MARSS Package Manual

E. E. Holmes

2019-11-27

Contents

Preface

The **MARSS** R package allows you to fit **constrained** multivariate autoregressive state-space models.

This manual covers the **MARSS R** package: what it does, how to set up your models, how to structure your input, and how to get different types of output. For vignettes showing how to use MARSS models to analyze data, see the companion book *MARSS Modeling for Environmental Data* by Holmes, Scheurell, and Ward.

Installation

To install and load the **MARSS** package from CRAN:

```
install.packages("MARSS")
library(MARSS)
```

The latest release on GitHub may be ahead of the CRAN release. To install the latest release on GitHub:

```
install.packages("devtools")
library(devtools)
install_github("nwfsc-timeseries/MARSS@*release")
library(MARSS)
```

The master branch on GitHub is not a ‘release’. It has work leading up to a GitHub release. The code here may be broken though usually preliminary work is done on a development branch. To install the master branch:

```
install_github("nwfsc-timeseries/MARSS")
```

If you are on a Windows machine and get an error saying ‘loading failed for i386’ or similar, then try

```
options(devtools.install.args = "--no-multiarch")
```

To install an **R** package from Github, you need to be able to build an **R** package on your machine. If you are on Windows, that means you will need to install Rtools. On a Mac, installation should work fine; you don’t need to install anything.

Author

Elizabeth E. Holmes is a research scientist at the Northwest Fisheries Science Center (NWFSC) a US Federal government research center. This work was conducted as part of her job for NOAA Fisheries, as such the work is in the public domain and cannot be copyrighted.

Links to more code and publications can be found on our academic websites:

- http://faculty.washington.edu/eeholmes

Citation

Holmes, E. E. 2019. MARSS Manual. NOAA Fisheries, Northwest Fisheries Science Center, 2725 Montlake Blvd E., Seattle, WA 98112. Contact eli.holmes@noaa.gov.

Part 1. Overview

Here the **MARSS** package and MARSS models are briefly introduced. Part 2 shows a series of short examples and Part 3 goes into output from **MARSS** fitted objects and MARSS models in general.

Chapter 1

Overview

MARSS stands for Multivariate Auto-Regressive(1) State-Space. The **MARSS** package is an ,**R** package for estimating the parameters of linear MARSS models with Gaussian errors. This class of model is extremely important in the study of linear stochastic dynamical systems, and these models are important in many different fields, including economics, engineering, genetics, physics and ecology (Appendix **??**). The model class has different names in different fields, for example in some fields they are termed dynamic linear models (DLMs) or vector autoregressive (VAR) state-space models. The **MARSS** package allows you to easily fit time-varying constrained and unconstrained MARSS models with or without covariates to multivariate time-series data via maximum-likelihood using primarily an EM algorithm. The EM algorithm in the **MARSS** package allows you to apply linear constraints on all the parameters within the model matrices. Fitting via the BFGS algorithm is also provided in the package using ,**R**'s `optim` function, but this is not the focus of the **MARSS** package.

MARSS, `MARSS()` and MARSS. MARSS model refers to the class of models which the **MARSS** package fits using, primarily, an EM algorithm. In the text, **MARSS** refers to the ,**R** package. Within the package, the main fitting function is `MARSS()`. When the class of model is being discussed, rather than the package or the function, "MARSS" is used.

A full MARSS model, with Gaussian errors, takes the form:

$$\begin{gathered}
\mathbf{x}_t = \mathbf{B}_t\mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t\mathbf{c}_t + \mathbf{G}_t\mathbf{w}_t,\ \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \\
\mathbf{y}_t = \mathbf{Z}_t\mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t\mathbf{d}_t + \mathbf{H}_t\mathbf{v}_t,\ \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t) \\
\mathbf{x}_1 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda}) \text{ or } \mathbf{x}_0 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda})
\end{gathered} \tag{1.1}$$

The $\mathbf{x}$ equation is termed the state process and the $\mathbf{y}$ equation is termed the observation process. Data enter the model as the $\mathbf{y}$; that is the $\mathbf{y}$ is treated as the data although there may be missing data. The $\mathbf{c}_t$ and $\mathbf{d}_t$ are inputs (aka, exogenous variables, covariates or indicator variables). The $\mathbf{G}_t$ and $\mathbf{H}_t$ are also typically inputs (fixed values with no missing values).

The bolded terms are matrices with the following definitions:

$\mathbf{x}$ is a $m \times T$ matrix of states. Each $\mathbf{x}_t$ is a realization of the random variable $\mathbf{X}_t$ at time t.

$\mathbf{w}$ is a $m \times T$ matrix of the process errors. The process errors at time t are multivariate normal with mean 0 and covariance matrix $\mathbf{Q}_t$.

$\mathbf{y}$ is a $n \times T$ matrix of the observations. Some observations may be missing.

$\mathbf{v}$ is a $n \times T$ column vector of the non-process errors. The observation erros at time t are multivariate normal with mean 0 and covariance matrix $\mathbf{R}_t$.

$\mathbf{B}_t$ and $\mathbf{Z}_t$ are parameters and are $m \times m$ and $n \times m$ matrices.

$\mathbf{u}_t$ and $\mathbf{a}_t$ are parameters and are $m \times 1$ and $n \times 1$ column vectors.

$\mathbf{Q}_t$ and $\mathbf{R}_t$ are parameters and are $g \times g$ (typically $m \times m$) and $h \times h$ (typically $n \times n$) variance-covariance matrices.

$\boldsymbol{\pi}$ is either a parameter or a fixed prior. It is a $m \times 1$ matrix.

$\boldsymbol{\Lambda}$ is either a parameter or a fixed prior. It is a $m \times m$ variance-covariance matrix.

$\mathbf{C}_t$ and $\mathbf{D}_t$ are parameters and are $m \times p$ and $n \times q$ matrices.

$\mathbf{c}$ and $\mathbf{d}$ are inputs (no missing values) and are $p \times T$ and $q \times T$ matrices.

$\mathbf{G}_t$ and $\mathbf{H}_t$ are inputs (no missing values) and are $m \times g$ and $n \times h$ matrices.

AR(p) models can be written in the above form by properly defining the $\mathbf{x}$ vector and setting some of the $\mathbf{R}$ variances to zero; see Chapter **??**. Although the model appears to only include i.i.d. errors ($\mathbf{v}_t$ and $\mathbf{w}_t$), in practice, AR(p) errors can be included by moving the error terms into the state model. Similarly, the model appears to have independent process ($\mathbf{v}_t$) and observation ($\mathbf{w}_t$) errors, however, in practice, these can be modeled as identical or correlated by using one of the state processes to model the errors with the $\mathbf{B}$ matrix set appropriately for AR or white noise—although one may have to fix many of the parameters associated with the errors to have an identifiable model. Study the application chapters and textbooks on MARSS models (Appendix **??**) for examples of how a wide variety of autoregressive models can be written in MARSS form.

1.1 Examples of MARSS models

Written in an unconstrained form, meaning all the elements in a parameter matrices are allowed to be different and none constrained to be equal or related, a MARSS model can be written out as follows. Two state processes ($\mathbf{x}$) and three observation processes ($\mathbf{y}$) are used here as an example.

$$\begin{bmatrix}x_1\\x_2\end{bmatrix}_t = \begin{bmatrix}b_{11} & b_{12}\\ b_{21} & b_{22}\end{bmatrix}\begin{bmatrix}x_1\\x_2\end{bmatrix}_{t-1} + \begin{bmatrix}w_1\\w_2\end{bmatrix}_t, \quad \begin{bmatrix}w_1\\w_2\end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix}u_1\\u_2\end{bmatrix}, \begin{bmatrix}q_{11} & q_{12}\\ q_{21} & q_{22}\end{bmatrix}\right)$$

$$\begin{bmatrix}y_1\\y_2\\y_3\end{bmatrix}_t = \begin{bmatrix}z_{11} & z_{12}\\ z_{21} & z_{22}\\ z_{31} & z_{32}\end{bmatrix}\begin{bmatrix}x_1\\x_2\end{bmatrix}_t + \begin{bmatrix}v_1\\v_2\\v_3\end{bmatrix}_t, \begin{bmatrix}v_1\\v_2\\v_3\end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix}a_1\\a_2\\a_3\end{bmatrix}, \begin{bmatrix}r_{11} & r_{12} & r_{13}\\ r_{21} & r_{22} & r_{23}\\ r_{31} & r_{32} & r_{33}\end{bmatrix}\right)$$

$$\begin{bmatrix}x_1\\x_2\end{bmatrix}_0 \sim \text{MVN}\left(\begin{bmatrix}\pi_1\\ \pi_2\end{bmatrix}, \begin{bmatrix}\nu_{11} & \nu_{12}\\ \nu_{21} & \nu_{22}\end{bmatrix}\right) \quad \textit{or} \quad \begin{bmatrix}x_1\\x_2\end{bmatrix}_1 \sim \text{MVN}\left(\begin{bmatrix}\pi_1\\ \pi_2\end{bmatrix}, \begin{bmatrix}\nu_{11} & \nu_{12}\\ \nu_{21} & \nu_{22}\end{bmatrix}\right)$$

However not all parameter elements can be estimated simultaneously. Constraints are required in order to specify a model with a unique solution. The MARSS package allows you to specify constraints by fixing elements in a parameter matrix or specifying that some elements are estimated—and have a linear relationship to other elements. Here is an example of a MARSS model with fixed and estimated parameter elements:

$$\begin{bmatrix}x_1\\x_2\end{bmatrix}_t = \begin{bmatrix}a&0\\0&a\end{bmatrix}\begin{bmatrix}x_1\\x_2\end{bmatrix}_{t-1} + \begin{bmatrix}w_1\\w_2\end{bmatrix}_t, \quad \begin{bmatrix}w_1\\w_2\end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix}0.1\\u\end{bmatrix}, \begin{bmatrix}q_{11}&q_{12}\\q_{12}&q_{22}\end{bmatrix}\right)$$

$$\begin{bmatrix}y_1\\y_2\\y_3\end{bmatrix}_t = \begin{bmatrix}d&d\\c&c\\1+2d+3c&2+3d\end{bmatrix}\begin{bmatrix}x_1\\x_2\end{bmatrix}_t + \begin{bmatrix}v_1\\v_2\\v_3\end{bmatrix}_t, \quad \begin{bmatrix}v_1\\v_2\\v_3\end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix}a_1\\a_2\\0\end{bmatrix}, \begin{bmatrix}r&0&0\\0&r&0\\0&0&r\end{bmatrix}\right)$$

$$\begin{bmatrix}x_1\\x_2\end{bmatrix}_0 \sim \text{MVN}\left(\begin{bmatrix}\pi\\\pi\end{bmatrix}, \begin{bmatrix}1&0\\0&1\end{bmatrix}\right)$$

Notice that some elements are fixed (in this case to 0, but could be any fixed number), some elements are shared (have the same value), and some elements are linear combinations of other estimated values (c, $1+2d+3c$ and $2+3d$ are linear combinations of c and d).

