and various helper functions. Most data analysis functions are Kalman filter related and implemented as SSMODEL methods, but for the user this is transparent since class methods and functions are called in the same way in MATLAB. Examples for data analysis functions include kalman, statesmo and loglik, all of which uses the kalman filter and outputs the filtered states, smoothed states and loglikelihood respectively. These and similar functions take three arguments, the data y, which must be p × n, where p is the number of variables in the data and n the data size, the model model, a SSMODEL object, and optional analysis option name value pairs (see function setopt in Appendix C.4), where settings such as tolerance can be specified. The functions linear and gauss calculates the linear Gaussian approximation models, and requires y, model and optional initial state estimate alpha0 as input arguments. The function estimate runs numerical optimization routines for model estimation; model updates, necessary approximations and loglikelihood calculations are done automatically in the procedure, input arguments are y, model and initial parameter estimate psi0, the function outputs the maximum likelihood model. There are also some functions specific to ARIMA-type models, such as Hillmer-Tiao decomposition [2] and TRAMO model selection [3].

Stock element functions create predefined models and components, they range from the general to the specific. These functions have names of the form (element)_(description), for example, the function mat_arma creates matrices for the ARMA model. Currently there are five types of stock element functions: ssm_* creates state space models (SSMODELs), fun_* creates model update functions, mat_* creates matrix structures, dist_* creates non-Gaussian distributions (SSDISTs), and x_* creates stock regression variables such as trading day variables.

Miscellaneous helper functions provide additional functionality unsuitable to be implemented in the previous two categories. The most important of which is the function setopt, which specifies the global options or settings structure that is used by almost every data analysis function. One time overrides of individual options are possible for each function call, and each option can be individually specified, examples of possible options include tolerance and verbosity. Other useful functions include ymarray which generates a year, month array from starting year, starting month and number of months and is useful for functions like x_td and x_ee which take year, month arrays as input.

8 Data Analysis Examples

Many of the data analysis examples are based on the data in the book by Durbin and Koopman [1], to facilitate easy comparison and understanding of the method used here.

8.1 Structural time series models

In this example data on road accidents in Great Britain [4] is analyzed using structural time series models following [1]. The purpose the the analysis is to assess the effect of seat belt laws on road accident casualties, with individual monthly figures for drivers, front seat passengers and rear seat passengers. The monthly price of petrol and average number of kilometers traveled will be used as regression variables. The data is from January 1969 to December 1984.

8.1.1 Univariate analysis

The drivers series will be analyzed with univariate structural time series model, which consists of local level component μ_t, trigonometric seasonal component γ_t, regression component (on price of petrol) and intervention component (introduction of seat belt law) βx_t. The model equation is

$$y_t = \mu_t + \gamma_t + \beta x_t + \varepsilon_t, \tag{7}$$

where ε_t is the observation noise. The following code example constructs this model:

```
lvl = ssm_llm;
seas = ssm_seasonal('trig1', 12);
intv = ssm_intv(n, 'step', 170);
reg = ssm_reg(petrol, 'price of petrol');
bstsm = [lvl seas intv reg];
bstsm.name = 'Basic structural time series model';
bstsm.param = [10 0.1 0.001];
```

The MATLAB display for object `bstsm` is

```
bstsm =

    Basic structural time series model
    ==================================

    p = 1
    m = 14
    r = 12
    n = 192


    Components (5):
    ==================

    [Gaussian noise]
        p               = 1
    [local polynomial trend]
        d               = 0
    [seasonal]
        subtype         = trig1
        s               = 12
    [intervention]
        subtype         = step
```

```
    tau             = 170
[regression]
    variable        = price of petrol


H matrix (1, 1)
-------------------

  10.0000


Z matrix (1, 14, 192)
----------------------

Stationary part:

    1    1    0    1    0    1    0    1    0    1    0    1  DYN  DYN


Dynamic part:

Columns 1 through 9

         0         0         0         0         0         0         0         0         0
  -2.2733   -2.2792   -2.2822   -2.2939   -2.2924   -2.2968   -2.2655   -2.2626   -2.2655

Columns 10 through 18

         0         0         0         0         0         0         0         0         0
  -2.2728   -2.2757   -2.2828   -2.2899   -2.2956   -2.3012   -2.3165   -2.3192   -2.3220


                                      ...

Columns 181 through 189

         1         1         1         1         1         1         1         1         1
  -2.1390   -2.1646   -2.1565   -2.1597   -2.1644   -2.1648   -2.1634   -2.1646   -2.1707

Columns 190 through 192

         1         1         1
  -2.1502   -2.1539   -2.1536


T matrix (14, 14)
-------------------
```

```
Columns 1 through 9

         1         0         0         0         0         0         0         0         0
         0    0.8660    0.5000         0         0         0         0         0         0
         0   -0.5000    0.8660         0         0         0         0         0         0
         0         0         0    0.5000    0.8660         0         0         0         0
         0         0         0   -0.8660    0.5000         0         0         0         0
         0         0         0         0         0    0.0000         1         0         0
         0         0         0         0         0        -1    0.0000         0         0
         0         0         0         0         0         0         0   -0.5000    0.8660
         0         0         0         0         0         0         0   -0.8660   -0.5000
         0         0         0         0         0         0         0         0         0
         0         0         0         0         0         0         0         0         0
         0         0         0         0         0         0         0         0         0
         0         0         0         0         0         0         0         0         0
         0         0         0         0         0         0         0         0         0

Columns 10 through 14

         0         0         0         0         0
         0         0         0         0         0
         0         0         0         0         0
         0         0         0         0         0
         0         0         0         0         0
         0         0         0         0         0
         0         0         0         0         0
         0         0         0         0         0
         0         0         0         0         0
   -0.8660    0.5000         0         0         0
   -0.5000   -0.8660         0         0         0
         0         0        -1         0         0
         0         0         0         1         0
         0         0         0         0         1


R matrix (14, 12)
-------------------

    1    0    0    0    0    0    0    0    0    0    0    0
    0    1    0    0    0    0    0    0    0    0    0    0
    0    0    1    0    0    0    0    0    0    0    0    0
    0    0    0    1    0    0    0    0    0    0    0    0
    0    0    0    0    1    0    0    0    0    0    0    0
    0    0    0    0    0    1    0    0    0    0    0    0
    0    0    0    0    0    0    1    0    0    0    0    0
    0    0    0    0    0    0    0    1    0    0    0    0
    0    0    0    0    0    0    0    0    1    0    0    0
    0    0    0    0    0    0    0    0    0    1    0    0
```

```
     0     0     0     0     0     0     0     0     0     0     1     0
     0     0     0     0     0     0     0     0     0     0     0     1
     0     0     0     0     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0     0     0     0     0


Q matrix (12, 12)
-------------------

Columns 1 through 9

   0.1000         0         0         0         0         0         0         0         0
        0    0.0010         0         0         0         0         0         0         0
        0         0    0.0010         0         0         0         0         0         0
        0         0         0    0.0010         0         0         0         0         0
        0         0         0         0    0.0010         0         0         0         0
        0         0         0         0         0    0.0010         0         0         0
        0         0         0         0         0         0    0.0010         0         0
        0         0         0         0         0         0         0    0.0010         0
        0         0         0         0         0         0         0         0    0.0010
        0         0         0         0         0         0         0         0         0
        0         0         0         0         0         0         0         0         0
        0         0         0         0         0         0         0         0         0

Columns 10 through 12

        0         0         0
        0         0         0
        0         0         0
        0         0         0
        0         0         0
        0         0         0
        0         0         0
        0         0         0
        0         0         0
   0.0010         0         0
        0    0.0010         0
        0         0    0.0010


P1 matrix (14, 14)
-------------------

  Inf     0     0     0     0     0     0     0     0     0     0     0     0     0
    0   Inf     0     0     0     0     0     0     0     0     0     0     0     0
    0     0   Inf     0     0     0     0     0     0     0     0     0     0     0
    0     0     0   Inf     0     0     0     0     0     0     0     0     0     0
    0     0     0     0   Inf     0     0     0     0     0     0     0     0     0
```

```
    0    0    0    0    0  Inf    0    0    0    0    0    0    0    0
    0    0    0    0    0    0  Inf    0    0    0    0    0    0    0
    0    0    0    0    0    0    0  Inf    0    0    0    0    0    0
    0    0    0    0    0    0    0    0  Inf    0    0    0    0    0
    0    0    0    0    0    0    0    0    0  Inf    0    0    0    0
    0    0    0    0    0    0    0    0    0    0  Inf    0    0    0
    0    0    0    0    0    0    0    0    0    0    0  Inf    0    0
    0    0    0    0    0    0    0    0    0    0    0    0  Inf    0
    0    0    0    0    0    0    0    0    0    0    0    0    0  Inf
```

```
Adjacency matrix (3 functions):
-------------------------------------------------------------------------
Function                H Z T R Q c a1 P1 Hd Zd Td Rd Qd cd Hng Qng Znl Tnl
fun_var/psi2var         1 0 0 0 0 0 0  0  0  0  0  0  0  0  0   0   0   0
fun_var/psi2var         0 0 0 0 1 0 0  0  0  0  0  0  0  0  0   0   0   0
fun_dupvar/psi2dupvar1  0 0 0 0 1 0 0  0  0  0  0  0  0  0  0   0   0   0


Parameters (3)
-------------------------------------
Name                Value
epsilon var         10
zeta var            0.1
omega var           0.001
```

The constituting components of the model and the model matrices are all displayed, note that `Z` is dynamic with the intervention and regression variables, and matrices `c` and `a1` are omitted if they are zero, also `H` and `Q` take on values determined by the parameters set.

