MARSS Package Manual

E. E. Holmes, M. D. Scheuerell, and E. J. Ward

2021-02-23

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, Scheuerell, 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.

Authors

The authors are research scientists with the US Federal government. This work was conducted as part of their jobs for NOAA Fisheries and USGS, as such the work is in the public domain and cannot be copyrighted.

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

- http://faculty.washington.edu/eeholmes
- https://faculty.washington.edu/scheuerl
- http://faculty.washington.edu/warde

Citation

Holmes, E. E., M.D. Scheuerell, E.J. Ward. MARSS Package Manual. Accessed 2021-02-23. https://nwfsc-timeseries.github.io/MARSS-Manual/

Holmes, E.E., E.J. Ward, and K. Wills. 2012. MARSS: multivariate autoregressive state-space models for analyzing time-series data. R Journal 4(1): 11-19. https://doi.org/10.32614/RJ-2012-002

Acknowledgments

Early development of the MARSS functions into an R package was done by Kellie Wills, a contractor at NWFSC. Students in the workshops on MAR and MARSS modeling and the course at University of Washington have helped us refine the package. The Kalman filter and smoother functions used by the MARSS package are now those written by Jouni Helske for the **KFAS** package. Use of the **KFAS** Kalman functions has made the **MARSS** package faster and more stable.

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. 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.

1.1 MARSS model form

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.

1.2 Examples of MARSS models

An unconstrained MARSS model, meaning all the elements in a parameter matrices are allowed to be different and none constrained to be equal or related.

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

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

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_0 \sim \text{MVN}\left(\begin{bmatrix} \pi_1 \\ \pi_2 \end{bmatrix}, \begin{bmatrix} \nu_{11} & \nu_{12} \\ \nu_{21} & \nu_{22} \end{bmatrix} \right) \quad \textit{or} \quad \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_1 \sim \text{MVN}\left(\begin{bmatrix} \pi_1 \\ \pi_2 \end{bmatrix}, \begin{bmatrix} \nu_{11} & \nu_{12} \\ \nu_{21} & \nu_{22} \end{bmatrix} \right) \tag{1.2}$$

A constrianed MARSS model. The MARSS package allows you to specify constraints by fixing elements in a parameter matrix or specifying that some elements are estimated—or have a linear relationship to other elements. Here is an example:

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

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

$$\begin{bmatrix}x_1\\x_2\end{bmatrix}_0 \sim \text{MVN}\left(\begin{bmatrix}\pi\\\pi\end{bmatrix}, \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\right) \tag{1.3}$$

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

Quick Start

If you already work with models in the form of Equation (1.1), 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.

The default MARSS model is:

$$\begin{gathered}
\mathbf{x}_t = \mathbf{B}_t\mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t\mathbf{c}_t + \mathbf{G}_t\mathbf{w}_t,\ \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \\
\mathbf{y}_t = \mathbf{Z}_t\mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t\mathbf{d}_t + \mathbf{H}_t\mathbf{v}_t,\ \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t) \\
\mathbf{x}_1 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda}) \text{ or } \mathbf{x}_0 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda})
\end{gathered} \tag{2.1}$$

c and **d** are inputs (aka, exogenous variables or covariates or indicator variables) and must have no missing values. They are not treated as 'data' in the likelihood but as inputs. In most cases, **G** and **H** are fixed (not estimated) and must have no missing values. $\mathbf{w}_t$ and $\mathbf{v}_t$ are uncorrelated.

The MARSS package is designed to handle linear constraints within the parameter matrices: **B**, **u**, **C**, **Q**, **Z**, **a**, **D**, **R**, $\boldsymbol{\pi}$, and $\boldsymbol{\Lambda}$ (and in limited situations **G** and **H**). Linear constraint means you can write the elements of the matrix as a linear equation of all the other elements.

Example: a mean-reverting random walk model with three observation time series:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t = \begin{bmatrix} b & 0 \\ 0 & b \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}_t, \quad \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} q_{11} & q_{12} \\ q_{12} & q_{22} \end{bmatrix} \right), \quad \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_0 \sim \text{MVN}\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix}_t = \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}_t, \quad \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}_t \sim MVN\left(\begin{bmatrix} a_1 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} r_{11} & 0 & 0 \\ 0 & r & 0 \\ 0 & 0 & r \end{bmatrix} \right)$$

To fit with `MARSS()`, we translate this model into equivalent matrices (or arrays if time-varying) in R. Matrices that combine fixed and estimated values are specified using a list matrix with numerical values for fixed values and character names for the estimated values.

```r
B1 <- matrix(list("b", 0, 0, "b"), 2, 2)
U1 <- matrix(0, 2, 1)
Q1 <- matrix(c("q11", "q12", "q12", "q22"), 2, 2)
Z1 <- matrix(c(1, 0, 1, 1, 1, 0), 3, 2)
A1 <- matrix(list("a1", 0, 0), 3, 1)
R1 <- matrix(list("r11", 0, 0, 0, "r", 0, 0, 0, "r"), 3, 3)
pi1 <- matrix(0, 2, 1)
V1 = diag(1, 2)
model.list <- list(B = B1, U = U1, Q = Q1, Z = Z1, A = A1, R = R1,
    x0 = pi1, V0 = V1, tinitx = 0)
```

Try printing these out and you will see the one-to-one correspondence between the model in R and the math version of the model. Matrix names in the model list must be `B`, `U`, `C`, `c`, `Q`, `Z`, `A`, `D`, `d`, `R`, `x0`, and `V0`, although in many cases you will use the default values and do not need to specify for the model list. The `tinitx` element tells `MARSS()` whether the initial state for x is at $t = 1$ (`tinitx=1`) or $t = 0$ (`tinitx=0`). The data must be entered as a $n \times T$ matrix, a ts object (which will be converted to a $n \times T$ matrix) or a vector (which will be converted to a $1 \times T$ matrxi). `MARSS()` has a number of text shortcuts for common parameter forms, such as "diagonal and unequal''.

The call to MARSS is

```r
fit <- MARSS(data, model=model.list)
```

The `R`, `Q` and `V0` variances can be set to zero to specify partially deterministic systems. This allows you to write MAR(p) models in MARSS form for example.

2.1 Linear constraints

Your model can have simple linear constraints within all the parameters except $\mathbf{Q}$, $\mathbf{R}$ and $\mathbf{\Lambda}$. For example $1 + 2a - 3b$ is a linear constraint. When entering this value for you matrix, you specify this as `‘‘1+2\emph{a+-3}b’’`. NOTE: $+$'s join parts so $+-3*b$ to specify $-3b$. Anything after `*` is a parameter. So `1*1` has a parameter called `"1"`. Example, let's change the $\mathbf{B}$ and $\mathbf{Q}$ matrices in the previous model to:

$$\mathbf{B} = \begin{bmatrix} b-0.1 & 0 \\ 0 & b+0.1 \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} q_{11} & 0 \\ 1 & 0 \end{bmatrix} \quad \mathbf{Z} = \begin{bmatrix} z_1 - z_2 & 2*z_1 \\ 0 & z_1 \\ z_2 & 0 \end{bmatrix}$$

This would be specified as (notice `"1*z1+-1*z2"` for `z1-z2`):

```
B1 <- matrix(list("-0.1+1*b", 0, 0, "0.1+1*b"), 2, 2)
Q1 <- matrix(list("q11", 0, 0, 1), 2, 2)
Z1 <- matrix(list("1*z1+-1*z2", 0, "z2", "2*z1", "z1", 0), 3,
    2)
model.list <- list(B = B1, U = U1, Q = Q1, Z = Z1, A = A1, R = R1,
    x0 = pi1, V0 = V1, tinitx = 0)
```

You can call `toLatex()` on your model to make sure you and `MARSS()` agree on what model you a trying to fit:

```
fit <- MARSS(data, model=model.list)
toLatex(fit$model)
```

2.2 Important

- 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.

- The code in the MARSS package is not particularly fast and EM algorithms are famously slow. You can try `method="BFGS"` and see if that is faster. For some models, it will be much faster and for others, much slower. "BFGS" can be quite sensitive to initial conditions. You can run EM a few iterations and then pass to "BFGS", and it will do better. `fit1 <- MARSS(data, model=model.list, control=list(minit=10, maxit=10))` `fit2 <- MARSS(data, model=model.list, method="BFGS", inits=fit1)`

2.3 Time-varying parameters and inputs

You can pass in an array of T matrices for a time-varying parameter (T is the number of time-steps in your data and is the 3rd dimension in the array):

$$\begin{gathered}
\mathbf{x}_t = \mathbf{B}_t\mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t\mathbf{c}_t + \mathbf{G}_t\mathbf{w}_t, \quad \mathbf{W}_t \sim \text{MVN}(0, \mathbf{Q}_t) \\
\mathbf{y}_t = \mathbf{Z}_t\mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t\mathbf{d}_t + \mathbf{H}_t\mathbf{v}_t, \quad \mathbf{V}_t \sim \text{MVN}(0, \mathbf{R}_t) \\
\mathbf{x}_{t_0} \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda})
\end{gathered} \tag{2.2}$$

Zeros are allowed on the diagonals of $\mathbf{Q}$, $\mathbf{R}$ and $\boldsymbol{\Lambda}$. NOTE(!!), the time indexing. Make sure you enter your arrays such that the right parameter (or input) at time t lines up with $\mathbf{x}_t$, e.g. it is common for state equations to have $\mathbf{B}_{t-1}$ lined up with $\mathbf{x}_t$ so you might need to enter the $\mathbf{B}$ array such that your $\mathbf{B}_{t-1}$ is entered at `Bt[,,t]` in the R code.

