State Space Recipes in R

Paul Teetor

2019-12-23

Contents

Chapter 1

Introduction

This monograph is a collection of recipes for creating state-space models in R. I like the power of state-space models, and R has several excellent packages for building them. Unfortunately, it's not quite an "out of the box" technology. Using any package involves numerous little details, and unless I used the package very recently, building a model requires pulling out the package documentation, reading it all over again, and trying to remember how the parts fit together. One day I got tired of that, so I put together these recipes.

This is not a tutorial for state-space models. For a general introduction to state-space modeling, I recommend the book by Commandeur and Koopman[1].

In these notes, I use the `StructTS` function to create the simpler models, and I use the `dlm` package for more complicated models. There isn't room here to cover other R packages. If you're interested in a survey of state-space packaqes for R, I recommend the excellent review by Tusell[2].

1.1 The StructTS function

R includes a function, `StructTS`, which can quickly and easily estimate the parameters of simple state-space models such as the *local level* model or the *local linear trend* model.[3]

`StructTS` is one function in a group of functions which, together, provide many features of state-space modeling.

[1] Commandeur and Koopman (2007). *An Introduction to State Space Time Series Analysis*, Oxford University Press (ISBN 978-0-19-922887-4)

[2] Tusell (2011). "Kalman Filtering in R", *Journal of Statistical Software* (http://www.jstatsoft.org/v39/i02/paper)

[3] Ripley (2002). "Time Series in R 1.5.0", *R News* (http://cran.r-project.org/doc/Rnews/Rnews_2002-2.pdf)

Function	Purpose
StructTS	Estimate parameters of a simple state-space model
tsdiag	Plot diagnostics for state-space model
KalmanLike	Calculate parameters' log-likelihood (Gaussian model)
KalmanRun	Filter time series data
tsSmooth	Smooth time series data (calls KalmanSmooth)
KalmanForecast	Forecast time series points from model
makeARIMA	Create state-space model equivalent to ARIMA model

1.2 The dlm package

For the advanced recipes, I use the `dlm` package originally created by Giovanni Petris.[4] The package is very well documented, and Petris has even written a book regarding state-space models in general and the `dlm` package in particular.[5] There is also an overview written by Petris and Petrone[6] which discusses several R packages with an emphasis on the `dlm` package.

The package contains many useful functions. This monograph uses these.

Function	Purpose
dlmModPoly	Construct polynomial model
dlmModReg	Construct regression model
dlmMLE	Estimate maximum likelihood parameters of model
dlmFilter	Filter a time series
dlmSmooth	Smooth a time series
dlmBSample	Draw from the posterior distribution

The package includes a very cool feature, which is the ability to "add" models together into a compound model. That feature is not illustrated here, but I urge any serious user to study the feature. It would let you, say, easily combine a regression model with an ARMA model to create a better model your data.

[4] Petris (2010). "An R Package for Dynamic Linear Models", *Journal of Statistical Software* (http://www.jstatsoft.org/v36/i12/paper)

[5] Petris, Petrone, and Campagnoli (2009). *Dynamic Linear Models with R*, Springer (ISBN 978-0-387-77237-0)

[6] Petris and Petrone (2011). "State Space Models in R", *Journal of Statistical Software* (http://www.jstatsoft.org/v41/i04/paper)

1.3 The examples

Many recipes includes an example. The examples are intended to be fully stand-alone, meaning you can cut and paste them directly into R and watch them run.

All examples use some concrete dataset, typically the Nile River data included with R. The recipes start by assigning the time series data to variable y, like this.

```
y <- datasets::Nile
```

The subsequent code is written in terms of y, not a specific dataset. My goal was to let you copy the recipe, substitute your data for the Nile River data, and try the recipe for yourself.

1.4 Online materials

R code examples are available on a public Github repository.

> https://github.com/pteetor/StateSpaceRecipes

Chapter 2

Random Walk Model

The *random walk* model is so simple that it's barely a model at all.

$$y_t = y_{t-1} + \epsilon_t, \qquad \epsilon_t \sim N(0, \sigma_\epsilon^2)$$

This says, "Today is like yesterday, only different." Nonetheless, I find the model useful for exploring new time series data. It answers the first basic question, how noisy is the data?

To estimate the model, we first expand the definition into the state-space framework expected by the software.

$$\begin{aligned} y_t &= \mu_t \\ \mu_t &= \mu_{t-1} + \xi_t, \qquad \xi_t \sim N(0, \sigma_\xi^2) \end{aligned}$$

Notice that there is no error term in the first equation. When we observe y_t, it's an uncorrupted copy of μ_t.

The model has two parameters.

σ_ξ^2	Variance of the observational errors, ξ_t
μ_0	Initial level of μ

The R software always assumes that y has an error term. We get around that by forcing its variance to be zero, effectively eliminating it.

2.1 Fitting a Random Walk Model

Problem

You want to fit your time series data to a random walk model.

