Tutorial: State-Space Modelling

by Kevin Kotzé

To start the tutorial, download and extract the `T5-statespace.zip` folder and open the `T5-statespace.proj` file.

1 The local level model

The first program for this session makes use of a local level model that is applied to the measure of the South African GDP deflator. The model is contained in the file `T5-llm.R`. Once again, the first thing that we do is clear all variables from the current environment and close all the plots. This is performed with the following commands:

```
rm(list=ls())
graphics.off()
```

Thereafter, you may need to install the `dlm` package that we will use in this session.

```
install.packages('dlm', repos='https://cran.rstudio.com/', dependencies=TRUE)
```

The next step is to make sure that you can access the routines in this package by making use of the `library` command, which would need to be run regardless of the machine that you are using.

```
library(tsm)
library(dlm)
library(tidyverse)
library(lubridate)
library(sarb2020q1)
```

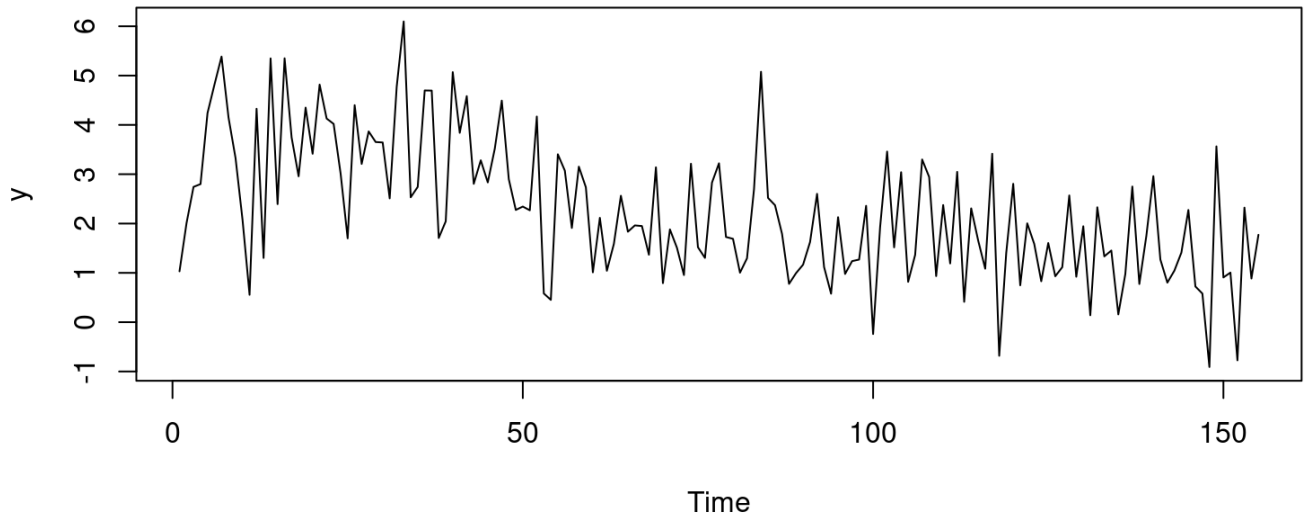
1.1 Transforming the data

As noted previously, South African GDP data has is contained in the `sarb2020q1` package. Real GDP is numbered KBP6006D and nominal GDP is KBP6006L. Hence, to retrieve the data from the package and create the object for the GDP deflator that will be stored in `dat`, we execute the following commands:

```
dat <- sarb_quarter %>%
  select(date, KBP6006L, KBP6006D) %>%
  mutate(defl = c(0, diff(log(KBP6006L / KBP6006D) * 100, lag = 1))) %>%
  filter(date > '1981-01-01')
```

Note that the object `dat$defl` is the quarter-on-quarter measure for the GDP deflation. Since we have already observed that there is a potential structural break in South African GDP in 1980, we make use of data from 1981 onwards. To make sure that these calculations and extractions have been performed correctly, we inspect a plot of the data.

```
y <- dat$defl
plot.ts(y)
```



1.2 Construct the model and estimate parameters

As already noted, the first model that we are going to build is a local level model, which includes a stochastic slope. The `dlm` package has a relatively convenient way to construct these models according to the order of the polynomial, where a first order polynominal is a local level model. We also need to allow for one parameter for the error in the mesurement equation and a second parameter for the error in the solitary state equation. These parameters are labelled `parm[1]` and `parm[2]`, respectively.

```
fn <- function(parm) {
  dlmModPoly(order = 1,
             dV = exp(parm[1]),
             dW = exp(parm[2]))
}
```

Thereafter, we are going to assume that the starting values for the parameters are small (i.e. zero) and proceed to fit the model to the data with the aid of maximum-likelihood methods.

```
fit <- dlmMLE(y, rep(0, 2), build = fn, hessian = TRUE)
(conv <- fit$convergence)  # zero for converged
```

```
## [1] 0
```

Note that by placing brackets around the command it will print the result in the console, where we note that convergence has been achieved. We can then construct the likelihood function and generate statistics for the information criteria.

```
loglik <- dlmLL(y, dlmModPoly(1))
n.coef <- 2
r.aic <- (2 * (loglik)) + 2 * (sum(n.coef)) #dlmLL caculates the neg. LL
r.bic <- (2 * (loglik)) + (log(length(y))) * (n.coef)
```

The statistics for the variance-covariance matrix could also be extracted from the model object with the use of the commands:

```
mod <- fn(fit$par)
obs.error.var <- V(mod)
state.error.var <- W(mod)
```

1.3 Kalman filter and smoother

To extract information relating to the Kalman filter and smoother, we use the instructions:

```
filtered <- dlmFilter(y, mod = mod)
smoothed <- dlmSmooth(filtered)
```

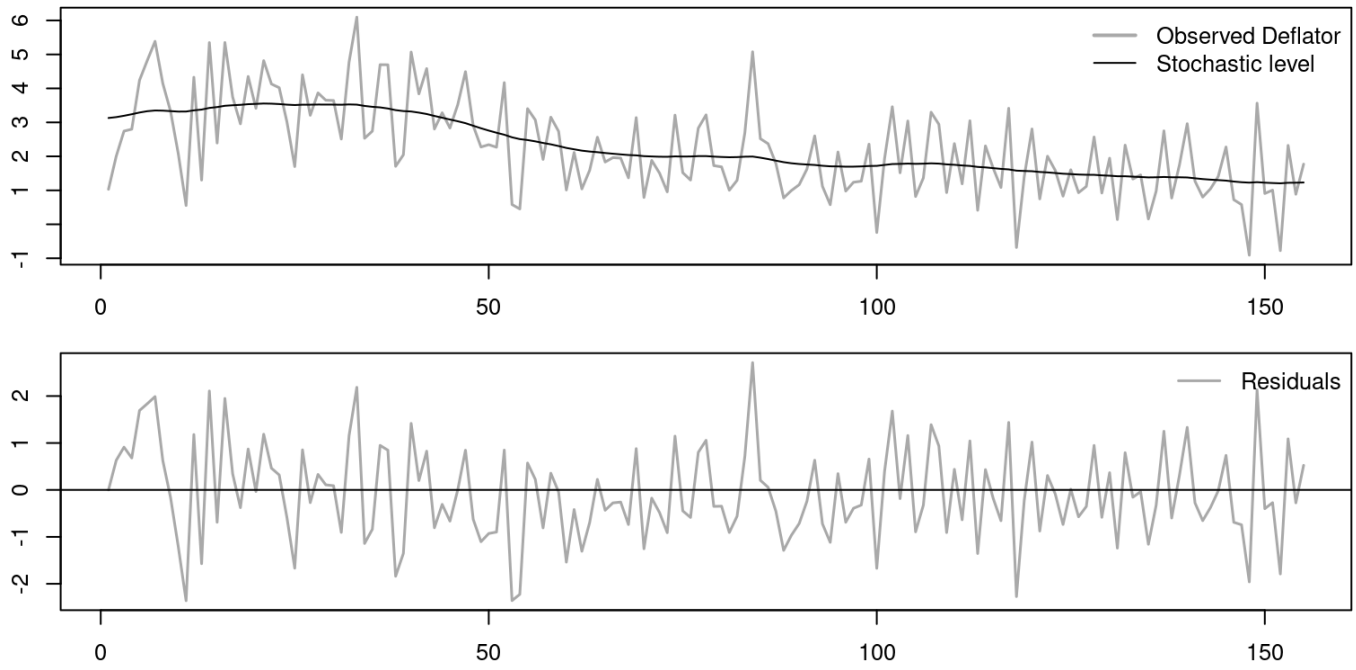
Thereafter, we can look at the results from the Kalman filter to derive the measurement errors. These have been stored in the `resids` object. We could also plot the smoothed values of the stochastic trend, which has been labeled `mu`.

```
resids <- residuals(filtered, sd = FALSE)
mu <- dropFirst(smoothed$s)
mu.1 <- mu[1]
mu.end <- mu[length(mu)]
```

1.4 Plot the results

To plot the results on two graphs, we use the `mfrow` command to create two rows and one column for the plot area. Thereafter, the first plot includes data for the actual variable and the black line refers to the stochastic trend and the grey line depicts the actual data. The graph below depicts the measurement errors.

```
par(mfrow = c(2, 1),
    mar = c(2.2, 2.2, 1, 1),
    cex = 0.8)
plot.ts(
  y,
  col = "darkgrey",
  xlab = "",
  ylab = "",
  lwd = 1.5
)
lines(mu , col = "black")
legend(
  "topright",
  legend = c("Observed Deflator", "Stochastic level"),
  lwd = c(2, 1),
  col = c("darkgrey", "black"),
  bty = "n"
)

plot.ts(
  resids,
  ylab = "",
  xlab = "",
  col = "darkgrey",
  lwd = 1.5
)
abline(h = 0)
legend(
  "topright",
  legend = "Residuals",
  lwd = 1.5,
  col = "darkgrey",
  bty = "n"
)
```



We can also print the results of the information criteria, as well as the variance of the respective errors.

```
cat("AIC", r.aic)
```

```
## AIC 274.6485
```

```
cat("BIC", r.bic)
```

```
## BIC 280.7353
```

```
cat("V.variance", obs.error.var)
```

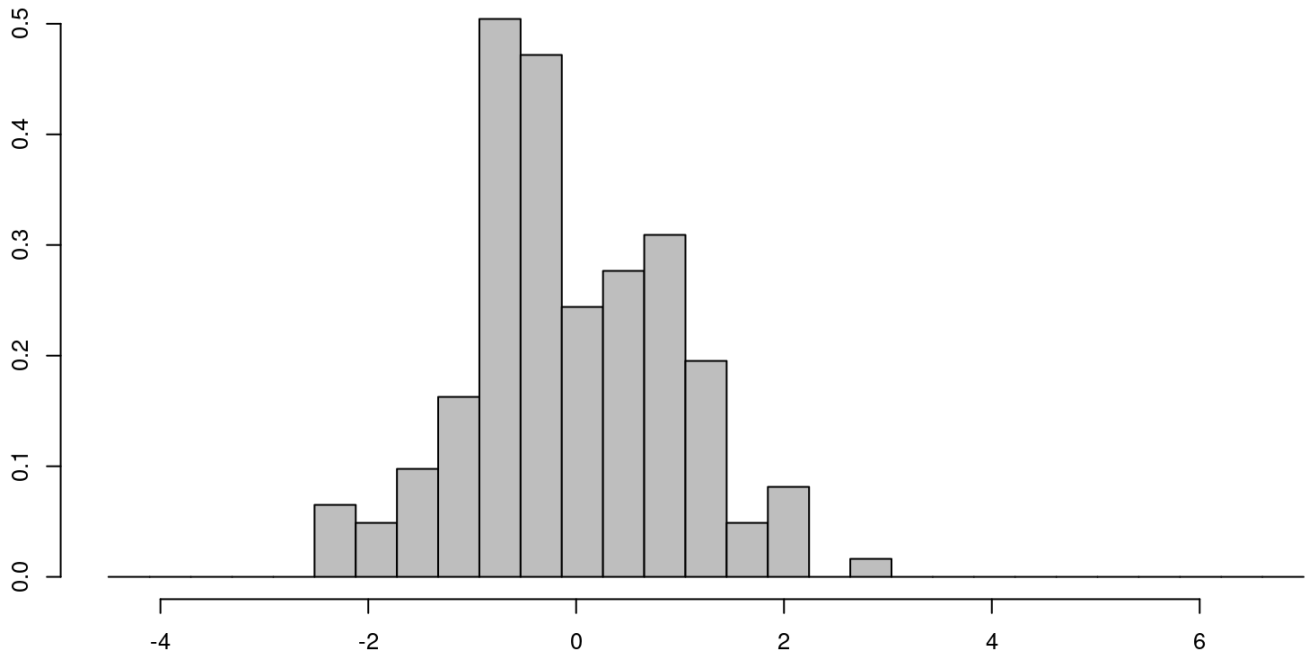
```
## V.variance 1.185696
```

```
cat("W.variance", state.error.var)
```

