Time-Series Analysis for Fisheries and Environmental Data

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

2017-05-04

Contents

Preface

These are computer labs developed as part of a course we teach at the University of Washington on applied time-series analysis for fisheries and environmental data. You can find our lectures on our course website https://nwfsc-timeseries.github.io/atsa/. You can find our initial labs in the MARSS User Guide.

The authors are research scientists at the Northwest Fisheries Science Center (NWFSC). This work was conducted as part of our jobs at the NWFSC, a research center for NOAA Fisheries which is a United States federal government agency.

Links to more code and publications on MARSS applications can be found by following the links at our academic websites:

- http://faculty.washington.edu/eeholmes
- http://faculty.washington.edu/scheuerl
- https://sites.google.com/site/ericward2

Holmes, E. E., M. D. Scheuerell, and E. J. Ward. Time-series analysis for fisheries and environmental data. NOAA Fisheries, Northwest Fisheries Science Center, 2725 Montlake Blvd E., Seattle, WA 98112. Contacts eli.holmes@noaa.gov, eric.ward@noaa.gov, and mark.scheuerell@noaa.gov

Chapter 1

Basic matrix math in R

This chapter reviews the basic matrix math operations that you will need to understand the course material and shows how to do these operations in R.

A script with all the R code in the chapter can be downloaded here.

1.1 Creating matrices in R

Create a 3×4 matrix, meaning 3 row and 4 columns, that is all 1s:

```
matrix(1, 3, 4)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    1    1    1
[2,]    1    1    1    1
[3,]    1    1    1    1
```

Create a 3×4 matrix filled in with the numbers 1 to 12 by column (default) and by row:

```
matrix(1:12, 3, 4)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

```
matrix(1:12, 3, 4, byrow = TRUE)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

Create a matrix with one column:

```
matrix(1:4, ncol = 1)
```

```
     [,1]
[1,]    1
[2,]    2
[3,]    3
[4,]    4
```

Create a matrix with one row:

```
matrix(1:4, nrow = 1)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
```

Check the dimensions of a matrix

```
A = matrix(1:6, 2, 3)
A
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
dim(A)
```

```
[1] 2 3
```

Get the number of rows in a matrix:

```
dim(A)[1]
```

```
[1] 2
```

```
nrow(A)
```

```
[1] 2
```

Create a 3D matrix (called array):

```
A = array(1:6, dim = c(2, 3, 2))
A
```

```
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
```

```
[1,]    1    3    5
[2,]    2    4    6
```

```
dim(A)
```

```
[1] 2 3 2
```

Check if an object is a matrix. A data frame is not a matrix. A vector is not a matrix.

```
A = matrix(1:4, 1, 4)
A
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
```

```
class(A)
```

```
[1] "matrix"
```

```
B = data.frame(A)
B
```

```
  X1 X2 X3 X4
1  1  2  3  4
```

```
class(B)
```

```
[1] "data.frame"
```

```
C = 1:4
C
```

```
[1] 1 2 3 4
```

```
class(C)
```

```
[1] "integer"
```

1.2 Matrix multiplication, addition and transpose

You will need to be very solid in matrix multiplication for the course. If you haven't done it in awhile, google 'matrix multiplication youtube' and you find lots of 5min videos to remind you.

In R, you use the `%*%` operation to do matrix multiplication. When you do matrix multiplication, the columns of the matrix on the left must equal the rows of the matrix on the right. The result is a matrix that has the number of rows of the matrix on the left and number of columns of the matrix on the right.

$$(n \times m)(m \times p) = (n \times p)$$

```
A=matrix(1:6, 2, 3) #2 rows, 3 columns
B=matrix(1:6, 3, 2) #3 rows, 2 columns
A%*%B #this works
```

```
     [,1] [,2]
[1,]   22   49
[2,]   28   64
```

```
B%*%A #this works
```

```
     [,1] [,2] [,3]
[1,]    9   19   29
[2,]   12   26   40
[3,]   15   33   51
```

```
try(B%*%B) #this doesn't
```

To add two matrices use `+`. The matrices have to have the same dimensions.

```
A+A #works
```

```
     [,1] [,2] [,3]
[1,]    2    6   10
[2,]    4    8   12
```

```
A+t(B) #works
```

```
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    6    9   12
```

```
try(A+B) #does not work since A has 2 rows and B has 3
```

The transpose of a matrix is denoted $\mathbf{A}^\top$ or $\mathbf{A}'$. To transpose a matrix in R, you use `t()`.

```
A=matrix(1:6, 2, 3) #2 rows, 3 columns
t(A) #is the transpose of A
```

```
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
```

```
try(A%*%A) #this won't work
A%*%t(A) #this will
```

```
     [,1] [,2]
[1,]   35   44
[2,]   44   56
```

1.3 Subsetting a matrix

To subset a matrix, we use `[ ]`:

```
A=matrix(1:9, 3, 3) #3 rows, 3 columns
#get the first and second rows of A
#it's a 2x3 matrix
A[1:2,]
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
```

```
#get the top 2 rows and left 2 columns
A[1:2,1:2]
```

```
     [,1] [,2]
[1,]    1    4
[2,]    2    5
```

```
#What does this do?
A[c(1,3),c(1,3)]
```

```
     [,1] [,2]
[1,]    1    7
[2,]    3    9
```

```
#This?
A[c(1,2,1),c(2,3)]
```

```
     [,1] [,2]
[1,]    4    7
[2,]    5    8
[3,]    4    7
```

If you have used matlab, you know you can say something like `A[1,end]` to denote the element of a matrix in row 1 and the last column. R does not have 'end'. To do, the same in R you do something like:

```
A=matrix(1:9, 3, 3)
A[1,ncol(A)]
```

```
[1] 7
```

```
#or
A[1,dim(A)[2]]
```

```
[1] 7
```

Warning R will create vectors from subsetting matrices!

