Recipes for State Space Models in R

Paul Teetor

July 2015

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 had 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].

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

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].

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

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]

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

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

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

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.

[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`)

The package contains many useful functions. This monograph uses .

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.

The examples

Every recipe includes an example. Many 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. They 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, easily substitute your data for the Nile River data, and try the recipe for yourself.

Online materials

R code examples are available on a public Github repository.

`https://github.com/pteetor/StateSpaceModels`.

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.

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
```

Fitting a 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

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
```

```r
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.

Filtering With a StructTS Model

The `KalmanRun` function can filter your data based on a state-space model create by `StructTS`.

```
filt <- KalmanRun(y, struct)
```

Example

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

```
y <- datasets::Nile

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

filt <- KalmanRun(y, struct$model)
str(filt)
```

```
## List of 3
##  $ values: Named num [1:2] 5.02 1.1
##   ..- attr(*, "names")= chr [1:2] "Lik" "s2"
##  $ resid : num [1:100] 2.3169 1.9826 0.0849 1.7917 0.9641 ...
##  $ states: num [1:100, 1:2] 877 952 954 1022 1059 ...
```

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

Smoothing With a StructTS Model

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

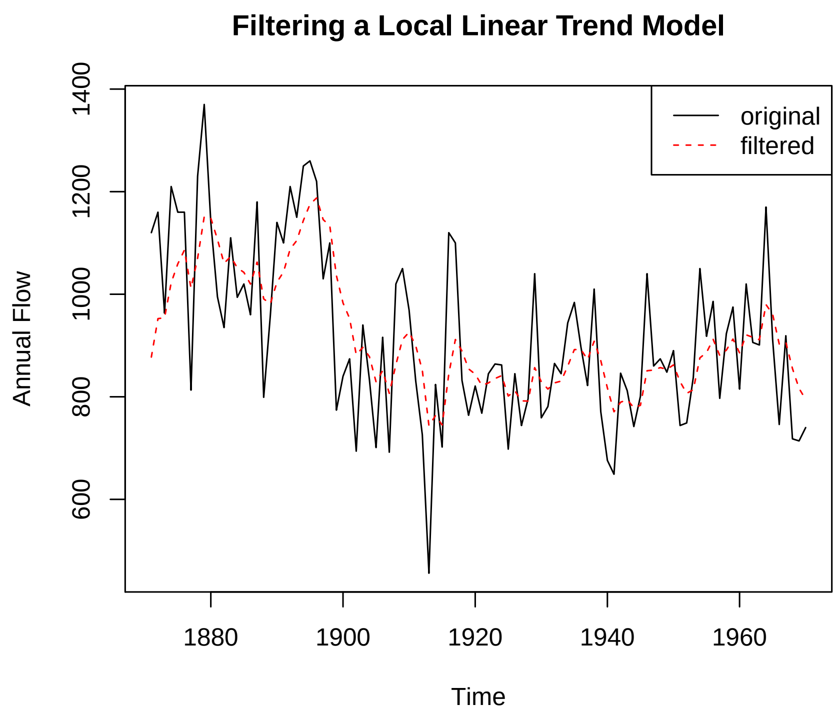
```
smoothed <- tsSmooth(struct)
```

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)
```

```
## mts [1:100, 1:2] 1115 1114 1107 1115 1113 ...
## - attr(*, "dimnames")=List of 2
## ..$ : NULL
## ..$ : chr [1:2] "level" "slope"
## - attr(*, "tsp")= num [1:3] 1871 1970 1
## - attr(*, "class")= chr [1:3] "mts" "ts" "matrix"
```