Solution

You can fit a random walk using the `StructTS` function. Fit the data to a local level model while forcing the observational variance to be zero.

```
model <- StructTS(y, type="level", fixed=c(0, NA))
```

Example

```
y <- datasets::Nile

model <- StructTS(y, type="level", fixed=c(0, NA))
print(model)
```

```
##
## Call:
## StructTS(x = y, type = "level", fixed = c(0, NA))
##
## Variances:
##   level  epsilon
##       0    28638
```

Discussion

Alt. Solution

We can also fit a random walk using the dlm package. The code for estimating parameters is very similar to the code for the local level model. The difference is that we force V, the variance of the observations, to be zero.

We define a function, `buildRandomWalk`, that builds a `dlm` model object from two input parameters, `dW` and `m0`. The parameters are packed into a single, 2-element vector.

```
buildRandomWalk <- function(v) {
  dW <- exp(v[1])
  m0 <- v[2]
```

```
  dlmModPoly(order=1, dV=0, dW=dW, m0=m0)
}
```

The function calls the `dlmModPoly` function from `dlm` to create the model object.

We need initial guesses for the model parameters.

```
varGuess <- var(diff(y), na.rm=TRUE)
mu0Guess <- as.numeric(y[1])
```

Next we call the `dlmMLE` function to estimate the MLE parameters using numerical optimzation. Always check for convergence.

```
parm <- c(log(varGuess), mu0Guess)
mle <- dlmMLE(y, parm=parm, buildRandomWalk)
if (mle$convergence != 0) stop(mle$message)
```

From the MLE estimates, we can build the final `dlm` model.

```
model <- buildRandomWalk(mle$par)
```

We can extract the estimated parameters from `model`, the returned object.

`model$W`	Variance of the random walk errors
`model$m0`	Initial level

Alt. Example

```
library(dlm)

y <- datasets::Nile

buildRandomWalk <- function(v) {
  dW <- exp(v[1])
  m0 <- v[2]
  dlmModPoly(order=1, dV=0, dW=dW, m0=m0)
}

varGuess <- var(diff(y), na.rm=TRUE)
mu0Guess <- as.numeric(y[1])

parm <- c(log(varGuess), mu0Guess)
mle <- dlmMLE(y, parm=parm, buildRandomWalk)
if (mle$convergence != 0) stop("Optimizer did not converge")

model <- buildRandomWalk(mle$par)
```

```r
cat("Transitional variance:", model$W, "\n",
    "Initial level:", model$m0, "\n")
```

```
## Transitional variance: 27996.75
##  Initial level: 1120
```

See Also

2.2 Diagnosing a Random Walk Model

Problem

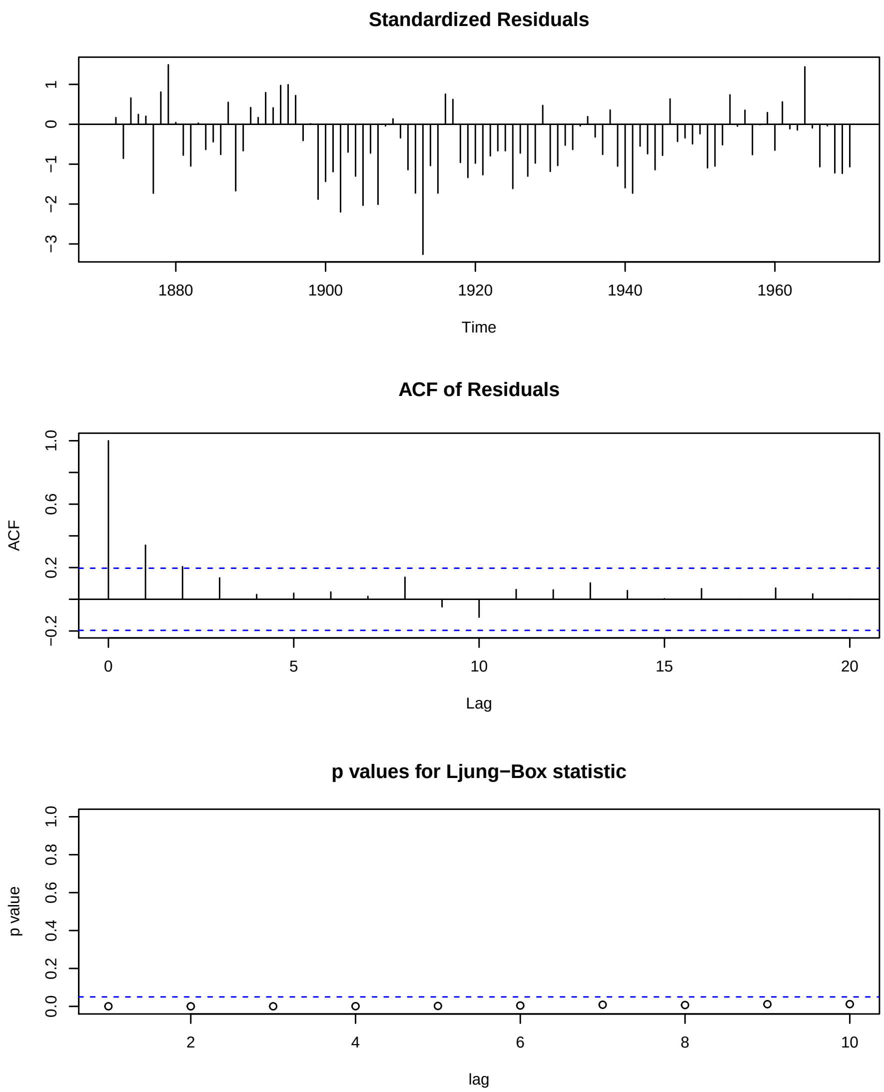
Solution

The `tsdiag` function creates diagnostic plots for models created using the `StructTS` function.

```r
tsdiag(model)
```