With the model constructed, estimation can proceed with the code:

```
[bstsm logL] = estimate(y, bstsm);
[alphahat V] = statesmo(y, bstsm);
irr = disturbsmo(y, bstsm);
ycom = signal(alphahat, bstsm);
ylvl = sum(ycom([1 3 4], :), 1);
yseas = ycom(2, :);
```

The estimated model parameters are `[0.0037862 0.00026768 1.162e-006]`, which can be obtained by displaying `bstsm.param`, the loglikelihood is `175.7790`. State and disturbance smoothing is performed with the estimated model, and the smoothed state is transformed into signal components by `signal`. Because $\mathtt{p} = 1$, the output `ycom` is $\mathtt{M} \times \mathtt{n}$, where `M` is the number of signal components. The level, intervention and regression are summed as the total data level, separating seasonal influence. Using MATLAB graphic functions the individual signal components and data are visualized:

```
subplot(3, 1, 1), plot(time, y, 'r:', 'DisplayName', 'drivers'), hold all,
                  plot(time, ylvl, 'DisplayName', 'est. level'), hold off,
                  title('Level'), ylim([6.875 8]), legend('show');
```

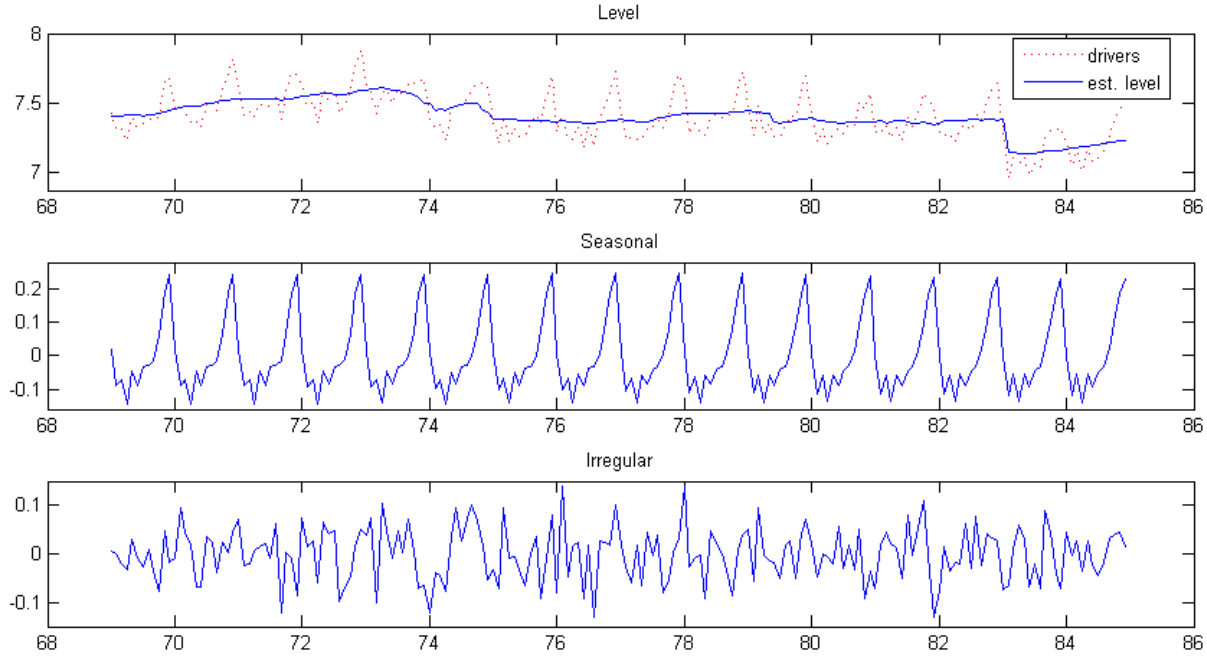

Figure 1: Driver casualties estimated by basic structural time series model

```
subplot(3, 1, 2), plot(time, yseas), title('Seasonal'), ylim([-0.16 0.28]);
subplot(3, 1, 3), plot(time, irr), title('Irregular'), ylim([-0.15 0.15]);
```

The graphical output is shown in figure 1. The coefficient for the intervention and regression is defined as part of the state vector in this model, so they can be obtained from the last two elements of the smoothed state vector (due to the order of component concatenation) at the last time point. Coefficient for the intervention is `alphahat(13, n) = -0.23773`, and coefficient for the regression (price of petrol) is `alphahat(14, n) = -0.2914`. In this way diagnostics of these coefficient estimates can also be obtained by the smoothed state variance `V`.

8.1.2 Bivariate analysis

The front seat passenger and rear seat passenger series will be analyzed together using bivariate structural time series model, with components as before. Specifically, separate level and seasonal components are defined for both series, but the disturbances are assumed to be correlated. To reduce the number of parameters estimated the seasonal component is assumed to be fixed, so that the total number of parameters is six. We also include regression on the price of petrol and kilometers traveled, and intervention for only the first series, since the seat belt law only effects the front seat passengers. The following is the model construction code:

```
bilvl = ssm_mvllm(2);
biseas = ssm_mvseasonal(2, [], 'trig fixed', 12);
biintv = ssm_mvintv(2, n, {'step' 'null'}, 170);
bireg = ssm_mvreg(2, [petrol; km]);
bistsm = [bilvl biseas biintv bireg];
bistsm.name = 'Bivariate structural time series model';
```

The model is then estimated with carefully chosen initial values, and state smoothing and signal extraction proceeds as before:

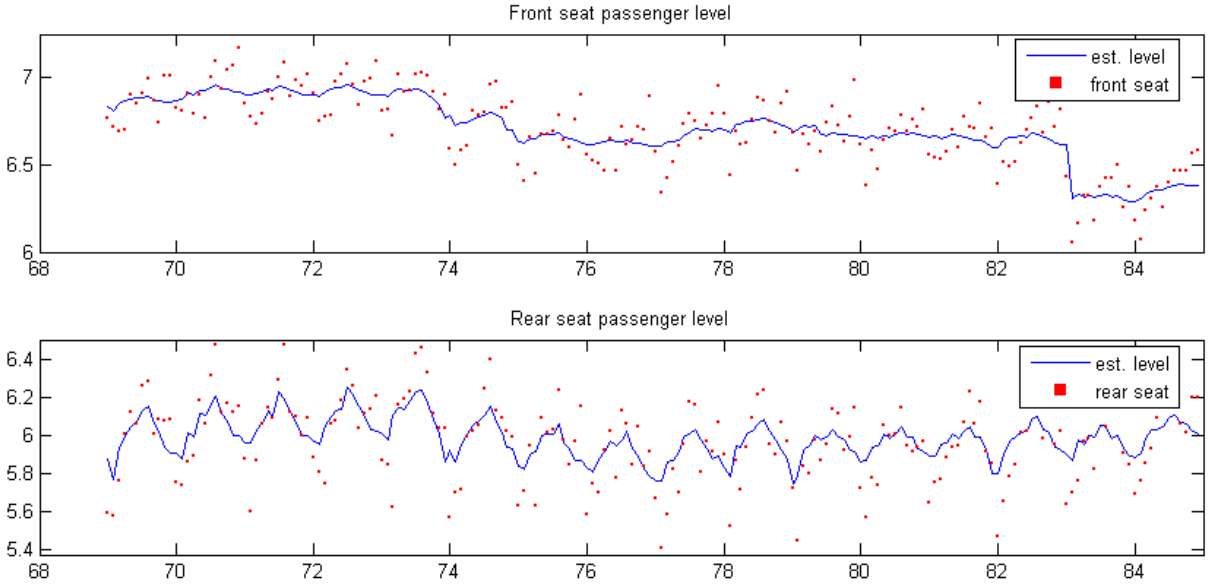

Figure 2: Passenger casualties estimated by bivariate structural time series model

```
[bistsm logL] = estimate(y2, bistsm, [0.0054 0.0086 0.0045 0.00027 0.00024 0.00023]);
[alphahat V] = statesmo(y2, bistsm);
y2com = signal(alphahat, bistsm);
y2lvl = sum(y2com(:,:, [1 3 4]), 3);
y2seas = y2com(:,:, 2);
```

When `p` > 1 the output from `signal` is of dimension `p` $\times$ `n` $\times$ `M`, where `M` is the number of components. The level, regression and intervention are treated as one component of data level as before, separated from the seasonal component. The components estimated for the two series is plotted:

```
subplot(2, 1, 1), plot(time, y2lvl(1, :), 'DisplayName', 'est. level'), hold all,
                  scatter(time, y2(1, :), 8, 'r', 's', 'filled', 'DisplayName', 'front seat'), hold off,
                  title('Front seat passenger level'), xlim([68 85]), ylim([6 7.25]), legend('show');
subplot(2, 1, 2), plot(time, y2lvl(2, :), 'DisplayName', 'est. level'), hold all,
                  scatter(time, y2(2, :), 8, 'r', 's', 'filled', 'DisplayName', 'rear seat'), hold off,
                  title('Rear seat passenger level'), xlim([68 85]), ylim([5.375 6.5]), legend('show');
```

The graphical output is shown in figure 2. The intervention coefficient for the front passenger series is obtained by `alphahat(25, n) = -0.30025`, the next four elements are the coefficients of the regression of the two series on the price of petrol and kilometers traveled.

8.2 ARMA models

In this example the difference of the number of users logged on an internet server [5] is analyzed by ARMA models, and model selection via BIC and missing data analysis are demonstrated. To select an appropriate ARMA(p, q) model for the data various values for p and q are tried, and the BIC of the estimated model for each is recorded, the model with the lowest BIC value is chosen.

```
for p = 0 : 5
    for q = 0 : 5
        [arma logL output] = estimate(y, ssm_arma(p, q), 0.1);
```

```
        BIC(p+1, q+1) = output.BIC;
    end
end
[m i] = min(BIC(:));
arma = estimate(y, ssm_arma(mod(i-1, 6), floor((i-1)/6)), 0.1);
```

The BIC values obtained for each model is as follows:

	$q=0$	$q=1$	$q=2$	$q=3$	$q=4$	$q=5$
$p=0$	6.3999	5.6060	5.3299	5.3601	5.4189	5.3984
$p=1$	5.3983	5.2736	5.3195	5.3288	5.3603	5.3985
$p=2$	5.3532	5.3199	5.3629	5.3675	5.3970	5.4436
$p=3$	5.2765	5.3224	5.3714	5.4166	5.4525	5.4909
$p=4$	5.3223	5.3692	5.4142	5.4539	5.4805	5.4915
$p=5$	5.3689	5.4124	5.4617	5.5288	5.5364	5.5871

The model with the lowest BIC is ARMA$(1,1)$, second lowest is ARMA$(3,0)$, the former model is chosen for subsequent analysis.

Next missing data is simulated by setting some time points to `NaN`, model and state estimation can still proceed normally with missing data present.

```
y([6 16 26 36 46 56 66 72 73 74 75 76 86 96]) = NaN;
arma = estimate(y, arma, 0.1);
yf = signal(kalman(y, arma), arma);
```

Forecasting is equivalent to treating future data as missing, thus the data set `y` is appended with as many `NaN` values as the steps ahead to forecast. Using the previous estimated ARMA$(1,1)$ model the Kalman filter will then effectively predict future data points.

```
[a P v F] = kalman([y repmat(NaN, 1, 20)], arma);
yf = signal(a(:, 1:end-1), arma);
conf50 = 0.675*realsqrt(F); conf50(1:n) = NaN;
plot(yf), hold all, plot([yf+conf50; yf-conf50]', 'g:'),
scatter(1:length(y), y, 10, 'r', 's', 'filled'), hold off, ylim([-15 15]);
```

The plot of the forecast and its confidence interval is shown in figure 3. Note that if the Kalman filter is replaced with the state smoother, the forecasted values will still be the same.

