`dlm`: an R package for Bayesian analysis of Dynamic Linear Models

Giovanni Petris
University of Arkansas, Fayetteville AR

2006-06-15

1 Introduction

The package `dlm` focuses on Bayesian analysis of Dynamic Linear Models (DLM), also known as linear state space models. The package also includes functions for maximum likelihood estimation of the parameters of a DLM and for Kalman filtering. The algorithms used for Kalman filtering, likelihood evaluation, and sampling from the state vectors are based on the singular value decomposition (SVD) of the relevant variance matrices (see [ZL]), which improves numerical stability over other algorithms.

The notation used for the model follows closely that used in [WH], and is as follows.

$$\begin{cases} y_t = F_t\theta_t + v_t & v_t \sim \mathcal{N}(0, V_t) \\ \theta_t = G_t\theta_{t-1} + w_t & w_t \sim \mathcal{N}(0, W_t) \end{cases}$$

for $t = 1, \ldots, n$. The initial distribution (prior) is specified by

$$\theta_0 \sim \mathcal{N}(m_0, C_0).$$

So (θ_t) is a sequence of unobservable *state vectors* and (y_t) is a sequence of (vector-valued) observations; (v_t) and (w_t) are independent sequences (within and between). We will denote by $\mathcal{Y}_t$ the observations up to time t, with $\mathcal{Y}_0 = \emptyset$.

The package introduces a class `dlm` to represent DLMs and provides functions to create several standard types of DLMs, such as polynomial, ARMA, regression, and seasonal models. Components of a model (e.g., trend and seasonal) can be created using those functions and added together to produce a model for the data. A class `dlmFiltered` is also defined, and method functions for `residuals` and `tsdiag` are available for computing one-step forecast errors and drawing standard diagnostic plots.

2 Class `dlm` and creator functions

2.1 Constant models

Many interesting DLMs are *time invariant*, that is, F_t, V_t, G_t, and W_t do not change over time. An object of class `dlm` representing a time-invariant DLM is essentially a list containing named components `FF`, `V`, `GG`, `W`, `m0`, `C0`, with the class attribute `"dlm"`. While `m0` can be a vector or a matrix, all the remaining components must be matrices. The function `dlm` can be used to create an object of class `dlm` from a list. It includes some basic consistency checks on the dimensions of the involved matrices.

A polynomial model of order two, sometimes called a local linear trend model, is defined by the following matrices:

$$F = \begin{bmatrix} 1 & 0 \end{bmatrix}, \qquad V = \begin{bmatrix} \sigma^2 \end{bmatrix},$$
$$G = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \qquad W = \begin{bmatrix} 0 & 0 \\ 0 & \tau \end{bmatrix}.$$

In R, taking $\sigma^2 = 1.4$ and $\tau = 0.2$, an object representing this model can be created as follows.

```
> FF <- matrix(c(1, 0), 1)
> V <- matrix(1.4)
> GG <- matrix(c(1, 0, 1, 1), 2)
> W <- matrix(c(0, 0, 0, 0.2), 2)
> mod <- dlm(list(FF = FF, V = V, GG = GG, W = W, m0 = rep(0, 2),
+     C0 = 1e+07 * diag(2)))
> mod

$FF
     [,1] [,2]
[1,]    1    0

$V
     [,1]
[1,]  1.4

$GG
     [,1] [,2]
[1,]    1    1
[2,]    0    1
```

```
$W
     [,1] [,2]
[1,]    0  0.0
[2,]    0  0.2

$m0
[1] 0 0

$C0
      [,1]  [,2]
[1,] 1e+07 0e+00
[2,] 0e+00 1e+07
```

The same model could have been created more simply as `dlmModPoly(dV=1.4, dW=c(0,0.2))`. The arguments `dV` and `dW` are used to provide the diagonals of the matrices V and W. If a more general, i.e., non-diagonal variance matrix, is desired, then one need to use `dlm`. The default for `dlmModPoly` is to create a second-order polynomial model, but the order of the model can be specified via the first argument `order`.

A seasonal DLM with s seasons can be obtained with the function `dlmModSeas`, which produces the dummy variable version of the seasonal DLM (see [H]). For quarterly data, for example, one would use something like the following:

```
> dlmModSeas(4, dV = 1.4, dW = c(0.2, 0, 0))

$FF
     [,1] [,2] [,3]
[1,]    1    0    0

$V
     [,1]
[1,]  1.4

$GG
     [,1] [,2] [,3]
[1,]   -1   -1   -1
```

```
[2,]    1    0    0
[3,]    0    1    0

$W
     [,1] [,2] [,3]
[1,]  0.2    0    0
[2,]  0.0    0    0
[3,]  0.0    0    0

$m0
[1] 0 0 0

$C0
      [,1]  [,2]  [,3]
[1,] 1e+07 0e+00 0e+00
[2,] 0e+00 1e+07 0e+00
[3,] 0e+00 0e+00 1e+07
```