Example

```r
y <- datasets::Nile

model <- StructTS(y, type="level", fixed=c(0, NA))
tsdiag(model)
```


Standardized Residuals
1
0
−1
−2
−3
1880
1900
1920
1940
1960
Time
ACF of Residuals
ACF
1.0
0.6
0.2
−0.2
0
5
10
15
20
Lag
p values for Ljung−Box statistic
p value
1.0
0.8
0.6
0.4
0.2
0.0
2
4
6
8
10
lag

Discussion

Alt. Solution

See Also

2.3 Smoothing With a Random Walk Model

2.4 Filtering With a Random Walk Model

2.5 Plotting Filtered or Smoothed Results with a Random Walk Model

2.6 Forecasting with a Random Walk Model

2.7 Plotting Forecasted Values with a Random Walk Model

Chapter 3

The Local Level Model

3.1 Fitting a Local Level Model

The *local level model* assumes that we observe a time series, y_t, and that time series is the sum of another time series, μ_t, and random, corrupting noise, ϵ_t. We would prefer to directly observe μ_t, a *latent* variable, but cannot due to the noise.

$$\begin{aligned} y_t &= \mu_t + \epsilon_t, \qquad \epsilon_t \sim N(0, \sigma_\epsilon^2) \\ \mu_t &= \mu_{t-1} + \xi_t, \qquad \xi_t \sim N(0, \sigma_\xi^2) \end{aligned}$$

In this model, the μ_t follow a random walk, so this is sometimes called the *random walk with noise* model. (The `dlm` package uses that name.)

The model has only three parameters.

σ_ϵ^2	Variance of the observation errors
σ_ξ^2	Variance of the state transitions
μ_0	Initial level of μ.

Problem

You want to fit your time series data to a local level model.

Solution

The `StructTS` function can estimate the parameters of a local level model by setting `type="level"`. (Here, I assume your time series data is y.)

```
struct <- StructTS(y, type="level")
```

The function returns a list that includes these elements.

`struct$coef`	2-element vector of estimated variances, labeled `level` and `epsilon`
`struct$model0`	Initial state; in particular `model0$a` is the initial level
`struct$model`	Final model
`struct$code`	Convergence code from optimizer, zero is good, non-zero is bad

Example

This example constructs a local level model for the Nile data.

```
y <- datasets::Nile

struct <- StructTS(y, type="level")
if (struct$code != 0) stop("optimizer did not converge")

print(struct$coef)
```

```
##     level   epsilon
##  1469.147 15098.577
```

```
cat("Transitional variance:", struct$coef["level"], "\n",
    "Observational variance:", struct$coef["epsilon"], "\n",
    "Initial level:", struct$model0$a, "\n")
```

```
## Transitional variance: 1469.147
##  Observational variance: 15098.58
##  Initial level: 1120
```

Discussion

Alt. Solution

The solution, above, uses the `StructTS` function because that's the easiest way to estimate the model parameters. Sometimes, however, you might want to use the `dlm` package instead, even though it's a bit more work. Why would one do that? The local level model might be your first step in model building, leading to more complicate models. Or you might want to bootstrap your model, which is more easily done using `dlm`. Or you might want to combine a local level model with another model using the model "addition" feature of `dlm`.

The `dlm` authors refer to the local level model as the *random walk with noise* model: the underlying level follows a random walk, and our observation of it is polluted by noise.

Mathematically, the local level models used by the `StructTS` function and the `dlm` package are the same, but they use different variable names and slightly different notational conventions.

$$
\begin{aligned}
Y_t &= \mu_t + v_t, \qquad v_t \sim N(0, V) \\
\mu_t &= \mu_{t-1} + w_t, \qquad w_t \sim N(0, W)
\end{aligned}
$$

Under these conventions, we observe Y_t (not y_t), and the variances of the error terms are generalized to be matrices V and W.

(*Move to footnote:* Generalizing V and W to matrices will open the door to the multivariate case.)

Following those conventions, the model has these three parameters.

`dV`	Variance of the observation errors
`dW`	Variance of the transition errors
`m0`	The initial value (μ_0)

The R code begins by defining the `buildModPoly1` function which can create the needed `dlm` model object from three parameters.

```
buildModPoly1 <- function(v) {
  dV <- exp(v[1])
  dW <- exp(v[2])
  m0 <- v[3]
  dlmModPoly(1, dV=dV, dW=dW, m0=m0)
}
```

The R function itself takes one parameter, a 3-element vector, into which the model parameters are packed. The first two parameters are log-variance, not variance, to prevent the optimizer from exploring negative values for variance.

