StateSpaceModels.jl: a Julia Package for Time-Series Analysis in a State-Space Framework

Raphael Saavedra[1, 2], Guilherme Bodin[1, 2], and Mario Souto[1, 2]

[1]Laboratory of Applied Mathematical Programming and Statistics (LAMPS)
[2]Pontifical Catholic University of Rio de Janeiro (PUC-Rio), Brazil

ABSTRACT

`StateSpaceModels.jl` is an open-source Julia package for modeling, forecasting and simulating time series in a state-space framework. The package represents a straightforward tool that can be useful for a wide range of applications that deal with time series. In addition, it contains features that are not present in related commercial software, such as Monte Carlo simulation and the possibility of setting any user-defined linear model.

Keywords

State-space models, time-series analysis, Kalman filter, forecasting, Monte Carlo simulation

1. Motivation

State-space modeling is a classical framework in control engineering that represents a system through the definition of input, state, and output variables [18]. Input variables are external entities that are inserted into the system and can serve as control inputs or noise. State variables represent unobserved components that evolve through time following a given state equation and also depending on the values of the input variables. Finally, output variables result from the realization of the state plus noise factors and represent the observable outcome of the system. In general, state-space models make use of the Kalman filter [8, 17] to obtain predictive estimates for the state.

Due to its comprehensive form and wide range of potential applications, state-space models found a niche in time-series modeling, forecasting, and simulation, representing a flexible framework for time-series analysis with time-varying parameters [5]. The ability to conveniently define the evolution of a time series allows the characterization of stochastic components, such as trend and seasonality, that are non-trivial to model in other frameworks.

There are several packages focused on state-space models for time-series analysis in other languages, such as `KFAS` in R [6] and `Statsmodels` in Python [16], as well as commercial software such as `STAMP` [9]. Among related Julia packages, we highlight:

— `Kalman.jl` [15], which implements Kalman filtering and smoothing.

— `StateSpace.jl` [1], a control-oriented package with several versions of the Kalman filter.

— `ControlSystems.jl` [2], a package aimed at control applications with numerous features, including state-space modeling.

The contributions brought by these packages notwithstanding, there are currently no state-space model packages aimed at time-series analysis in Julia. Thus, the main objectives of `StateSpaceModels.jl` [14] are:

(1) to fill this gap by implementing a general and intuitive framework in Julia for modeling, estimating, forecasting and simulating time series with state-space models;

(2) to provide an open-source package that is capable of performing the same functions as related commercial software such as `STAMP` and more;

(3) to develop a package for time-series analysis with state-space models that is fully implemented in Julia, contrary to related packages in other languages that depend on C or Fortran routines [6].

The remainder of this paper is organized as follows. In Section 2, the Gaussian state-space framework is introduced. In Section 3, the procedures for model specification are presented. Section 4 explains the filtering, estimation and smoothing processes. Section 5 discusses the available diagnostics to evaluate the model specification and estimation. In Section 6, the forecasting and simulation procedures are presented. Section 7 contains examples of applications that illustrate the use of the package. Finally, conclusions and future work are provided in Section 8.

2. Gaussian state-space framework

For the sake of consistency and readability, we utilize in the package the same notation as [5]. Suppose we have a series of observations $y_1, \ldots, y_n$. In a state-space framework, it is assumed that the observations depend on a set of unobserved components denominated states, or simply the state, and denoted by $\alpha_1, \ldots, \alpha_n$. The state components often have physical interpretations, such as trend and seasonality in a time-series context or position and speed in a control setting.

The main idea behind state-space modeling is to define the evolution of the state and its relation with the observed variables. This is done through the following two equations, denoted the observation equation and the state equation, respectively:

$$y_t = Z_t \alpha_t + \varepsilon_t, \qquad \varepsilon_t \sim N(0, H_t), \tag{1}$$

$$\alpha_{t+1} = T_t \alpha_t + R_t \eta_t, \quad \eta_t \sim N(0, Q_t), \tag{2}$$

where y_t is a $p \times 1$ vector of observations and α_t is an unobserved $m \times 1$ vector representing the state at instant t. Note that the behavior of the system over time is determined by α_t as defined in Eq. (2), but α_t cannot be directly observed, contrary to y_t.