One of the really bad things that R does with matrices is create a vector if you happen to subset a matrix to create a matrix with 1 row or 1 column. Look at this:

```
A=matrix(1:9, 3, 3)
#take the first 2 rows
B=A[1:2,]
#everything is ok
dim(B)
```

```
[1] 2 3
```

```
class(B)
```

```
[1] "matrix"
```

```
#take the first row
B=A[1,]
#oh no! It should be a 1x3 matrix but it is not.
dim(B)
```

```
NULL
```

```
#It is not even a matrix any more
class(B)
```

```
[1] "integer"
```

```
#and what happens if we take the transpose?
#Oh no, it's a 1x3 matrix not a 3x1 (transpose of 1x3)
t(B)
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
```

```
#A%*%B should fail because A is (3x3) and B is (1x3)
A%*%B
```

```
     [,1]
[1,]   66
[2,]   78
[3,]   90
```

```
#It works? That is horrible!
```

This will create hard to find bugs in your code because you will look at `B=A[1,]` and everything looks fine. Why is R saying it is not a matrix! To stop R from doing this use `drop=FALSE`.

```
B=A[1,,drop=FALSE]
#Now it is a matrix as it should be
dim(B)
```

```
[1] 1 3
```

```
class(B)
```

```
[1] "matrix"
```

```
#this fails as it should (alerting you to a problem!)
try(A%*%B)
```

1.4 Replacing elements in a matrix

Replace 1 element.

```
A=matrix(1, 3, 3)
A[1,1]=2
A
```

```
     [,1] [,2] [,3]
[1,]    2    1    1
[2,]    1    1    1
[3,]    1    1    1
```

Replace a row with all 1s or a string of values

```
A=matrix(1, 3, 3)
A[1,]=2
A
```

```
     [,1] [,2] [,3]
[1,]    2    2    2
[2,]    1    1    1
[3,]    1    1    1
```

```
A[1,]=1:3
A
```

```
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    1    1    1
[3,]    1    1    1
```

Replace group of elements. This often does not work as one expects so be sure look at your matrix after trying something like this. Here I want to replace elements (1,3) and (3,1) with 2, but it didn't work as I wanted.

```
A=matrix(1, 3, 3)
A[c(1,3),c(3,1)]=2
A
```

```
     [,1] [,2] [,3]
[1,]    2    1    2
[2,]    1    1    1
[3,]    2    1    2
```

How do I replace elements (1,1) and (3,3) with 2 then? It's tedious. If you have a lot of elements to replace, you might want to use a for loop.

```
A=matrix(1, 3, 3)
A[1,3]=2
A[3,1]=2
A
```

```
     [,1] [,2] [,3]
[1,]    1    1    2
[2,]    1    1    1
[3,]    2    1    1
```

1.5 Diagonal matrices and identity matrices

A diagonal matrix is one that is square, meaning number of rows equals number of columns, and it has 0s on the off-diagonal and non-zeros on the diagonal. In R, you form a diagonal matrix with the `diag()` function:

```
diag(1,3) #put 1 on diagonal of 3x3 matrix
```

```
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

```
diag(2, 3) #put 2 on diagonal of 3x3 matrix
```

```
     [,1] [,2] [,3]
[1,]    2    0    0
[2,]    0    2    0
[3,]    0    0    2
```

```
diag(1:4) #put 1 to 4 on diagonal of 4x4 matrix
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    2    0    0
[3,]    0    0    3    0
[4,]    0    0    0    4
```

The `diag()` function can also be used to replace elements on the diagonal of a matrix:

```
A = matrix(3, 3, 3)
diag(A) = 1
A
```

```
     [,1] [,2] [,3]
[1,]    1    3    3
[2,]    3    1    3
[3,]    3    3    1
```

```
A = matrix(3, 3, 3)
diag(A) = 1:3
A
```

```
     [,1] [,2] [,3]
[1,]    1    3    3
[2,]    3    2    3
[3,]    3    3    3
```

```
A = matrix(3, 3, 4)
diag(A[1:3, 2:4]) = 1
A
```

```
     [,1] [,2] [,3] [,4]
[1,]    3    1    3    3
[2,]    3    3    1    3
[3,]    3    3    3    1
```

The `diag()` function is also used to get the diagonal of a matrix.

```
A = matrix(1:9, 3, 3)
diag(A)
```

```
[1] 1 5 9
```

The identity matrix is a special kind of diagonal matrix with 1s on the diagonal. It is denoted $\mathbf{I}$. $\mathbf{I}_3$ would mean a 3×3 diagonal matrix. A identity matrix has the property that $\mathbf{AI} = \mathbf{A}$ and $\mathbf{IA} = \mathbf{A}$ so it is like a 1.

```
A = matrix(1:9, 3, 3)
I = diag(3)  #shortcut for 3x3 identity matrix
A %*% I
```

```
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

1.6 Taking the inverse of a square matrix

The inverse of a matrix is denoted $\mathbf{A}^{-1}$. You can think of the inverse of a matrix like $1/a$. $1/a \times a = 1$. $\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$. The inverse of a matrix does not always exist; for one it has to be square. We'll be using inverses for variance-covariance matrices and by definition (of a variance-covariance matrix), the inverse of those exist. In R, there are a couple way common ways to take the inverse of a variance-covariance matrix (or something with the same properties). `solve()` is the most common probably:

```
A = diag(3, 3) + matrix(1, 3, 3)
invA = solve(A)
invA %*% A
```

```
              [,1]          [,2] [,3]
[1,] 1.000000e+00 -6.938894e-18    0
[2,] 2.081668e-17  1.000000e+00    0
[3,] 0.000000e+00  0.000000e+00    1
```

```
A %*% invA
```

```
              [,1]          [,2] [,3]
[1,] 1.000000e+00 -6.938894e-18    0
[2,] 2.081668e-17  1.000000e+00    0
[3,] 0.000000e+00  0.000000e+00    1
```

Another option is to use `chol2inv()` which uses a Cholesky decomposition[1]:

```
A = diag(3, 3) + matrix(1, 3, 3)
invA = chol2inv(chol(A))
invA %*% A
```

```
               [,1]         [,2]          [,3]
[1,]  1.000000e+00 6.938894e-17  0.000000e+00
[2,]  2.081668e-17 1.000000e+00 -2.775558e-17
[3,] -5.551115e-17 0.000000e+00  1.000000e+00
```

```
A %*% invA
```

```
              [,1]          [,2]          [,3]
[1,] 1.000000e+00  2.081668e-17 -5.551115e-17
[2,] 6.938894e-17  1.000000e+00  0.000000e+00
[3,] 0.000000e+00 -2.775558e-17  1.000000e+00
```

For the purpose of this course, `solve()` is fine.