Chapter 2

How to get started (quickly)

If you already work with models in the form of Equation **??**, you can immediately fit your model with the **MARSS** package. Install the **MARSS** package and then type `library(MARSS)` at the command line to load the package. Look at the Quick Start Guide and then skim through Chapter **??** to, hopefully, find an example similar to your application. Appendix **??** also has many examples of how to specify different forms for your parameter matrices.

Chapter 3

Getting your data in right format

Your data need to be a matrix (not data frame nor a **ts** object) with time across the columns ($n \times T$ matrix). The **MARSS** functions assume discrete time steps and you will need a column for each time step. Replace any missing time steps with NA. Why does **MARSS** require your data in matrix form? Because **MARSS** will not make any guesses about your intentions. You must be 100% explicit in terms of what model you trying to fit and what you consider to be `data`. MARSS models are used in many different fields in different ways. There is no 'guess' that would work for all models. Instead **MARSS** requires that you write your model in matrix form, and then pass everything in in a format that is one-to-one with that mathematical model. That way **MARSS** knows exactly what you are trying to fit.

3.1 ts objects

A ,**R** **ts** object (time series object) stores information about the time steps of the data and often seasonal information (the quarter or month). **MARSS** needs this information in matrix form. If you have your data in **ts** form, then you may be using year and season (quarter, month) as covariates to estimate trend and seasonality. The next sections give examples of converting your data from **ts** form to matrix form.

3.1.1 Univariate ts object

This converts a univariate `ts` object with year and quarter into a matrix with a row for the response (here called `Temp`), year, and quarter.

```
z <- ts(rnorm(10), frequency = 4, start = c(1959, 2))
dat <- data.frame(Yr = floor(time(z) + .Machine$double.eps),
      Qtr = cycle(z), Temp=z)
dat <- t(dat)
class(dat)
```

Notice that the class of `dat` is matrix, which is what we want. There are three rows, first is the reponse and the second and third are the covariates, Year and Quarter. When you call MARSS, `dat["Temp",]` is the data. `dat[c("Yr","Qtr"),]` are your covariates.

3.1.2 Multivariate ts object

In this example, we have two temperature readings, our responses, and a salinity reading, a covariate. The data are monthly.

```
z <- ts(matrix(rnorm(300), 100, 3), start = c(1961, 1),
     frequency = 12, names=c("Temp1","Temp2","Sal"))
dat <- data.frame(Yr = floor(time(z) + .Machine$double.eps),
     Month = cycle(z), z)
dat <- t(dat)
```

When you call MARSS, `dat[c("Temp1","Temp2"),]` are the data and `dat[c("Yr","Month","Sal"),]` are your covariates.

See the *MARMES* chapters that discuss seasonality for examples of how to model season. The brute force method of treating month or quarter as a factor requires estimation of more parameters than is necessary in many cases.

3.2 tsibble objects

There are many ways that you can transform a **tsibble** object into a matrix with each row being an observed time series.

```
library(tidyverse)
library(tsibble)
dat <- tourism %>% as_tibble %>% tidyr::spread(Quarter, Trips)
dat.matrix <- as.matrix(dat[, -1 * (1:3)])
```

Part 2. Examples

Here a series of short examples are shown for different types of models specified in MARSS structure. Output is shown briefly. See the chapters on different types of output from **MARSS** model objects and MARSS models in general.

In these examples, we will use the default `form="marxss"` argument for a `MARSS()` call. This specifies a MARSS model of the form:

$$\begin{gathered}
\mathbf{x}_t = \mathbf{B}_t\mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t\mathbf{c}_t + \mathbf{G}_t\mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \\
\mathbf{y}_t = \mathbf{Z}_t\mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t\mathbf{d}_t + \mathbf{H}_t\mathbf{v}_t, \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t) \\
\mathbf{x}_1 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda}) \text{ or } \mathbf{x}_0 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda})
\end{gathered} \tag{3.1}$$

The **x** on the left are the hidden states. The **y** on the left are the observed data. Missing values are allowed in **y**. The **c** and **d** are inputs (not estimated). Bolded capitalized values on the right are parameters and 2D (or 3D if time-varying) matrices. Parameters can be estimated, constrained or fixed at a specific value. Within a parameter matrix, you can have a combination of estimated, constrained (shared), or fixed values. The **u** and **a** are parameter column-matrices and can be similarly estimated, constrained or fixed. **w** and **v** are the errors and are computed values, after parameters are estimated.

Chapter 4

Univariate Models

4.1 Random walk with drift

A univariate random walk with drift observed with error is:

$$\begin{gathered} x_t = x_{t-1} + u + w_t,\ w_t \sim \mathrm{N}(0,q) \\ y_t = x_t + v_t,\ v_t \sim \mathrm{N}(0,r) \end{gathered} \tag{4.1}$$

To fit this model to a simulated random walk:

```
u <- 0.01
r <- 0.01
q <- 0.1
TT <- 100
yt <- cumsum(rnorm(TT, u, sqrt(q))) + rnorm(TT, 0, sqrt(r))
fit <- MARSS(yt)
```

```
Success! abstol and log-log tests passed at 35 iterations.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Estimation converged in 35 iterations.
```

```
Log-likelihood: -24.85696
AIC: 57.71391   AIC: 58.13496

      Estimate
R.R     0.0184
U.U     0.0408
Q.Q     0.0633
x0.x0  -0.2653
Initial states (x0) defined at t=0

Standard errors have not been calculated.
Use MARSSparamCIs to compute CIs and bias estimates.
```

4.1.1 Output

See the chapter on Outputs. This is a brief introduction to the outputs.

Use the **broom** package to get the model output in **tidy** form. The confidence intervals shown are approximate and based on the estimated Hessian matrix. See `?tidy.marssMLE` for information on how to change to a different type of confidence interval.

```r
broom::tidy(fit)
```

```
   term    estimate   std.error      conf.low  conf.high
1   R.R  0.01843982 0.009894122 -0.0009523011 0.03783194
2   U.U  0.04083656 0.025345531 -0.0088397685 0.09051289
3   Q.Q  0.06328006 0.017784821  0.0284224487 0.09813767
4 x0.x0 -0.26534896 0.281394026 -0.8168711121 0.28617320
```

The get the estimated states use:

```r
head(broom::augment(fit, type = "states"))
```

```
  .rownames t        xtT     .fitted   .se.fit    .resids .std.resid
1      X.Y1 1 -0.2246253 -0.22451240        NA         NA         NA
2      X.Y1 2 -0.1370475 -0.18378875 0.2080856 0.04674122  0.2778325
3      X.Y1 3  0.1814517 -0.09621097 0.2073928 0.27766272  1.7317264
4      X.Y1 4  0.2708522  0.22228831 0.2073676 0.04856391  0.3034844
```

```
5      X.Y1 5  0.3617262  0.31168877 0.2073666 0.05003744  0.3127155
6      X.Y1 6  0.7986076  0.40256277 0.2073666 0.39604480  2.4751399
```

Note that generic rownames were given since none were specified for `dat`. You can also get all the Kalman filter and smoother estimates for x from `MARSSkf()`. See `?MARSSkf`.

The get the fitted values, the estimated y, use:

```
head(broom::augment(fit, type = "observations"))
```

```
  .rownames t          y    .fitted    .se.fit      .resids .std.resid
1         Y1 1 -0.2382786 -0.2246253 0.07978438 -0.013653305  0.2553149
2         Y1 2 -0.2043381 -0.1370475 0.07698023 -0.067290571  2.0850049
3         Y1 3  0.2482112  0.1814517 0.07687621  0.066759440  1.3890582
4         Y1 4  0.2704228  0.2708522 0.07687242 -0.000429388  0.5309852
5         Y1 5  0.2608996  0.3617262 0.07687228 -0.100826613  2.9353642
6         Y1 6  0.9039099  0.7986076 0.07687228  0.105302282  1.7417058
```

There are two types of fitted values that are used in the state-space literature: the one-step-ahead which uses on the data up to $t-1$ and the smoothed fitted values which uses all the data. Read up on fitted values for MARSS models at `?fitted.marssMLE`.