To start, we need some reasonable guesses at the parameters. They don't need to be perfect, but being in the right ballpark is useful.

```
varGuess <- var(diff(y), na.rm=TRUE)
mu0Guess <- as.numeric(y[1])
```

The `dlmMLE` function finds the maximum likelihood estimate of the parameters, starting with our reasonable guesses and repeatedly calling our `buildModPoly1` until it converges on the MLE solution. Always check for convergence.

```
parm <- c(log(varGuess), log(varGuess), mu0Guess)
mle <- dlmMLE(y, parm=parm, buildModPoly1)

if (mle$convergence != 0) stop(mle$message)
```

From the MLE parameter estimates, we can build the final model.

```
model <- buildModPoly1(mle$par)
```

3.1.1 Alt. Example

```
library(dlm)

y <- datasets::Nile

buildModPoly1 <- function(v) {
  dV <- exp(v[1])
  dW <- exp(v[2])
  m0 <- v[3]
  dlmModPoly(1, dV=dV, dW=dW, m0=m0)
}

varGuess <- var(diff(y), na.rm=TRUE)
mu0Guess <- as.numeric(y[1])

parm <- c(log(varGuess), log(varGuess), mu0Guess)

mle <- dlmMLE(y, parm=parm, buildModPoly1)
if (mle$convergence != 0) stop(mle$message)

model <- buildModPoly1(mle$par)

cat("Observational variance:", model$V, "\n",
    "Transitional variance:", model$W, "\n",
    "Initial level:", model$m0, "\n")
```

```
## Observational variance: 15098.68
##  Transitional variance: 1469.053
##  Initial level: 1120
```

See Also

3.2 Smoothing With a Local Level Model

Problem

Solution

Example

Discusion

Alt. Solution

See Also

3.3 Filtering With a Local Level Model

3.4 Diagnosing a Local Level Model

3.5 Plotting a Local Level Model

Chapter 4

The Local Linear Trend Model

The local linear trend model builds in the local level model, adding a time-varying trend, ν_t, that follows a random walk. As before, we observe y, which is the underlying level plus noise.

$$
\begin{aligned}
y_t &= \mu_t + \epsilon_t, \qquad \epsilon_t \sim N(0, \sigma^2_\epsilon) \\
\mu_t &= \mu_{t-1} + \nu_{t-1} + \xi_t, \qquad \xi_t \sim N(0, \sigma^2_\xi) \\
\nu_t &= \nu_{t-1} + \zeta_t, \qquad \zeta_t \sim N(0, \sigma^2_\zeta)
\end{aligned}
$$

This model has five parameters.

σ^2_ϵ	Variance of observation errors, ϵ
σ^2_ξ	Variance of transition errors, ξ
σ^2_ζ	Variance of transition errors, ζ
μ_0	Initial level of μ
λ_0	Initial level of λ

4.1 Fitting a Local Linear Trend Model

Problem

You want to fit your time series data to a local linear trend model.

Solution

Estimate the parameters by calling `StructTS` with `type="trend"`.

```
struct <- StructTS(y, type="trend")
if (struct$code != 0) stop("optimizer did not converge")
```

`StructTS` returns a list that contains these elements, among others.

`struct$coef`	Vector of estimated parameters
`struct$model0`	List of initial state and levels

Example

This code constructs a local linear trend model for the Nile River data.

```
y <- datasets::Nile

struct <- StructTS(y, type="trend")
if (struct$code != 0) stop("optimizer did not converge")

print(struct$coef)
```

```
##     level     slope   epsilon
##  1426.736     0.000 15047.326
```

```
cat("Transitional variance:", struct$coef["level"], "\n",
    "Slope variance:", struct$coef["slope"], "\n",
    "Observational variance:", struct$coef["epsilon"], "\n",
    "Initial level of mu:", struct$model0$a[1], "\n",
    "Initial level of lambda:", struct$model0$a[2], "\n" )
```

```
## Transitional variance: 1426.736
##  Slope variance: 0
##  Observational variance: 15047.33
##  Initial level of mu: 1120
##  Initial level of lambda: 0
```

Oh darn. The slope component's variance is zero, indicating that the slope is best held constant. We can conclude that the local linear trend model is overkill and the simpler local level model is sufficient. That makes for a lousy example, but its a good reminder so check and interpret the MLE parameters carefully. They might be telling you a story.

Discusion

Alt. Solution

This Solution uses the `dlm` package.

The `dlm` documentation refers to this as the *linear growth* model.

The `dlm` code for estimating a local linear trend model begins by defining a function capable of creating the appropriate *dlm* model object from five parameters.

```
buildModPoly2 <- function(v) {
  dV <- exp(v[1])
  dW <- exp(v[2:3])
  m0 <- v[4:5]
  dlmModPoly(order=2, dV=dV, dW=dW, m0=m0)
}
```

Notice that the five model parameters are packed into one 5-element R vector.

The `dlmMLE` uses our `buildModPoly2` function to find the maximum likelihood estimates (MLE) of the parameters. It uses numerical optimization, so always check for convergence.

```
varGuess <- var(diff(y), na.rm=TRUE)
mu0Guess <- as.numeric(y[1])
lambda0Guess <- 0.0

parm <- c(log(varGuess), log(varGuess), log(varGuess),
          mu0Guess, lambda0Guess)
mle <- dlmMLE(y, parm=parm, buildModPoly2)

if (mle$convergence != 0) stop(mle$message)
```

From the MLE parameters, we can construct the final model object.

```
model <- buildModPoly2(mle$par)
```

The `model` object contains the estimated parameters (among other things).