In many applications, for example in business and economics, a model including both a local linear trend and a seasonal component is often used. Combining component models is easily achieved via the method `dlm` of the function `"+"`:

```
> dlmModPoly(dV = 1.4, dW = c(0, 0.2)) + dlmModSeas(4, dV = 0,
+     dW = c(0.1, 0, 0))
```

An alternative representation of seasonal components is the one based on trigonometric functions, see [WH] for details. Trigonometric functions may also be employed to describe a periodic function with a generic period[1], such as those used to explain the business cycle. There is a subtle difference between the two concepts. We think of a seasonal component as a periodic function defined on the integers and having period s; its Fourier representation as a sum of harmonics has only a finite number of terms, namely $\lfloor s/2 \rfloor$. On the other hand, a periodic function with noninteger period is periodic only when viewed as a function on the real line and, in this case, its Fourier representation contains an infinite number of harmonics. Package `dlm` provides the creator function `dlmModTrig` to deal with both cases,

[1] By this we mean that the period in not an integer multiple of the sampling interval

using different sets of arguments. To create a seasonal component DLM, one specifies the integer period s (number of seasons) via the argument `s`. Optionally, the argument `q` can be used to set the number of harmonics to retain in the model (the default value is all, i.e., $\lfloor s/2 \rfloor$). For a cycle-type periodic DLM, on the other hand, one specifies the noninteger period τ via the argument `tau`. Since in this case the Fourier reresentation is an infinite sum, the argument `q` for the number of harmonics to retain has to be specified. Instead of the period, one may alternatively specify the frequency $\omega = 2\pi/\tau$ via the argument `om`. The following example illustrates the creation of a seasonal model for quarterly data, and of a model for a cycle having period $\tau = 8.4$. In both cases we retain only the first two harmonics. Note the structure of the system variance W.

```
> dlmModTrig(s = 4, q = 2, dV = 1.4, dW = 0.2)
> dlmModTrig(tau = 8.4, q = 2, dV = 1.4, dW = 0.2)
```

Autoregressive moving average (ARMA) models can be represented in many different ways as DLMs. The representation used by the creator function `dlmModARMA` is the one described below. Consider a zero-mean ARMA model described by the equation

$$y_t = \sum_{i=1}^{p} \phi_i y_{t-i} + \sum_{i=1}^{q} \psi_i \epsilon_{t-i} + \epsilon_t, \qquad \epsilon_t \overset{\text{iid}}{\sim} \mathcal{N}(0, \sigma^2).$$

Let $r = \max\{p, q+1\}$, and set $\phi_j = 0$ for $j > p$ and $\psi_j = 0$ for $j > q$. Define the matrices

$$\begin{aligned} F &= \begin{bmatrix} 1 & 0 & \dots & 0 \end{bmatrix}, \\ G &= \begin{bmatrix} \phi_1 & 1 & 0 & \dots & 0 \\ \phi_2 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & & \ddots & \\ \phi_{r-1} & 0 & \dots & 0 & 1 \\ \phi_r & 0 & \dots & 0 & 0 \end{bmatrix}, \\ R &= \begin{bmatrix} 1 & \psi_1 & \dots & \psi_{r-2} & \psi_{r-1} \end{bmatrix}'. \end{aligned} \tag{1}$$

If one introduces an r-dimensional state vector θ_t, whose components are defined by

$$\begin{aligned} \theta_{t1} &= y_t, \\ \theta_{t,j+1} &= \sum_{i=j+1}^{r} \phi_i y_{t+j-i} + \sum_{i=j}^{r-1} \psi_i \epsilon_{t+j-i}, \qquad j = 1, 2, \dots, r-1, \end{aligned} \tag{2}$$

then the given ARMA model has the following DLM representation:

$$\begin{cases} y_t = F\theta_t, \\ \theta_t = G\theta_{t-1} + R\epsilon_t. \end{cases} \tag{3}$$

This is a DLM with $V = 0$ and $W = RR'\sigma^2$. For example, consider the ARMA(2,1) model expressed by

$$y_t = 0.8y_{t-1} - 0.2y_{t-2} + 0.3\epsilon_{t-1} + \epsilon_t, \qquad \epsilon_t \overset{\text{iid}}{\sim} \mathcal{N}(0, 3.2).$$

Its DLM representation can be obtained with the command:

```
> dlmModARMA(ar = c(0.8, -0.2), ma = 0.3, sigma2 = 3.2)
```

