SsfPack 2.0: Statistical algorithms for models in state space

An Ox link to underlying C code

Siem Jan KOOPMAN
CentER, Tilburg University, 5000 LE Tilburg, The Netherlands
Neil SHEPHARD
Nuffield College, Oxford, OX1 1NF, UK
Jurgen A DOORNIK
Nuffield College, Oxford, OX1 1NF, UK

March 1998

Abstract

This paper discusses and documents the algorithms provided by *SsfPack 2.0* (release date March 30, 1998). *SsfPack* is a suite of C routines for carrying out computations involving the statistical analysis of univariate and multivariate models in state space form. Functions are available for prediction, filtering, moment smoothing, simulation smoothing and forecasting. The headers of these routines are documented here. The emphasis is on documenting the link we have made to the Ox computing environment. Therefore, *SsfPack* can be easily used for implementing, fitting and analysing Gaussian and non-Gaussian models relevant to many areas of econometrics and statistics. Some Gaussian illustrations are given.

Keywords: C code; Importance sampling; Kalman filtering and smoothing; Markov chain Monte Carlo; Ox; Simulation smoother; State space.

Contents

1 Introduction **4**
- 1.1 Installation . . . 4
- 1.2 Copyrights and references . . . 4
- 1.3 Overview of SsfPack functions . . . 5

2 State space form **5**
- 2.1 Initial conditions . . . 6
- 2.2 Time-varying state space form . . . 7
- 2.3 Formulating the state space . . . 7
- 2.4 Missing values . . . 8

3 Univariate linear Gaussian models **8**
- 3.1 ARMA models . . . 8
- 3.2 Unobserved components time series models . . . 9
- 3.3 Regression models . . . 13
- 3.4 Nonparametric cubic spline models . . . 14

4 Review of algorithms: the basic functions **16**
- 4.1 State space recursion . . . 16
- 4.2 Kalman filter . . . 18
- 4.3 Moment smoothing . . . 20
  - 4.3.1 Disturbance smoothing . . . 20
  - 4.3.2 Quick state smoothing . . . 21
- 4.4 Simulation smoother . . . 23
  - 4.4.1 Disturbance simulation smoothing . . . 23
  - 4.4.2 State simulation smoothing . . . 25
- 4.5 Missing values . . . 27

5 Gaussian applications: ready-to-use functions **29**
- 5.1 Likelihood and score evaluation . . . 29
- 5.2 Prediction, forecasting and smoothing . . . 31
  - 5.2.1 Prediction . . . 31
  - 5.2.2 Forecasting . . . 31
  - 5.2.3 State smoothing . . . 31
  - 5.2.4 An example . . . 32
  - 5.2.5 Disturbance smoothing . . . 33
- 5.3 The conditional density: mean calculation and simulation . . . 34
  - 5.3.1 Quick mean calculation of states . . . 34
  - 5.3.2 Simulation for states . . . 35
  - 5.3.3 Mean calculation of disturbances . . . 35
  - 5.3.4 Simulation for disturbances . . . 35

6 Gaussian illustrations **37**
- 6.1 Maximum likelihood estimation . . . 37
- 6.2 Detecting outliers and structural breaks . . . 39
- 6.3 Regression analysis . . . 40
- 6.4 Spline smoothing and interpolation . . . 42
- 6.5 Seasonal adjustment and detrending . . . 43
- 6.6 Monte Carlo simulations and parametric bootstrap tests . . . 44

6.7 Bayesian parameter estimation . . . 46
6.7.1 The basics . . . 46
6.7.2 Metropolis algorithm . . . 47
6.7.3 Augmentation . . . 47

7 Header files of C functions **49**
7.1 Kalman filter . . . 50
7.2 Smoothing . . . 50
7.3 Simulation smoothing: weights . . . 51
7.4 Simulation smoothing: draws . . . 52

A Nile data **52**

B Airline data **53**

1 Introduction

This paper documents the package *SsfPack 2.0* which carries out computations for the statistical analysis of general univariate and multivariate state space models. *SsfPack* is a suite of C routines collected into a library which can be linked to different computing environments. In particular, it can be used in many areas of econometrics and statistics as will become apparent from the illustrations given.

Standard econometrics and statistical packages such as SAS, STAMP, SPSS, PcGive, Splus and Minitab have many canned options for the fitting of standard time series models. However, when we work on new areas of time series modelling it is important to have generic tools to handle them which offer enormous flexibility and can carry out the more routine aspects of the computational problem. *SsfPack* provides fast hard coded general filtering, moment smoothing and simulation smoothing routines. These can be tailored towards particular applications by the users.

This version of the C library *SsfPack* is linked to the *Ox* matrix programming language of Doornik (1996). Other implementations of *SsfPack* are due to follow. However, the *SsfPack* header file includes four important C functions which can be called directly. Thus a part of the dynamic link library of *SsfPack* can also be used in other programming environments such as Gauss and Splus. *SsfPack* is free of charge and it can be used together with the free versions of *Ox*. The files of *SsfPack* can be downloaded from the Internet or they can be mailed to you on request.

1.1 Installation

General information about the package *SsfPack 2.0* can be obtained from the website

`http://center.kub.nl/stamp/ssfpack.htm`

Also it enables the visitor to download `ssfpack.zip` which contains the files:

```
ssfpack.dll  // the dynamic link library
ssfpack.h    // the header file
ssfpack.ps   // postscript file of documentation
read.me      // installation notes
ssf*.ox      // example ox programs from this documentation
```

When the internet is not available to you, please contact the first author who will be happy to provide you with the *SsfPack* disk (send a letter or email `s.j.koopman@kub.nl`).

It is assumed that *Ox* 1.20a or higher is installed on your computer. If this is not the case, we advise you to go to the website of Jurgen A Doornik:

`http://www.nuff.ox.ac.uk/users/doornik/`

Installation is simple: just follow the guidelines given by the `read.me` file.

1.2 Copyrights and references

 Therefore, if you publish work for which you have used *SsfPack*, we like you to refer to *SsfPack* as follows:

> Koopman S.J., N. Shephard and J.A. Doornik (1998)
> SsfPack 2.0: Statistical algorithms for models in state space.
> http://center.kub.nl/stamp/ssfpack.htm.

1.3 Overview of SsfPack functions

Models in state space form

`GetSsfArma`	puts ARMA model in state space (section 3.1).
`GetSsfReg`	puts regression model in state space (section 3.3).
`GetSsfSpline`	puts nonparametric cubic spline model in state space (section 3.4).
`GetSsfStsm`	puts unobserved components time series model in state space (section 3.2).

General state space algorithms

`KalmanFil`	returns output of the Kalman filter (section 4.2).
`KalmanSmo`	returns output of the basic smoothing algorithm (section 4.3).
`SimSmoDraw`	returns a sample from the simulation smoother (section 4.4).
`SimSmoWgt`	returns covariance output of the simulation smoother (section 4.4).

Ready-to-use functions

`SsfLik`	returns log-likelihood function (section 5.1).
`SsfLikConc`	returns profile log-likelihood function (section 5.1).
`SsfLikSco`	returns score vector (section 5.1).
`SsfRecursion`	returns output of the state space recursion (section 4.1).
`SsfCondDens`	returns mean or a draw from the conditional density (section 5.3)
`SsfMomentEst`	returns output from prediction, forecasting and smoothing (section 5.2).

2 State space form

The state space form provides a unified representation of a wide range of linear Gaussian time series models including ARMA models, time-varying regression models, dynamic linear models and unobserved components time series models. This framework also encapsulates different specifications for nonparametric and spline regressions. The Gaussian state space form consists of a transition equation and a measurement equation; we formulate it as

$$\begin{aligned} \alpha_{t+1} &= d_t + T_t\alpha_t + H_t\varepsilon_t, \qquad \alpha_1 \sim \mathrm{N}(a, P), \qquad t = 1, \ldots, n, && (1)\\ y_t &= c_t + Z_t\alpha_t + G_t\varepsilon_t, \qquad \varepsilon_t \sim \mathrm{NID}(0, I), && (2) \end{aligned}$$

where $\mathrm{NID}(\mu, \Psi)$ indicates a normally identical distributed variable with mean μ and variance matrix Ψ and, similarly, N stands for a normally distributed variable. The state equation (1) has a Markovian structure to describe the serial correlation of the time series y_t. The measurement equation (2) relates the $N \times 1$ vector of observations y_t in terms of the $m \times 1$ state vector α_t and the $r \times 1$ vector of disturbances ε_t, for $t = 1, \ldots, n$. The deterministic matrices T_t, Z_t, H_t and G_t are referred to as system matrices and they are often sparse selection matrices. The vectors d_t and c_t are fixed and known, for $t = 1, \ldots, n$, and they are often zero. When the system matrices are constant over time, we drop the time-indices to obtain the matrices T, Z, H and G. The resulting state space form is referred to as time-invariant. The variance matrix P of the initial state vector α_1 may contain diffuse elements, that is

$$P = P_* + \kappa P_\infty, \qquad \kappa \text{ is large}, \tag{3}$$

where P_* is a symmetric $m \times m$ matrix, P_∞ is a diagonal $m \times m$ matrix composed of zero and unity values and, for example, $\kappa = 10^7$. When the i-th diagonal element of P_∞ is unity, the corresponding i-th column and row of P_* are assumed to be zero.

The state space form in *SsfPack* is represented by,

$$\begin{pmatrix} \alpha_{t+1} \\ y_t \end{pmatrix} = \delta_t + \Phi_t \alpha_t + u_t, \qquad u_t \sim \mathrm{NID}\left(0, \Omega_t\right), \qquad t = 1, \ldots, n, \tag{4}$$

with

$$\delta_t = \begin{pmatrix} d_t \\ c_t \end{pmatrix}, \qquad \Phi_t = \begin{pmatrix} T_t \\ Z_t \end{pmatrix}, \qquad u_t = \begin{pmatrix} H_t \\ G_t \end{pmatrix} \varepsilon_t, \qquad \Omega_t = \begin{pmatrix} H_t H_t' & H_t G_t' \\ G_t H_t' & G_t G_t' \end{pmatrix}, \tag{5}$$

and $\alpha_1 \sim \mathrm{N}\left(a, P\right)$. Note that the vector δ_t is $m + N \times 1$, the matrix Φ_t is $m + N \times m$ and the matrix Ω_t is $m + N \times m + N$. Specifying a model in state space form within *SsfPack* can be done in different ways depending on its complexity. At the most elementary level, the state space form is supposed to be time-invariant with $\delta = 0$, $a = 0$ and $P = \kappa I$ so only two matrices are required, that is

$$\Phi = \begin{pmatrix} T \\ Z \end{pmatrix}, \qquad \Omega = \begin{pmatrix} HH' & HG' \\ GH' & GG' \end{pmatrix}.$$

For example, consider the local linear trend model,

$$\begin{array}{rcll} \mu_{t+1} &=& \mu_t + \beta_t + \eta_t, & \eta_t \sim \mathrm{NID}\left(0, 2\right), \\ \beta_{t+1} &=& \beta_t + \zeta_t, & \zeta_t \sim \mathrm{NID}\left(0, 1\right), \\ y_t &=& \mu_t + \xi_t, & \xi_t \sim \mathrm{NID}\left(0, 5\right), \end{array} \tag{6}$$

with $\mu_1 \sim \mathrm{NID}\left(0, \kappa\right)$ and $\beta_1 \sim \mathrm{NID}\left(0, \kappa\right)$ where κ is large; for more details about this model, see section 4.2. The matrices Φ and Ω for this model are given by

$$\Phi = \begin{pmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad \Omega = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 5 \end{pmatrix}.$$

In *Ox* code, this model is inputted by

```
mPhi = <1,1;0,1;1,0>;
mOmega = diag(<2,1,5>);
```

The specification implies that $\alpha_t = (\mu_t, \beta_t)'$.

2.1 Initial conditions

When the initial state conditions are not explicitly defined, it will be assumed that

$$a = 0, \qquad P_* = 0, \qquad P_\infty = I, \tag{7}$$

such that the initial state vector is fully diffuse, that is $\alpha_1 \sim \mathrm{N}\left(0, \kappa I\right)$. To specify the initial state conditions (3) explicitly, the $m + 1 \times m$ matrix

$$\Sigma = \begin{pmatrix} P \\ a' \end{pmatrix}, \tag{8}$$

is required. The block matrix P in Σ is equal to matrix P_* except when a diagonal element of P is equal to -1 indicating that the corresponding initial state vector element is diffuse. When a diagonal element of P is -1, the corresponding row and column of P are not used. For example, the initial state vector of the local linear trend model (6) is fully diffuse, so in *Ox* we have

```
mSigma = <-1,0;0,-1;0,0>;
```

2.2 Time-varying state space form

When some elements of the system matrices are not constant but change over time, additional administration is required. We introduce the index matrices J_Φ, J_Ω and J_δ which must have the same dimension as Φ, Ω and δ, respectively. The elements of the index matrices are all set to -1 except the elements for which the corresponding elements in Φ, Ω and δ are time varying. The non-negative index value indicates the row of some data matrix which contain the time varying values. Also, when no element of a system matrix is time-varying, the corresponding index matrix can be set to the 1×1 zero matrix; in *Ox*, that is <0>. For example, the local linear trend model (6) with time-varying variances (instead of being a set of constants) is defined as

```
mJ_Phi = <0>;
mJ_Omega= <4,-1,-1;-1,0,-1;-1,-1,2>;
mJ_Delta = <0>;
```

for which the variances of ξ_t are found in the third row of an accompanying data matrix (note that indexing starts at value 0 in *Ox*). The variances of η_t and ζ_t are to be found in the fifth row and the first row, respectively, of the data matrix which must have at least five rows and n columns. None of the elements of matrix Φ is time-varying, therefore we set J_Φ and J_δ to a zero matrix.

2.3 Formulating the state space

SsfPack allows for a full range of different state space forms: from a simple time-invariant model to a complicated time-varying model. The algorithms of section 4 require knowledge of the state space which can be given in different levels. The most elementary state space form only requires the matrix specifications of Φ and Ω; in this case, it is assumed that $\delta = 0$, $a = 0$ and $P = \kappa I$ with $\kappa = 10^7$. The initial conditions can explicitly be given by defining an appropriate matrix Σ. The implicit $m + N \times 1$ vector δ can also be given when it is nonzero. A time-invariant state space form can be inputted in three different levels, that is

```
mPhi, mOmega
mPhi, mOmega, mSigma
mPhi, mOmega, mSigma, mDelta
```

Other ways of inputting a time-invariant state space form is not possible for the algorithms of section 4. A state space form with time-varying system elements can be given as discussed in section 2.2. The fourth possible way of inputting a state space for the algorithms of section 4 is

```
mPhi, mOmega, mSigma, mDelta, mJ_Phi, mJ_Omega, mJ_Delta, mX
```

where `mX` is the data matrix with n columns as discussed in section 2.2. When `mSigma`, `mDelta`, `mJ_Phi`, `mJ_Omega` or `mJ_Delta` is not relevant for the state space form, it can be inputted as a 1×1 zero matrix, that is <0> in *Ox*. For the algorithms of sections 4 and 5, the input statement of the state space form is indicated by

```
{Ssf}
```

which refers to one of the four input statements of above.

2.4 Missing values

The algorithms of *SsfPack* can handle missing values. A missing value is only recognised within the data matrix $(y_1, \ldots, y_n)$ and it must have the value -9999.99. It is assumed that the system matrices are given and therefore no missing values are allowed within the matrices Φ, Ω, Σ and δ or their time-varying counterparts. This implies that if the value -9999.99 is present within these state space quantities, it will be taken as the numerical value -9999.99 and not as a missing value.

3 Univariate linear Gaussian models

3.1 ARMA models

The autoregressive moving average model of order p and q, denoted by ARMA(p, q), is given by

$$y_t = \phi_1 y_{t-1} + \ldots + \phi_p y_{t-p} + \xi_t + \theta_1 \xi_{t-1} + \ldots + \theta_q \xi_{t-q}, \quad \xi_t \sim \text{NID}\left(0, \sigma^2\right). \tag{9}$$

The lag polynomial of order p is defined as $A(L) = 1 + A_1 L + \ldots + A_p L^p$ where L is the lag operator such that $L^r y_t = y_{t-r}$. The model (9) is stationary when the roots of the polynomial $\phi(L) = 1 - \phi_1 L - \ldots - \phi_p L^p$ are within the unit circle and the model is non-invertible when the roots of the polynomial $\theta(L) = 1 + \theta_1 L + \ldots + \theta_q L^q$ are within the unit circle. The parameter space can be restricted to obtain a stationary non-invertible ARMA model by following the arguments in Ansley and Kohn (1986). Any ARMA model can be written as a first order vector autoregressive, VAR(1), model. Such a representation, which is not unique, is called a companion form or Markov representation. The most commonly quoted companion form of the ARMA model is $y_t = (1, 0, 0, ..., 0)\alpha_t$ and

$$\alpha_{t+1} = \begin{pmatrix} \phi_1 & 1 & 0 & \cdots & 0 \\ \phi_2 & 0 & 1 & \cdots & 0 \\ \vdots & & & \ddots & \\ \phi_{m-1} & 0 & 0 & \cdots & 1 \\ \phi_m & 0 & 0 & \cdots & 0 \end{pmatrix} \alpha_t + \begin{pmatrix} 1 \\ \theta_1 \\ \vdots \\ \theta_{m-2} \\ \theta_{m-1} \end{pmatrix} \xi_t, \quad \xi_t \sim \text{NID}\left(0, \sigma^2\right), \tag{10}$$

with $m = \max(p, q+1)$. This can be compactly written as $\alpha_{t+1} = T\alpha_t + h\xi_t$ where the time-invariant matrices T and h are given in (10). Multivariate or vector ARMA models can also be written in companion VAR(1) form. In the case of a stationary ARMA model in state space form, the unconditional distribution of the state vector is $\alpha_t \sim \text{N}(0, V)$, where $V = TVT' + \sigma^2 hh'$. There are different ways of numerically solving out for V. The most straightforward way is to invert a matrix in order to solve the linear equations $(I - T \otimes T)\, vec(V) = \sigma^2 vec(hh')$ for V, where the $m^2 \times 1$ vector $vec(V)$ stacks the columns of V; see, for example, Magnus and Neudecker (1988, Theorem 2, p. 30).

SsfPack implementation The *SsfPack* routine `GetSsfArma` provides the appropriate system matrices for any univariate ARMA model. The routine requires two vectors containing the autoregressive parameters $\phi_1, \ldots, \phi_p$ and the moving average parameters $\theta_1, \ldots, \theta_q$ which must be chosen in such a way that the implied ARMA model is stationary and non-invertible; *SsfPack* does not verify this. The function call

```
GetSsfArma(mAr, mMa, StDev, &mPhi, &mOmega, &mSigma);
```

places the ARMA coefficients within the appropriate state elements and it solves the set of linear equations for the variance matrix of the initial state vector. The matrices `mAr` and `mMa`,

containing the autoregressive and the moving average parameters, respectively, should be either row vectors or column vectors. The scalar value `StDev` represents σ in (10). The function returns three pointers (addresses) to the matrices Φ, Ω and Σ.

Example The following example outputs the relevant state space matrices for the ARMA(2,1) model $y_t = 0.6y_{t-1} + 0.2y_{t-1} + \xi_t - 0.2\xi_{t-1}$ with $\xi_t \sim \text{NID}(0, 1)$.