`V`	Variance of the observations (scalar)
`W`	Variance of the state variables' error terms (matrix)
`m0`	Initial values of the state variables (vector)

Alt. Example

```
library(dlm)

y <- datasets::Nile

buildModPoly2 <- function(v) {
  dV <- exp(v[1])
  dW <- exp(v[2:3])
  m0 <- v[4:5]
  dlmModPoly(order=2, dV=dV, dW=dW, m0=m0)
}

varGuess <- var(diff(y), na.rm=TRUE)
mu0Guess <- as.numeric(y[1])
lambda0Guess <- 0.0

parm <- c(log(varGuess), log(varGuess), log(varGuess),
          mu0Guess, lambda0Guess)
mle <- dlmMLE(y, parm=parm, buildModPoly2)

if (mle$convergence != 0) stop(mle$message)

model <- buildModPoly2(mle$par)
```

See Also

4.2 Diagnosing a Local Linear Trend Model

Problem

After fitting a local linear trend model using `StructTS`, you want to assess the quality of the model.

Solution

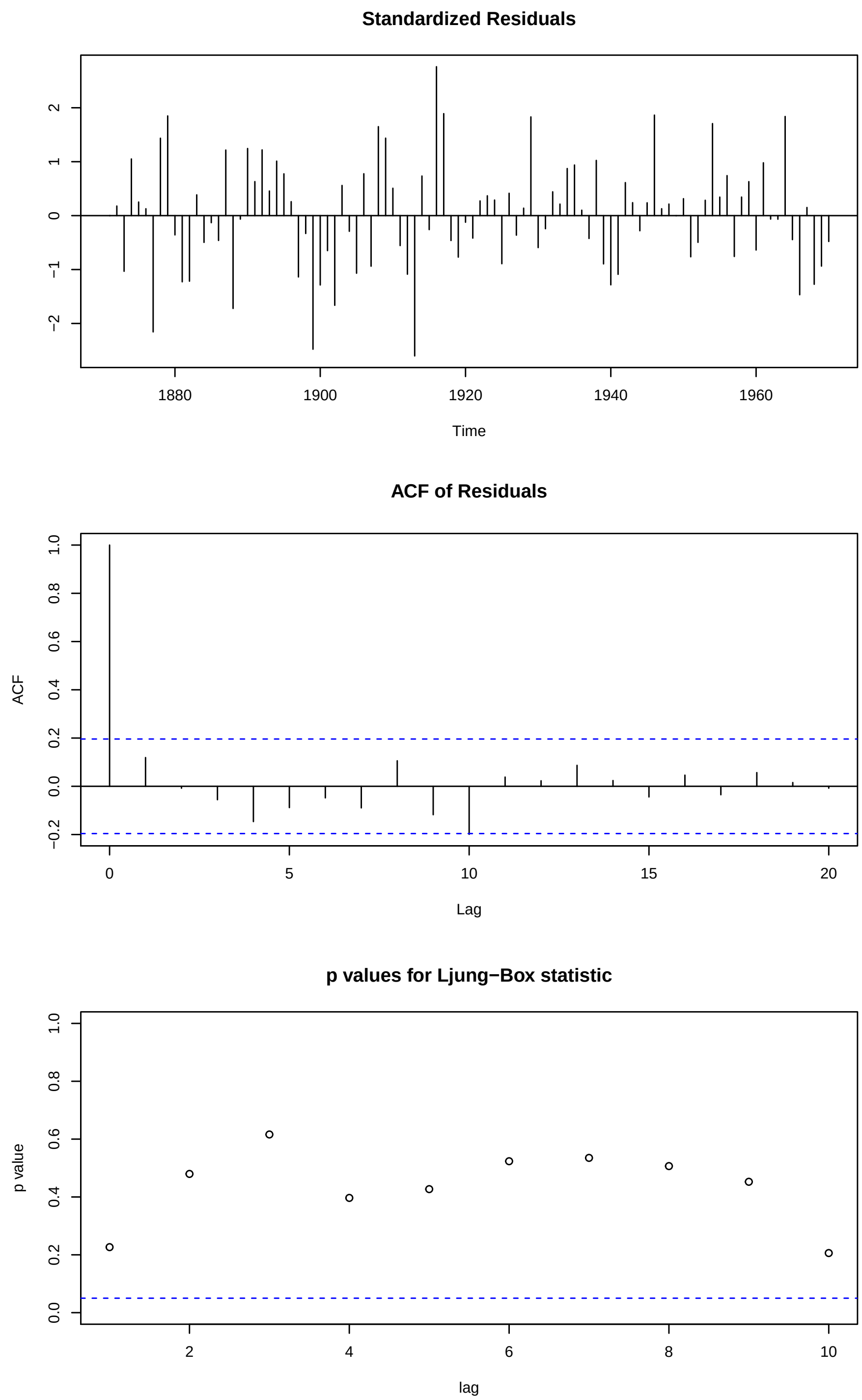
The `tsdiag` function produces plots that are useful for evaluating your StructTS model.

```
tsdiag(struct)
```