The length of the 3rd dimension must be the same as your data. For example, say in your mean-reverting random walk model (the example on the first page) you wanted $\mathbf{B}(2,2)$ to be one value before $t = 20$ and another value after but $\mathbf{B}(1,1)$ to be time constant. You can pass in the following:

```
TT <- dim(data)[2]
B1 <- array(list(),dim=c(2,2,TT))
B1[,,1:20] <- matrix(list("b",0,0,"b_1"),2,2)
B1[,,21:TT] <- matrix(list("b",0,0,"b_2"),2,2)
```

Notice the specification is one-to-one to your $\mathbf{B}_t$ matrices on paper.

2.4 Inputs

Inputs are specified in exactly the same manner. **C** and **D** are the estimated parameters and **c** and **d** are the inputs. Let's say you have temperature data and you want to include a linear effect of temperature that is different for each **x** time series:

```
C1 <- matrix(c("temp1","temp2"),2,1)
model.list <- list(B=B1,U=U1,C=C1,c=temp,Q=Q1,Z=Z1,A=A1,R=R1,x0=pi1,V0=V1,tinitx=0)
```

If you want a factor effect, then you'll need to recode your factor as a matrix with T columns and a row for each factor. Then you have 0 or 1 if that factor applies in time period t. **C** then has a column for each estimated factor effect. See the covariate chapters in the MARMES for examples.

2.5 Outputs

There are `plot`, `autoplot`, `print`, `summary`, `coef`, `fitted`, `residuals` and `predict` functions for marssMLE objects. See the man files, e.g. `?plot` to see examples.

2.6 Tips and Tricks

Use `plot(fit)` (or `autoplot(fit)`) to see a series of plots and diagnostics for your model. Try `MARSSinfo()` if you get errors you don't understand or fitting is taking a long time to converge. When fitting a model with `MARSS()`, pass in `silent=2` to see what `MARSS()` is doing. This puts it in verbose mode. Use `fit=FALSE` to set up a model without fitting. Let's say you do `fit <- MARSS(..., fit=FALSE)`. Now you can do `summary(fit$model)` to see what `MARSS()` thinks you are trying to fit. You can also try `toLatex(fit$model)` to make a LaTeX file and pdf version of your model (saved in the working directory). This loads the **Hmisc** package (and all its dependencies) and requires that you are able to process LaTeX files (e.g. you have the **tinytex** package).

Let's say you specified your model with some text shortcuts, like `Q="unconstrained"`, but you want the list matrix form for a next step. `a <- summary(fit$model)` returns that list (invisibly). Because the model argument of `MARSS()` will understand a list of list matrices, you can pass in `model=a` to specify the model. `MARSSkfas(fit, return.kfas.model=TRUE)` will return your model in KFAS form (class SSModel), thus you can use all the functions available in the KFAS package on your model.

Chapter 3

Data format

The first argument to `MARSS()` is your data, aka your response variables or the $\mathbf{y}$ in your MARSS equation.

```
MARSS(data, ...)
```

Your data need to be a matrix with time across the columns ($n \times T$ matrix). Note, you can pass in a **ts** object or a vector (not data frame) and `MARSS()` will convert this to a matrix. Here is an example of a data matrix with three observation time series and six time steps. Note that NAs are fine and it is not necessary for all observation time series to have observations at time step t.

Tip: put rownames on your data matrix and those will be used in the output.

$$\mathbf{y} = \begin{bmatrix} 1 & 2 & NA & NA & 3.2 & 8 \\ 2 & 5 & 3 & NA & 5.1 & NA \\ NA & NA & NA & 2.2 & NA & 7 \end{bmatrix}$$

where NA denotes a missing value. However, the **MARSS** functions assume discrete time steps and you will need a value 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 do.

3.1 ts objects

A `ts` object (time series object) stores information about the time steps of the data and often seasonal information (the quarter or month). If you pass in a `ts` object as data into `MARSS()`, it will convert this to a matrix but it will ignore any information about the seasonality. It will store this information, but it doesn't use it.

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 with the season information.

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)])
```

Chapter 4

Specifying the model

The argument `model` specifies the structure of the MARSS model. It is a list where the list elements for each model parameter specify the form of that parameter matrix.

```
MARSS(data, model=list(...))
```

There are text shortcuts for common matrix structures and most of our applications use one of these:

- `"identity"`, `"diagonal and equal"`, `"diagonal and unequal"`, `"unconstrained"`, `"equalvarcov"`, `"zero"`, `"unequal"`, `"equal"`
- Then there are some special shortcuts for $\mathbf{Z}$ and $\mathbf{a}$.

The default model structures are

- `Z="identity"` each y in $\mathbf{y}$ corresponds to one x in $\mathbf{x}$
- `B="identity"` no interactions among the x's in $\mathbf{x}$
- `U="unequal"` the u's in $\mathbf{u}$ are all different
- `Q="diagonal and unequal"` process errors are independent but have different variances
- `R="diagonal and equal"` the observations are i.i.d.
- `A="scaling"` $\mathbf{a}$ is a set of scaling factors
- `C="zero"` and `D="zero"` no inputs.

- `c="zero"` and `d="zero"` no inputs.
- `x0="unequal"` all initial states are different
- `V0="zero"` the initial condition on the states ($\mathbf{x}_0$ or $\mathbf{x}_1$) is fixed but unknown
- `tinitx=0` the initial state refers to $t = 0$ instead of $t = 1$.

The shortcuts and general specifications for each parameter are discussed below.

4.1 General matrix specification

However `MARSS()` is not limited to the standard forms. `MARSS()` will fit a general class of constrained MARSS models with linear constraints.

The most general way to specify model structure is to use a list matrix. The list matrix allows one to combine fixed and estimated elements in the parameter specification. It allows a one-to-one correspondence between how you write the parameter matrix on paper and how you specify it in R. For example, let's say $\mathbf{Q}$ and $\mathbf{u}$ have the following forms in your model:

$$\mathbf{Q} = \begin{bmatrix} q & 0 & 0 \\ 0 & q & 0 \\ 0 & 0 & 1 \end{bmatrix} \text{ and } \mathbf{u} = \begin{bmatrix} 0.05 \\ u_1 \\ u_2 \end{bmatrix}$$

So $\mathbf{Q}$ is a diagonal matrix with the 3rd variance fixed at 1 and the 1st and 2nd estimated and equal. The 1st element of $\mathbf{u}$ is fixed, and the 2nd and 3rd are estimated and different. You can specify this using a list matrix:

```
Q <- matrix(list("q", 0, 0, 0, "q", 0, 0, 0, 1), 3, 3)
U <- matrix(list(0.05, "u1", "u2"), 3, 1)
```

If you print out `Q` and `U`, you will see they look exactly like $\mathbf{Q}$ and $\mathbf{u}$ written above. MARSS will keep the fixed values fixed and estimate q, $u1$, and $u2$.

4.2 Linear constraints

The most general matrix specification is a list matrix with linear constraints. Your model can have simple linear constraints within all the parameters except $\mathbf{Q}$, $\mathbf{R}$ and $\mathbf{\Lambda}$. For example $1 + 2a - 3b$ is a linear constraint. When entering this value for you matrix, you specify this as `"1+2*a+-3*b"`. NOTE: $+$'s join parts so $+ - 3 * b$ to specify $-3b$. Anything after `*` is a parameter. So `1*1` has a parameter called `"1"`. Example, let's specify the following matrices:

$$\mathbf{B} = \begin{bmatrix} b-0.1 & 0 \\ 0 & b+0.1 \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} q_{11} & 0 \\ 1 & 0 \end{bmatrix} \quad \mathbf{Z} = \begin{bmatrix} z_1 - z_2 & 2 * z_1 \\ 0 & z_1 \\ z_2 & 0 \end{bmatrix}$$

This would be specified as (notice `"1*z1+-1*z2"` for `z1-z2`):

```r
B1 <- matrix(list("-0.1+1*b", 0, 0, "0.1+1*b"), 2, 2)
Q1 <- matrix(list("q11", 0, 0, 1), 2, 2)
Z1 <- matrix(list("1*z1+-1*z2", 0, "z2", "2*z1", "z1", 0), 3,
    2)
model.list <- list(B = B1, U = U1, Q = Q1, Z = Z1, A = A1, R = R1,
    x0 = pi1, V0 = V1, tinitx = 0)
```

4.3 Time-varying parameters

All parameters can be time-varying. Specify these with 3-dimensional array where time is the 3rd dimension. If the data have T time steps, the 3rd dimension of your array must be equal to T. No text short-cuts allowed.

Example, for t 1 to 3, u is

$$\begin{bmatrix} 0.01 \\ u_1 \\ u \end{bmatrix}$$

and for t 4 to 10, u is

$$\begin{bmatrix} 0.02 \\ u_2 \\ u \end{bmatrix}$$

Specify this as