Ox code of `ssfarma.ox`:

```
#include <oxstd.h>
#include "SsfPack.h"

main()
{
 decl mPhi, mOmega, mSigma;

 // ARMA(2,1) model with standard deviation 1
 GetSsfArma(<0.6, 0.2>, <-0.2>, 1, &mPhi, &mOmega, &mSigma);
 // note: AR(2) model -> GetSsfArma(<0.6,0.2>, <0>, 1, ...);
 // note: MA(1) model -> GetSsfArma(<0>, <-0.2>, 1, ...);

 // print state space
 print("Phi", mPhi);
 print("Omega", mOmega);
 print("Sigma", mSigma);
}
```

Ox output:

```
Phi
      0.60000       1.0000
      0.20000      0.00000
       1.0000      0.00000
Omega
       1.0000     -0.20000      0.00000
     -0.20000     0.040000      0.00000
      0.00000      0.00000      0.00000
Sigma
       1.5631    -0.015538
    -0.015538      0.10252
      0.00000      0.00000
```

3.2 Unobserved components time series models

The state space model also deals directly with unobserved components time series models used in structural time series and dynamic linear models; see, for example, West and Harrison (1997), Kitagawa and Gersch (1996) and Harvey (1989). Ideally such component models should be constructed from subject matter considerations, tailored to the particular problem at hand. However, in practice there are a group of commonly used components which are used extensively. For example, a specific time series model may include the trend μ_t, the seasonal γ_t, the cycle ψ_t and the irregular ε_t component which is given by

$$y_t = \mu_t + \gamma_t + \psi_t + \xi_t, \quad \text{where} \quad \xi_t \sim \text{NID}\left(0, \sigma_\xi^2\right), \qquad t = 1, \ldots, n. \tag{11}$$

Explanatory variables (*i.e.* regression and intervention effects) can be included in this model straightforwardly.

The trend component μ_t is usually specified as

$$\begin{array}{rcll} \mu_{t+1} & = & \mu_t + \beta_t + \eta_t, & \eta_t \sim \text{NID}\left(0, \sigma_\eta^2\right), \\ \beta_{t+1} & = & \beta_t + \zeta_t, & \zeta_t \sim \text{NID}\left(0, \sigma_\zeta^2\right), \end{array} \tag{12}$$

with $\mu_0 \sim \text{N}(0, \kappa)$ and $\beta_0 \sim \text{N}(0, \kappa)$ where κ is large. The model with trend and irregular is easily placed into state space form; see also section 2. Sometimes σ_η^2 of (12) is set to zero, and so we refer to μ_t as a smooth trend or an integrated random walk component. When σ_η^2 and σ_ς^2 are both set to zero, we obtain a deterministic linear trend in which $\mu_t = \mu_0 + \beta_0 t$.

The specification of the 'dummy' seasonal component γ_t is given by

$$S(L)\gamma_t = \omega_t, \quad \text{where} \quad \omega_t \sim \text{NID}(0, \sigma_\omega^2) \quad \text{and} \quad S(L) = 1 + L + ... + L^{s-1}, \tag{13}$$

with s equal to the number of seasons, for $t = 1, \ldots, n$. When σ_ω^2 of (13) is set to zero, the seasonal component is fixed. In this case, the seasonal effects sum to zero over the previous 'year'; this ensures that it cannot be confounded with the other components. The space representation for $s = 4$ is given by

$$\begin{pmatrix} \gamma_t \\ \gamma_{t-1} \\ \gamma_{t-2} \end{pmatrix} = \begin{pmatrix} -1 & -1 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} \gamma_{t-1} \\ \gamma_{t-2} \\ \gamma_{t-3} \end{pmatrix} + \begin{pmatrix} \omega_t \\ 0 \\ 0 \end{pmatrix}, \qquad \begin{pmatrix} \gamma_1 \\ \gamma_0 \\ \gamma_{-1} \end{pmatrix} \sim \text{N}(0, \kappa I_3).$$

The seasonal component also has other representations. The so-called trigonometric seasonal γ_t is generated by

$$\gamma_t = \sum_{j=1}^{[s/2]} \gamma_{j,t}^+, \quad \text{where} \quad \begin{pmatrix} \gamma_{j,t+1}^+ \\ \gamma_{j,t+1}^* \end{pmatrix} = \begin{pmatrix} \cos\lambda_j & \sin\lambda_j \\ -\sin\lambda_j & \cos\lambda_j \end{pmatrix} \begin{pmatrix} \gamma_{j,t}^+ \\ \gamma_{j,t}^* \end{pmatrix} + \begin{pmatrix} \omega_{j,t}^+ \\ \omega_{j,t}^* \end{pmatrix}, \tag{14}$$

with $\lambda_j = 2\pi j/s$ as the j-th seasonal frequency and

$$\begin{pmatrix} \omega_{j,t}^+ \\ \omega_{j,t}^* \end{pmatrix} \sim \text{NID}\left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \sigma_\omega^2 I_2 \right\}, \qquad j = 1, \ldots, [s/2].$$

Note that for s even $[s/2] = s/2$, while for s odd, $[s/2] = (s-1)/2$. For s even, the process $\gamma_{j,t}^*$, with $j = s/2$, can be dropped. The state space representation is straightforward and the initial conditions are $\gamma_{j,1}^+ \sim \text{N}(0, \kappa)$ and $\gamma_{j,1}^* \sim \text{N}(0, \kappa)$, for $j = 1, \ldots, [s/2]$. The dummy and trigonometric specifications for γ_t have different dynamic properties; see Harvey (1989). For example, the trigonometric seasonal process evolves more smoothly; it can be shown that the sum of the seasonals over the past 'year' follows a MA$(s-2)$ rather than white noise. The same property holds for the Harrison and Stevens seasonal representation for which all s individual seasonal effects collected in the vector $\gamma_t^\times$ follow a random walk, that is

$$\begin{array}{rcl} \gamma_{t+1}^\times & = & \begin{pmatrix} \gamma_1 \\ \gamma_2 \\ \vdots \\ \gamma_s \end{pmatrix}_{t+1} = \gamma_t^\times + \omega_t, \\ \text{where} \quad \omega_t & = & \begin{pmatrix} \omega_1 \\ \omega_2 \\ \vdots \\ \omega_s \end{pmatrix}_t \sim \text{NID}\left\{0, \sigma_\omega^2 \left(\frac{sI_s - i_s i_s'}{s-1}\right)\right\}, \end{array}$$

and i_s is a $s \times 1$ vector of ones; see Harrison and Stevens (1976). The specific covariance structure between the s disturbance terms enforces the seasonal effects to sum to zero over the previous 'year'. Also, the covariances between the s seasonal disturbances are equal. The state space form is set up such that it selects the appropriate seasonal effect from $\gamma_t^{\times}$; this implies a time-varying state space framework. However, the state space representation can be modified to a time-invariant form as follows. Let $\gamma_t = (1, 0')\gamma_t^{\times}$, then

$$\begin{aligned} \gamma_{t+1}^{\times} = \begin{pmatrix} 0 & I_{s-1} \\ 1 & 0 \end{pmatrix} \gamma_t^{\times} + \omega_t, \quad \text{where} \quad & \omega_t \sim \text{NID}\left(0, \sigma_\omega^2 \tfrac{sI_s - i_s i_s'}{s-1}\right), \\ \text{and} \quad & \gamma_1^{\times} \sim \text{N}\left(0, \kappa \tfrac{sI_s - i_s i_s'}{s-1}\right). \end{aligned} \tag{15}$$

The implications of the different seasonal specifications are discussed in more detail by Harvey, Koopman, and Penzer (1998).

An interesting seasonal component related to the Harrison and Stevens seasonal is given by $\gamma_t = (1, 0')\gamma_t^{\times}$ with

$$\begin{aligned} \gamma_{t+1}^{\times} = \rho\gamma_t^{\times} + \omega_t, \quad \text{where} \quad & \omega_t \sim \text{NID}\left\{\bar{\gamma}, \sigma_\omega^2 (1-\rho^2) \tfrac{sI_s - i_s i_s'}{s-1}\right\} \\ \text{and} \quad & \gamma_1^{\times} \sim \text{N}\left\{\bar{\gamma}, \sigma_\omega^2 \tfrac{sI_s - i_s i_s'}{s-1}\right\}. \end{aligned}$$

This specification provides a stationary seasonal model around some average seasonal pattern given by the unknown fixed $s \times 1$ vector of means $\bar{\gamma}$. It is possible to have both stationary and nonstationary seasonal components in a single unobserved components model, but in that case identification requirements stipulate that $\bar{\gamma}$ have to be set to zero.

The cycle component ψ_t is specified as

$$\begin{pmatrix} \psi_{t+1} \\ \psi_{t+1}^* \end{pmatrix} = \rho \begin{pmatrix} \cos\lambda_c & \sin\lambda_c \\ -\sin\lambda_c & \cos\lambda_c \end{pmatrix} \begin{pmatrix} \psi_t \\ \psi_t^* \end{pmatrix} + \begin{pmatrix} \chi_t \\ \chi_t^* \end{pmatrix}, \tag{16}$$

with

$$\begin{pmatrix} \chi_t \\ \chi_t^* \end{pmatrix} \sim \text{NID}\left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \sigma_\psi^2 \left(1 - \rho^2\right) I_2 \right\},$$

and for which $0 < \rho \leq 1$ is the 'damping factor', the frequency is $\lambda_c = 2\pi/c$ and c is the 'period' of the cycle. The initial conditions are $\psi_0 \sim \text{N}\left(0, \sigma_\psi^2\right)$ and $\psi_0^* \sim \text{N}\left(0, \sigma_\psi^2\right)$ with $\text{cov}\left(\psi_0, \psi_0^*\right) = 0$. The variance of χ_t and χ_t^* is given in terms of σ_ψ^2 and ρ so that when $\rho \to 1$ the cycle component reduces to a deterministic (but stationary) sine-cosine wave; see Harvey and Streibel (1998).

SsfPack implementation The *SsfPack* routine `GetSsfStsm` provides the relevant system matrices for any univariate structural time series model. The routine requires one matrix containing the model information in the following form

$$\texttt{mStsm} = \begin{pmatrix} \text{CMP\_LEVEL} & \sigma_\eta & 0 & 0 \\ \text{CMP\_SLOPE} & \sigma_\zeta & 0 & 0 \\ \text{CMP\_SEAS\_...} & \sigma_\omega & s & 0 \\ \text{CMP\_CYC\_0} & \sigma_\psi & \lambda_c & \rho \\ \vdots & \vdots & \vdots & \vdots \\ \text{CMP\_CYC\_9} & \sigma_\psi & \lambda_c & \rho \\ \text{CMP\_IRREG} & \sigma_\xi & 0 & 0 \end{pmatrix}.$$

The input matrix may contain less number of rows than the above setup and the rows may have a different sequential order. However, the state vector is organised in the sequence level,

slope, seasonal, cycle and irregular. The first column of `mStsm` is of string type and the other columns contain real values. The seasonal component can be set as `CMP_SEAS_DUMMY` for the seasonal dummy specification (13), `CMP_SEAS_TRIG` for the trigonometric specification (14) and `CMP_SEAS_HS` for the Harrison and Stevens specification (15). The model may contain ten different cycle components. *SsfPack* will not verify whether the parameters of the different cycles are the same. If the first column of `mStsm` contains a string which have been identified earlier (from top to bottom), the corresponding component will not be considered in the state space formulation. The function `GetSsfStsm` returns three pointers (addresses) to the matrices Φ, Ω and Σ.

Example The following example outputs the relevant state space matrices for a basic structural time series model with trend (including slope), dummy seasonal with $s = 3$ and irregular.

Ox code of `ssfstsm.ox`:

```
#include <oxstd.h>
#include "SsfPack.h"

main()
{
 decl mPhi, mOmega, mSigma;

 // stuctural time series model
 GetSsfStsm(
 <CMP_IRREG,      1.0, 0, 0;
  CMP_LEVEL,      0.5, 0, 0;
  CMP_SEAS_DUMMY, 0.2, 3, 0;
  CMP_SLOPE,      0.1, 0, 0>, &mPhi, &mOmega, &mSigma);

 // print state space
 print("Phi", mPhi);
 print("Omega", mOmega);
 print("Sigma", mSigma);
}
```

Ox output:

```
Phi
       1.0000       1.0000      0.00000      0.00000
      0.00000       1.0000      0.00000      0.00000
      0.00000      0.00000      -1.0000      -1.0000
      0.00000      0.00000       1.0000      0.00000
       1.0000      0.00000       1.0000      0.00000
Omega
      0.25000      0.00000      0.00000      0.00000      0.00000
      0.00000     0.010000      0.00000      0.00000      0.00000
      0.00000      0.00000     0.040000      0.00000      0.00000
      0.00000      0.00000      0.00000      0.00000      0.00000
      0.00000      0.00000      0.00000      0.00000       1.0000
Sigma
      -1.0000      0.00000      0.00000      0.00000
      0.00000      -1.0000      0.00000      0.00000
```

```
0.00000     0.00000     -1.0000     0.00000
0.00000     0.00000     0.00000     -1.0000
0.00000     0.00000     0.00000     0.00000
```

3.3 Regression models

The regression model can also be represented as a state space model. The Kalman filter for the regression model in state space form is equivalent to the 'recursive least squares' algorithm for the standard regression model; see Harvey (1993, section 4.5). The univariate multiple regression model $y_t = X_t\beta + \xi_t$ with $\xi_t \sim \text{NID}\left(0, \sigma^2 I\right)$ and the $k \times 1$ vector of coefficients β, for $t = 1, \ldots, n$, is in state space given by

$$\alpha_{t+1} = \alpha_t, \qquad y_t = X_t\alpha_t + G_t\varepsilon_t, \qquad t = 1, \ldots, n,$$

so that the system matrices are set to $T_t = I$, $Z_t = X_t$, $G_t = \sigma I$ and $H_t = 0$. The vector of coefficients β is fixed and unknown so that the initial conditions are $\alpha_1 \sim \text{N}\left(0, \kappa I\right)$ where κ is large. The regression model enforces a time-varying state space due to the measurement equation. Time-varying regression coefficients may be introduced by setting H_t such that $H_t \neq 0$, for $t = 1, \ldots, n$.

SsfPack implementation The *SsfPack* routine `GetSsfReg` provides the time-varying state space structure for a univariate (single equation) regression model. The function call is

```
GetSsfReg(mX, &mPhi, &mOmega, &mSigma, &mJ_Phi);
```

where $(k \times n)$ matrix `mX` is a data matrix containing the explanatory variables. The function returns three pointers (addresses) to the composite matrices Φ, Ω and Σ and to the index matrix J_Φ; see section 2. The index matrix J_Φ refers to the inputted data matrix `mX`. The structure of the output matrices is clearified in the example below

Example The following example outputs the relevant state space matrices for a standard regression model with three explanatory variables.

Ox code of `ssfreg.ox`:

```
#include <oxstd.h>
#include "SsfPack.h"

main()
{
 decl mX, mPhi, mOmega, mSigma, mJ_Phi;
 mX = rann(3,20);
 // regression model in state space
 GetSsfReg(mX, &mPhi, &mOmega, &mSigma, &mJ_Phi);

 // print state space
 print("Phi", mPhi);
 print("Time-varying index for Phi", mJ_Phi);
 print("Omega", mOmega);
 print("Sigma", mSigma);
}
```

Ox output:

```
Phi
       1.0000      0.00000      0.00000
      0.00000       1.0000      0.00000
      0.00000      0.00000       1.0000
      0.00000      0.00000      0.00000
Time-varying index for Phi
   -1   -1   -1
   -1   -1   -1
   -1   -1   -1
    0    1    2
Omega
      0.00000      0.00000      0.00000      0.00000
      0.00000      0.00000      0.00000      0.00000
      0.00000      0.00000      0.00000      0.00000
      0.00000      0.00000      0.00000       1.0000
Sigma
      -1.0000      0.00000      0.00000
      0.00000      -1.0000      0.00000
      0.00000      0.00000      -1.0000
      0.00000      0.00000      0.00000
```

3.4 Nonparametric cubic spline models

Suppose we work with a continuous variable t for which associated vector observations $x(t)$ are made at points $\tau_1, ..., \tau_n$; see the work by Bergstrom (1984). Define $\delta_t = \tau_t - \tau_{t-1}$ as the gap between observations. A possible representation for such observations is the zero mean multivariate Ornstein-Uhlenbeck process

$$dx(t) = \phi x(t)\, dt + RdW(t), \tag{17}$$

where $W(t)$ is Brownian motion and so $dW(t)$ is normally distributed with $\mathrm{N}(0, \Lambda)$.

Let us now focus on using smoothing spline techniques for estimating a signal $\mu(t)$ from univariate observations $y(t)$ via the relationship

$$y(t) = \mu(t) + \psi(t),$$

where $\psi(t)$ is a stationary error process. The task at hand is to find a curve which minimizes $\sum_{i=1}^{n} \{y(\tau_t) - \mu(\tau_i)\}^2$ subject to the function $\mu(t)$ being 'smooth'. The common approach is to select the fitted $\widehat{\mu}(t)$ by minimising the penalised likelihood, that is

$$\sum_{i=1}^{n} \{y(\tau_i) - \mu(\tau_t)\}^2 + q \int \left\{ \frac{\partial^2 \mu(t)}{\partial t^2} \right\}^2 dt, \tag{18}$$

for a given value of q; see Kohn and Ansley (1987), Hastie and Tibshirani (1990) and Green and Silverman (1994).

The so-called cubic spline model puts $d\mu(t) = \beta(t)\, dt$, where $d\beta(t) = d\zeta(t)$ and $\zeta(t)$ is Brownian motion with variance of $\sigma_\zeta^2 t$. Thus $d^2\mu(t) = d\zeta(t)\, dt$, and so the log density of this process is, ignoring constants,

$$-\frac{1}{2\sigma_\zeta^2} \int \left\{ \frac{d\zeta(t)}{dt} \right\}^2 dt = -\frac{1}{2\sigma_\zeta^2} \int \left\{ \frac{\partial^2 \mu(t)}{\partial t^2} \right\}^2 dt.$$

It can be shown that the penalty function in (18) is the same as the log density from a continuous time Gaussian smooth trend model of the form (17) where

$$R = I, \qquad \phi = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, \qquad \Lambda = \begin{pmatrix} 0 & 1 \\ 0 & \sigma_\zeta^2 t \end{pmatrix}.$$

Taking the continuous time process $\mu(t)$ at discrete intervals, leads us to the following exact discrete time model for $\mu(\tau_{t+1})$ when $\delta_t = \tau_{t+1} - \tau_t$

$$\begin{array}{rcl} \mu(\tau_{t+1}) & = & \mu(\tau_t) + \delta_t \beta(\tau_t) + \eta(\tau_t), \\ \beta(\tau_{t+1}) & = & \beta(\tau_t) + \zeta(\tau_t), \end{array} \tag{19}$$

where

$$\begin{pmatrix} \eta(\tau_t) \\ \zeta(\tau_t) \end{pmatrix} \sim \text{NID}\left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \sigma_\zeta^2 \delta_t \begin{pmatrix} \frac{1}{3}\delta_t^2 & \frac{1}{2}\delta_t \\ \frac{1}{2}\delta_t & 1 \end{pmatrix} \right\}.$$

This can be combined with the more straightforward measurement $y(\tau_t) = \mu(\tau_t) + \varepsilon(\tau_t)$, where $\varepsilon(\tau_t) \sim \text{NID}(0, \sigma_\varepsilon^2)$, to give the model which has a joint density which is the same as the penalized likelihood (18) with signal-to-noise ratio $q = \sigma_\zeta^2/\sigma_\varepsilon^2$. Hence the usual state space framework with

$$\Phi_t = \begin{pmatrix} 1 & \delta_t \\ 0 & 1 \\ 1 & 0 \end{pmatrix}, \qquad \Omega_t = \begin{pmatrix} q\delta_t^3/3 & q\delta_t^2/2 & 0 \\ q\delta_t^2/2 & q\delta_t & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

can be used for filtering, smoothing and prediction.