Matrices Z_t, T_t, and R_t are the ones that define how the observations relate to the state and how the state evolves over time, and are generally assumed to be known. The error terms ε_t and η_t are supposed to be serially independent and independent of each other. Finally, H_t and Q_t are the covariance matrices of the error terms. In this package, it is assumed that T, R, H, and Q are not time-varying, as this is the case for the vast majority of practical applications. Conversely, Z_t is allowed to vary over time.

Next, for illustration purposes, we present an example model and show how it can be inserted into the state-space framework. Consider the following model, called the linear trend model:

$$y_t = \mu_t + \varepsilon_t, \qquad \varepsilon_t \sim N(0, \sigma_\varepsilon^2), \qquad (3)$$
$$\mu_{t+1} = \mu_t + \nu_t + \xi_t, \quad \xi_t \sim N(0, \sigma_\xi^2), \qquad (4)$$
$$\nu_{t+1} = \nu_t + \zeta_t, \qquad \zeta_t \sim N(0, \sigma_\zeta^2), \qquad (5)$$

where μ_t represents the trend component and ν_t the slope component. Note that the we can rewrite this model in the framework of Eq. (1)–(2) in the following manner:

$$y_t = \begin{bmatrix} 1 & 0 \end{bmatrix} \alpha_t + \varepsilon_t, \qquad \varepsilon_t \sim N(0, \sigma_\varepsilon^2), \qquad (6)$$
$$\alpha_{t+1} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \alpha_t + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \eta_t, \quad \eta_t \sim N\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} \sigma_\xi^2 & 0 \\ 0 & \sigma_\zeta^2 \end{bmatrix} \right). \qquad (7)$$

Similarly, more complex and sophisticated models can also be modeled in the state-space framework.

3. Model specification

Model specification is the step wherein the user defines the state-space model they want to consider. This is done through the creation of a `StateSpaceModel` structure that contains the observations y_t as well as matrices Z_t, T, and R.

3.1 Predefined models

A set of predefined classical models is available, namely the local level model, the linear trend model, the basic structural model, and the structural model with exogenous variables. These models can be conveniently defined by calling a function and providing the observations y_t. Note that all models automatically extend to the multivariate case.

3.1.1 Local level model. The local level model consists of a stochastic level component that is defined by a random walk:

$$y_t = \mu_t + \varepsilon_t, \quad \varepsilon_t \sim N(0, \sigma_\varepsilon^2), \qquad (8)$$
$$\mu_{t+1} = \mu_t + \xi_t, \quad \xi_t \sim N(0, \sigma_\xi^2). \qquad (9)$$

A local level `StateSpaceModel` can be created using:

```
model = local_level(y)
```

where `y` is an observation vector in the univariate case or a matrix in column-wise fashion in the multivariate case, i.e., each column representing a variable and each line representing a time period. While very simple, the local level model can be successfully applied to a wide array of problems wherein one wants to estimate some magnitude that varies over time. For instance, in [9], an illustrative example utilizes water inflow data from the river Nile.

3.1.2 Linear trend model. The linear trend model consists of a local level model with the addition of a stochastic slope term, as seen in Eq. (3)–(5). Note that it differs from the local level model due to the inclusion of the slope ν_t in (4). A linear trend `StateSpaceModel` can be created using:

```
model = linear_trend(y)
```

3.1.3 Structural model. The basic structural model consists of three stochastic components: trend, slope, and seasonality, defined in the following manner:

$$y_t = \mu_t + \gamma_t + \varepsilon_t, \qquad \varepsilon_t \sim N(0, \sigma_\varepsilon^2), \qquad (10)$$
$$\mu_{t+1} = \mu_t + \nu_t + \xi_t, \qquad \xi_t \sim N(0, \sigma_\xi^2), \qquad (11)$$
$$\nu_{t+1} = \nu_t + \zeta_t, \qquad \zeta_t \sim N(0, \sigma_\zeta^2), \qquad (12)$$
$$\gamma_{t+1} = \sum_{j=1}^{s-1} \gamma_{t+1-j} + \omega_t, \quad \omega_t \sim N(0, \sigma_\omega^2). \qquad (13)$$

In this model, there is a stochastic trend that contains a slope component, exactly as the linear trend model. Furthermore, a stochastic seasonal component is modeled such that its sum over the seasonality period s is equal to zero except for a noise factor. A basic structural `StateSpaceModel` can be created using:

```
model = structural(y, s)
```