8.3 Cubic spline smoothing

The general state space formulation can also be used to do cubic spline smoothing, by putting the cubic spline into an equivalent state space form, and accounting for the continuous nature of such smoothing procedures. Here the continuous acceleration data of a simulated motorcycle accident [6] is smoothed by the cubic spline model, which is predefined.

```
spline = estimate(y, ssm_spline(delta), [1 0.1]);
[alphahat V] = statesmo(y, spline);
conf95 = squeeze(1.96*realsqrt(V(1, 1, :)))';
[eps eta epsvar] = disturbsmo(y, spline);
```

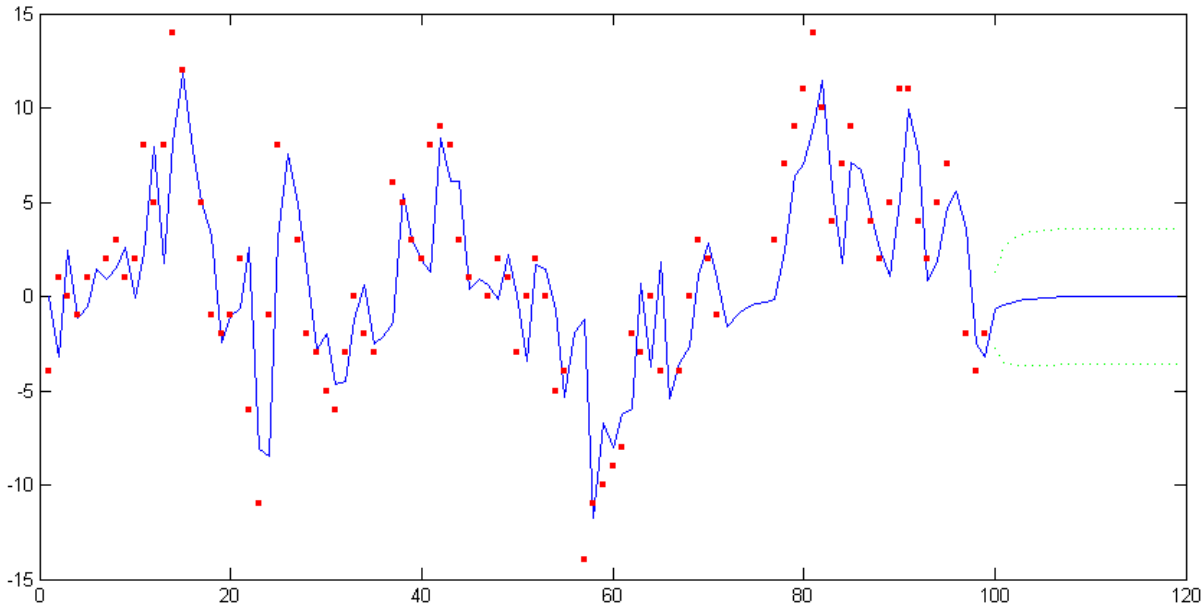

Figure 3: Forecast using ARMA$(1, 1)$ model with 50% confidence interval

The smoothed data and standardized irregular is plotted and shown in figure 4.

```
subplot(2, 1, 1), plot(time, alphahat(1, :), 'b'), hold all,
                  plot(time, [alphahat(1, :) + conf95; alphahat(1, :) - conf95], 'b:'),
                  scatter(time, y, 10, 'r', 's', 'filled'), hold off,
                  title('Spline and 95% confidence intervals'), ylim([-140 80]),
                  set(gca,'YGrid','on');
subplot(2, 1, 2), scatter(time, eps./realsqrt(epsvar), 10, 'r', 's', 'filled'),
                  title('Standardized irregular'), set(gca,'YGrid','on');
```

It is seen from figure 4 that the irregular may be heteroscedastic, an easy *ad hoc* solution is to model the changing variance of the irregular by a continuous version of the local level model. Assume the irregular variance is $\sigma_\varepsilon^2 h_t^2$ at time point t and $h_1 = 1$, then we model the absolute value of the smoothed irregular `abs(eps)` with h_t as its level. The continuous local level model with level h_t needs to be constructed from scratch.

```
contllm = ssmodel('', 'continuous local level', ...
                  0, 1, 1, 1, ssmat(0, [], true, zeros(1, n), true), ...
                  'Qd', {@(X) exp(2*X)*delta}, {[]}, ssparam({'zeta var'}, '1/2 log'));
contllm = [ssm_gaussian contllm];
alphahat = statesmo(abs(eps), estimate(abs(eps), contllm, [1 0.1]));
h2 = (alphahat/alphahat(1)).^2;
```

`h2` is then the relative magnitude of the noise variance h_t^2 at each time point, which can be used to construct a custom dynamic observation noise model as follows.

```
hetnoise = ssmodel('', 'Heteroscedastic noise', ...
                   ssmat(0, [], true, zeros(1, n), true), zeros(1, 0), [], [], [], ...
                   'Hd', {@(X) exp(2*X)*h2}, {[]}, ssparam({'epsilon var'}, '1/2 log'));
hetspline = estimate(y, [hetnoise spline], [1 0.1]);
[alphahat V] = statesmo(y, hetspline);
```

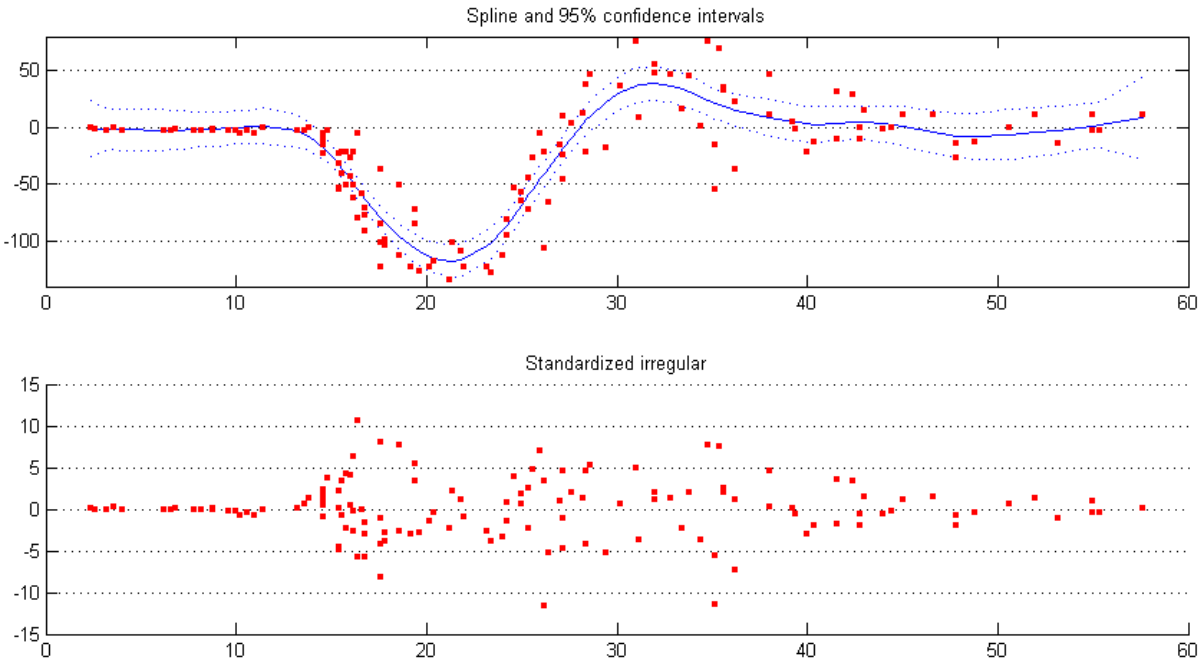

Figure 4: Cubic spline smoothing of motorcycle acceleration data

```
conf95 = squeeze(1.96*realsqrt(V(1, 1, :)))';
[eps eta epsvar] = disturbsmo(y, hetspline);
```

The smoothed data and standardized irregular with heteroscedastic assumption is shown in figure 5, it is seen that the confidence interval shrank, especially at the start and end of the series, and the irregular is slightly more uniform.

In summary the motorcycle series is first modeled by a cubic spline model with Gaussian irregular assumption, then the smoothed irregular magnitude itself is modeled with a local level model. Using the irregular level estimated at each time point as the relative scale of irregular variance, a new heteroscedastic irregular continuous model is constructed with the estimated level built-in, and plugged into the cubic spline model to obtain new estimates for the motorcycle series.

8.4 Poisson distribution error model

The road accident casualties and seat belt law data analyzed in section 8.1 also contains a monthly van driver casualties series. Due to the smaller numbers of van driver casualties the Gaussian assumption is not justified in this case, previous methods cannot be applied. Here a poisson distribution is assumed for the data, the mean is $\exp(\theta_t)$ and the observation distribution is

$$\log p(y_t|\theta_t) = \theta_t^T y_t - \exp(\theta_t) - \log y_t!$$

The signal θ_t in turn is modeled with the structural time series model. The total model is then constructed by concatenating a poisson distribution model with a standard structural time series model, the former model will replace the default Gaussian noise model.

```
pbstsm = [ssm_poisson ssm_llm ...
          ssm_seasonal('dummy fixed', 12) ssm_intv(n, 'step', 170)];
pbstsm.name = 'Poisson basic STSM';
```

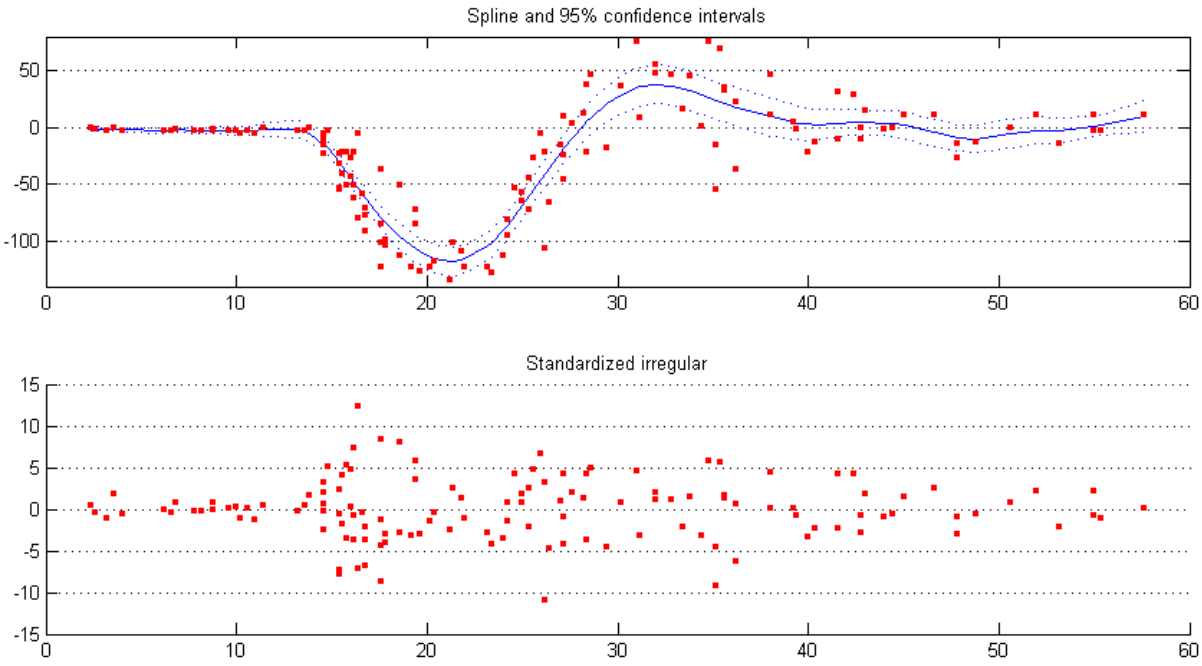

Figure 5: Cubic spline smoothing of motorcycle acceleration data with heteroscedastic noise