SsfPack implementation The *SsfPack* routine `GetSsfSpline` provides the time-varying state space structure for the cubic spline model (19). The function call is

```
GetSsfSpline(dq, mDelta, &mPhi, &mOmega, &mJ_Phi, &mJ_Omega, &mX);
```

where `dq` is the signal-to-noise ratio q and `mDelta` is the $1 \times n$ data matrix with δ_t $(t = 1, \ldots, n)$. The routine returns the state space matrices Φ and Ω together with

$$\texttt{mJ\_Phi} = \begin{pmatrix} -1 & 0 \\ -1 & -1 \\ -1 & -1 \end{pmatrix}, \quad \texttt{mJ\_Omega} = \begin{pmatrix} 3 & 2 & -1 \\ 2 & 1 & -1 \\ -1 & -1 & -1 \end{pmatrix}, \quad \texttt{mX} = \begin{pmatrix} \delta_1 & \ldots & \delta_n \\ q\delta_1 & \ldots & q\delta_n \\ q\delta_1^2/2 & \ldots & q\delta_n^2/2 \\ q\delta_1^3/3 & \ldots & q\delta_n^3/3 \end{pmatrix}.$$

Example The following example outputs the relevant state space matrices for the non-parametric cubic spline model with $q = 0.2$.

Ox code of `ssfspl.ox`:

```
#include <oxstd.h>
#include "SsfPack.h"

main()
{
 decl mDelta, mPhi, mOmega, mJ_Phi, mJ_Omega, mX;

 // cubic spline model with q = 0.2
 mDelta = <1,2,4,3,5,3,3,2>;
 GetSsfSpline(0.2, mDelta, &mPhi, &mOmega, &mJ_Phi, &mJ_Omega, &mX);
```

```
 // print state space
 print("Phi", mPhi);
 print("Time-varying index for Phi", mJ_Phi);
 print("Omega", mOmega);
 print("Time-varying index for Omega", mJ_Omega);
 print("Data matrix", mX');
}
```

Ox output:

```
Phi
       1.0000      0.00000
      0.00000       1.0000
       1.0000      0.00000
Time-varying index for Phi
   -1    0
   -1   -1
   -1   -1
Omega
      0.00000      0.00000      0.00000
      0.00000      0.00000      0.00000
      0.00000      0.00000       1.0000
Time-varying index for Omega
    3    2   -1
    2    1   -1
   -1   -1   -1
Data matrix
       1.0000      0.20000      0.10000     0.066667
       2.0000      0.40000      0.40000      0.53333
       4.0000      0.80000       1.6000       4.2667
       3.0000      0.60000      0.90000       1.8000
       5.0000       1.0000       2.5000       8.3333
       3.0000      0.60000      0.90000       1.8000
       3.0000      0.60000      0.90000       1.8000
       2.0000      0.40000      0.40000      0.53333
```

4 Review of algorithms: the basic functions

4.1 State space recursion

To generate samples from the unconditional distribution implied by a statistical model in state space form or to generate artificial data sets, we use the state space form (4) as a recursive set of equations. For a given set of errors $(H_t', G_t')' \varepsilon_t = u_t = u_t^{(\cdot)}$ $(t = 1, \ldots, n)$, actual values for $\alpha_{t+1}^{(i)}$ and $y_t^{(i)}$ are recursively calculated by

$$\begin{pmatrix} \alpha_{t+1}^{(\cdot)} \\ y_t^{(\cdot)} \end{pmatrix} = \delta_t + \Phi_t \alpha_t^{(\cdot)} + u_t^{(\cdot)}, \qquad t = 1, \ldots, n, \tag{20}$$

with the initialization $\alpha_1^{(\cdot)} = a + P u_0^{(\cdot)}$ where vectors a and $u_0^{(\cdot)}$ and matrix P are given. The quantities a and P must be placed in Σ as given by (8); see section 2.1. Note that the role of P for the initial state vector $\alpha_1^{(\cdot)}$ is different compared to the initialisation of (4).

SsfPack implementation The *SsfPack* function `SsfRecursion` calls the recursion (20) for a given sample of $u_t^{(\cdot)}$ $(t = 0, \ldots, n)$. The recursion is called by:

```
mD = SsfRecursion(mR, {Ssf});
```

where `mR` is the $m + N \times n + 1$ data matrix with structure

$$\texttt{mR} = \begin{pmatrix} u_0^{(\cdot)} & u_1^{(\cdot)} & \ldots & u_n^{(\cdot)} \end{pmatrix}.$$

The value -9999.99 is not recognised as a missing value within `mR`. The input sequence {`Ssf`} is discussed in section 2.3. The matrix Ω does not play a role in this routine but to be consistent with other *SsfPack* routines, it must be inputted as an argument anyway. The matrix Σ may play a slightly different role compared to other *SsfPack* routines. The function `SsfRecursion` returns the $m + N \times n + 1$ data matrix

$$\texttt{mD} = \begin{pmatrix} \alpha_1^{(\cdot)} & \alpha_2^{(\cdot)} & \ldots & \alpha_{n+1}^{(\cdot)} \\ 0 & y_1^{(\cdot)} & & y_n^{(\cdot)} \end{pmatrix}.$$

Example The following *Ox* program generates artificial data from the local linear trend model (6) with $\sigma_\eta^2 = 0$, $\sigma_\zeta^2 = 0.1$ and $\sigma_\xi^2 = 1$. The *Ox* function `rann` produces a matrix of standard normal random deviates. The initial state vector $\alpha_1 = (\mu_1, \beta_1)'$ is set equal to $(5, 2)'$.

Ox code of `ssfrec.ox`:

```
#include <oxstd.h>
#include "ssfpack.h"
main()
{
 decl mPhi, mOmega, mSigma, mR, mD;
 mPhi = <1,1;0,1;1,0>;
 mOmega = diag(<0,0.1,1>);
 mSigma = <0,0;0,0;5,2>;

 mR = sqrt(mOmega) * rann(3, 8);
 mD = SsfRecursion(mR, mPhi, mOmega, mSigma);

 print("Errors", mR');
 print("Monte Carlo simulation", mD');
}
```

Ox output:

```
Errors
      0.00000      0.18097     -0.34213
      0.00000    -0.065792      0.53748
      0.00000     -0.18757     -0.89710
      0.00000      0.11576      0.65835
      0.00000      0.34257      0.20503
      0.00000      0.47684       1.5993
      0.00000     -0.15472      0.31418
      0.00000     -0.35452     -0.77329
Monte Carlo simulation
       5.0000       2.0000      0.00000
```

```
7.0000      1.9342      5.5375
8.9342      1.7466      6.1029
10.681      1.8624      9.5926
12.543      2.2050      10.886
14.748      2.6818      14.143
17.430      2.5271      15.062
19.957      2.1726      16.657
```

4.2 Kalman filter

The Kalman filter is a recursive algorithm for the evaluation of moments of the normal distribution of state vector α_{t+1} conditional on the data set $Y_t = \{y_1, \ldots, y_t\}$, that is

$$a_{t+1|t} = \mathrm{E}\left(\alpha_{t+1}|Y_t\right), \qquad P_{t+1|t} = \mathrm{cov}\left(\alpha_{t+1}|Y_t\right),$$

for $t = 1, \ldots, n$; see Anderson and Moore (1979, page 36) and Harvey (1989, page 104). The Kalman filter is given by

$$\begin{aligned} v_t &= y_t - c_t - Z_t a_{t|t-1}, \\ F_t &= Z_t P_t Z_t' + G_t G_t', \\ K_t &= \left(T_t P_t Z_t' + H_t G_t'\right) F_t^{-1}, \\ a_{t+1} &= d_t + T_t a_t + K_t v_t, \\ P_{t+1} &= T_t P_t T_t' + H_t H_t' - K_t F_t K_t', \qquad t = 1, \ldots, n, \end{aligned} \tag{21}$$

where $a_1 = a$ and $P_1 = P_* + \kappa P_\infty$ with $\kappa = 10^7$.

The *SsfPack* computer program for the Kalman filter is written in a computationally efficient way. The steps are given by

(i) Set $t = 1$, $a_1 = a$ and $P_1 = P_* + 10^7 P_\infty$.

(ii) Calculate:

$$\begin{pmatrix} \bar{a}_{t+1} \\ \hat{y}_t \end{pmatrix} = \delta_t + \Phi_t a_t, \qquad \begin{pmatrix} \bar{P}_{t+1} & M_t \\ M_t' & F_t \end{pmatrix} = \Phi_t P_t \Phi_t' + \Omega_t, \qquad K_t = M_t F_t^{-1},$$

where δ_t, Φ_t and Ω_t are defined in (5).

(iii) Update:

$$v_t = y_t - \hat{y}_t, \qquad a_{t+1} = \bar{a}_{t+1} + K_t v_t, \qquad P_{t+1} = \bar{P}_{t+1} - K_t M_t'.$$

(iv) Set $t = t + 1$ and goto (ii) until $t = n$.

The program stops with an error message when $|F_t| \leq 0$ or when no computer memory is available.

SsfPack implementation The *SsfPack* function `KalmanFil` calls the Kalman filter and returns the output v_t, F_t and K_t $(t = 1, \ldots, n)$ as a data matrix. The Kalman filter function call is

```
mKF = KalmanFil(mY, {Ssf});
```

where `mY` is a $N \times n$ data matrix. The input sequence {`Ssf`} is discussed in section 2.3. In the simplest time-invariant case, {`Ssf`} is replaced by

`mPhi`, `mOmega`

see the example below. The Kalman filter is available for univariate and multivariate state space models but the call is the same. The row dimension of data matrix `mY` determines whether the univariate or the multivariate Kalman filter is used. The function returns the data matrix `mKF` with dimension $k \times T$ where

$$k = N\left(1 + m + \frac{N+1}{2}\right).$$

For univariate models in state space form, the returned storage matrix is simply the $m + 2 \times n$ matrix

$$\texttt{mKF} = \begin{pmatrix} v_1 & \dots & v_n \\ K_1 & \dots & K_n \\ F_1^{-1} & \dots & F_n^{-1} \end{pmatrix}.$$

In multivariate cases, the returned data matrix is organized as

$$\texttt{mKF} = \begin{bmatrix} v_1 & \dots & v_T \\ (K_1)_{*1} & \dots & (K_T)_{*1} \\ \left(F_1^{-1}\right)_{11} & \dots & \left(F_T^{-1}\right)_{11} \\ (K_1)_{*2} & \dots & (K_T)_{*2} \\ \left(F_1^{-1}\right)_{12} & \dots & \left(F_T^{-1}\right)_{12} \\ \left(F_1^{-1}\right)_{22} & \dots & \left(F_T^{-1}\right)_{22} \\ \vdots & & \vdots \\ (K_1)_{*N} & \dots & (K_T)_{*N} \\ \left(F_1^{-1}\right)_{*N} & & \left(F_T^{-1}\right)_{*N} \end{bmatrix},$$

where M_{*i} refers to the i-th column of matrix M and the element M_{ij} refers to the (i, j) element of matrix M. The storage of the inverse matrix F_t^{-1} is limited to its upper triangular part.

Example The following *Ox* program applies the Kalman filter to the local linear trend model (6) with $\sigma_\eta^2 = 0$, $\sigma_\zeta^2 = 0.1$ and $\sigma_\xi^2 = 1$.

Ox code of `ssfkf.ox`:

```
#include <oxstd.h>
#include "ssfpack.h"
main()
{
 decl mY, mPhi, mOmega, mKF;
 mY = <1,9,2,5,8,4,6,7,3>;
 mPhi = <1,1;0,1;1,0>;
 mOmega = diag(<0.0,0.1,1>);
 mKF = KalmanFil(mY, mPhi, mOmega);
 print("Kalman filter output", mKF');
}
```

Ox output:

```
Kalman filter output
       1.0000      0.99999      0.00000  9.9999e-006
       8.0000       2.0000      0.99998  9.9998e-006
      -15.000       1.3443      0.50819      0.16394
      0.16392       1.0382      0.32579      0.28760
       2.6167      0.88282      0.25053      0.36771
      -4.1238      0.80786      0.22140      0.41353
      0.12163      0.77684      0.21246      0.43562
      0.85411      0.76694      0.21099      0.44405
      -3.9998      0.76497      0.21132      0.44635
```

4.3 Moment smoothing

The Kalman filter is a forwards recursion which evaluates one-step ahead estimators. The associated moment smoothing algorithm is a backwards recursion which evaluates the mean and variance of specific conditional distributions given the data set $Y_n = \{y_1, \ldots, y_n\}$ using the output of the Kalman filter; see Anderson and Moore (1979), Kohn and Ansley (1989), de Jong (1988b), de Jong (1989) and Koopman (1993). The backwards recursions are given by

$$\begin{aligned} e_t &= F_t^{-1} v_t - K_t' r_t, \\ D_t &= F_t^{-1} + K_t' N_t K_t, \\ r_{t-1} &= Z_t' F_t^{-1} v_t + L_t' r_t, \\ N_{t-1} &= Z_t' F_t^{-1} Z_t + L_t' N_t L_t, \qquad t = n, \ldots, 1, \end{aligned} \tag{22}$$

with $L_t = T_t - K_t Z_t$ and with the initialisations $r_n = 0$ and $N_n = 0$.

4.3.1 Disturbance smoothing

The moment smoother (22) generates quantities from which different kinds of estimators can be obtained. For example, it can be shown that the mean and variance of the conditional density $f(\varepsilon_t|Y_n)$ is given by, respectively,

$$\begin{aligned} \mathrm{E}(\varepsilon_t|Y_n) &= G_t' e_t + H_t' r_t, \\ \mathrm{var}(\varepsilon_t|Y_n) &= G_t' (D_t G_t - K_t' N_t H_t) + H_t' (N_t H_t - N_t K_t G_t), \end{aligned}$$

and expressions for $\mathrm{E}(u_t|Y_n)$ and $\mathrm{var}(u_t|Y_n)$, where u_t is defined in (4), follow directly from this. It is also clear that, when $H_t G_t' = 0$,

$$\begin{aligned} \mathrm{E}(G_t\varepsilon_t|Y_n) &= G_t G_t' e_t, \\ \mathrm{var}(G_t\varepsilon_t|Y_n) &= G_t G_t' D_t G_t G_t', \\ \mathrm{E}(H_t\varepsilon_t|Y_n) &= H_t H_t' r_t, \\ \mathrm{var}(H_t\varepsilon_t|Y_n) &= H_t H_t' N_t H_t H_t', \end{aligned}$$

for $t = 1, \ldots, n$; see Koopman (1993) for more general results.

The *SsfPack* computer program for the moment smoother is written in a similar way as the Kalman filter. The steps are given by

(i) Set $t = n$, $r_n = 0$ and $N_n = 0$.

(ii) Calculate:

$$\bar{r}_t = \begin{pmatrix} r_t \\ e_t = F_t^{-1} v_t - K_t' r_t \end{pmatrix}, \qquad \bar{N}_t = \begin{pmatrix} N_t & -N_t K_t \\ -K_t' N_t & D_t = F_t^{-1} + K_t' N_t K_t \end{pmatrix}.$$

(iii) Update:

$$r_{t-1} = \Phi_t' \bar{r}_t, \qquad N_{t-1} = \Phi_t' \bar{N}_t \Phi_t.$$

where Φ_t is defined in (5).

(iv) Set $t = t - 1$ and goto (ii) until $t = 1$.

The program stops with an error message when no computer memory is available. The vector δ_t and the matrix Ω_t do not play a role in the basic smoothing recursions. Finally, it should be noted that the smoothed estimator $\hat{u}_t = \mathrm{E}(u_t|Y_n)$, where u_t is from (4), is simply obtained by $\Omega_t \bar{r}_t$ and the corresponding variance matrix is $\mathrm{var}(u_t|Y_n) = \Omega_t \bar{N}_t \Omega_t$; see section 5.2.5 for further details.

4.3.2 Quick state smoothing

The generated output from the basic smoothing recursions can also be used to obtain $\hat{\alpha}_t = \mathrm{E}(\alpha_t|Y_n)$, that is the smoothed estimator of the state vector, using the recursion

$$\hat{\alpha}_{t+1} = d_t + T_t \hat{\alpha}_t + H_t \hat{\varepsilon}_t, \qquad t = 1, \ldots, n,$$

with $\hat{\alpha}_1 = a + P r_0$ and $\hat{\varepsilon}_t = \mathrm{E}(\varepsilon_t|Y_n) = G_t' e_t + H_t' r_t$; see Koopman (1993) for details. This simple recursion is similar to the state space recursion (20). A further discussion on state smoothing is found in sections 5.2.3 and 5.3.1. This method of state smoothing is illustrated in the example below using the *SsfPack* function `SsfRecursion`.

SsfPack implementation The *SsfPack* function `KalmanSmo` calls the moment smoother and stores the output e_t, D_t, r_{t-1} and N_{t-1} for $t = 1, \ldots, n$, into a data matrix. The Kalman smoother algorithm is called by

```
mKS = KalmanSmo(mKF, {Ssf});
```

where the sequence {`Ssf`} is discussed in section 2.3. Matrix `mKF` is the data matrix which is produced by the function `KalmanFil` using the same state space form as implied by {`Ssf`}. Matrix `mKS` is a data matrix of dimension $k \times n + 1$, with

$$k = 2(m + N),$$

and the structure of the matrix is

$$\texttt{mKS} = \left\{ \begin{array}{cccc} r_0 & r_1 & \ldots & r_n \\ 0 & e_1 & \ldots & e_n \\ \mathrm{diag}(N_0) & \mathrm{diag}(N_1) & \ldots & \mathrm{diag}(N_n) \\ 0 & \mathrm{diag}(D_1) & \ldots & \mathrm{diag}(D_n) \end{array} \right\},$$

where $\mathrm{diag}(A)$ vectorizes the diagonal elements of the square matrix A. The output matrix is organised in this way partly because the first two blocks of rows of `mKS` can be used as the input matrix `mR` of the *SsfPack* function `SsfRecursion`; see section 4.1. More elaborate and more 'easy-to-use' functions for moment smoothing of the disturbance and state vector are given in section 5.3.

Example The following *Ox* program applies the Kalman filter smoother to the local linear trend model (6) with $\sigma_\eta^2 = 0$, $\sigma_\zeta^2 = 0.1$ and $\sigma_\xi^2 = 1$. It outputs the matrix `mKS` and it also outputs the smoothed disturbances and states.