The idea of the seasonal component is to account for effects that have a periodic nature, such as monthly temperature over each year. Additionally, a structural model can contain exogenous variables, also known as explanatory variables. Exogenous variables represent external factors that are correlated to the phenomenon in question and can, thus, be used to improve the estimation and forecasting by increasing the available information. This model is identical to (10)–(13) except for the observation equation, which has an additional exogenous factor:

$$y_t = \mu_t + \gamma_t + \theta^\top X_t + \varepsilon_t, \quad \varepsilon_t \sim N(0, \sigma_\varepsilon^2) \qquad (14)$$

A structural model with exogenous variables can be created with:

```
model = structural(y, s; X = X)
```

where `X` is a matrix containing the exogenous variables observations in column-wise fashion.

3.2 User-defined models

Apart from the well-known predefined models presented in Section 3.1, the package allows the input of any user-defined linear model through the definition of matrices Z_t, T, and R through the following constructor:

```
model = StateSpaceModel{Float64}(y, Z, T, R)
```

In the simpler case where Z_t is constant, the argument `Z` can be input as an `Array{Float64, 2}`. Conversely, if Z_t is time-varying, it must be input as an `Array{Float64, 3}` where the third dimension represents the time periods. Any desired float precision can be employed within the model.

4. Filtering, estimation, and smoothing

Next, the model needs to be estimated. In this step, values for the covariance matrices H and Q, initially assumed to be unknown, are obtained via maximum likelihood estimation [4]. In order to do that, the Kalman filter needs to be utilized. Additionally, a smoother is applied in order to obtain the so-called smoothed estimates for the state.

After specifying the model, the estimation step can be called with the function `statespace`:

```
ss = statespace(model; verbose = 1)
```

In this step, the Kalman filtering, the maximum likelihood estimation of the fixed parameters, and the smoothing are conducted. The optional argument `verbose`, which defaults to 1, specifies the verbosity: 0 for no output, 1 for some progress information, 2 for progress information and summary of the optimization result and 3 for progress information as well as the full optimization log.

An example of the estimation log with default verbosity can be seen below.

```
julia> ss = statespace(model)
==============================================================
                   StateSpaceModels.jl v0.2.0
   (c) Raphael Saavedra, Guilherme Bodin, and Mario Souto, 2019
--------------------------------------------------------------
            Starting state-space model estimation.
     Initiating maximum likelihood estimation with 3 seeds.
--------------------------------------------------------------
            Seed 0 is aimed at degenerate cases.
--------------------------------------------------------------
||   seed   |    log-likelihood    |    time (s)   ||
||    0     |      -16217.4939     |      0.21     ||
||    1     |       -1350.4763     |      2.65     ||
||    2     |       -1350.4763     |      4.08     ||
||    3     |       -1350.4763     |      5.69     ||
--------------------------------------------------------------
            Maximum likelihood estimation complete.
                 Log-likelihood: -1350.4763
             End of state-space model estimation.
==============================================================
```

4.1 Filtering

The filtering step computes the predictive and filtered states as well as their covariance matrices at every time period via the Kalman filter. The predictive state and its variance are given by

$$a_{t+1} = \mathbb{E}[\alpha_{t+1}|Y_t], \tag{15}$$
$$P_{t+1} = \mathbb{V}[\alpha_{t+1}|Y_t], \tag{16}$$

while the filtered state and its variance are given by

$$a_{t|t} = \mathbb{E}[\alpha_t|Y_t], \tag{17}$$
$$P_{t|t} = \mathbb{V}[\alpha_t|Y_t], \tag{18}$$

where Y_t denotes the set of time periods $y_1, \ldots, y_t$.

Furthermore, the filter also computes the innovations, or prediction errors, as well as their covariance matrix at every time period:

$$v_t = y_t - Za_t, \tag{19}$$
$$F_t = \mathbb{V}[v_t]. \tag{20}$$

The package contains an implementation of the standard Kalman filter and the square-root Kalman filter [5]. The latter is a variant which utilizes Cholesky decomposition to ensure that the computed covariance matrices are positive semidefinite, thus avoiding numerical errors.