4.2 AR(1) observed with error

With the addition of b in front of x_{t-1} we have an AR(1) process.

$$\begin{gathered} x_t = bx_{t-1} + u + w_t,\ \ w_t \sim \text{N}(0,q) \\ y_t = x_t + v_t,\ \ w_t \sim \text{N}(0,r) \end{gathered} \tag{4.2}$$

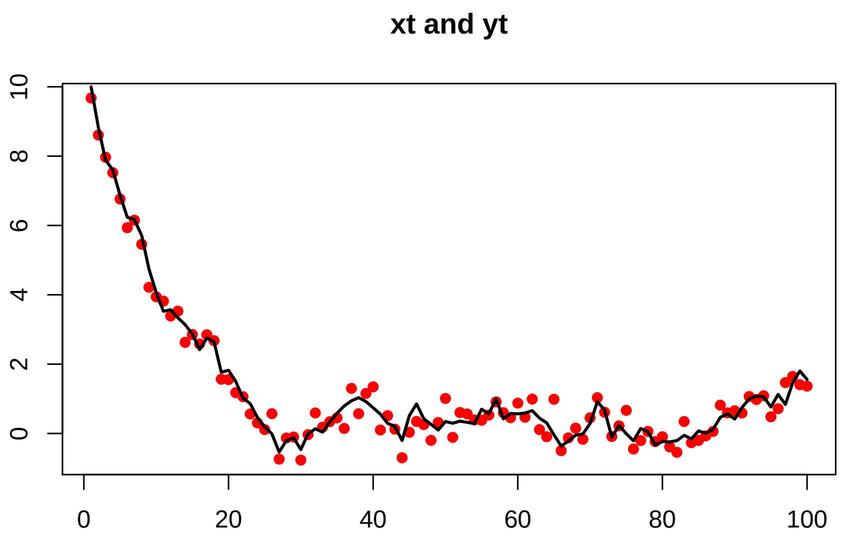
To fit this model to a simulated, non-stationary, AR(1) process:

```
set.seed(123)
u <- 0.01
r <- 0.1
q <- 0.1
b <- 0.9
TT <- 100
```

```
x0 <- 10
xt.ns <- rep(x0, TT)
for (i in 2:TT) xt.ns[i] <- b * xt.ns[i - 1] + u + rnorm(1, 0,
    sqrt(q))
yt.ns <- xt.ns + rnorm(TT, 0, sqrt(r))
```

The process was purposefully made to be non-stationary by setting `x0` well outside the stationary distribution of x. The EM algorithm in **MARSS** does not require that the underlying state processes be stationary and it is not necessary to remove the initial non-stationary part of the time-series.

```
plot(yt.ns, xlab = "", ylab = "", main = "xt and yt", pch = 16,
    col = "red")
lines(xt.ns, lwd = 2)
```



```
fit <- MARSS(yt.ns, model = list(B = matrix("b")))
```

```
Success! abstol and log-log tests passed at 24 iterations.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
```

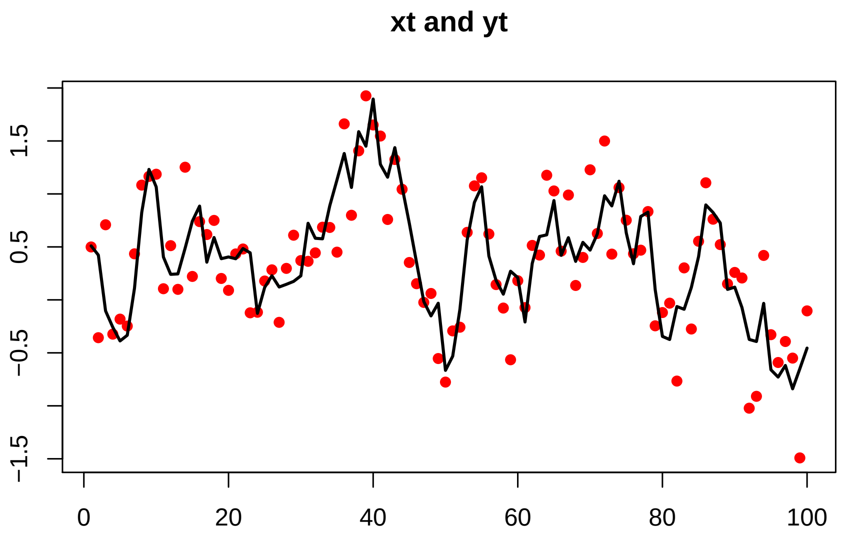
```
Estimation converged in 24 iterations.
Log-likelihood: -61.67475
AIC: 133.3495   AICc: 133.9878

      Estimate
R.R     0.1075
B.b     0.9042
U.U     0.0346
Q.Q     0.0535
x0.x0  10.6409
Initial states (x0) defined at t=0

Standard errors have not been calculated.
Use MARSSparamCIs to compute CIs and bias estimates.
```

We could also simulate AR(1) data with `stats::arima.sim()` however this will produce stationary data:

```
xt.s <- arima.sim(n = TT, model = list(ar = b), sd = sqrt(q))
yt.s <- xt.s + rnorm(TT, 0, sqrt(r))
yt.s <- as.vector(yt.s)
xt.s <- as.vector(xt.s)
```

These stationary data can be fit as before but the data must be a matrix with time across the columns not a **ts** object. If you pass in a vector, `MARSS()` will convert that to a matrix with one row.

```
plot(yt.s, xlab = "", ylab = "", main = "xt and yt", pch = 16,
    col = "red", type = "p")
lines(xt.s, lwd = 2)
```



```
fit <- MARSS(yt.s, model = list(B = matrix("b")))
```

```
Success! abstol and log-log tests passed at 27 iterations.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Estimation converged in 27 iterations.
Log-likelihood: -74.40797
AIC: 158.8159   AICc: 159.4542

      Estimate
R.R     0.1064
B.b     0.7884
U.U     0.0735
Q.Q     0.1142
x0.x0   0.2825
Initial states (x0) defined at t=0

Standard errors have not been calculated.
Use MARSSparamCIs to compute CIs and bias estimates.
```

Note that u is estimated however `arima.sim()` does not include a u. We can set u to zero if we happened to know that it was zero.

```
fit <- MARSS(yt.s, model = list(B = matrix("b"), U = matrix(0)))
```

```
Success! abstol and log-log tests passed at 16 iterations.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Estimation converged in 16 iterations.
Log-likelihood: -75.80277
AIC: 159.6055   AICc: 160.0266

      Estimate
R.R      0.117
B.b      0.874
Q.Q      0.100
x0.x0    0.339
Initial states (x0) defined at t=0

Standard errors have not been calculated.
Use MARSSparamCIs to compute CIs and bias estimates.
```

If we know r (or q), we could set those too:

```
fit <- MARSS(yt.s, model = list(B = matrix("b"), U = matrix(0),
    R = matrix(r)))
```

```
Success! abstol and log-log tests passed at 18 iterations.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Estimation converged in 18 iterations.
Log-likelihood: -75.88583
```

```
AIC: 157.7717   AICc: 158.0217

      Estimate
B.b      0.859
Q.Q      0.114
x0.x0    0.370
Initial states (x0) defined at t=0

Standard errors have not been calculated.
Use MARSSparamCIs to compute CIs and bias estimates.
```

We can fit to just the x data, an AR(1) with no error, by setting r to zero: If we know r (or q), we could set those too:

```
fit <- MARSS(xt.s, model = list(B = matrix("b"), U = matrix(0),
    R = matrix(0)))
```

```
Success! algorithm run for 15 iterations. abstol and log-log tests passed.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Algorithm ran 15 (=minit) iterations and convergence was reached.
Log-likelihood: -26.9401
AIC: 59.8802   AICc: 60.1302

      Estimate
B.b      0.883
Q.Q      0.100
x0.x0    0.578
Initial states (x0) defined at t=0

Standard errors have not been calculated.
Use MARSSparamCIs to compute CIs and bias estimates.
```

We can fit `xt.s` with `arima()` also. The results will be similar but not identical because `arima()`'s algorithm assumes the data come from a stationary process and the initial conditions are treated differently.

```
arima(xt.s, order = c(1, 0, 0), include.mean = FALSE, method = "ML")
```

```

Call:
arima(x = xt.s, order = c(1, 0, 0), include.mean = FALSE, method = "ML")

Coefficients:
         ar1
      0.8793
s.e.  0.0454

sigma^2 estimated as 0.1009:  log likelihood = -27.98,  aic = 59.96
```

If we try fitting the non-stationary data with `arima()`, the estimates will be poor since `arima()` assumes stationary data:

```
arima(xt.ns, order = c(1, 0, 0), include.mean = FALSE, method = "ML")
```

```

Call:
arima(x = xt.ns, order = c(1, 0, 0), include.mean = FALSE, method = "ML")

Coefficients:
         ar1
      0.9985
s.e.  0.0021

sigma^2 estimated as 0.1348:  log likelihood = -44.59,  aic = 93.19
```

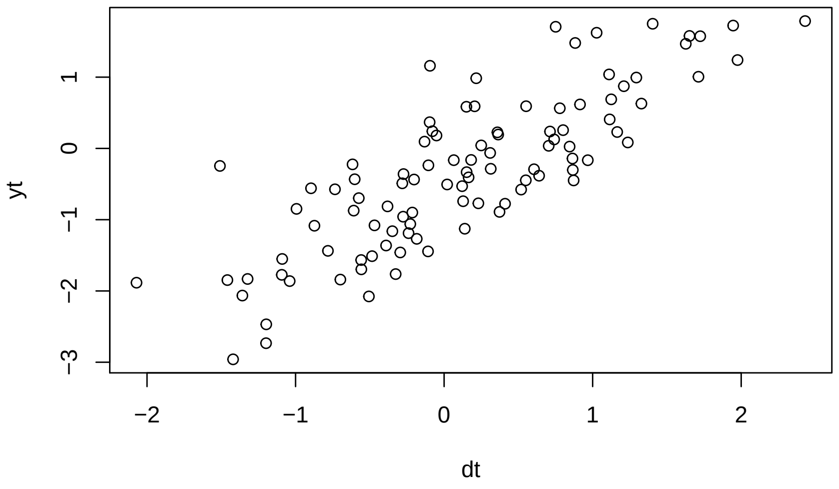
4.3 Linear regression with AR(1) errors

A simple linear regression of one covariate with AR(1) errors is written:

$$\begin{gathered} x_t = bx_{t-1} + w_t,\ w_t \sim \mathrm{N}(0,q) \\ y_t = \beta * d_t + x_t \end{gathered} \tag{4.3}$$