Ox code of `ssfsmo.ox`:

```
#include <oxstd.h>
#include "ssfpack.h"
main()
{
 decl mY, mPhi, mOmega, mKF, mKS;
 mY = <1,9,2,5,8,4,6,7,3>;
 mPhi = <1,1;0,1;1,0>;
 mOmega = diag(<0.0,0.1,1>);
 mKF = KalmanFil(mY, mPhi, mOmega);
 mKS = KalmanSmo(mKF, mPhi, mOmega);
 print("Basic smoother output", mKS');

 mKS[0:2][1:9] = mOmega * mKS[0:2][1:9];
 print("Smoothed disturbances", mKS[0:2][1:9]');

 mKS = SsfRecursion(mKS[0:2][], mPhi, mOmega);
 print("Smoothed states", mKS');
}
```

Ox output:

```
Basic smoother output
  3.6106e-005  7.6158e-006      0.00000  9.9999e-006  1.0000e-005      0.00000
       2.6107      -2.6107      -2.6106      0.44636      0.44636      0.44636
      -2.0171     -0.59351       4.6278      0.46830       1.1227      0.70660
      0.85557      -1.4491      -2.8727      0.46923       1.6135      0.77271
       1.1694      -2.6185     -0.31387      0.48497       1.7810      0.77866
      -1.2204      -1.3981       2.3899      0.48497       1.6135      0.77675
      0.42407      -1.8221      -1.6445      0.46923       1.1227      0.77866
    -0.036817      -1.7853      0.46089      0.46830      0.44635      0.77271
      -1.7853      0.00000       1.7485      0.44635      0.00000      0.70660
      0.00000      0.00000      -1.7853      0.00000      0.00000      0.44635
Smoothed disturbances
      0.00000     -0.26107      -2.6106
      0.00000    -0.059351       4.6278
      0.00000     -0.14491      -2.8727
      0.00000     -0.26185     -0.31387
      0.00000     -0.13981       2.3899
      0.00000     -0.18221      -1.6445
      0.00000     -0.17853      0.46089
      0.00000      0.00000       1.7485
      0.00000      0.00000      -1.7853
Smoothed states
       3.6106      0.76158      0.00000
       4.3722      0.50051       1.0000
       4.8727      0.44116       9.0000
       5.3139      0.29625       2.0000
```

```
5.6101      0.034399      5.0000
5.6445      -0.10541      8.0000
5.5391      -0.28762      4.0000
5.2515      -0.46616      6.0000
4.7853      -0.46616      7.0000
4.3192      -0.46616      3.0000
```

4.4 Simulation smoother

4.4.1 Disturbance simulation smoothing

The simulation smoother is developed by de Jong and Shephard (1995) and it allows drawing random numbers from the multivariate conditional Gaussian density of

$$\tilde{u} = (\tilde{u}_1', ..., \tilde{u}_n')', \qquad \text{where } \tilde{u} \sim \Gamma u | Y_n, \qquad t = 1, \ldots, n, \tag{23}$$

with $u = (u_1', ..., u_n')'$ and u_t as defined in (5). The $m + N \times m + N$ diagonal selection matrix Γ consists of unity and zero values on the diagonal. It is introduced to avoid degeneracies in sampling and to be able to select samples which are required (computational efficiency). For example, when we consider the local linear trend model (6) and we wish to generate samples from the multivariate conditional density of the disturbance ζ_t $(t = 1, \ldots, n)$, then matrix $\Gamma = \text{diag}\begin{pmatrix} 0 & 1 & 0 \end{pmatrix}$. When we wish to generate samples from the multivariate joint conditional density of η_t and ζ_t $(t = 1, \ldots, n)$ for model (6), matrix Γ is given by

$$\Gamma = \text{diag}\begin{pmatrix} 1 & 1 & 0 \end{pmatrix}.$$

Finally, generating conditional samples for $G_t\varepsilon_t$ of the state space form, which for univariate cases requires

$$\Gamma = \text{diag}\begin{pmatrix} 0 & \ldots & 0 & 1 \end{pmatrix},$$

also implicitly produces samples from $f(\theta_t|Y_n)$, with signal $\theta_t = c_t + Z_t\alpha_t$, since $y_t - G_t\varepsilon_t = \theta_t$ $(t = 1, \ldots, n)$. The simulation algorithms use the $s \times m + N$ zero-unity matrix Γ^* which is the same as Γ but where the zero rows are deleted from Γ. For example,

$$\Gamma = \text{diag}\begin{pmatrix} 1 & 0 & 1 \end{pmatrix} \text{ becomes } \Gamma^* = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ and } \Gamma^{*\prime}\Gamma^* = \Gamma,$$

with $s = 2$.

The simulation smoother is a backwards recursion and requires the output of the Kalman filter. The equations are given by

$$\begin{aligned}
C_t &= \Gamma^* \begin{pmatrix} H_t \\ G_t \end{pmatrix} \left(I - G_t' F_t^{-1} G_t - J_t' N_t J_t\right) \begin{pmatrix} H_t \\ G_t \end{pmatrix}' \Gamma^{*\prime}, \\
W_t &= \Gamma^* \begin{pmatrix} H_t \\ G_t \end{pmatrix} \left(G_t' F_t^{-1} Z_t + J_t' N_t L_t\right), \quad \xi_t \sim \text{N}(0, C_t), \\
r_{t-1} &= Z_t' F_t^{-1} v_t - W_t' C_t^{-1} \xi_t + L_t' r_t, \\
N_{t-1} &= Z_t' F_t^{-1} Z_t + W_t' C_t^{-1} W_t + L_t' N_t L_t, \qquad t = n, \ldots, 1,
\end{aligned} \tag{24}$$

where $L_t = T_t - K_t Z_t$ and $J_t = H_t - K_t G_t$. The initialization is $r_n = 0$ and $N_n = 0$. The notation for r_t and N_t is the same as for the moment smoother (22) since the nature of both recursions is very similar. However, their actual values are different. It can be shown that

$$\tilde{u}_t = \Gamma^{*\prime} \left\{ \Gamma^* \begin{pmatrix} H_t \\ G_t \end{pmatrix} \left(G_t' F_t^{-1} v_t + J_t' r_t\right) + \xi_t \right\},$$

is a draw as indicated by (23). The selection matrix Γ must be chosen so that $\Gamma^*\Omega_t\Gamma^{*\prime}$ is nonsingular and $r\left(\Gamma^*\Omega_t\Gamma^{*\prime}\right) \leq m$; the latter condition is required to avoid degenerate sampling and matrix C_t being singular. These conditions are not sufficient to avoid degenerate sampling; see de Jong and Shephard (1995). However, the conditions firmly exclude the special case of $\Gamma = I_{m+N}$.

The structure of the *SsfPack* computer program for the simulation smoother is similar to the moment smoother. In the following we introduce the matrix $A_t = C_t^{-\frac{1}{2}} W_t$. The steps of the program are given by

(i) Set $t = n$, $r_n = 0$ and $N_n = 0$.

(ii) Calculate:

$$\bar{r}_t = \begin{pmatrix} r_t \\ e_t = F_t^{-1} v_t - K_t' r_t \end{pmatrix}, \qquad \bar{N}_t = \begin{pmatrix} N_t & -N_t K_t \\ -K_t' N_t & D_t = F_t^{-1} + K_t' N_t K_t \end{pmatrix}.$$

(iii) Calculate:

$$C_t = \Gamma^* \left(\Omega_t - \Omega_t \bar{N}_t \Omega_t\right) \Gamma^{*\prime},$$

apply a Choleski decomposition to C_t such that

$$C_t = B_t B_t'.$$

and solve recursively:

$$B_t A_t = \Gamma^* \Omega_t \bar{N}_t,$$

with respect to A_t. The matrices Φ_t and Ω_t are defined in (5).

(iv) Update:

$$r_{t-1} = \Phi_t' \left(\bar{r}_t - A_t' \pi_t\right), \qquad N_{t-1} = \Phi_t' \left(\bar{N}_t + A_t' A_t\right) \Phi_t.$$

with $\pi_t \sim \mathrm{N}\left(0, I_s\right)$.

(v) Set $t = t - 1$ and goto (ii) until $t = 1$.

The program stops with an error message when the Choleski decomposition for C_t fails or when no computer memory is available. The vector δ_t does not play a role in simulation smoothing.

A draw from the Gaussian density for (23) is obtained by

$$\tilde{u}_t = \Gamma^{*\prime} \left(\Gamma^* \Omega_t \bar{r}_t + B_t \pi_t\right), \qquad t = 1, \ldots, n.$$

When M different samples are required from the same model and conditional on the same data-set Y_n, the simulation smoother can be simplified to generate such multiple draws. In such circumstances, the matrices A_t and B_t (the so-called weights) should be stored so that repeatedly samples ($i = 1, \ldots, M$) can be generated via the recursion

$$\begin{array}{rcll} r_{t-1} & = & \Phi_t' \left(\bar{r}_t - A_t' \pi_t^{(i)}\right), & \pi_t^{(i)} \sim \mathrm{N}\left(0, I\right), \\ \tilde{u}_t^{(i)} & = & \Gamma^{*\prime} \left(\Gamma^* \Omega_t \bar{r}_t + B_t \pi_t^{(i)}\right), & t = n, \ldots, 1, \end{array} \tag{25}$$

which is computationally very efficient. Note that $\bar{r}_t = (r_t', e_t')'$; see step (ii) of the algorithm. When $s = 1$, the storage of A_t and B_t ($t = 1, \ldots, n$) requires a matrix of dimension $1 + p + N \times n$.

4.4.2 State simulation smoothing

It is mentioned earlier that generated samples from the simulation smoother (24) can be used to get simulation samples from the multivariate density $f(\theta|Y_n)$, where $\theta = (\theta_1', \ldots, \theta_n')'$ and $\theta_t = c_t + Z_t\alpha_t$, by setting Γ such that $\Gamma^*(H_t', G_t')' = G_t^*$, for $t = 1, \ldots, n$, and where G_t^* is equal to G_t but without the zero rows (in the same spirit of Γ and Γ^*). This follows from the identity $\theta_t = y_t - G_t\varepsilon_t$. In a similar way, it is also possible to obtain samples from the multivariate density $f(\alpha|Y_n)$, where $\alpha = (\alpha_1', \ldots, \alpha_n')'$, by applying the simulation smoother (24) with Γ such that $\Gamma^*(H_t', G_t')' = H_t^*$, for $t = 1, \ldots, n$, and where H_t^* is H_t but without the zero rows. Then the generated sample $\tilde{u}_t$ $(t = 1, \ldots, n)$ is inputted into the state space recursion (20) with initialisation $\alpha_1^{(\cdot)} = a + Pr_0^{(\cdot)}$; see de Jong and Shephard (1995) for details. In this way a sample from $f(\theta|Y_n)$ can also be obtained but now via the identity $\theta_t^{(\cdot)} = c_t + Z_t\alpha_t^{(\cdot)}$ (rather than $\theta_t^{(\cdot)} = y_t - \{G_t\varepsilon_t\}^{(\cdot)}$) so that this sample is consistent with the sample from $f(\alpha|Y_n)$. Note that sampling directly from $f(\alpha, \theta|Y_n)$ is not possible because of degeneracies; this matter is further discussed in section 5.4. A simple illustration is given by the example below.

SsfPack implementation The *SsfPack* function `SimSmoWgt` calls the simulation smoother, but only with respect to the quantities C_t, W_t and N_t, and it stores the output $A_t = B_t^{-1}W_t$ and B_t (note that $C_t = B_tB_t'$), for $t = 1, \ldots, n$, into a data matrix. The call is given by

```
mWgt = SimSmoWgt(mGamma, mKF, {Ssf});
```

where matrix `mGamma` is the $m + N$ diagonal 'selection' matrix Γ and matrix `mKF` is the data matrix which is produced by the function `KalmanFil` for the same state space form implied by `{Ssf}`; see section 2.3. The output matrix `mWgt` is a data matrix of dimension $k \times n$ where

$$k = s\left(m + N + \frac{s+1}{2}\right),$$

and the structure of the matrix is

$$\texttt{mWgt} = \begin{pmatrix} vec(A_1) & \ldots & vec(A_n) \\ vech(B_1) & \ldots & vech(B_n) \end{pmatrix}$$

where $vec(A)$ vectorizes matrix A and $vech(A)$ vectorizes the lower triangular part (including its diagonal) of matrix A.

The *SsfPack* function `SimSmoDraw` generates a sample from the distribution (23) which is calculated by the equations (25). This function requires the weight matrices A_t and B_t $(t = 1, \ldots, n)$. The function call is given by

```
mD = SimSmoDraw(mGamma, mPi, mWgt, mKF, {Ssf});
```

Matrix `mGamma` is the diagonal 'selection' matrix Γ, matrix `mPi` is a $s \times n$ data matrix containing the random deviates from the standard normal distribution, matrix `mWgt` is the data matrix obtained from function `SimSmoWgt`, matrix `mKF` is the data matrix returned by the function `KalmanFil`. The `SimSmoDraw` function returns the $m + N \times n + 1$ data matrix `mD` where

$$\texttt{mD} = \begin{pmatrix} \bar{r}_0^{(\cdot)} & \tilde{u}_1 & \ldots & \tilde{u}_n \end{pmatrix}.$$

where $\bar{r}_0^{(\cdot)} = \left(r_0^{(\cdot)\prime}, 0'\right)'$ and $\tilde{u}_t$ is defined in (23). Repeated samples can be generated consecutively; see example below. The output matrix `mD` is constructed such that it can be used as the input matrix `mR` for the *SsfPack* function `SsfRecursion` which enables state simulation samples; see next *Ox* example.

Example The following *Ox* program draws two samples from the multivariate conditional Gaussian density $f(\zeta|Y_n)$, with $\zeta = (\zeta_1, \ldots, \zeta_n)'$, of the local linear trend model (6) with $\sigma_\eta^2 = 0$, $\sigma_\zeta^2 = 0.1$ and $\sigma_\varepsilon^2 = 1$. This draw is also used to generate samples from the densities $f(\alpha|Y_n)$ and $f(\theta|Y_n)$. Note that Γ is selected such that $\Gamma^* (H', G')' = H$ but without the zero rows. Thus $\Gamma = \text{diag}(0, 1, 0)$ because $\sigma_\eta^2 = 0$.

Ox code of `ssfsim.ox`:

```
#include <oxstd.h>
#include "ssfpack.h"
main()
{
 decl mY, mPhi, mOmega, mKF, mGamma, mWgt, mD;
 mY = <1,9,2,5,8,4,6,7,3>;
 mPhi = <1,1;0,1;1,0>;
 mOmega = diag(<0,0.1,1>);
 mKF = KalmanFil(mY, mPhi, mOmega);

 mGamma = diag(<0,1,0>);
 mWgt = SimSmoWgt(mGamma, mKF, mPhi, mOmega);
 print("Simulation smoother weights", mWgt');

 mD = SimSmoDraw(mGamma, rann(1, 9), mWgt, mKF, mPhi, mOmega);
 print("Draw 1 for slope disturbances", mD');
 mD = SsfRecursion(mD, mPhi, mOmega);
 print("Draw 1 for state and signal", mD');

 mD = SimSmoDraw(mGamma, rann(1, 9), mWgt, mKF, mPhi, mOmega);
 print("Draw 2 for slope disturbances", mD');
 mD = SsfRecursion(mD, mPhi, mOmega);
 print("Draw 2 for state and signal", mD');
}
```

Ox output:

```
Simulation smoother weights
     -0.19091      0.19092      0.19091      0.30683
     -0.29644      0.55114     0.041741      0.28987
     -0.25945      0.78649    -0.050924      0.27934
     -0.16301      0.78553    -0.086685      0.27938
    -0.069415      0.62468    -0.095221      0.28653
    0.0022409      0.38815    -0.087744      0.29742
     0.033656      0.14441    -0.056826      0.30909
      0.00000      0.00000      0.00000      0.31623
      0.00000      0.00000      0.00000      0.31623
Draw 1 for slope disturbances
  3.4129e-005  9.4503e-006      0.00000
      0.00000     -0.45379      0.00000
      0.00000      0.12184      0.00000
      0.00000     -0.21032      0.00000
      0.00000     -0.44214      0.00000
      0.00000     -0.37313      0.00000
      0.00000     0.022659      0.00000
```

```
      0.00000      0.15776      0.00000
      0.00000      0.31542      0.00000
      0.00000      0.18097      0.00000
Draw 1 for state and signal
       3.4129      0.94503      0.00000
       4.3579      0.49124       3.4129
       4.8491      0.61307       4.3579
       5.4622      0.40276       4.8491
       5.8650    -0.039384       5.4622
       5.8256     -0.41252       5.8650
       5.4131     -0.38986       5.8256
       5.0232     -0.23210       5.4131
       4.7911     0.083326       5.0232
       4.8744      0.26430       4.7911
Draw 2 for slope disturbances
  3.7420e-005  8.1724e-006      0.00000
      0.00000     -0.34201      0.00000
      0.00000     -0.31491      0.00000
      0.00000     -0.15183      0.00000
      0.00000    -0.015562      0.00000
      0.00000      0.32872      0.00000
      0.00000     -0.30777      0.00000
      0.00000     -0.52505      0.00000
      0.00000     -0.10819      0.00000
      0.00000      0.16996      0.00000
Draw 2 for state and signal
       3.7420      0.81724      0.00000
       4.5592      0.47523       3.7420
       5.0344      0.16032       4.5592
       5.1948    0.0084960       5.0344
       5.2033   -0.0070664       5.1948
       5.1962      0.32165       5.2033
       5.5178     0.013881       5.1962
       5.5317     -0.51117       5.5178
       5.0206     -0.61936       5.5317
       4.4012     -0.44940       5.0206
```

4.5 Missing values

The algorithms of *SsfPack* can deal with missing values. When observations within the data matrix `mY` are missing, they must be given the value -9999.99. The Kalman filter recognises these values only within `mY`. A vector of observations y_t with missing elements is reduced to vector $y_t^\dagger$ without missing values and the measurement equation is adjusted accordingly. For example, the measurement equation $y_t = c_t + Z_t\alpha_t + G_t\varepsilon_t$ with

$$y_t = \begin{pmatrix} 5 \\ -9999.99 \\ 3 \\ -9999.99 \\ -9999.99 \end{pmatrix}, \quad c_t = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{pmatrix}, \quad Z_t = \begin{pmatrix} Z_{1,t} \\ Z_{2,t} \\ Z_{3,t} \\ Z_{4,t} \\ Z_{5,t} \end{pmatrix}, \quad G_t = \begin{pmatrix} G_{1,t} \\ G_{2,t} \\ G_{3,t} \\ G_{4,t} \\ G_{5,t} \end{pmatrix},$$

reduces to the measurement equation $y_t^\dagger = c_t^\dagger + Z_t^\dagger \alpha_t + G_t^\dagger \varepsilon_t$ with

$$y_t^\dagger = \begin{pmatrix} 5 \\ 3 \end{pmatrix}, \quad c_t^\dagger = \begin{pmatrix} 1 \\ 3 \end{pmatrix}, \quad Z_t^\dagger = \begin{pmatrix} Z_{1,t} \\ Z_{3,t} \end{pmatrix}, \quad G_t^\dagger = \begin{pmatrix} G_{1,t} \\ G_{3,t} \end{pmatrix}.$$

The Kalman filter smoother and the simulation smoother step at time t is based on $y_t^\dagger$ instead of y_t. When the full vector y_t is missing, for example when a single observation is missing in univariate cases, the Kalman filter reduces to a prediction step, that is

$$a_{t+1} = d_t + T_t a_t, \qquad P_{t+1} = T_t P_t T_t' + H_t H_t',$$

such that $v_t = 0$, $F_t^{-1} = 0$ and $K_t = 0$. The moment and simulation smoother deal with these specific values of v_t, F_t^{-1} and K_t without further complications.

Example The following *Ox* program applies the Kalman filter and smoother for the local linear trend model (6) with $\sigma_\eta^2 = 0$, $\sigma_\zeta^2 = 0.1$ and $\sigma_\varepsilon^2 = 1$ using data with missing entries.