ARMA models for multivariate, m-dimensional, observations, are formally defined as in the univariate case, through the recursive relation

$$y_t = \sum_{j=1}^{p} \Phi_j y_{t-j} + \epsilon_t + \sum_{j=1}^{q} \Psi_j \epsilon_{t-j}, \tag{4}$$

where (ϵ_t) is an m-variate Gaussian white noise sequence with variance Σ and the Φ_j and Ψ_j are m by m matrices. Here, without loss of generality, we have taken the mean of the process to be zero. A DLM representation of a multivariate ARMA process can be formally obtained by a simple generalization of the representation given for univariate ARMA processes. Namely, in the G matrix each ϕ_j needs to be replaced by a block containing the matrix Φ_j; similarly for the ψ_j in the matrix R, that have to be replaced by Ψ_j blocks. Finally, all the occurrences of a "one" in F, G, and R must be replaced by the identity matrix of order m, and all the occurrences of a "zero" with a block of zeroes of order m by m. For example, let us consider the bivariate ARMA(1,1) process

$$y_t = \Phi_1 y_{t-1} + +\epsilon_t + \Psi_1 \epsilon_{t-1}, \qquad \epsilon_t \sim \mathcal{N}(0, \Sigma), \tag{5}$$

with

$$\Phi_1 = \begin{bmatrix} 1.2 & -0.5 \\ 0.6 & 0.3 \end{bmatrix}, \qquad \Psi_1 = \begin{bmatrix} -0.6 & 0.3 \\ 0.2 & 0.5 \end{bmatrix}, \qquad \Sigma = \begin{bmatrix} 1.00 & 0.50 \\ 0.50 & 1.25 \end{bmatrix}. \tag{6}$$

Then the system and observation matrices needed to define the DLM representation of (5) are the following:

$$
\begin{aligned}
F &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, \\
G &= \begin{bmatrix} 1.2 & -0.5 & 1 & 0 \\ 0.6 & 0.3 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \\
R &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ -0.6 & 0.3 \\ 0.2 & 0.5 \end{bmatrix}, \qquad W = R\Sigma R'.
\end{aligned}
\tag{7}
$$

In R, the DLM representation would be obtained with the command

```
> dlmModARMA(ar = list(matrix(c(1.2, 0.6, -0.5, 0.3), 2)),
+     ma = list(matrix(c(-0.6, 0.2, 0.3, 0.5), 2)),
+     sigma2 = matrix(c(1, 0.5, 0.5, 1.25), 2))

$FF
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0

$V
     [,1] [,2]
[1,]    0    0
[2,]    0    0

$GG
     [,1] [,2] [,3] [,4]
[1,]  1.2 -0.5    1    0
[2,]  0.6  0.3    0    1
[3,]  0.0  0.0    0    0
[4,]  0.0  0.0    0    0

$W
      [,1]  [,2]    [,3]    [,4]
[1,]  1.00 0.500 -0.4500  0.4500
```

```
[2,]  0.50 1.250  0.0750  0.7250
[3,] -0.45 0.075  0.2925 -0.0525
[4,]  0.45 0.725 -0.0525  0.4525

$m0
[1] 0 0 0 0

$C0
      [,1]  [,2]  [,3]  [,4]
[1,] 1e+07 0e+00 0e+00 0e+00
[2,] 0e+00 1e+07 0e+00 0e+00
[3,] 0e+00 0e+00 1e+07 0e+00
[4,] 0e+00 0e+00 0e+00 1e+07
```

2.2 Time-varying models

For time-varying DLM, the structure of an object of class `"dlm"` is more complicated. The components `FF, V, GG, W` are still part of the structure: they serve the two-fold purpose of implicitely specifying the dimensions of the system and observation matrices, and to set the value of the elements that stay constant over time. The time-varying elements are specified via an additional *data matrix* `X`, and auxiliary *indicator* matrices `JFF, JV, JGG, JW`. Any one of the auxiliary matrices may be substituted by a `NULL` component of the same name in case the corresponding matrix is time invariant. For example, a component `JFF` which is either `NULL` or nonexistent signals that the matrix `FF` does not vary over time. Otherwise, the matrix `JFF` must have integer entries and be of the same dimensions as `FF`. A zero entry in `JFF` signals that the corresponding entry of `FF` is constant, while an entry j signals that the jth column of `X` contains the values of the corresponding entry of `FF`:

`JFF[i,j] == 0`	$\Longrightarrow$	`FF[i,j]` is constant in time,
`JFF[i,j] == r`	$\Longrightarrow$	`FF[i,j] == X[n,r]` at time `n`.

Similarly for the other matrices and their indicator matrices. The prior mean, `m0`, and variance, `C0`, retain their meaning.

Regression models (static or dynamic) can be cast into the DLM framework. The state vector θ_t has the role of the vector of regression parameters

at time t, while the matrix F_t contains the covariates for case/time t. The creator function `dlmModReg` is used to set up univariate regression models:

```
> z <- rnorm(30)
> mod <- dlmModReg(z)
> mod

$FF
     [,1] [,2]
[1,]    1    1

$V
     [,1]
[1,]    1

$GG
     [,1] [,2]
[1,]    1    0
[2,]    0    1

$W
     [,1] [,2]
[1,]    0    0
[2,]    0    0

$JFF
     [,1] [,2]
[1,]    0    1

$X
     [,1]
[1,] 0.6387
[2,] 0.9474
[3,] ...

$m0
[1] 0 0

$C0
      [,1]  [,2]
```

```
[1,] 1e+07 0e+00
[2,] 0e+00 1e+07
```

The default for `dlmModReg` is to add an intercept and to set up a static regression model. Note that the `dlm` method of `print` does not show the entire data matrix. In case one wants to see the whole thing, `mod$X` or `print.default(mod)` will do.

