A simple and efficient simulation smoother for state space time series analysis.

RK

April 26, 2014

Abstract

This document contains my notes relating to the paper, "A simple and efficient simulation smoother for state space time series analysis", by Durbin and Koopman(2002).

Contents

1 Introduction

Firstly, the basic state space model representation

$$y_t = Z_t\alpha_t + \epsilon_t, \quad \epsilon_t \sim N(0, H_t)$$
$$\alpha_{t+1} = T_t\alpha_t + R_t\eta_t, \quad \eta_t \sim N(0, Q_t), t = 1, 2 \ldots, n$$

where y_t is vector of observations and α_t is the state vector.

For any model that you build, it is imperative that you are able to sample random realizations of the various parameters of the model. In a structural model, be it a linear Gaussian or nonlinear state space model, an additional requirement is that you have to sample the state and disturbance vector given the observations. To state it precisely, the problem we are dealing is to draw samples from the conditional distributions of $\epsilon = (\epsilon_1', \epsilon_2', \ldots, \epsilon_n')'$, $\eta = (\eta_1', \eta_2', \ldots, \eta_n')'$ and $\alpha = (\alpha_1', \alpha_2', \ldots, \alpha_n')'$ given $y = (y_1', y_2', \ldots, y_n')'$.

The past literature on this problem is :

- Fruhwirth-Schnatter(1994), Data augmentation and dynamic linear models. : The method comprised drawing samples of $\alpha|y$ recursively by first sampling $\alpha_n|y$, then sampling $\alpha_{n-1}|\alpha_n, y$, then $\alpha_{n-2}|\alpha_{n-1}, \alpha_n, y$.
- de Jong and Shephard(1995),The simulation smoother for time series models. : This paper was a significant advance as the authors first considered sampling the disturbances and subsequently sampling the states. This is more efficient than sampling the states directly when the dimension of η is smaller than the dimension of α

In this paper, the authors present a new simulation smoother which is simple and is computationally efficient relative to that of de Jong and Shephard(1995).

R excursion

Before proceeding with the actual contents of the paper, it is better to get a sense of the difference between conditional and unconditional sampling. Durbin and Koopman in their book on State space models illustrate this difference via Nile Data. Here is some `R` code that can be used for a visual that highlights the difference. The dataset used is "Nile" dataset. The State space model used is *Local level model*

$$y_t = \alpha_t + \epsilon_t, \quad \epsilon_t \sim N(0, \sigma_\epsilon^2)$$
$$\alpha_{t+1} = \alpha_t + \eta_t, \quad \eta_t \sim N(0, \sigma_\eta^2)$$

```r
library(datasets)
library(dlm)
library(KFAS)
data(Nile)
# MLE for estimating observational and state evolution variance.
fn              <- function(params){
```

```
                dlmModPoly(order= 1, dV= exp(params[1]) , dW = exp(params[2]))
                }
y                <- c(Nile)
fit              <- dlmMLE(y, rep(0,2),fn)
mod              <- fn(fit$par)
(obs.error.var   <- V(mod))

##       [,1]
## [1,] 15100

(state.error.var <- W(mod))

##      [,1]
## [1,] 1468

filtered         <- dlmFilter(y,mod)
smoothed         <- dlmSmooth(filtered)
mu.tilde         <- dropFirst(smoothed$s)
```

Now that we have estimated the variance, one can simulate the uncondition state vector

```
set.seed(1)
n                    <- length(y)
theta.uncond         <- numeric(n)
theta.uncond[1]      <- y[1]
for(i in 2:n){
      theta.uncond[i] <- theta.uncond[i-1] + rnorm(1,0,sqrt(state.error.var))
}
```

Now let's simulate conditional state vector

```
theta.cond <- dlmBSample(filtered)
```

One can compare the smoothed estimate, the simulated conditional state vector and unconditional state vector

```
ylim = c(400,max(c(mu.tilde,theta.cond,theta.uncond))+50)
plot(1:100, y, type="l", col = "grey", ylim = ylim)
points(1:100, mu.tilde , type="l", col = "green", lwd = 2)
points(1:100, theta.cond[-1], type="l", col = "blue", lwd = 2)
points(1:100, theta.uncond, type="l", col = "red", lwd = 2)
leg <-c("actual data","smoothed estimate","conditional sample","unconditional sample")
legend("topleft",leg,col=c("grey","green","blue","red"),lwd=c(1,2,2,2),cex=0.7,bty="n")
```