Ox code of `ssfmiss.ox`:

```
#include <oxstd.h>
#include "ssfpack.h"

main()
{
 decl mY, mPhi, mOmega, mKF, mKS;
 mY = <1,-9999.99,2,5,-9999.99,4,6,-9999.99,3>;
 mPhi = <1,1;0,1;1,0>;
 mOmega = diag(<0.0,0.1,1>);
 mKF = KalmanFil(mY, mPhi, mOmega);
 mKS = KalmanSmo(mKF, mPhi, mOmega);
 print("Kalman filter output", mKF');
 print("Kalman smoother output", mKS');
}
```

Ox output:

```
Kalman filter output
       1.0000      0.99999      0.00000  9.9999e-006
      -10000.      0.00000      0.00000      0.00000
       1.0000       1.5000      0.50000  2.5000e-006
       2.5000       1.0345      0.31034      0.27586
      -10000.      0.00000      0.00000      0.00000
      -2.8621       1.0355      0.25434      0.21887
      0.82566      0.81893      0.20802      0.38909
      -10000.      0.00000      0.00000      0.00000
      -4.1181      0.95750      0.23380      0.27629
Kalman smoother output
  1.3683e-005  8.0384e-006      0.00000  9.9999e-006  1.0000e-005      0.00000
      0.36827     -0.36826     -0.36826      0.27629      0.27630      0.27630
      0.36827     -0.73654      0.00000      0.27629       1.1052      0.00000
       1.3074      -2.0439     -0.93912      0.38929       1.4647      0.68048
    -0.060121      -1.9838       1.3675      0.29058       1.1741      0.69991
    -0.060121      -1.9237      0.00000      0.29058       1.4647      0.00000
```

```
 0.35193     -2.2756     -0.41205     0.38929      1.1052     0.69991
-1.1378      -1.1378       1.4897     0.27629     0.27629     0.68048
-1.1378      0.00000      0.00000     0.27629     0.00000     0.00000
 0.00000     0.00000      -1.1378     0.00000     0.00000     0.27629
```

5 Gaussian applications: ready-to-use functions

5.1 Likelihood and score evaluation

The Kalman filter allows the computation of the Gaussian log-likelihood function via the prediction error decomposition; see Schweppe (1965), Jones (1980) and Harvey (1989). The log-likelihood function is, ignoring constants, given by

$$\begin{aligned} l &= \log p\left(y_1,\ldots,y_n;\varphi\right) = \sum_{t=1}^{n} \log p\left(y_t|y_1,\ldots,y_{t-1};\varphi\right) \\ &= \text{constant } -\frac{1}{2}\sum_{t=1}^{n}\left(\log|F_t| + v_t' F_t^{-1} v_t\right) \end{aligned} \tag{26}$$

where φ is the vector of parameters for a specific statistical model represented in state space form. The innovations v_t and its variances F_t $(t = 1,\ldots,n)$ are computed by the Kalman filter applied to the model in state space for a given vector φ.

The score vector for Gaussian models in state space form is usually evaluated numerically. Koopman and Shephard (1992) present a method to calculate the exact score for any parameter within the system matrices T, Z, H and G. A computational gain (compared to numerical evaluation) is achieved for the exact score with respect to parameters in H and G. Let the ith element of φ, that is φ_i, be associated with the time-invariant system matrix Ω of (5), then the exact score for this element is given by

$$\frac{\partial l}{\partial \varphi_i} = \frac{1}{2} tr\left(S\frac{\partial \Omega}{\partial \varphi_i}\right), \quad \text{with} \quad S = \sum_{t=1}^{n} \bar{r}_t \bar{r}_t' - \bar{N}_t, \tag{27}$$

where $\bar{r}_t$ and $\bar{N}_t$ are defined in (and calculated by) the smoothing algorithm of section 4.3.

Consider the model in state space form of the structure

$$y_t = \theta_t + G_t^{\dagger}\varepsilon_t^{\dagger}, \qquad \varepsilon_t^{\dagger} \sim \mathrm{N}\left(0, \sigma^2 I\right), \qquad \sigma^2 > 0,$$

with unknown variance σ^2, with signal $\theta_t = Z_t\alpha_t$. The state space form (1) and (2) applies but with $G_t = \sigma G_t^{\dagger}$ and $H_t = \sigma H_t^{\dagger}$. Note that at least one unknown element of $G_t^{\dagger}$ or $H_t^{\dagger}$, which is part of φ, must be set equal to a known scaling constant (usually this constant is equal to unity). For this (general) class of models, the variance σ^2 can be concentrated out of the log-likelihood function which leads to the concentrated or profile likelihood. The log-likelihood (26) is then based on the usual state space form but with G_t replaced by $G_t^{\dagger}$ and H_t replaced by $H_t^{\dagger}$ such that (26) becomes

$$l = \text{constant } -\frac{1}{2}\sum_{t=1}^{n}\left\{\log\left|\sigma^2 F_t\right| + \frac{1}{\sigma^2} v_t' F_t^{-1} v_t\right\}.$$

The maximum likelihood estimator for σ^2 requires $\partial l \;/\; \partial\sigma^2 = 0$ so that,

$$\tilde{\sigma}^2 = \frac{1}{n}\sum_{t=1}^{n} v_t' F_t^{-1} v_t. \tag{28}$$

Replacing σ^2 by its estimator $\tilde{\sigma}^2$ in the equation for l and dropping a constant, leads to the concentrated or profile log-likelihood function

$$l_c = \text{constant} \; - \frac{n}{2} \log \tilde{\sigma}^2 - \frac{1}{2} \sum_{t=1}^{n} \log |F_t| \, . \tag{29}$$

The exact score function of the concentrated or profile log-likelihood function can not be obtained as easy as in the non-concentrated case. It is more straightforward to get the score related to (29) by numerical methods.

SsfPack implementation The following *SsfPack* functions are provided for log-likelihood and score evaluation:

```
SsfLik(&dLik, &dVar, mY, {Ssf});
SsfLikConc(&dConcLik, &dVar, mY, {Ssf});
SsfLikSco(&dLik, &mSco, mY, {Ssf});
```

Each function returns two pointers (addresses) to two matrices which are either `dLik` (scalar) or `dConcLik` (scalar) or `dVar` (scalar) or `mSco` (matrix). These four quantities are defined by (26) without constant, (29) without constant, (28) and matrix S in (27), respectively.

Example The following *Ox* program calculates these different quantities for the local linear trend model (6) with $\sigma_\eta = 0$, $\sigma_\zeta = 0.1$ and $\sigma_\xi = 1$.

Ox code of `ssflik.ox`:

```
#include <oxstd.h>
#include "SsfPack.h"

main()
{
 decl mY, mPhi, mOmega, mSigma;
 decl dLik, dVar, mSco;

 mY = <1,9,2,5,8,4,6,7,3>;
 GetSsfStsm(<CMP_IRREG,      1.0, 0, 0;
             CMP_LEVEL,      0.1, 0, 0;
             CMP_SLOPE,      1.0, 0, 0>, &mPhi, &mOmega, &mSigma);

 SsfLik(&dLik, &dVar, mY, mPhi, mOmega, mSigma);
 print("\n log-likelihood ", dLik);

 SsfLikConc(&dLik, &dVar, mY, mPhi, mOmega, mSigma);
 print("\n concentrated   ", dLik);
 print("\n MLE variance   ", dVar);

 SsfLikSco(&dLik, &mSco, mY, mPhi, mOmega, mSigma);
 print("\n score vector   ", diagonal(mSco));
}
```

Ox output:

```
log-likelihood -28.298989
concentrated   -11.997636
MLE variance     6.534648
score vector
      10.889       3.3349       35.299
```

5.2 Prediction, forecasting and smoothing

5.2.1 Prediction

The Kalman filter of section 4.2 produces the one-step ahead prediction of the state vector, that is the conditional mean $\mathrm{E}(\alpha_t|Y_{t-1})$ denoted by a_t, together with the variance matrix P_t, for $t = 1, \ldots, n$. The *SsfPack* function `SsfMomentEst` can be used to obtain these quantities.

SsfPack implementation The call of the state prediction function is given by

```
mState = SsfMomentEst(ST_PRED, &mPred, mY, {Ssf});
```

where output matrix `mState` contains a_{n+1} and P_{n+1}. The constant `ST_PRED` is pre-defined and must be given when state prediction is required. The data matrix `mY` and the sequence `{Ssf}` are as usual; see sections 2 and 4. This function also returns a data matrix containing a_t and the diagonal elements of P_t, for $t = 1, \ldots, n$. The structure of the output data matrices are given by

$$\texttt{mState} = \begin{bmatrix} P_{n+1} \\ a'_{n+1} \end{bmatrix}, \qquad \texttt{mPred} = \begin{bmatrix} a_1 & \ldots & a_n \\ \bar{y}_1 & \ldots & \bar{y}_n \\ \mathrm{diag}(P_1) & \ldots & \mathrm{diag}(P_n) \\ \mathrm{diag}(F_1) & \ldots & \mathrm{diag}(F_n) \end{bmatrix},$$

where $\bar{y}_t = \mathrm{E}(y_t|Y_{t-1})$ and $F_t = \mathrm{var}(y_t|Y_{t-1}) = \mathrm{var}(v_t)$ with $v_t = y_t - \bar{y}_t$. The operation $\mathrm{diag}(A)$ vectorizes the diagonal elements of A into a column vector. The output is directly obtained from the Kalman filter. An *Ox* example is given in section 5.2.4.

5.2.2 Forecasting

Forecasts, together with their mean square errors, can be generated by the Kalman filter by extending the data set $y_1, \ldots, y_n$ with a set of missing values. When y_τ is missing, the Kalman filter step at time $t = \tau$ reduces to

$$a_{\tau+1} = T_\tau a_\tau, \qquad P_{\tau+1} = T_\tau P_\tau T'_\tau + H_\tau H'_\tau,$$

which are the state space forecasting equations; see Harvey (1989, page 147) and West and Harrison (1997, page 39). A sequence of missing values at the end of the sample will therefore produce a set of multi-step forecasts. It follows that we can use the *SsfPack* function `SsfMomentEst` also for forecasting. The *Ox* example in section 5.2.4 shows how this can be implemented in a clever way.

5.2.3 State smoothing

The evaluation of $\hat{\alpha}_t = \mathrm{E}(\alpha_t|Y_n)$ and variance matrix $V_t = \mathrm{var}(\alpha_t|Y_n)$ is referred to as moment state smoothing. The usual state smoothing algorithm can be found in Anderson and Moore (1979, page 165) and Harvey (1989, page 149). Computationally more efficient algorithms are developed by de Jong (1988a) and Kohn and Ansley (1989). Koopman (1997) shows how the different algorithms are related. The state smoother in *SsfPack* is given by

$$\hat{\alpha}_t = a_t + P_t r_{t-1}, \qquad V_t = P_t - P_t N_{t-1} P_t, \qquad t = n, \ldots, 1, \tag{30}$$

where r_{t-1} and N_{t-1} are evaluated by (22). Evaluation of these quantities requires a substantial amount of storage space due to the requirement of a_t and P_t. Note that a_t and P_t are evaluated in a forwards fashion and the state smoother is a backwards operation. This storage requirement is additional to the storage space required for the evaluation of r_{t-1} and N_{t-1}; see section 4.3. When only the smoothed state $\hat{\alpha}_t$ is required, more efficient methods of calculation are at present; see sections 4.3.2 and 5.3.1.

SsfPack implementation The call for state smoothing function is

```
mSmo = SsfMomentEst(ST_SMO, &mStSmo, mY, {Ssf});
```

where `mSmo` contains r_0 and N_0. The constant `ST_SMO` is pre-defined and must be given when state smoothing is required. The data matrix `mY` and the sequence {`Ssf`} are as usual; see sections 2 and 4. The function returns also a data matrix containing $\hat{\alpha}_t$ and the diagonal elements of V_t, for $t = 1, \ldots, n$. The structure of the output data matrices are given by

$$\texttt{mSmo} = \begin{bmatrix} N_0 \\ r_0' \end{bmatrix}, \qquad \texttt{mStSmo} = \begin{bmatrix} \hat{\alpha}_1 & \ldots & \hat{\alpha}_n \\ \hat{\theta}_1 & \ldots & \hat{\theta}_n \\ \text{diag}\,(V_1) & \ldots & \text{diag}\,(V_n) \\ \text{diag}\,(S_1) & \ldots & \text{diag}\,(S_n) \end{bmatrix},$$

where $\hat{\theta}_t = c_t + Z_t\hat{\alpha}_t$ is the smoothed estimate of the signal $\theta_t = c_t + Z_t\alpha_t$ with variance matrix $S_t = Z_tV_tZ_t'$. The operation $\text{diag}\,(A)$ vectorizes the diagonal elements of A into a column vector. An *Ox* example is given in section 5.2.4.

5.2.4 An example

The following *Ox* example shows how to obtain predictions, forecasts and smoothed estimates of the state vector for the local linear trend model using an artificial data set.

Ox code of `ssfstate.ox`:

```
#include <oxstd.h>
#include "ssfpack.h"

main()
{
 decl mY, mYF, mPhi, mOmega, mState, mKF, mKS, mFor;
 mY = <1,9,2,5,8,4,6,7,3>;    // data matrix
 mYF = -9999.99 * ones(1, 3); // matrix of missing values
 mPhi = <1,1;0,1;1,0>;
 mOmega = diag(<0.0,0.1,1>);

 mState = SsfMomentEst(ST_PRED, &mKF, mY, mPhi, mOmega);
 print("Final state matrix", mState);
 print("State prediction", mKF');

 SsfMomentEst(ST_PRED, &mFor, mYF, mPhi, mOmega, mState);
 print("Forecasts 1-3 steps ahead", mFor');

 mState = SsfMomentEst(ST_SMO, &mKS, mY, mPhi, mOmega);
 print("Final smoothing matrix", mState);
 print("State smoothing", mKS');
}
```

Ox output:

```
Final state matrix
    1.2387   0.47372
   0.47372    0.3624
    4.3192  -0.46616
State prediction
         0         0         0    1e+005    1e+005    1e+005
   0.99999         0   0.99999    1e+005    1e+005    1e+005
        17    7.9999        17    5.0999       2.2    6.0999
    4.8361   0.37706    4.8361     2.477   0.72459     3.477
    5.3833   0.43047    5.3833    1.7195   0.45554    2.7195
    8.1238     1.086    8.1238    1.4182   0.38484    2.4182
    5.8784   0.17303    5.8784    1.2956   0.36631    2.2956
    6.1459   0.19887    6.1459     1.252    0.3627     2.252
    6.9998   0.37908    6.9998    1.2404   0.36244    2.2404
Forecasts 1-3 steps ahead
    4.3192  -0.46616    4.3192    1.2387    0.3624    2.2387
     3.853  -0.46616     3.853    2.5485    0.4624    3.5485
    3.3869  -0.46616    3.3869    4.6831    0.5624    5.6831
Final smoothing matrix
  9.999946e-006  2.113182e-011
  2.113182e-011  9.999985e-006
  3.610624e-005  7.615753e-006
State smoothing
    3.6106   0.76158    3.6106   0.55364    0.1624   0.55364
    4.3722   0.50051    4.3722    0.2934    0.1002    0.2934
    4.8727   0.44116    4.8727   0.22729  0.072275   0.22729
    5.3139   0.29625    5.3139   0.22134  0.063335   0.22134
    5.6101  0.034399    5.6101   0.22325  0.063335   0.22325
    5.6445  -0.10541    5.6445   0.22134  0.072275   0.22134
    5.5391  -0.28762    5.5391   0.22729    0.1002   0.22729
    5.2515  -0.46616    5.2515    0.2934    0.1624    0.2934
    4.7853  -0.46616    4.7853   0.55365    0.2624   0.55365
```

5.2.5 Disturbance smoothing

The smoothed estimate of the disturbance vector $u_t = (H_t', G_t')' \varepsilon_t$ of the state space form (4), denoted by $\hat{u}_t$ $(t = 1, \ldots, n)$, is discussed in section 4.3. It is noticable that disturbance smoothing can be represented as the simple algorithm

$$\begin{array}{rclcrcl} \hat{u}_t & = & \Omega_t \bar{r}_t, & & \mathrm{var}\,(\hat{u}_t) & = & \Omega_t \bar{N}_t \Omega_t, \\ r_{t-1} & = & \Phi_t \bar{r}_t, & & N_{t-1} & = & \Phi_t \bar{N}_t, \Phi_t', \qquad t = n, \ldots, 1, \end{array}$$

where $\bar{r}_t$ and $\bar{N}_t$ are defined in step (ii) of the algorithm in section 4.3; see also Koopman (1993).

SsfPack implementation The call of the disturbance smoothing function is given by

```
mSmo = SsfMomentEst(DS_SMO, &mDisturb, mY, {Ssf});
```

where `mSmo` contains r_0 and N_0. The constant `DS_SMO` is pre-defined and must be given when disturbance smoothing is required. The data matrix `mY` and the sequence {`Ssf`} are as usual; see

section 4.2. This function also returns a data matrix containing $\hat{\varepsilon}_t$ and the diagonal elements of $\text{var}(\hat{\varepsilon}_t)$, for $t = 1, \ldots, n$. The structure of the output data matrices are given by

$$\texttt{mSmo} = \begin{bmatrix} N_0 \\ r_0' \end{bmatrix}, \qquad \texttt{mDisturb} = \begin{bmatrix} H_1\hat{\varepsilon}_1 & \ldots & H_n\hat{\varepsilon}_n \\ G_1\hat{\varepsilon}_1 & \ldots & G_n\hat{\varepsilon}_n \\ \text{diag}\{\text{var}(H_1\hat{\varepsilon}_1)\} & \ldots & \text{diag}\{\text{var}(H_n\hat{\varepsilon}_n)\} \\ \text{diag}\{\text{var}(G_1\hat{\varepsilon}_1)\} & \ldots & \text{diag}\{\text{var}(G_n\hat{\varepsilon}_n)\} \end{bmatrix}.$$

The operation $\text{diag}(A)$ vectorizes the diagonal elements of A into a column vector. An *Ox* example is given in below.

Example The following *Ox* example shows how to obtain smoothed estimates of the disturbance vector for the local linear trend model using an artificial data set.

Ox code of `ssfds.ox`:

```
#include <oxstd.h>
#include "ssfpack.h"

main()
{
 decl mY, mPhi, mOmega, mDS, mSmo;
 mY = <1,9,2,5,8,4,6,7,3>;
 mPhi = <1,1;0,1;1,0>;
 mOmega = diag(<0.0,0.1,1>);

 mSmo = SsfMomentEst(DS_SMO, &mDS, mY, mPhi, mOmega);
 print("Final smoothing matrix", mSmo);
 print("Disturbance smoothing", mDS');
}
```

Ox output:

```
Final smoothing matrix
  9.9999e-006  2.1132e-011
  2.1132e-011  1.0000e-005
  3.6106e-005  7.6158e-006
Disturbance smoothing
      0.00000     -0.26107      -2.6106      0.00000    0.0044636      0.44636
      0.00000    -0.059351       4.6278      0.00000     0.011227      0.70660
      0.00000     -0.14491      -2.8727      0.00000     0.016135      0.77271
      0.00000     -0.26185     -0.31387      0.00000     0.017810      0.77866
      0.00000     -0.13981       2.3899      0.00000     0.016135      0.77675
      0.00000     -0.18221      -1.6445      0.00000     0.011227      0.77866
      0.00000     -0.17853      0.46089      0.00000    0.0044635      0.77271
      0.00000      0.00000       1.7485      0.00000      0.00000      0.70660
      0.00000      0.00000      -1.7853      0.00000      0.00000      0.44635
```

5.3 The conditional density: mean calculation and simulation

5.3.1 Quick mean calculation of states

When only the mean of the multivariate conditional density $f(\alpha_1, \ldots, \alpha_n|Y_n)$, that is smoothed state vector $\hat{\alpha}_t = \text{E}(\alpha_t|Y_n)$, for $t = 1, \ldots, n$, is required, the following simple forwards recursion

can be used, that is

$$\hat{\alpha}_{t+1} = d_t + T_t\hat{\alpha}_t + H_t\hat{\varepsilon}_t, \qquad t = 1, \ldots, n-1,$$

with $\hat{\alpha}_1 = a + Pr_0$ and $\hat{\varepsilon}_t = \mathrm{E}(\varepsilon_t|Y_n) = H_t'r_t + G_t'e_t$; see Koopman (1993) and section 4.3.2. The smoothing quantities e_t and r_t are obtained from (22). This algorithm does not require the additional storage space from the Kalman filter, that is a_t and P_t for $t = 1, \ldots, n$, as required for general state moment smoothing. It is also computationally much more efficient. In most practical situations interest is limited to the smoothed state vector $\hat{\alpha}_t$ and the smoothed estimate of the signal $c_t + Z_t\hat{\alpha}_t$. The *SsfPack* function `SsfCondDens` calculates these quantities using this quick state smoothing method.