Model estimation will automatically calculate the Gaussian approximation to the poisson model. Since this is an exponential family distribution the data y_t also need to be transformed to $\tilde{y}_t$, which is stored in the output argument `output`, and used in place of y_t for all functions implementing linear Gaussian (Kalman filter related) algorithms. The following is the code for model and state estimation

```
[pbstsm logL output] = estimate(y, pbstsm, 0.0006, [], 'fmin', 'bfgs', 'disp', 'iter');
[alphahat V] = statesmo(output.ytilde, pbstsm);
thetacom = signal(alphahat, pbstsm);
```

Note that the original data `y` is input to the model estimation routine, which also calculates the transform `ytilde`. The model estimated then has its Gaussian approximation built-in, and will be treated by the state smoother as a linear Gaussian model, hence the transformed data `ytilde` needs to be used as input. The signal components `thetacom` obtained from the smoothed state `alphahat` is the components of θ_t, the mean of `y` can then be estimated by `exp(sum(thetacom, 1))`.

The exponentiated level $\exp(\theta_t)$ with the seasonal component eliminated is compared to the original data in figure 6. The effect of the seat belt law can be clearly seen.

```
thetalvl = thetacom(1, :) + thetacom(3, :);
plot(time, exp(thetalvl), 'DisplayName', 'est. level'), hold all,
plot(time, y, 'r:', 'DisplayName', 'data'), hold off, ylim([1 18]), legend('show');
```

8.5 t-distribution models

In this example another kind of non-Gaussian models, t-distribution models is used to analyze quarterly demand for gas in UK [7]. A structural time series model with a t-distribution heavy-tailed observation noise is constructed similar to the last section, and model estimation and state smoothing performed.

```
tstsm = [ssm_t ssm_llt ssm_seasonal('dummy', 4)];
```

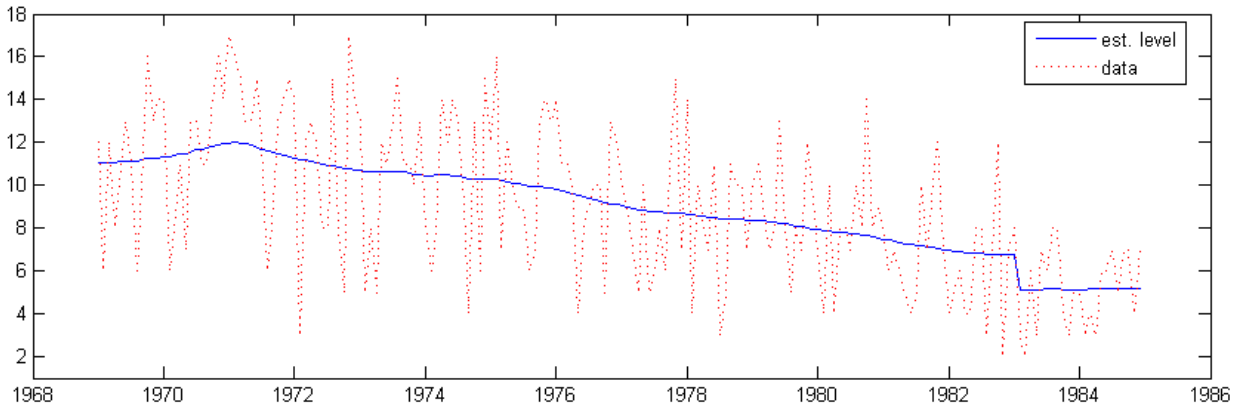

Figure 6: Van driver casualties and estimated level

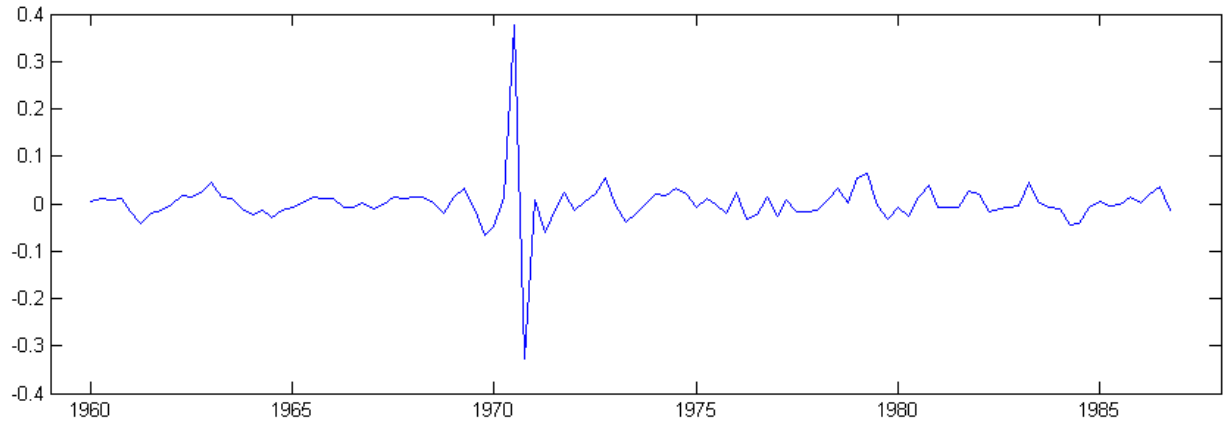

Figure 7: t-distribution irregular

```
tstsm = estimate(y, tstsm, [0.0018 4 7.7e-10 7.9e-6 0.0033], [], 'fmin', 'bfgs');
[alpha irr] = fastsmo(y, tstsm);
plot(time, irr), title('Irregular component'), xlim([1959 1988]), ylim([-0.4 0.4]);
```

Since t-distribution is not an exponential family distribution, data transformation is not necessary, and `y` is used throughout. A plot of the irregular in figure 7 shows that potential outliers with respect to the Gaussian assumption has been detected by the use of heavy-tailed distribution.

8.6 Hillmer-Tiao decomposition

In this example seasonal adjustment is performed by the Hillmer-Tiao decomposition [2] of airline models. The data (Manufacturing and reproducing magnetic and optical media, U.S. Census Bureau) is fitted with the airline model, then the estimated model is Hillmer-Tiao decomposed into an ARIMA components model with trend and seasonal components. The same is done for the generalized airline model[1] [8], and the seasonal adjustment results are compared. The following are the code to perform seasonal adjustment with the airline model:

```
air = estimate(y, ssm_airline, 0.1);
aircom = ssmhtd(air);
```

[1] Currently renamed Frequency Specific ARIMA model.

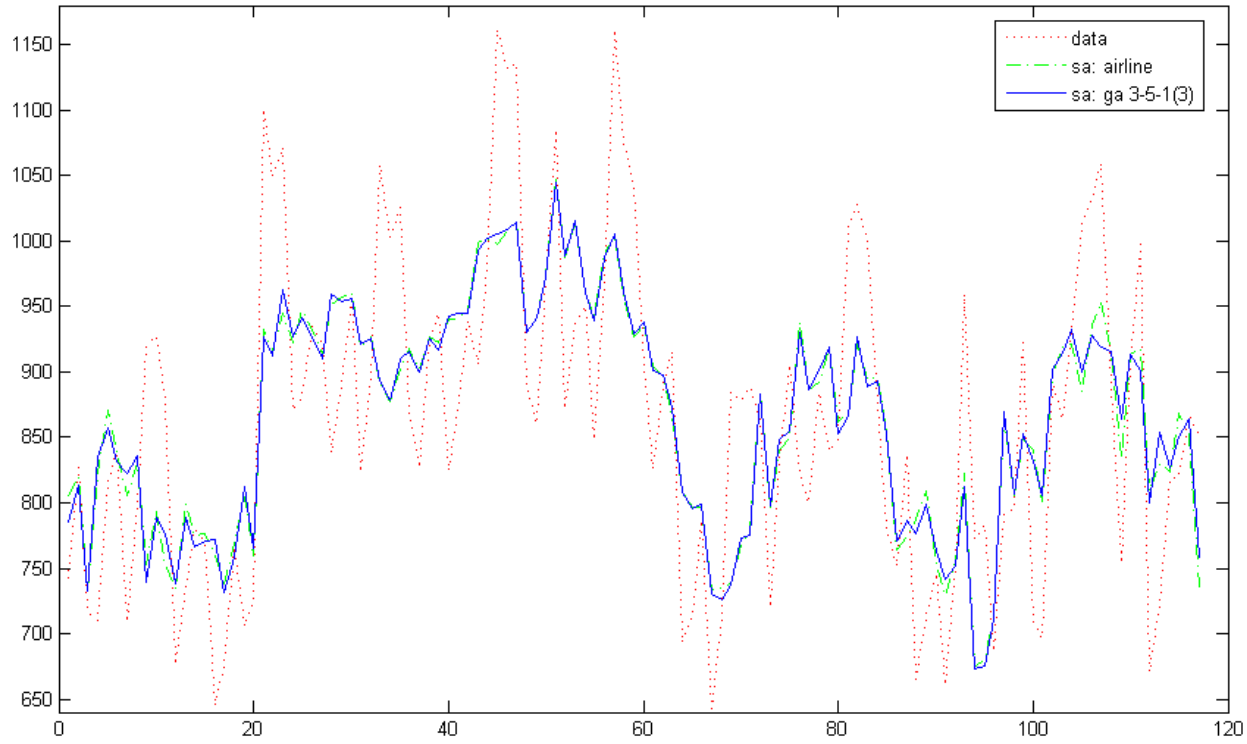

Figure 8: Seasonal adjustment with airline and generalized airline models

```
ycom = signal(statesmo(y, aircom), aircom);
airseas = ycom(2, :);
```

`aircom` is the decomposed ARIMA components model corresponding to the estimated airline model, and `airseas` is the seasonal component, which will be subtracted out of the data `y` to obtain the seasonal adjusted series. `ssmhtd` automatically decompose ARIMA type models into trend, seasonal and irregular components, plus any extra MA components as permissible.