3 Maximum Likelihood estimation

This function evaluates the MLE of any unknown parameters in the matrices defining the DLM - except for m_0 and C_0, which must be specified completely. Strictly speaking, since we are including the prior information about the initial state vector, summarized by m_0 and C_0, the model, and so the MLE, is slightly different from the usual frequentist formulation, which typically assumes for θ_0 a diffuse prior (see [H]); however, by specifying a very large C_0, there is no noticeable difference in the MLE.

The main arguments of `dlmMLE` are the data, `y`, a vector, matrix, or a time series, the initial value of the unknown (vector) parameter for the optimization routine, `parm`, and a function, `build`, that from a vector of the same length as `parm` outputs a `"dlm"` object (or a list that can be coherced to `"dlm"`). The use of the argument `build` makes the function very flexible in finding MLE for very general DLM. `dlmMLE` calls `optim` internally and returns its output. In evaluating the log likelihood of the given model, `dlmMLE` uses a robust algorithm based on SVD of the relevant variance matrices. This algorithm needs the observation matrix V to be nonsingular, while there is no such restriction on W. Due to the stability of SVD, a very small diagonal matrix can be added to a singular V to approximate a model with singular observation variance.

Example

The data set `NelPlo` is a bivariate time series containing the first difference of the log of industrial production and stock prices, in percent, for a number of years. It is a subset of the data set with the same name in package `tseries`. The model we consider is a seemingly unrelated time series model, specified by the matrices

$$F = G = \mathrm{diag}(1, 1).$$

The variances V and W need to be estimated from the data. Since we are not assuming the two are diagonal, and three parameters are nedeed to specify each of them, we need to set up a function that "builds" V and W from a vector of length 6. One possibility is to parametrize the variances (diagonal elements) in V and W in terms of the log of their square root, and express the correlations in term of tanh of other two parameters. This results in the following function definition:

```
> buildSu <- function(x) {
+     a <- diag(exp(0.5 * x[1:2]), nr = 2)
+     a[1, 2] <- x[3]
+     V <- crossprod(a)
+     a <- diag(exp(0.5 * x[4:5]), nr = 2)
+     a[1, 2] <- x[6]
+     W <- crossprod(a)
+     return(dlm(m0 = rep(0, 2), C0 = 1e+07 * diag(2), FF = diag(2),
+         GG = diag(2), V = V, W = W))
+ }
```

One can the call `dlmMLE`. It is always advisable to check the `convergence` component of the returned value: a nonzero value signals that something went wrong in the optimization process. The estimated variance matrices can be recovered using the same `build` function used in the optimization process (actually, in order to avoid errors, this is the recommended way of proceeding!).

```
> data(NelPlo)
> suMLE <- dlmMLE(NelPlo, rep(0, 6), buildSu)
> suMLE$convergence

[1] 0


> buildSu(suMLE$par)[c("V", "W")]

$V
      [,1]  [,2]
[1,] 2.391 0.638
[2,] 0.638 9.028

$W
```

```
        [,1]   [,2]
[1,] 0.00049 0.0040
[2,] 0.00400 0.0327
```

The function `stats::StructTS` can be used to fit a model with the same type of dynamics, but only for univariate observations. Doing this for the two series separately amounts to fitting a seemingly unrelated time series model with the additional contraint that V and W are diagonal. We try, for comparison purposes.

```
> StructTS(NelPlo[, 1], type = "level")


Call:
StructTS(x = NelPlo[, 1], type = "level")

Variances:
  level  epsilon
   0.00     2.39


> StructTS(NelPlo[, 2], type = "level")


Call:
StructTS(x = NelPlo[, 2], type = "level")

Variances:
  level  epsilon
 0.0324   9.0308
```

The estimates are in fairly good agreement, which also shows that using the value specified above for C_0 probably did not affect much the MLE.

We try now, for the same data set, to fit the same model with the additional constraint $W = qV$, where q is a nonnegative scalar, the so-called signal-to-noise ratio. This results in a local level model with a homogeneity restriction. A different `build` function has to be defined, but otherwise the constant matrices remain the same as before.

```
> buildHo <- function(x) {
+     a <- diag(exp(0.5 * x[1:2]), nr = 2)
+     a[1, 2] <- x[3]
+     V <- crossprod(a)
+     return(dlm(m0 = rep(0, 2), C0 = 1e+07 * diag(2), FF = diag(2),
+         GG = diag(2), V = V, W = x[4]^2 * V))
+ }
> hoMLE <- dlmMLE(NelPlo, rep(0, 4), buildHo)
> hoMLE$convergence

[1] 0


> buildHo(hoMLE$par)[c("V", "W")]

$V
      [,1]  [,2]
[1,] 2.394 0.667
[2,] 0.667 9.260

$W
     [,1] [,2]
[1,]    0    0
[2,]    0    0



> hoMLE$par[4]

[1] 0
```

A word of caution about MLE with DLM is in order. For general DLM containing unknown parameters, the likelihood function may be relatively flat, suggesting a scarcely identifiable parametrization, or/and it may have several local maxima. We suggest, especially for highly complex models, to repeat the optimization process several times, starting from different initial values.