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

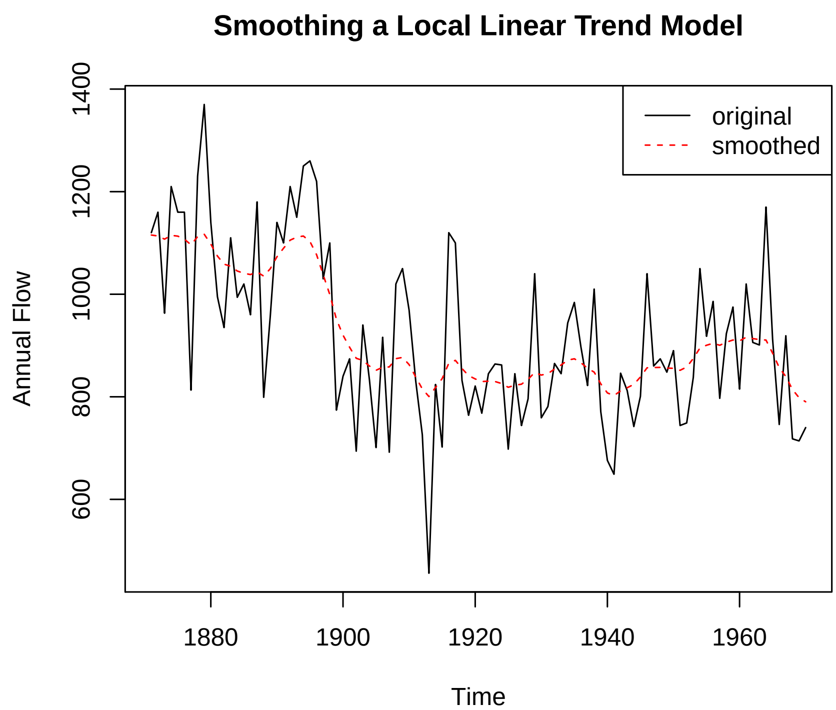
Diagnostics for a StructTS Model

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

```
tsdiag(struct)
```

Example

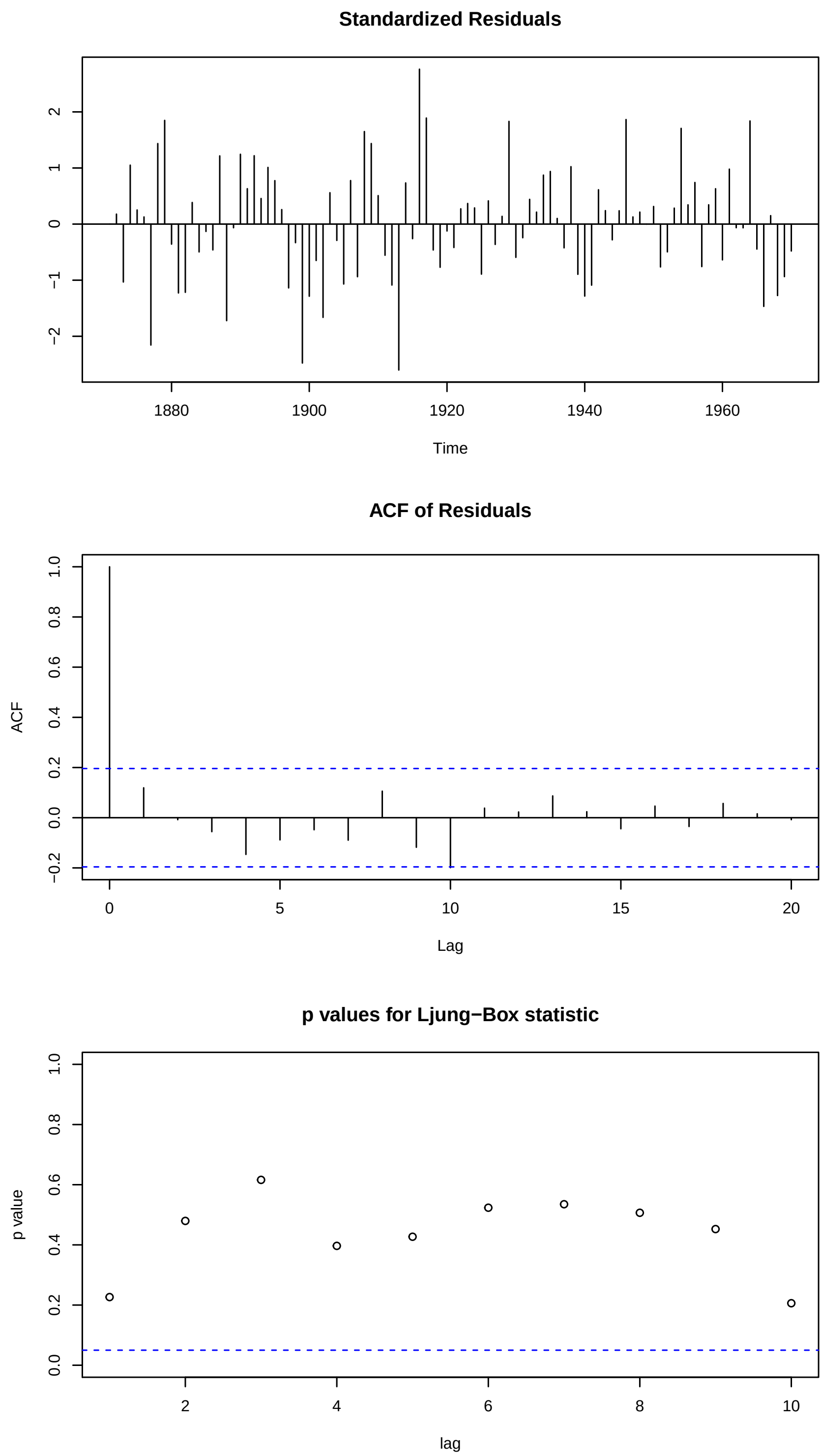
This code constructs a local linear trend model for the Nile data, then produces diagnostics plots.




```
y <- datasets::Nile

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

tsdiag(struct)
```