[1]The Cholesky decomposition is a handy way to keep your variance-covariance matrices valid when doing a parameter search. Don't search over the raw variance-covariance matrix. Search over a matrix where the lower triangle is 0, that is what a Cholesky decomposition looks like. Let's call it `B`. Your variance-covariance matrix is `t(B)%*%B`.

1.7 Problems

1. Build a 4×3 matrix with the numbers 1 through 4 in each row.
2. Extract the elements in the 1st and 2nd rows and 1st and 2nd columns (you'll have a 2×2 matrix). Show the R code that will do this.
3. Build a 4×3 matrix with the numbers 1 through 12 by row (meaning the first row will have the numbers 1 through 4 in it).
4. Extract the 3rd row of the above. Show R code to do this where you end up with a vector and how to do this where you end up with a 1×3 matrix.
5. Build a 4×3 matrix that is all 1s except a 2 in the (2,3) element (2nd row, 3rd column).
6. Take the transpose of the above.
7. Build a 4×4 diagonal matrix with 1 through 4 on the diagonal.
8. Build a 5×5 identity matrix.
9. Replace the diagonal in the above matrix with 2 (the number 2).
10. Build a matrix with 2 on the diagonal and 1s on the offdiagonals.
11. Take the inverse of the above.
12. Build a 3×3 matrix with the first 9 letters of the alphabet. First column should be "a", "b", "c". `letters[1:9]` gives you these letters.
13. Replace the diagonal of this matrix with the word "cat".
14. Build a 4×3 matrix with all 1s. Multiply by a 3×4 matrix with all 2s.
15. If $\mathbf{A}$ is a 4×3 matrix, is $\mathbf{AA}$ possible? Is $\mathbf{AA}^\top$ possible? Show how to write $\mathbf{AA}^\top$ in R.
16. In the equation, $\mathbf{AB} = \mathbf{C}$, let $\mathbf{A} = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$. Build a $\mathbf{B}$ matrix with only 1s and 0s such that the values on the diagonal of $\mathbf{C}$ are 1, 8, 6 (in that order). Show your R code for $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{AB}$.
17. Same $\mathbf{A}$ matrix as above and same equation $\mathbf{AB} = \mathbf{C}$. Build a 3×3 $\mathbf{B}$ matrix such that $\mathbf{C} = 2\mathbf{A}$. So $\mathbf{C} = \begin{bmatrix} 2 & 8 & 14 \\ 4 & 10 & 16 \\ 6 & 12 & 18 \end{bmatrix}$. Hint, $\mathbf{B}$ is diagonal.
18. Same $\mathbf{A}$ and $\mathbf{AB} = \mathbf{C}$ equation. Build a $\mathbf{B}$ matrix to compute the row sums of $\mathbf{A}$. So the first 'row sum' would be $1 + 4 + 7$, the sum of all elements in row 1 of $\mathbf{A}$. $\mathbf{C}$ will be $\begin{bmatrix} 12 \\ 15 \\ 18 \end{bmatrix}$, the row sums of $\mathbf{A}$. Hint, $\mathbf{B}$ is a column matrix (1 column).
19. Same $\mathbf{A}$ matrix as above but now equation $\mathbf{BA} = \mathbf{C}$. Build a $\mathbf{B}$ matrix to compute the column sums of $\mathbf{A}$. So the first 'column sum' would be $1 + 2 + 3$. $\mathbf{C}$ will be a 1×3 matrix.

20. Let $\mathbf{AB} = \mathbf{C}$ equation but $\mathbf{A} = \begin{bmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{bmatrix}$ (so A=`diag(3)+1`). Build a $\mathbf{B}$ matrix such that $\mathbf{C} = \begin{bmatrix} 3 \\ 3 \\ 3 \end{bmatrix}$. Hint, you need to use the inverse of $\mathbf{A}$.

Chapter 2

Linear regression in matrix form

This chapter shows how to write linear regression models in matrix form. The purpose is to get you comfortable writing multivariate linear models in different matrix forms before we start working with time-series versions of these models. Each matrix form is an equivalent model for the data, but written in different forms. You do not need to worry which form is better or worse at this point. Simply get comfortable writing multivariate linear models in different matrix forms.

A script with all the R code in the chapter can be downloaded here.

Data and packages

This chapter uses the **stats**, **MARSS** and **datasets** packages. Install those packages, if needed, and load:

```
library(stats)
library(MARSS)
library(datasets)
```

We will work with the `stackloss` dataset available in R. The dataset consists of 21 observations on the efficiency of a plant that produces nitric acid as a function of three explanatory variables: air flow, water temperature and acid concentration. We are going to use just the first 4 datapoints so that it is easier to write the matrices, but the concepts extend to as many datapoints as you have.

```
data(stackloss)
dat = stackloss[1:4, ]  #subsetted first 4 rows
dat
```

```
  Air.Flow Water.Temp Acid.Conc. stack.loss
1       80         27         89         42
2       80         27         88         37
3       75         25         90         37
```

```
4          62          24          87          28
```

2.1 A simple regression: one explanatory variable

We will start by regressing stack loss against air flow. In R using the `lm()` function this is

```
# the dat data.frame is defined on the first page of the
# chapter
lm(stack.loss ~ Air.Flow, data = dat)
```

This fits the following model for the i-th measurment:

$$stack.loss_i = \alpha + \beta air_i + e_i, \text{ where } e_i \sim \mathrm{N}(0, \sigma^2) \tag{2.1}$$

We will write the model for all the measurements together in two different ways, Form 1 and Form 2.