Additionally, in order to provide a flexible framework, we allow the utilization of any user-implemented variant of the Kalman filter through an abstract type called `AbstractFilter`:

```
struct MyKalmanFilter <: AbstractFilter
```

The choice of the filter type must be done in the estimation step as follows:

```
ss = statespace(model; filter_type = SquareRootFilter{Float64})
```

Filters allow any desired float precision. If no `filter_type` is provided, the standard type `KalmanFilter{Float64}` is utilized as a default option. Further information on Kalman filtering can be found in Chapter 4.3 of [5].

4.2 Estimation

The Kalman filter is responsible for deriving estimates for the predictive state and for its covariance matrix at each time period, but it is dependent on the values of the noise covariances H and Q. Thus, these constant parameters need to be estimated via maximum likelihood [4]. The log-likelihood function is given by

$$\ell(Y_n) = -\frac{np}{2} \log 2\pi - \frac{1}{2} \sum_{t=1}^{n} (\log |F_t| + v_t^\top F_t^{-1} v_t), \tag{21}$$

which depends on the innovations and their covariance matrix at each time period. Therefore, at each iteration of the optimization, the Kalman filter needs to be executed to compute the objective function.

The implemented optimization method `RandomSeedsLBFGS` generates random initial values for the parameters and then uses the L-BFGS algorithm [10] to maximize the log-likelihood function via the unconstrained optimization package `Optim.jl` [12].

Additionally, similar to the filtering step, users are able to define any desired optimization method through abstract type `AbstractOptimizationMethod`:

```
struct MyOptimizationMethod <: AbstractOptimizationMethod
```

As with the filter type, the choice of optimization method must be done in the estimation step:

```
ss = statespace(model; optimization_method = RandomSeedsLBFGS())
```

More advanced parameters, such as optimizer tolerances and the number of seeds, can be set within the `RandomSeedsLBFGS` constructor.

4.3 Smoothing

The smoothing step is conducted after estimation is complete. It consists in obtaining the so-called smoothed state and its covari-

ance matrix at each time period, i.e.,

$$\hat{\alpha}_t = \mathbb{E}[\alpha_t | Y_n], \quad (22)$$
$$V_t = \mathbb{V}[\alpha_t | Y_n]. \quad (23)$$

Similar to the filtering step, the smoothing is conducted through an iterative process. The procedure is done backwards, starting from the final time period n. The smoothed state estimates are useful for analyzing the behavior of the state components. Further information on state smoothing can be found on Chapter 4.4 of [5].

4.4 Missing observations

One interesting property of the Kalman filter and smoother is their ability to treat missing observations. Suppose we have a time series spanning time periods $t = 1, \ldots, n$, but for some periods $\tau, \ldots, \tau^*$ we have no observations available. This represents an obstacle for estimating a model in several frameworks. Nonetheless, the Kalman filter and smoother allow the derivation of minimum variance linear unbiased estimates for the missing observations so that a completion of the time series is possible. Conveniently, the only necessary modification in the filtering and smoothing equations is considering $Z_t = 0$ for the missing period.

In the package, this is implemented so that any `NaN` values in the observations are considered missing observations and automatically treated in the filtering and smoothing steps. In this way, the estimation is successfully conducted even when the time series of interest is incomplete.

Consider the following simulated time series composed by a growing trend plus a Gaussian noise. Additionally, we remove observations 10 through 20 to treat them as missing:

```
# Create growing trend series with Gaussian noise
y = collect(1:0.25:20) + 0.5*randn(77)
# Remove observations 10 through 20
y[10:20] .= NaN
# Specify the state-space model and estimate it
model = linear_trend(y)
ss = statespace(model)
```

The results of the estimation step are presented in Fig. 1. The predictive, filtered, and smoothed state estimates can be accessed in `ss.filter.a`, `ss.filter.att`, and `ss.smoother.alpha`, respectively. It is clear in Fig. 1 that the filter and smoother are able to effectively capture the growing linear trend of the series even in the period where observations are missing.

4.5 Estimation outputs

At the end of the filtering, estimation, and smoothing routine executed by the `statespace` function, a `StateSpace` structure is returned. It contains six fields:

— `model`: the previously defined `StateSpaceModel`.

— `filter`: a `FilterOutput` structure representing the result of the Kalman filter. It contains the predictive and filtered states as well as the innovations and their covariance matrices at each time period. Furthermore, there is a flag indicating if steady state was attained and the time period when it was attained.