The same seasonal adjustment procedure is then done with the generalized airline model, using parameter estimates from the airline model as initial parameters:

```
param0 = air.param([1 2 2 3]); param0(1:3) = -param0(1:3); param0(2:3) = param0(2:3).^(1/12);
ga = estimate(y, ssm_genair(3, 5, 3), param0);
gacom = ssmhtd(ga);
ycom = signal(statesmo(y, gacom), gacom);
gaseas = ycom(2, :);
```

The code creates a generalized airline 3-5-1 model, Hillmer-Tiao decomposition produces the same components as for the airline model since the two models have the same order. From the code it can be seen that the various functions work transparently across different ARIMA type models. Figure 8 shows the comparison between the two seasonal adjustment results.

References

[1] J. Durbin and S. J. Koopman. *Time Series Analysis by State Space Methods*. Oxford University Press, New York, 2001.

[2] S. C. Hillmer and G. C. Tiao. An arima-model-based approach to seasonal adjustment. *J. Am. Stat. Assoc.*, 77:63–70, 1982.

[3] V. Gomez and A. Maravall. Automatic modelling methods for univariate series. In D. Pena, G. C. Tiao, and R. S. Tsay, editors, *A Course in Time Series Analysis*. John Wiley & Sons, New York, 2001.

[4] A. C. Harvey and J. Durbin. The effects of seat belt legislation on british road casualties: A case study in structural time series modelling. *J. Roy. Stat. Soc. A*, 149:187–227, 1986.

[5] S. Makridakis, S. C. Wheelwright, and R. J. Hyndman. *Forecasting: Methods and Applications, 3rd Ed.* John Wiley and Sons, New York, 1998.

[6] B. W. Silverman. Some aspects of the spline smoothing approach to non-parametric regression curve fitting. *J. Roy. Stat. Soc. B*, 47:1–52, 1985.

[7] S. J. Koopman, A. C. Harvey, J. A. Doornik, and N. Shephard. *STAMP 6.0: Structural Time Series Analyser, Modeller and Predictor*. Timberlake Consultants, London, 2000.

[8] J.A.D. Aston, D.F. Findley, T.S. McElroy, K.C. Wills, and D.E.K. Martin. New arima models for seasonal time series and their application to seasonal adjustment and forecasting. *Research Report Series, US Census Bureau*, 2007.

A Predefined Model Reference

The predefined models can be organized into two categories, observation disturbance models and normal models. The former contains only specification of the observation disturbance, and is used primarily for model combination; the latter contains all other models, and can in turn be partitioned into structural time series models, ARIMA type models, and other models.

The following is a list of each model and function name.

- **Observation disturbance models**
  - **Gaussian noise:** `ssm_gaussian([p, cov])` or `ssm_normal([p, cov])`
    `p` is the number of variables, default `1`.
    `cov` is true if they are correlated, default `true`.
  - **Null noise:** `ssm_null([p])`
    `p` is the number of variables, default `1`.
  - **Poisson error:** `ssm_poisson`
  - **Binary error:** `ssm_binary`
  - **Binomial error:** `ssm_binomial(k)`
    `k` is the number of trials, can be a scalar or row vector.
  - **Negative binomial error:** `ssm_negbinomial(k)`
    `k` is the number of trials, can be a scalar or row vector.
  - **Exponential error:** `ssm_exp`
  - **Multinomial error:** `ssm_multinomial(h, k)`
    `h` is the number of cells.
    `k` is the number of trials, can be a scalar or row vector.

  - **Exponential family error:** `ssm_expfamily(b, d2b, id2bdb, c)`

    $$p(y|\theta) = \exp\left(y^T\theta - b(\theta) + c(y)\right)$$

    `b` is the function $b(\theta)$.
    `d2b` is $\ddot{b}(\theta)$, the second derivative of $b(\theta)$.
    `id2bdb` is $\ddot{b}(\theta)^{-1}\dot{b}(\theta)$.
    `c` is the function $c(y)$.
  - **t-distribution noise:** `ssm_t([nu])`
    `nu` is the degree of freedom, will be estimated as model parameter if not specified.
  - **Gaussian mixture noise:** `ssm_mix`
  - **General error noise:** `ssm_err`

- **Structural time series models**
  - **Integrated random walk:** `ssm_irw(d)`
    `d` is the order of integration.
  - **Local polynomial trend:** `ssm_lpt(d)`
    `d` is the order of the polynomial.
  - **Local level model:** `ssm_llm`
  - **Local level trend:** `ssm_llt`
  - **Seasonal components:** `ssm_seasonal(type, s)`
    `type` can be `'dummy'`,`'dummy fixed'`,`'h&s'`,`'trig1'`,`'trig2'` or `'trig fixed'`.
    `s` is the seasonal period.
  - **Cycle component:** `ssm_cycle`
  - **Regression components:** `ssm_reg(x[, varname])`
  - **Dynamic regression components:** `ssm_dynreg(x[, varname])`
    `x` is a $m \times n$ matrix, m is the number of regression variables.
    `varname` is the name of the variables.
  - **Intervention components:** `ssm_intv(n, type, tau)`
    `n` is the total time duration.
    `type` can be `'step'`,`'pulse'`,`'slope'` or `'null'`.
    `tau` is the onset time.
  - **Constant components:** `ssm_const`
  - **Trading day variables:** `ssm_td(y, m, N, td6)`
    `'td6'` is set to true to use six variables.
  - **Length-of-month variables:** `ssm_lom(y, m, N)`
  - **Leap-year variables:** `ssm_ly(y, m, N)`
  - **Easter effect variables:** `ssm_ee(y, m, N, d)`
    `y` is the starting year.
    `m` is the starting month.
    `N` is the total number of months.
    `d` is the number of days before Easter.

- **Structural time series models:** `ssm_stsm(lvl, seas, s[, cycle, x])`
  `lvl` is `'level'` or `'trend'`.
  `seas` is the seasonal type (see seasonal components).
  `s` is the seasonal period.
  `cycle` is `true` if there's a cycle component in the model, default `false`.
  `x` is explanatory (regression) variables (see regression components).
- **Common levels models:** `ssm_commonlvls(p, A_ast, a_ast)`

$$\begin{aligned} y_t &= \begin{bmatrix} 0 \\ a^* \end{bmatrix} + \begin{bmatrix} I_r \\ A^* \end{bmatrix} \mu_t^* + \varepsilon_t \\ \mu_{t+1}^* &= \mu_t^* + \eta_t^* \end{aligned}$$

  `p` is the number of variables (length of y_t).
  `A_ast` is A^*, a $(p-r) \times r$ matrix.
  `a_ast` is a^*, a $(p-r) \times 1$ vector.
- **Multivariate local level models:** `ssm_mvllm(p[, cov])`
- **Multivariate local level trend:** `ssm_mvllt(p[, cov])`
- **Multivariate seasonal components:** `ssm_mvseasonal(p, cov, type, s)`
- **Multivariate cycle component:** `ssm_mvcycle(p[, cov])`
  `p` is the number of variables.
  `cov` is a logical vector that is true for each correlated disturbances, default all `true`.
  Other arguments are the same as the univariate versions.
- **Multivariate regression components:** `ssm_mvreg(p, x[, dep])`
  `dep` is a $p \times m$ logical matrix that specifies dependence of data variables on regression variables.
- **Multivariate intervention components:** `ssm_mvintv(p, n, type, tau)`
- **Multivariate structural time series models:** `ssm_mvstsm(p, cov, lvl, seas, s[, cycle, x, dep])`
  `cov` is a logical vector that specifies the covariance structure of each component in turn.

- **ARIMA type models**
  - **ARMA models:** `ssm_arma(p, q, mean)`
  - **ARIMA models:** `ssm_arima(p, d, q, mean)`
  - **Multiplicative seasonal ARIMA models:** `ssm_sarima(p, d, q, P, D, Q, s, mean)`
  - **Seasonal sum ARMA models:** `ssm_sumarma(p, q, D, s, mean)`
  - **SARIMA with Hillmer-Tiao decomposition:** `ssm_sarimahtd(p, d, q, P, D, Q, s, gauss)`
    `p` is the order of AR.
    `d` is the order of I.
    `q` is the order of MA.
    `P` is the order of seasonal AR.
    `D` is the order of seasonal I.
    `Q` is the order of seasonal MA.
    `s` is the seasonal period.
    `mean` is true if the model has mean.
    `gauss` is true if the irregular component is Gaussian.

  - **Airline models:** `ssm_airline([s])`
    `s` is the period, default 12.
  - **Frequency specific SARIMA models:** `ssm_freqspec(p, d, q, P, D, nparam, nfreq, freq)`
  - **Frequency specific airline models:** `ssm_genair(nparam, nfreq, freq)`
    `nparam` is the number of parameters, `3` or `4`.
    `nfreq` is the size of the largest subset of frequencies sharing the same parameter.
    `freq` is an array containing the members of the smallest subset.
  - **ARIMA component models:** `ssm_arimacom(d, D, s, phi, theta, ksivar)`
    The arguments match those of the function `htd`, see its description for details.

- **Other models**
  - **Cubic spline smoothing (continuous time):** `ssm_spline(delta)`
    `delta` is the time duration of each data point.
  - **1/f noise models (approximated by AR):** `ssm_oneoverf(m)`
    `m` is the order of the approximating AR process.

B Class Reference

B.1 SSMAT

The class `SSMAT` represents state space matrices, can be stationary or dynamic. `SSMAT` can be horizontally, vertically, and block diagonally concatenated with each other, as well as with ordinary matrices. Parenthesis reference can also be used, where two or less indices indicate the stationary part, and three indices indicates a 3-d matrix with time as the third dimension.

B.1.1 Subscripted reference and assignment

`.mat`	RA[2]	Stationary part of `SSMAT`	Assignment cannot change size.
`.mmask`	R	Variable mask	A logical matrix the same size as `mat`.
`.n`	R	Time duration	Is equal to `size(dvec, 2)`.
`.dmmask`	R	Dynamic mask	A logical matrix the same size as `mat`.
`.d`	R	Number of dynamic elements	Is equal to `size(dvec, 1)`.
`.dvec`	RA	Dynamic vector sequence	A `d` $\times$ `n` matrix. Assignment cannot change `d`.
`.dvmask`	R	Dynamic vector variable mask	A `d` $\times$ 1 logical vector.
`.const`	R	True if `SSMAT` is constant	A logical scalar.
`.sta`	R	True if `SSMAT` is stationary	A logical scalar.
`(i, j)`	RA	Stationary part of `SSMAT`	Assignment cannot change size.
`(i, j, t)`	RA	`SSMAT` as a 3-d matrix	Assignment cannot change size.