Standardized Residuals
Time
ACF of Residuals
ACF
Lag
p values for Ljung−Box statistic
p value
lag

Regression Model, Fixed Coefficients

This model adds an explanatory varible with fixed coefficient λ. The coefficient is “fixed” in the sense that it does not vary over time.

In the next section, we will consider models with time-varying coefficients.

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

The state vector is $\alpha_t = (\mu_t, \lambda)^\top$, where the subscript, t, indicates that λ_t varies over time.

This is a four-parameter model.

σ^2_ϵ	Variance of observation errors, ϵ
σ^2_ξ	Variance of transition errors, ξ
μ_0	Initial level of μ
λ	Coefficient of x

To estimate the model parameters, we first define a function that constructs a `dlm` model object from four parameters. A key fact here is that we set the second component of W to be zero. That forces `dlm` to keep the second state variable, λ, constant.

```
buildModReg <- function(v) {
    dV <- exp(v[1])
    dW <- c(exp(v[2]), 0)  # Note zero variance for lambda
    m0 <- v[3:4]
    dlmModReg(x, dV = dV, dW = dW, m0 = m0)
}
```

The argument to the function is a 4-element vector containing the model parameters.

- `v[1]` = Log of σ^2_ϵ
- `v[2]` = Log of σ^2_ξ
- `v[3]` = Initial level for μ
- `v[4]` = Value of λ

We need guesses for the parameters. Fortunately, reasonable guess will do.

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

The dlmMLE function uses numerical optimzation to find the maximum likelihood estimates (MLE) for the model parameters. Starting with our reasonable guesses for parameters, it will repeatedly call our buildModReg function, calculate the model's likelihood, and find the MLE values. Always check for convergence.

```
parm <- c(log(varGuess), log(varGuess/5), mu0Guess,
    lambdaGuess)
mle <- dlmMLE(y, parm = parm, build = buildModReg)

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

The function returns the final parameter values, not the final model, so we construct the final model ourselves from the parameters.

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

Example

This example uses an explanatory variable to account for a change in the river's level. The example is taken from the excellent paper by Petris and Petrone.[7]

The explanatory variable is quite simple. It has value 0.0 *before* the Aswan Dam was built and value 1.0 *after* the dam was built. The dam had a significant effect on the river's level, so it makes sense as an explanatory variable.

Here, the explanatory variable is called x. We can construct it "manually" from our knowledge of the data: the dam was built after the 27th observation.