— `smoother`: a `SmoothedState` structure representing the result of the smoothing procedure. It contains the smoothed state as well as its covariance matrix at each time period.

— `covariance`: a `StateSpaceCovariance` structure which contains the covariance matrices of the observation and state noises.
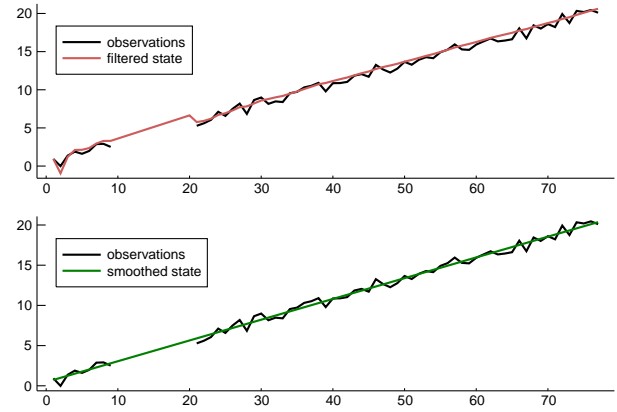

Fig. 1. Automatic completion of missing observations in the filtering and smoothing.

— `filter_type`: the defined filter type.

— `optimization_method`: the defined optimization method.

5. Diagnostics

As a way of evaluating the model specification and estimation, the package contains basic statistical diagnostics that can be conducted over the residuals. The available diagnostics are the Jarque-Bera normality test [7], the Ljung-Box independence test [3], and an homoscedasticity test [5]. The diagnostics can be run through function `diagnostics`.

An example of the diagnostics log can be seen below.

```
julia> diagnostics(ss)
============================================================
                    Running diagnostics...
              Jarque-Bera: p-value = 0.89683
                 Ljung-Box: p-value = 0.95855
          Homoscedasticity: p-value = 0.30752
============================================================
```

6. Forecasting and Simulation

Finally, after the model has been estimated, forecasting and Monte Carlo simulation can be conducted. In the case of forecasting, estimates for the future values of the time series are computed along with their probability distributions. Alternatively, it is possible to simulate an arbitrary number of future scenarios of the series, which is useful for a wide range of applications such as stochastic optimization.

The minimum mean square error forecasts for y_t can be obtained by treating future values of y_t as missing values. Forecasting is conducted with the function `forecast`, which receives a `StateSpace` structure coming from the estimation step and the number of time periods ahead to be forecast, and outputs the minimum square error forecasts and the predictive distributions of y_t at each time period:

```
pred, dist = forecast(ss, N)
```

Alternatively, another powerful tool is the simulation of future scenarios, since these are used as input by several applications, such

as certain stochastic optimization models. Performing Monte Carlo simulation in a time-series state-space framework involves sampling several scenarios for the observation and state errors utilizing the estimated variances H and Q at each time period, and then computing the state-space recursions for each scenario.

Monte Carlo simulation can be conducted with the use of the function `simulate`, which receives a `StateSpace` structure coming from the estimation step, the number of time periods ahead N, and the number of scenarios S to be simulated, and outputs an $N \times S$ matrix of scenarios for y_t:

```
scenarios = simulate(ss, N, S)
```

7. Applications

In this section, we present several practical applications which can be addressed via state-space modeling. We utilize `StateSpaceModels.jl` to tackle the presented problems and provide the results. Furthermore, for the sake of reproducibility, the code of some of the examples is in the folder `examples` in the `StateSpaceModels.jl` repository [14].

7.1 Airline passengers

As a first example, let us use the classical monthly airline passengers time series. In order to avoid multiplicative effects, we use the well-known approach of taking the log of the series [11]. Fig. 2 shows the log-airline passengers time series. We can estimate a structural `StateSpaceModel` as follows.

```
# Load the AirPassengers dataset
AP = CSV.read("AirPassengers.csv")
# Take the log of the series
logAP = log.(Vector{Float64}(AP[:Passengers]))
# Specify the state-space model
model = structural(logAP, 12)
# Estimate the state-space model
ss = statespace(model)
```

By estimating a structural model, we can analyze the individual components of the series, such as trend and seasonality. These components are presented in Fig. 3 and represent dimensions 1 and 3 of the smoothed state, respectively.