5.3.2 Simulation for states

The simulation smoother is also able to generate simulations from $f(\alpha|Y_n)$, where $\alpha = (\alpha_1', \ldots, \alpha_n')'$, for a given model in state space form; see de Jong and Shephard (1995) and section 4.4. The simulations are denoted by $\alpha^{(\cdot)} = \left(\alpha_1^{(\cdot)\prime}, \ldots, \alpha_n^{(\cdot)\prime}\right)'$. The simulation smoother, with an appropriate choice of the selection matrix outputs the simulation draws $H_1\varepsilon_1^{(\cdot)}, \ldots, H_n\varepsilon_n^{(\cdot)}$ from which the simulated states can be obtained via the state space recursion

$$\alpha_{t+1}^{(\cdot)} = d_t + T_t\alpha_t^{(\cdot)} + H_t\varepsilon_t^{(\cdot)}, \qquad t = 1, \ldots, n,$$

with the initialisation $\alpha_1^{(\cdot)} = a + Pr_0$ where r_0 is obtained from the simulation smoother (24). Consistent simulations for the signal θ_t are obtained via the relation $\theta_t^{(\cdot)} = c_t + Z_t\alpha_t^{(\cdot)}$, for $t = 1, \ldots, n$. The *SsfPack* function `SsfCondDens` outputs these simulations. Note that when no consistency is required between $\theta^{(\cdot)}$ and $\alpha^{(\cdot)}$, simulation samples for $\theta^{(\cdot)}$ are much easier obtained using the relationship $\theta_t^{(\cdot)} = y_t - G_t\varepsilon_t^{(\cdot)}$; see the discussion in section 4.4.

5.3.3 Mean calculation of disturbances

The mean of the multivariate conditional density $f(u_1, \ldots, u_n|Y_n)$, where $u_t = (H_t', G_t')'\varepsilon_t$ as defined in (4), is denoted by $\hat{u} = (\hat{u}_1, \ldots, \hat{u}_n)$ and its calculation is discussed in sections 4.3 and 5.2.5. The *SsfPack* function `SsfCondDens` allows calculation of $\hat{u}$ in a computationally efficient way and outputs only $\hat{u}$. The *SsfPack* function `SsfCondDens` produces these simulations.

5.3.4 Simulation for disturbances

Generating samples from $f(u|Y_n)$ for a given model in state space form is done via the simulation smoother; the details are given in section 4.4. The *SsfPack* function `SsfCondDens` is able to output a draw from $f(u|Y_n)$ which is denoted by $u^{(\cdot)}$. As pointed out by de Jong and Shephard (1995), the simulation smoother cannot draw from $f(u|Y_n)$ directly because of the implied identities within the state space form (4); this problem is referred to as degenerate sampling. However it can simulate from $f(H_1\varepsilon_1, \ldots, H_n\varepsilon_n|Y_n)$ directly and then computing the sample $\theta^{(\cdot)}$ as discussed in section 5.3.2. The identity $G_t\varepsilon_t = y_t - \theta_t$ allows the generation of simulation samples from $f(G_1\varepsilon_1, \ldots, G_n\varepsilon_n|Y_n)$ which are consitent with the sample from $f(H_1\varepsilon_1, \ldots, H_n\varepsilon_n|Y_n)$. In this way the *SsfPack* function `SsfCondDens` computes consistent simulation samples for the disturbance vector u. Finally, when the rank of H_t is smaller than G_t the described method of getting simulations from $f(u|Y_n)$ is not valid and the *SsfPack* function should not be used for simulation. Instead, the simulation smoother should be applied directly as described in section 4.4.

SsfPack implementation The *SsfPack* call for calculating mean and simulation for the multivariate conditional densities of the disturbances and the states is given by

```
mD = SsfCondDens(iSel, mY, {Ssf});
```

where the structure of the output matrix `mD` depends on the value of `iSel` which can only take the value of the predefined constants:

`DS_SMO`	computes the mean of $f(u|Y_n)$;
`DS_SIM`	produces simulation sample from $f(u|Y_n)$;
`ST_SMO`	computes the mean of $f(\alpha|Y_n)$;
`ST_SIM`	produces simulation sample from $f(\alpha|Y_n)$.

The output matrix mD for the different tunings is given by

$$\begin{aligned}
\texttt{DS_SMO} \quad &: \quad \texttt{mD} = \begin{bmatrix} \hat{u}_1 & \dots & \hat{u}_n \end{bmatrix}, \\
\texttt{DS_SIM} \quad &: \quad \texttt{mD} = \begin{bmatrix} u_1^{(\cdot)} & \dots & u_n^{(\cdot)} \end{bmatrix}, \\
\texttt{ST_SMO} \quad &: \quad \texttt{mD} = \begin{bmatrix} \hat{\alpha}_1 & \dots & \hat{\alpha}_n \\ \hat{\theta}_1 & \dots & \hat{\theta}_n \end{bmatrix}, \\
\texttt{ST_SIM} \quad &: \quad \texttt{mD} = \begin{bmatrix} \alpha_1^{(\cdot)} & \dots & \alpha_n^{(\cdot)} \\ \theta_1^{(\cdot)} & \dots & \theta_n^{(\cdot)} \end{bmatrix},
\end{aligned}$$

where $\hat{\theta}_t = c_t + Z_t\hat{\alpha}_t$ is the smoothed estimate of the signal $c_t + Z_t\alpha_t$ and $\theta_t^{(\cdot)}$ is the associated simulation. The inputs `mY` and {`Ssf`} are as usual.

Example The following *Ox* example produces the mean and a simulation sample from the conditional density of the disturbances and the states for the local linear trend model using an artificial data set. The *Ox* function `ranseed` is used to let the two simulation samples be comparable with each other.

Ox code of `ssfcond.ox`:

```
#include <oxstd.h>
#include "ssfpack.h"

main()
{
 decl mY, mPhi, mOmega, mD;
 mY = <1,9,2,5,8,4,6,7,3>;
 mPhi = <1,1;0,1;1,0>;
 mOmega = diag(<0.0,0.1,1>);

 mD = SsfCondDens(DS_SMO, mY, mPhi, mOmega);
 print("mean disturbance", mD');
 ranseed(5);
 mD = SsfCondDens(DS_SIM, mY, mPhi, mOmega);
 print("simulation disturbance", mD');
 mD = SsfCondDens(ST_SMO, mY, mPhi, mOmega);
 print("mean state", mD');
 ranseed(5);
 mD = SsfCondDens(ST_SIM, mY, mPhi, mOmega);
 print("simulation state", mD');
}
```

Ox output:

```
mean disturbance
      0.00000     -0.26107      -2.6106
      0.00000    -0.059351       4.6278
      0.00000     -0.14491      -2.8727
      0.00000     -0.26185     -0.31387
      0.00000     -0.13981       2.3899
      0.00000     -0.18221      -1.6445
      0.00000     -0.17853      0.46089
      0.00000      0.00000       1.7485
      0.00000      0.00000      -1.7853
simulation disturbance
      0.00000     -0.30023      -2.3523
      0.00000    0.0035786       4.7010
      0.00000     -0.41161      -2.9454
      0.00000     -0.34580     -0.59544
      0.00000     -0.16944       2.1661
      0.00000     -0.14802      -1.7265
      0.00000      0.17320      0.55037
      0.00000      0.43605       1.9752
      0.00000      0.17320      -1.7731
mean state
       3.6106      0.76158       3.6106
       4.3722      0.50051       4.3722
       4.8727      0.44116       4.8727
       5.3139      0.29625       5.3139
       5.6101     0.034399       5.6101
       5.6445     -0.10541       5.6445
       5.5391     -0.28762       5.5391
       5.2515     -0.46616       5.2515
       4.7853     -0.46616       4.7853
simulation state
       3.3523      0.94668       3.3523
       4.2990      0.64645       4.2990
       4.9454      0.65002       4.9454
       5.5954      0.23841       5.5954
       5.8339     -0.10739       5.8339
       5.7265     -0.27683       5.7265
       5.4496     -0.42485       5.4496
       5.0248     -0.25166       5.0248
       4.7731      0.18439       4.7731
```

6 Gaussian illustrations

6.1 Maximum likelihood estimation

The exact maximum likelihood estimates of, for example, parameters of an ARMA model (see section 3.1) or unknown variances of a structural time series model (see section 3.2) can be obtained by maximizing a likelihood criterion via a numerical optimisation routine. A number of optimisation routines are provided by *Ox*. In the following we use the `MaxBFGS` function of

Ox, see Doornik (1996), for estimating a local level model which is given by

$$\begin{aligned} y_t &= \mu_t + \xi_t, \qquad \varepsilon_t \sim \text{NID}\left(0, \sigma_\xi^2\right), \\ \mu_{t+1} &= \mu_t + \eta_t, \qquad \eta_t \sim \text{NID}\left(0, \sigma_\eta^2\right), \qquad t = 1, \ldots, n, \end{aligned} \tag{31}$$

with $\mu_1 \sim N(0, \kappa)$ and κ large. This model has two unknown variances which are re-parameterised as

$$\sigma_\eta^2 = \exp(2\varphi_0), \qquad \sigma_\xi^2 = \exp(2\varphi_1), \tag{32}$$

so that the likelihood criterion can be maximised without constraints with respect to $\varphi = (\varphi_0, \varphi_1)'$. Likelihood evaluation for a given vector $\varphi = \varphi^*$ takes place via the *SsfPack* function `SsfLik` after the variances σ_η^{2*} and σ_ξ^{2*} are computed using (32). The score (27) is calculated by

$$\left.\frac{\partial l}{\partial \varphi_0}\right|_{\varphi = \varphi^*} = \sigma_\eta^{2*} S_{00}, \qquad \left.\frac{\partial l}{\partial \varphi_1}\right|_{\varphi = \varphi^*} = \sigma_\xi^{2*} S_{11},$$

where S_{ij} is the (i,j)-th element of matrix S in (27) for $\varphi = \varphi^*$. Matrix S can be obtained directly from the *SsfPack* function `SsfLikSco`.

Example The following *Ox* program estimates the parameters of the local level model (31), with starting values $\sigma_\eta = 0.5$ and $\sigma_\xi = 1$, for the Nile data of Appendix A. The maximization routine is provided by *Ox*.

Ox code of `ssfmle.ox`:

```
#include <oxstd.h>
#include <maximize.h>
#include "SsfPack.h"
#pragma link("maximize.oxo")

static decl mY, mPhi, mOmega, mSigma;

Likelihood(const vP, const pdLik, const pvSco, const pmHes)
{
 decl dVar, mSco;
 mOmega[0][0] = exp(2.0 * vP[0][0]);
 mOmega[1][1] = exp(2.0 * vP[1][0]);
 if (pvSco)
 {
  SsfLikSco(pdLik, &mSco, mY, mPhi, mOmega, mSigma);
  pvSco[0][0][0] = mOmega[0][0] * mSco[0][0];
  pvSco[0][1][0] = mOmega[1][1] * mSco[1][1];
 }
 else
  SsfLik(pdLik, &dVar, mY, mPhi, mOmega, mSigma);
 return 1;
}

main()
{
 decl vp, vs, mhes, ir, dLik, dVar;
 mY = loadmat("Nile.dat")';
```

```
 GetSsfStsm(<CMP_IRREG,      1.0, 0, 0;
             CMP_LEVEL,      0.5, 0, 0>, &mPhi, &mOmega, &mSigma);

 // scale initial parameter estimates
 vp = zeros(2, 1);
 vp[0] = log(0.5);
 SsfLik(&dLik, &dVar, mY, mPhi, mOmega, mSigma);
 vp += 0.5 * log(dVar);

 // maximum likelihood estimation
 vs = zeros(2, 1);
 mhes = unit(2);
 Likelihood(vp, &dLik, 0, 0);
 MaxControl(50, 10);
 ir = MaxBFGS(Likelihood, &vp, &dLik, &mhes, FALSE);

 print("\n", MaxConvergenceMsg(ir),
       " using analytical derivatives",
       "\nFunction value = ", dLik, "; parameters:", vp);
 print("Omega", mOmega);
}
```

Ox output:

```
Starting values
parameters
       4.0325       4.7257
gradients
      -2.1358       2.6475
Initial function =      -548.011132936

Position after 7 BFGS iterations
Status: Strong convergence
parameters
       3.6537       4.8124
gradients
 -1.4887e-009 -4.7100e-010
function value =      -547.529493002

Strong convergence using analytical derivatives
Function value = -547.529; parameters:
       3.6537
       4.8124
Omega
       1491.4      0.00000
      0.00000       15135.
```

6.2 Detecting outliers and structural breaks

General procedures for testing for outliers and structural breaks based on models in state space form are discussed by Harvey, Koopman, and Penzer (1998). Such irregularities in data can be modelled in terms of impulse interventions applied to the equations of the state space form. For

example, an outlier can be captured within the measurement equation by a dummy explanatory variable, known as an impulse intervention variable, which takes the value one at the time of the outlier and zero elsewhere. The estimated regression coefficient of this variable is an indication whether an outlying observation is present. In the case of unobserved component time series models, this approach reduces to a procedure based on the so-called auxiliary residuals. The standardised residuals associated with the measurement and system equations are computed via a single filter and smoothing step; see section 4.3 and section 5.2.5. These auxiliary residuals are introduced and studied in detail by Harvey and Koopman (1992); they show that these residuals are an effective tool for detecting outliers and breaks in time series and for distinguishing between them. de Jong and Penzer (1998) have shown that auxiliary residuals are equivalent to t-statistics for the impulse intervention variables.

Example The following *Ox* example reports whether the absolute values of the auxiliary residuals exceed some benchmark implying that an outlier or break is present in the series. The Nile data of Appendix A and the local level model (31), with standard deviations equal to their estimates of the example in section 6.1, are considered.

Ox code of `ssfoutl.ox`:

```
#include <oxstd.h>
#include "SsfPack.h"
main()
{
 decl mY, mPhi, mOmega, mSigma, mD, mS;
 // data
 mY = loadmat("Nile.dat")';
 // stuctural time series model
 GetSsfStsm(<CMP_LEVEL, 38, 0, 0;
             CMP_IRREG, 123, 0, 0>, &mPhi, &mOmega, &mSigma);
 // smoothed state vector
 SsfMomentEst(DS_SMO, &mD, mY, mPhi, mOmega, mSigma);
 // auxiliary residuals
 mS = mD[0:1][]./sqrt(mD[2:3][]);
 if (fabs(mS[0][1:98]) < 3) print("\n no structural break");
 else        print("\n structural break(s)");
 if (fabs(mS[1][]) < 3)    print("\n no outlier");
 else        print("\n outlier(s)");
}
```

Ox output:

```
 structural break(s)
 outlier(s)
```

6.3 Regression analysis

When the standard regression model $y_t = X_t\beta + \xi_t$ with $\xi_t \sim \text{NID}\left(0, \sigma^2\right)$ is placed in the state space form, the Kalman filter reduces to what is known as 'recursive least squares' algorithm; see ... The state prediction a_t is the least squares estimate $\left(\sum_{j=1}^{t-1} X_t X_t'\right)^{-1}\left(\sum_{j=1}^{t-1} X_t y_t\right)$ and matrix P_t is the matrix $\left(\sum_{j=1}^{t-1} X_t X_t'\right)^{-1}$. Therefore, the *SsfPack* function `SsfStateEst` can be used to obtain these quantities and to obtain the final OLS estimates, that is a_{n+1} and P_{n+1}. It

is easy within *Ox* to get, for example, the residual sum of squares, the goodness-of-fit measure R^2 and the t-test statistics.

Additional statistical output is obtained from smoothing. Following the arguments of de Jong and Penzer (1998), the output of the basic smoothing recursions can be used to construct t-tests for structural changes in regression models. Consider the state space framework for a regression model with the $1 \times k$ vector of explanatory variables $X_t = (x_{1,t}, \ldots, x_{k,t})$ and the $k \times 1$ vector of coefficients $\beta = (\beta_1, \ldots \beta_k)'$, that is

$$y_t = x_{1,t}\beta_1 + \ldots + x_{1,t}\beta_k + \xi_t, \qquad t = 1, \ldots, n,$$

where $\xi_t \sim \text{NID}\left(0, \sigma^2\right)$. The null hypothesis with respect to the ith explanatory variable,

$$\begin{aligned} y_t &= \ldots + x_{i,t}\beta_i + \ldots + \xi_t, \qquad \text{for } t = 1, \ldots, \tau, \\ y_t &= \ldots + x_{i,t}\beta_i^* + \ldots + \xi_t, \qquad \text{for } t = \tau + 1, \ldots, n, \end{aligned}$$

against the alternative $\beta_i = \beta_i^*$ can be tested via the t-test

$$r_{i,\tau}/\sqrt{N_{ii,\tau}}, \qquad \tau = 1, \ldots, n-1,$$

where $r_t = (r_{1,t}, \ldots r_{p,t})'$ and N_t, with the element (i,i) denoted as $N_{ii,t}$, are evaluated using the basic smoothing recursions (22). It is remarkable that $(n-1)\,k$ t-tests can be computed from a single run of the basic smoother. The t-test is distributed as a t distribution with $n-k$ degrees of freedom. When the t-test is relatively large, the null hypothesis is not rejected.

Example The following *Ox* example outputs the least squares estimates, the standard errors and the t-tests for a standard regression model with three explanatory variables. The t-tests for structural changes are also outputted. The data is generated such that the null hypothesis of a structural change of regression coefficient 2 at time $\tau = 10$ should be accepted.