2.2 Matrix Form 1

In this form, we have the explanatory variables in a matrix on the left of our parameter matrix:

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} 1 & air_1 \\ 1 & air_2 \\ 1 & air_3 \\ 1 & air_4 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} \tag{2.2}$$

You should work through the matrix algebra to make sure you understand why Equation (2.2) is Equation (2.1) for all the i data points together.

We can write the first line of Equation (2.2) succinctly as

$$\mathbf{y} = \mathbf{Z}\mathbf{x} + \mathbf{e} \tag{2.3}$$

where $\mathbf{x}$ are our parameters, $\mathbf{y}$ are our response variables, and $\mathbf{Z}$ are our explanatory variables (with a 1 column for the intercept). The `lm()` function uses Form 1, and we can recover the $\mathbf{Z}$ matrix for Form 1 by using the `model.matrix()` function on the output from a `lm()` call:

```
fit = lm(stack.loss ~ Air.Flow, data = dat)
Z = model.matrix(fit)
Z[1:4, ]
```

```
  (Intercept) Air.Flow
1           1       80
2           1       80
3           1       75
4           1       62
```

2.2.1 Solving for the parameters

Note: You will not need to know how to solve linear matrix equations for this course. This section just shows you what the `lm()` function is doing to estimate the parameters.

Notice that $\mathbf{Z}$ is not a square matrix and its inverse does not exist but the inverse of $\mathbf{Z}^\top\mathbf{Z}$ exists—if this is a solveable problem. We can go through the following steps to solve for $\mathbf{x}$, our parameters α and β.

Start with $\mathbf{y} = \mathbf{Z}\mathbf{x} + \mathbf{e}$ and multiply by $\mathbf{Z}^\top$ on the left to get

$$\mathbf{Z}^\top\mathbf{y} = \mathbf{Z}^\top\mathbf{Z}\mathbf{x} + \mathbf{Z}^\top\mathbf{e}$$

Multiply that by $(\mathbf{Z}^\top\mathbf{Z})^{-1}$ on the left to get

$$(\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{y} = (\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{Z}\mathbf{x} + (\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{e}$$

$(\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{Z}$ equals the identity matrix, thus

$$(\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{y} = \mathbf{x} + (\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{e}$$

Move $\mathbf{x}$ to the right by itself, to get

$$(\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{y} - (\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{e} = \mathbf{x}$$

Let's assume our errors, the $\mathbf{e}$, are i.i.d. which means that

$$\mathbf{e} \sim \text{MVN}\left(0, \begin{bmatrix} \sigma^2 & 0 & 0 & 0 \\ 0 & \sigma^2 & 0 & 0 \\ 0 & 0 & \sigma^2 & 0 \\ 0 & 0 & 0 & \sigma^2 \end{bmatrix}\right)$$

This equation means $\mathbf{e}$ is drawn from a multivariate normal distribution with a variance-covariance matrix that is diagonal with equal variances. Under that assumption, the expected value of $(\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{e}$ is zero. So we can solve for $\mathbf{x}$ as

$$\mathbf{x} = (\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{y}$$

Let's try that with R and compare to what you get with `lm()`:

```
y = matrix(dat$stack.loss, ncol = 1)
Z = cbind(1, dat$Air.Flow)  #or use model.matrix() to get Z
solve(t(Z) %*% Z) %*% t(Z) %*% y
```

```
            [,1]
[1,] -11.6159170
[2,]   0.6412918
```

```
coef(lm(stack.loss ~ Air.Flow, data = dat))
```

```
(Intercept)    Air.Flow
-11.6159170   0.6412918
```

As you see, you get the same values.

2.2.2 Form 1 with multiple explanatory variables

We can easily extend Form 1 to multiple explanatory variables. Let's say we wanted to fit this model:

$$stack.loss_i = \alpha + \beta_1 air_i + \beta_2 water_i + \beta_3 acid_i + e_i \tag{2.4}$$

With `lm()`, we can fit this with

```
fit1.mult = lm(stack.loss ~ Air.Flow + Water.Temp + Acid.Conc.,
    data = dat)
```

Written in matrix form (Form 1), this is

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} 1 & air_1 & water_1 & acid_1 \\ 1 & air_2 & water_2 & acid_2 \\ 1 & air_3 & water_3 & acid_3 \\ 1 & air_4 & water_4 & acid_4 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta_1 \\ \beta_2 \\ \beta_3 \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} \tag{2.5}$$

Now $\mathbf{Z}$ is a matrix with 4 columns and $\mathbf{x}$ is a column vector with 4 rows. We can show the $\mathbf{Z}$ matrix again directly from our `lm()` fit:

```
Z = model.matrix(fit1.mult)
Z
```

```
  (Intercept) Air.Flow Water.Temp Acid.Conc.
1           1       80         27         89
2           1       80         27         88
3           1       75         25         90
4           1       62         24         87
```

```
attr(,"assign")
[1] 0 1 2 3
```

We can solve for $\mathbf{x}$ just like before and compare to what we get with `lm()`:

```
y = matrix(dat$stack.loss, ncol = 1)
Z = cbind(1, dat$Air.Flow, dat$Water.Temp, dat$Acid.Conc)
# or Z=model.matrix(fit2)
solve(t(Z) %*% Z) %*% t(Z) %*% y
```

```
            [,1]
[1,] -524.904762
[2,]   -1.047619
[3,]    7.619048
[4,]    5.000000
```

```
coef(fit1.mult)
```

```
(Intercept)    Air.Flow  Water.Temp  Acid.Conc.
-524.904762   -1.047619    7.619048    5.000000
```

Take a look at the $\mathbf{Z}$ we made in R. It looks exactly like what is in our model written in matrix form (Equation (2.5)).