B.1.2 Class methods

- **`ssmat`**
  `SSMAT` constructor.
  `SSMAT(m[, mmask])` creates a `SSMAT`. `m` can be a 2-d or 3-d matrix, where the third dimension is time.

[2] R - reference, A - assignment.

`mmask` is a 2-d logical matrix masking the variable part of matrix, which can be omitted or set to `[]` for constant `SSMAT`.
`SSMAT(m, mmask, dmmask[, dvec, dvmask])` creates a dynamic `SSMAT`. `m` is a 2-d matrix forming the stationary part of the `SSMAT`. `mmask` is a logical matrix masking the variable part of matrix, which can be set to `[]` for constant `SSMAT`. `dmmask` is a logical matrix masking the dynamic part of `SSMAT`. `dvec` is a `nnz(dmmask)` $\times$ `n` matrix, `dvec(:, t)` is the values for the dynamic part at time `t` (`m(dmmask) = dvec(:, t)` for time `t`), default is a `nnz(dmmask)` $\times$ 1 zero matrix. `dvmask` is a `nnz(dmmask)` $\times$ 1 logical vector masking the variable part of `dvec`.

- **isconst**
  `ISCONST(·`[3]`)` returns true if the stationary part of `SSMAT` is constant.

- **isdconst**
  `ISDCONST(·)` returns true if the dynamic part of `SSMAT` is constant.

- **issta**
  `ISSTA(·)` returns true if `SSMAT` is stationary.

- **size**
  `[ ... ] = SIZE(·, ...)` is equivalent to `[ ... ] = SIZE(·.mat, ...)`.

- **getmat**
  `GETMAT(·)` returns `SSMAT` as a matrix if stationary, or as a cell array of matrices if dynamic.

- **getdvec**
  `GETDVEC(·, mmask)` returns the dynamic elements specified in matrix mask `mmask` as a vector sequence.

- **getn**
  `GETN(·)` returns the time duration of `SSMAT`.

- **setmat**
  `SETMAT(·, vec)` updates the variable stationary matrix, `vec` must have size `nnz(mmask)` $\times$ 1.

- **setdvec**
  `SETDVEC(·, dsubvec)` updates the variable dynamic vector sequence, `dsubvec` must have `nnz(dvmask)` rows.

- **plus**
  `SSMAT` addition.

B.2 SSDIST

The class `SSDIST` represents state space distributions, a set of non-Gaussian distributions that governs a disturbance vector (which can also have Gaussian elements), and is a child class of `SSMAT`. `SSDIST` are constrained to be square matrices, hence can only be block diagonally concatenated, any other types of concatenation will ignore the non-Gaussian distribution part. Predefined non-Gaussian distributions can be constructed via the `SSDIST` constructor.

[3] An instance of the class in question.

B.2.1 Subscripted reference and assignment

`.nd`	R	Number of distributions	A scalar.
`.type`	R	Type of each distribution	A 0-1 row vector.
`.matf`	R	Approximation functions	A cell array of functions.
`.logpf`	R	Log probability functions	A cell array of functions.
`.diagmask`	R	Non-Gaussian masks	A `size(mat, 1)` $\times$ `nd` logical matrix.
`.dmask`	R	Variable mask for the distributions	A 1 $\times$ `nd` logical vector.
`.const`	R	True if `SSDIST` is constant	A logical scalar.

B.2.2 Class methods

- **`ssdist`**
  `SSDIST` constructor.
  `SSDIST(type[, matf, logpf, n, p])` creates a single univariate non-Gaussian distribution. `type` is 0 for exponential family distributions and 1 for additive noise distributions. `matf` is the function that generates the approximated Gaussian variance matrix given observations and signal or disturbances. `logpf` is the function that calculates the log probability of observations given observation and signal or disturbances. If the last two arguments are omitted, then the `SSDIST` is assumed to be variable. `SSDIST` with multiple non-Gaussian distributions can be formed by constructing a `SSDIST` for each and then block diagonally concatenating them.

- **`isdistconst`**
  `ISDISTCONST(·)` returns true if the non-Gaussian part of `SSDIST` is constant.

- **`setdist`**
  `SETDIST(·, distf)` updates the non-Gaussian distributions. `distf` is a cell matrix of functions.

- **`setgauss`**
  `SETGAUSS(·, eps)` or
  `[· ytilde] = SETGAUSS(·, y, theta)` calculates the approximating Gaussian covariance, the first form is used if there are no exponential family distributions, and the data `y` need to be transformed into `ytilde` in the second form if there are.

- **`logprobrat`**
  `LOGPROBRAT(·, N, eps)` or
  `LOGPROBRAT(·, N, y, theta, eps)` calculates the log probability ratio between the original non-Gaussian distribution and the approximating Gaussian distribution of the observations, it is used when calculating the non-Gaussian loglikelihood by importance sampling. The first form is used if there are no exponential family distributions. `N` is the number of importance samples.

B.2.3 Predefined non-Gaussian distributions

- **Poisson distribution:** `dist_poisson`
- **Binary distribution:** `dist_binary`
- **Binomial distribution:** `dist_binomial(k)`
- **Negative binomial distribution:** `dist_negbinomial(k)`
  `k` is the number of trials, can be a scalar or row vector.

- **Exponential distribution:** `dist_exp`

- **Multinomial distribution:** `dist_multinomial(h, k)`
  `h` is the number of cells.
  `k` is the number of trials, can be a scalar or row vector.

- **General exponential family distribution:** `dist_expfamily(b, d2b, id2bdb, c)`

$$p(y|\theta) = \exp\left(y^T\theta - b(\theta) + c(y)\right)$$

  `b` is the function $b(\theta)$.
  `d2b` is $\ddot{b}(\theta)$, the second derivative of $b(\theta)$.
  `id2bdb` is $\ddot{b}(\theta)^{-1}\dot{b}(\theta)$.
  `c` is the function $c(y)$.

B.3 SSFUNC

The class `SSFUNC` represents state space functions, a set of nonlinear functions that describe state vector transformations, and is a child class of `SSMAT`. Since linear transformations in SSM are represented by left multiplication of matrices, it is only possible for nonlinear functions (and hence it's matrix approximation) to be either block diagonally or horizontally concatenated. The former concatenation method produces multiple resulting output vectors, whereas the latter produces the sum of the output vectors of all functions as output. Vertical concatenation will ignore the nonlinear part of `SSFUNC`.

B.3.1 Subscripted reference and assignment

`.nf`	R	Number of functions	A scalar.
`.f`	R	The nonlinear functions	A cell array of functions.
`.df`	R	Derivative of each function	A cell array of functions.
`.horzmask`	R	Nonlinear function output masks	A `size(mat, 1)` $\times$ `nf` logical matrix.
`.vertmask`	R	Nonlinear function input masks	A `size(mat, 2)` $\times$ `nf` logical matrix.
`.fmask`	R	Variable mask for the functions	A 1 $\times$ `nf` logical vector.
`.const`	R	True if `SSFUNC` is constant	A logical scalar.

B.3.2 Class methods

- **`ssfunc`**
  `SSFUNC` constructor.
  `SSFUNC(p, m[, f, df])` creates a single nonlinear function. `p` is the input vector dimension and `m` is the output vector dimension. Hence the approximating linear matrix will be `p` $\times$ `m`. `f` and `df` is the nonlinear function and its derivative. `f` maps `m` $\times$ 1 vectors and time `t` to `p` $\times$ 1 vectors, and `df` maps `m` $\times$ 1 vectors and time `t` to `p` $\times$ `m` matrices. If `f` and `df` are not both provided, the `SSFUNC` is assumed to be variable. `SSFUNC` with multiple functions can be constructed by defining individual `SSFUNC` for each function, and concatenating them.

- **`isfconst`**
  `ISFCONST(·)` returns true if the nonlinear part of `SSFUNC` is constant.

- **getfunc**
  `GETFUNC(·, x, t)` returns `f(x, t)`, the transform of `x` at time `t`.

- **getvertmask**
  `GETVERTMASK(·)` returns `vertmask`.

- **setfunc**
  `SETFUNC(·, funcf)` updates the nonlinear functions. `funcf` is a cell matrix of functions.

- **setlinear**
  `[· c] = SETLINEAR(·, alpha)` calculates the approximating linear matrix, `c` is an additive constant in the approximation.

B.4 SSPARAM

The class `SSPARAM` represents state space model parameters. `SSPARAM` can be seen as a row vector of parameter values, hence they are combined by horizontal concatenation with a slight difference, as described below.

B.4.1 Subscripted reference and assignment

`.w`	R	Number of parameters	A scalar.
`.name`	R	Parameter names	A cell array of strings.
`.value`	RA	Transformed parameter values	A $1 \times$ `w` vector. Assignment cannot change size.

B.4.2 Class methods

- **ssparam**
  `SSPARAM` constructor.
  `SSPARAM(w[, transform, group])` creates a `SSPARAM` with `w` parameters.
  `SSPARAM(name[, transform, group])` creates a `SSPARAM` with parameter names `name`, which is a cell array of strings.
  `transform` is a cell array of strings describing transforms used for each parameter.
  `group` is a row vector that specifies the number of parameters included in each parameter subgroup, and `transform` is of the same length which specifies transforms for each group.

- **get**
  `GET(·)` returns the untransformed parameter values.

- **set**
  `SET(·, value)` sets the untransformed parameter values.

- **remove**
  `REMOVE(·, mask)` removes the parameters specified by `mask`.

- **horzcat**
  `[· pmask] = HORZCAT(·1, ·2, ...)` combines multiple `SSPARAM` into one, and optionally generates a cell array of parameter masks `pmask`, masking each original parameter sets with respect to the new combined parameter set.

B.5 SSMODEL

The class SSMODEL represents state space models and embeds all the previous classes. SSMODEL can be combined by horizontal or block diagonal concatenation. The former is the basis for additive model combination, where the observation is the sum of individual signals. The latter is an independent combination in which the observation is the vertical concatenation of the individual signals, which obviously needs to be further combined or changed to be useful. Only SSMODEL objects are used directly in data analysis functions, knowledge of embedded objects are needed only when defining custom models. For a list of predefined models see appendix A.