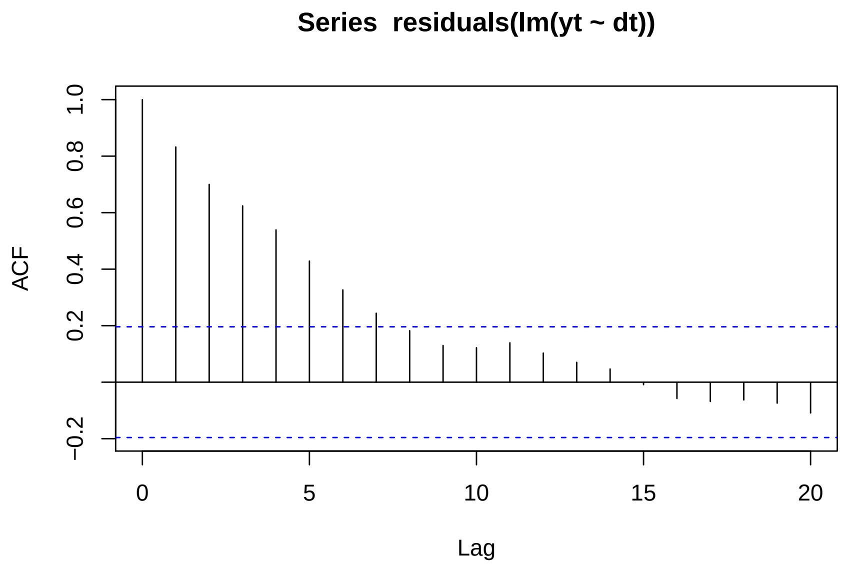
Let's create some simulated data with this structure:

```r
beta <- 1.1
dt <- rnorm(TT, 0, 1)  #our covariate
wt <- arima.sim(n = TT, model = list(ar = b), sd = sqrt(q))
yt <- beta * dt + wt
yt <- as.vector(yt)  # not ts object
plot(dt, yt)
```



If we looked at an ACF of the residuals of a linear regression, we'd see that the residuals are highly autocorrelated:

```r
acf(residuals(lm(yt ~ dt)))
```



We can fit this model (Equation @ref(eq:short.lr.ar1)) with `MARSS()`. Please note that there are many better ,**R** packages specifically designed for linear regression models with correlated errors. This simple example is to help you understand model specification with the **MARSS** package.

To fit this model, we need match our Equation @ref(eq:short.lr.ar1) with the full MARSS model written in matrix form (Equation @ref(eq:marss.part2)). Here it is with the parameters that are zero dropped. $\mathbf{Z}_t$ is identity and is also dropped. The $\mathbf{B}$ and $\mathbf{D}$ are time-constant so the t subscript is dropped. The $\mathbf{x}_t$ are the AR(1) errors and the $\mathbf{y}_t$ is the linear regression with $\mathbf{D}$ being the effect sizes and the $\mathbf{d}$ being the covariate.

$$\begin{gathered}\mathbf{x}_t = \mathbf{B}\mathbf{x}_{t-1} + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{ MVN}(0, \mathbf{Q}_t)\\ \mathbf{y}_t = \mathbf{x}_t + \mathbf{D}\mathbf{d}_t\end{gathered} \tag{4.4}$$

Here is what the model looks like if we write the parameters explicitly in matrix form. The matrices are 1×1.

$$\begin{gathered}\begin{bmatrix}x\end{bmatrix}_t = \begin{bmatrix}b\end{bmatrix}\begin{bmatrix}x\end{bmatrix}_{t-1} + \begin{bmatrix}0\end{bmatrix} + \begin{bmatrix}w\end{bmatrix}_t\\ \begin{bmatrix}y\end{bmatrix}_t = \begin{bmatrix}x\end{bmatrix}_t + \begin{bmatrix}\beta\end{bmatrix}\begin{bmatrix}d\end{bmatrix}_t\end{gathered} \tag{4.5}$$

To create the model list for `MARSS()`, we specify the parameter matrices one-to-one like they look in Equation @ref(eq:short.lr.ar1.mat).

```
R <- matrix("r")  # no v_t
D <- matrix("beta")
U <- matrix(0)  # since arima.sim was used, no u
B <- matrix("b")
d <- matrix(dt, nrow = 1)
A <- matrix(0)
```

`MARSS()` requires **d** be a matrix also. Each row is a covariate and each column is a time step. No missing values allowed as this is an input.

How should we treat the **R** matrix? It is zero, and we could set **R** to zero:

```
R <- matrix(0)
```

However, the EM algorithm in the **MARSS** package will not perform well at all with **R** set to zero and it has to do with how $\mathbf{R} = 0$ affects the update equations. You can use the BFGS algorithm or estimate **R**.

```
R <- matrix("r")
mod.list <- list(B = B, U = U, R = R, D = D, d = d, A = A)
fit <- MARSS(yt, model = mod.list)
```

```
Warning! Abstol convergence only. Maxit (=500) reached before log-log converge

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
WARNING: Abstol convergence only no log-log convergence.
 maxit (=500) reached before log-log convergence.
 The likelihood and params might not be at the ML values.
 Try setting control$maxit higher.
Log-likelihood: -25.53381
AIC: 61.06762   AICc: 61.70592

       Estimate
R.r     0.00264
B.b     0.91256
Q.Q     0.09294
x0.x0  -1.07486
D.beta  1.10169
```

```
Initial states (x0) defined at t=0

Standard errors have not been calculated.
Use MARSSparamCIs to compute CIs and bias estimates.

Convergence warnings
 Warning: the  R.r  parameter value has not converged.
 Type MARSSinfo("convergence") for more info on this warning.
```

Or use the BFGS algorithm for fitting:

```
R <- matrix(0)
mod.list <- list(B = B, U = U, R = R, D = D, d = d, A = A)
fit <- MARSS(yt, model = mod.list, method = "BFGS")
```

```
Success! Converged in 122 iterations.
Function MARSSkfas used for likelihood calculation.

MARSS fit is
Estimation method: BFGS
Estimation converged in 122 iterations.
Log-likelihood: -25.48386
AIC: 58.96772   AICc: 59.38878

       Estimate
B.b      0.9090
Q.Q      0.0975
x0.x0   -1.0754
D.beta   1.1017
Initial states (x0) defined at t=0

Standard errors have not been calculated.
Use MARSSparamCIs to compute CIs and bias estimates.
```

This is the same model you are fitting when you pass in `xreg` with the `arima()` function:

```
stats::arima(yt, order = c(1, 0, 0), xreg = dt, include.mean = FALSE,
    method = "ML")
```

```
Call:
stats::arima(x = yt, order = c(1, 0, 0), xreg = dt, include.mean = FALSE, meth

Coefficients:
         ar1      dt
      0.9143  1.1023
s.e.  0.0384  0.0266

sigma^2 estimated as 0.09906:  log likelihood = -27.19,  aic = 60.39
```

Again the estimates are slightly different due to different treatment of the initial conditons.

4.4 Linear regression with AR(1) errors and independent errors

We can add some independent error to our model:

$$\begin{gathered} x_t = bx_{t-1} + w_t,\ w_t \sim \mathrm{N}(0,q) \\ y_t = \beta * d_t + x_t + v_t,,\ v_t \sim \mathrm{N}(0,r) \end{gathered} \tag{4.6}$$

We'll generate this data by adding independent error to `yt` from the previous example.

```r
yt.r <- yt + rnorm(TT, 0, sqrt(r))
```

We can fit as:

```r
R <- matrix("r")
mod.list <- list(B = B, U = U, R = R, D = D, d = d, A = A)
fit <- MARSS(yt.r, model = mod.list)
```

```
Success! abstol and log-log tests passed at 30 iterations.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.
```

```
MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Estimation converged in 30 iterations.
Log-likelihood: -71.13411
AIC: 152.2682   AICc: 152.9065

       Estimate
R.r      0.1172
B.b      0.9259
Q.Q      0.0747
x0.x0   -0.8778
D.beta   1.0974
Initial states (x0) defined at t=0

Standard errors have not been calculated.
Use MARSSparamCIs to compute CIs and bias estimates.
```

This is not a model that can be fit with `arima()`.