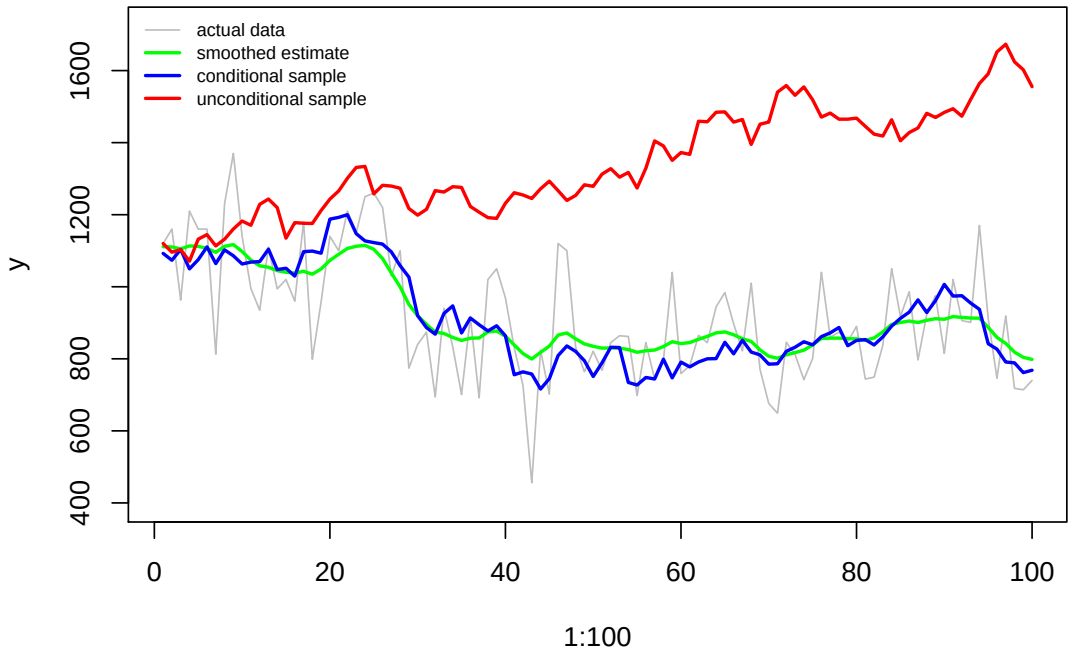

Figure 1.1: Comparison of Conditional and Unconditional State vector simulated sample

As one can see from figure 1.1 the unconditional simulated state vector is useless. That's the reason one needs a method to simulate state vector given the observations.

The dataset used in the paper is the one that is used in the book, "An Introduction to State Space Time Series Analysis" by Jacques J.F.Commandeur and Siem Jan Koopman. This dataset is present in the package `datasets` as `UKDriverDeaths`

2 The New Simulation Smoother

This section gives the algorithm needed to generate a simulation smoother for the disturbances η and ϵ.

2.1 Simulation of Observation and State disturbances

Algorithm 1

1. Draw a random vector w^+ from density $p(w)$ and use it to generate y^+ by means of the recursive observation and system equation with the error terms related by w^+, where the recursion is initialized by the draw $\alpha_1^+ \sim N(a_1, P_1)$

$$p(w) = N(0, \Omega), \quad \Omega = \text{diag}(H_1, \ldots, H_n, Q_1, \ldots, Q_n)$$

2. Compute $\hat{w} = (\hat{\epsilon}', \hat{\eta}')' = E(w|y)$ and $\hat{w}^+ = (\hat{\epsilon^+}', \hat{\eta^+}')' = E(w^+|y^+)$ by means of standard Kalman filtering and disturbance smoothing using the following equations

$$\begin{aligned} \hat{\epsilon}_t &= H_t F_t^{-1} \nu_t - H_t K_t' r_t \\ \hat{\eta}_t &= Q_t R_t' r_t \\ r_{t-1} &= Z_t F_t^{-1} \nu_t + L_t' r_t \end{aligned}$$

3. Take $\tilde{w} = \hat{w} - \hat{w}^+ + w^+$

The next section of the paper presents the simulation based on innovation DLM and says that computational gain is small compared to algorithm stated above.

2.2 Simulation of State vector

This section gives the following algorithm for simulating the state vector.

Algorithm 2

1. Draw a random vector w^+ from density $p(w)$ and use it to generate α^+, y^+ by means of the recursive observation and system equation with the error terms related by w^+, where the recursion is initialized by the draw $\alpha_1^+ \sim N(a_1, P_1)$

$$p(w) = N(0, \Omega), \quad \Omega = \text{diag}(H_1, \ldots, H_n, Q_1, \ldots, Q_n)$$

2. Compute $\hat{\alpha} == E(\alpha|y)$ and $\hat{\alpha}^+ = E(\alpha^+|y^+)$ by means of standard Kalman filtering and smoothing equations
3. Take $\tilde{\alpha} = \hat{\alpha} - \hat{\alpha}^+ + \alpha^+$

This section also gives the algorithm for diffuse initial conditions. Also there is a mention of the use of Antithetic sampling as an additional step for uncorrelated samples.

R excursion

Now let's use the above algorithms to generate samples from the state vector and disturbance vector. Let me use the same "Nile" dataset.