Ox code of `ssfregan.ox`:

```
#include <oxstd.h>
#include "ssfpack.h"

main()
{
 decl mY, mX, mPhi, mOmega, mSigma, mJ_Phi, mKF, mKS, mOLS;
 mY = rann(1,20);
 mX = rann(3,20);
 mX[1][10:19] = (mX[1][10:19]+(3.0 * mY[0][10:19])) / 4.0;

 GetSsfReg(mX, &mPhi, &mOmega, &mSigma, &mJ_Phi);

 mOLS = SsfMomentEst(ST_PRED, &mKF, mY, mPhi, mOmega, mSigma,
                     <0>, mJ_Phi, <0>, <0>, mX);
 print("Regression");
 print("%r",{"x1","x2","x3"},"%c",{"ols","s.e.","t-test"},
 mOLS[3][]'~diagonal(mOLS)'~(mOLS[3][]./sqrt(diagonal(mOLS)))');

 mKF = KalmanFil(mY,  mPhi, mOmega, mSigma, <0>, mJ_Phi, <0>, <0>, mX);
 mKS = KalmanSmo(mKF, mPhi, mOmega, mSigma, <0>, mJ_Phi, <0>, <0>, mX);
 print("\nStructural change t-test");
 print("%c", {"obs.nr","x1", "x2", "x3"},
```

```
 <2:20>'~(mKS[0:2][1:19]./sqrt(mKS[4:6][1:19]))');
}
```

Ox output:

```
Regression
                ols         s.e.       t-test
x1       -0.0065340     0.051600    -0.028764
x2          0.44674     0.084515       1.5367
x3          0.20491     0.070790      0.77016

Structural change t-test
       obs.nr           x1           x2           x3
       2.0000      0.91488      0.91500     -0.91499
       3.0000     -0.53003       1.1276      -1.1032
       4.0000     -0.50773      0.83053     -0.98702
       5.0000      -1.0655       1.6978      -1.7035
       6.0000      -1.1224       1.2906      -1.0307
       7.0000     -0.74924       1.2659      -1.0426
       8.0000     -0.64998       1.6776     -0.42568
       9.0000     -0.38892       1.4079     -0.42345
       10.000     -0.40212       2.0990      -1.0300
       11.000     -0.39656       2.0958     -0.83555
       12.000     -0.20730       2.0803     -0.92720
       13.000     -0.69921       1.9515     -0.70051
       14.000      0.11300       1.6878     -0.41237
       15.000     -0.25228       1.2906     -0.87842
       16.000     -0.25383       1.2874     -0.82397
       17.000      0.39532      0.82284     -0.27782
       18.000      0.57649      0.79318     -0.27915
       19.000      0.57986      0.78607     -0.35359
       20.000     -0.38331      0.38331      0.38331
```

6.4 Spline smoothing and interpolation

The nonparametric spline method can be regarded as an interpolation technique. Consider a set of observations which are spaced at equal intervals but some observations are missing. To 'fill in the gaps' the spline model of section 3.4 can be considered. Applying filtering and smoothing to this model, we obtain the estimated signal. In this way, a graphical representation of the nonparametric spline can be produced.

Example The following *Ox* example outputs the least squares estimates, the standard errors and the t-tests for a standard regression model with three explanatory variables. The t-tests for structural changes are also outputted. The data is generated such that the null hypothesis of a structural change of regression coefficient 2 at time $\tau = 10$ should be accepted.

Ox code of `ssfsplan.ox`:

```
#include <oxstd.h>
#include "SsfPack.h"
main()
{
 decl mDelta, mPhi, mOmega, mJ_Phi, mJ_Omega, mY, mX, mD;
```

```
 mDelta = <1,1,1,1,1,1,1,1,1>;
 mY = <30,-9999.99,-9999.99,45,-9999.99,65,-9999.99,-9999.99,35>;
 // cubic spline model with q = 0.7
 GetSsfSpline(0.7, mDelta, &mPhi, &mOmega, &mJ_Phi, &mJ_Omega, &mX);
 // smoothed state vector
 mD = SsfCondDens(ST_SMO, mY, mPhi, mOmega,<0>,<0>,mJ_Phi, mJ_Omega,<0>,mX);
 print("Data and interpolations", "%12.2f", mY'~mD[0][]');
}
```

Ox output:

```
Data and interpolations
       30.00       30.06
    -9999.99       36.56
    -9999.99       43.01
       45.00       49.36
    -9999.99       55.08
       65.00       57.55
    -9999.99       54.56
    -9999.99       47.35
       35.00       38.02
```

6.5 Seasonal adjustment and detrending

Seasonal adjustment is a relatively easy task when time series are modelled as an unobserved components time series model in which a seasonal component is included; see section 3.2. The estimated seasonal component is substracted from the original time series in order to get the seasonally adjusted series. In the same way the original time series is detrended by substracting the estimated trend component.

Example The following *Ox* example outputs the seasonally adjusted series for the Airline data of Appendix B. The original data is logged and is modelled by a model with trend and trigonometric seasonal components.

Ox code of `ssfseas.ox`:

```
#include <oxstd.h>
#include "SsfPack.h"
main()
{
 decl mY, mPhi, mOmega, mSigma, mD, mS;
 // data
 mY = log(loadmat("Airline.dat"))';
 // stuctural time series model
 GetSsfStsm(<CMP_IRREG,     0.9,  0, 0;
             CMP_LEVEL,     1.0,  0, 0;
             CMP_SLOPE,     0.0,  0, 0;
             CMP_SEAS_TRIG, 0.1, 12, 0>, &mPhi, &mOmega, &mSigma);
 // smoothed state vector
 mD = SsfCondDens(ST_SMO, mY, mPhi, mOmega, mSigma);
 mS = mY - <0,0,1,0,1,0,1,0,1,0,1,0,1,0> * mD;
 print("Seasonally adjusted data", mS);
}
```

Ox output:

```
Seasonally adjusted data
      4.8191       4.8232       4.8247       4.8415       4.8297       4.8213
      4.8196       4.8251       4.8406       4.8567       4.8705       4.8602
      4.8477       4.8899       4.8883       4.8948       4.8616       4.9178
...
      6.0542       6.0786       6.0807       6.0796       6.1060       6.1196
      6.1153       6.1128       6.0819       6.1488       6.1539       6.1564
      6.1756       6.1633       6.1758       6.1959       6.1841       6.1874
```

6.6 Monte Carlo simulations and parametric bootstrap tests

Statistical methods such as Monte Carlo and bootstrap require random samples from the unconditional distribution implied by the model in state space form. The *SsfPack* function `SsfRecursion` can be useful in this respect. For illustrative purposes, we will present a simple parametric bootstrap procedure for testing for a unit root when the null is stationarity. This problem has been extensively studied in the literature. The initial work was carried out by Nyblom and Makelainen (1983) and Tanaka (1983), while the more recent work is reviewed in Tanaka (1996, Ch. 10).

Consider the local level model (31) and the vector of univariate observations $y = (y_1, \ldots, y_n)'$. The hypothesis

$$H_o : \sigma_\eta^2 = 0, \qquad H_1 : \sigma_\eta^2 > 0,$$

imply that y_t is a stationary series under the null hypothesis and that y_t has a unit root otherwise. The null also implies the level $\mu_0 = \cdots = \mu_n = \mu$ and that the constrained maximum likelihood estimators of μ and σ_ε^2 are simply the sample average $\overline{y}$ and the sample variance $\widehat{\sigma_\varepsilon^2} = \frac{1}{n}\sum_{t=1}^n (y_t - \overline{y})^2$.

The null hypothesis can be tested using a score test which has the form of

$$s = \left.\frac{\partial l(y;\theta)}{\partial \sigma_\eta^2}\right|_{\sigma_\eta^2 = 0, \mu = \overline{y}, \sigma_\varepsilon^2 = \widehat{\sigma_\varepsilon^2}},$$

and the null hypothesis is rejected if the score is relatively large. This statistic is, up to a constant, the same as the locally best invariant (LBI) test and is known to be asymptotically pivotal (see, for example, Tanaka (1996, Ch. 10.7)) although the form of the distribution is complicated and has to be derived by numerically inverting a characteristic function or by simulation.

A bootstrap test for the null hypothesis is particularly straightforward for this problem. Define $y^{(j)}$ as a sample of size n drawn from $NID(\overline{y}, \widehat{\sigma_\varepsilon^2})$. Then for each draw the corresponding score statistic is computed, that is

$$s_j = \left.\frac{\partial l(y^{(j)};\theta)}{\partial \sigma_\eta^2}\right|_{\sigma_\eta^2 = 0, \mu = \overline{y^{(j)}}, \sigma_\varepsilon^2 = \frac{1}{n}\sum_{t=1}^n \left(y_t^{(j)} - \overline{y^{(j)}}\right)^2}.$$

The observed value s is compared with a population of simulated score statistics s_j, $j = 1, \ldots, M$, for some predefined integer M. This bootstrap test is easily generalised for more general settings.

Interestingly the bootstrap test for the local level model can be made exact if we simulate $y^{(j)}$ in a slightly different way. Define $u^{(j)}$ as a sample of size n drawn from $NID(0, I)$. Transforming

the generated sample by

$$y^{*(j)} = \overline{y} + \widehat{\sigma_\varepsilon} \frac{u^{(j)} - \overline{u^{(j)}}}{\sqrt{\frac{1}{n}\sum_{t=1}^{n}\left(u_i^{(j)} - \overline{u^{(j)}}\right)^2}},$$

for each $y^{*(j)}$, it follows that the identies $\overline{y^{*(j)}} = \overline{y}$ and $\frac{1}{n}\sum_{t=1}^{n}\left(y_t^{*(j)} - \overline{y^{*j}}\right)^2 = \widehat{\sigma_\varepsilon^2}$ apply for $j = 1, \ldots, M$. Thus y^* is being simulated conditionally on the null hypothesis' sufficient statistics. Consequently, under the null hypothesis, the distribution of y^* is parameter free. As a result simulations from

$$s_j^* = \left.\frac{\partial l(y^{*j};\theta)}{\partial \sigma_\eta^2}\right|_{\sigma_\eta^2 = 0, \mu = \overline{y}, \sigma_\varepsilon^2 = \widehat{\sigma_\varepsilon^2}},$$

provide an exact benchmark for the distribution of s. For example, a test with 5% size can be constructed using 100 simulations by recording s and then simulating $s_1^*, ..., s_{99}^*$. If s is one of the largest five in the population of $s, s_1^*, ..., s_{99}^*$ then the hypothesis is rejected.

This exact testing procedure is difficult to extend to more complicated dynamic models and so one usually has to rely on the asymptotic pivotal nature of the score statistic to produce good results.

Example The exact testing procedure is implemented for the local level model with the null hypothesis $\sigma_\eta^2 = 0$ using the Nile data of Appendix A.

Ox code of `ssfboot.ox`:

```
#include <oxstd.h>
#include "SsfPack.h"

main()
{
 decl i, mY, cT, mPhi, mOmega, mSigma;
 decl y_i, mn_y, sd_y, dLik, dVar, mSco, cBoot, mBoot;

 mY = loadmat("Nile.dat");
 GetSsfStsm(<CMP_LEVEL, 0.0, 0, 0;
             CMP_IRREG, 0.0, 0, 0>, &mPhi, &mOmega, &mSigma);
 cT = rows(mY);
 mOmega[1][1] = varc(mY);
 mn_y = meanc(mY); sd_y = varc(mY)^0.5;

 cBoot = 1000;
 mBoot = zeros(1, cBoot);

 SsfLikSco(&dLik, &mSco, mY', mPhi, mOmega, mSigma);
 mBoot[0][0] = mSco[0][0];
 print("\n bootstrap test for Nile is ", mSco[0][0]);
 for (i=1; i<cBoot; i++)
 {
  y_i = mn_y + (sd_y * standardize(rann(cT, 1)));
  SsfLikSco(&dLik, &mSco, y_i', mPhi, mOmega, mSigma);
```

```
  mBoot[0][i] = mSco[0][0];
 }
 mBoot = sortr(mBoot);
 print("\n 90% crit.value is ", mBoot[0][89 * cBoot / 100]);
 print("\n 95%     -         ", mBoot[0][94 * cBoot / 100]);
 print("\n 99%     -         ", mBoot[0][98 * cBoot / 100]);
}
```

Ox output:

```
bootstrap test for Nile is 0.748234
90% crit.value is 0.0703607
95%     -         0.106543
99%     -         0.180178
```

6.7 Bayesian parameter estimation

6.7.1 The basics

Bayesian inference on parameters indexing models has attracted a great deal of interest. Recall that if we have a prior on the parameters φ of $f(\varphi)$, then

$$f(\varphi|y) \propto f(\varphi) \int f(y|\alpha,\varphi) f(\alpha|\varphi) d\alpha = f(\varphi) f(y|\varphi).$$

In the Gaussian case we can evaluate $f(y|\varphi) = \int f(y|\alpha,\varphi) f(\alpha|\varphi) d\alpha$ using the Kalman filter. Even though we have the posterior density up to proportionality it is still not easy to compute posterior moments or quantiles about φ as this involves a further level of integration. Thus it looks like Bayesian inference is harder than maximum likelihood estimation.

In recent years there has been enormous advances in numerical methods for computing functionals of the posterior density $f(\varphi|y)$ due to the advent of Markov chain Monte Carlo (MCMC) methods, namely the Metropolis-Hastings algorithm and its special case the Gibbs sampling algorithm. These methods have had a widespread influence on the theory and practice of Bayesian inference. Early work on these methods appears in Metropolis, Rosenbluth, Rosenbluth, Teller, and Teller (1953), Hastings (1970), Ripley (1977) and Geman and Geman (1984) while some of the more recent developments, spurred by Tanner and Wong (1987) and Gelfand and Smith (1990), are included in Chib and Greenberg (1996), Gilks, Richardson, and Spiegelhalter (1996) and Tanner (1996, Ch. 6). Chib and Greenberg (1995) provide a detailed exposition of the Metropolis-Hastings algorithm and include a derivation of the algorithm from the logic of reversibility.

The idea behind MCMC methods is to produce variates from a given multivariate density (the posterior density in Bayesian applications) by repeatedly sampling a Markov chain whose invariant distribution is the target density of interest — $f(\varphi|y)$ in the above case. There are typically many different ways of constructing a Markov chain with this property and an important goal of the literature on MCMC methods in state space models is to isolate those that are simulation–efficient. It should be kept in mind that sample variates from a MCMC algorithm are a high-dimensional (correlated) sample from the target density of interest. These draws can be used as the basis for making inferences by appealing to suitable ergodic theorems for Markov chains. For example, posterior moments and marginal densities can be estimated (simulation consistently) by averaging the relevant function of interest over the sampled variates. The posterior mean of φ is simply estimated by the sample mean of the simulated φ values. These estimates can be made arbitrarily accurate by increasing the simulation sample size. The accuracy of the resulting estimates (the so called numerical standard error) can be assessed by

standard time series methods that correct for the serial correlation in the draws. The serial correlation can be quite high for badly behaved algorithms.

To be able to use an MCMC algorithm we need to be able to evaluate the target density up to proportionality. This is the case for our problem as we know $f(\varphi|y) \propto f(\varphi)f(y|\varphi)$ using the Kalman filter. The next subsection will review the nuts and bolts of the sampling mechanism.

6.7.2 Metropolis algorithm

We will use an independence chain Metropolis algorithm to simulate from the abstract joint distribution of $\psi_1, \psi_2, .., \psi_m$. Proposals z are made to possibly replace the current ψ_i, keeping constant $\psi_{\backslash i}$ which is notation for all elements of ψ except ψ_i. The proposal density is proportional to $q(z, \psi_{\backslash i})$ while the true density is proportional to $f(\psi_i|\psi_{\backslash i})$. Both of these densities are assumed to be everywhere positive, with compact support and known up to proportionality. If $\psi^{(k)}$ is the current state of the sampler then the proposal to take $\psi^{(k+1)} = (z, \psi_{\backslash i}^{(k)})$ is accepted if

$$c < \min\left\{\frac{f(z|\psi_{\backslash i}^{(k)})q(\psi_i^{(k)}, \psi_{\backslash i}^{(k)})}{f(\psi_i^{(k)}|\psi_{\backslash i}^{(k)})q(z, \psi_{\backslash i}^{(k)})}, 1\right\}, \qquad \text{where} \qquad c \sim \text{UID}(0,1),$$

while if it is rejected we set $\psi^{(k+1)} = \psi^{(k)}$. Typically we wish to design $q(\psi_i^{(k)}, \psi_{\backslash i}^{(k)})$ to be close to $f(z|\psi_{\backslash i}^{(k)})$ but preferably with heavier tails (see, for example, Chib and Greenberg (1996)).

In the context of learning about parameters in a Gaussian state space model this algorithm has $\psi = \varphi|y$. Then the task of performing MCMC on the parameters is one of designing a proposal density $q(\varphi_i^{(k)}, \varphi_{\backslash i}^{(k)})$ which will be typically be close to being proportional to $f(\varphi_i|\varphi_{\backslash i}^{(k)}, y) \propto f(\varphi_i, \varphi_{\backslash i}^{(k)}|y)$. This is not particularly easy to do, although generic methods are available: see, for example, Gilks, Best, and Tan (1995).

In the rather simpler case where we can choose

$$q(\psi_i, \psi_{\backslash i}^{(k)}) = f(\psi_i|\psi_{\backslash i}^{(k)}),$$

then the Metropolis algorithm is called a Gibbs sampler (Geman and Geman (1984) and Gelfand and Smith (1990)) and there is never any rejection of the suggestions. Unfortunately for the unknown parameter problems in a Gaussian model $f(\varphi_i|\varphi_{\backslash i}^{(k)}, y)$ is only known up to proportionality and so the simplicity of the Gibbs sampler is not available to us.

6.7.3 Augmentation

As the design of proposal densities for the Metropolis algorithm is sometimes difficult an alternative method has been put forward by Fruhwirth-Schnatter (1994). The suggestion is of added interest as it is the only available way to make progress when we move to non-Gaussian problems where evaluating $f(y|\varphi) = \int f(y|\alpha, \varphi)f(\alpha|\varphi)d\alpha$ is generally not possible.

The suggestion is to design MCMC methods for simulating from the density $\pi(\varphi, \alpha|y)$, where $\alpha = (\alpha_1, ..., \alpha_n)$ is the vector of n latent states, rather than $\pi(\varphi|y)$. The draws from this joint density provide draws from the marginal density $\pi(\varphi|y)$ by simply ignoring the draws from the states and so solve the original problem. It turns out that rather simple Markov chain Monte Carlo procedures can be developed to sample $\pi(\varphi, \alpha|y)$. In particular we could

1. Initialize φ
2. Sample from the multivariate Gaussian distribution of $\alpha|y, \varphi$ using a simulation smoother.
3. Sample from $\varphi|y, \alpha$ directly or do a Gibbs or Metropolis update on the elements.

4. Goto 2.

The key features are that the simulation smoother allows all the states to be drawn as a block in a simple and generic way while we can usually draw from $\varphi|y,\alpha$ in a relatively trivial way. To illustrate this second point, suppose the model is a local linear trend (12) with added measurement error $\varepsilon_t \sim \text{NID}(0,\sigma_\varepsilon^2)$. When we draw from $\varphi|y,\alpha$ we act as if y,α is known. Knowing α tells us both $\{\mu_t\}$ and $\{\beta_t\}$. Thus we can unwrap the disturbances

$$\begin{array}{rcll} \eta_t &=& \mu_{t+1}-\mu_t+\beta_t & \sim \text{NID}(0,\sigma_\eta^2), \\ \zeta_t &=& \beta_{t+1}-\beta_t & \sim \text{NID}(0,\sigma_\zeta^2), \\ \varepsilon_t &=& y_t-\mu_t & \sim \text{NID}(0,\sigma_\varepsilon^2). \end{array}$$

Let the prior densities, for some choices of shape parameters $c_\varepsilon, c_\eta, c_\zeta$ and scales $S_{\sigma_\varepsilon}, S_{\sigma_\eta}, S_{\sigma_\zeta}$, be given by

$$\sigma_\varepsilon^2 \sim \mathcal{IG}\left(\frac{c_\varepsilon}{2},\frac{S_{\sigma_\varepsilon}}{2}\right), \qquad \sigma_\eta^2 \sim \mathcal{IG}\left(\frac{c_\eta}{2},\frac{S_{\sigma_\eta}}{2}\right), \qquad \sigma_\zeta^2 \sim \mathcal{IG}\left(\frac{c_\zeta}{2},\frac{S_{\sigma_\zeta}}{2}\right).$$

For example, the inverse gamma distribution $\mathcal{IG}$ for σ_ε^2 implies that the prior mean and variance of σ_ε^2 is given by

$$\frac{S_{\sigma_\varepsilon}}{c_\varepsilon-2}, \qquad \frac{2S_{\sigma_\varepsilon}^2}{(c_\varepsilon-2)^2(c_\varepsilon-4)},$$

respectively. The posteriors are then given by

$$\begin{array}{rcl} \sigma_\varepsilon^2|y,\alpha &\sim& \mathcal{IG}\left(\frac{c_\varepsilon+n}{2},\frac{S_{\sigma_\varepsilon}+\sum\varepsilon_t^2}{2}\right), \quad \sigma_\eta^2|y,\alpha \sim \mathcal{IG}\left(\frac{c_\eta+n}{2},\frac{S_{\sigma_\eta}+\sum\eta_t^2}{2}\right), \\ \sigma_\zeta^2|y,\alpha &\sim& \mathcal{IG}\left(\frac{c_\zeta+n}{2},\frac{S_{\sigma_\zeta}+\sum\zeta_t^2}{2}\right). \end{array}$$

Each of these densities are easy to sample from as shown in the *Ox* example program.