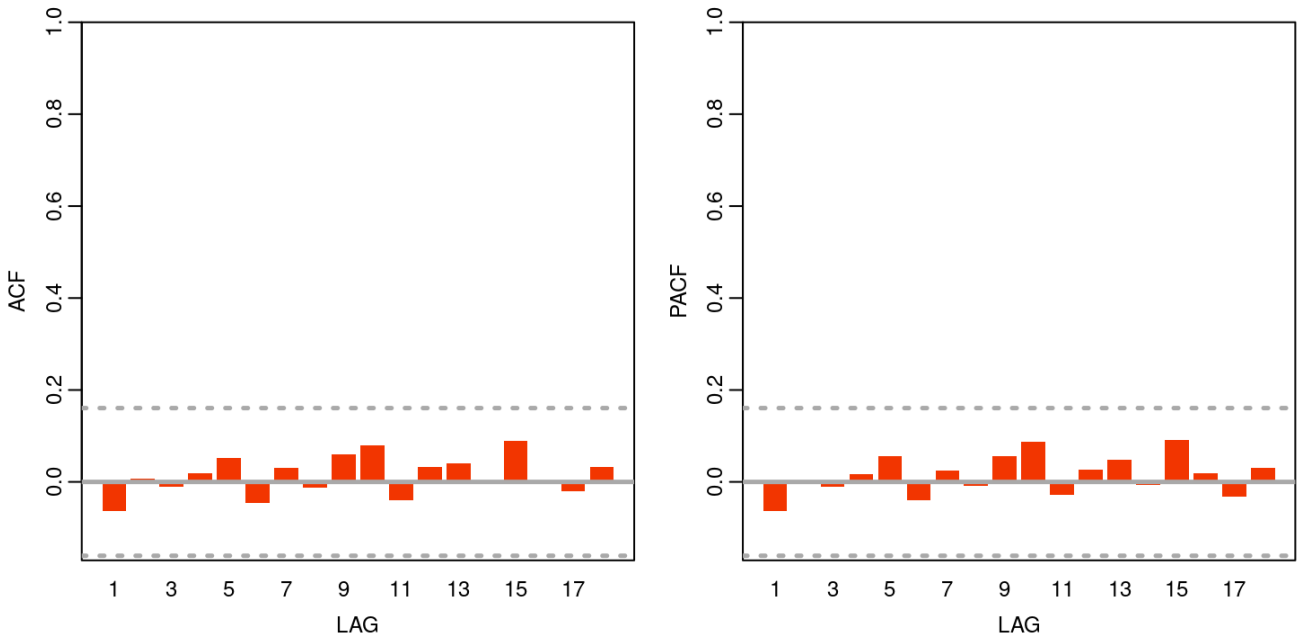
```
## W.variance 0.01443121
```

The diagnostics of the residuals, which are included below, suggest that there is no evidence of remaining serial correlation and their distribution is relatively normal.

```
par(mfrow = c(1, 1),
    mar = c(2.2, 2.2, 1, 1),
    cex = 0.8)
hist(
  resids,
  prob = TRUE,
  col = "grey",
  main = "",
  breaks = seq(-4.5, 7, length.out = 30)
)
```




```
ac(resids)  # acf
```




```
Box.test(resids,
         lag = 12,
         type = "Ljung",
         fitdf = 2)  # joint autocorrelation
```

```
##
##  Box-Ljung test
##
## data:  resids
## X-squared = 3.7862, df = 10, p-value = 0.9565
```

```
shapiro.test(resids)  # normality
```

```
##
##  Shapiro-Wilk normality test
##
## data:  resids
## W = 0.99046, p-value = 0.3817
```

In this case all appears to be satisfactory, as there is no unexplained autocorrelation when the p-value is less than 5% and the Shapiro test would suggest that the residuals are normally distributed when the p-value is less than 5%.

2 The local level trend model

An example of the local level trend model is contained in the file `T5-llm_trend.R`. To estimate values for the parameters and unobserved components for this model, we could start off exactly as we did before.

```
rm(list=ls())
graphics.off()

dat <- sarb_quarter %>%
  select(date, KBP6006L, KBP6006D) %>%
  mutate(defl = c(0, diff(log(KBP6006L / KBP6006D) * 100, lag = 1))) %>%
  dplyr::filter(date > '1981-01-01')

y <- dat$defl
plot.ts(y)
```

2.1 Construct the model and estimate parameters

However, in this case we need to incorporate an additional state equation for the slope of the model. This is a second order polynomial model and as there are two independent stochastic errors we also need to include this feature in the model.

```
fn <- function(parm) {
  dlmModPoly(order = 2,
             dV = exp(parm[1]),
             dW = exp(parm[2:3]))
}
```

We can then calculate the information criteria in the same way that we did previously, using the following commands.

```
fit <- dlmMLE(y, rep(0, 3), build = fn, hessian = TRUE)
conv <- fit$convergence  # zero for converged

loglik <- dlmLL(y, dlmModPoly(2))
n.coef <- 3
r.aic <- (2 * (loglik)) + 2 * (sum(n.coef)) #dlmLL caculates the neg. LL
r.bic <- (2 * (loglik)) + (log(length(y))) * (n.coef)

mod <- fn(fit$par)
obs.error.var <- V(mod)
state.error.var <- diag(W(mod))
```

2.2 Kalman filter and smoother

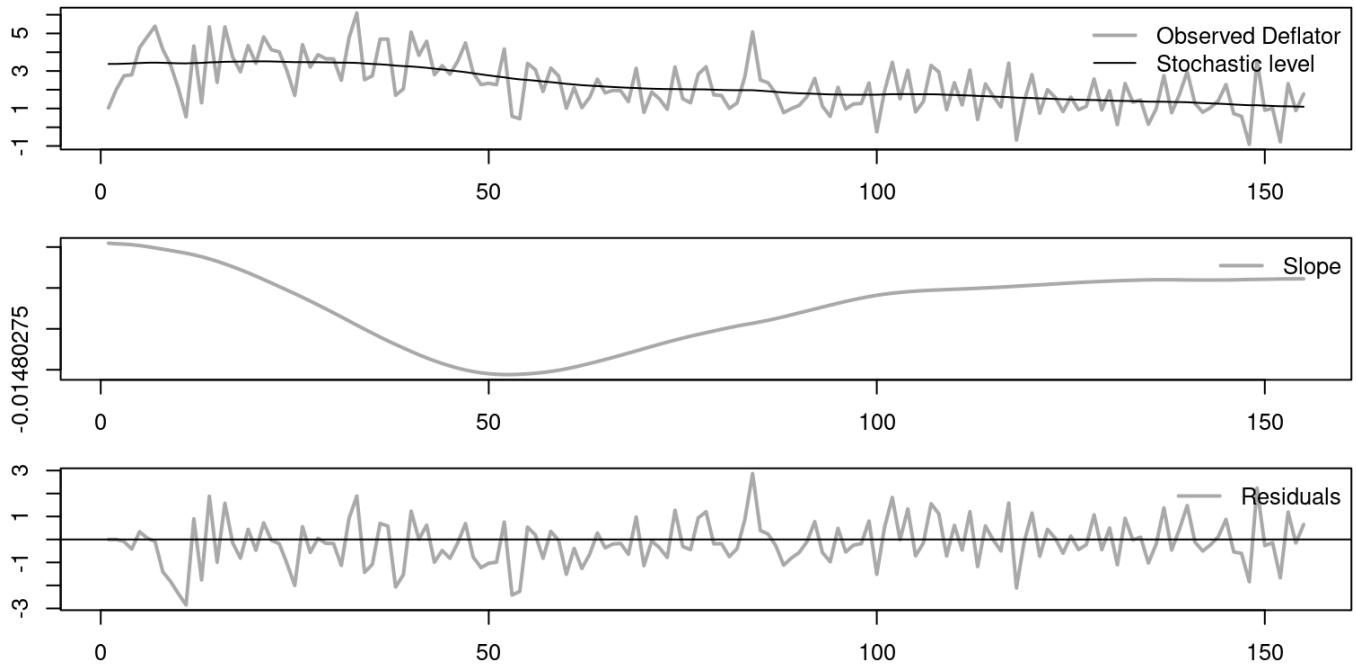
The values for the Kalman filter and smoother could also be extracted as before, but note that in this case we are also interested in the values of the stochastic slope, which is allocated to the `upsilon` object.

```
filtered <- dlmFilter(y, mod = mod)
smoothed <- dlmSmooth(filtered)
resids <- residuals(filtered, sd = FALSE)
mu <- dropFirst(smoothed$s[, 1])
upsilon <- dropFirst(smoothed$s[, 2])
mu.1 <- mu[1]
mu.end <- mu[length(mu)]
ups.1 <- upsilon[1]
ups.end <- upsilon[length(mu)]
```

2.3 Plot the results

We are now going to plot the results for the three graphs underneath one another, using the commands.

```
par(mfrow = c(3, 1),
    mar = c(2.2, 2.2, 1, 1),
    cex = 0.8)
plot.ts(
  y,
  col = "darkgrey",
  xlab = "",
  ylab = "",
  lwd = 2
)
lines(mu , col = "black")
legend(
  "topright",
  legend = c("Observed Deflator", "Stochastic level"),
  lwd = c(2, 1),
  col = c("darkgrey", "black"),
  bty = "n"
)

plot.ts(
  upsilon,
  col = "darkgrey",
  xlab = "",
  ylab = "",
  lwd = 2
)
legend(
  "topright",
  legend = "Slope",
  lwd = 2,
  col = "darkgrey",
  bty = "n"
)

plot.ts(
  resids,
  ylab = "",
  xlab = "",
  col = "darkgrey",
  lwd = 2
)
abline(h = 0)
legend(
  "topright",
  legend = "Residuals",
  lwd = 2,
  col = "darkgrey",
  bty = "n"
)
```



We can also print the results of the information criteria, as well as the variance of the respective errors.

```
cat("AIC", r.aic)
```

```
## AIC 380.3524
```

```
cat("BIC", r.bic)
```

```
## BIC 389.4827
```

```
cat("V.variance", obs.error.var)
```

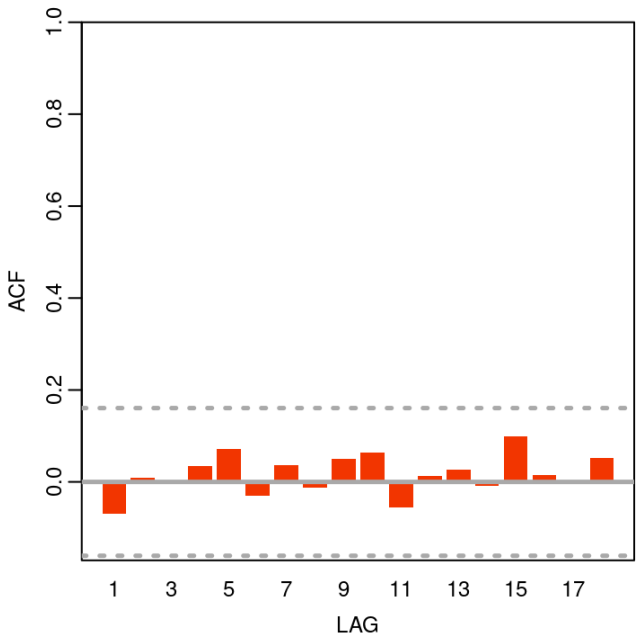
```
## V.variance 1.202044
```

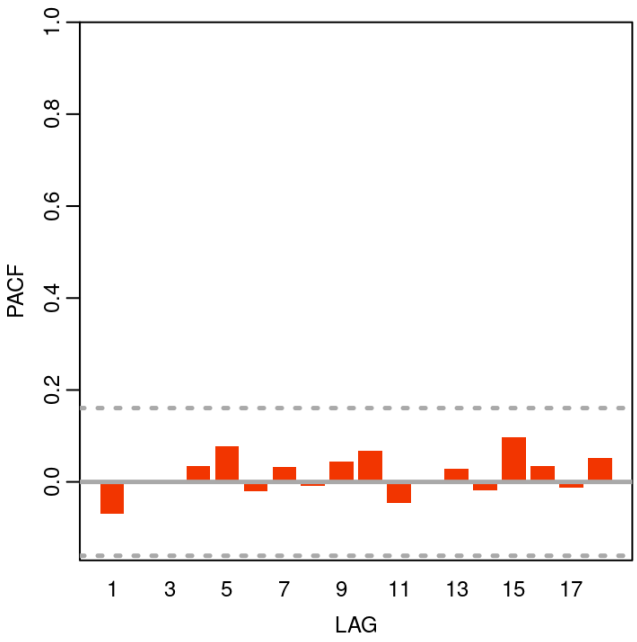
```
cat("W.variance", state.error.var)
```

```
## W.variance 0.008605704 8.253995e-12
```

The diagnostics of the residuals, which are included below, suggest that there is no evidence of remaining serial correlation and their distribution is relatively normal.

```
ac(resids)  # acf
```






```
Box.test(resids, lag=12, type="Ljung", fitdf=2)  # joint autocorrelation
```

```
##
##  Box-Ljung test
##
## data:  resids
## X-squared = 3.8828, df = 10, p-value = 0.9525
```

```
shapiro.test(resids)  # normality
```

```
##
##  Shapiro-Wilk normality test
##
## data:  resids
## W = 0.99347, p-value = 0.7121
```

3 The local level model with seasonal

An example of the local level trend model is contained in the file `T5-llm_seas.R`. To estimate values for the parameters and unobserved components in this model we could start off exactly as we did before.

```
rm(list=ls())
graphics.off()

dat <- sarb_quarter %>%
   select(date, KBP6006L, KBP6006D) %>%
   mutate(defl = c(0, diff(log(KBP6006L / KBP6006D) * 100, lag = 1))) %>%
   dplyr::filter(date > '1981-01-01')

y <- dat$defl
plot.ts(y)
```

3.1 Construct the model and estimate parameters

However, in this case we need to incorporate an additional state equations for the seasonal components. This is a first order polynomial model with the additional seasonal component that is measured at a quarterly frequency.

```
fn <- function(parm) {
   mod = dlmModPoly(order = 1) + dlmModSeas(frequency = 4)
   V(mod) = exp(parm[1])
   diag(W(mod))[1:2] = exp(parm[2:3])
   return(mod)
}
```