4.5 Linear regression with AR(1) driven by covariate

We can model a situation where the regression errors are autocorrelated but some of the variance is driven by a covariate. For example, good and bad 'years' are driven partially by, say, temperature, which we will model by `ct`. We will use an autocorrelated `ct` in the example, but it could be anything. How are autocorrelated errors different? There is memory in the errors. The `ct` in the past still affects the current error (w_t in this model).

$$\begin{gathered} x_t = bx_{t-1} + \beta * c_t + w_t,\ w_t \sim \text{N}(0, q) \\ y_t = x_t + v_t,\ v_t \sim \text{N}(0, r) \end{gathered} \tag{4.7}$$

Let's create some simulated data with this structure:

```
beta <- 1.1
x0 <- 0
ct <- arima.sim(n = TT, model = list(ar = 0.8), sd = sqrt(1))  # our covariat
ct <- as.vector(ct)
xt <- rep(x0, TT)
for (i in 2:TT) xt[i] <- b * xt[i - 1] + beta * ct[i] + rnorm(1,
    0, sqrt(q))
yt <- xt + rnorm(TT, 0, sqrt(r))
```

To fit this with `MARSS()`, we match up the model to the full MARSS model form:

$$\begin{gathered}\mathbf{x}_t = \mathbf{B}\mathbf{x}_{t-1} + \mathbf{C}\mathbf{c}_t + \mathbf{w}_t,\ \ \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \\ \mathbf{y}_t = \mathbf{x}_t + \mathbf{v}_t,\ \ \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t)\end{gathered} \tag{4.8}$$

The model list for `MARSS()` is:

```
R <- matrix("r")  # no v_t
C <- matrix("beta")
U <- matrix(0)  # no u
B <- matrix("b")
c <- matrix(ct, nrow = 1)
A <- matrix(0)
```

Now fit:

```
mod.list <- list(B = B, U = U, R = R, C = C, c = c, A = A)
fit <- MARSS(yt, model = mod.list)
```

```
Success! abstol and log-log tests passed at 18 iterations.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Estimation converged in 18 iterations.
Log-likelihood: -79.55739
AIC: 169.1148   AICc: 169.7531
```

```
       Estimate
R.r       0.100
B.b       0.900
Q.Q       0.135
x0.x0    -1.006
C.beta    1.120
Initial states (x0) defined at t=0

Standard errors have not been calculated.
Use MARSSparamCIs to compute CIs and bias estimates.
```

4.6 Flat level model

For the next examples, we will use the Nile river flow from 1871 to 1970, a data set in the **datasets** package.

```r
nile <- as.vector(datasets::Nile)
year <- as.vector(time(Nile))
```

The first model we will fit is a flat level model:

$$y_t = a + v_t, \;\; v_t \sim \text{N}(0, r) \tag{4.9}$$

where y_t is the river flow volume at year t and a is some constant average flow level (notice it has no t subscript).

To fit this model with MARSS, we explicitly show all the MARSS parameters.

$$\begin{gathered}
x_t = 1 \times x_{t-1} + 0 + w_t, \;\; w_t \sim \text{N}(0, 0) \\
y_t = 0 \times x_t + a + v_t, \;\; v_t \sim \text{N}(0, r) \\
x_0 = 0
\end{gathered} \tag{4.10}$$

The model list and fit for this equation is

```r
mod.list1 <- list(Z = matrix(0), A = matrix("a"), R = matrix("r"),
    B = matrix(1), U = matrix(0), Q = matrix(0), x0 = matrix(0))
fit1 <- MARSS(nile, model = mod.list1)
```

```
Success! abstol and log-log tests passed at 16 iterations.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Estimation converged in 16 iterations.
Log-likelihood: -654.5157
AIC: 1313.031   AICc: 1313.155

    Estimate
A.a      919
R.r    28352
Initial states (x0) defined at t=0

Standard errors have not been calculated.
Use MARSSparamCIs to compute CIs and bias estimates.
```

MARSS includes the state process x_t but we are setting $\mathbf{Z}$ to zero so that x_t does not appear in our observation model. We need to fix all the state parameters to zero so that the algorithm doesn't "chase its tail" trying to fit x_t to the data.

An equivalent way to write this model is to use x_t as the average flow level and make it be a constant level by setting $q = 0$. The average flow appears as the x_0 parameter. In MARSS form, this model is:

$$\begin{gathered} x_t = 1 \times x_{t-1} + 0 + w_t \text{ where } w_t \sim \text{N}(0, 0) \\ y_t = 1 \times x_t + 0 + v_t \text{ where } v_t \sim \text{N}(0, r) \\ x_0 = a \end{gathered} \tag{4.11}$$

The model list and fit for this equation is

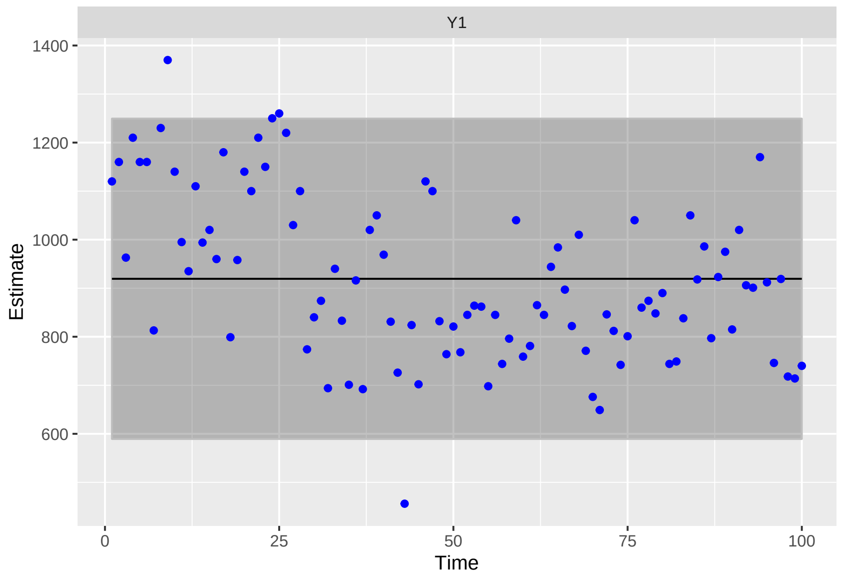
```r
mod.list2 <- list(Z = matrix(1), A = matrix(0), R = matrix("r"),
    B = matrix(1), U = matrix(0), Q = matrix(0), x0 = matrix("a"))
fit2 <- MARSS(nile, model = mod.list2)
```

```
Success! algorithm run for 15 iterations. abstol and log-log tests passed.
```

```
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Algorithm ran 15 (=minit) iterations and convergence was reached.
Log-likelihood: -654.5157
AIC: 1313.031   AICc: 1313.155

     Estimate
R.r     28352
x0.a      919
Initial states (x0) defined at t=0

Standard errors have not been calculated.
Use MARSSparamCIs to compute CIs and bias estimates.
```

The results are the same. We just formatted the model in different ways. We can plot the fitted model against the Nile river flow (blue dots) using `autoplot()`.

```
ggplot2::autoplot(fit2, plot.type = "observations")
```



4.7 Linear trend model

Looking at the data, we might expect that a declining average river flow would be better. In MARSS form, that model would be:

$$\begin{gathered}
x_t = 1 \times x_{t-1} + 0 + w_t,\ w_t \sim \text{N}(0, 0) \\
y_t = 0 \times x_t + a + \beta * t + v_t,\ v_t \sim \text{N}(0, r) \\
x_0 = 0
\end{gathered} \tag{4.12}$$

where t is the year and u is the average per-year decline in river flow volume.

The model list and fit for this equation is

```
mod.list1 <- list(Z = matrix(0), A = matrix("a"), R = matrix("r"),
    D = matrix("beta"), d = matrix(1:100, nrow = 1), B = matrix(1),
    U = matrix(0), Q = matrix(0), x0 = matrix(0))
fit1 <- MARSS(nile, model = mod.list1)
```

```
Success! algorithm run for 15 iterations. abstol and log-log tests passed.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Algorithm ran 15 (=minit) iterations and convergence was reached.
Log-likelihood: -642.3147
AIC: 1290.629   AICc: 1290.879

       Estimate
A.a     1056.42
R.r    22212.64
D.beta    -2.71
Initial states (x0) defined at t=0

Standard errors have not been calculated.
Use MARSSparamCIs to compute CIs and bias estimates.
```

We can also write this model as follows by modeling the trend with x_t:

$$\begin{gathered}
x_t = 1 \times x_{t-1} + u + w_t,\ w_t \sim \mathrm{N}(0,0) \\
y_t = 1 \times x_t + 0 + v_t,\ v_t \sim \mathrm{N}(0,r) \\
x_0 = a
\end{gathered} \tag{4.13}$$

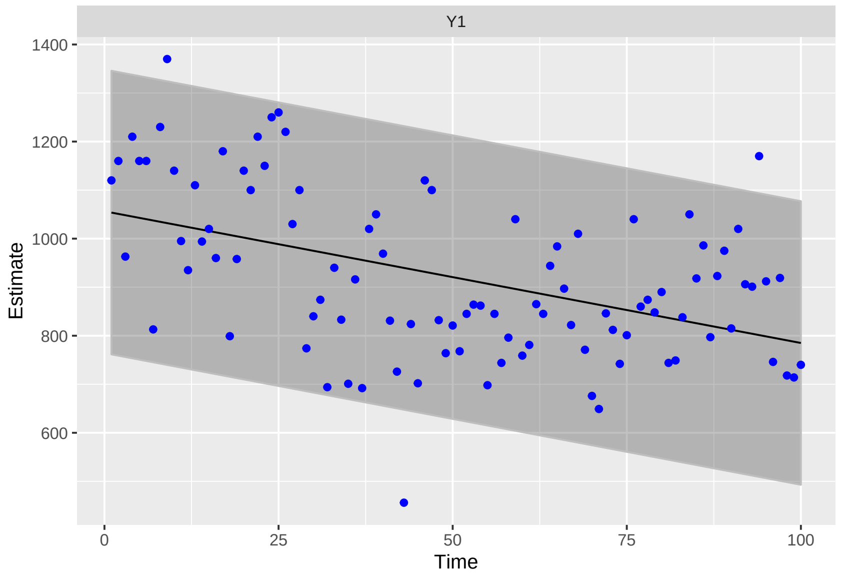
The model is specified as a list as follows. To fit, we need to force the algorithm to run a bit longer as it is showing convergence a bit early.

```
mod.list2 = list(Z = matrix(1), A = matrix(0), R = matrix("r"),
    B = matrix(1), U = matrix("u"), Q = matrix(0), x0 = matrix("a"))
fit2 <- MARSS(nile, model = mod.list2, control = list(minit = 30))
```

```
Success! algorithm run for 30 iterations. abstol and log-log tests passed.
Alert: conv.test.slope.tol is 0.5.
Test with smaller values (<0.1) to ensure convergence.

MARSS fit is
Estimation method: kem
Convergence test: conv.test.slope.tol = 0.5, abstol = 0.001
Algorithm ran 30 (=minit) iterations and convergence was reached.
Log-likelihood: -642.3147
AIC: 1290.629   AICc: 1290.879

     Estimate
R.r  22212.64
U.u     -2.71
x0.a  1056.37
Initial states (x0) defined at t=0

Standard errors have not been calculated.
Use MARSSparamCIs to compute CIs and bias estimates.
```