Example

This code constructs a local linear trend model for the Nile data, then produces diagnostics plots.

```
y <- datasets::Nile

struct <- StructTS(y, type="trend")
if (struct$code != 0) stop("optimizer did not converge")

tsdiag(struct)
```


Standardized Residuals
2
1
0
−1
−2
1880
1900
1920
1940
1960
Time
ACF of Residuals
ACF
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
0
5
10
15
20
Lag
p values for Ljung−Box statistic
p value
1.0
0.8
0.6
0.4
0.2
0.0
2
4
6
8
10
lag

Discusion

Alt. Solution

*This Solution uses the `dlm` package.‘

The `tsdiag` function is a generic function for diagnosing time series models, and the `dlm` package has an implementation. It produces useful plots for identifying problems in your model.

The diagnostics are based on the posterior distribution defined by the model, so call `dlmFilter` first to construct the posterior, then apply `tsdiag` to the result.

```
filt <- dlmFilter(y, model)
tsdiag(filt)
```

Alt. Example

See Also

4.3 Smoothing With a Local Linear Trend Model

Problem

After building a local linear trend model using `StructTS`, you want to smooth the data; that is, remove the noise component.

Solution

The `tsSmooth` function can smooth your data. based on a state-space model created by `StructTS`.

```
smoothed <- tsSmooth(struct)
```

Example

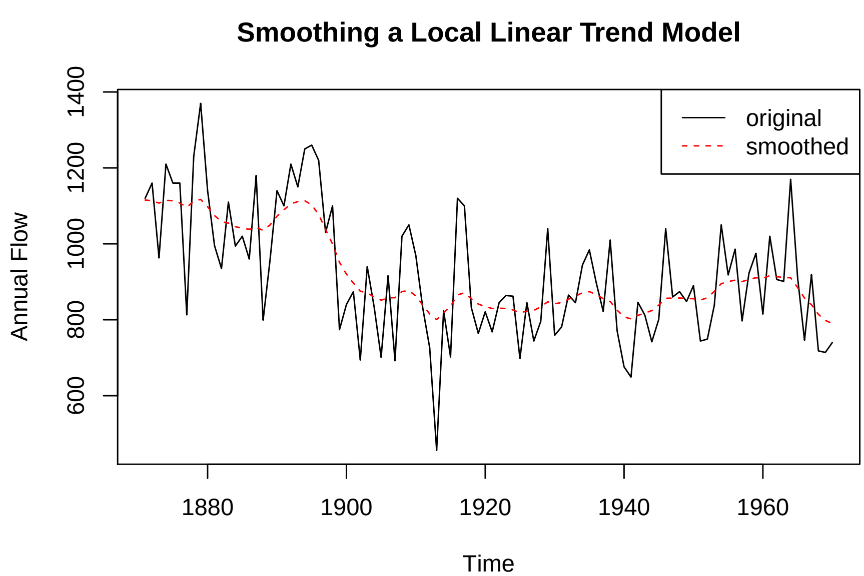
This code estimates a local linear trend model for the Nile data, constructs the smoothed time series, and dumps the result.

```
y <- datasets::Nile

struct <- StructTS(y, type="trend")
```

```
if (struct$code != 0) stop("optimizer did not converge")

smoothed <- tsSmooth(struct)
str(smoothed)
```

```
##  Time-Series [1:100, 1:2] from 1871 to 1970: 1115 1114 1107 1115 1113 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr [1:2] "level" "slope"
```

A plot below illustrates the effect of smoothing based on a local linear trend model of the Nile River data.



Discusion

Alt. Solution

*This solution uses the `dlm` package.‘

The `dlm` package provides a function, `dlmSmooth`, for smoothing your data based on a model. If y is your data and `model` is any model created by `dlm`, such as the recipes in this monograph, then this call will compute the smoothed data.

```
smooth <- dlmSmooth(y, model)
## smooth$s contains the smoothed values
```

See Also

For an example of diagnosing a model built with the `dlm` package, see Diagnosing a Regression Model, Fixed Coefficients.

For an example of smoothing with the `dlm` package, see Smoothing With a Regression Model, Fixed Coefficients.