We can also forecast the following two years of the time series by using the `forecast` function. The result is displayed in Fig. 2.

```
# Number of months ahead to be forecast
N = 24
# Perform forecasting
pred, dist = forecast(ss, N)
```

7.2 Electricity consumption

Another practical example is the study of monthly electricity consumption in a given area. In this example, we will use real data from a Brazilian distribution company. In addition, we will consider a temperature series, which is highly correlated to electricity consumption in Brazil, as an exogenous variable. A similar study using `StateSpaceModels.jl` was conducted in [13].

In this case, our objective is to simulate a large set of future scenarios that can be used as inputs in a stochastic optimization problem
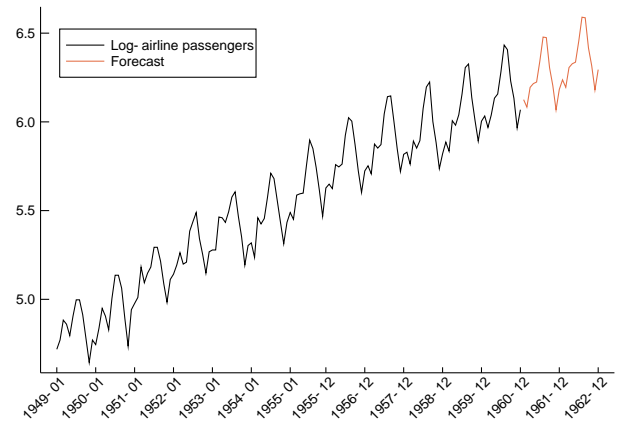

Fig. 2. Log-airline passengers historical data and forecast for the following two years.
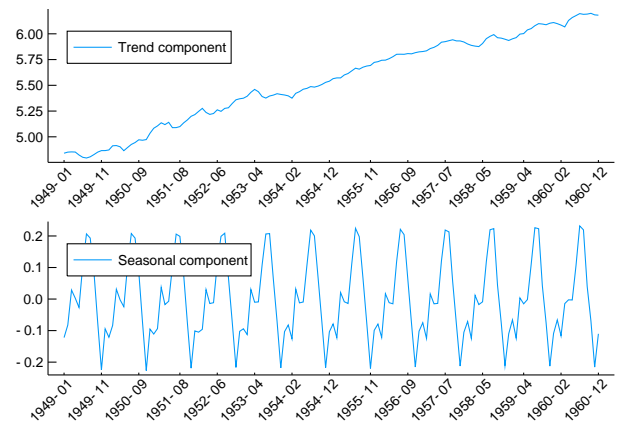

Fig. 3. Smoothed trend and seasonal components of the log-airline passengers time series.

with the goal of obtaining the best contracting strategy for a distribution company. To this end, the function `simulate` will be used to perform Monte Carlo simulation. We will also use the square-root Kalman filter for this application.

```
# Specify the state-space model
model = structural(consumption, 12; X = temperature)
# Estimate the state-space model
ss = statespace(model; filter_type = SquareRootFilter{Float64})
# Number of months ahead to be simulated
N = 24
# Number of scenarios to be simulated
S = 1000
# Perform simulation
sim = simulate(ss, N, S)
```

The time series and the resulting simulation can be seen in Fig. 4. The future scenarios are graphically represented by their mean, which is identical to the forecast at each time period, as well as the 5% and 95% quantiles.
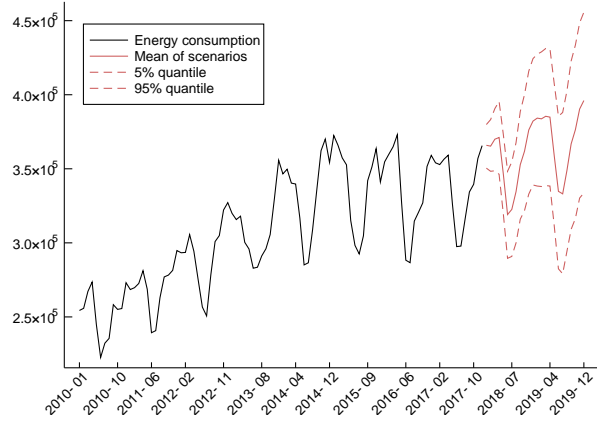
Fig. 4.   Monthly energy consumption historical data and simulation of future scenarios.