2.2.3 When does Form 1 arise?

This form of writing a regression model will come up when you work with dynamic linear models (DLMs). With DLMs, you will be fitting models of the form $\mathbf{y}_t = \mathbf{Z}_t\mathbf{x}_t + \mathbf{e}_t$. In these models you have multiple $\mathbf{y}$ at regular time points and you allow your regression parameters, the $\mathbf{x}$, to evolve through time as a random walk.

2.2.4 Matrix Form 1b: The transpose of Form 1

We could also write Form 1 as follows:

$$\begin{bmatrix} stack.loss_1 & stack.loss_2 & stack.loss_3 & stack.loss_4 \end{bmatrix} = \begin{bmatrix} \alpha & \beta_1 & \beta_2 & \beta_3 \end{bmatrix} \begin{bmatrix} 1 & 1 & 1 & 1 \\ air_1 & air_2 & air_3 & air_4 \\ wind_1 & wind_2 & wind_3 & wind_4 \\ acid_1 & acid_2 & acid_3 & acid_4 \end{bmatrix} + \begin{bmatrix} e_1 & e_2 & e_3 & e_4 \end{bmatrix} \tag{2.6}$$

This is just the transpose of Form 1. Work through the matrix algebra to make sure you understand why Equation (2.6) is Equation (2.1) for all the i data points together and why it is equal to the transpose of Equation (2.2). You'll need the relationship $(\mathbf{AB})^\top = \mathbf{B}^\top\mathbf{A}^\top$.

Let's write Equation (2.6) as $\mathbf{y} = \mathbf{Dd}$, where $\mathbf{D}$ contains our parameters. Then we can solve for $\mathbf{D}$ following the steps in Section 2.2.1 but multiplying from the right instead of from the left. Work through the steps to show that $\mathbf{d} = \mathbf{y}\mathbf{d}^\top(\mathbf{d}\mathbf{d}^\top)^{-1}$.

```
y = matrix(dat$stack.loss, nrow = 1)
d = rbind(1, dat$Air.Flow, dat$Water.Temp, dat$Acid.Conc)
y %*% t(d) %*% solve(d %*% t(d))
```

```
          [,1]      [,2]     [,3] [,4]
[1,] -524.9048 -1.047619 7.619048    5
```

```
coef(fit1.mult)
```

```
(Intercept)    Air.Flow  Water.Temp  Acid.Conc.
-524.904762   -1.047619    7.619048    5.000000
```

2.3 Matrix Form 2

In this form, we have the explanatory variables in a matrix on the right of our parameter matrix as in Form 1b but we arrange everything a little differently:

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} \alpha & \beta & 0 & 0 & 0 \\ \alpha & 0 & \beta & 0 & 0 \\ \alpha & 0 & 0 & \beta & 0 \\ \alpha & 0 & 0 & 0 & \beta \end{bmatrix} \begin{bmatrix} 1 \\ air_1 \\ air_2 \\ air_3 \\ air_4 \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} \tag{2.7}$$

Work through the matrix algebra to make sure you understand why Equation (2.7) is the same as Equation (2.1) for all the i data points together.

We will write Form 2 succinctly as

$$\mathbf{y} = \mathbf{Z}\mathbf{x} + \mathbf{e} \tag{2.8}$$

2.3.1 Form 2 with multiple explanatory variables

The $\mathbf{x}$ is a column vector of the explanatory variables. If we have more explanatory variables, we add them to the column vector at the bottom. So if we had air flow, water temperature and acid concentration as explanatory variables, $\mathbf{x}$ looks like

$$\begin{bmatrix} 1 \\ air_1 \\ air_2 \\ air_3 \\ air_4 \\ water_1 \\ water_2 \\ water_3 \\ water_4 \\ acid_1 \\ acid_2 \\ acid_3 \\ acid_4 \end{bmatrix} \tag{2.9}$$

Add columns to the $\mathbf{Z}$ matrix for each new variable.

$$\begin{bmatrix} \alpha & \beta_1 & 0 & 0 & 0 & \beta_2 & 0 & 0 & 0 & \beta_3 & 0 & 0 & 0 \\ \alpha & 0 & \beta_1 & 0 & 0 & 0 & \beta_2 & 0 & 0 & 0 & \beta_3 & 0 & 0 \\ \alpha & 0 & 0 & \beta_1 & 0 & 0 & 0 & \beta_2 & 0 & 0 & 0 & \beta_3 & 0 \\ \alpha & 0 & 0 & 0 & \beta_1 & 0 & 0 & 0 & \beta_2 & 0 & 0 & 0 & \beta_3 \end{bmatrix} \tag{2.10}$$

The number of rows of $\mathbf{Z}$ is always n, the number of rows of $\mathbf{y}$, because the number of rows on the left and right of the equal sign must match. The number of columns in $\mathbf{Z}$ is determined by the size of $\mathbf{x}$. If there is an intercept, there is a 1 in $\mathbf{x}$. Then each explanatory variable (like air flow and wind) appears n times. So if the number of explanatory variables is k, the number of columns in $\mathbf{Z}$ is $1 + k \times n$ if there is an intercept term and $k \times n$ if there is not.

2.3.2 When does Form 2 arise?

Form 2 is similar to how multivariate time-series models are typically written for reading by humans (on a whiteboard or paper). In these models, we see equations like this:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}_t = \begin{bmatrix} \beta_a & \beta_b \\ \beta_a & 0.1 \\ \beta_b & \beta_a \\ 0 & \beta_a \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}_t + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix}_t \tag{2.11}$$

In this case, $\mathbf{y}_t$ is the set of 4 observations at time t and $\mathbf{x}_t$ is the set of 2 explanatory variables at time t. The $\mathbf{Z}$ is showing how we are modeling the effects of x_1 and x_2 on the ys. Notice that the effects are not consistent across the x and y. This model would not be possible to fit with `lm()` but will be easy to fit with `MARSS()`.