4.4 Filtering With a Local Linear Trend Model

Problem

Solution

Example

Discusion

Alt. Solution

Alt. Example

See Also

4.5 Plotting a Local Linear Trend Model

Chapter 5

Regression Model, Fixed Coefficients

5.1 Fitting a Regression Model, Fixed Coefficients

5.2 Diagnosing a Regression Model, Fixed Coefficients

Problem

Solution

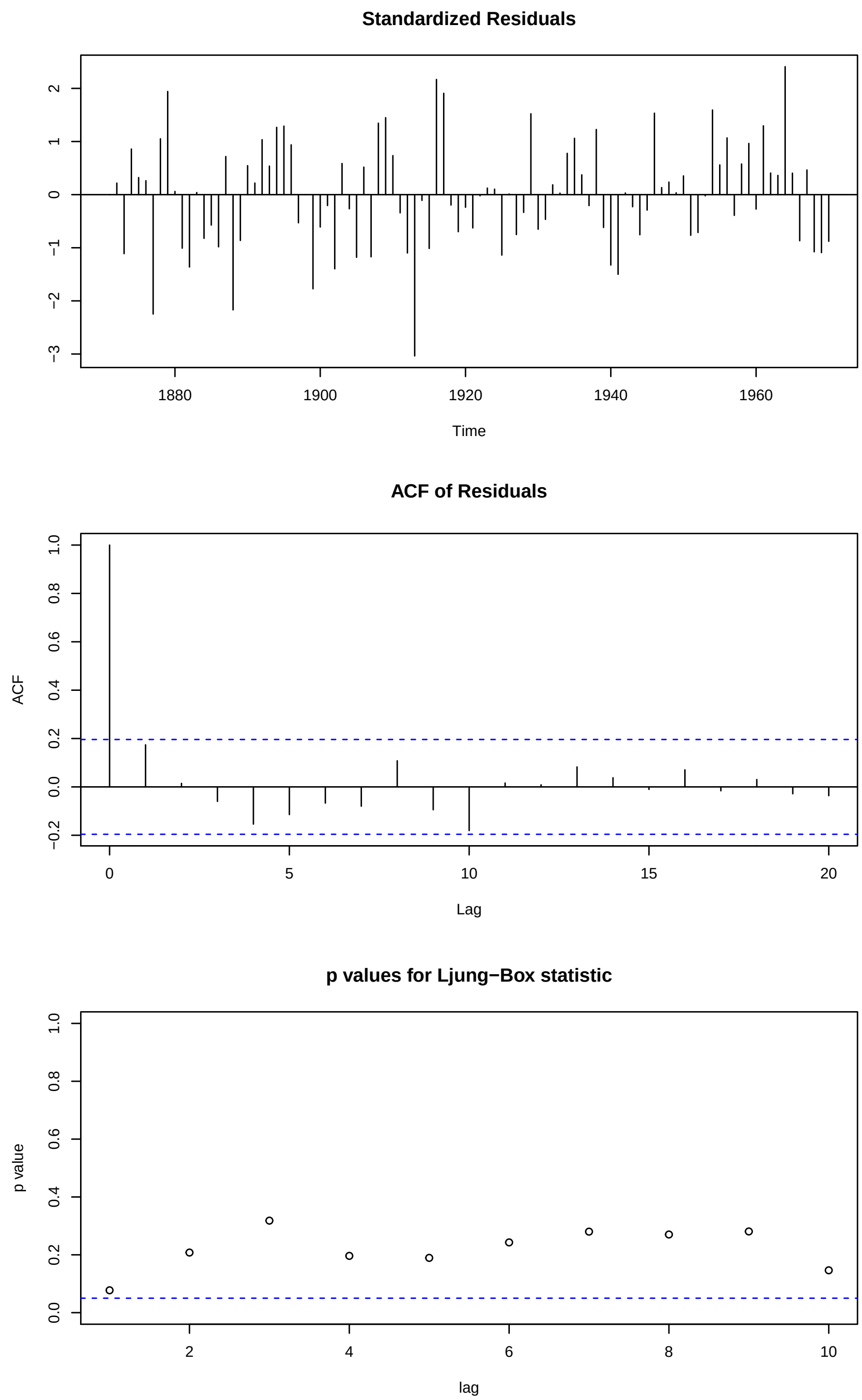
Example

This code assumes that `model` was fit by the recipe, above, for estimating a regression with fixed coefficients.

(*Move to footnote:* The code also assumes that `x` and `y` are the regressor and time series data, respectively, as in that recipe.)

It produces the diagnostic plots for the model.

```
filt <- dlmFilter(y, model)
tsdiag(filt,
       main="Diagnostics for Regression Model" )
```


Standardized Residuals
Time
1880
1900
1920
1940
1960
ACF of Residuals
ACF
Lag
p values for Ljung−Box statistic
p value
lag

Discusion

See Also

5.3 Smoothing With a Regression Model, Fixed Coefficients

Problem

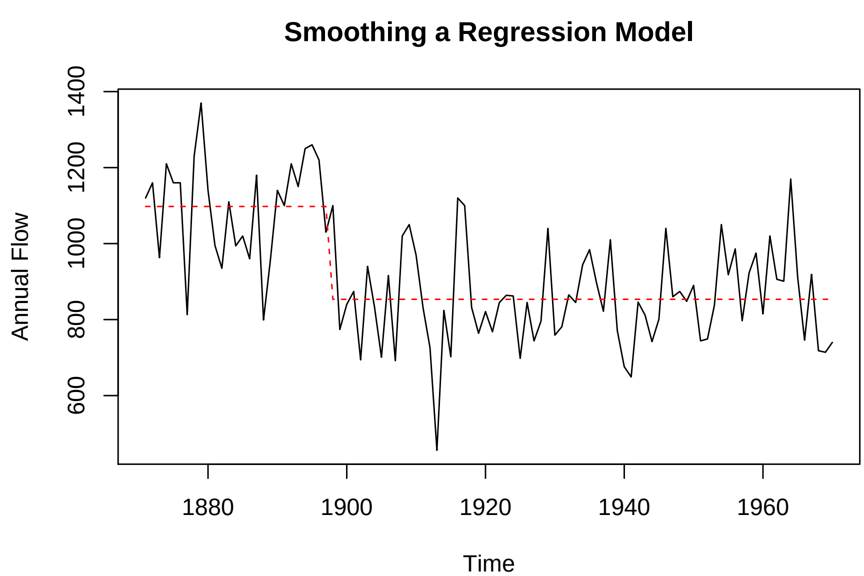
Solution

Example

This example assumes that `model` was created by the example, above, for estimating a regression with fixed coefficients.