The fits are the same with either formulation of the model as long as we force the algorithm to run longer for the second form.

```
ggplot2::autoplot(fit2, plot.type = "observations")
```



4.8 Stochastic level model

We will now model the average river flow at year t as a random walk, specifically an autoregressive process which means that average river flow at year t is a function of average river flow in year $t-1$.

$$\begin{gathered} x_t = x_{t-1} + w_t \text{ where } w_t \sim \mathrm{N}(0,q) \\ y_t = x_t + v_t \text{ where } v_t \sim \mathrm{N}(0,r) \\ x_0 = \pi \end{gathered} \tag{4.14}$$

With all the MARSS parameters shown, the model is:

$$\begin{gathered} x_t = 1 \times x_{t-1} + 0 + w_t \text{ where } w_t \sim \mathrm{N}(0,q) \\ y_t = 1 \times x_t + 0 + v_t \text{ where } v_t \sim \mathrm{N}(0,r) \\ x_0 = \pi \end{gathered} \tag{4.15}$$

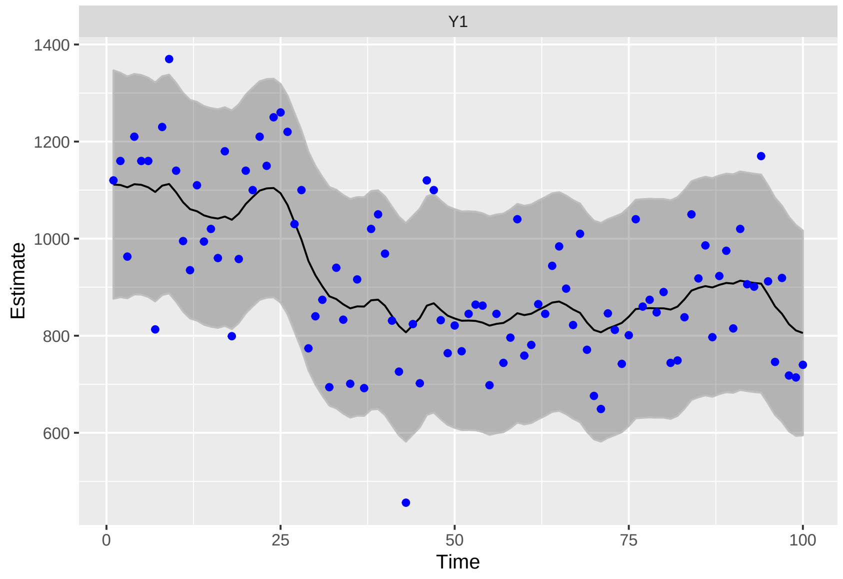
The model is specified as a list as follows. We can use the BFGS algorithm to ‘polish’ off the fit and get closer to the MLE. Why not just start with BFGS? First, it happens to take a long long time to fit and more importantly, the

BFGS algorith is sensitive to starting conditions and can catostrophically fail. In this case, it is slow but works fine. For some models, it does work better (faster and stable), but using the EM algorithm to get decent starting conditions for the BFGS algorithm is a common fitting strategy.

```
mod.list = list(Z = matrix(1), A = matrix(0), R = matrix("r"),
    B = matrix(1), U = matrix(0), Q = matrix("q"), x0 = matrix("pi"))
fit1 <- MARSS(nile, model = mod.list, silent = TRUE)
fit2 <- MARSS(nile, model = mod.list, inits = fit1, method = "BFGS")
```

```
Success! Converged in 12 iterations.
Function MARSSkfas used for likelihood calculation.

MARSS fit is
Estimation method: BFGS
Estimation converged in 12 iterations.
Log-likelihood: -637.7451
AIC: 1281.49   AICc: 1281.74

      Estimate
R.r      15337
Q.q       1218
x0.pi     1112
Initial states (x0) defined at t=0

Standard errors have not been calculated.
Use MARSSparamCIs to compute CIs and bias estimates.
```

```
ggplot2::autoplot(fit2, plot.type = "observations")
```



This is the same model fit in **?**, p. 148 except that we estimate x_1 as parameter rather than specifying x_1 via a diffuse prior. As a result, the log-likelihood value and **R** and **Q** are a little different than in **?**.

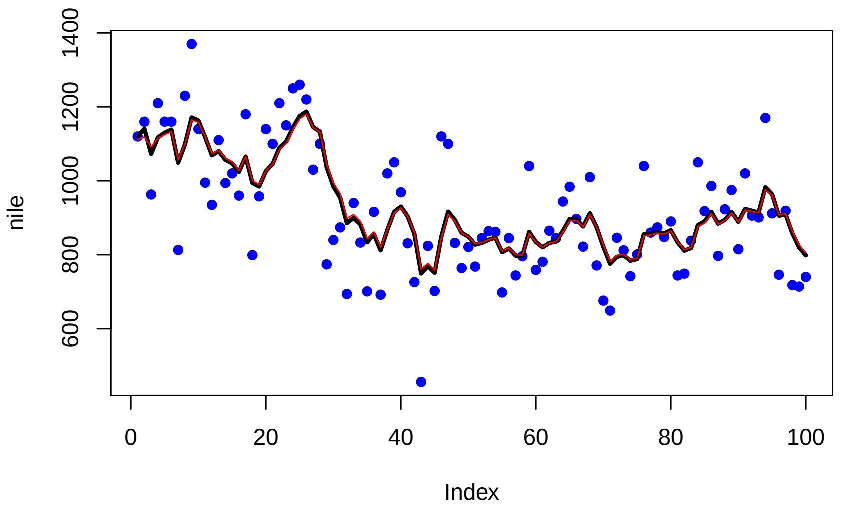
We can fit the Koopman model with `stats::StructTS()`. The estimates are slightly different since the initial conditions are treated differently.

```
fit.ts <- stats::StructTS(nile, type = "level")
fit.ts
```

```


Call:
stats::StructTS(x = nile, type = "level")

Variances:
  level  epsilon
   1469    15099
```

The fitted values returned by `fitted()` applied to a **StructTS** object are different than that returned by `fitted()` applied to a **marssMLE** object. The former returns $\hat{y}$ conditioned on the data up to time t, while the latter returns the $\hat{y}$ conditioned on all the data. If you want to compare use:

```
plot(nile, type = "p", pch = 16, col = "blue")
lines(fitted(fit.ts), col = "black", lwd = 3)
lines(MARSSkfss(fit2)$xtt[1, ], col = "red", lwd = 1)
```



The black line is the `StrucTS()` fit and the red line is the equivalent `MARSS()` fit.

4.9 Stochastic slope model

We can also model the β as a random walk:

$$\begin{gathered}\beta_t = \beta_{t-1} + w_t \text{ where } w_t \sim \text{N}(0,q)\\ y_t = a + \beta_t * t + v_t \text{ where } v_t \sim \text{N}(0,r)\\ x_0 = \pi\end{gathered} \tag{4.16}$$

The β_t is model with x_t. With all the MARSS parameters shown, the model is:

$$\begin{gathered}x_t = 1 \times x_{t-1} + 0 + w_t \text{ where } w_t \sim \text{N}(0,q)\\ y_t = t \times x_t + a + v_t \text{ where } v_t \sim \text{N}(0,r)\\ x_0 = \pi\end{gathered} \tag{4.17}$$

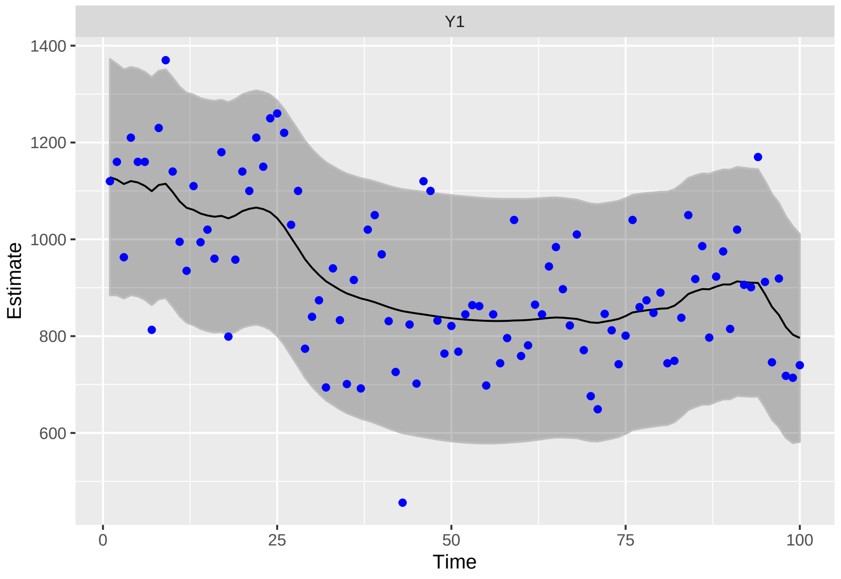
The trick here is to recognize that $\mathbf{Z}_t$, the matrix in front of $\mathbf{x}_t$ in the $\mathbf{y}_t$ equation, can be time-varying and can be fixed. In a time-varying matrix in **MARSS**, the time element is in the 3rd dimension. We are going to fix $\mathbf{Z}[1,1,t] = t$, where t is `year-mean(year)`. $\mathbf{Z}$ is a $1 \times 1 \times 100$ array. Demeaning the covariate stablizes the fitting. Try without demeaning to see the difference.