2.3.3 Solving for the parameters for Form 2

You can just skim this section if you want but make sure you carefully look at the code in refsec-mlr-solveform2code. You will need to adapt that for the homework. Though you will not need any of the math discussed here for the course, this section will help you practice matrix multiplication and will introduce you to 'permutation' matrices which will be handy in many other contexts.

To solve for α and β, we need our parameters in a column matrix like so $\left[\begin{smallmatrix}\alpha\\ \beta\end{smallmatrix}\right]$. We do this by rewritting $\mathbf{Zx}$ in Equation (2.8) in 'vec' form: if $\mathbf{Z}$ is a $n \times m$ matrix and $\mathbf{x}$ is a matrix with 1 column and m rows, then $\mathbf{Zx} = (\mathbf{x}^\top \otimes \mathbf{I}_n)\,\text{vec}(\mathbf{Z})$. The symbol $\otimes$ means Kronecker product and just ignore it since you'll never see it again in our course (or google 'kronecker product' if you are curious). The "vec" of a matrix is that matrix rearranged as a single column:

$$\text{vec}\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix} = \begin{bmatrix}1\\ 3\\ 2\\ 4\end{bmatrix}$$

Notice how you just take each column one by one and stack them under each other. In R, the vec is

```
A = matrix(1:6, nrow = 2, byrow = TRUE)
vecA = matrix(A, ncol = 1)
```

$\mathbf{I}_n$ is a $n \times n$ identity matrix, a diagonal matrix with all 0s on the off-diagonals and all 1s on the diagonal. In R, this is simply `diag(n)`.

To show how we solve for α and β, let's use an example with only 3 data points so Equation (2.7) becomes:

$$\begin{bmatrix}stack.loss_1\\ stack.loss_2\\ stack.loss_3\end{bmatrix} = \begin{bmatrix}\alpha & \beta & 0 & 0\\ \alpha & 0 & \beta & 0\\ \alpha & 0 & 0 & \beta\end{bmatrix}\begin{bmatrix}1\\ air_1\\ air_2\\ air_3\end{bmatrix} + \begin{bmatrix}e_1\\ e_2\\ e_3\end{bmatrix} \tag{2.12}$$

Using $\mathbf{Zx} = (\mathbf{x}^\top \otimes \mathbf{I}_n)\,\text{vec}(\mathbf{Z})$, this means

$$\begin{bmatrix} \alpha & \beta & 0 & 0 \\ \alpha & 0 & \beta & 0 \\ \alpha & 0 & 0 & \beta \end{bmatrix} \begin{bmatrix} 1 \\ air_1 \\ air_2 \\ air_3 \end{bmatrix} = \left(\begin{bmatrix} 1 & air_1 & air_2 & air_3 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \right) \begin{bmatrix} \alpha \\ \alpha \\ \alpha \\ \beta \\ 0 \\ 0 \\ 0 \\ \beta \\ 0 \\ 0 \\ 0 \\ \beta \end{bmatrix} \tag{2.13}$$

We need to rewrite the vec($\mathbf{Z}$) as a 'permutation' matrix times $\left[\begin{smallmatrix} \alpha \\ \beta \end{smallmatrix}\right]$:

$$\begin{bmatrix} \alpha \\ \alpha \\ \alpha \\ \beta \\ 0 \\ 0 \\ 0 \\ \beta \\ 0 \\ 0 \\ 0 \\ \beta \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \mathbf{P}\mathbf{p} \tag{2.14}$$

where $\mathbf{P}$ is the permutation matrix and $\mathbf{p} = \left[\begin{smallmatrix} \alpha \\ \beta \end{smallmatrix}\right]$. Thus,

$$\mathbf{y} = \mathbf{Z}\mathbf{x} + \mathbf{e} = (\mathbf{x}^\top \otimes \mathbf{I}_n)\mathbf{P} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \mathbf{M}\mathbf{p} + \mathbf{e} \tag{2.15}$$

where $\mathbf{M} = (\mathbf{x}^\top \otimes \mathbf{I}_n)\mathbf{P}$. We can solve for $\mathbf{p}$, the parameters, using

$$(\mathbf{M}^\top\mathbf{M})^{-1}\mathbf{M}^\top\mathbf{y}$$

as before.

2.3.4 Code to solve for parameters in Form 2

In the homework, you will use the R code in this section to solve for the parameters in Form 2. Later when you are fitting multivariate time-series models, you will not solve for parameters

this way but you will need to both construct **Z** matrices in R and read **Z** matrices. The homework will give you practice creating **Z** matrices in R.

```
#make your y and x matrices
y=matrix(dat$stack.loss, ncol=1)
x=matrix(c(1,dat$Air.Flow),ncol=1)
#make the Z matrix
n=nrow(dat) #number of rows in our data file
k=1
#Z has n rows and 1 col for intercept, and n cols for the n air data points
#a list matrix allows us to combine "characters" and numbers
Z=matrix(list(0),n,k*n+1)
Z[,1]="alpha"
diag(Z[1:n,1+1:n])="beta"
#this function creates that permutation matrix for you
P=MARSS:::convert.model.mat(Z)$free[,,1]
M=kronecker(t(x),diag(n))%*%P
solve(t(M)%*%M)%*%t(M)%*%y
```

```
             [,1]
alpha -11.6159170
beta    0.6412918
```

```
coef(lm(dat$stack.loss ~ dat$Air.Flow))
```

```
 (Intercept) dat$Air.Flow
 -11.6159170    0.6412918
```

Go through this code line by line at the R command line. Look at `Z`. It is a list matrix that allows you to combine numbers (the 0s) with character string (names of parameters). Look at the permutation matrix `P`. Try `MARSS:::convert.model.mat(Z)$free` and see that it returns a 3D matrix, which is why the `[,,1]` appears (to get us a 2D matrix). To use more data points, you can redefine `dat` to say `dat=stackloss` to use all 21 data points.