4 Filtering

The function `dlmFilter` computes and returns the posterior mean and variance (m_t and C_t in [WH]) of θ_t given $\mathcal{Y}_t$ ($t = 0, \ldots, n$), and the mean and variance (a_t and R_t in [WH]) of θ_t given $\mathcal{Y}_{t-1}$ ($t = 1, \ldots, n$). The main arguments are `y` and `mod`. The first one is the matrix or time series of the observations, the second is a `dlm` object. The function returnes a list. The components `m` and `a` contain the time series or matrices/vectors corresponding to the m_t and a_t, respectively. The variances C_t and R_t are returned in terms of their SVD. If S is a symmetric nonnegative definite matrix, we write the singular value decomposition as $S = UD^2U'$, where U is an orthogonal matrix and D diagonal. The U matrices of the SVD of the C_t are returned in the list `U.C`, while the diagonal elements of the D matrices are the rows of the matrix `D.C`. Similarly for the SVD of the R_t, returned in list `U.R` and matrix `D.R`. If needed, the variance matrices can be reconstructed using the function `dlmSvd2var`. This is the output when the argument `simplify` is `TRUE`, and is typically used when the function is called repeatedly within a Gibbs sampler. The default value for `simplify` is `FALSE`, in which case the output list has the additional components `y` and `mod`, which are essentially copies of the corresponding input arguments. Moreover, the output has a `"dlmFiltered"` class attribute, which is used for calculating residuals (one-step-ahead forecast errors) and drawing diagnostic plots.

Example

Consider the data set used in the example for `dlmMLE` above and the seemingly unrelated time series model discussed in the same example. The variance matrices V and W are generally considered parameters that need to be estimated, but to illustrate the filtering procedure, we cansider them known, using the parameter estimates obtained with `dlmMLE`.

```
> mod <- buildSu(suMLE$par)
> modFilt <- dlmFilter(NelPlo, mod)
> names(modFilt)

[1] "y"   "mod" "m"   "U.C" "D.C" "a"   "U.R" "D.R" "f"
```

The `m` component of `modFilt` is a bivariate time series representing the filtered hypothetical "true levels" of the observed series. One can plot them together with the data.

```
> plot(NelPlo, plot.type = "single", lty = 2:3, type = "b")
> for (i in 1:2) lines(seq(start(NelPlo)[1] - 1, end(NelPlo)[1]),
+     modFilt$m[, i], col = "red", lty = i + 1, type = "b", pch = 5)
> leg <- as.vector(t(outer(dimnames(NelPlo)[[2]], c("observed",
+     "filtered"), paste)))
> legend(1965, 8, legend = leg, lty = rep(2:3, each = 2), col = c("black",
+     "red"), pch = c(1, 5), bty = "n")
```

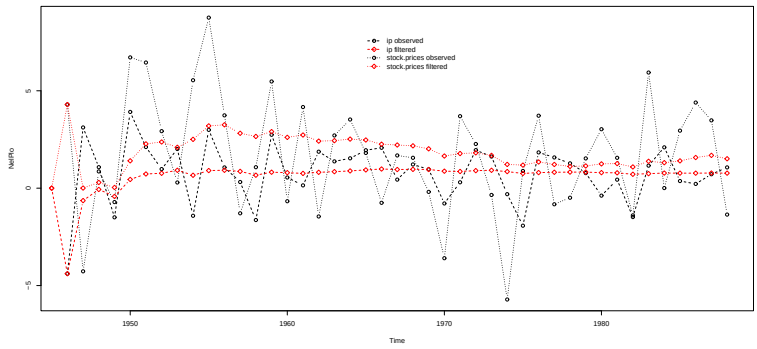

Figure 1: Observed and filtered levels for NelPlo data

If the model is correct for the given data, the standardized one-step forecast errors, for each series, should be independent identically distributed according to a standard normal distribution. Method `dlmFiltered` of function `residuals` can be used to compute the standardized one-step forecast errors as well as their standard deviation. Method `dlmFiltered` of function `tsdiag`, which is modeled after `tsdiag.Arima`, calls `residuals` internally and draws standard diagnostic plots for the standardized errors.