Algorithm 1 - Implementation

```
# MLE for estimating observational and state evolution variance.
fn              <- function(params){
                dlmModPoly(order= 1, dV= exp(params[1]) , dW = exp(params[2]))
                }
y                <- c(Nile)
fit              <- dlmMLE(y, rep(0,2),fn)
mod              <- fn(fit$par)
(obs.error.var   <- V(mod))

##       [,1]
## [1,] 15100

(state.error.var <- W(mod))

##      [,1]
## [1,] 1468

filtered         <- dlmFilter(y,mod)
smoothed         <- dlmSmooth(filtered)
smoothed.state   <- dropFirst(smoothed$s)
w.hat            <- c(Nile- smoothed.state,c(diff(smoothed.state),0))
```

Step 1 :

```
set.seed(1)
n             <- length(Nile)

# Step 1
w.plus        <- c( rnorm(n,0,sqrt(obs.error.var)),rnorm(n,0,sqrt(state.error.var)))
alpha0        <- rnorm(1, mod$m0, sqrt(mod$C0))

y.temp        <- numeric(n)
alpha.temp    <- numeric(n)

#Step 2
for(i in 1:n){
  if(i==1){
    alpha.temp[i] <- alpha0 + w.plus[100+i]
  }else{
    alpha.temp[i] <- alpha.temp[i-1] + w.plus[100+i]
```

```
  }
  y.temp[i]       <- alpha.temp[i] + w.plus[i]
}
temp.smoothed     <- dlmSmooth(y,mod)
alpha.smoothed    <- dropFirst(temp.smoothed$s)
```

Step 2 :

```
w.hat.plus        <- c(Nile- alpha.smoothed,c(diff(alpha.smoothed),0))
```

Step 3 :

```
w.tilde        <- w.hat - w.hat.plus + w.plus
```

This generates one sample of the observation and state disturbance vector.
Let's simulate a few samples and overlay them on the actual observation and state disturbance vectors

```
set.seed(1)
n             <- length(Nile)
disturb.samp  <- matrix(data= 0,nrow = n*2, ncol = 25)

for(b in 1:25){
  # Step 1
  w.plus        <- c( rnorm(n,0,sqrt(obs.error.var)),rnorm(n,0,sqrt(state.error.var)))
  alpha0        <- rnorm(1, mod$m0, sqrt(mod$C0))

  y.temp        <- numeric(n)
  alpha.temp    <- numeric(n)

  #Step 2
  for(i in 1:n){
    if(i==1){
      alpha.temp[i] <- alpha0 + w.plus[100+i]
    }else{
      alpha.temp[i] <- alpha.temp[i-1] + w.plus[100+i]
    }
    y.temp[i]       <- alpha.temp[i] + w.plus[i]
  }
  temp.smoothed     <- dlmSmooth(y.temp,mod)
  alpha.smoothed    <- dropFirst(temp.smoothed$s)
  w.hat.plus        <- c(y.temp- alpha.smoothed,c(diff(alpha.smoothed),0))
  #Step 3
  w.tilde           <- w.hat - w.hat.plus + w.plus
  disturb.samp[,b]  <- w.tilde
}
```

```
temp  <- cbind(disturb.samp[101:200,],w.hat[101:200])
plot.ts(temp, plot.type="single",col =c(rep("grey",25),"blue"),
        ylim = c(-150,200),lwd=c(rep(0.5,25),3), ylab = "",xlab="")
leg <-c("realized state error","simulated state error")
legend("topright",leg,col=c("blue","grey"),lwd=c(3,1),cex=0.7,bty="n")
```

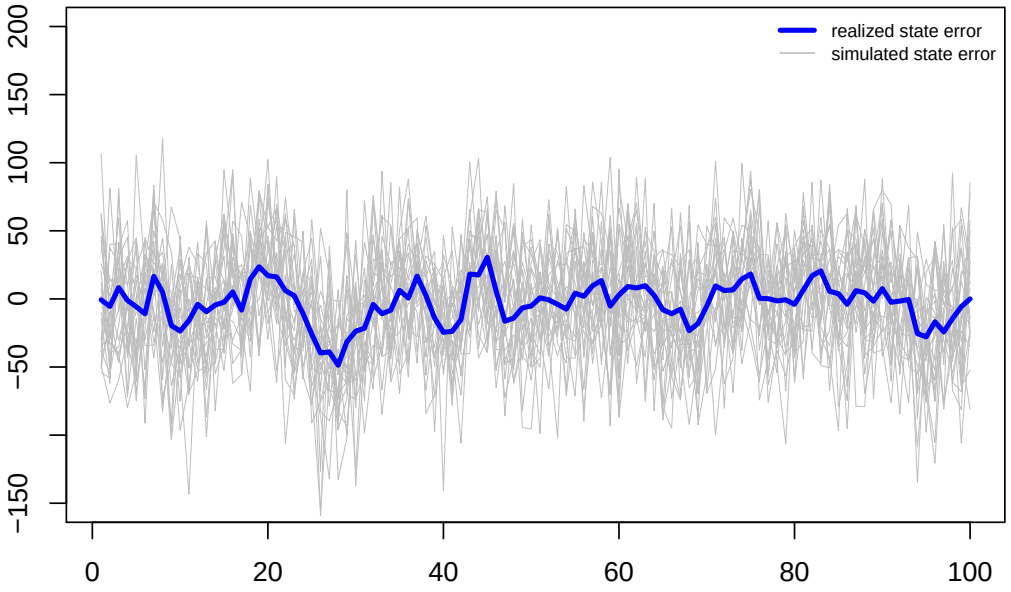

Figure 2.1: Comparison of Simulated and Realized State disturbance

```
temp  <- cbind(disturb.samp[1:100,],w.hat[1:100])
plot.ts(temp, plot.type="single",col =c(rep("grey",25),"blue"),
        ,lwd=c(rep(0.5,25),3), ylab = "",xlab="")
leg <-c("realized observation error","simulated observation error")
legend("topright",leg,col=c("blue","grey"),lwd=c(3,1),cex=0.7,bty="n")
```