2.4 Groups of intercepts

Let's say that the odd numbered plants are in the north and the even numbered are in the south. We want to include this as a factor in our model that affects the intercept. Let's go back to just having air flow be our explanatory variable. Now if the plant is in the north our model is

$$stack.loss_i = \alpha_n + \beta air_i + e_i, \text{ where } e_i \sim \text{N}(0, \sigma^2) \tag{2.16}$$

If the plant is in the south, our model is

$$stack.loss_i = \alpha_s + \beta air_i + e_i, \text{ where } e_i \sim \text{N}(0, \sigma^2) \tag{2.17}$$

We'll add north/south as a factor called 'reg' (region) to our dataframe:

```
dat = cbind(dat, reg = rep(c("n", "s"), n)[1:n])
dat
```

```
  Air.Flow Water.Temp Acid.Conc. stack.loss reg
1       80         27         89         42   n
2       80         27         88         37   s
3       75         25         90         37   n
4       62         24         87         28   s
```

And we can easily fit this model with `lm()`.

```
fit2 = lm(stack.loss ~ -1 + Air.Flow + reg, data = dat)
coef(fit2)
```

```
  Air.Flow       regn       regs
 0.5358166 -2.0257880 -5.5429799
```

The -1 is added to the `lm()` call to get rid of α. We just want the α_n and α_s intercepts coming from our regions.

2.4.1 North/South intercepts in Form 1

Written in matrix form, Form 1 for this model is

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} air_1 & 1 & 0 \\ air_2 & 0 & 1 \\ air_3 & 1 & 0 \\ air_4 & 0 & 1 \end{bmatrix} \begin{bmatrix} \beta \\ \alpha_n \\ \alpha_s \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} \tag{2.18}$$

Notice that odd plants get α_n and even plants get α_s. Use `model.matrix()` to see that this is the $\mathbf{Z}$ matrix that `lm()` formed. Notice the matrix output by `model.matrix()` looks exactly like $\mathbf{Z}$ in Equation (2.18).

```
Z = model.matrix(fit2)
Z[1:4, ]
```

```
  Air.Flow regn regs
1       80    1    0
2       80    0    1
3       75    1    0
4       62    0    1
```

We can solve for the parameters using $\mathbf{x} = (\mathbf{Z}^\top\mathbf{Z})^{-1}\mathbf{Z}^\top\mathbf{y}$ as we did for Form 1 before by adding on the 1s and 0s columns we see in the $\mathbf{Z}$ matrix in Equation (2.18). We could build this $\mathbf{Z}$ using the following R code:

```
Z = cbind(dat$Air.Flow, c(1, 0, 1, 0), c(0, 1, 0, 1))
colnames(Z) = c("beta", "regn", "regs")
```

Or just use `model.matrix()`. This will save time when models are more complex.

```
Z = model.matrix(fit2)
Z[1:4, ]
```

```
  Air.Flow regn regs
1       80    1    0
2       80    0    1
3       75    1    0
4       62    0    1
```

Now we can solve for the parameters:

```
y = matrix(dat$stack.loss, ncol = 1)
solve(t(Z) %*% Z) %*% t(Z) %*% y
```

```
               [,1]
Air.Flow  0.5358166
regn     -2.0257880
regs     -5.5429799
```

Compare to the output from `lm()` and you will see it is the same.

```
coef(fit2)
```

```
  Air.Flow       regn       regs
 0.5358166 -2.0257880 -5.5429799
```

2.4.2 North/South intercepts in Form 2

We would write this model in Form 2 as

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} \alpha_n & \beta & 0 & 0 & 0 \\ \alpha_s & 0 & \beta & 0 & 0 \\ \alpha_n & 0 & 0 & \beta & 0 \\ \alpha_s & 0 & 0 & 0 & \beta \end{bmatrix} \begin{bmatrix} 1 \\ air_1 \\ air_2 \\ air_3 \\ air_4 \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} = \mathbf{Zx} + \mathbf{e} \tag{2.19}$$

To estimate the parameters, we need to be able to write a list matrix that looks like $\mathbf{Z}$ in Equation (2.19). We can use the same code we used in Section 2.3.4 with $\mathbf{Z}$ changed to look like that in Equation (2.19).

```
y = matrix(dat$stack.loss, ncol = 1)
x = matrix(c(1, dat$Air.Flow), ncol = 1)
n = nrow(dat)
k = 1
# list matrix allows us to combine numbers and character
# strings
Z = matrix(list(0), n, k * n + 1)
Z[seq(1, n, 2), 1] = "alphanorth"
Z[seq(2, n, 2), 1] = "alphasouth"
diag(Z[1:n, 1 + 1:n]) = "beta"
P = MARSS:::convert.model.mat(Z)$free[, , 1]
M = kronecker(t(x), diag(n)) %*% P
solve(t(M) %*% M) %*% t(M) %*% y
```

```
                [,1]
alphanorth -2.0257880
alphasouth -5.5429799
beta        0.5358166
```

Make sure you understand the code used to form the **Z** matrix. Also notice that `class(Z[1,3])="numeric"` while `class(Z[1,2])="character"`. This is important. `0` in R is a number while `"0"` would be a character (the name of a parameter).

2.5 Groups of β's

Now let's say that the plants have different owners, Sue and Aneesh, and we want to have β for the air flow effect vary by owner. If the plant is in the north and owned by Sue, the model is

$$stack.loss_i = \alpha_n + \beta_s air_i + e_i, \text{ where } e_i \sim \text{N}(0, \sigma^2) \tag{2.20}$$

If it is in the south and owned by Aneesh, the model is

$$stack.loss_i = \alpha_s + \beta_a air_i + e_i, \text{ where } e_i \sim \text{N}(0, \sigma^2) \tag{2.21}$$

You get the idea.