```
U <- array(list(0), dim = c(3, 1, 10))
U[, , 1:3] <- list(0.01, "u1", "u")
U[, , 4:10] <- list(0.02, "u2", "u")
```

The **MARSS** algorithms will become rather slow when you use time-varying parameters. Note, dynamic linear models are a way to have stochastic time-varying parameters. See the MARMES chapter on this class of model.

4.4 Model structures

4.4.1 u, a and π

$\mathbf{u}$, $\mathbf{a}$ and $\boldsymbol{\pi}$ are all row matrices and the options for specifying their structures are the same. $\mathbf{a}$ has one special option, `"scaling"` described below. The allowable structures are shown using $\mathbf{u}$ as an example. Note that you should be careful about specifying shared structure in $\boldsymbol{\pi}$ because you need to make sure the structure in $\boldsymbol{\Lambda}$ matches. For example, if you require that all the $\boldsymbol{\pi}$ values are shared (equal) then $\boldsymbol{\Lambda}$ cannot be a diagonal matrix since that would be saying that the $\boldsymbol{\pi}$ values are independent, which they are clearly not if you force them to be equal.

- `U=matrix(list(),m,1)`: This is the most general form and allows one to specify fixed and estimated elements in $\mathbf{u}$. Each character string in $\mathbf{u}$ is the name of one of the $\mathbf{u}$ elements to be estimated. For example if `U=matrix(list(0.01,"u","u"),3,1)`, then $\mathbf{u}$ in the model has the following structure:
$$\begin{bmatrix} 0.01 \\ u \\ u \end{bmatrix}$$
- `U=matrix(c(),m,1)`, where the values in `c()` are all character strings: each character string is the name of an element to be estimated. For example if `U=matrix(c("u1","u1","u2"),3,1)`, then $\mathbf{u}$ in the model has the following structure:
$$\begin{bmatrix} u_1 \\ u_1 \\ u_2 \end{bmatrix}$$

with two values being estimated. `U=matrix(list("u1","u1","u2"),3,1)` has the same effect.

- `U="unequal"` or `U="unconstrained"`: Both of these stings indicate that each element of $\mathbf{u}$ is estimated. If $m = 3$, then $\mathbf{u}$ would have the form:
$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix}$$
- `U="equal"`: There is only one value in $\mathbf{u}$:
$$\begin{bmatrix} u \\ u \\ u \end{bmatrix}$$
- `U=matrix(c(),m,1)`, where the values in `c()` are all numerical values: $\mathbf{u}$ is fixed and has no estimated values. If `U=matrix(c(0.01,1,-0.5),3,1)`, then $\mathbf{u}$ in the model is:
$$\begin{bmatrix} 0.01 \\ 1 \\ -0.5 \end{bmatrix}$$
`U=matrix(list(0.01,1,-0.5),3,1)` would have the same effect.
- `U="zero"`: $\mathbf{u}$ is all zero:
$$\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

The $\mathbf{a}$ parameter has a special option, `"scaling"`, which is the default behavior. In this case, $\mathbf{a}$ is treated like a scaling parameter. If there is only one $\mathbf{y}$ row associated with an $\mathbf{x}$ row, then the corresponding $\mathbf{a}$ element is 0. If there are more than one $\mathbf{y}$ rows associated with an $\mathbf{x}$ row, then the first $\mathbf{a}$ element is set to 0 and the others are estimated. For example, say $m = 2$ and $n = 4$ and $\mathbf{Z}$ looks like the following:

$$\mathbf{Z} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Then the 1st-3rd rows of **y** are associated with the first row of **x**, and the 4th row of **y** is associated with the last row of **x**. Then if **a** is specified as `"scaling"`, **a** has the following structure:

$$\begin{bmatrix} 0 \\ a_1 \\ a_2 \\ 0 \end{bmatrix}$$

4.4.2 Q, R, Λ

The possible **Q**, **R**, and **Λ** model structures are identical, except that **R** is $n \times n$ while **Q** and **Λ** are $m \times m$. All types of structures can be specified using a list matrix, but there are also text shortcuts for specifying common structures. The structures are shown using **Q** as the example.

- `Q=matrix(list(),m,m)`: This is the most general way to specify the parameters and allows there to be fixed and estimated elements. Each character string in the list matrix is the name of one of the **Q** elements to be estimated, and each numerical value is a fixed value. For example if

```
Q=matrix(list("s2a",0,0,0,"s2a",0,0,0,"s2b"),3,3)
```

then **Q** has the following structure:

$$\begin{bmatrix} \sigma^2_a & 0 & 0 \\ 0 & \sigma^2_a & 0 \\ 0 & 0 & \sigma^2_b \end{bmatrix}$$

Note that `diag(c("s2a","s2a","s2b"))` will not have the desired effect of producing a matrix with numeric 0s on the off-diagonals. It will have character 0s and MARSS will interpret `"0"` as the name of an element of **Q** to be estimated. Instead, the following two lines can be used:

```
Q <- matrix(list(0), 3, 3)
diag(Q) = c("s2a", "s2a", "s2b")
```

- `Q="diagonal and equal"`: There is only one process variance value in this case:
$$\begin{bmatrix} \sigma^2 & 0 & 0 \\ 0 & \sigma^2 & 0 \\ 0 & 0 & \sigma^2 \end{bmatrix}$$

- `Q="diagonal and unequal"`: There are m process variance values in this case:
$$\begin{bmatrix} \sigma_1^2 & 0 & 0 \\ 0 & \sigma_2^2 & 0 \\ 0 & 0 & \sigma_3^2 \end{bmatrix}$$

- `Q="unconstrained"`: There are values on the diagonal and the off-diagonals of **Q** and the variances and covariances are all different:
$$\begin{bmatrix} \sigma_1^2 & \sigma_{1,2} & \sigma_{1,3} \\ \sigma_{1,2} & \sigma_2^2 & \sigma_{2,3} \\ \sigma_{1,3} & \sigma_{2,3} & \sigma_3^2 \end{bmatrix}$$
There are m process variances and $(m^2 - m)/2$ covariances in this case, so $(m^2 + m)/2$ values to be estimated. Note that variance-covariance matrices are never truly unconstrained since the upper and lower triangles of the matrix must be equal.

- `Q="equalvarcov"`: There is one process variance and one covariance:
$$\begin{bmatrix} \sigma^2 & \beta & \beta \\ \beta & \sigma^2 & \beta \\ \beta & \beta & \sigma^2 \end{bmatrix}$$

- `Q=matrix(c(), m, m)`, where all values in `c()` are character strings: Each element in **Q** is estimated and each character string is the name of a value to be estimated. Note if $m = 1$, you still need to wrap its value in `matrix()` so that its class is matrix. You must be careful that your **Q** specifies a statistically valid variance-covariance matrix. `MARSS()` will throw an error/warning telling you if it is not valid.

- `Q=matrix(c(), m, m)`, where all values in `c()` are numeric values: Each element in **Q** is fixed to the values in the matrix.
- `Q="identity"`: The **Q** matrix is the identity matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- `Q="zero"`: The **Q** matrix is all zeros:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

\end{itemize}

Be careful when setting $\mathbf{\Lambda}$ model structures. Mis-specifying the structure of $\mathbf{\Lambda}$ can have catastrophic, but difficult to discern, effects on your estimates. The default behavior for $\mathbf{\Lambda}$ is safe.

4.4.3 B

Like the variance-covariance matrices (**Q**, **R** and $\mathbf{\Lambda}$), **B** can be specified with a list matrix to allow you to have both fixed and shared elements in the **B** matrix. Character matrices and matrices with fixed values operate the same way as for the variance-covariance matrices. In addition, the same text shortcuts are available: `"unconstrained"`, `"identity"`, `"diagonal and equal"`, `"diagonal and unequal"`, `"equalvarcov"`, and `"zero"`. A fixed **B** can be specified with a numeric matrix, but all eigenvalues must fall within the unit circle; meaning `ll(abs(eigen(B)\$values)<=1)` must be true.

4.4.4 Z

Like **B** and the variance-covariance matrices, **Z** can be specified with a list matrix to allow you to have both fixed and estimated elements in **Z**. If **Z** is a square matrix, many of the same text shortcuts are available: `"diagonal` `and equal"`, `"diagonal and equal"`, and `"equalvarcov"`. If $\mathbf{Z}$ is a design matrix (a matrix with only 0s and 1s and where the row sums are all equal to 1), then a special shortcut is available using `factor()` which allows you to specify which $\mathbf{y}$ rows are associated with which $\mathbf{x}$ rows.

- `Z=factor(c(1,1,1))`: All $\mathbf{y}$ time series are observing the same (and only) hidden state trajectory x ($n = 3$ and $m = 1$):

$$\mathbf{Z} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

- `Z=factor(c(1,2,3))`: Each time series in $\mathbf{y}$ corresponds to a different hidden state trajectory. This is the default $\mathbf{Z}$ model and in this case $n = m$:

$$\mathbf{Z} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- `Z=factor(c(1,1,2))`: The first two time series in $\mathbf{y}$ corresponds to one hidden state trajectory and the third $\mathbf{y}$ time series corresponds to a different hidden state trajectory. Here $n = 3$ and $m = 2$:

$$\mathbf{Z} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

  The $\mathbf{Z}$ model can be specified using either numeric or character factor levels. `c(1,1,2)` is the same as `c("north","north","south")`.

- `Z="identity"`: This is the default behavior. This means $\mathbf{Z}$ is a $n \times n$ identity matrix and $m = n$. If $n = 3$, it is the same as `Z=factor(c(1,2,3))`.

- `Z=matrix(c(), n, m)`, where the elements in `c()` are all character strings: Passing in a $n \times m$ character matrix, means that each character string is a value to be estimated. Be careful that you are specifying an identifiable model when using this option.

- `Z=matrix(c(), n, m)`, where the elements in `c()` are all numeric: Passing in a $n \times m$ numeric matrix means that **Z** is fixed to the values in the matrix. The matrix must be numeric but it does not need to be a design matrix.

- `Z=matrix(list(), n, m)`: Passing in a $n \times m$ list matrix allows you to combine fixed and estimated values in the **Z** matrix. Be careful that you are specifying an identifiable model.

Chapter 5

Covariates format

The first argument to `MARSS()` is your data, aka your response variables or the $\mathbf{y}$ in your MARSS equation.

```
MARSS(data, model=list(c=..., d=...))
```

Your covariates must a matrix with time across the columns ($p \times T$ matrix) where p is the number of covariates. No NAs are allowed in covariates. See MARMES chapters dealing with missing values in your covariates.

Here is an example of a covariate matrix for 2 covariates.

$$\mathbf{y} = \begin{bmatrix} 1 & 2 & 5 & 7 & 3.2 & 8 \\ 2 & 5 & 3 & 8 & 5.1 & 10.2 \end{bmatrix}$$

5.1 Factor covariates

If your covariates are factors, like site number or month, and you are estimating a a value, i.e. level or intercept, for each then you will use a matrix with 0s and 1s. This is identical to how say `lm()` would translate your model with factors.

Say your covariate is quarters and you have 3 years of data:

```
covariate <- rep(paste0("q", 1:4), 3)
```

You translate this to a matrix with four rows and 10 columns. Each row is for a different quarter.