7.3   Vehicle tracking

Finally, in order to illustrate one application that does not fall into any of the predefined models, thus requiring a user-defined model, let us consider an example from control theory. More precisely, we are going to use `StateSpaceModels.jl` to track a vehicle from noisy sensor data. In this case, y_t is a 2×1 observation vector representing the corrupted measurements of the vehicle's position on the two-dimensional plane in instant t. Since sensors collect the observations with the presence of additive Gaussian noise, we need to filter the observation in order to obtain a better estimate of the vehicle's position.

The position and speed in each dimension compose the state of the vehicle. Let us refer to $x_t^{(d)}$ as the position on the axis d and to $\dot{x}_t^{(d)}$ as the speed on the axis d in instant t. Additionally, let $\eta_t^{(d)}$ be the input drive force on the axis d, which acts as state noise. For a single dimension, we can describe the vehicle dynamics as

$$
\begin{aligned}
x_{t+1}^{(d)} &= x_t^{(d)} + \Big(1 - \frac{\rho \Delta_t}{2}\Big)\Delta_t \dot{x}_t^{(d)} + \frac{\Delta_t^2}{2}\eta_t^{(d)}, \\
\dot{x}_{t+1}^{(d)} &= (1-\rho)\dot{x}_t^{(d)} + \Delta_t \eta_t^{(d)},
\end{aligned}
\tag{24}
$$

where Δ_t is the time step and ρ is a known damping effect on speed.

We can cast the dynamical system (24) as a state-space model in the following manner:

$$
y_t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}\alpha_{t+1} + \varepsilon_t,
$$

$$
\alpha_{t+1} = \begin{bmatrix} 1 & (1 - \frac{\rho \Delta_t}{2})\Delta_t & 0 & 0 \\ 0 & (1-\rho) & 0 & 0 \\ 0 & 0 & 1 & (1 - \frac{\rho \Delta_t}{2})\Delta_t \\ 0 & 0 & 0 & (1-\rho) \end{bmatrix}\alpha_t + \begin{bmatrix} \frac{\Delta_t^2}{2} & 0 \\ \Delta_t & 0 \\ 0 & \frac{\Delta_t^2}{2} \\ 0 & \Delta_t \end{bmatrix}\eta_t,
$$

where $\alpha_t = (x_t^{(1)}, \dot{x}_t^{(1)}, x_t^{(2)}, \dot{x}_t^{(2)})^\top$ and $\eta_t = (\eta_t^{(1)}, \eta_t^{(2)})^\top$.

We can formulate the vehicle tracking problem in the `StateSpaceModels.jl` framework as:

```
# State transition matrix
T = kron(Matrix{Float64}(I, p, p), [1 (1 - ρ * Δ / 2) * Δ;
0 (1 - ρ * Δ)])
```

```
# Input matrix
R = kron(Matrix{Float64}(I, p, p), [.5 * Δ^2; Δ])
# Output (measurement) matrix
Z = kron(Matrix{Float64}(I, p, p), [1 0])
# User defined model
model = StateSpaceModel{Float64}(y, Z, T, R)
# Estimate vehicle speed and position
ss = statespace(model)
```

In this example, we define the noise variances H and Q, generate the noises and simulate a random vehicle trajectory using the state-space equations.

```
# Generate random actuators
Q = .5 * Matrix{Float64}(I, q, q)
η = MvNormal(zeros(q), Q)
# Generate random measurement noise
H = 2. * Matrix{Float64}(I, p, p)
ε = MvNormal(zeros(p), H)
# Simulate vehicle trajectory
α = Matrix{Float64}(undef, n + 1, m)
y = Matrix{Float64}(undef, n, p)
for t in 1:n
    y[t, :] = Z * α[t, :] + rand(ε)
    α[t + 1, :] = T * α[t, :] + R * rand(η)
end
```