B.5.1 Subscripted reference and assignment

.name	RA	Model name	A string.
.Hinfo	R	Observation disturbance info	A structure.
.info	R	Model component info	A cell array of structures.
.p	R	Model observation dimension	A scalar.
.m	R	Model state dimension	A scalar.
.r	R	Model state disturbance dimension	A scalar.
.n	R	Model time duration	A scalar. Value is 1 for stationary models.
.H	R	Observation disturbance	A SSMAT or SSDIST.
.Z	R	State to observation transform	A SSMAT or SSFUNC.
.T	R	State update transform	A SSMAT or SSFUNC.
.R	R	Disturbance to state transform	A SSMAT.
.Q	R	State disturbance	A SSMAT or SSDIST.
.c	R	State update constant	A SSMAT.
.a1	RA	Initial state mean	A stationary SSMAT, can assign matrices.
.P1	RA	Initial state variance	A stationary SSMAT, can assign matrices.
.sta	R	True for stationary SSMODEL	A logical scalar.
.linear	R	True for linear SSMODEL	A logical scalar.
.gauss	R	True for Gaussian SSMODEL	A logical scalar.
.w	R	Number of parameters	A scalar.
.paramname	R	Parameter names	A cell array of strings.
.param	RA	Parameter values	A $1 \times$ w vector. Assignment cannot change w.
.psi	RA	Transformed parameter values	A $1 \times$ w vector. Assignment cannot change w.
.q	R	Number of initial diffuse elements	A scalar.

B.5.2 Class Methods

- **ssmodel**

  SSMODEL constructor.

  SSMODEL(info, H, Z, T, R, Q) creates a constant state space model with complete diffuse initialization and c $= 0$. SSMODEL(info, H, Z, T, R, Q, A, func, grad, psi[, pmask, P1, a1, c]) create a state space model. info is a string or a free form structure with field 'type' describing the model component. H is a SSMAT or SSDIST. Z is a SSMAT or SSFUNC. T is a SSMAT or SSFUNC. R is a SSMAT. Q is a SSMAT or SSDIST. A is a cell array of strings or a single string, each corresponding to func, a cell array of functions or a single function. Each string of A is a concatenation of some of 'H', 'Z', 'T', 'R', 'Q', 'c', 'a1', 'P1', 'Hd', 'Zd', 'Td', 'Rd', 'Qd', 'cd', 'Hng', 'Qng', 'Znl', 'Tnl',

representing each part of each element of SSMODEL. For example, 'Qng' specifies the non-Gaussian part of Q, and 'Zd' specifies the dynamic part of Z. A complicated example of a string in A is

```
['H' repmat('ng', 1,  gauss) repmat('T', 1, p+P>0) 'RQP1'],
```

which is the adjacency string for a possibly non-Gaussian ARIMA components model. func is a cell array of functions that updates model matrices. If multiple parts of the model are updated the output for each part must be ordered as for the adjacency matrix, but parts not updated can be skipped, for example, a function that updates H, T and Q must output [Hvec Tvec Qvec] in this order. grad is the derivative of func, each of which can be [] if not differentiable. For the example function above the gradient function would output [Hvec Tvec Qvec Hgrad Tgrad Qgrad] in order. psi is a SSPARAM. pmask is a cell array of parameter masks for the corresponding functions, can be omitted or set to [] if there's only one update function (which presumably uses all parameters). P1 and a1 are both stationary SSMAT. c is a SSMAT. All arguments expecting SSMAT can also take numeric matrices.

- **isgauss**
  ISGAUSS(·) is true for Gaussian SSMODEL.

- **islinear**
  ISLINEAR(·) is true for linear SSMODEL.

- **issta**
  ISSTA(·) is true for stationary SSMODEL. Note that all SSFUNC are implicitly assumed to be dynamic, but are treated as stationary by this and similar functions.

- **set**
  SET(·, A, M, func, grad, psi, pmask) changes one specific element of SSMODEL. A is an adjacency string, specifying which elements the function or functions func updates. A is a adjacency string as described earlier, corresponding to func. M should be the new value for one of H, Z, T, R, Q, c, a1 or P1, and A is constrained to contain only references to that element. grad is the same type as func and is its gradient, set individual functions to 0 if gradient does not exist or needed. psi is a SSPARAM that stores all the parameters used by functions in func, and pmask is the parameter mask for each function.

- **setparam**
  SETPARAM(·, psi, transformed) sets the parameters for SSMODEL. If transformed is true, psi is the new value for the transformed parameters, else psi is the new value for the untransformed parameters.

C Function Reference

C.1 User-defined update functions

This section details the format of update functions for various parts of classes SSMAT, SSDIST and SSFUNC.

- **Stationary part**
  vec = func(psi) updates the stationary part of SSMAT. vec must be a column vector such that mat(mmask) = vec would correctly update the stationary matrix mat given parameters psi.

- **Dynamic part**
  `dsubvec = func(psi)` updates the dynamic part of `SSMAT`. `dsubvec` must be a matrix such that `dvec(dvmask, 1:size(dsubvec, 2)) = dsubvec` would correctly update the dynamic vector sequence `dvec` given parameters `psi`.

- **Non-Gaussian part**
  `distf = func(psi)` updates the non-Gaussian part of `SSDIST`. `distf` must be a cell matrix of function handles such that `[matfdmask] = distf:, 1` and `[logpfdmask] = distf:, 2` would correctly update the distributions given parameters `psi`.

- **Nonlinear part**
  `funcf = func(psi)` updates the nonlinear part of `SSFUNC`. `funcf` must be a cell matrix of function handles such that `[ffmask] = funcf:, 1` and `[dffmask] = funcf:, 2` would correctly update the nonlinear functions given parameters `psi`.

Note that an update function can return more than one of the four kind of output, thus updating multiple part of a `SSMODEL`, but the order of the output arguments must follow the convention: stationary part of `H`, `Z`, `T`, `R`, `Q`, `c`, `a1`, `P1`, dynamic part of `H`, `Z`, `T`, `R`, `Q`, `c`, non-Gaussian part of `H`, `Q`, nonlinear part of `Z`, `T`, with possible omissions.

C.2 Data analysis functions

Most functions in this section accepts analysis settings options, specified as option name and option value pairs (e.g. `('disp', 'off')`). These groups of arguments are specified at the end of each function that accepts them, and are represented by `opt` in this section.

- **`batchsmo`**
  `[alphahat epshat etahat] = BATCHSMO(y, model[, opt])` performs batch smoothing of multiple data sets. `y` is the data of dimension $\mathtt{p} \times \mathtt{n} \times \mathtt{N}$, where `n` is the data length and `N` is the number of data sets, there must be no missing values. `model` is a `SSMODEL`. The output is respectively the smoothed state, smoothed observation disturbance and smoothed state disturbance, each of dimensions $\mathtt{m} \times \mathtt{n} \times \mathtt{N}$, $\mathtt{p} \times \mathtt{n} \times \mathtt{N}$ and $\mathtt{r} \times \mathtt{n} \times \mathtt{N}$. This is equivalent to doing `fastsmo` on each data set.

- **`disturbsmo`**
  `[epshat etahat epsvarhat etavarhat] = DISTURBSMO(y, model[, opt])` performs disturbance smoothing. `y` is the data of dimension $\mathtt{p} \times \mathtt{n}$, and `model` is a `SSMODEL`. The output is respectively the smoothed observation disturbance ($\mathtt{p} \times \mathtt{n}$), smoothed state disturbance ($\mathtt{r} \times \mathtt{n}$), smoothed observation disturbance variance ($\mathtt{p} \times \mathtt{p} \times \mathtt{n}$ or $1 \times \mathtt{n}$ if $\mathtt{p} = 1$) and smoothed state disturbance variance ($\mathtt{r} \times \mathtt{r} \times \mathtt{n}$ or $1 \times \mathtt{n}$ if $\mathtt{r} = 1$).

- **`estimate`**
  `[model logL output] = ESTIMATE(y, model[, param0, alpha0, opt])` estimates the parameters of `model` starting from the initial parameter value `param0`. `y` is the data of dimension $\mathtt{p} \times \mathtt{n}$, and `model` is a `SSMODEL`. `param0` can be empty if the current parameter values of `model` is used as initial value, and a scalar `param0` sets all parameters to the same value. Alternatively `param0` can be a logical row vector specifying which parameters to estimate, or a $2 \times \mathtt{w}$ matrix with the first row as initial value and second row as estimated parameter mask. The initial state sequence estimate `alpha0` is needed only when `model` is non-Gaussian or nonlinear. `output` is a structure that contains optimization routine information, approximated observation sequence $\tilde{y}$ if non-Gaussian or nonlinear, and the AIC and BIC of the output `model`.

- **fastsmo**
  `[alphahat epshat etahat] = fastsmo(y, model[, opt])` performs fast smoothing. `y` is the data of dimension `p` × `n`, and `model` is a `SSMODEL`. The output is respectively the smoothed state (`m` × `n`), smoothed observation disturbance (`p` × `n`), and smoothed state disturbance (`r` × `n`).

- **gauss**
  `[model ytilde] = GAUSS(y, model[, alpha0, opt])` calculates the Gaussian approximation. `y` is the data of dimension `p` × `n`, and `model` is a `SSMODEL`. `alpha0` is the initial state sequence estimate and can be empty or omitted.

- **kalman**
  `[a P v F] = KALMAN(y, model[, opt])` performs Kalman filtering. `y` is the data of dimension `p` × `n`, and `model` is a `SSMODEL`. The output is respectively the filtered state (`m` × `n+1`), filtered state variance (`m` × `m` × `n+1`, or 1 × `n+1` if `m` = 1), one-step prediction error (innovation) (`p` × `n`), one-step prediction variance (`p` × `p` × `n`, or 1 × `n` if `p` = 1).

- **linear**
  `[model ytilde] = LINEAR(y, model[, alpha0, opt])` calculates the linear approximation. `y` is the data of dimension `p` × `n`, and `model` is a `SSMODEL`. `alpha0` is the initial state sequence estimate and can be empty or omitted.

- **loglik**
  `LOGLIK(y, model[, ytilde, opt])` returns the log likelihood of `model` given `y`. `y` is the data of dimension `p` × `n`, and `model` is a `SSMODEL`. `ytilde` is the approximating observation $\tilde{y}$ and is needed for some non-Gaussian or nonlinear models.

- **sample**
  `[y alpha eps eta] = SAMPLE(model, n[, N])` generates observation samples from `model`. `model` is a `SSMODEL`, `n` specifies the sampling data length, and `N` specifies how many sets of data to generate. `y` is the sampled data of dimension `p` × `n` × `N`, `alpha`, `eps`, `eta` are respectively the corresponding sampled state (`m` × `n` × `N`), observation disturbance (`p` × `n` × `N`), and state disturbance (`r` × `n` × `N`).

- **signal**
  `SIGNAL(alpha, model)` generates the signal for each component according to the state sequence and model specification. `alpha` is the state of dimension `m` × `n`, and `model` is a `SSMODEL`. The output is a cell array of data each with dimension `p` × `n`, or a `M` × `n` matrix where `M` is the number of components if `p` = 1.