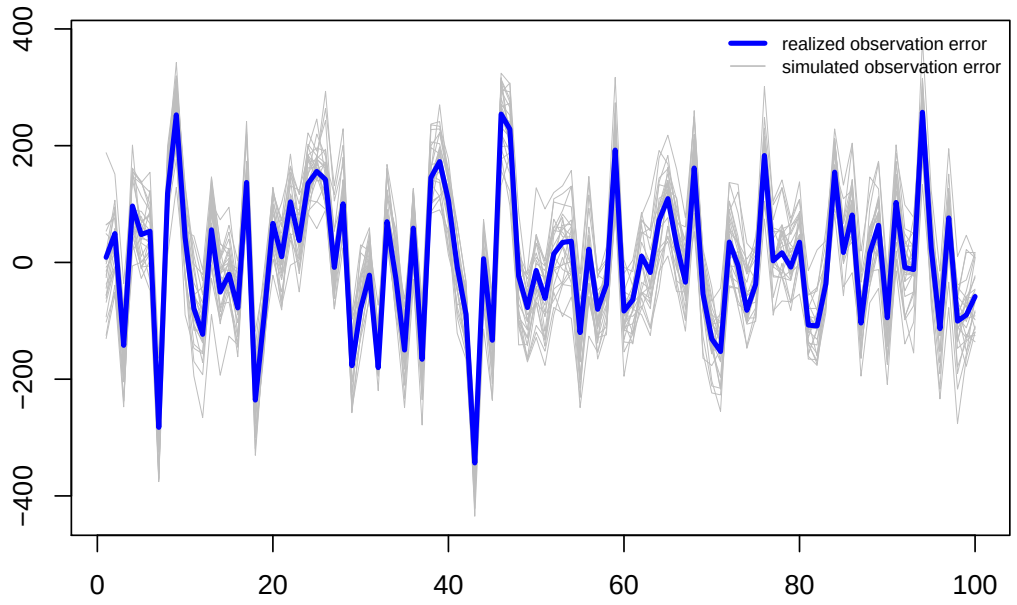

Figure 2.2: Comparison of Simulated and Realized Observation disturbance

Algorithm 2 - Implementation

```
set.seed(1)
n             <- length(Nile)
alpha.samp    <- matrix(data= 0,nrow = n, ncol = 25)
for(b in 1:25){
  # Step 1
  w.plus        <- c( rnorm(n,0,sqrt(obs.error.var)),rnorm(n,0,sqrt(state.error.var)))
  alpha0        <- rnorm(1, mod$m0, sqrt(mod$C0))

  y.temp        <- numeric(n)
  alpha.temp    <- numeric(n)

  #Step 2
  for(i in 1:n){
    if(i==1){
      alpha.temp[i] <- alpha0 + w.plus[100+i]
    }else{
      alpha.temp[i] <- alpha.temp[i-1] + w.plus[100+i]
    }
    y.temp[i]       <- alpha.temp[i] + w.plus[i]
  }
  temp.smoothed     <- dlmSmooth(y.temp,mod)
  alpha.smoothed    <- dropFirst(temp.smoothed$s)
  alpha.hat.plus    <- alpha.smoothed

  #Step 3
  alpha.tilde      <- smoothed.state - alpha.hat.plus + alpha.temp
  alpha.samp[,b]   <- alpha.tilde
}
```

```
temp       <- cbind(alpha.samp,smoothed.state)
plot.ts(temp, plot.type="single",col =c(rep("grey",25),"blue"),
        ,lwd=c(rep(0.5,25),3), ylab = "",xlab="")
leg <-c("realized state vector","simulated state vector")
legend("topright",leg,col=c("blue","grey"),lwd=c(3,1),cex=0.7,bty="n")
```