```
vals <- unique(covariate)
TT <- length(covariate)
p <- length(vals)
c <- matrix(0, p, TT)
for (i in 1:p) c[i, ] <- covariate == vals[i]
rownames(c) <- vals
c
```

```
   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
q1    1    0    0    0    1    0    0    0    1     0     0     0
q2    0    1    0    0    0    1    0    0    0     1     0     0
q3    0    0    1    0    0    0    1    0    0     0     1     0
q4    0    0    0    1    0    0    0    1    0     0     0     1
```

Part 2. Examples

Here a series of short examples are shown for different types of models specified in MARSS structure. Output is shown briefly.

The MARSS model has the form:

$$\begin{gathered}
\mathbf{x}_t = \mathbf{B}_t\mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t\mathbf{c}_t + \mathbf{G}_t\mathbf{w}_t,\ \text{where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \\
\mathbf{y}_t = \mathbf{Z}_t\mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t\mathbf{d}_t + \mathbf{H}_t\mathbf{v}_t,\ \text{where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t) \\
\mathbf{x}_1 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda}) \text{ or } \mathbf{x}_0 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda})
\end{gathered} \tag{5.1}$$

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.

5.1.1 Output

This is a brief summary of outputs. The output are shown with the following simple fit:

```
dat <- cumsum(rnorm(100, 0, 0.5)) + rnorm(100, 0, 0.5)
fit <- MARSS(dat)
```

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.

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

```
   term    estimate  std.error     conf.low   conf.up
1   R.R  0.12727996 0.06250867  0.004765227 0.2497947
2   U.U -0.09669412 0.06244193 -0.219078047 0.0256898
3   Q.Q  0.38394414 0.10885937  0.170583701 0.5973046
4 x0.x0  0.01028019 0.70069260 -1.363052072 1.3836125
```

The get the estimated states use:

```
head(stats::tsSmooth(fit, type = "xtT"))
```

```
  .rownames t   .estimate       .se
1      X.Y1 1 -0.08649281 0.2825700
2      X.Y1 2  0.20588708 0.2886156
3      X.Y1 3  0.09379051 0.2888742
4      X.Y1 4 -0.46718796 0.2888854
5      X.Y1 5 -1.30803192 0.2888859
6      X.Y1 6 -1.16070018 0.2888859
```

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`.

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`.

To get the fitted values, the estimated y, using all the data:

```
head(fitted(fit, type = "ytT"))
```

```
  .rownames t          y     .fitted
1        Y1 1 -0.2154995 -0.08649281
2        Y1 2  0.3399737  0.20588708
3        Y1 3  0.2425978  0.09379051
4        Y1 4 -0.3744107 -0.46718796
5        Y1 5 -1.6356185 -1.30803192
6        Y1 6 -1.1682209 -1.16070018
```

To get the fitted values, the estimated y, using the data up to time $t-1$:

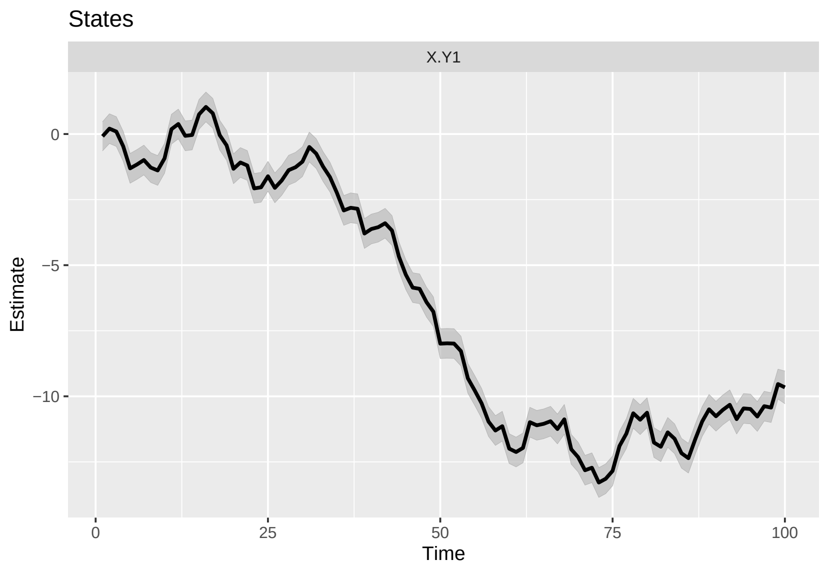
```
head(fitted(fit, type = "ytt1"))
```

```
  .rownames t          y     .fitted
1        Y1 1 -0.2154995 -0.08641393
2        Y1 2  0.3399737 -0.28005507
3        Y1 3  0.2425978  0.11322799
4        Y1 4 -0.3744107  0.11898963
5        Y1 5 -1.6356185 -0.36849468
6        Y1 6 -1.1682209 -1.46879885
```

Note, this is the default for `fitted(fit)`.

To make some generic plots and diagnostic plots, you can use `autoplot()` in the **ggplot2** package:

```
ggplot2::autoplot(fit, plot.type = "xtT")
```



To get residuals, use:

```
residuals(fit)
```

```
    type .rownames  name   t       value      .fitted      .resids    .sigma
1   ytt1         Y1 model   1  -0.2154995  -0.08641393 -0.129085584 0.7149994
2   ytt1         Y1 model   2   0.3399737  -0.28005507  0.620028721 0.7789833
3   ytt1         Y1 model   3   0.2425978   0.11322799  0.129369770 0.7821809
4   ytt1         Y1 model   4  -0.3744107   0.11898963 -0.493400373 0.7823202
5   ytt1         Y1 model   5  -1.6356185  -0.36849468 -1.267123868 0.7823262
6   ytt1         Y1 model   6  -1.1682209  -1.46879885  0.300577901 0.7823264
7   ytt1         Y1 model   7  -0.8385996  -1.32742386  0.488824261 0.7823264
8   ytt1         Y1 model   8  -1.3389574  -1.03695060 -0.302006792 0.7823264
9   ytt1         Y1 model   9  -1.5782833  -1.37284557 -0.205437766 0.7823264
10  ytt1         Y1 model  10  -1.1400991  -1.63225420  0.492155147 0.7823264
11  ytt1         Y1 model  11   0.4873179  -1.33914275  1.826460616 0.7823264
12  ytt1         Y1 model  12   0.5985266   0.01078931  0.587737330 0.7823264
13  ytt1         Y1 model  13  -0.2295780   0.37960547 -0.609183501 0.7823264
14  ytt1         Y1 model  14  -0.2864466  -0.19958512 -0.086861503 0.7823264
15  ytt1         Y1 model  15   0.9178574  -0.36507685  1.282934274 0.7823264
16  ytt1         Y1 model  16   1.2088933   0.55436170  0.654531573 0.7823264
```

```
17  ytt1         Y1 model  17   0.9846100   0.97608145  0.008528545 0.7823264
18  ytt1         Y1 model  18  -0.1898469   0.88614226 -1.075989186 0.7823264
19  ytt1         Y1 model  19  -0.2635590  -0.06277616 -0.200782882 0.7823264
20  ytt1         Y1 model  20  -1.7014612  -0.31849795 -1.382963269 0.7823264
21  ytt1         Y1 model  21  -0.9673000  -1.51055151  0.543251469 0.7823264
22  ytt1         Y1 model  22  -0.9462215  -1.17696983  0.230748311 0.7823264
23  ytt1         Y1 model  23  -2.3756943  -1.09090253 -1.284791742 0.7823264
24  ytt1         Y1 model  24  -2.1572390  -2.20520051  0.047961512 0.7823264
25  ytt1         Y1 model  25  -1.3217978  -2.26390729  0.942109490 0.7823264
26  ytt1         Y1 model  26  -2.2941688  -1.61441491 -0.679753923 0.7823264
27  ytt1         Y1 model  27  -1.8050204  -2.24949996  0.444479587 0.7823264
28  ytt1         Y1 model  28  -1.2839263  -1.99414936  0.710223075 0.7823264
29  ytt1         Y1 model  29  -1.2978006  -1.52831983  0.230519253 0.7823264
30  ytt1         Y1 model  30  -1.1713467  -1.44243395  0.271087221 0.7823264
31  ytt1         Y1 model  31  -0.2242015  -1.32441669  1.100215235 0.7823264
32  ytt1         Y1 model  32  -0.6649617  -0.54969856 -0.115263154 0.7823264
33  ytt1         Y1 model  33  -1.2735559  -0.73768548 -0.535870395 0.7823264
34  ytt1         Y1 model  34  -1.5802520  -1.25880930 -0.321442690 0.7823264
35  ytt1         Y1 model  35  -2.2150017  -1.61009825 -0.604903423 0.7823264
36  ytt1         Y1 model  36  -3.1703643  -2.18589885 -0.984465435 0.7823264
37  ytt1         Y1 model  37  -2.7653248  -3.06232699  0.297002182 0.7823264
38  ytt1         Y1 model  38  -2.5495997  -2.92378410  0.374184451 0.7823264
39  ytt1         Y1 model  39  -4.1631004  -2.72410993 -1.438990473 0.7823264
40  ytt1         Y1 model  40  -3.5869050  -3.96053916  0.373634114 0.7823264
41  ytt1         Y1 model  41  -3.5781022  -3.76130087  0.183198715 0.7823264
42  ytt1         Y1 model  42  -3.2583273  -3.71289465  0.454567392 0.7823264
43  ytt1         Y1 model  43  -3.4429161  -3.44955413  0.006638001 0.7823264
44  ytt1         Y1 model  44  -4.7774380  -3.54099070 -1.236447306 0.7823264
45  ytt1         Y1 model  45  -5.4241711  -4.61699805 -0.807173027 0.7823264
46  ytt1         Y1 model  46  -6.0050026  -5.35300387 -0.651998774 0.7823264
47  ytt1         Y1 model  47  -5.7476859  -5.96610578  0.218419932 0.7823264
48  ytt1         Y1 model  48  -6.4560010  -5.88980302 -0.566197930 0.7823264
49  ytt1         Y1 model  49  -6.4918749  -6.43494741 -0.056927514 0.7823264
50  ytt1         Y1 model  50  -8.4004432  -6.57673028 -1.823712965 0.7823264
51  ytt1         Y1 model  51  -7.9718483  -8.11787434  0.146026081 0.7823264
52  ytt1         Y1 model  52  -7.8974521  -8.09891026  0.201458155 0.7823264
53  ytt1         Y1 model  53  -8.0290019  -8.03604187  0.007040014 0.7823264
54  ytt1         Y1 model  54  -9.5008291  -8.12716003 -1.373669111 0.7823264
```