The model is specified as a list as follows.

```
Z <- array(0, dim = c(1, 1, length(nile)))
Z[1, 1, ] <- year - mean(year)
mod.list = list(Z = Z, A = matrix("a"), R = matrix("r"), B = matrix(1),
    U = matrix(0), Q = matrix("q"), x0 = matrix("pi"))
fit1 <- MARSS(nile, model = mod.list, silent = TRUE)
fit2 <- MARSS(nile, model = mod.list, inits = fit1, method = "BFGS")
```

```
Success! Converged in 13 iterations.
Function MARSSkfas used for likelihood calculation.

MARSS fit is
Estimation method: BFGS
Estimation converged in 13 iterations.
Log-likelihood: -636.6226
AIC: 1281.245   AICc: 1281.666

       Estimate
A.a     836.164
R.r   16835.484
Q.q       0.749
x0.pi    -5.905
Initial states (x0) defined at t=0

Standard errors have not been calculated.
Use MARSSparamCIs to compute CIs and bias estimates.
```

```
ggplot2::autoplot(fit2, plot.type = "observations")
```


Y1
1400
1200
1000
800
600
Estimate
0
25
50
75
100
Time

Part 3. Outputs

Part 3 discusses how to get outputs from **MARSS** fitted objects. Specifically:

- Estimated states
- Kalman filter and smoother output
- Residuals
- Confidence intervals
- Predictions
- Bootstrap resamples: parametric and innovations
- Simulated data

Chapter 5

MARSS outputs

MARSS models are used in many different ways and different users will want different types of output. Some users will want the parameter estimates while others want the smoothed states and others want to use MARSS models to interpolate missing values and want the expected values of missing data.

The best way to find out how to get output is to type `?print.MARSS` at the command line after installing the **MARSS** package. The print help page discusses how to get parameter estimates in different forms, the smoothed and filtered states, all the Kalman filter and smoother output, all the expectations of y (missing data), confidence intervals and bias estimates for the parameters, and standard errors of the states. If you are looking only for Kalman filter and smoother output, see the relevant section in Chapter **??** and see the help page for the `MARSSkf()` function (type `?MARSSkf` at the ,**R** command line).

You might also want to look at the `augment()`, `tidy()` and `glance()` functions which will summarize commonly needed output from a MARSS model fit. Type `?augment.marssMLE` at the command line to see examples. These functions work as they do in the **broom** R package.

Chapter 6

MARSS Residuals

Chapter 7

Confidence Intervals

Chapter 8

Predictions

Part 4. Tips and Tricks

Part 4 discusses troubleshooting, error messages, and tips to get MARSS models to fit quicker and better.

Chapter 9

Troubleshooting

Numerical errors due to ill-conditioned matrices are not uncommon when fitting MARSS models. The Kalman and EM algorithms need inverses of matrices. If those matrices become ill-conditioned, for example all elements are close to the same value, then the algorithm becomes unstable. Warning messages will be printed if the algorithms are becoming unstable and you can set `control$trace=1`, to see details of where the algorithm is becoming unstable. Whenever possible, you should avoid using shared $\boldsymbol{\pi}$ values in your model[1]. The way our algorithm deals with $\mathbf{\Lambda}$ tends to make this case unstable, especially if $\mathbf{R}$ is not diagonal. In general, estimation of a non-diagonal $\mathbf{R}$ is more difficult, more prone to ill-conditioning, and more data-hungry.

You may also see non-convergence warnings, especially if your MLE model turns out to be degenerate. This means that one of the elements on the diagonal of your $\mathbf{Q}$ or $\mathbf{R}$ matrix are going to zero (are degenerate). It will take the EM algorithm forever to get to zero. BFGS will have the same problem, although it will often get a bit closer to the degenerate solution. If you are using `method="kem"`, MARSS will warn you if it looks like the solution is degenerate. If you use `control=list(allow.degen=TRUE)`, the EM algorithm will attempt to set the degenerate variances to zero (instead of trying to get to zero using an infinite number of iterations). However, if one of the variances is going to zero, first think about why this is happening. This is typically caused by one of three problems: 1) you made a mistake in

[1]An example of a $\boldsymbol{\pi}$ with shared values is $\boldsymbol{\pi} = \left[\begin{smallmatrix} a \\ a \\ a \end{smallmatrix}\right]$.

inputting your data, e.g. used -99 as the missing value in your data but did not replace these with NAs before passing to MARSS, 2) your data are not sufficient to estimate multiple variances or 3) your data are inconsistent with the model you are trying fit.

The algorithms in the **MARSS** package are designed for cases where the $\mathbf{Q}$ and $\mathbf{R}$ diagonals are all non-minuscule. For example, the EM update equation for $\mathbf{u}$ will grind to a halt (not update $\mathbf{u}$) if $\mathbf{Q}$ is tiny (like 1E-7). Conversely, the BFGS equations are likely to miss the maximum-likelihood when $\mathbf{R}$ is tiny because then the likelihood surface becomes hyper-sensitive to $\boldsymbol{\pi}$. The solution is to use the degenerate likelihood function for the likelihood calculation and the EM update equations. **MARSS** will implement this automatically when $\mathbf{Q}$ or $\mathbf{R}$ diagonal elements are set to zero and will try setting $\mathbf{Q}$ and $\mathbf{R}$ terms to zero automatically if `control$allow.degen=TRUE`.

One odd case can occur when $\mathbf{R}$ goes to zero (a matrix of zeros), but you are estimating $\boldsymbol{\pi}$. If `model$tinitx=1`, then $\boldsymbol{\pi} = \mathbf{x}_1^0$ and $\mathbf{y}_1 - \mathbf{Z}\mathbf{x}_1^0$ can go to 0 as well as $\text{var}(\mathbf{y}_1 - \mathbf{Z}\mathbf{x}_1^0)$ by driving $\mathbf{R}$ to zero. But as this happens, the log-likelihood associated with $\mathbf{y}_1$ will go (correctly) to infinity and thus the log-likelihood goes to infinity. But if you set $\mathbf{R} = 0$, the log-likelihood will be finite. The reason is that $\mathbf{R} \approx 0$ and $\mathbf{R} = 0$ specify different likelihoods associated with $\mathbf{y}_1 - \mathbf{Z}\mathbf{x}_1^0$. With $\mathbf{R} = 0$, $\mathbf{y}_1 - \mathbf{Z}\mathbf{x}_1^0$ does not have a distribution; it is just a fixed value. So there is no likelihood to go to infinity. If some elements of the diagonal of $\mathbf{R}$ are going to zero, you should be suspect of the parameter estimates. Sometimes the structure of your data, e.g. one data value followed by a long string of missing values, is causing an odd spike in the likelihood at $\mathbf{R} \approx 0$. Try manually setting $\mathbf{R}$ equal to zero to get the correct log-likelihood[2].

[2]The likelihood returned when $\mathbf{R} \approx 0$ is not incorrect. It is just not the likelihood that you probably want. You want the likelihood where the $\mathbf{R}$ term is dropped because it is zero.

Chapter 10

EM algorithm

The **MARSS** package fits models via maximum likelihood. The MARSS package is unusual among packages for fitting MARSS models in that fitting is performed via a constrained EM algorithm (**?**) based on a vectorized form of Equation **??** (See Chapter **??** for the vectorized form used in the algorithm). Although fitting via the BFGS algorithm is also provided using `method="BFGS"` and the optim function in ,**R**, the examples in this guide use the EM algorithm primarily because it gives robust estimation for datasets replete with missing values and for high-dimensional models with various constraints. However, there are many models/datasets where BFGS is faster and we typically try both for problems. The EM algorithm is also often used to provide initial conditions for the BFGS algorithm (or an MCMC routine) in order to improve the performance of those algorithms. In addition to the main model fitting function, the MARSS package supplies functions for bootstrap and approximate confidence intervals, parametric and non-parametric bootstrapping, model selection (AIC and bootstrap AIC), simulation, and bootstrap bias correction.

10.1 Important notes about the algorithms

Specification of a properly constrained model with a unique solution is the responsibility of the user because MARSS has no way to tell if you have specified an insufficiently constrained model—with correspondingly an infinite number of solutions. How do you know if the model is properly constrained? If you are using a MARSS model form that is widely used, then you can probably assume that it is properly constrained. If you go to papers where someone developed the model or method, the issue of constraints necessary to ensure "identifiability" will likely be addressed if it is an issue. Are you fitting novel MARSS models? Then you will need to do some study on identifiability in this class of models using textbooks (Appendix **??**). Often textbooks do not address identifiability explicitly. Rather it is addressed implicitly by only showing a model constructed in such a way that it is identifiable. In our work, if we suspect identification problems, we will often first do a Bayesian analysis with flat priors and look for oddities in the posteriors, such as ridges, plateaus or bimodality.