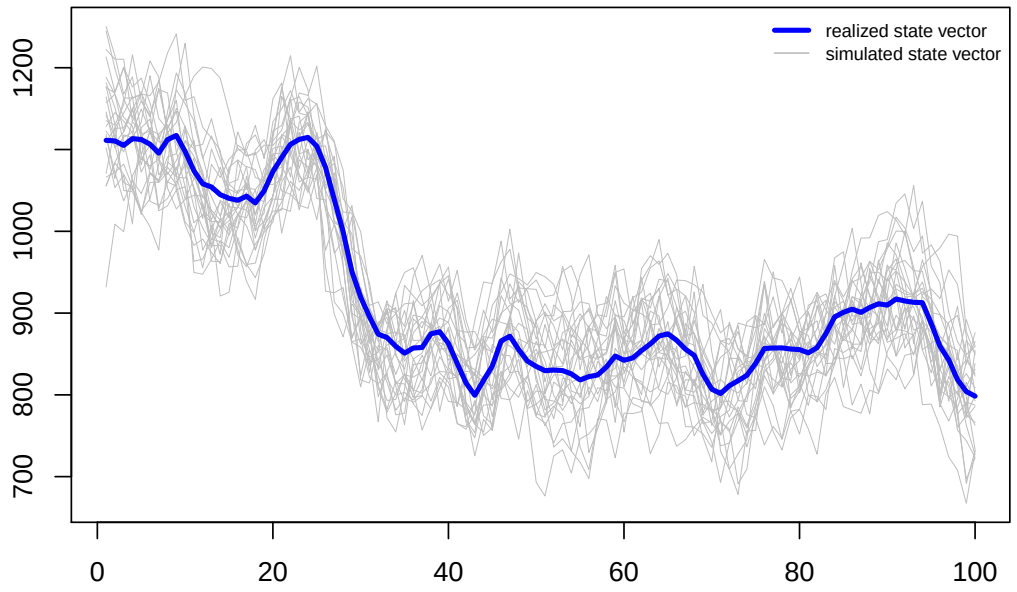

Figure 2.3: Comparison of Simulated and Realized State vector

2.3 Simulation via KFAS package

KFAS package closely follows the implementations suggested by the authors. One can simulate the state observations using `simulateSSM()` function. One can also turn on the antithetic sampling suggested in this paper via the function's argument.

```
modelNile   <-SSModel(Nile~SSMtrend(1,Q=list(matrix(NA))),H=matrix(NA))
modelNile   <-fitSSM(inits=c(log(var(Nile)),log(var(Nile))),
                     model=modelNile,
                      method='BFGS',
                    control=list(REPORT=1,trace=0))$model
out          <- KFS(modelNile,filtering='state',smoothing='state')
state.sample <- simulateSSM(modelNile, type = c("states"))
```

```
temp       <- cbind(state.sample,smoothed.state)
plot.ts(temp, plot.type="single",col =c(rep("grey",1),"blue"),
        ,lwd=c(rep(1,1),2), ylab = "",xlab="")
leg <-c("realized observation error","simulated observation error")
legend("topright",leg,col=c("blue","grey"),lwd=c(3,1),cex=0.7,bty="n")
```

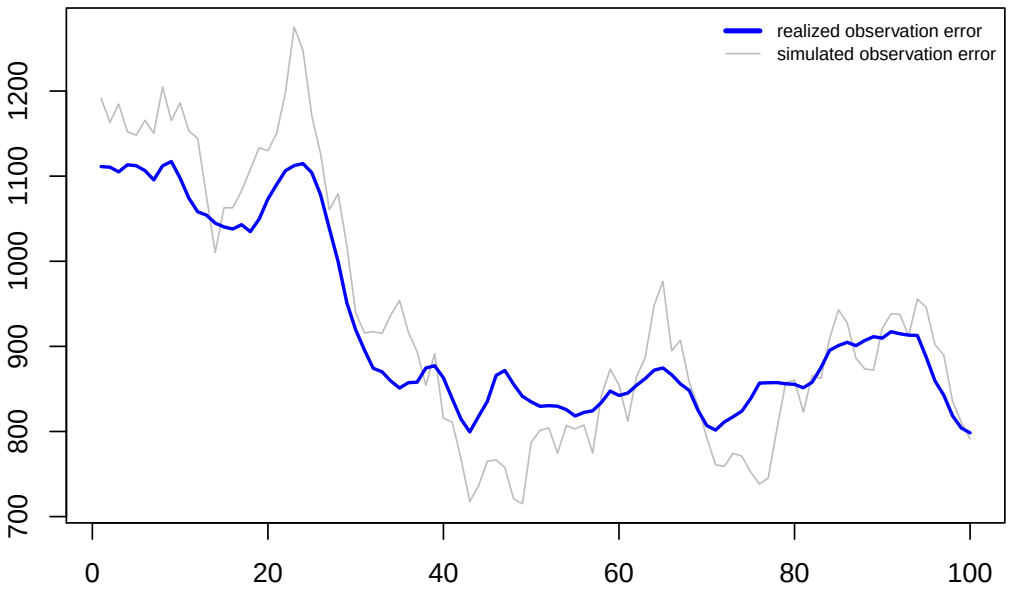

Figure 2.4: Comparison of Simulated and Realized State vector

3 Bayesian analysis based on Gibbs Sampling

This section illustrates Bayesian analysis of a structural model where the simulation smoother algorithm is used to generate posterior distribution of the metaparameters of the model. Consider first, a

Local level model

$$y_t = \mu_t + \epsilon_t, \quad \epsilon_t \sim N(0, \sigma_\epsilon^2)$$
$$\mu_{t+1} = \mu_t + \eta_t, \quad \eta_t \sim N(0, \sigma_\eta^2)$$

In the above model, the parameters are $\psi = (\sigma_\epsilon^2, \sigma_\eta^2)$. The estimation of these parameters is done in two steps. First place an Inverse gamma prior on ψ. Repeat the following M^* times.