An illustration of the results can be seen in Fig. 5. It can be seen that the measurements are reasonably noisy when compared to the true position. Furthermore, the estimated positions, represented by the smoothed state, effectively estimate the true positions with small inaccuracies.
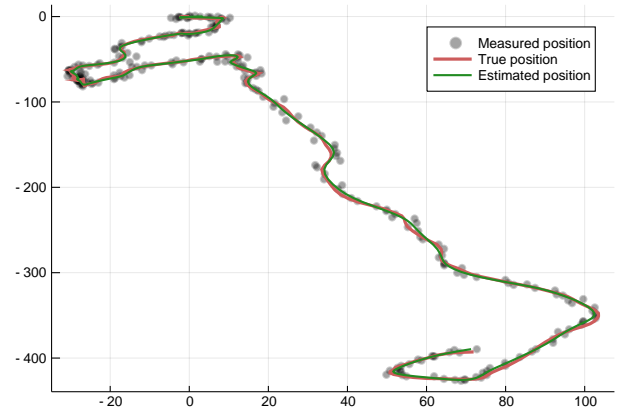

Fig. 5.   Vehicle tracking using a state-space model.

8.   Conclusion

`StateSpaceModels.jl` is a flexible package for time-series modeling, forecasting, and simulating that is fully implemented in Julia. The package contains an implementation of the Kalman filter and smoother as well as their square-root variants. Additionally, users have the ability to use any implemented filter or optimization method. Missing observations in the form of `NaN` values are automatically treated in the filtering and smoothing steps.

Besides comprising several predefined classical models, it is also possible to define any linear model with the `StateSpaceModel`

constructor. Forecasting and Monte Carlo simulation of future scenarios are also available. Finally, the package documentation contains a manual with straightforward examples that are simple to reproduce.

9. References

[1] StateSpace.jl, [Online]. Available: `https://github.com/ElOceanografo/StateSpace.jl`

[2] Fredrik B. Carlson and Mattias Fält. ControlSystems.jl: A Control Systems Toolbox for Julia, 2016 [Online]. Available: `https://github.com/JuliaControl/ControlSystems.jl`

[3] George EP Box and David A Pierce. Distribution of residual autocorrelations in autoregressive-integrated moving average time series models. *Journal of the American statistical Association*, 65(332):1509–1526, 1970.

[4] George Casella and Roger L. Berger. *Statistical Inference*, volume 2. Duxbury Pacific Grove, CA, 2002.

[5] James Durbin and Siem Jan Koopman. *Time Series Analysis by State Space Methods*. Oxford University Press, 2012.

[6] Jouni Helske. KFAS: Exponential family state space models in R. *Journal of Statistical Software*, 78(10):1–39, 2017.

[7] Carlos M Jarque and Anil K Bera. Efficient tests for normality, homoscedasticity and serial independence of regression residuals. *Economics letters*, 6(3):255–259, 1980.

[8] Rudolph Emil Kalman. A new approach to linear filtering and prediction problems. *Journal of basic Engineering*, 82(1):35–45, 1960.

[9] Siem Jan Koopman, Andrew C. Harvey, Jurgen A. Doornik, and Neil Shephard. STAMP 6.0: Structural time series analyser, modeller and predictor. *London: Timberlake Consultants*, 2000.

[10] Dong C. Liu and Jorge Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

[11] Helmut Lütkepohl and Fang Xu. The role of the log transformation in forecasting economic variables. *Empirical Economics*, 42(3):619–638, 2012.

[12] Patrick Kofod Mogensen and Asbjørn Nilsen Riseth. Optim: A mathematical optimization package for Julia. *Journal of Open Source Software*, 3(24), 2018.

[13] Raphael Saavedra, Guilherme Bodin, et al. Simulating low and high-frequency energy demand scenarios in a unified framework–Part I: Low-frequency simulation. In *Proceedings of the L Simpósio Brasileiro de Pesquisa Operacional*, 2018.

[14] Raphael Saavedra, Guilherme Bodin, and Mario Souto. StateSpaceModels.jl, [Online]. Available: `https://github.com/LAMPSPUC/StateSpaceModels.jl`

[15] Moritz Schauer. Kalman.jl, [Online]. Available: `https://github.com/mschauer/Kalman.jl`

[16] Skipper Seabold and Josef Perktold. Statsmodels: Econometric and statistical modeling with Python. In *9th Python in Science Conference*, 2010.

[17] Greg Welch, Gary Bishop, et al. An introduction to the Kalman filter. Technical report, Department of Computer Science, University of North Carolina at Chapel Hill, 1995.

[18] Lotfi Zadeh and Charles Desoer. *Linear System Theory: The State Space Approach*. Courier Dover Publications, 2008.