```
55  ytt1         Y1 model  55  -9.7698406  -9.31185226 -0.457988303 0.7823264
56  ytt1         Y1 model  56 -10.2019336  -9.77129051 -0.430643077 0.7823264
57  ytt1         Y1 model  57 -11.0788592 -10.20907031 -0.869788901 0.7823264
58  ytt1         Y1 model  58 -11.4726771 -10.99467028 -0.478006850 0.7823264
59  ytt1         Y1 model  59 -10.8015780 -11.46996398  0.668385934 0.7823264
60  ytt1         Y1 model  60 -12.2360753 -11.03727105 -1.198804202 0.7823264
61  ytt1         Y1 model  61 -12.2184312 -12.08346363 -0.134967605 0.7823264
62  ytt1         Y1 model  62 -12.2403544 -12.28705722  0.046702855 0.7823264
63  ytt1         Y1 model  63 -10.6220609 -12.34676091  1.724700035 0.7823264
64  ytt1         Y1 model  64 -11.1634874 -11.07742709 -0.086060353 0.7823264
65  ytt1         Y1 model  65 -11.0634075 -11.24228428  0.178876829 0.7823264
66  ytt1         Y1 model  66 -10.8186864 -11.19730116  0.378614731 0.7823264
67  ytt1         Y1 model  67 -11.4676432 -10.99411803 -0.473525120 0.7823264
68  ytt1         Y1 model  68 -10.3759729 -11.46586204  1.089889099 0.7823264
69  ytt1         Y1 model  69 -12.2988526 -10.69932259 -1.599530016 0.7823264
70  ytt1         Y1 model  70 -12.2516741 -12.06290524 -0.188768814 0.7823264
71  ytt1         Y1 model  71 -13.0164547 -12.30911143 -0.707343256 0.7823264
72  ytt1         Y1 model  72 -12.5042312 -12.96604828  0.461817032 0.7823264
73  ytt1         Y1 model  73 -13.5216286 -12.69696577 -0.824662850 0.7823264
74  ytt1         Y1 model  74 -13.1967017 -13.44682419  0.250122507 0.7823264
75  ytt1         Y1 model  75 -13.0577707 -13.34541178  0.287641072 0.7823264
76  ytt1         Y1 model  76 -11.7387135 -13.21428325  1.475569785 0.7823264
77  ytt1         Y1 model  77 -11.5239229 -12.14227006  0.618347131 0.7823264
78  ytt1         Y1 model  78 -10.3077744 -11.74920976  1.441435363 0.7823264
79  ytt1         Y1 model  79 -11.0638032 -10.70423233 -0.359570850 0.7823264
80  ytt1         Y1 model  80 -10.1581558 -11.08572022  0.927564457 0.7823264
81  ytt1         Y1 model  81 -12.0865411 -10.44774806 -1.638792996 0.7823264
82  ytt1         Y1 model  82 -12.1580520 -11.84242848 -0.315623553 0.7823264
83  ytt1         Y1 model  83 -11.1118809 -12.18910844  1.077227495 0.7823264
84  ytt1         Y1 model  84 -11.5166961 -11.43259747 -0.084098585 0.7823264
85  ytt1         Y1 model  85 -12.2905423 -11.59590087 -0.694641424 0.7823264
86  ytt1         Y1 model  86 -12.6624280 -12.24277738 -0.419650618 0.7823264
87  ytt1         Y1 model  87 -11.6120454 -12.67185073  1.059805307 0.7823264
88  ytt1         Y1 model  88 -10.8890452 -11.92913879  1.040093554 0.7823264
89  ytt1         Y1 model  89 -10.2550242 -11.20203931  0.947015086 0.7823264
90  ytt1         Y1 model  90 -10.9288298 -10.54866152 -0.380168256 0.7823264
91  ytt1         Y1 model  91 -10.5006472 -10.94646334  0.445816109 0.7823264
92  ytt1         Y1 model  92 -10.0770883 -10.69005416  0.612965818 0.7823264
```

```
93  ytt1        Y1 model  93 -11.1879057 -10.30125608 -0.886649649 0.7823264
94  ytt1        Y1 model  94 -10.3191358 -11.10021040  0.781074558 0.7823264
95  ytt1        Y1 model  95 -10.3952711 -10.57826380  0.182992689 0.7823264
96  ytt1        Y1 model  96 -10.9967755 -10.53002076 -0.466754750 0.7823264
97  ytt1        Y1 model  97 -10.2274072 -10.99640237  0.768995193 0.7823264
98  ytt1        Y1 model  98 -10.7452618 -10.48402309 -0.261238732 0.7823264
99  ytt1        Y1 model  99  -9.1927649 -10.78762821  1.594863342 0.7823264
100 ytt1        Y1 model 100  -9.6690609  -9.62112999 -0.047930952 0.7823264
     .std.resids
1   -0.180539437
2    0.795946057
3    0.165396224
4   -0.630688565
5   -1.619687428
6    0.384210334
7    0.624834127
8   -0.386036793
9   -0.262598519
10   0.629091794
11   2.334652784
12   0.751268646
13  -0.778681973
14  -0.111029742
15   1.639896338
16   0.836647638
17   0.010901517
18  -1.375371101
19  -0.256648465
20  -1.767757278
21   0.694405094
22   0.294951439
23  -1.642270625
24   0.061306265
25   1.204240883
26  -0.868887824
27   0.568151043
28   0.907834676
29   0.294658649
```

```
30   0.346514197
31   1.406337777
32  -0.147333834
33  -0.684970319
34  -0.410880511
35  -0.773211012
36  -1.258381894
37   0.379639706
38   0.478297075
39  -1.839373423
40   0.477593613
41   0.234171702
42   0.581045668
43   0.008484950
44  -1.580474893
45  -1.031759863
46  -0.833410115
47   0.279192827
48  -0.723736150
49  -0.072766956
50  -2.331140631
51   0.186656199
52   0.257511626
53   0.008998819
54  -1.755877124
55  -0.585418408
56  -0.550464680
57  -1.111797901
58  -0.611006891
59   0.854356819
60  -1.532358017
61  -0.172520826
62   0.059697401
63   2.204578464
64  -0.110005680
65   0.228647300
66   0.483960031
67  -0.605278171
```

```
68   1.393138509
69  -2.044581292
70  -0.241291618
71  -0.904153579
72   0.590312439
73  -1.054116033
74   0.319716287
75   0.367673973
76   1.886130517
77   0.790395280
78   1.842498576
79  -0.459617404
80   1.185648858
81  -2.094768755
82  -0.403442265
83   1.376953956
84  -0.107498072
85  -0.887917604
86  -0.536413693
87   1.354684239
88   1.329487913
89   1.210511405
90  -0.485945807
91   0.569859438
92   0.783516678
93  -1.133349963
94   0.998399788
95   0.233908352
96  -0.596624022
97   0.982959476
98  -0.333925477
99   2.038616168
100           NA
```

There are many types of residuals possible for MARSS models. The default `residuals(fit)` will return the innovations residuals, which are what are typically used for residuals diagnostics for state-space models. However `residuals()` and the companion `MARSSresiduals()` will return all possible residuals for your model. Read about MARSS residuals in `?residuals.marssMLE`.

To get your parameter estimates, use:

```
coef(fit)
```

```
$Z
     [,1]

$A
     [,1]

$R
     [,1]
R 0.12728

$B
     [,1]

$U
         [,1]
U -0.09669412

$Q
       [,1]
Q 0.3839441

$x0
         [,1]
x0 0.01028019

$V0
     [,1]

$G
     [,1]
```

```
$H
     [,1]

$L
     [,1]

$C
     [,1]

$D
     [,1]

$c
     [,1]

$d
     [,1]
```

Type `?coef.marssMLE` to see the different forms that you can get your estimated parameters in. A common form is as a matrix. To get that use `coef(fit, type="matrix")`.

Chapter 6

Univariate Models

These models can be written in the form:

$$\begin{aligned} x_t &= b_t x_{t-1} + u_t + \beta_t c_t + w_t,\ w_t \sim \mathrm{N}(0, q_t) \\ y_t &= z_t x_t + a_t + \psi_t d_t + v_t,\ v_t \sim \mathrm{N}(0, r_t) \end{aligned} \tag{6.1}$$

where c_t and d_t are inputs (no missing values) and y_t are observations (missing values allowed). x_t are the states. Everything else is an estimated parameter. Parameters can be time-varying. If time-varying, then specified as a 3D array with time in the 3rd dimension.

6.1 Random walk with drift

A univariate random walk with drift observed with error is:

$$\begin{aligned} x_t &= x_{t-1} + u + w_t,\ w_t \sim \mathrm{N}(0, q) \\ y_t &= x_t + v_t,\ v_t \sim \mathrm{N}(0, r) \end{aligned} \tag{6.2}$$

Create some data from this equation:

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

Fit with `MARSS()`:

```
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   AICc: 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.
```

Get confidence intervals with `tidy()`:

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

```
   term    estimate   std.error      conf.low    conf.up
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:

```
head(stats::tsSmooth(fit, type = "xtT"))
```

```
  .rownames t  .estimate       .se
1      X.Y1 1 -0.2246253 0.1098830
2      X.Y1 2 -0.1370475 0.1118654
3      X.Y1 3  0.1814517 0.1119369
4      X.Y1 4  0.2708522 0.1119395
5      X.Y1 5  0.3617262 0.1119396
6      X.Y1 6  0.7986076 0.1119396
```

The get the fitted values (note these are the smoothed fitted values conditioned on all the data):

```
head(fitted(fit, type = "ytT"))
```

```
  .rownames t          y    .fitted
1        Y1 1 -0.2382786 -0.2246253
2        Y1 2 -0.2043381 -0.1370475
3        Y1 3  0.2482112  0.1814517
4        Y1 4  0.2704228  0.2708522
5        Y1 5  0.2608996  0.3617262
6        Y1 6  0.9039099  0.7986076
```

Read up on fitted values for MARSS models at `?fitted.marssMLE`.