1. sample μ^i from $p(\mu|y, \psi^{(i-1)})$ using Algorithm 1 given in the paper to obtain $\epsilon^{(i)}$ and $mu^{(i)}$
2. sample $\psi^{(i)}$ from $p(\psi|y, \mu^{(i)})$ using the inverse gamma density

The above steps run a MCMC chain and one can infer ψ from the chain values.OK, now let's run the above framework on `UKDriverDeaths`.

3.1 Parameter estimation via New Simulation Smoother

The following code use `KFAS` package to run gibbs sampling.

```
data(UKDriverDeaths)
y       <- log(c(UKDriverDeaths))
set.seed(1)
t1      <- Sys.time()
a1      <- 2
b1      <- 0.0001
a2      <- 2
b2      <- 0.0001
psi1    <- 1
psi2    <- 1

model   <- SSModel(y~SSMtrend(1,Q=1/psi1),H=1/psi2)
mc      <- 24000
psi1.r  <- numeric(mc)
psi2.r  <- numeric(mc)
n       <- length(UKDriverDeaths)
sh1     <- a1 + n/2
sh2     <- a2 + n/2

for(it in 1:mc){
  level        <- simulateSSM(model, type = "states")
```

```
  rate          <- b1 + crossprod(y - level)/2
  psi1          <- rgamma(1, shape = sh1, rate= rate)

  rate          <- b2 + crossprod(diff(level))/2
  psi2          <- rgamma(1, shape = sh2, rate= rate)
  model         <- SSModel(y~SSMtrend(1,Q=1/psi2),H=1/psi1)

  psi1.r[it]    <- psi1
  psi2.r[it]    <- psi2
}
t2              <- Sys.time()
delta.algo1     <- t2 - t1
```

```
burn    <- 8000
idx     <- seq(burn,mc,1)
plot(1/psi1.r[idx],type="l",xlab="",ylab="",
     main=expression(sigma[epsilon]^2), col = "blue")
```

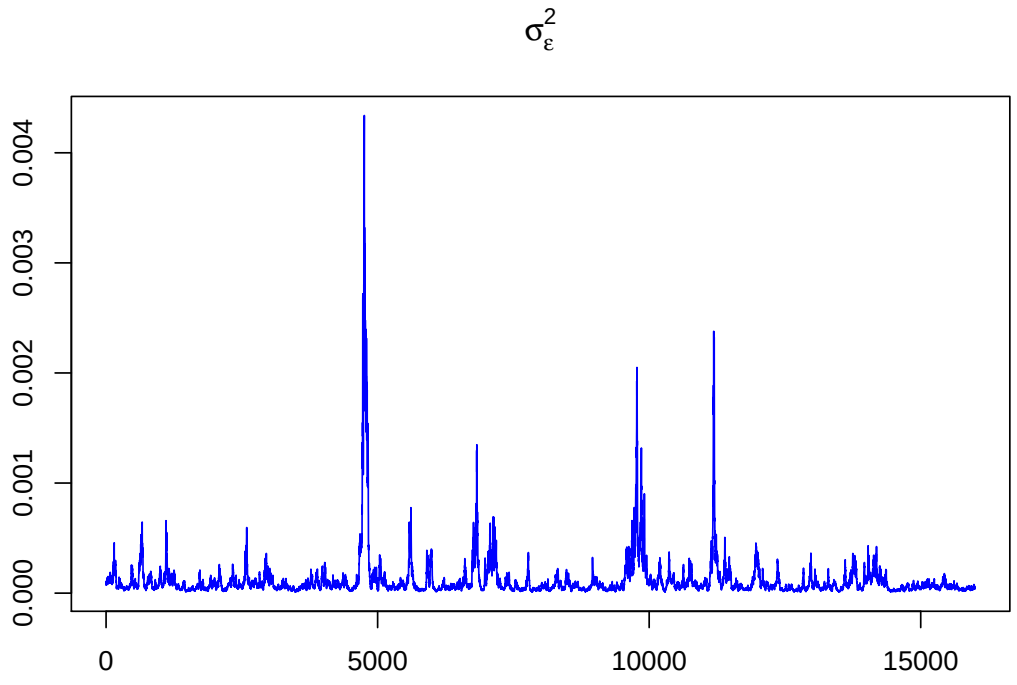

Figure 3.1: MCMC chain for observational error variance

```
plot(1/psi2.r[idx],type="l",xlab="",ylab="",
     main=expression(sigma[eta]^2), col = "blue")
```