(*Move to footnote:* The example code also assumes that `x` and `y` are the predictor and the time series data, respectively, from that recipe.)

It smooths the original data based on that model, then plots both the data and smoothed values.



Discusion

See Also

5.4 Filtering With a Regression Model, Fixed Coefficients

5.5 Plotting a Regression Model, Fixed Coefficients

Chapter 6

Regression Model, Time-Varying Coefficients

6.1 Fitting a Regression Model, Time-Varying Coefficients

6.2 Smoothing With a Regression Model, Time-Varying Coefficients

6.3 Filtering With a Regression Model, Time-Varying Coefficients

6.4 Diagnosing a Regression Model, Time-Varying Coefficients

6.5 Plotting a Regression Model, Time-Varying Coefficients

Chapter 7

Boostrapping a State-Space Model

Chapter 8

ARMA Models

8.1 Fitting an ARMA Model

8.2 Diagnosing an ARMA Model

8.3 Smoothing With an ARMA Model

8.4 Filtering With an ARMA Model

8.5 Plotting Filtered or Smoothed Results with an ARMA Model

8.6 Forecasting with an ARMA Model

8.7 Plotting Forecasted Values with an ARMA Model

References

Chapter 9

TEMPLATE CHAPTER

You can label chapter and section titles using `{#label}` after them, e.g., we can reference Chapter 1. If you do not manually label them, there will be automatic labels anyway, e.g., Chapter 1.

Figures and tables with captions will be placed in `figure` and `table` environments, respectively.

```
par(mar = c(4, 4, .1, .1))
plot(pressure, type = 'b', pch = 19)
```

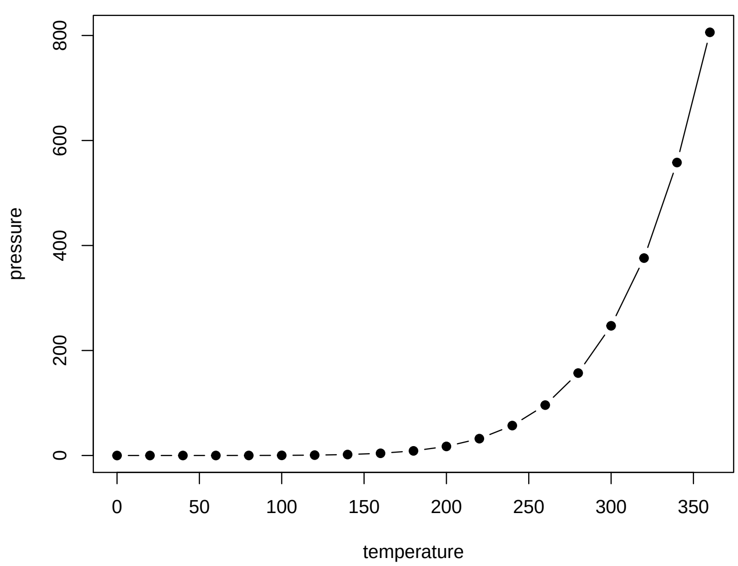


Figure 9.1: Here is a TEMPLATE figure!

Table 9.1: Here is a TEMPLATE table!

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa
5.4	3.7	1.5	0.2	setosa
4.8	3.4	1.6	0.2	setosa
4.8	3.0	1.4	0.1	setosa
4.3	3.0	1.1	0.1	setosa
5.8	4.0	1.2	0.2	setosa
5.7	4.4	1.5	0.4	setosa
5.4	3.9	1.3	0.4	setosa
5.1	3.5	1.4	0.3	setosa
5.7	3.8	1.7	0.3	setosa
5.1	3.8	1.5	0.3	setosa

Reference a figure by its code chunk label with the `fig:` prefix, e.g., see Figure 9.1. Similarly, you can reference tables generated from `knitr::kable()`, e.g., see Table 9.1.

```
knitr::kable(
  head(iris, 20), caption = 'Here is a TEMPLATE table!',
  booktabs = TRUE
)
```

You can write citations, too. For example, we are using the **bookdown** package (Xie, 2019) in this sample book, which was built on top of R Markdown and **knitr** (Xie, 2015).

9.1 Fitting an XXX Model

9.2 Diagnosing an XXX Model

9.3 Smoothing With an XXX Model

9.4 Filtering With an XXX Model

9.5 Plotting Filtered or Smoothed Results with an XXX Model

9.6 Forecasting with an XXX Model

9.7 Plotting Forecasted Values with an XXX Model

Bibliography

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2019). *bookdown: Authoring Books and Technical Documents with R Markdown.* R package version 0.16.