6.2 Random walk with time-varying parameters

Let's fit a random walk where the first 50 time steps have one process variance and the rest of the time series has a different process variance. The model is

$$\begin{gathered} x_t = x_{t-1} + u + w_t,\ \ w_t \sim \mathrm{N}(0, q_t) \\ y_t = x_t + v_t,\ \ v_t \sim \mathrm{N}(0, r) \end{gathered} \tag{6.3}$$

Create some data:

```
set.seed(123)
TT <- 100
sT <- 50
u <- 0.01
r <- 0.01
q <- c(rep(0.1, sT), rep(0.2, TT - sT))
yt <- cumsum(rnorm(TT, u, sqrt(q))) + rnorm(TT, 0, sqrt(r))
```

Fit with MARSS:

```
Q <- array(0, dim = c(1, 1, TT))
Q[1, 1, ] <- c(rep("q1", sT), rep("q2", TT - sT))
fit <- MARSS(yt, model = list(Q = Q), silent = TRUE)
broom::tidy(fit)
```

```
   term    estimate  std.error     conf.low    conf.up
1   R.R  0.02103312 0.01287598 -0.004203342 0.04626958
2   U.U  0.04315283 0.02999130 -0.015629036 0.10193470
3  Q.q1  0.06849550 0.02514959  0.019203209 0.11778779
4  Q.q2  0.12424866 0.03710242  0.051529256 0.19696806
5 x0.x0 -0.26716186 0.29456436 -0.844497393 0.31017367
```

We can do the same with the drift term, u. We have one u for the first 50 time steps and another for the last 50. Let's fit a random walk where the first 50 time steps have one process variance and the rest of the time series has a different process variance. The model is

$$\begin{gathered} x_t = x_{t-1} + u + w_t, \;\; w_t \sim \text{N}(0, q_t) \\ y_t = x_t + v_t, \;\; v_t \sim \text{N}(0, r) \end{gathered} \tag{6.4}$$

Create some data:

```
set.seed(123)
q <- 0.1
u <- c(rep(0.1, sT), rep(-0.1, TT - sT))
yt <- cumsum(rnorm(TT, u, sqrt(q))) + rnorm(TT, 0, sqrt(r))
```

Fit with MARSS:

```
U <- array(0, dim = c(1, 1, TT))
U[1, 1, ] <- c(rep("u1", sT), rep("u2", TT - sT))
fit <- MARSS(yt, model = list(U = U), silent = TRUE)
broom::tidy(fit)
```

```
   term    estimate   std.error      conf.low    conf.up
1   R.R  0.01878371 0.009873106 -0.0005672203 0.03813464
2  U.u1  0.11495619 0.035842744  0.0447057067 0.18520668
3  U.u2 -0.05358255 0.035480829 -0.1231236961 0.01595860
4   Q.Q  0.06237204 0.017600922  0.0278748647 0.09686921
5 x0.x0 -0.24493471 0.281911468 -0.7974710376 0.30760161
```

6.3 AR(1) observed with error

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

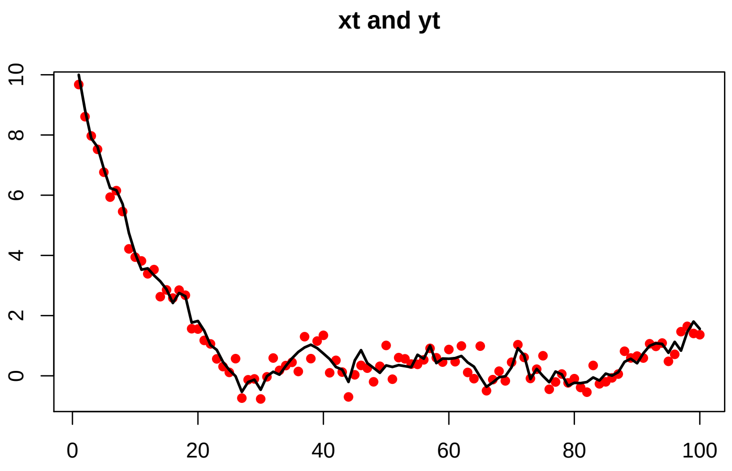
$$\begin{gathered} x_t = bx_{t-1} + u + w_t,\ w_t \sim \mathrm{N}(0, q) \\ y_t = x_t + v_t,\ w_t \sim \mathrm{N}(0, r) \end{gathered} \tag{6.5}$$

Create some simulated, non-stationary, AR(1) data:

```
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 data were 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
```

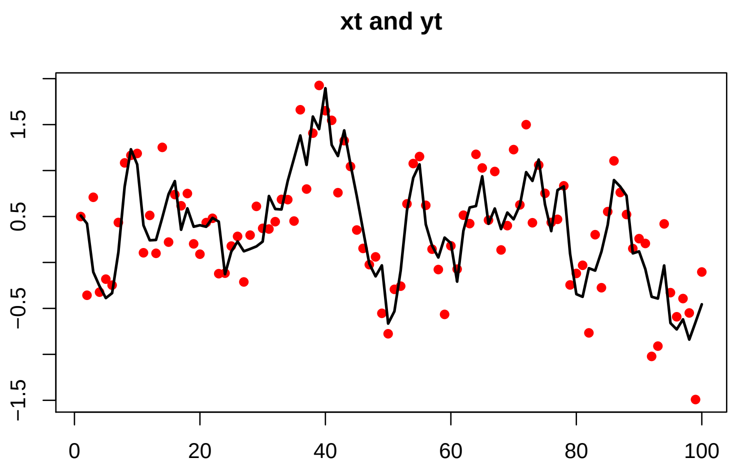
```
AIC: 133.3495   AIC c: 133.9878

      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:

```r
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
```

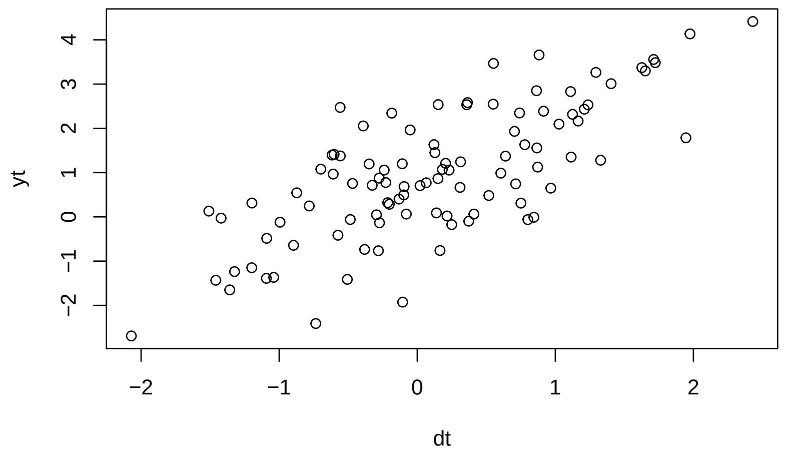
6.4 Linear regression

A simple linear regression of one covariate is written:

$$y_t = \alpha + \beta * d_t + v_t,\ w_t \sim \mathrm{N}(0, q) \tag{6.6}$$

Let's create some simulated data with this structure:

```r
beta <- 1.1
alpha <- 1
r <- 1
dt <- rnorm(TT, 0, 1)  #our covariate
vt <- rnorm(TT, 0, r)
yt <- alpha + beta * dt + vt
plot(dt, yt)
```



To fit this model, we need to write it in MARSS form. Here's the parts we need with the parameters, we don't need removed.

$$\begin{gathered}\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{w}_t,\ \mathbf{w}_t \sim \mathrm{MVN}(0, 0)\\ \mathbf{y}_t = 0 \times \mathbf{x}_t + \mathbf{D}\mathbf{d}_t + \mathbf{v}_t,\ \mathbf{v}_t \sim \mathrm{MVN}(0, \mathbf{R})\end{gathered} \tag{6.7}$$

We write out our parameters in matrix form. We want to set most of the **x** parameters to 0 so the algorithm doesn't try to estimate them.