5 Smoothing

To find the conditional mean and variance of θ_t $(t = 0, \ldots, n)$ given $\mathcal{Y}_n$, one can use the function `dlmSmooth`. The returned value is a list: the component `s` is the vector or matrix of smoothed values, while `U.S` and `D.S` provide the SVD of the variance matrices. The variance matrices can be reconstructed from their SVD using the funcion `dlmSvd2var`. The code below shows how 90% probability intervals can be obtained for the (unobservable) "true" stock price level in the NelPlo data set.

```
> modSmooth <- dlmSmooth(modFilt)
> v <- dlmSvd2var(modSmooth$U.S, modSmooth$D.S)
> se <- sapply(seq(along = v), function(i) sqrt(v[[i]][2, 2]))
> x <- ts(se %o% (c(-1, 1, 0) * qnorm(0.05, lower = F)) + as.vector(modSmooth$s[,
+     2]), start = start(NelPlo)[1] - 1)
> plot(NelPlo[, "stock.prices"], type = "b", lty = 3, ylab = "Stock price")
> for (i in 1:2) lines(x[, i], col = "blue", lty = 2)
> lines(x[, 3], col = "red", type = "b", pch = 5, cex = 0.75)
> leg <- c("observed", "smoothed", "90% probability limit")
> legend(1963, 8, legend = leg, lty = c(3, 1, 2), col = c("black",
+     "red", "blue"), pch = c(1, 5, NA), bty = "n")
```

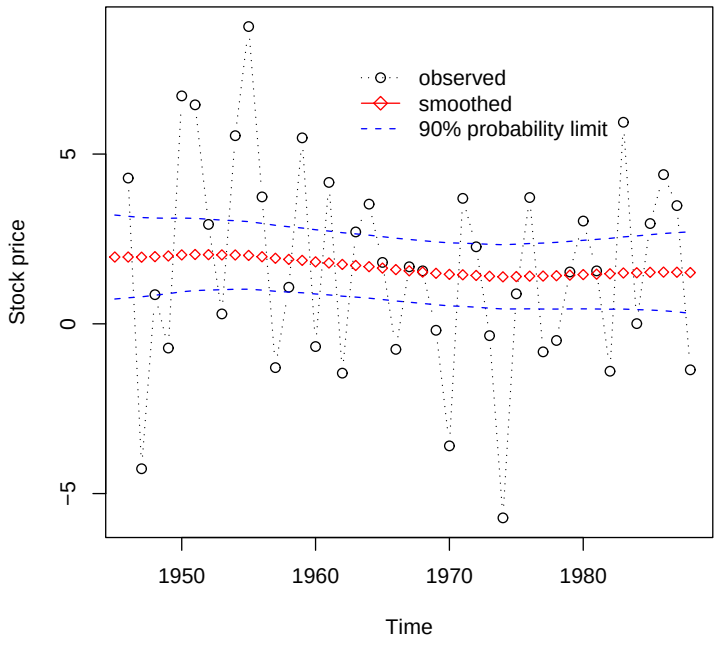

Figure 2: 90% probability interval for stock price level

6 Gibbs sampling: Forward filtering backward sampling

Taking a simulation-based Bayesian approach to the estimation of unknown parameters occurring in the specification of a DLM, it is usually convenient to simulate the states $\theta_0, \theta_1, \ldots, \theta_n$ from their full conditional distribution at each sweep of the sampler. The function `dlmBSample` generates a realization from the posterior distribution of the state vectors $\theta_0, \ldots, \theta_n$. The algorithm used is the so-called forward filtering backward sampling algorithm, described in [WH], with an implementation based on SVD. Strictly speaking, the function `dlmBSample` only performs the "backward sampling" part, starting from the "forward filtering" provided in the output of `dlmFilter`. The argument of `dlmBSample` is the output returned by `dlmFilter`. Since the data are not needed, to avoid copying possibly long vectors at each call of `dlmFilter`, one can specify the argument `simplify = TRUE`. The output of `dlmBSample` can be used within a Gibbs sampler, thinking of the θ_t as latent variables, to generate the state vectors for a given value of the parameters specifying the DLM.

Example

Consider once again Nelson-Plotter data and the seemingly unrelated time series model described above. Taking a Bayesian approach, we put independent inverse Wishart priors with parameter Λ_0^{-1} and ν_0 degrees of freedom on V and W. Those are the only unknown parameters of the model, and we generate them within a Gibbs sampler which include also the state vectors as latent variables. The likelihood for V is proportional to

$$|V|^{-\frac{n}{2}} \exp\{-\frac{1}{2}\operatorname{tr}(V^{-1}S)\},$$

where S is the multivariate sum of squares $\sum_{t=1}^{n}(y_t - F\theta_t)(y_t - F\theta_t)'$. Therefore the full conditional distribution is again inverse Wishart with parameter $\Lambda_0^{-1} + S$ and $\nu_0 + n$ degrees of freedom. Similarly, it can be seen that the full conditional distribution of W is inverse Wishart with parameter $\Lambda_0^{-1} + S_\theta$ and $\nu_0 + n$ degrees of freedom, with $S_\theta = \sum_{t=1}^{n}(\theta_t - G\theta_{t-1})(\theta_t - G\theta_{t-1})'$. Before running the actual sampler we specify the hyperparameters of the prior, create objects that will be used to store the variables generated by the sampler, and perform some initializations.

```
> MC <- 1000
> n <- NROW(NelPlo)
```

```
> mod <- buildSu(suMLE$par)
> gibbsTheta <- array(0, dim = c(MC, NROW(NelPlo) + 1, 2))
> gibbsV <- array(0, dim = c(MC + 1, 2, 2))
> gibbsW <- array(0, dim = c(MC + 1, 2, 2))
> nu <- 2
> L0inv <- var(NelPlo)/2
> gibbsV[1, , ] <- mod$V
> gibbsW[1, , ] <- mod$W
> set.seed(6324)
```