We can then calculate the information criteria in the same way that we did previously, using the following commands.

```
fit <- dlmMLE(y, rep(0,3), build=fn, hessian=TRUE)
conv <- fit$convergence  # zero for converged

loglik <- dlmLL(y, dlmModPoly(1)+dlmModSeas(4))

n.coef <- 3
r.aic <- (2*(loglik)) + 2*(sum(n.coef)) #dlmLL caculates the neg. LL
r.bic <- (2*(loglik)) + (log(length(y)))*(n.coef)

mod <- fn(fit$par)
obs.error.var <- V(mod)
state.error.var <- diag(W(mod))
```

3.2 Kalman filter and smoother

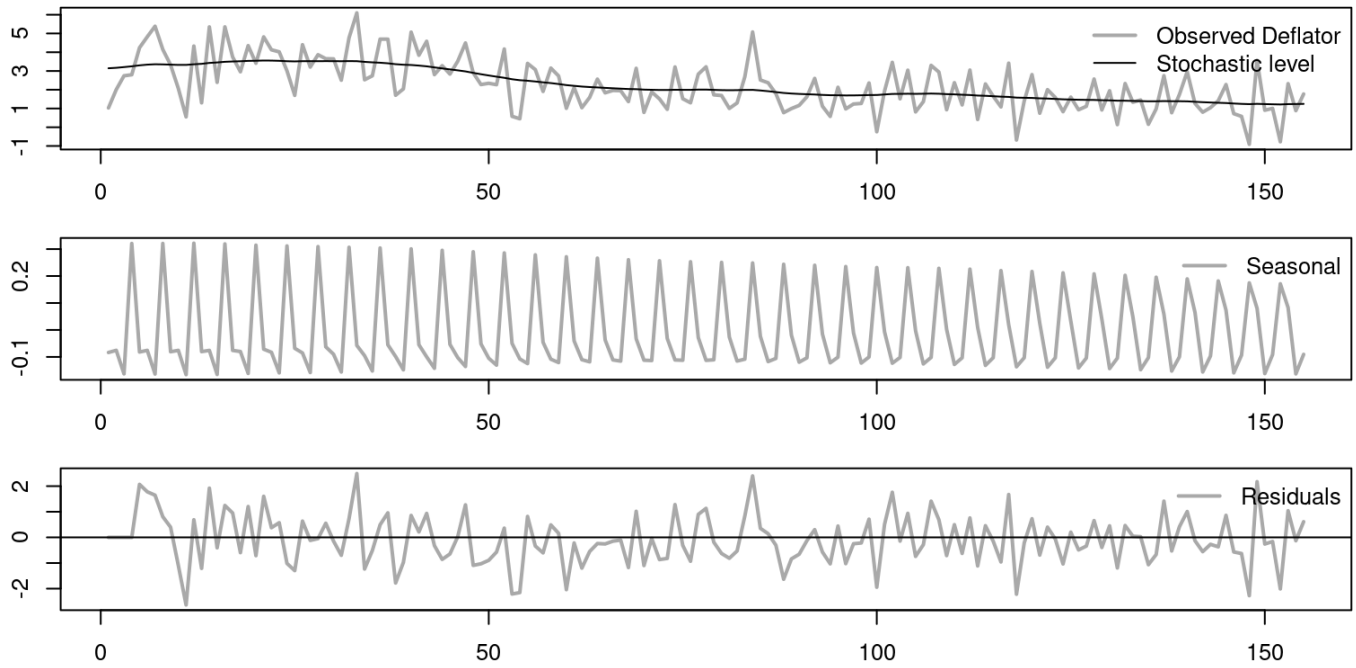
The values for the Kalman filter and smoother could also be extracted as before, but note that in this case we are also interested in the values of the seasonal, which is allocated to the `gamma` object.

```
filtered <- dlmFilter(y, mod=mod)
smoothed <- dlmSmooth(filtered)
resids <- residuals(filtered,sd=FALSE)
mu <- dropFirst(smoothed$s[,1])
gammas <- dropFirst(smoothed$s[,2])
mu.1 <- mu[1]
mu.end <- mu[length(mu)]
gammas.1 <- gammas[1]
gammas.end <- gammas[length(mu)]
```

3.3 Plot the results

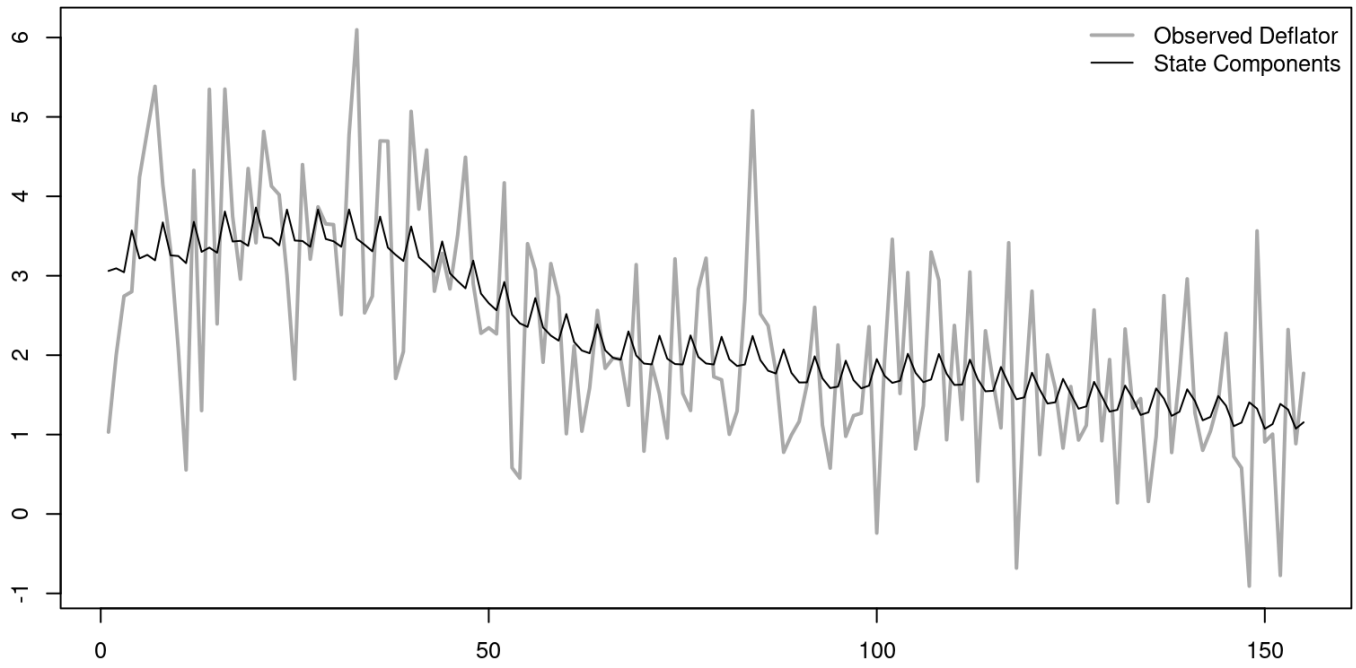
We are now going to plot the results for the three graphs underneath one another, using the commands.

```
par(mfrow=c(3,1),mar=c(2.2,2.2,1,1),cex=0.8)
plot.ts(y, col="darkgrey", xlab="", ylab="", lwd=2)
lines(mu , col="black")
legend("topright", legend=c("Observed Deflator","Stochastic level"),
       lwd=c(2,1), col=c("darkgrey","black"), bty="n")

plot.ts(gammas, col="darkgrey", xlab="", ylab="", lwd=2)
legend("topright", legend="Seasonal", lwd=2, col="darkgrey", bty="n")

plot.ts(resids, ylab="", xlab="", col="darkgrey", lwd=2)
abline(h=0)
legend("topright", legend="Residuals", lwd=2, col="darkgrey", bty="n")
```



Of course, as these model elements are additive, we could combine the seasonal and the level, as in the following graph.

```
alpha <- mu + gammas

par(mfrow=c(1,1),mar=c(2.2,2.2,1,1),cex=0.8)
plot.ts(y, col="darkgrey", xlab="", ylab="", lwd=2)
lines(alpha , col="black")
legend("topright", legend=c("Observed Deflator","State Components"),
       lwd=c(2,1), col=c("darkgrey","black"), bty="n")
```



We can also print the results of the information criteria, as well as the variance of the respective errors.

```
cat("AIC",r.aic)
```

```
## AIC 430.8364
```

```
cat("BIC",r.bic)
```

```
## BIC 439.9667
```

```
cat("V.variance",obs.error.var)
```

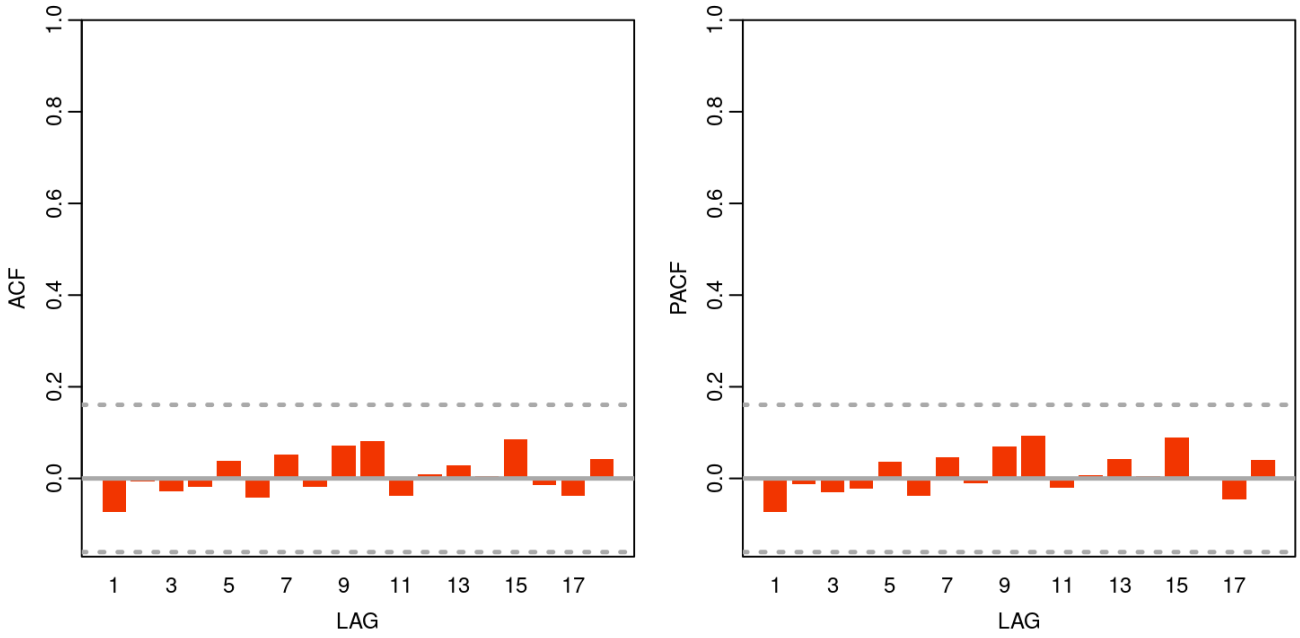
```
## V.variance 1.176257
```

```
cat("W.variance",state.error.var)
```

```
## W.variance 0.01434486 0.0006917049 0 0
```

The diagnostics of the residuals, which are included below, suggest that there is no evidence of remaining serial correlation and their distribution is relatively normal.

```
ac(resids)  # acf
```




```
Box.test(resids, lag=12, type="Ljung", fitdf=2)  # joint autocorrelation
```

```
##
##  Box-Ljung test
##
## data:  resids
## X-squared = 4.267, df = 10, p-value = 0.9345
```

```
shapiro.test(resids)  # normality
```

```
##
##  Shapiro-Wilk normality test
##
## data:  resids
## W = 0.99037, p-value = 0.3732
```

4 Confidence intervals

To create confidence intervals around the estimates from the Kalman filter or smoother, we can use the example of the local level model. An example of a local level model with confidence intervals is contained in `T5-llm_conf.R` , which contains the following commands:

```
rm(list=ls())
graphics.off()

dat <- sarb_quarter %>%
   select(date, KBP6006L, KBP6006D) %>%
   mutate(defl = c(0, diff(log(KBP6006L / KBP6006D) * 100, lag = 1))) %>%
   dplyr::filter(date > '1981-01-01')

y <- dat$defl
plot.ts(y)
```

Thereafter, we can use the same model structure that we had originally.

```
fn <- function(parm) {
   dlmModPoly(order = 1,
              dV = exp(parm[1]),
              dW = exp(parm[2]))
}

fit <- dlmMLE(y, rep(0, 2), build = fn, hessian = TRUE)
conv <- fit$convergence  # zero for converged

loglik <- dlmLL(y, dlmModPoly(1))
n.coef <- 2
r.aic <- (2 * (loglik)) + 2 * (sum(n.coef)) #dlmLL caculates the neg. LL
r.bic <- (2 * (loglik)) + (log(length(y))) * (n.coef)

mod <- fn(fit$par)
obs.error.var <- V(mod)
state.error.var <- W(mod)

filtered <- dlmFilter(y, mod = mod)
smoothed <- dlmSmooth(filtered)
resids <- residuals(filtered, sd = FALSE)
mu <- dropFirst(smoothed$s)
mu.1 <- mu[1]
mu.end <- mu[length(mu)]
```

4.1 Constructing confidence intervals

To construct the confidence intervals around the smoothed stochastic trend, we would invoke the commands:

```
conf.tmp <- unlist(dlmSvd2var(smoothed$U.S, smoothed$D.S))
conf <- ts(as.numeric(conf.tmp)[-1],
           start = c(1960, 2),
           frequency = 4)
wid <- qnorm(0.05, lower = FALSE) * sqrt(conf)

conf.pos <- mu + wid
conf.neg <- mu - wid
```