Although it is not always possible to sample the $\varphi|y,\alpha$ as easily as this, it is usually the case that it is much easier to update the parameters having augmented the MCMC with the states than when the states are integrated out. Of course it is often true that the MCMC algorithm is in such a large dimension that the algorithm will only converge quite slowly. This danger needs to be assessed carefully in applied work.

Example The Bayesian procedure for σ_ε^2 and σ_η^2 is implemented for the local level model (31) using the Nile data of Appendix A. The prior density parameters are set to $c_\eta = c_\varepsilon = 5$ and $S_\eta = S_\varepsilon = 5000$.

Ox code of `ssfbayes.ox`:

```
#include <oxstd.h>
#include <oxprob.h>
#include "SsfPack.h"

main()
{
 decl k, i, j, cIter, cInit, mY, mPhi, mOmega, mSigma;
 decl S_eta, S_eps, c_eta, c_eps, mD, mPsi;

 mY = loadmat("Nile.dat")';
```

```
 GetSsfStsm(<CMP_LEVEL, 0.0, 0, 0;
            CMP_IRREG, 0.0, 0, 0>, &mPhi, &mOmega, &mSigma);

 mOmega[0][0] = mOmega[1][1] = 1.0;
 S_eta = S_eps = 5000;
 c_eta = c_eps = 2.5 + (0.5 * columns(mY));

 cIter = 1000; cInit = 200;
 mPsi = zeros(2, cIter);
 for (i=0; i<cIter; i++)
 {
  mD = SsfCondDens(DS_SIM, mY, mPhi, mOmega, mSigma);
  mD = mD * mD';
  mPsi[0][i] = 1.0 / rangamma(1,1, c_eta, (S_eta + mD[0][0])/2);
  mPsi[1][i] = 1.0 / rangamma(1,1, c_eps, (S_eps + mD[1][1])/2);
  mOmega = diag(mPsi[][i]);
 }
 print("%r", {"var_eta", "var_eps"},
       "%c", {"mean", "st.dev."}, "%15.3f",
       meanr(mPsi[][cInit:cIter-1])~
       sqrt(varr(mPsi[][cInit:cIter-1])));
}
```

Ox output:

```
                        mean        st.dev.
var_eta             1805.559       1064.927
var_eps            14125.302       2707.487
```

7 Header files of C functions

Here we document the header files of four `C` functions which can be used in other environments such as Gauss or S+ and for other programming compilers. The functions relate to the Kalman filter and the associated smoothing algorithm and two functions relate to the simulation smoother. The function calls are given and the details of the variables are given. Input and output information for the variables is provided: the type of variable, indication whether the variable NULL is allowed as input and a description of the variables are given. The details of variable types are given below.

Type	C definition	Description
B	`boolean`	value 0 or 1
D	`double`	double precision real value
I	`int`	integer value
V	`*double`	vector of double values
M	`**double`	matrix of double values
IV	`*int`	vector of integer values
IM	`**int`	matrix of integer values
pD	`&double`	address (pointer) to double value
pI	`&int`	address (pointer) to integer value

7.1 Kalman filter

The Kalman filter is discussed in section 4.2. The function call is

```
fMultiKalmanFil(cY, cT, mY, cSt, mPhi, mOmega, vDelta,
                imPhi, imOmega, ivDelta, mTV,
                mSigma, mP, mN,
                mKF, pdLogF, pdVar, piTmD);
```

with return value of type boolean (B) and it returns 0 if F_t has become a negative definite matrix. Information about the arguments are given below.

Variable	I/O	Type	NULL	Description
`cY`	I	I	no	N, number of rows data matrix
`cT`	I	I	no	n, number of columns data matrix
`mY`	I	M	no	data matrix $(y_1, \ldots y_n)$
`cSt`	I	I	no	m, dimension of state vector
`mPhi`	I	M	no	Φ, system matrix associated with T and Z
`mOmega`	I	M	no	Ω, system matrix associated with H and G
`vDelta`	I	V	yes	δ, vector of constants associated with d and c
`imPhi`	I	IM	yes	J_Φ, index matrix for time-varying Φ
`imOmega`	I	IM	yes	J_Ω, index matrix for time-varying Ω
`ivDelta`	I	IV	yes	J_δ, index vector for time-varying δ
`mTV`	I	M	yes	data matrix with time-varying values
`mSigma`	I	M	yes	Σ, initial state matrix $(P', a')'$
`mP`	I	M	no	$p + 1 \times p$ workspace matrix
`mN`	I	M	no	$p + 2 \times p$ workspace matrix
`mKF`	I	M	yes	$k \times n$ workspace matrix
	O	M		$k \times n$ data matrix with Kalman filter output
`pdLogF`	I	pD	yes	address to double
	O	pD		address to double 'log-likelihood'
`pdVar`	I	pD	yes	address to double
	O	pD		address to double 'prediction error variance'
`piTmD`	I	pI	yes	address to integer
	O	pI		address to initial integer

with $p = m + N$ and k as given in section 4.2.

7.2 Smoothing

The smoothing algorithm is discussed in section 4.3. Smoothing can be started after the Kalman filter. The function call is

```
void MultiKalmanSmo(cY, cT, cSt, mPhi, imPhi, mTV,
                    mP, mN, mKF, mSco, mKS)
```

with no return value. Information about the arguments are given below.

Variable	I/O	Type	NULL	Description
cY	I	I	no	N, number of rows data matrix
cT	I	I	no	n, number of columns data matrix
cSt	I	I	no	m, dimension of state vector
mPhi	I	M	no	Φ, system matrix associated with T and Z
imPhi	I	IM	yes	J_Φ, index matrix for time-varying Φ
mTV	I	M	yes	data matrix with time-varying values
mP	I	M	no	$p+1 \times p$ workspace matrix
mN	I	M	no	$p+2 \times p$ workspace matrix
mKF	I	M	no	data matrix with Kalman filter output
mSmo	I	M	yes	$m+N \times m+N$ workspace matrix
	O	M		$m+N \times m+N$score matrix S
mKS	I	M	yes	$k \times n$ workspace matrix
	O	M		$k \times n$ data matrix with smoothing output

with $p = m + N$, k as given in section 4.3 and S as given in section 5.1.

7.3 Simulation smoothing: weights

The simulation smoothing algorithm is discussed in section 4.4. Simulation smoothing can be started after the Kalman filter. The function call for calculating the simulation weights is

```
int iMultiSimSmoWgt(cY, cT, cSt, mPhi, mOmega, imPhi, imOmega, mTV,
                    mP, mN, mKF,
                    cSel, ivSel, mC, mWeight);
```

with return value of type integer (I) and it returns 0 (no error), 1 (C_t is negative definite), 2 (C_t is singular) or 3 (C_t is NULL). Information about the arguments are given below.

Variable	I/O	Type	NULL	Description
cY	I	I	no	N, number of rows data matrix
cT	I	I	no	n, number of columns data matrix
cSt	I	I	no	m, dimension of state vector
mPhi	I	M	no	Φ, system matrix associated with T and Z
mOmega	I	M	no	Ω, system matrix associated with H and G
imPhi	I	IM	yes	J_Φ, index matrix for time-varying Φ
imOmega	I	IM	yes	J_Ω, index matrix for time-varying Ω
mTV	I	M	yes	data matrix with time-varying values
mP	I	M	no	$p+1 \times p$ workspace matrix
mN	I	M	no	$p+2 \times p$ workspace matrix
mKF	I	M	no	data matrix with Kalman filter output
cSel	I	I	no	s, number of required simulations
ivSel	I	IV	no	$p \times 1$ vector of diagonal 0/1 elements of Λ
mC	I	M	no	$s \times s$ workspace matrix
mWeight	I	M	yes	$k \times n$ workspace matrix
	O	M		$k \times n$ data matrix with simulation weights

with $p = m + N$ and k and s as given in section 4.4.

7.4 Simulation smoothing: draws

The simulation smoothing algorithm is discussed in section 4.4. This function to obtain draws from the simulation smoother can be started after the Kalman filter and after the function for the simulation weight matrix. The function call for drawing is

```
void MultiSimSmoDraw(cY, cT, cSt, mPhi, mOmega, imPhi, imOmega, mTV,
                     mP, mN, mKF,
                     cSel, ivSel, mC, mWeight,
                     mRann, mDraw);
```

with no return value. Information about the arguments are given below.

Variable	I/O	Type	NULL	Description
`cY`	I	I	no	N, number of rows data matrix
`cT`	I	I	no	n, number of columns data matrix
`cSt`	I	I	no	m, dimension of state vector
`mPhi`	I	M	no	Φ, system matrix associated with T and Z
`mOmega`	I	M	no	Ω, system matrix associated with H and G
`imPhi`	I	IM	yes	J_Φ, index matrix for time-varying Φ
`imOmega`	I	IM	yes	J_Ω, index matrix for time-varying Ω
`mTV`	I	M	yes	data matrix with time-varying values
`mP`	I	M	no	$p + 1 \times p$ workspace matrix
`mN`	I	M	no	$p + 2 \times p$ workspace matrix
`mKF`	I	M	no	data matrix with Kalman filter output
`cSel`	I	I	no	s, number of required simulations
`ivSel`	I	IV	no	$p \times 1$ vector of diagonal 0/1 elements of Λ
`mC`	I	M	no	$s \times s$ workspace matrix
`mWeight`	I	M	no	data matrix with simulation weights
`mRann`	I	M	no	$s \times n$ data matrix with standard normal deviates
`mDraw`	I	M	yes	$p \times n + 1$ workspace matrix
	O	M		$p \times n + 1$ data matrix with simulation draws

with $p = m + N$ and s as given in section 4.4.

Appendices

A Nile data

The Nile data is a series of readings of the annual flow volume of the Nile river at Aswan for 1871 to 1970. This series is originally considered by Cobb (1978) and it is analysed more recently by Balke (1993). The observations are given below (read row-wise).

```
1120.0    1160.0    963.00    1210.0    1160.0    1160.0
813.00    1230.0    1370.0    1140.0    995.00    935.00
1110.0    994.00    1020.0    960.00    1180.0    799.00
958.00    1140.0    1100.0    1210.0    1150.0    1250.0
1260.0    1220.0    1030.0    1100.0    774.00    840.00
874.00    694.00    940.00    833.00    701.00    916.00
692.00    1020.0    1050.0    969.00    831.00    726.00
456.00    824.00    702.00    1120.0    1100.0    832.00
```

764.00	821.00	768.00	845.00	864.00	862.00
698.00	845.00	744.00	796.00	1040.0	759.00
781.00	865.00	845.00	944.00	984.00	897.00
822.00	1010.0	771.00	676.00	649.00	846.00
812.00	742.00	801.00	1040.0	860.00	874.00
848.00	890.00	744.00	749.00	838.00	1050.0
918.00	986.00	797.00	923.00	975.00	815.00
1020.0	906.00	901.00	1170.0	912.00	746.00
919.00	718.00	714.00	740.00		

B Airline data

This is a well-known data set consisting of the number of UK airline passengers in thousands from January 1949 to December 1960; see Box and Jenkins (1976). The observations are given below (read row-wise).

112.00	118.00	132.00	129.00	121.00	135.00
148.00	148.00	136.00	119.00	104.00	118.00
115.00	126.00	141.00	135.00	125.00	149.00
170.00	170.00	158.00	133.00	114.00	140.00
145.00	150.00	178.00	163.00	172.00	178.00
199.00	199.00	184.00	162.00	146.00	166.00
171.00	180.00	193.00	181.00	183.00	218.00
230.00	242.00	209.00	191.00	172.00	194.00
196.00	196.00	236.00	235.00	229.00	243.00
264.00	272.00	237.00	211.00	180.00	201.00
204.00	188.00	235.00	227.00	234.00	264.00
302.00	293.00	259.00	229.00	203.00	229.00
242.00	233.00	267.00	269.00	270.00	315.00
364.00	347.00	312.00	274.00	237.00	278.00
284.00	277.00	317.00	313.00	318.00	374.00
413.00	405.00	355.00	306.00	271.00	306.00
315.00	301.00	356.00	348.00	355.00	422.00
465.00	467.00	404.00	347.00	305.00	336.00
340.00	318.00	362.00	348.00	363.00	435.00
491.00	505.00	404.00	359.00	310.00	337.00
360.00	342.00	406.00	396.00	420.00	472.00
548.00	559.00	463.00	407.00	362.00	405.00
417.00	391.00	419.00	461.00	472.00	535.00
622.00	606.00	508.00	461.00	390.00	432.00

References

Anderson, B. D. O. and J. B. Moore (1979). *Optimal Filtering*. Englewood Cliffs: Prentice-Hall.

Ansley, C. F. and R. Kohn (1986). A note on reparameterizing a vector autoregressive moving average model to enforce stationarity. *J. Statistical Computation and Simulation 24*, 99–106.

Balke, N. S. (1993). Detecting level shifts in time series. *J. Business and Economic Statist. 11*, 81–92.

Bergstrom, A. R. (1984). Gaussian estimation of structural parameters in higher order continuous time dynamic models. In Z. Griliches and M. Intriligator (Eds.), *The Handbook of Econometrics, Volume 2*, pp. 1145–1212. North-Holland.

Box, G. E. P. and G. M. Jenkins (1976). *Time Series Analysis: Forecasting and Control* (2nd ed.). San Francisco, CA: Holden-Day.

Chib, S. and E. Greenberg (1995). Understanding the Metropolis-Hastings algorithm. *The American Statistican 49*, 327–35.

Chib, S. and E. Greenberg (1996). Markov chain Monte Carlo simulation methods in econometrics. *Econometric Theory 12*, 409–31.

Cobb, G. W. (1978). The problem of the Nile: conditional solution to a change point problem. *Biometrika 65*, 243–51.

de Jong, P. (1988a). A cross validation filter for time series models. *Biometrika 75*, 594–600.

de Jong, P. (1988b). The likelihood for a state space model. *Biometrika 75*, 165–169.

de Jong, P. (1989). Smoothing and interpolation with the state space model. *J. Am. Statist. Assoc. 84*, 1085–8.

de Jong, P. and J. Penzer (1998). Diagnosing shocks in time series. *J. Am. Statist. Assoc. 93*. Forthcoming.

de Jong, P. and N. Shephard (1995). The simulation smoother for time series models. *Biometrika 82*, 339–50.

Doornik, J. A. (1996). *Ox: Object Oriented Matrix Programming, 1.10*. London: Chapman & Hall.

Fruhwirth-Schnatter, S. (1994). Data augmentation and dynamic linear models. *J. Time Series Analysis 15*, 183–202.

Gelfand, A. E. and A. F. M. Smith (1990). Sampling-based approaches to calculating marginal densities. *J. Am. Statist. Assoc. 85*, 398–409.

Geman, S. and D. Geman (1984). Stochastic relaxation, Gibbs distribution and the Bayesian restoration of images. *IEEE Transactions, PAMI 6*, 721–41.

Gilks, W. K., S. Richardson, and D. J. Spiegelhalter (1996). *Markov Chain Monte Carlo in Practice*. London: Chapman & Hall.

Gilks, W. R., N. G. Best, and K. K. C. Tan (1995). Adaptive rejection Metropolis sampling within Gibbs sampling. *Applied Statistics 44*, 155–73.

Green, P. and B. W. Silverman (1994). *Nonparameteric Regression and Generalized Linear Models: A Roughness Penalty Approach*. London: Chapman & Hall.

Harrison, J. and C. F. Stevens (1976). Bayesian forecasting (with discussion). *J. R. Statist. Soc. B 38*, 205–247.

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

Harvey, A. C. (1993). *Time Series Models* (2nd ed.). Hemel Hempstead: Harvester Wheatsheaf.

Harvey, A. C. and S. J. Koopman (1992). Diagnostic checking of unobserved components time series models. *J. Business and Economic Statist. 10*, 377–389.

Harvey, A. C., S. J. Koopman, and J. Penzer (1998). Messey time series. In T. B. Fomby and R. C. Hill (Eds.), *Advances in Econometrics, volume 13*. New York: JAI Press.

Harvey, A. C. and M. Streibel (1998). Testing for nonstationary unobserved components. *J. Time Series Analysis 19*. Forthcoming.

Hastie, T. and R. Tibshirani (1990). *Generalized Additive Models*. London: Chapman & Hall.

Hastings, W. K. (1970). Monte-carlo sampling methods using markov chains and their applications. *Biometrika 57*, 97–109.

Jones, R. H. (1980). Maximum likelihood fitting of ARIMA models to time series with missing observations. *Technometrics 22*, 389–95.

Kitagawa, G. and W. Gersch (1996). *Smoothness Priors Analysis of Time Series*. New York: Springer Verlag.

Kohn, R. and C. F. Ansley (1987). A new algorithm for spline smoothing based on smoothing a stochastic process. *SIAM J Sci. Statistcal Computing 8*, 33–48.

Kohn, R. and C. F. Ansley (1989). A fast algorithm for signal extraction, influence and cross-validation. *Biometrika 76*, 65–79.

Koopman, S. J. (1993). Disturbance smoother for state space models. *Biometrika 80*, 117–126.

Koopman, S. J. (1997). Kalman filtering and smoothing. In P. Armitage and T. Colton (Eds.), *Encyclopedia of Biostatistics*. Chichester: Wiley and Sons.

Koopman, S. J. and N. Shephard (1992). Exact score for time series models in state space form. *Biometrika 79*, 823–6.

Magnus, J. R. and H. Neudecker (1988). *Matrix Differential Calculus with Applications in Statistics and Econometrics*. New York: Wiley.

Metropolis, N., A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller (1953). Equations of state calculations by fast computing machines. *J Chemical Physics 21*, 1087–92.

Nyblom, J. and T. Makelainen (1983). Comparison of tests of for the presence of random walk coefficients in a simple linear models. *J. Am. Statist. Assoc. 78*, 856–64.

Ripley, B. D. (1977). Modelling spatial patterns (with discussion). *J. R. Statist. Soc. B 39*, 172–212.

Schweppe, F. (1965). Evaulation of likelihood functions for Gaussian signals. *IEEE Transactions on Information Theory 11*, 61–70.

Tanaka, K. (1983). Non-normality of the Lagrange multiplier statistics for testing the constancy of regression coefficients. *Econometrica 51*, 1577–82.

Tanaka, K. (1996). *Time Series Analysis: Nonstationary and Noninvertible Distribution Theory*. New York: Wiley.

Tanner, M. A. (1996). *Tools for Statistical Inference: methods for exploration of posterior distributions and likelihood functions* (3 ed.). New York: Springer-Verlag.

Tanner, M. A. and W. H. Wong (1987). The calculation of posterior distributions by data augmentation (with discussion). *J. Am. Statist. Assoc. 82*, 528–50.

West, M. and J. Harrison (1997). *Bayesian Forecasting and Dynamic Models* (2 ed.). New York: Springer-Verlag.