- **simsmo**
  `[alphatilde epstilde etatilde] = SIMSMO(y, model, N[, antithetic, opt])` generates observation samples from `model` conditional on data `y`. `y` is the data of dimension `p` × `n`, and `model` is a `SSMODEL`. `antithetic` should be set to `1` if antithetic variables are used. The output is respectively the sampled state sequence (`m` × `n` × `N`), sampled observation disturbance (`p` × `n` × `N`), and sampled state disturbance (`r` × `n` × `N`).

- **statesmo**
  `[alphahat V] = STATESMO(y, model[, opt])` performs state smoothing. `y` is the data of dimension `p` × `n`, and `model` is a `SSMODEL`. The output is respectively the smoothed state (`m` × `n`), smoothed state variance (`m` × `m` × `n`, or 1 × `n` if `p` = 1). If only the first output argument is specified, fast state smoothing is automatically performed instead.

- **oosforecast**
  `[yf err SS] = OOSFORECAST(y, model, n1, h)` performs out-of-sample forecast. `y` is the data of dimension `p` × `n`, `model` is a `SSMODEL`, `n1` is the number of time points to exclude at the end, and `h` is the number of steps ahead to forecast, which can be an array. The output `yf` is the forecast obtained, `err` is the forecast error, and `SS` is the forecast error cumulative sum of squares.

C.3 Stock element functions

- **fun_arma**
  `[fun grad param] = FUN_ARMA(p, q)` creates matrix update functions for ARMA models.
- **fun_cycle**
  `[fun grad param] = FUN_CYCLE()` creates matrix update functions for cycle components.
- **fun_dupvar**
  `[fun grad param] = FUN_DUPVAR(p, cov, d[, name])` creates matrix update functions for duplicated variances.
- **fun_genair**
  `[fun grad param] = FUN_GENAIR(nparam, nfreq, freq)` creates matrix update functions for generalized airline models.
- **fun_homovar**
  `[fun grad param] = FUN_HOMOVAR(p, cov, q[, name])` creates matrix update functions for homogeneous variances.
- **fun_interlvar**
  `[fun grad param] = FUN_INTERLVAR(p, q, cov[, name])` creates matrix update functions for q-interleaved variances.
- **fun_mvcycle**
  `[fun grad param] = FUN_MVCYCLE(p)` creates matrix update functions for multivariate cycle component.
- **fun_oneoverf**
  `[fun grad param] = FUN_ONEOVERF(m)` creates matrix update functions for 1/f noise.
- **fun_sarimahtd**
  `[fun grad param] = FUN_SARIMAHTD(p, d, q, P, D, Q, s, gauss)` creates matrix update functions for SARIMA with Hillmer-Tiao decomposition and non-Gaussian irregular.
- **fun_sarma**
  `[fun grad param] = FUN_SARMA(p, q, P, Q, s)` creates matrix update functions for seasonal ARMA models.
- **fun_spline**
  `[fun grad param] = FUN_SPLINE(delta)` creates matrix update functions for the cubic spline models.
- **fun_t**
  `[fun grad param] = FUN_T([nu])` creates update functions for the t-distribution.

- **fun_var**
  `[fun grad param] = FUN_VAR(p, cov[, name])` creates matrix update functions for variances.

- **fun_wvar**
  `[fun grad param] = FUN_WVAR(p, s[, name])` creates matrix update functions for W structure variances.

- **mat_arima**
  `[Z T R P1 Tmmask Rmmask P1mmask] = MAT_ARIMA(p, d, q, mean)` creates base matrices for ARIMA models.

- **mat_arma**
  `[Z T R P1 Tmmask Rmmask P1mmask] = MAT_ARMA(p, q, mean)` or
  `[T R P1 Tmmask Rmmask P1mmask] = MAT_ARMA(p, q, mean)` creates base matrices for ARMA models.

- **mat_commonlvls**
  `[Z T R a1 P1] = MAT_COMMONLVLS(p, A, a)` creates base matrices for common levels models.

- **mat_dummy**
  `[Z T R] = MAT_DUMMY(s[, fixed])` creates base matrices for dummy seasonal component.

- **mat_dupvar**
  `[m mmask] = MAT_DUPVAR(p, cov, d)` creates base matrices for duplicated variances.

- **mat_homovar**
  `[H Q Hmmask Qmmask] = MAT_HOMOVAR(p, cov, q)` creates base matrices for homogeneous variances.

- **mat_hs**
  `[Z T R] = MAT_HS(s)` creates base matrices for Harrison and Stevens seasonal component.

- **mat_interlvar**
  `[m mmask] = MAT_INTERLVAR(p, q, cov)` creates base matrices for q-interleaved variances.

- **mat_lpt**
  `[Z T R] = MAT_LPT(d, stochastic)` creates base matrices for local polynomial trend or integrated random walk models.

- **mat_mvcycle**
  `[Z T Tmmask R] = MAT_MVCYCLE(p)` creates base matrices for multivariate cycle components.

- **mat_mvdummy**
  `[Z T R] = MAT_MVDUMMY(p, s[, fixed])` creates base matrices for multivariate dummy seasonal components.

- **mat_mvhs**
  `[Z T R] = MAT_MVHS(p, s)` creates base matrices for multivariate Harrison and Stevens seasonal components.

- **mat_mvintv**
  `[Z Zdmmask Zdvec T R] = MAT_MVINTV(p, n, type, tau)` creates base matrices for multivariate intervention components.

- **mat_mvllt**
  `[Z T R] = MAT_MVLLT(p)` creates base matrices for multivariate local level trend models.

- **mat_mvreg**
  `[Z Zdmmask Zdvec T R] = MAT_MVREG(p, x, dep)` creates base matrices for multivariate regression components.

- **mat_mvtrig**
  `[Z T R] = MAT_MVTRIG(p, s[, fixed])` creates base matrices for multivariate trigonometric seasonal components.

- **mat_reg**
  `[Z Zdmmask Zdvec T R] = MAT_REG(x[, dyn])` creates base matrices for regression components.

- **mat_sarima**
  `[Z T R P1 Tmmask Rmmask P1mmask] = MAT_SARIMA(p, d, q, P, D, Q, s, mean)` or `[Z T R P1 Rmmask P1mmask] = MAT_SARIMA(p, d, q, P, D, Q, s, mean)` creates base matrices for SARIMA models.

- **mat_sarimahtd**
  `[Z T R Qmat P1 Tmmask Rmmask Qmmask P1mmask] = MAT_SARIMAHTD(p, d, q, P, D, Q, s)` creates base matrices for SARIMA with Hillmer-Tiao decomposition.

- **mat_spline**
  `[Z T Tdmmask Tdvec R] = MAT_(delta)` creates base matrices for the cubic spline model.

- **mat_sumarma**
  `[Z T R P1 Tmmask Rmmask P1mmask] = MAT_SUMARMA(p, q, D, s, mean)` creates base matrices for sum integrated ARMA models.

- **mat_trig**
  `[Z T R] = MAT_TRIG(s[, fixed])` creates base matrices for trigonometric seasonal components.

- **mat_var**
  `[m mmask] = MAT_VAR(p, cov)` creates base matrices for variances.

- **mat_wvar**
  `[m mmask] = MAT_WVAR(p, s)` creates base matrices for W structure variances.

- **x_ee**
  `X_EE(Y, M, d)` generates Easter effect variable.

- **x_intv**
  `X_INTV(n, type, tau)` generates intervention variables.

- **x_ly**
  `X_LY(Y, M)` generates leap-year variable.

- **x_td**
  `X_TD(Y, M, td6)` generates trading day variables.

C.4 Miscellaneous helper functions

- **setopt**
  `SETOPT(optname1, optvalue1, optname2, optvalue2, ...)` sets the global analysis settings and returns a structure containing the new settings, thus a call with no arguments just returns the current settings. The available options are: `'disp'` can be set to `'off'`, `'notify'`, `'final'` or `'iter'`. `tol` is the tolerance. `fmin` is the minimization algorithm to use (`'bfgs'` or `'simplex'`). `maxiter` is the maximum number of iterations. `nsamp` is the number of samples to use in simulation. Note that the same form of argument sequence can be used at the end of most data analysis functions to specify one time overrides.

- **ymarray**
  `[Y M] = YMARRAY(y, m, N)` generates an array of years and months from specified starting year `y`, starting month `m`, and number of months `N`.

C.5 TRAMO model selection and Hillmer-Tiao decomposition

- **arimaselect**
  `[p d q mean] = ARIMASELECT(y)` or
  `[p d q P D Q mean] = ARIMASELECT(y, s)` performs TRAMO model selection. `y` is the data of dimension `p` × `n`, and `s` is the seasonal period. The first form selects among ARIMA models with or without a mean. The second form selects among SARIMA models with or without a mean.

- **armadegree**
  `[p q] = ARMADEGREE(y[, mean, mr])` or
  `[p q P Q] = ARMADEGREE(y, s[, mean, mr, ms])` determines the degrees of ARMA or SARMA from observation `y`. `y` is the data of dimension `p` × `n`, and `s` is the seasonal period. `mean` is true if models with mean are used for the selection. `mr` and `ms` is the largest degree to consider for regular ARMA and seasonal ARMA respectively.

- **diffdegree**
  `[d mean] = DIFFDEGREE(y[, ub, tsig])` or
  `[d D mean] = DIFFDEGREE(y, s[, ub, tsig])` performs unit root detection. The first form only considers regular differencing. `y` is the data of dimension `p` × `n`, `s` is the seasonal period, `ub` is the threshold for unit roots, default is `[0.97 0.88]`, and `tsig` is the t-value threshold for mean detection, default is `1.5`.

- **loglevel**
  `LOGLEVEL(y, s)` or
  `LOGLEVEL(y, p, q)` or
  `LOGLEVEL(y, p, d, q[, P, D, Q, s])` determines the log level specification with respect to various models. `y` is the observation data, `s` is the seasonal period. The output is `1` if no logs are needed, else `0`. The first form uses the airline model with specified seasonal period, second form uses ARMA models, and the third form uses SARIMA models.

- **htd**
  `[theta ksivar] = HTD(d, D, s, Phi, Theta[, etavar])` performs Hillmer-Tiao decomposition. `d` is the order of regular differencing, `D` is the order of seasonal differencing, `s` is the seasonal period, `Phi` is a cell array of increasing order polynomials representing autoregressive factors for each component, `Theta` is an increasing order polynomial representing the moving average factor, `etavar` is the disturbance variance. The output `theta` is a cell array of decomposed moving average factors for each component, and `ksivar` is the corresponding disturbance variance for each component (including the irregular variance at the end).

- **ssmhtd**
  `SSMHTD(model)` performs Hillmer-Tiao decomposition on `SSMODEL model` and returns an ARIMA component model. This is a wrapper for the function `htd`.