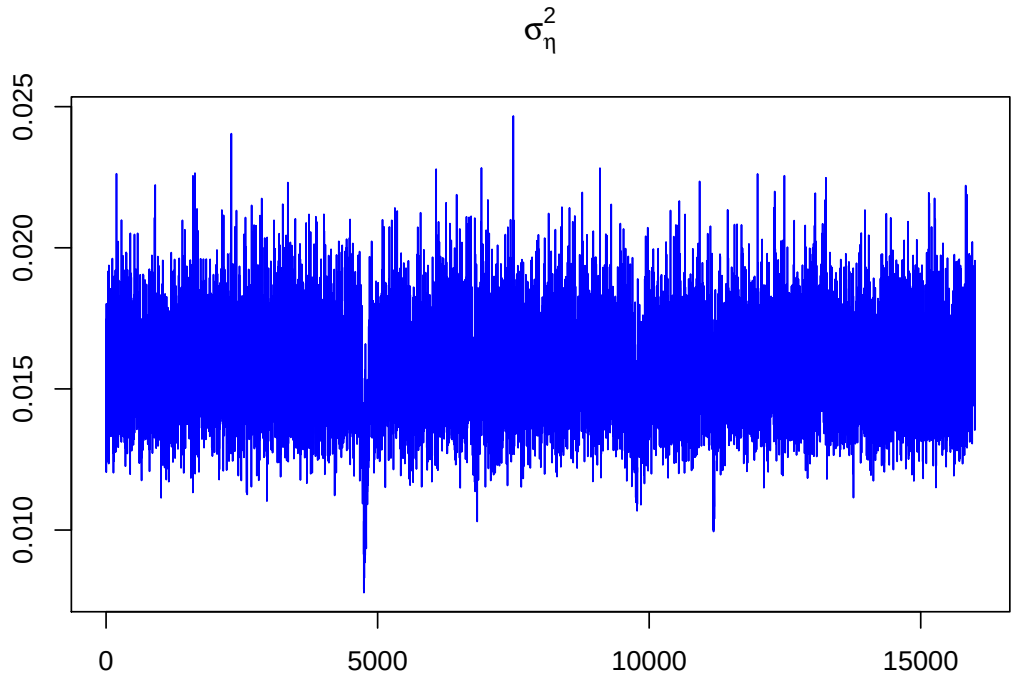

Figure 3.2: MCMC chain for state error variance

```r
burn    <- 15000
idx     <- seq(burn,mc,1)

#mean of Posterial variances
mus     <- apply(cbind(1/psi2.r[idx],1/psi1.r[idx]),2,mean)
#sd of Posterial variances
sds     <- apply(cbind(1/psi2.r[idx],1/psi1.r[idx]),2,sd)

res           <- cbind(mus,sds)
rownames(res) <- c("state.error","obs.error")
colnames(res) <- c("mu","sd")
res
```

```
##                    mu        sd
## state.error 0.0157734 0.0016529
## obs.error   0.0001083 0.0001587
```

3.2 Parameter estimation via de Jong and Shephard Algorithm

The following code use `dlm` package to run gibbs sampling.

```
t1              <- Sys.time()
out.dlm         <- dlmGibbsDIG(y,mod=dlmModPoly(1),
                   a.y=1,b.y=1000,a.theta=10,b.theta=1000,
                   n.sample=12000,thin-1,save.states=FALSE)
t2              <- Sys.time()
delta.dejong    <- t2 - t1
```

```
burn  <- 2000
use   <-  12000-burn
from  <- 0.05*use
plot(ergMean(out.dlm$dV[-(1:burn)],from),type="l",xaxt="n",
     xlab="",ylab="",main=expression(sigma[epsilon]^2), col = "blue")
at <- pretty(c(0,use),n=3)
at <- at[at>=from]
axis(1,at = at-from, labels = format(at))
```