The code below gives the actual Gibbs sampler, generating in turn the state vectors θ_t, V, and W. Since the code is not optimized for speed and it may take several seconds to run, we set the number of MCMC samples to 1000.

```
> for (i in 1:MC) {
+     mod$V[] <- gibbsV[i, , ]
+     mod$W[] <- gibbsW[i, , ]
+     modFilt <- dlmFilter(NelPlo, mod, simplify = TRUE)
+     gibbsTheta[i, , ] <- dlmBSample(modFilt)
+     Linv <- crossprod(NelPlo - gibbsTheta[i, -1, ] %*% mod$FF) +
+         L0inv
+     tmp <- La.svd(Linv, nu = 0)
+     if (any(is.infinite(q <- 1/sqrt(tmp$d))))
+         stop("singular Linv in 'sample V'")
+     sqrtL <- q * tmp$vt
+     gibbsV[i + 1, , ] <- solve(rwishart(df = nu + n, SqrtSigma = sqrtL))
+     Linv <- crossprod(gibbsTheta[i, -1, ] - gibbsTheta[i, -(n +
+         1), ] %*% mod$GG) + L0inv
+     tmp <- La.svd(Linv, nu = 0)
+     if (any(is.infinite(q <- 1/sqrt(tmp$d))))
+         stop("singular Linv in 'sample W'")
+     sqrtL <- q * tmp$vt
+     gibbsW[i + 1, , ] <- solve(rwishart(df = nu + n, SqrtSigma = sqrtL))
+ }
```

It is straightforward to compute posterior means from the output and to plot the posterior mean of the true level of the two series. We remove the first 100 iterations as burn-in. A graphical summary of the Bayesian estimate of the "true" level of the two series is shown in Figure 3.

```
> apply(gibbsV[-(1:100), , ], c(2, 3), mean)

      [,1]  [,2]
[1,] 2.325 0.435
[2,] 0.435 8.009

> apply(gibbsW[-(1:100), , ], c(2, 3), mean)

       [,1]   [,2]
[1,] 0.4673 0.0248
[2,] 0.0248 1.8418

> meanGibbsTheta <- apply(gibbsTheta[-(1:100), , ], c(2, 3), mean)
> plot(NelPlo, plot.type = "single", lty = 2:3, col = "blue", type = "o")
> lines(seq(start(NelPlo)[1] - 1, end(NelPlo)[1]), meanGibbsTheta[,
+     1], lty = 2, col = "red", type = "o", pch = 5)
> lines(seq(start(NelPlo)[1] - 1, end(NelPlo)[1]), meanGibbsTheta[,
+     2], lty = 3, col = "red", type = "o", pch = 5)
> leg <- as.vector(t(outer(dimnames(NelPlo)[[2]], c("- observed",
+     "- Bayes estimate of level"), paste)))
> legend(1965, 8, legend = leg, lty = rep(2:3, each = 2), col = c("blue",
+     "red"), pch = c(1, 5), bty = "n")
```

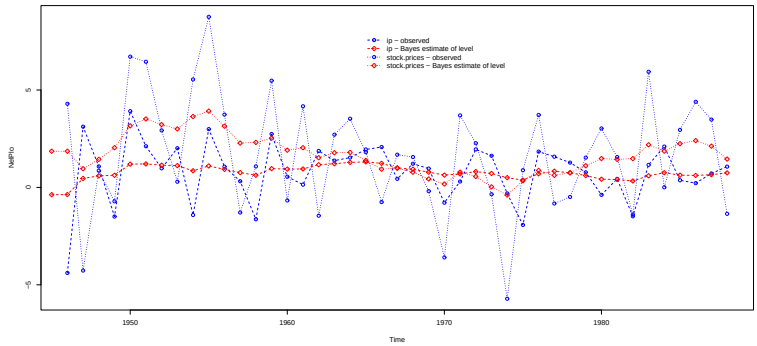

Figure 3: Bayesian estimate of level for NelPlo data

7 Adaptive rejection Metropolis Sampling

The package also contains the function `arms`, which allows to use adaptive rejection sampling (see [GBT]) to generate an observation from a given density. The function is based on a C function written by W. Gilks, that works for univariate distributions. Our R interface can be used also with multivarate densities, in which case the univariate C function is called on the given density along a randomly selected straight line through the current point. The help page contains several examples unrelated to DLM. In the DLM framework, `arms` can be used within a Gibbs sampler for parameters whose full conditional distribution is nonstandard.