All the EM code in the **MARSS** package is currently in native ,**R**. Thus the model fitting is slow. The classic Kalman filter/smoother algorithm, as shown in **?**, p. 331-335, is based on the original smoother presented in **?**. This Kalman filter is provided in function `MARSSkfss()`, but the default Kalman filter and smoother used in the **MARSS** package is based on the algorithm in **?** and papers by Koopman et al. This Kalman filter and smoother is provided in the **KFAS** package (Helske 2012). Table 2 in **?** indicates that the classic algorithm is 40-100 times slower than the algorithm given in **?**, **?**, and **?**. The MARSS package function `MARSSkfas()` provides a translator between the model objects in MARSS and those in **KFAS** so that the **KFAS** functions can be used. `MARSSkfas()` also includes a lag-one covariance smoother algorithm as this is not output by the **KFAS** functions, and it provides proper formulation of the priors so that one can use the **KFAS** functions when the prior on the states is set at $t = 0$ instead of $t = 1$. Simply off-setting your data to start at t=2 and sending that value to $t_{init} = 1$ in the **KFAS** Kalman filter would not be mathematically correct!

EM algorithms will quickly get in the vicinity of the maximum likelihood, but the final approach to the maximum is generally slow relative to quasi-Newton methods. On the flip side, EM algorithms are quite robust to initial conditions choices and can be extremely fast at getting close to the MLE values for high-dimensional models. The **MARSS** package also allows one to use the BFGS method to fit MARSS models, thus one can use an EM algorithm to "get close" and then the BFGS algorithm to polish off the estimate. Restricted maximum-likelihood algorithms are also available for AR(1) state-space models, both univariate (**?**) and multivariate (**?**). REML can give parameter estimates with lower variance than plain maximum-likelihood algorithms. However, the algorithms for REML when there are missing values are not currently available (although that will probably change in the near future). Another maximum-likelihood method is data-cloning which adapts MCMC algorithms used in Bayesian analysis for maximum-likelihood estimation (**?**).

Missing values are seamlessly accommodated with the **MARSS** package. Simply specify missing data with NAs. The likelihood computations are exact and will deal appropriately with missing values. However, no innovations, referring to the non-parametric bootstrap developed by Stoffer and Wall (1991), bootstrapping can be done if there are missing values. Instead parametric bootstrapping must be used.

You should be aware that maximum-likelihood estimates of variance in MARSS models are fundamentally biased, regardless of the algorithm used. This bias is more severe when one or the other of $\mathbf{R}$ or $\mathbf{Q}$ is very small, and the bias does not go to zero as sample size goes to infinity. The bias arises because variance is constrained to be positive. Thus if $\mathbf{R}$ or $\mathbf{Q}$ is essentially zero, the mean estimate will not be zero and thus the estimate will be biased high while the corresponding bias of the other variance will be biased low. You can generate unbiased variance estimates using a bootstrap estimate of the bias. The function `MARSSparamCIs()` will do this. However be aware that adding an estimated bias to a parameter estimate will lead to an increase in the variance of your parameter estimate. The amount of variance added will depend on sample size.

You should also be aware that mis-specification of the prior on the initial states ($\boldsymbol{\pi}$ and $\boldsymbol{\Lambda}$) can have catastrophic effects on your parameter estimates if your prior conflicts with the distribution of the initial states implied by the MARSS model. These effects can be very difficult to detect because the model will appear to be well-fitted. Unless you have a good idea of what the parameters should be, you might not realize that your prior conflicts.

The most common problems we have found with priors on $\mathbf{x}_0$ are the following. Problem 1) The correlation structure in $\boldsymbol{\Lambda}$ (whether the prior is diffuse or not) does not match the correlation structure in $\mathbf{x}_0$ implied by your model. For example, you specify a diagonal $\boldsymbol{\Lambda}$ (independent states), but the implied distribution has correlations. Problem 2) The correlation structure in $\boldsymbol{\Lambda}$ does not match the structure in $\mathbf{x}_0$ implied by constraints you placed on $\boldsymbol{\pi}$. For example, you specify that all values in $\boldsymbol{\pi}$ are shared, yet you specify that $\mathbf{\Lambda}$ is diagonal (independent).

Unfortunately, using a diffuse prior does not help with these two problems because the diffuse prior still has a correlation structure and can still conflict with the implied correlation in $\mathbf{x}_0$. One way to get around these problems is to set $\mathbf{\Lambda}$=0 (a $m \times m$ matrix of zeros) and estimate $\boldsymbol{\pi} \equiv \mathbf{x}_0$ only. Now $\boldsymbol{\pi}$ is a fixed but unknown (estimated) parameter, not the mean of a distribution. In this case, $\mathbf{\Lambda}$ does not exist in your model and there is no conflict with the model.
Be aware however that estimating $\boldsymbol{\pi}$ as a parameter is not always robust. If you specify that $\mathbf{\Lambda}$=0 and specify that $\boldsymbol{\pi}$ corresponds to $\mathbf{x}_0$, but your model "explodes" when run backwards in time, you cannot estimate $\boldsymbol{\pi}$ because you cannot get a good estimate of $\mathbf{x}_0$. Sometimes this can be avoided by specifying that $\boldsymbol{\pi}$ corresponds to $\mathbf{x}_1$ so that it can be constrained by the data $\mathbf{y}_1$.

In summary, if the implied correlation structure of your initial states is independent (diagonal variance-covariance matrix), you should generally be ok with a diagonal and high variance prior or with treating the initial states as parameters (with $\mathbf{\Lambda} = 0$). But if your initial states have an implied correlation structure that is not independent, then proceed with caution. 'With caution' means that you should assume you have problems and test how your model fits with simulated data.

10.2 State-space form of ARMA(p,q) models

There is a large class of models in the statistical finance literature that have the form

$$\begin{aligned}\mathbf{x}_{t+1} &= \mathbf{B}\mathbf{x}_t + \mathbf{\Gamma}\boldsymbol{\eta}_t\\ \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \boldsymbol{\eta}_t\end{aligned}$$

For example, ARMA(p,q) models can be written in this form. The MARSS model framework in this package will not allow you to write models in that form. You can put the $\boldsymbol{\eta}_t$ into the $\mathbf{x}_t$ vector and set $\mathbf{R} = 0$ to make models of this form using the MARSS form, but the EM algorithm in the MARSS package won't let you estimate parameters because the parameters will drop out of the full likelihood being maximized in the algorithm. You can try using BFGS by passing in the `method` argument to the MARSS() call.

Chapter 11

Other related packages

Packages that will do Kalman filtering and smoothing are many, but packages that estimate the parameters in a MARSS model, especially constrained MARSS models, are much less common. The following are those with which we are familiar, however there are certainly more packages for estimating MARSS models in engineering and economics of which we are unfamiliar. The **MARSS** package is unusual in that it uses an EM algorithm for maximizing the likelihood as opposed to a Newton-esque method (e.g. BFGS). The package is also unusual in that it allows you to specify the initial conditions at $t = 0$ or $t = 1$, allows degenerate models (with some of the diagonal elements of $\mathbf{R}$ or $\mathbf{Q}$ equal to zero). Lastly, model specification in the **MARSS** package has a one-to-one relationship between the model list in `MARSS()` and the model as you would write it on paper as a matrix equation. This makes the learning curve a bit less steep. However, the **MARSS** package has not been optimized for speed and probably will be really slow if you have time-series data with a lot of time points.

atsar atsar is an **R** package we wrote for fitting MARSS models using STAN. It allows fast and flexible fitting of MARSS models in a Bayesian framework. Our book from our time-series class has example applications Applied Time-Series Analysis for Fisheries and Environmental Sciences.

stats The **stats** package (part of base R) has functions for fitting univariate structural time series models (MARSS models with a univariate y). Read the help file at ‘?StructTS‘. The Kalman filter and smoother functions are described here: ‘?KalmanLike‘.

DLM DLM is an **R** package for fitting MARSS models. Our impression is that it is mainly Bayesian focused but it does allow MLE estimation via the ‘optim()‘ function. It has a book, *Dynamic Linear Models* with **R** by Petris et al., which has many examples of how to write MARSS models for different applications.

sspir sspir an **R** package for fitting ARSS (univariate) models with Gaussian, Poisson and binomial error distributions.

dse dse (Dynamic Systems Estimation) is an **R** package for multivariate Gaussian state-space models with a focus on ARMA models.

SsfPack SsfPack is a package for Ox/Splus that fits constrained multivariate Gaussian state-space models using mainly (it seems) the BFGS algorithm but the newer versions support other types of maximization. **SsfPack** is very flexible and written in C to be fast. It has been used extensively on statistical finance problems and is optimized for dealing with large (financial) data sets. It is used and documented in *Time Series Analysis by State Space Methods* by Durbin and Koopman, *An Introduction to State Space Time Series Analysis* by Commandeur and Koopman, and *Statistical Algorithms for Models in State Space Form: SsfPack 3.0*, by Koopman, Shephard, and Doornik.

Brodgar The Brodgar software was developed by Alain Zuur to do (among many other things) dynamic factor analysis, which involves a special type of MARSS model. The methods and many example analyses are given in *Analyzing Ecological Data* by Zuur, Ieno and Smith. This is the one package that we are aware of that also uses an EM algorithm for parameter estimation.

eViews **eViews** is a commercial economics software that will estimate at least some types of MARSS models.

KFAS The KFAS **R** package provides a fast Kalman filter and smoother. Examples in the package show how to estimate MARSS models using the **KFAS** functions and **R**’s ‘optim()‘ function. The **MARSS** package uses the filter and smoother functions from the **KFAS** package.

S+FinMetrics S+FinMetrics is a S-plus module for fitting MAR models, which are called vector autoregressive (VAR) models in the economics and finance literature. It has some support for state-space VAR models, though we haven't used it so are not sure which parameters it allows you to estimate. It was developed by Andrew Bruce, Doug Martin, Jiahui Wang, and Eric Zivot, and it has a book associated with it: *Modeling Financial Time Series with S-plus* by Eric Zivot and Jiahui Wang.

kftrack The kftrack **R** package provides a suite of functions specialized for fitting MARSS models to animal tracking data.

Bibliography