```
R <- matrix("r")  # no v_t
D <- matrix("beta")
d <- matrix(dt, nrow = 1)
Z <- matrix(0)
A <- matrix("alpha")
Q <- U <- x0 <- 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.

```
mod.list <- list(U = U, Q = Q, x0 = x0, Z = Z, A = A, D = D,
    d = d, R = R)
fit <- MARSS(yt, model = mod.list)
```

```
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: -140.495
AIC: 286.9899   AICc: 287.2399

        Estimate
A.alpha    0.810
R.r        0.972
D.beta     1.227
Initial states (x0) defined at t=0

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

The estimates are the same as with `lm()`:

```
lm(yt ~ dt)
```

```

Call:
lm(formula = yt ~ dt)

Coefficients:
(Intercept)           dt
     0.8096       1.2273
```

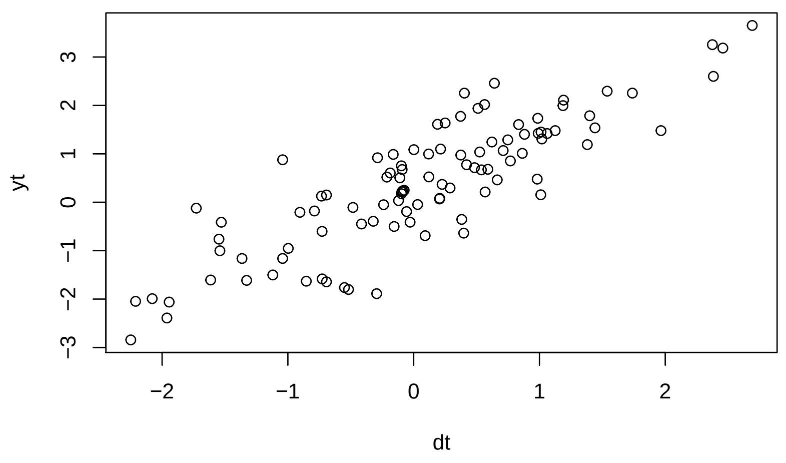
6.5 Linear regression with AR(1) errors

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

$$\begin{gathered}
x_t = bx_{t-1} + w_t,\ w_t \sim \text{N}(0, q) \\
y_t = \beta * d_t + x_t
\end{gathered} \tag{6.8}$$

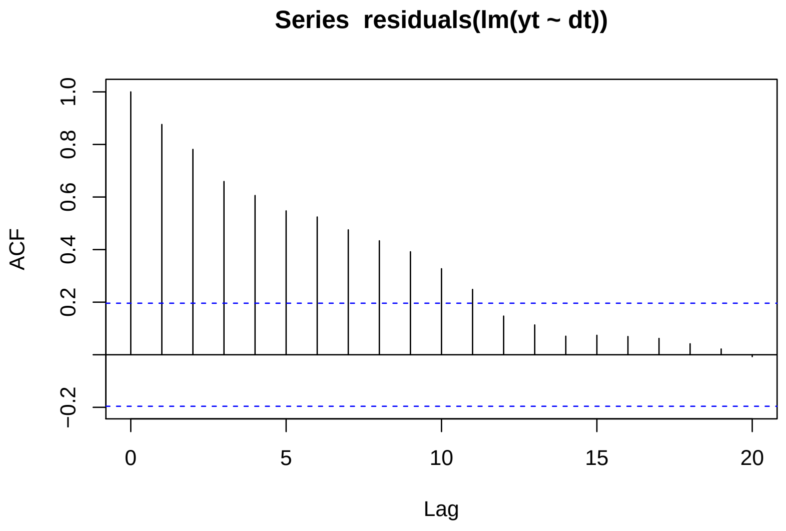
Let's create some simulated data with this structure:

```
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:

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



We can fit this model (Equation (6.8)) 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 (6.8) with the full MARSS model written in matrix form (Equation (5.1)). 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.

$$\begin{gathered}\mathbf{x}_t = \mathbf{B}\mathbf{x}_{t-1} + \mathbf{w}_t, \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t) \\ \mathbf{y}_t = \mathbf{x}_t + \mathbf{D}\mathbf{d}_t\end{gathered} \tag{6.9}$$

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

$$\begin{gathered}\begin{bmatrix}x\end{bmatrix}_t = \begin{bmatrix}b\end{bmatrix}\begin{bmatrix}x\end{bmatrix}_{t-1} + \begin{bmatrix}0\end{bmatrix} + \begin{bmatrix}w\end{bmatrix}_t \\ \begin{bmatrix}y\end{bmatrix}_t = \begin{bmatrix}x\end{bmatrix}_t + \begin{bmatrix}\beta\end{bmatrix}\begin{bmatrix}d\end{bmatrix}_t\end{gathered} \tag{6.10}$$

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 $\mathbf{d}$ 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 $\mathbf{R}$ matrix? It is zero, and we could set $\mathbf{R}$ 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)
```

```
Success! abstol and log-log tests passed at 75 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 75 iterations.
Log-likelihood: -31.93036
AIC: 73.86072   AICc: 74.49902

       Estimate
R.r      0.0116
B.b      0.9150
Q.Q      0.0911
x0.x0    0.0916
D.beta   1.0817
Initial states (x0) defined at t=0

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

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 169 iterations.
```

```
Function MARSSkfas used for likelihood calculation.

MARSS fit is
Estimation method: BFGS
Estimation converged in 169 iterations.
Log-likelihood: -32.16264
AIC: 72.32527   AICc: 72.74632

       Estimate
B.b      0.8970
Q.Q      0.1114
x0.x0    0.0936
D.beta   1.0858
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
      0.8887  1.0859
s.e.  0.0428  0.0251

sigma^2 estimated as 0.1115:  log likelihood = -32.97,  aic = 71.93
```

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

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

We can add some independent error to our model:

$$\begin{gathered}x_t = bx_{t-1} + w_t,\ w_t \sim \mathrm{N}(0, q)\\ y_t = \beta * d_t + x_t + v_t,,\ v_t \sim \mathrm{N}(0, r)\end{gathered} \tag{6.11}$$

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

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

We can fit as:

```
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 55 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 55 iterations.
Log-likelihood: -161.9318
AIC: 333.8635   AICc: 334.5018

       Estimate
R.r       1.034
B.b       0.810
Q.Q       0.255
x0.x0     1.229
D.beta    0.973
```

```
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()`.

6.7 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).

$$\begin{gathered} x_t = bx_{t-1} + \beta * c_t + w_t, \;\; w_t \sim \text{N}(0,q) \\ y_t = x_t + v_t, \;\; v_t \sim \text{N}(0,r) \end{gathered} \tag{6.12}$$

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:

$$\begin{gathered} \mathbf{x}_t = \mathbf{B}\mathbf{x}_{t-1} + \mathbf{C}\mathbf{c}_t + \mathbf{w}_t, \;\; \mathbf{w}_t \sim \text{MVN}(0,\mathbf{Q}_t) \\ \mathbf{y}_t = \mathbf{x}_t + \mathbf{v}_t, \;\; \mathbf{v}_t \sim \text{MVN}(0,\mathbf{R}_t) \end{gathered} \tag{6.13}$$

The model list for `MARSS()` is:

```r
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:

```r
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 56 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 56 iterations.
Log-likelihood: -142.9528
AIC: 295.9055   AICc: 296.5438

       Estimate
R.r       0.764
B.b       0.893
Q.Q       0.107
x0.x0     0.043
C.beta    1.045
Initial states (x0) defined at t=0

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

6.8 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.

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

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

$$y_t = a + v_t,\ v_t \sim \mathrm{N}(0, r) \tag{6.14}$$

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.

$$\begin{gathered} x_t = 1 \times x_{t-1} + 0 + w_t,\ w_t \sim \mathrm{N}(0, 0) \\ y_t = 0 \times x_t + a + v_t,\ v_t \sim \mathrm{N}(0, r) \\ x_0 = 0 \end{gathered} \tag{6.15}$$

The model list and fit for this equation is

```
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:

$$\begin{gathered} x_t = 1 \times x_{t-1} + 0 + w_t \text{ where } w_t \sim \mathrm{N}(0,0) \\ y_t = 1 \times x_t + 0 + v_t \text{ where } v_t \sim \mathrm{N}(0,r) \\ x_0 = a \end{gathered} \tag{6.16}$$

The model list and fit for this equation is

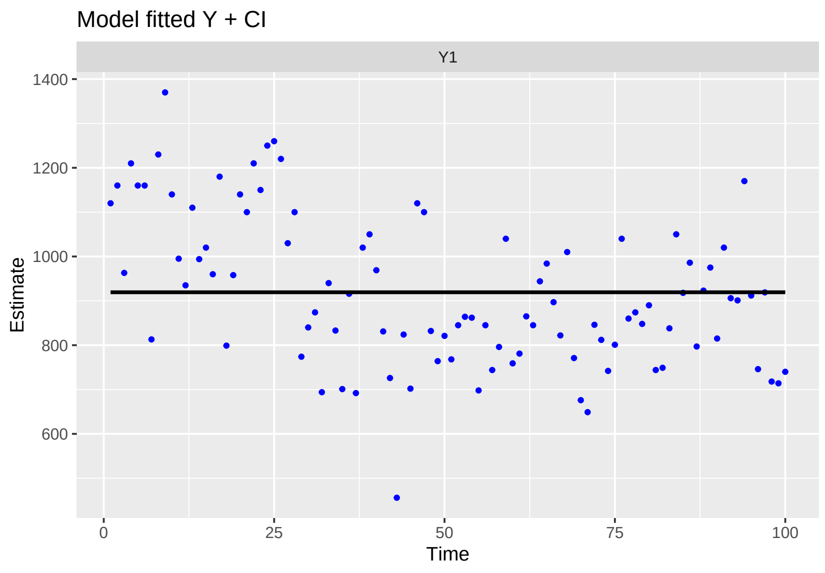
```r
mod.list2 <- list(Z = matrix(1), A = matrix(0), R = matrix("r"),
    B = matrix(1), U = matrix(0), Q = matrix(0), x0 = matrix("a"))
fit2 <- MARSS(nile, model = mod.list2)
```