Example

Consider the US macroeconomic data in the data set `USecon`, and consider again a seemingly unrelated time series model with homogeneity constraints like the one fitted above for the Nelson-Plosser data. Here $W = qV$ and the unknown parameters are V and q. We take an inverse Wishart distribution prior for V and a uniform prior on a reasonably large interval (ϵ, E) for q. In the code below we use `arms` to generate q.

```
> data(USecon)
> FF <- GG <- diag(2)
> MC <- 1000
> n <- NROW(USecon)
> gibbsTheta <- array(0, dim = c(MC, NROW(USecon) + 1, 2))
> gibbsV <- array(0, dim = c(MC + 1, 2, 2))
> gibbsQ <- rep(0, MC + 1)
> m0 <- rep(0, 2)
> C0 <- 1e+08 * diag(2)
> nu <- 2
> L0inv <- var(USecon)/2
> gibbsV[1, , ] <- V <- matrix(c(1.65, 0.1, 0.1, 0.7), 2, 2)
> gibbsQ[1] <- 0.25
> W <- gibbsQ[1] * V
> mod <- list(m0 = rep(0, 2), C0 = 1e+07 * diag(2), FF = FF, V = V,
+     GG = GG, W = W)
> ind <- function(x) x > .Machine$double.eps && x < 400
> ldens <- function(q) -0.5 * sum(diag(solve(V, SStheta)))/q -
+     n * log(q)
> for (i in 1:MC) {
```

```
+     mod$V[] <- V
+     mod$W[] <- W
+     modFilt <- dlmFilter(USecon, mod, simplify = TRUE)
+     gibbsTheta[i, , ] <- dlmBSample(modFilt)
+     SStheta <- crossprod(gibbsTheta[i, -1, ] - gibbsTheta[i,
+         -(n + 1), ] %*% GG)
+     Linv <- crossprod(USecon - gibbsTheta[i, -1, ] %*% FF) +
+         SStheta/gibbsQ[i] + L0inv
+     tmp <- La.svd(Linv, nu = 0)
+     sqrtL <- (1/sqrt(tmp$d)) * t(tmp$v)
+     gibbsV[i + 1, , ] <- V <- solve(rwishart(df = nu + 2 * n,
+         SqrtSigma = sqrtL))
+     gibbsQ[i + 1] <- arms(gibbsQ[i], ldens, ind, 5)[5]
+     W <- V * gibbsQ[i + 1]
+ }
```

The argument `5` in the call to `arms` asks the function to generate five draws from the target distribution. Of the five, we use the last one; this is just a way to speed up the sampler. Posterior summaries can be computed and plotted (Figure 4) in the usual way.

```
> apply(gibbsV[-(1:100), , ], c(2, 3), mean)

       [,1]   [,2]
[1,] 1.4214 0.0936
[2,] 0.0936 0.6085

> mean(gibbsQ[-(1:100)])

[1] 0.707

> meanGibbsTheta <- apply(gibbsTheta[-(1:100), , ], c(2, 3), mean)
> plot(USecon, plot.type = "single", lty = 2:3, col = "blue", type = "o")
> lines(as.numeric(time(USecon)), meanGibbsTheta[-1, 1], lty = 2,
+     col = "red", type = "o", pch = 5)
> lines(as.numeric(time(USecon)), meanGibbsTheta[-1, 2], lty = 3,
+     col = "red", type = "o", pch = 5)
> leg <- as.vector(t(outer(dimnames(USecon)[[2]], c("- observed",
+     "- Bayes estimate of level"), paste)))
> legend(1984, -2, legend = leg, lty = rep(2:3, each = 2), col = c("blue",
+     "red"), pch = c(1, 5), bty = "n")
```

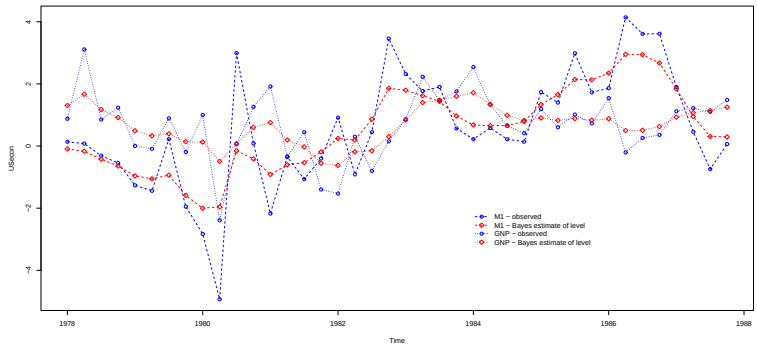

Figure 4: US macroeconomic data: Bayesian analysis

8 Downloading

The current version of the package can be downloaded from the following URL.

http://definetti.uark.edu/~gpetris/DLM

References

[GBT] Gilks, W.R., Best, N.G. and Tan, K.K.C. (1995). Adaptive rejection Metropolis sampling within Gibbs sampling. *Applied Statistics* **44**, 455–472.

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

[WH] West, M. and Harrison, J. (1997). *Bayesian forecasting and dynamic models*. (Second edition. First edition: 1989), Springer, N.Y.

[ZL] Zhang, Y. and Li, R. (1996). Fixed-interval smoothing algorithm based on singular value decomposition. *Proceedings of the 1996 IEEE International Conference on Control Applications*, Dearborn, 916–921.