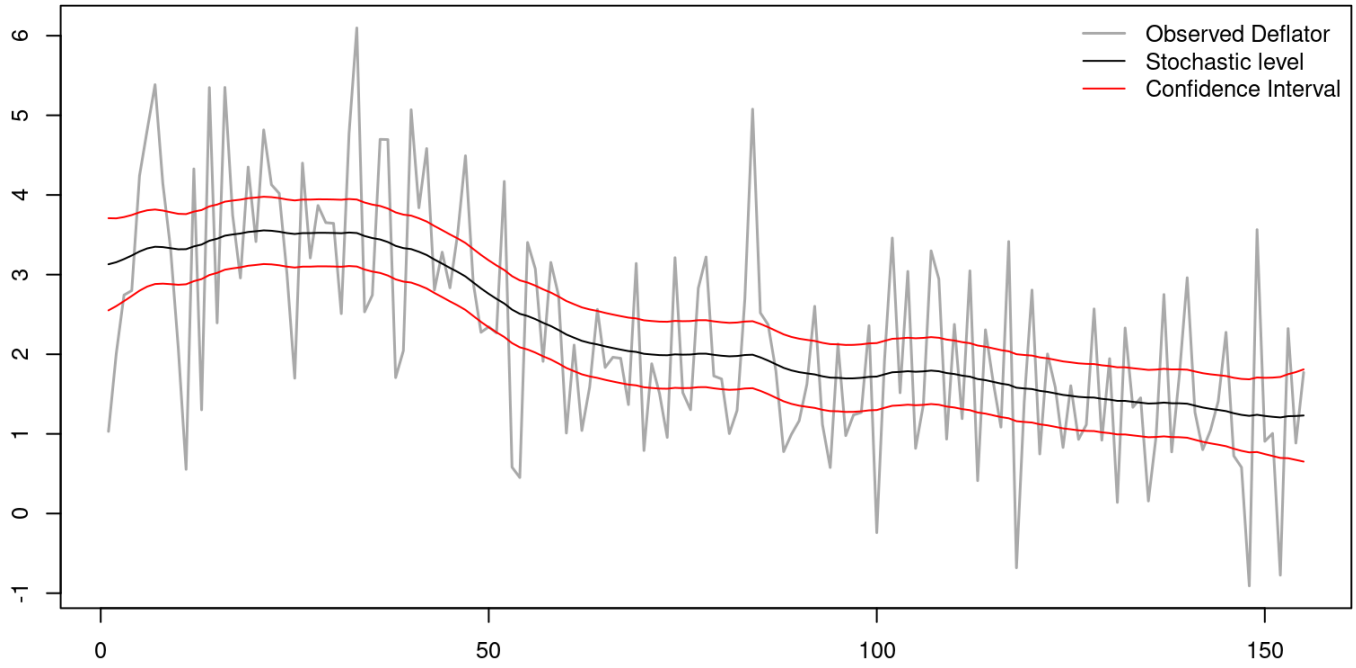
Similarly, if we wanted to do the same for the filtered values we would utilise information relating to the prediction errors.

```
mu.f <- dropFirst(filtered$a)
cov.tmp <- unlist(dlmSvd2var(filtered$U.R, filtered$D.R))
if (sum(dim(mod$FF)) == 2) {
   variance <- cov.tmp + as.numeric(V(mod))
} else {
   variance <-
      (sapply(cov.tmp, function(x)
         mod$FF %*% x %*% t(mod$FF))) + V(mod)
}
```

4.2 Plotting the results

This would allow us to plot the results

```
par(mfrow = c(1, 1),
    mar = c(2.2, 2.2, 1, 1),
    cex = 0.8)
plot.ts(
   y,
   col = "darkgrey",
   xlab = "",
   ylab = "",
   lwd = 1.5
)
lines(mu, col = "black")
lines(as.numeric(conf.pos) , col = "red")
lines(as.numeric(conf.neg), col = "red")
legend(
   "topright",
   legend = c("Observed Deflator", "Stochastic level", "Confidence Interval"),
   lwd = c(1.5, 1, 1),
   col = c("darkgrey", "black", "red"),
   bty = "n"
)
```



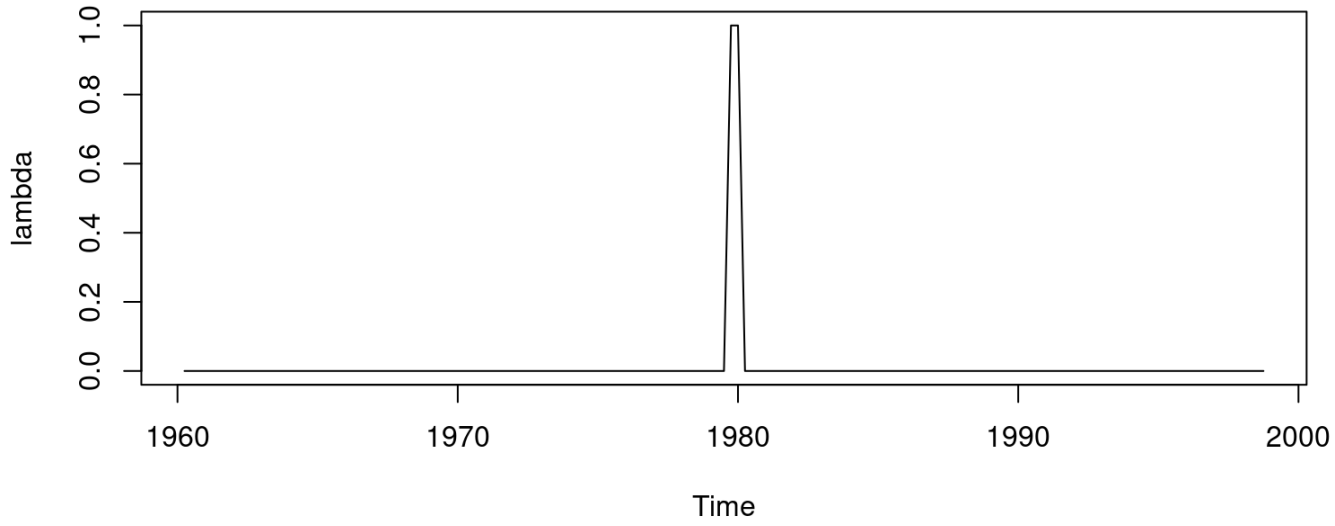
5 Intervention variables

To incorporate a dummy variable we will use the example of the local level model once again. Such an example is contained in the file `T5-llm_int.R`.

```
rm(list=ls())
graphics.off()

dat <- sarb_quarter %>%
   select(date, KBP6006L, KBP6006D) %>%
   mutate(defl = c(0, diff(log(KBP6006L / KBP6006D) * 100, lag = 1))) %>%
   dplyr::filter(date > '1981-01-01')

y <- dat$defl
plot.ts(y)
```

To then create the dummy variable we create a vector of zeros that is called `lambda`, which is the same length as the sample of data. We then include a value of `1` for the first two quarters in 1980 (which relates to observation 79 and 80). This vector of values is then transformed to a time series object.

```
lambda <- rep(0,length(y))
lambda[79:80] <- 1
lambda <- ts(lambda, start=c(1960,2), frequency=4)
plot(lambda)
```



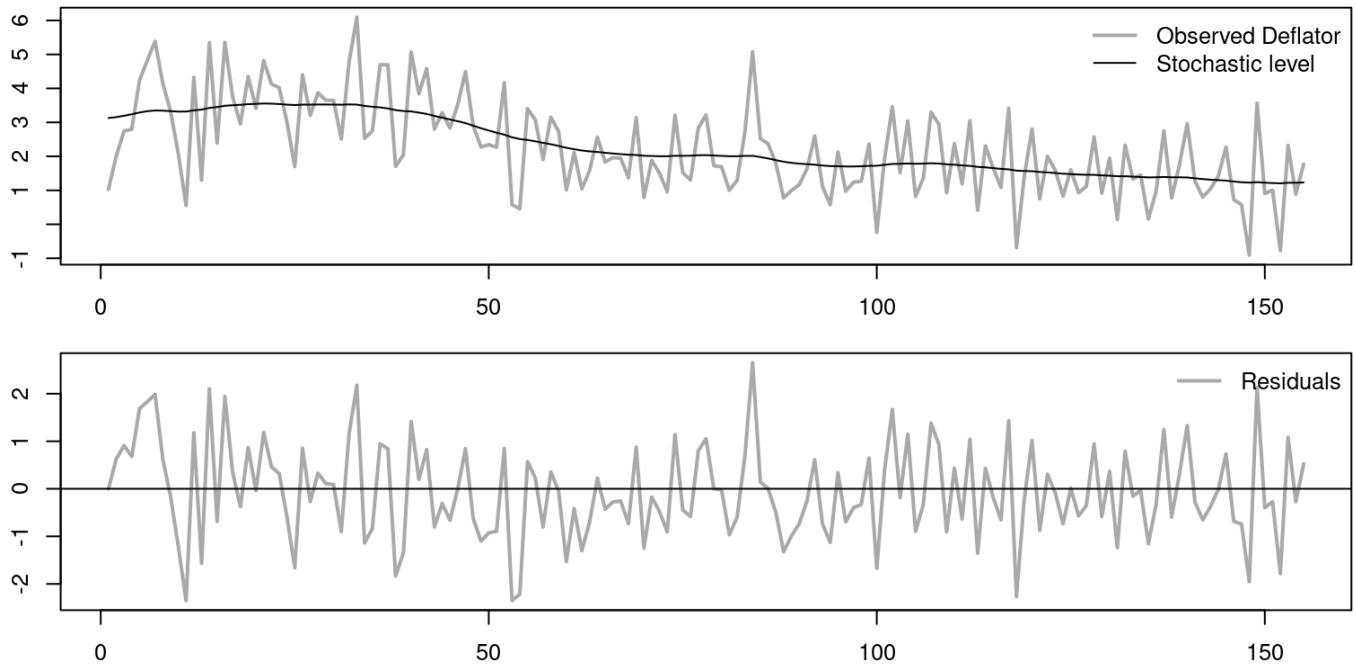
The model structure needs to incorporate this dummy variable, which is an additional regressor in the measurement equation. The rest of the code is pretty straightforward and is largely as it was before.

```
fn <- function(parm){
  mod=dlmModPoly(order=1)+dlmModReg(lambda, addInt=FALSE)
  V(mod)=exp(parm[1])
  diag(W(mod))[1]=exp(parm[2])
  return(mod)
}

fit <- dlmMLE(y, rep(0,2), build=fn, hessian=TRUE)
conv <- fit$convergence  # zero for converged

loglik <- dlmLL(y, dlmModPoly(1)+dlmModReg(lambda, addInt=FALSE))
n.coef <- 2
r.aic <- (2*(loglik)) + 2*(sum(n.coef)) #dlmLL caculates the neg. LL
r.bic <- (2*(loglik)) + (log(length(y)))*(n.coef)

mod <- fn(fit$par)
obs.error.var <- V(mod)
state.error.var <- W(mod)

filtered <- dlmFilter(y, mod=mod)
smoothed <- dlmSmooth(filtered)
resids <- residuals(filtered,sd=FALSE)
mu <- dropFirst(smoothed$s[,1])
mu.1 <- mu[1]
mu.end <- mu[length(mu)]
```

5.1 Plotting the results

The plotting of the results would also be very similar to what we had previously.

```
par(mfrow=c(2,1),mar=c(2.2,2.2,1,1),cex=0.8)
plot.ts(y, col="darkgrey", xlab="", ylab="", lwd=2)
lines(mu , col="black")
legend("topright", legend=c("Observed Deflator","Stochastic level"),
       lwd=c(2,1), col=c("darkgrey","black"), bty="n")

plot.ts(resids, ylab="", xlab="", col="darkgrey", lwd=2)
abline(h=0)
legend("topright", legend="Residuals", lwd=2, col="darkgrey", bty="n")
```



6 Forecasting

To use the model to forecast forward, consider the example that is contained in the file `T5-llm_fore.R` . The initial lines of code are exactly the same as what we've had all along.

```
rm(list=ls())
graphics.off()

dat <- sarb_quarter %>%
   select(date, KBP6006L, KBP6006D) %>%
   mutate(defl = c(0, diff(log(KBP6006L / KBP6006D) * 100, lag = 1))) %>%
   dplyr::filter(date > '1981-01-01')

y <- dat$defl
plot.ts(y)
```

We can use a local level model structure again, which may be constructed as before, where we include confidence intervals.

```
fn <- function(parm) {
  dlmModPoly(order = 1,
             dV = exp(parm[1]),
             dW = exp(parm[2]))
}

fit <- dlmMLE(y, rep(0, 2), build = fn, hessian = TRUE)
conv <- fit$convergence  # zero for converged

loglik <- dlmLL(y, dlmModPoly(1))
n.coef <- 2
r.aic <- (2 * (loglik)) + 2 * (sum(n.coef)) #dlmLL caculates the neg. LL
r.bic <- (2 * (loglik)) + (log(length(y))) * (n.coef)

mod <- fn(fit$par)
obs.error.var <- V(mod)
state.error.var <- W(mod)

filtered <- dlmFilter(y, mod = mod)
smoothed <- dlmSmooth(filtered)
resids <- residuals(filtered, sd = FALSE)
mu <- dropFirst(smoothed$s)

conf.tmp <- unlist(dlmSvd2var(smoothed$U.S, smoothed$D.S))
conf <- ts(conf.tmp[-1], start = c(1960, 2), frequency = 4)
wid <- qnorm(0.05, lower = FALSE) * sqrt(conf)

conf.pos <- mu + wid
conf.neg <- mu - wid

comb.state <- cbind(mu, conf.pos, conf.neg)
```