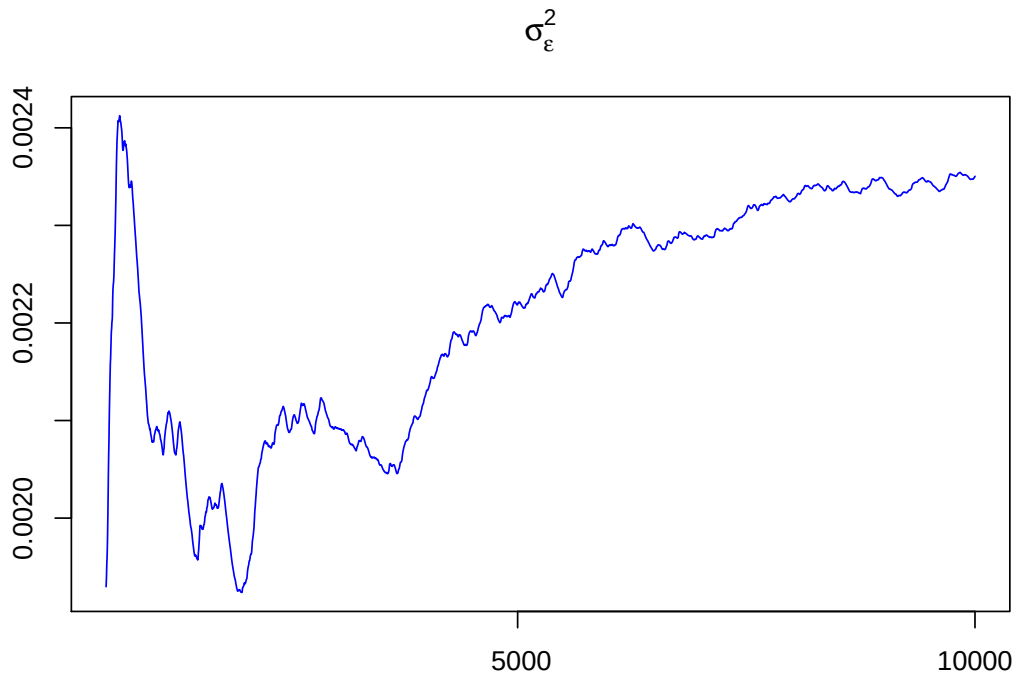

Figure 3.3: MCMC chain for observational error variance

```
plot(ergMean(out.dlm$dW[-(1:burn)],from),type="l",xaxt="n",
     xlab="",ylab="",main=expression(sigma[eta]^2), col = "blue")
at <- pretty(c(0,use),n=3)
at <- at[at>=from]
axis(1,at = at-from, labels = format(at))
```

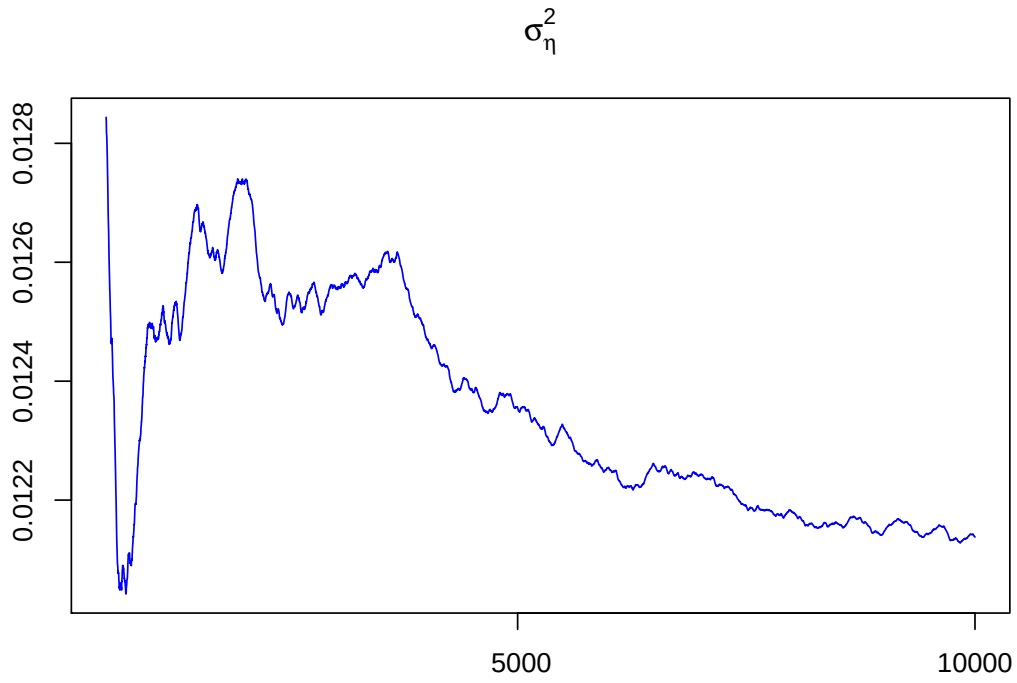

Figure 3.4: MCMC chain for state error variance

```
mean(out.dlm$dV[-(1:burn)]);sd(out.dlm$dV[-(1:burn)])

## [1] 0.001113

mean(out.dlm$dW[-(1:burn)]);sd(out.dlm$dW[-(1:burn)])

## [1] 0.002195
```

3.3 Comparing estimates with MLE

```
y       <- log(c(UKDriverDeaths))
fn <- function(params){
      dlmModPoly(order= 1, dV= exp(params[1]) , dW = exp(params[2]))
}

fit               <- dlmMLE(y, rep(0,2),fn)
mod               <- fn(fit$par)
obs.error.var     <- V(mod)
state.error.var   <- W(mod)
data.frame(sigma2eta = state.error.var,sigma2epsilon= obs.error.var )

##   sigma2eta sigma2epsilon
## 1   0.01187      0.002222
```

Gibbs sampling via Algorithm 1 took 2.329 minutes and Gibbs sampling de Jong and Shephard Algorithm took 5.7562 minutes. Clearly the algorithm mentioned in the paper is much faster.

The section ends with an example where the observation equation follows a general class of exponential family densities.

4 Conclusion

The paper presents a simulation smoother for drawing samples from the conditional distribution of the disturbances given the observations. Subsequently the paper highlights the advantages of this simulation technique over the previous methods.

- derivation is simple
- the method requires only the generation of simulated observations from the model together with the Kalman Filter and standard smoothing algorithms
- no inversion of matrices are needed beyond those in the standard KF
- diffuse initialization of state vector is handled easily
- this approach solves problems arising from the singularity of the conditional variance matrix W

5 Takeaway

This paper gives the details of a useful algorithm that speeds up simulating state vectors from a state space model. The algorithm runs very quick as compared to other methods. I ran the algorithm for a simple local level model and found the speed to be considerably faster than other Forward Filter Backward Sampling algorithms. For a more generic Bayesian inference, using this algorithm will no doubt cut the computation time significantly.