```
library(dlm)

y <- datasets::Nile
x <- cbind(c(rep(0, 27), rep(1, length(y) - 27)))

buildModReg <- function(v) {
    dV <- exp(v[1])
    dW <- c(exp(v[2]), 0)
    m0 <- v[3:4]
    dlmModReg(x, dV = dV, dW = dW, m0 = m0)
}

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

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

```
lambdaGuess <- mean(diff(y), na.rm = TRUE)

parm <- c(log(varGuess), log(varGuess/5), mu0Guess,
    lambdaGuess)
mle <- dlmMLE(y, parm = parm, build = buildModReg)

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

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

Regression Model, Time-Varying Coefficients

This recipe is similar to the previous recipe, but now the coefficient of the explanatory variable *does* vary over time. The equational formulation is similar. The difference is that the slope, λ, becomes λ_t, subscripted by time.

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

The state vector is $\alpha_t = (\mu_t, \lambda_t)^\top$, where both components vary over time.

The λ_t follow a random walk with error terms ζ_t, and that introduces a new parameter, σ_ζ^2, the variance of the errors. The full set of five parameters is:

σ_ϵ^2	Variance of observation errors, ϵ
σ_ξ^2	Variance of transition errors, ξ
σ_ζ^2	Variance of transition errors, ζ
μ_0	Initial level of μ
λ_0	Initial level of λ

The model-building function is similar to the previous recipe, but does *not* force the variance of λ to zero.

```
buildModReg <- function(v) {
    dV <- exp(v[1])
    dW <- exp(v[2:3])  # Variances for mu, lambda
    m0 <- v[4:5]  # Initial levels for mu, lambda
    dlmModReg(x, dV = dV, dW = dW, m0 = m0)
}
```

We need reasonable guesses for the parameters: variances and initial levels.

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

We call `dlmMLE` to estimate the MLE parameters through numerical optimization, checking for convergence.

```
parm <- c(log(varGuess), log(varGuess/5), log(varGuess/5),
    mu0Guess, lambda0Guess)
mle <- dlmMLE(y, parm = parm, build = buildModReg)

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

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

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

Example

This is yet another model of the Nile River data, using the same explanatory variable, `x`, as the previous recipe but letting its coefficient vary over time.

```
library(dlm)

y <- datasets::Nile
x <- cbind(c(rep(0, 27), rep(1, length(y) - 27)))

buildModReg <- function(v) {
    dV <- exp(v[1])
    dW <- exp(v[2:3])
    m0 <- v[4:5]
    dlmModReg(x, dV = dV, dW = dW, m0 = m0)
}

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

parm <- c(log(varGuess), log(varGuess/5), log(varGuess/5),
    mu0Guess, lambda0Guess)
mle <- dlmMLE(y, parm = parm, build = buildModReg)

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

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

Filtering With a dlm Model

The dlm package provides a function, dlmFilter, which can filter your data based on model.