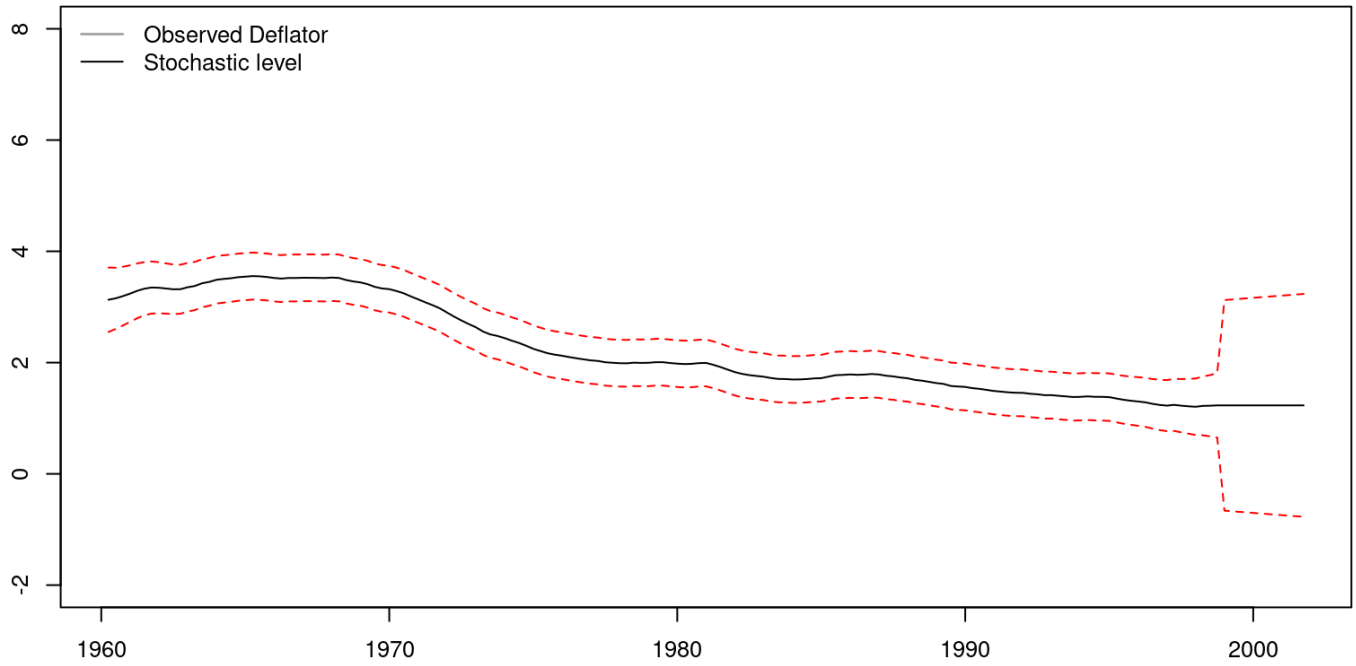
To forecast forward we use the `dlmForecast` command, where in this case we are interested in forecasting ten steps ahead.

```
forecast <- dlmForecast(filtered, nAhead = 12)
var.2 <- unlist(forecast$Q)
wid.2 <- qnorm(0.05, lower = FALSE) * sqrt(var.2)
comb.fore <- cbind(forecast$f, forecast$f + wid.2, forecast$f - wid.2)

result <-
  ts(rbind(comb.state, comb.fore),
     start = c(1960, 2),
     frequency = 4)
```

6.1 Plotting the results

To draw a graph that contains the results, we would utilise the following commands:

```
par(mfrow = c(1, 1),
    mar = c(2.2, 2.2, 1, 1),
    cex = 0.8)
plot.ts(
  result,
  col = c("black", "red", "red"),
  plot.type = "single",
  xlab = "",
  ylab = "",
  lty = c(1, 2, 2),
  ylim = c(-2, 8)
)
lines(y , col = "darkgrey", lwd = 1.5)
abline(
  v = c(2019, 4),
  col = 'blue',
  lwd = 1,
  lty = 3
)
legend(
  "topleft",
  legend = c("Observed Deflator", "Stochastic level"),
  lwd = c(1.5, 1),
  col = c("darkgrey", "black"),
  bty = "n"
)
```



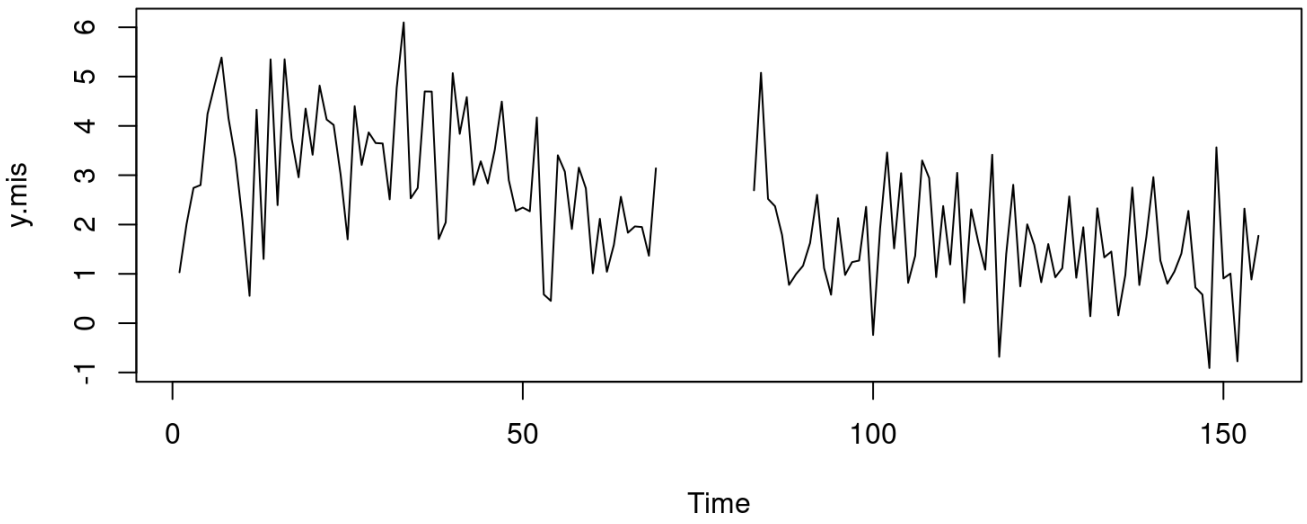
7 Missing data

To model data that has a number of missing observations, consider the example that is contained in the file `T5-1lm_miss.R` . The initial lines of code are exactly the same as what we've had all along.

```
rm(list=ls())
graphics.off()

dat <- sarb_quarter %>%
   select(date, KBP6006L, KBP6006D) %>%
   mutate(defl = c(0, diff(log(KBP6006L / KBP6006D) * 100, lag = 1))) %>%
   dplyr::filter(date > '1981-01-01')

y <- dat$defl
plot.ts(y)
```

To remove some data from the time series for inflation, we can set a number of observations equal to `NA`, which in this case relates to observations 70 to 82.

```
y.mis <- y
y.mis[70:82] <- NA
plot.ts(y.mis)
```



We can use a local level model structure again, which may be constructed as before, where the model will now be applied to the variable `inf.mis`.

```
fn <- function(parm) {
  dlmModPoly(order = 1,
             dV = exp(parm[1]),
             dW = exp(parm[2]))
}

## Estimate parameters & generate statistics
fit <- dlmMLE(y.mis, rep(0, 2), build = fn, hessian = TRUE)
conv <- fit$convergence  # zero for converged

loglik <- dlmLL(y.mis, dlmModPoly(1))
n.coef <- 2
r.aic <- (2 * (loglik)) + 2 * (sum(n.coef)) #dlmLL caculates the neg. LL
r.bic <- (2 * (loglik)) + (log(length(y))) * (n.coef)

mod <- fn(fit$par)
obs.error.var <- V(mod)
state.error.var <- W(mod)

filtered <- dlmFilter(y.mis, mod = mod)
smoothed <- dlmSmooth(filtered)
resids <- residuals(filtered, sd = FALSE)
mu <- dropFirst(smoothed$s)

conf.tmp <- dlmSvd2var(smoothed$U.S, smoothed$D.S)
conf <- ts(as.numeric(conf.tmp)[-1],
           start = c(1960, 2),
           frequency = 4)
wid <- qnorm(0.05, lower = FALSE) * sqrt(conf)

conf.pos <- mu + wid
conf.neg <- mu - wid

cat("AIC", r.aic)
```

```
## AIC 257.2241
```

```
cat("BIC", r.bic)
```

```
## BIC 263.311
```

```
cat("V.variance", obs.error.var)
```

```
## V.variance 1.229809
```

```
cat("W.variance", state.error.var)
```

```
## W.variance 0.01486668
```

7.1 Plotting the results

To draw a graph that contains the results, we would utilise the following commands:

```
par(mfrow = c(1, 1),
    mar = c(2.2, 2.2, 1, 1),
    cex = 0.8)
plot.ts(
  comb.state,
  col = c("black", "red", "red", "grey"),
  plot.type = "single",
  xlab = "",
  ylab = "",
  lty = c(1, 2, 2, 3),
  ylim = c(-2, 8)
)
legend(
  "topright",
  legend = c("Observed Deflator", "Stochastic level", "Confidence"),
  lwd = c(1.5, 1),
  col = c("darkgrey", "black", "red"),
  bty = "n"
)
```

8 Regression effects with time-varying parameters

To estimate a model with a time-varying parameter, we could make use of the data for the quantity sold and the price of spirits. This data is contained in a `csv` file that has been labelled `spirits.csv` and it contains information that pertains to two variables, *spirits* and *price*. As long as this file is in the same folder as the `*.Proj` file then it will not be necessary to specify the location of this file. If it is not in the same folder then you can set the working directory with the aid of the `setwd()` command. Note that if your data is saved in a `*.xls` or `*.xlsx` file then you could make use of the `readxl` package that makes use of similar routines.

```
rm(list=ls())
graphics.off()

#setwd("~/git/tsm/ts-4-tut/tut")

dat_tmp <- read_csv(file = "spirits.csv")
spirit <-
  ts(
    dat_tmp$spirits,
    start = c(1871, 1),
    end = c(1939, 1),
    frequency = 1
  )

price <-
  ts(
    dat_tmp$price,
    start = c(1871, 1),
    end = c(1939, 1),
    frequency = 1
  )

plot(ts.union(spirit, price), plot.type = "single")
lines(price, col = "red")
```

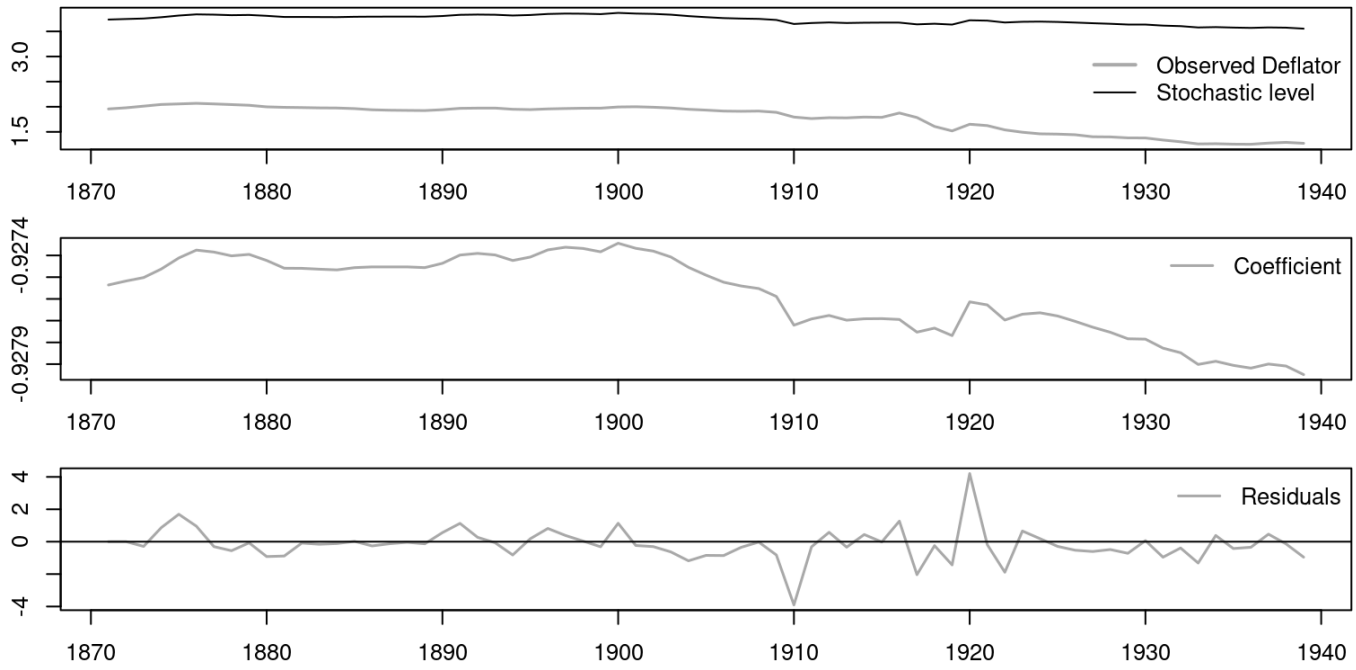
We can then construct the model in much the same way as we did before, where in this case we are making use as price as an explanatory variable that is subject to a time-varying parameter.

```
fn <- function(parm) {
  mod = dlmModPoly(order = 1) + dlmModReg(price, addInt = FALSE)
  V(mod) = exp(parm[1])
  diag(W(mod))[1:2] = exp(parm[2:3])
  return(mod)
}

fit <- dlmMLE(spirit, rep(0, 3), build = fn, hessian = TRUE)
conv <- fit$convergence  # zero for converged

loglik <-
  dlmLL(spirit, dlmModPoly(1) + dlmModReg(price, addInt = FALSE))

n.coef <- 3
r.aic <- (2 * (loglik)) + 2 * (sum(n.coef)) #dlmLL caculates the neg. LL
r.bic <- (2 * (loglik)) + (log(length(spirit))) * (n.coef)

mod <- fn(fit$par)
obs.error.var <- V(mod)
state.error.var <- diag(W(mod))

filtered <- dlmFilter(spirit, mod = mod)
smoothed <- dlmSmooth(filtered)
resids <- residuals(filtered, sd = FALSE)
mu <- dropFirst(smoothed$s[, 1])
betas <- dropFirst(smoothed$s[, 2])
mu.1 <- mu[1]
mu.end <- mu[length(mu)]
beta.1 <- betas[1]
beta.end <- betas[length(mu)]
```