```
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 = "model.ytT")
```



6.9 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:

$$\begin{gathered} x_t = 1 \times x_{t-1} + 0 + w_t,\ w_t \sim \mathrm{N}(0,0) \\ y_t = 0 \times x_t + a + \beta * t + v_t,\ v_t \sim \mathrm{N}(0,r) \\ x_0 = 0 \end{gathered} \tag{6.17}$$

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:

$$\begin{gathered}
x_t = 1 \times x_{t-1} + u + w_t,\ w_t \sim \text{N}(0,0) \\
y_t = 1 \times x_t + 0 + v_t,\ v_t \sim \text{N}(0,r) \\
x_0 = a
\end{gathered} \tag{6.18}$$

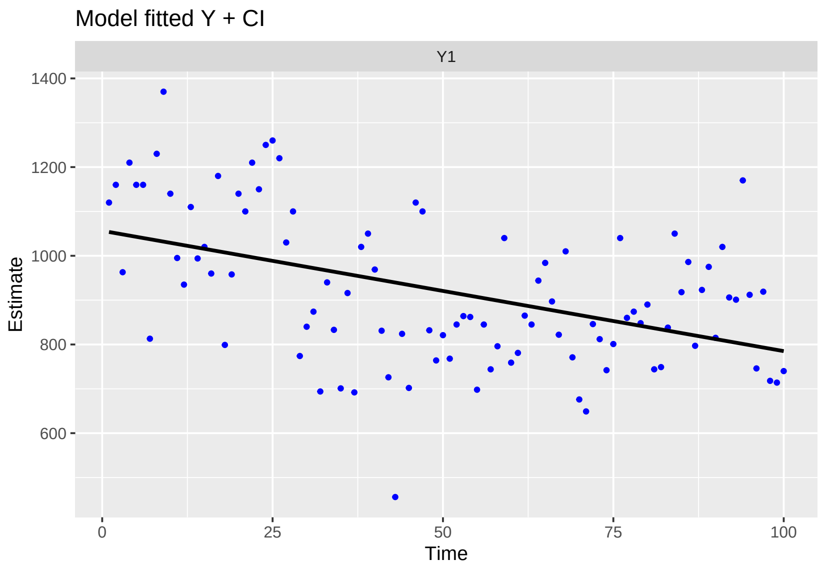
The model is specified as a list as follows. To fit, we need to force the algorithm to run a bit longer as it is showing convergence a bit early.

```
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 = "model.ytT")
```



6.10 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 $t-1$.

$$\begin{gathered} x_t = x_{t-1} + w_t \text{ where } w_t \sim \mathrm{N}(0,q) \\ y_t = x_t + v_t \text{ where } v_t \sim \mathrm{N}(0,r) \\ x_0 = \pi \end{gathered} \tag{6.19}$$

With all the MARSS parameters shown, the model is:

$$\begin{gathered} x_t = 1 \times x_{t-1} + 0 + w_t \text{ where } w_t \sim \mathrm{N}(0,q) \\ y_t = 1 \times x_t + 0 + v_t \text{ where } v_t \sim \mathrm{N}(0,r) \\ x_0 = \pi \end{gathered} \tag{6.20}$$

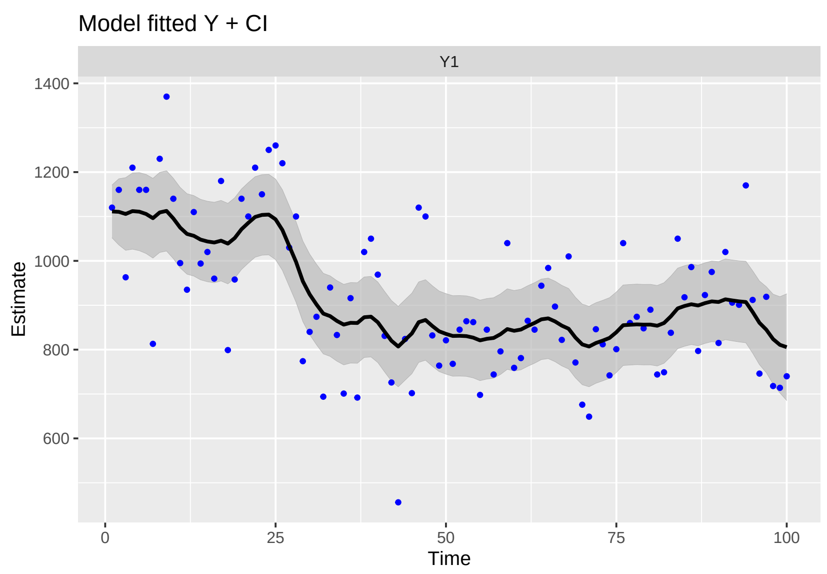
The model is specified as a list as follows. We can use the BFGS algorithm to ‘polish’ off the fit and get closer to the MLE. Why not just start with BFGS?

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 = "model.ytT")
```



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 ?.

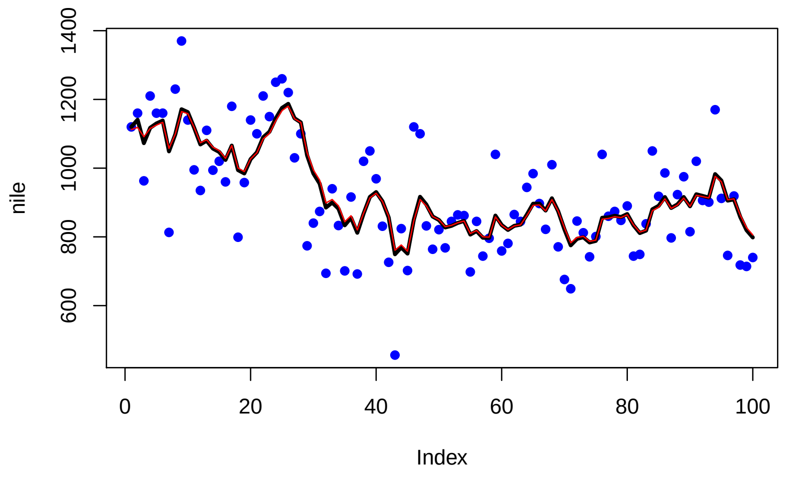
We can fit the Koopman model with `stats::StructTS()`. The estimates are slightly different since the initial conditions are treated differently.

```
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.

6.11 Stochastic slope model

We can also model the β as a random walk:

$$\begin{gathered}\beta_t = \beta_{t-1} + w_t \text{ where } w_t \sim \text{N}(0,q)\\ y_t = a + \beta_t * t + v_t \text{ where } v_t \sim \text{N}(0,r)\\ x_0 = \pi\end{gathered} \tag{6.21}$$

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

$$\begin{gathered}x_t = 1 \times x_{t-1} + 0 + w_t \text{ where } w_t \sim \text{N}(0,q)\\ y_t = t \times x_t + a + v_t \text{ where } v_t \sim \text{N}(0,r)\\ x_0 = \pi\end{gathered} \tag{6.22}$$

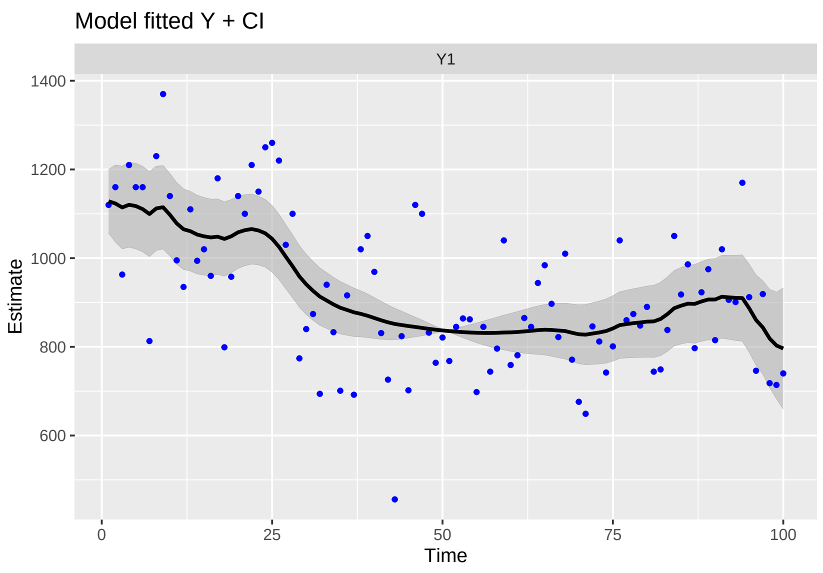
The trick here is to recognize that $\mathbf{Z}_t$, the matrix in front of $\mathbf{x}_t$ in the $\mathbf{y}_t$ equation, can be time-varying and can be fixed. In a time-varying matrix in **MARSS**, the time element is in the 3rd dimension. We are going to fix $\mathbf{Z}[1,1,t] = t$, where t is `year-mean(year)`. $\mathbf{Z}$ 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 = "model.ytT")
```


Model fitted Y + CI
Y1
Estimate
1400
1200
1000
800
600
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 7

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 8

MARSS Residuals

Chapter 9

Confidence Intervals

Chapter 10

Predictions

Part 4. Tips and Tricks

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

Chapter 11

Troubleshooting

Tip: Use `MARSSinfo()` for information on various common error and warning messages.

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. An example of a $\boldsymbol{\pi}$ with shared values is $\boldsymbol{\pi} = \begin{bmatrix} a \\ a \\ a \end{bmatrix}$. The way the EM algorithm deals with $\boldsymbol{\Lambda}$ 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 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. 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 12

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 (1.1). 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.

12.1 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. 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. 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 default behavior for `MARSS()` is to set $\boldsymbol{\Lambda}$ to zero and estimate $\boldsymbol{\pi}$. This does not put any constraints on $\boldsymbol{\Lambda}$ (there is no $\boldsymbol{\Lambda}$ to put constraints on) and circumvents this problem. However if you plan to put contraints on $\boldsymbol{\pi}$ or $\boldsymbol{\Lambda}$, you should verse yourself in the most common problems. The 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 $\boldsymbol{\Lambda}$ 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. This is the default behavior of `MARSS()`. Unfortunately 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.

12.2 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 13

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 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.
- 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 is an R package for fitting MARSS models. 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 an R package for fitting ARSS (univariate) models with Gaussian, Poisson and binomial error distributions.

- dse (Dynamic Systems Estimation) is an R package for multivariate Gaussian state-space models with a focus on ARMA models.
- 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.
- 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** is commercial economics software that will estimate at least some types of MARSS models.
- 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 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 is an R package provides a suite of functions specialized for fitting MARSS models to animal tracking data.
- crawl is an R package provides for fitting MARSS models to animal tracking data with covariates.

Bibliography