```
filt <- dlmFilter(y, model)
## filt$m contains the filtered values
```

Example

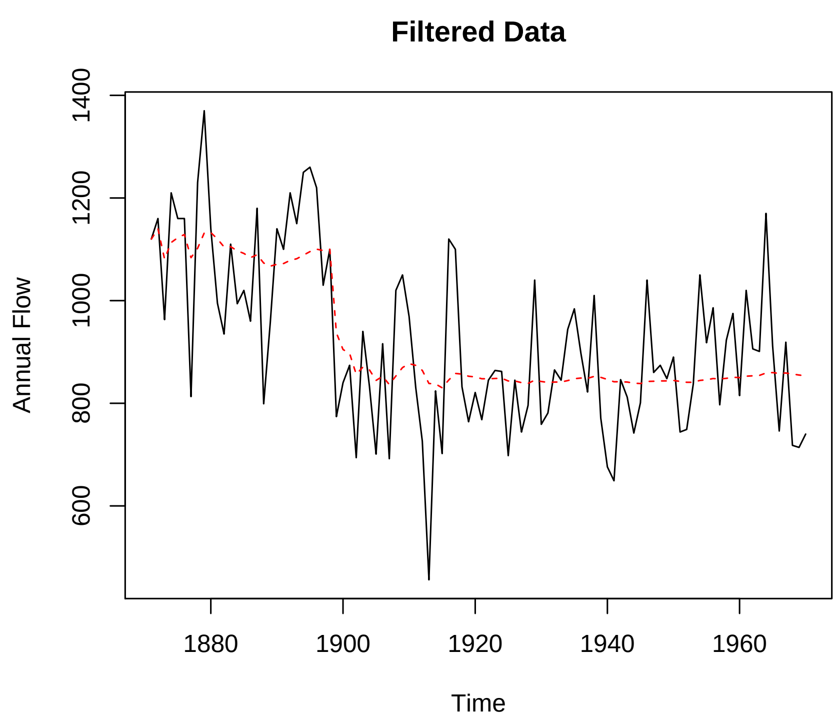
This example uses the model created in the example, above, of regression with fixed coefficients.

The example also assumes that ‘x‘ and ‘y‘ are the regressor and time series data from that example.

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

## The final, filtered data is this linear
## combination
filtered <- filt$m[-1, 1] + x * filt$m[-1, 2]

both <- cbind(y = y, filtered = filtered)
plot(both, plot.type = "single", lty = c("solid",
    ALT_STYLE), col = c("black", ALT_COLOR), main = "Filtered Data",
    ylab = "Annual Flow")
```



Smoothing With a dlm Model

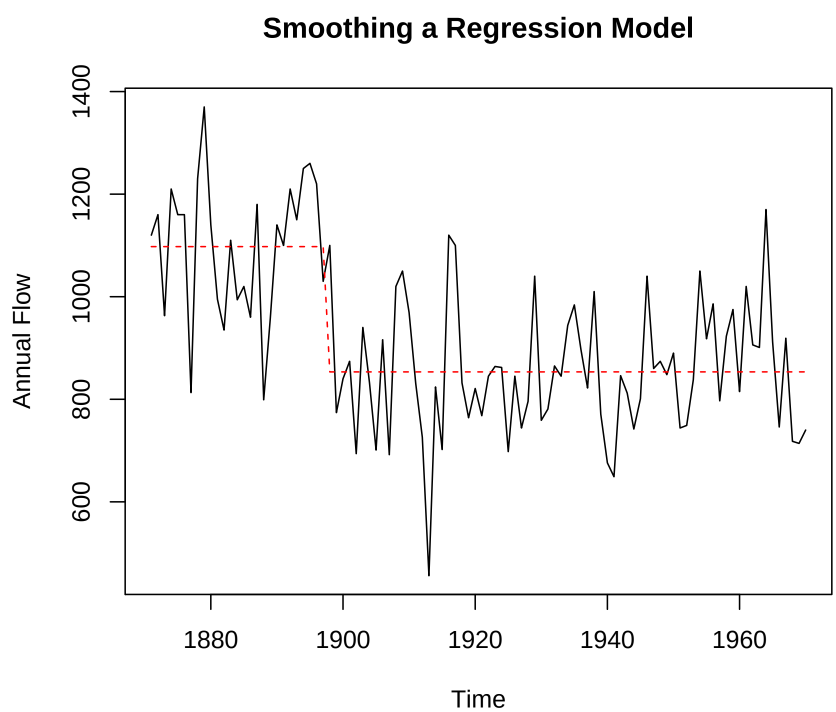
The `dlm` package provides a function, `dlmSmooth`, for smoothing your data based on a model. If y is your data and `model` is any model created by `dlm`, such as the recipes in this monograph, then this call will compute the smoothed data.

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

Example

This example assumes that `model` was created by the example, above, for estimating a regression with fixed coefficients. It smooths the original data based on that model, then plots both the data and smoothed values.

The example code also assumes that 'x' and 'y' are the predictor and the time series data, respectively, from that recipe.



Diagnostics for a dlm Model

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.