8.1 Plotting the results

We can then display the results in much the same way as we did before, where we make use of three graphs that are displayed underneath one another.

```
par(mfrow = c(3, 1),
    mar = c(2.2, 2.2, 1, 1),
    cex = 0.8)
plot.ts(
  ts.union(spirit, mu),
  col = c("darkgrey", "black"),
  plot.type = "single",
  xlab = "",
  ylab = "",
  lwd = c(1.5, 1)
)
legend(
  "right",
  legend = c("Observed Deflator", "Stochastic level"),
  lwd = c(2, 1),
  col = c("darkgrey", "black"),
  bty = "n"
)

plot.ts(
  betas,
  col = "darkgrey",
  xlab = "",
  ylab = "",
  lwd = 1.5
)
legend(
  "topright",
  legend = "Coefficient",
  lwd = 1.5,
  col = "darkgrey",
  bty = "n"
)

plot.ts(
  resids,
  ylab = "",
  xlab = "",
  col = "darkgrey",
  lwd = 1.5
)
abline(h = 0)
legend(
  "topright",
  legend = "Residuals",
  lwd = 1.5,
  col = "darkgrey",
  bty = "n"
)
```



9 Coding the Kalman Filter

To both understand and appreciate each of the individual steps of the Kalman filter and Kalman smoother, you may want to take a look at the following code, that was used to generate the example hypothetical example where we made use of six observations for y_t. This code makes use of general form of the model and matrix algebra for all the calculations, so it could be used for any of the models that we discussed. Note that in this case the values for variance of the errors take on calibrated values, while all of the above models estimate these values to provide the most appropriate fit of the data.

To clear the working environment and close all figures before loading the six observations that we used in class, we would proceed in the normal manner.

```
rm(list=ls())
graphics.off()

yt <- c(6.07, 6.09, 5.89, 5.83, 6.00, 6.03)
num <- length(yt)
```

Thereafter, the setup filter and smoother could make use of the following values:

```
alpha0 <- mean(yt)    # assume alpha_0 = N(mean(yt),1)
sigma0 <- 1           # assume alpha_0 = N(mean(yt),1)
Gt <- 1        # coefficient in state equation
Ft <- 1        # coefficient in measurement equation
cW <- 1        # state variance-covariance matrix
cV <- 1        # measurement variance-covariance matrix
```

We could establish the matrices for the Kalman filter with the aid of the following commands.

```
W <- t(cW) %*% cW  # convert scalar to matrix
V <- t(cV) %*% cV  # convert scalar to matrix

Gt <- as.matrix(Gt)
pdim <- nrow(Gt)
yt <- as.matrix(yt)
qdim <- ncol(yt)
alphap <- array(NA, dim = c(pdim, 1, num))     # alpha_p= a_{t+1}
Pp <- array(NA, dim = c(pdim, pdim, num))      # Pp=P_{t+1}
alphaf <- array(NA, dim = c(pdim, 1, num))     # alpha_f=a_t
Pf <- array(NA, dim = c(pdim, pdim, num))      # Pf=P_t
innov <- array(NA, dim = c(qdim, 1, num))      # innovations
sig <- array(NA, dim = c(qdim, qdim, num))     # innov var-cov matrix
Kmat <- rep(NA, num)                           # store Kalman gain
```

Before we initialise the filter.

```
alpha00 <- as.matrix(alpha0, nrow = pdim, ncol = 1)  # state: mean starting value
P00 <- as.matrix(sigma0, nrow = pdim, ncol = pdim)   # state: variance start value
alphap[, , 1] <- Gt %*% alpha00                      # predicted value a_{t+1}
Pp[, , 1] <- Gt %*% P00 %*% t(Gt) + W                   # variance for predicted state value
sigtemp <- Ft %*% Pp[, , 1] %*% t(Ft) + V               # variance for measurement value
sig[, , 1] <- (t(sigtemp) + sigtemp) / 2                # innov var - make sure it's symmetri
        c
siginv <- solve(sig[, , 1])                             # 1 / innov var
K <- Pp[, , 1] %*% t(Ft) %*% siginv                     # K_t = predicted variance / innov va
        riance
Kmat[1] <- K                                            # store Kalman gain
innov[, , 1] <- yt[1, ] - Ft %*% alphap[, , 1]          # epsilon_t = y_t - a_t
alphaf[, , 1] <- alphap[, , 1] + K %*% innov[, , 1]     # a_{t+1} = a_t + K_t(epsilon_t)
Pf[, , 1] <- Pp[, , 1] - K %*% Ft %*% Pp[, , 1]              # variance of forecast
sigmat <- as.matrix(sig[, , 1], nrow = qdim, ncol = qdim)    # collect variance of measuremen
        t errors
like <- log(det(sigmat)) + t(innov[, , 1]) %*% siginv %*% innov[, , 1]   # calculate -log(lik
        elihood)
```

After the filter is initialised we could then use a `for` loop for the iterations of the Kalman filter.

```
for (id in 2:num) {
  if (num < 2)
    break
  alphap[, , id] <- Gt %*% alphaf[, , id - 1]         # predicted value a_{t+2}
  Pp[, , id] <- Gt %*% Pf[, , id - 1] %*% t(Gt) + W   # variance of predicted state estimate
  sigtemp <- Ft %*% Pp[, , id] %*% t(Ft) + V         # variance of measurement error
  sig[, , id] <- (t(sigtemp) + sigtemp) / 2          # innov var - make sure it's symmetric
  siginv <- solve(sig[, , id])
  K <- Pp[, , id] %*% t(Ft) %*% siginv               # K_t = predicted variance / innov varia
        nce
  Kmat[id] <- K                                      # store Kalman gain
  innov[, , id] <- yt[id, ] - Ft %*% alphap[, , id]    # epsilon_t = y_t - a_t
  alphaf[, , id] <- alphap[, , id] + K %*% innov[, , id]   # a_{t+1} = a_t + K_t(epsilon_t)
  Pf[, , id] <- Pp[, , id] - K %*% Ft %*% Pp[, , id]       # variance of forecast
  sigmat <- as.matrix(sig[, , id], nrow = qdim, ncol = qdim)   # collect variance of measurem
        ent errors
  like <- like + log(det(sigmat)) + t(innov[, , id]) %*% siginv %*% innov[, , id]  # calculat
        e -log(likelihood)
}
```

These results could then be stored in the following two objects:

```
like <- 0.5 * like

kf <- list(
  alphap = alphap,
  Pp = Pp,
  alphaf = alphaf,
  Pf = Pf,
  like = like,
  innov = innov,
  sig = sig,
  Kn = K
)
```

The next step would be to initialise the Kalman smoother.

```
pdim <- nrow(as.matrix(Gt))
alphas <- array(NA, dim = c(pdim, 1, num))      # alpha_s = smoothed alpha values
Ps <- array(NA, dim = c(pdim, pdim, num))       # Ps=P_t^s
J <- array(NA, dim = c(pdim, pdim, num))        # J=J_t
alphas[, , num] <- kf$alphaf[, , num]           # starting value for a_T^s
Ps[, , num] <- kf$Pf[, , num]                   # starting value for prediction variance
```

And once it is initialised we could use another `for` loop, where we begin iterating from at the end of the sample and work our way to the start of the sample.

```
for (k in num:2)  {
  J[, , k - 1] <- (kf$Pf[, , k - 1] %*% t(Gt)) %*% solve(kf$Pp[, , k])
  alphas[, , k - 1] <- kf$alphaf[, , k - 1] + J[, , k - 1] %*% (alphas[, , k] - kf$alphap[, ,
         k])
  Ps[, , k - 1] <- kf$Pf[, , k - 1] + J[, , k - 1] %*% (Ps[, , k] - kf$Pp[, , k]) %*% t(J[, ,
         k - 1])
}
```

Since **R** can't count backward to zero we need to clean up a bit:

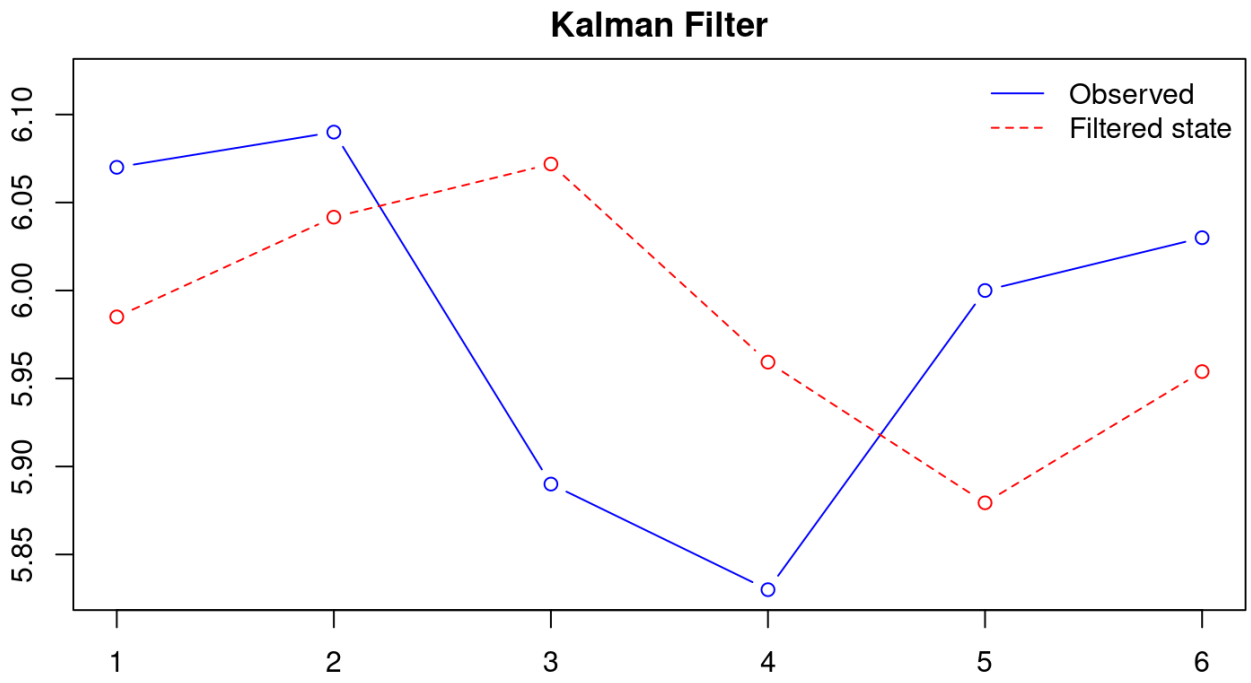
```
alpha00 <- alpha0
P00 <- sigma0
J0 <- as.matrix((P00 %*% t(Gt)) %*% solve(kf$Pp[, , 1]), nrow = pdim, ncol =
                pdim)
alpha0n <- as.matrix(alpha00 + J0 %*% (alphas[, , 1] - kf$alphap[, , 1]),
                     nrow = pdim,
                     ncol = 1)
P0n <- P00 + J0 %*% (Ps[, , k] - kf$Pp[, , k]) %*% t(J0)
```

Before we store the results in a particular object:

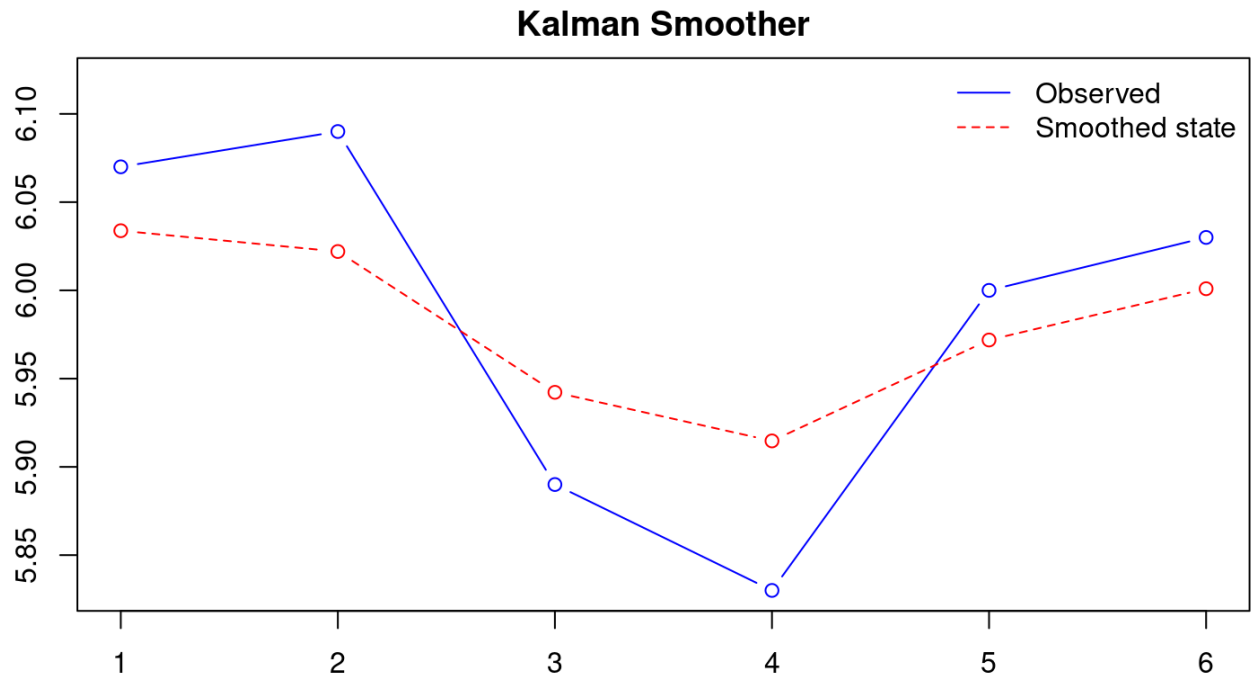
```
ks <- list(
  alphas = alphas,
  Ps = Ps,
  alpha0n = alpha0n,
  P0n = P0n,
  J0 = J0,
  J = J,
  alphap = kf$alphap,
  Pp = kf$Pp,
  alphaf = kf$alphaf,
  Pf = kf$Pf,
  like = kf$like,
  Kn = kf$K
)
```

Then finally, we could draw the graphs for the Kalman Filter and Kalman Smoother:

```
Time <- 1:num

par(mfrow = c(1, 1), mar = c(2, 5, 2, 1))
plot(
  Time,
  yt,
  main = "Kalman Filter",
  ylim = c(5.83, 6.12),
  xlab = "",
  ylab = "",
  col = "blue",
  type = 'b',
  lty = 1
)
lines(kf$alphap, lty = 2, type = 'b', col = "red")
legend("topright", legend =  c("Observed", "Filtered state"), col = c("blue", "red"),
       lty = c(1, 2), bty = "n")
```

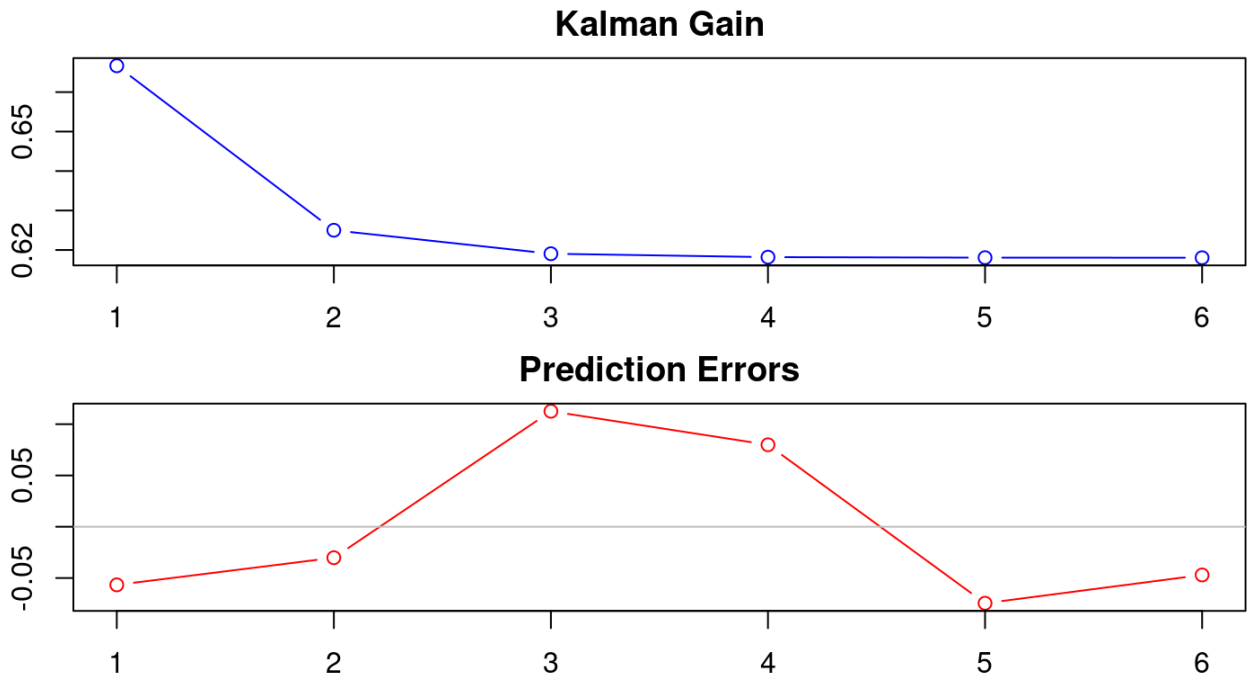



```
plot(
  Time,
  yt,
  main = "Kalman Smoother",
  ylim = c(5.83, 6.12),
  xlab = "",
  ylab = "",
  col = "blue",
  type = 'b',
  lty = 1
)
lines(ks$alphas, lty = 2, type = 'b', col = "red")
legend("topright", legend =  c("Observed", "Smoothed state"), col = c("blue", "red"),
       lty = c(1, 2), bty = "n")
```



Before we include the graphs for the Kalman gain and the prediction errors.

```
par(mfrow = c(2, 1), mar = c(2, 5, 2, 1))
pred.error <- ks$alphap[1, 1, ] - ks$alphaf[1, 1, ]
plot(Kmat,
     main = "Kalman Gain",
     xlab = "",
     ylab = "",
     col = "blue",
     type = 'b',
     lty = 1
)

plot(
  pred.error,
  main = "Prediction Errors",
  xlab = "",
  ylab = "",
  col = "red",
  type = 'b',
  lty = 1
)
abline(h=0, col="grey")
```



10 State-space modelling with Stan

As noted in the lecture, it may be preferable to model parameters in these models with Bayesian techniques, where one of the relatively new platforms for statistical modeling and high-performance statistical computation is called `Stan`. Should you wish, you can find out further details about this platform from the website: https://mc-stan.org/ (https://mc-stan.org/).

To install this software, you could use the following command:

```
install.packages('rstan', repos='https://cran.rstudio.com/', dependencies=TRUE)
install.packages('bayesplot', repos='https://cran.rstudio.com/', dependencies=TRUE)
```

And then once this is installed we could proceed as usual.

```
rm(list=ls())
graphics.off()

library(tsm)
library(rstan)
library(bayesplot)
library(tidyverse)
library(lubridate)
library(sarb2020q1)
```

To prepare the data for `Stan`, which makes use of C++ routines, we could make use of the following commands:

```
dat <- sarb_quarter %>%
  select(date, KBP6006L, KBP6006D) %>%
  mutate(defl = c(0, diff(log(KBP6006L/KBP6006D)*100, lag = 1))) %>%
  dplyr::filter(date > '1981-01-01')

standata <- within(list(), {
  y <- dat$defl
  n <- length(y)
})
```

The model file would then need to be written in C++ notation, where for the local level model, it would look as follows:

```
data {
  int<lower=1> n;
  vector[n] y;
}

parameters {
  vector[n] mu;
  real<lower=0> sigma_level;
  real<lower=0> sigma_irreg;
}

transformed parameters {
  vector[n] yhat;
  yhat = mu;
}

model {
  for(t in 2:n)
    mu[t] ~ normal(mu[t-1], sigma_level);
    y ~ normal(yhat, sigma_irreg);
}
```

You will note that in the `state-space-stan/mod` folder I have included code for a relatively large array of state-space models. The above code for the local level model is saved as `mod/llm.stan`. We can then read the model structure into **R**.

```
model_file <- 'state-space-stan/mod/llm.stan'
cat(paste(readLines(model_file)), sep = '\n')
```

Before we estimate the model parameters.

```
fit <- stan(file = model_file, data = standata,
            warmup = 4000, iter = 10000, chains = 2)
```

```
##
## SAMPLING FOR MODEL 'llm' NOW (CHAIN 1).
## Chain 1:
## Chain 1: Gradient evaluation took 1.9e-05 seconds
## Chain 1: 1000 transitions using 10 leapfrog steps per transition would take 0.19 seconds.
## Chain 1: Adjust your expectations accordingly!
## Chain 1:
## Chain 1:
## Chain 1: Iteration:    1 / 10000 [  0%]  (Warmup)
## Chain 1: Iteration: 1000 / 10000 [ 10%]  (Warmup)
## Chain 1: Iteration: 2000 / 10000 [ 20%]  (Warmup)
## Chain 1: Iteration: 3000 / 10000 [ 30%]  (Warmup)
## Chain 1: Iteration: 4000 / 10000 [ 40%]  (Warmup)
## Chain 1: Iteration: 4001 / 10000 [ 40%]  (Sampling)
## Chain 1: Iteration: 5000 / 10000 [ 50%]  (Sampling)
## Chain 1: Iteration: 6000 / 10000 [ 60%]  (Sampling)
## Chain 1: Iteration: 7000 / 10000 [ 70%]  (Sampling)
## Chain 1: Iteration: 8000 / 10000 [ 80%]  (Sampling)
## Chain 1: Iteration: 9000 / 10000 [ 90%]  (Sampling)
## Chain 1: Iteration: 10000 / 10000 [100%]  (Sampling)
## Chain 1:
## Chain 1:  Elapsed Time: 4.424 seconds (Warm-up)
## Chain 1:                6.9022 seconds (Sampling)
## Chain 1:                11.3262 seconds (Total)
## Chain 1:
##
## SAMPLING FOR MODEL 'llm' NOW (CHAIN 2).
## Chain 2:
## Chain 2: Gradient evaluation took 1.4e-05 seconds
## Chain 2: 1000 transitions using 10 leapfrog steps per transition would take 0.14 seconds.
## Chain 2: Adjust your expectations accordingly!
## Chain 2:
## Chain 2:
## Chain 2: Iteration:    1 / 10000 [  0%]  (Warmup)
## Chain 2: Iteration: 1000 / 10000 [ 10%]  (Warmup)
## Chain 2: Iteration: 2000 / 10000 [ 20%]  (Warmup)
## Chain 2: Iteration: 3000 / 10000 [ 30%]  (Warmup)
## Chain 2: Iteration: 4000 / 10000 [ 40%]  (Warmup)
## Chain 2: Iteration: 4001 / 10000 [ 40%]  (Sampling)
## Chain 2: Iteration: 5000 / 10000 [ 50%]  (Sampling)
## Chain 2: Iteration: 6000 / 10000 [ 60%]  (Sampling)
## Chain 2: Iteration: 7000 / 10000 [ 70%]  (Sampling)
## Chain 2: Iteration: 8000 / 10000 [ 80%]  (Sampling)
## Chain 2: Iteration: 9000 / 10000 [ 90%]  (Sampling)
## Chain 2: Iteration: 10000 / 10000 [100%]  (Sampling)
## Chain 2:
## Chain 2:  Elapsed Time: 4.4684 seconds (Warm-up)
## Chain 2:                7.8588 seconds (Sampling)
## Chain 2:                12.3272 seconds (Total)
## Chain 2:
```

To provide a summary of the results:

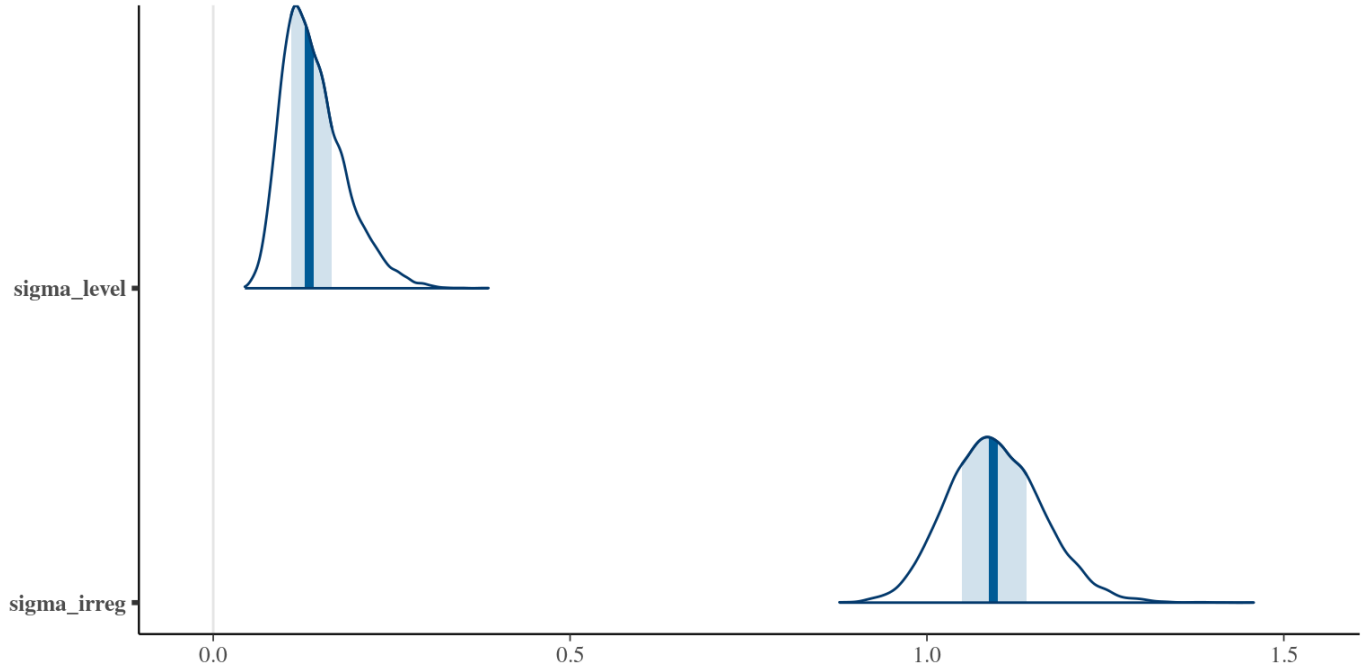
```
summary(fit)$summary %>%
  head()
```

```
##           mean     se_mean        sd     2.5%      25%
## mu[1] 3.069694 0.007673305 0.3954222 2.244882 2.816625
## mu[2] 3.106289 0.006424366 0.3681964 2.338208 2.867927
## mu[3] 3.158561 0.004627225 0.3417422 2.468480 2.936923
## mu[4] 3.219992 0.003312471 0.3247230 2.567534 3.009682
## mu[5] 3.288070 0.002943742 0.3158494 2.663255 3.078441
## mu[6] 3.337572 0.003038107 0.3084613 2.728944 3.132415
##            50%      75%    97.5%     n_eff     Rhat
## mu[1] 3.090182 3.333716 3.799223  2655.567 1.000693
## mu[2] 3.120377 3.353837 3.792727  3284.721 1.000644
## mu[3] 3.167655 3.392235 3.806572  5454.513 1.000461
## mu[4] 3.223694 3.439782 3.847547  9609.970 1.000079
## mu[5] 3.290372 3.497861 3.907508 11512.258 1.000022
## mu[6] 3.335345 3.541556 3.940858 10308.494 1.000373
```

```
mu <- get_posterior_mean(fit, par = 'mu')[, 'mean-all chains']
sigma_irreg <- get_posterior_mean(fit, par = 'sigma_irreg')[, 'mean-all chains']
sigma_level <- get_posterior_mean(fit, par = 'sigma_level')[, 'mean-all chains']
resids <- dat$defl - mu

print(fit, pars = c('sigma_level',
                    'sigma_irreg'))
```

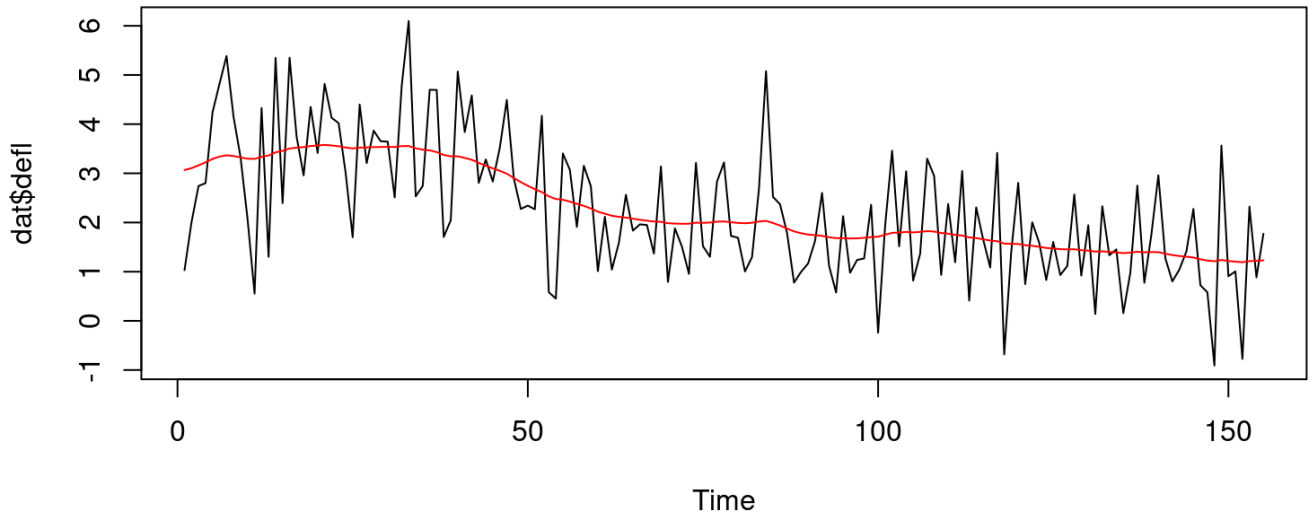
```
## Inference for Stan model: llm.
## 2 chains, each with iter=10000; warmup=4000; thin=1;
## post-warmup draws per chain=6000, total post-warmup draws=12000.
##
##             mean se_mean   sd 2.5%  25%  50%  75%
## sigma_level 0.14       0 0.04 0.07 0.11 0.13 0.16
## sigma_irreg 1.10       0 0.07 0.97 1.05 1.09 1.14
##             97.5% n_eff Rhat
## sigma_level  0.24   160 1.01
## sigma_irreg  1.23 11997 1.00
##
## Samples were drawn using NUTS(diag_e) at Thu Sep  3 18:18:45 2020.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

```
mcmc_areas(as.matrix(fit),
                     regex_pars = c("sigma_level", "sigma_irreg"))
```



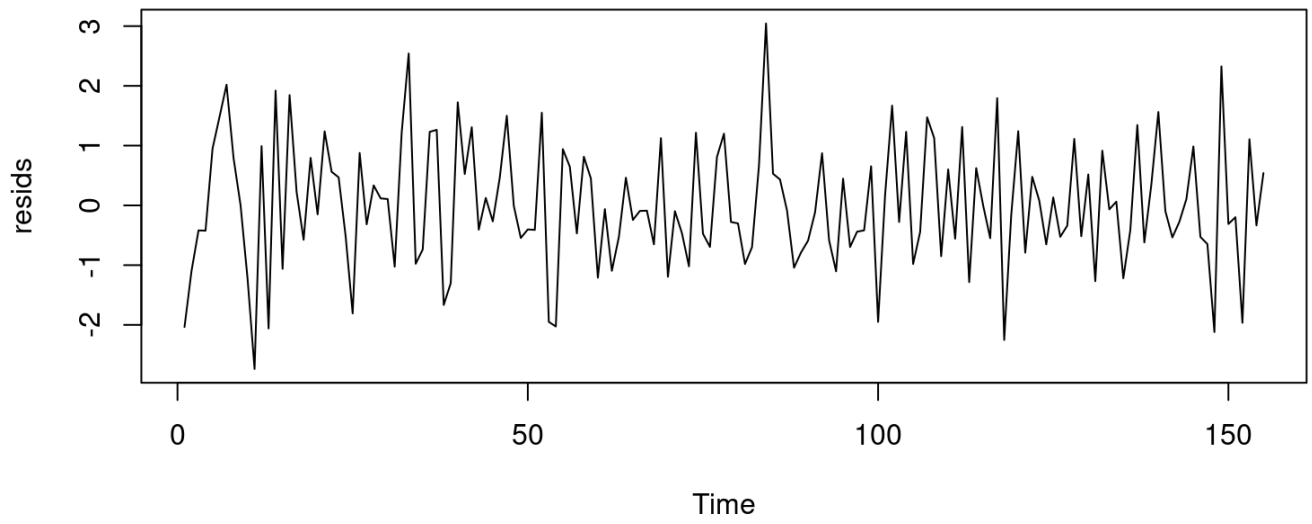
Thereafter we could plot the results for the estimated values for mu.

```
plot.ts(dat$defl)
lines(mu, col = "red")
```

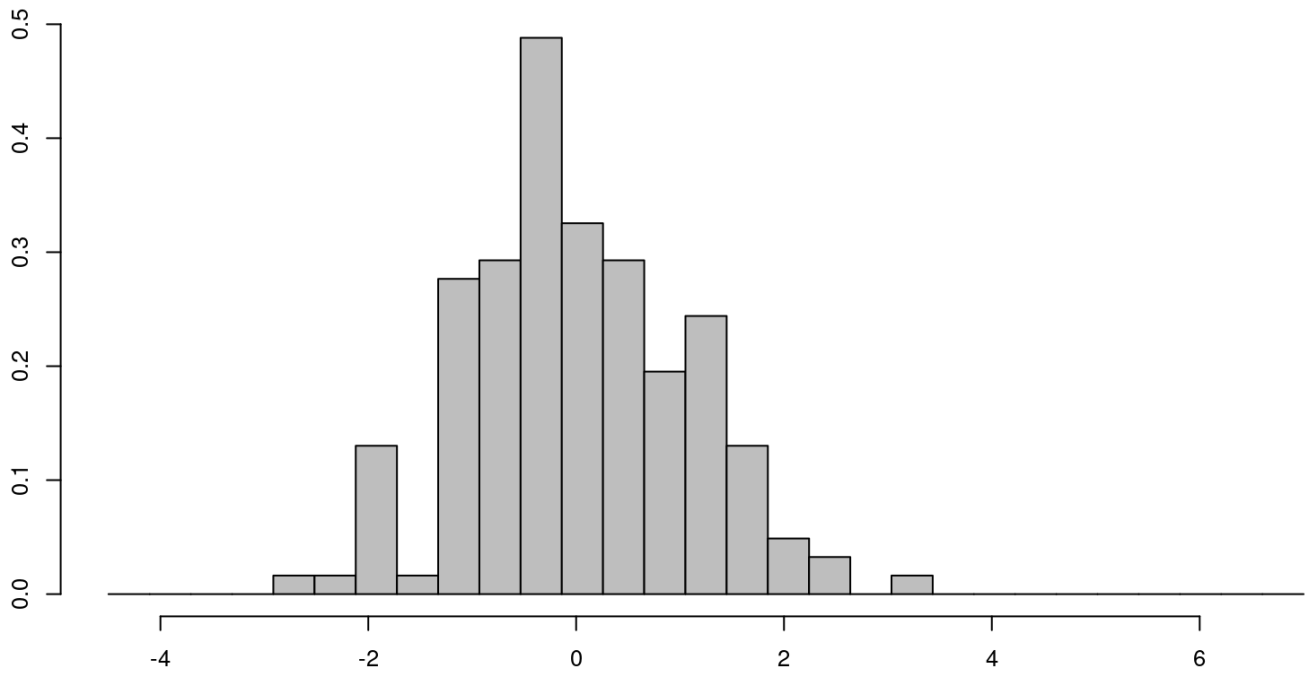


Before looking at the residuals in greater detail:

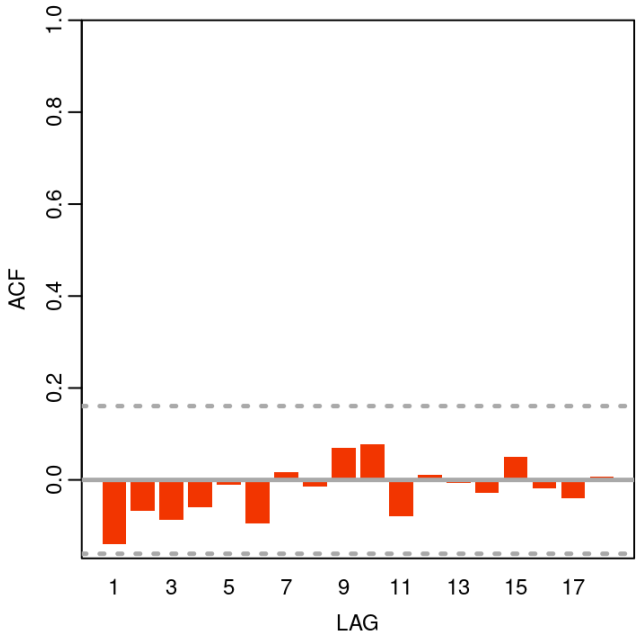
```
# plot results for residuals
plot.ts(resids)
```

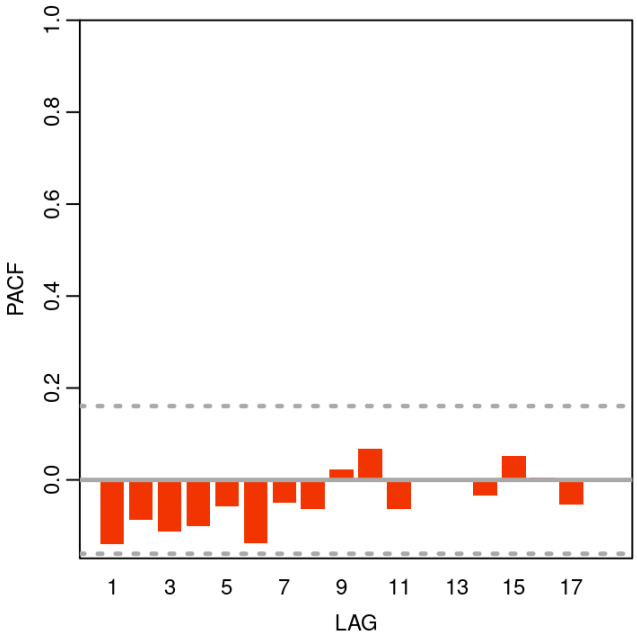



```
# Diagnostics
par(mfrow = c(1, 1), mar = c(2.2, 2.2, 1, 1), cex = 0.8)
hist(
  resids,
  prob = TRUE,
  col = "grey",
  main = "",
  breaks = seq(-4.5, 7, length.out = 30)
)
```




```
# acf
ac(resids)
```


ACF
1.0
0.8
0.6
0.4
0.2
0.0
1
3
5
7
9
11
13
15
17
LAG


PACF
1.0
0.8
0.6
0.4
0.2
0.0
1
3
5
7
9
11
13
15
17
LAG