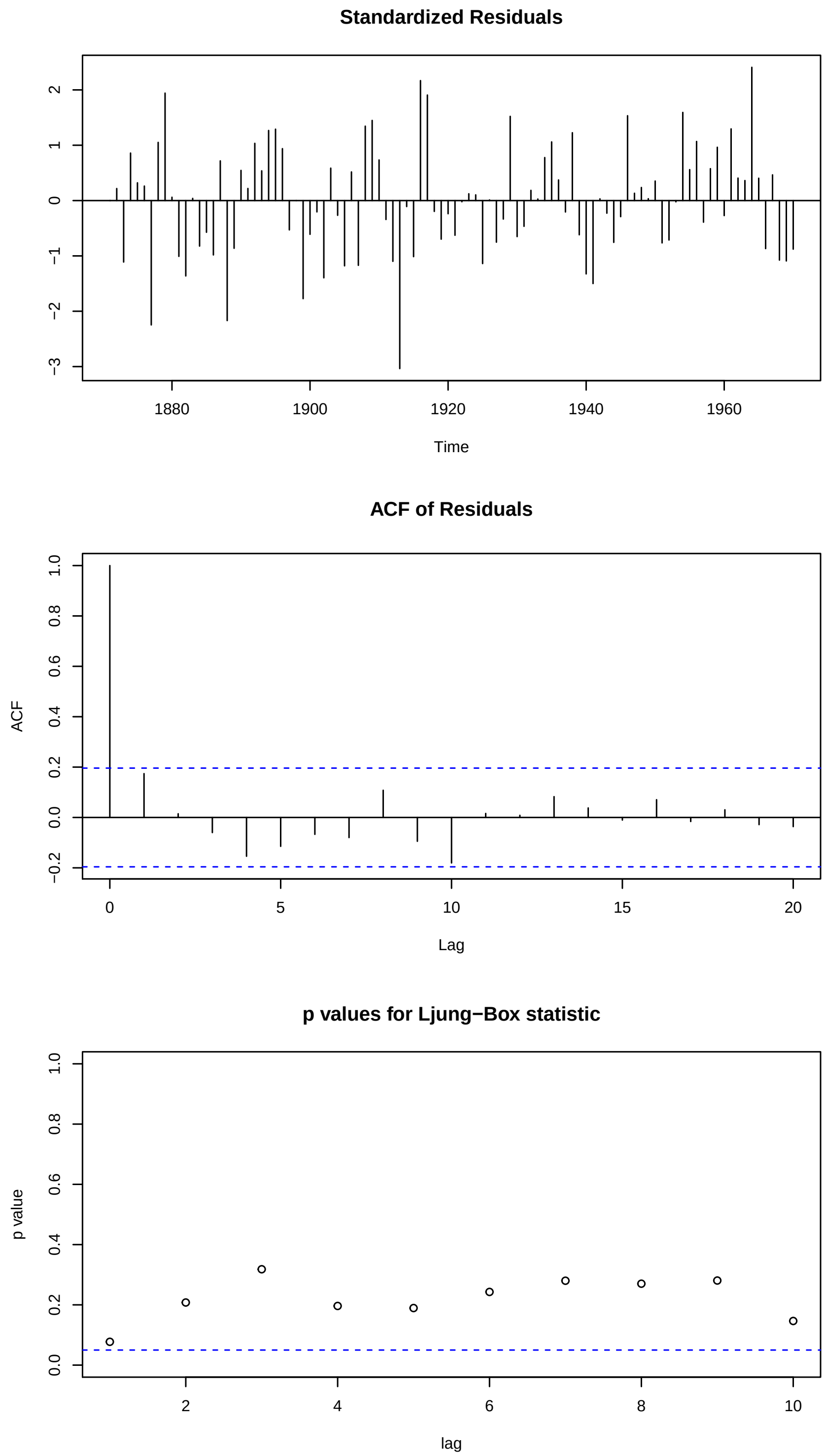
```
filt <- dlmFilter(y, model)
tsdiag(filt)
```

Example

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

The code also assumes that ‘x‘ and ‘y‘ are the regressor and time series data, respectively, as in that recipe.

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


Standardized Residuals
Time
ACF of Residuals
ACF
Lag
p values for Ljung−Box statistic
p value
lag

Boostrapping a State-Space Model

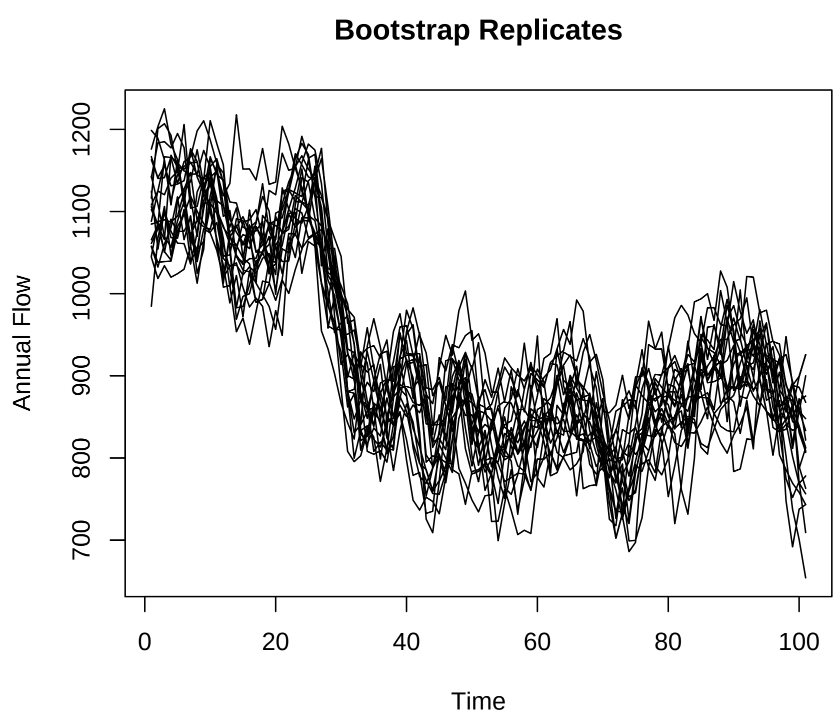
Bootstrapping is a powerful technique for studying the variance of your data, and the `dlmBSample` provides an easy means for generating the time-series replicates from a `dlm` model.

Bootstrapping is done from the posterior distribution, so we call `dlmFilter` and then use `dlmBSample` to draw samples from the distribution. Each call to `dlmBSample` draws one sample. We typically use the `replicate` function to draw samples repeatedly and form them into an array.

An Appendix shows how to create a local level model with 'dlm'.

For sake of example, assume that `model` is a local level model. This code will construct the posterior distribution using `dlmFilter`, create 20 bootstrap replicates using `dlmBSample`, and plot all the replicates in a single panel.

```
filt <- dlmFilter(y, model)
repls <- replicate(20, dlmBSample(filt))
plot(as.ts(repls), plot.type = "single", main = "Bootstrap Replicates",
    ylab = "Annual Flow")
```



Appendix: Estimating the Local Level Model Via the `dlm` Package

The Local Level Model recipe, 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.

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.

The `dlmMLE` function finds the maximum likelihood estimate of the parameters by repeatedly calling our `buildModPoly1` until it converges on the MLE solution. Always check for convergence.

```
mle <- dlmMLE(y, parm = c(1, 1, y[1]), buildModPoly1)

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

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

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

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)
}

mle <- dlmMLE(y, parm = c(1, 1, y[1]), buildModPoly1)
if (mle$convergence != 0) stop(mle$message)

model <- buildModPoly1(mle$par)

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

```
## Observational variance: 15098.68
##  Transitional variance: 1469.009
##  Initial state: 1120
```

Appendix: Estimating the Local Linear Trend Model Via 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

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)

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

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

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

Appendix: The 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.

Estimation via `StructTS`

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

```
struct = StructTS(y, type = "level", fixed = c(0,
    NA))
```

Estimation via `dlm`

The R 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

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)

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

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