Now we need to add an operator variable as a factor in our stackloss dataframe. Plants 1,3 are run by Sue and plants 2,4 are run by Aneesh.

```
dat = cbind(dat, owner = c("s", "a"))
dat
```

```
  Air.Flow Water.Temp Acid.Conc. stack.loss reg owner
1       80         27         89         42   n     s
2       80         27         88         37   s     a
3       75         25         90         37   n     s
4       62         24         87         28   s     a
```

Since the operator names can be replicated the length of our data set, R fills in the operator colmun by replicating our string of operator names to the right length, conveniently (or alarmingly).

We can easily fit this model with `lm()` using the ":" notation.

```
coef(lm(stack.loss ~ -1 + Air.Flow:owner + reg, data = dat))
```

```
          regn           regs Air.Flow:ownera Air.Flow:owners
         -38.0           -3.0            0.5             1.0
```

Notice that we have 4 datapoints and are estimating 4 parameters. We are not going to be able to estimate any more parameters than data points. If we want to estimate any more, we'll need to use the fuller stackflow dataset (which has 21 data points).

2.5.1 Owner β's in Form 1

Written in Form 1, this model is

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & air_1 \\ 0 & 1 & air_2 & 0 \\ 1 & 0 & 0 & air_3 \\ 0 & 1 & air_4 & 0 \end{bmatrix} \begin{bmatrix} \alpha_n \\ \alpha_s \\ \beta_a \\ \beta_s \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} = \mathbf{Zx} + \mathbf{e} \tag{2.22}$$

The air data have been written to the right of the 1s and 0s for north/south intercepts because that is how `lm()` writes this model in Form 1 and I want to duplicate that (for teaching purposes). Also the β's are ordered to be alphabetical because `lm()` writes the $\mathbf{Z}$ matrix like that.

Now our model is more complicated and using `model.matrix()` to get our $\mathbf{Z}$ saves us a lot tedious matrix building.

```
fit3 = lm(stack.loss ~ -1 + Air.Flow:owner + reg, data = dat)
Z = model.matrix(fit3)
Z[1:4, ]
```

```
  regn regs Air.Flow:ownera Air.Flow:owners
1    1    0               0              80
2    0    1              80               0
3    1    0               0              75
4    0    1              62               0
```

Notice the matrix output by `model.matrix()` looks exactly like **Z** in Equation (2.22) (ignore the attributes info). Now we can solve for the parameters:

```
y = matrix(dat$stack.loss, ncol = 1)
solve(t(Z) %*% Z) %*% t(Z) %*% y
```

```
                 [,1]
regn            -38.0
regs             -3.0
Air.Flow:ownera   0.5
Air.Flow:owners   1.0
```

Compare to the output from `lm()` and you will see it is the same.

2.5.2 Owner β's in Form 2

To write this model in Form 2, we just add subscripts to the β's in our Form 2 **Z** matrix:

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} \alpha_n & \beta_s & 0 & 0 & 0 \\ \alpha_s & 0 & \beta_a & 0 & 0 \\ \alpha_n & 0 & 0 & \beta_s & 0 \\ \alpha_s & 0 & 0 & 0 & \beta_a \end{bmatrix} \begin{bmatrix} 1 \\ air_1 \\ air_2 \\ air_3 \\ air_4 \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} = \mathbf{Zx} + \mathbf{e} \tag{2.23}$$

To estimate the parameters, we change the β's in our **Z** list matrix to have owner designations:

```
y = matrix(dat$stack.loss, ncol = 1)
x = matrix(c(1, dat$Air.Flow), ncol = 1)
n = nrow(dat)
k = 1
Z = matrix(list(0), n, k * n + 1)
Z[seq(1, n, 2), 1] = "alpha.n"
Z[seq(2, n, 2), 1] = "alpha.s"
diag(Z[1:n, 1 + 1:n]) = rep(c("beta.s", "beta.a"), n)[1:n]
P = MARSS:::convert.model.mat(Z)$free[, , 1]
M = kronecker(t(x), diag(n)) %*% P
solve(t(M) %*% M) %*% t(M) %*% y
```

```
         [,1]
alpha.n -38.0
alpha.s  -3.0
beta.s    1.0
beta.a    0.5
```

The parameters estimates are the same, though β's are given in reversed order simply due to the way `convert.model.mat()` is ordering the columns in Form 2's **Z**.

2.6 Seasonal effect as a factor

Let's imagine that the data were taken consecutively in time by quarter. We want to model the seasonal effect as an intercept change. We will drop all other effects for now. If the data were collected in quarter 1, the model is

$$stack.loss_i = \alpha_1 + e_i, \text{ where } e_i \sim \text{N}(0, \sigma^2) \tag{2.24}$$

If collected in quarter 2, the model is

$$stack.loss_i = \alpha_2 + e_i, \text{ where } e_i \sim \text{N}(0, \sigma^2) \tag{2.25}$$

etc.

We add a column to our dataframe to account for season:

```
dat = cbind(dat, qtr = paste(rep("qtr", n), 1:4, sep = ""))
dat
```

```
  Air.Flow Water.Temp Acid.Conc. stack.loss reg owner  qtr
1       80         27         89         42   n     s qtr1
2       80         27         88         37   s     a qtr2
3       75         25         90         37   n     s qtr3
4       62         24         87         28   s     a qtr4
```

And we can easily fit this model with `lm()`.

```
coef(lm(stack.loss ~ -1 + qtr, data = dat))
```

```
qtrqtr1 qtrqtr2 qtrqtr3 qtrqtr4
     42      37      37      28
```

The -1 is added to the `lm()` call to get rid of α. We just want the α_1, α_2, etc. intercepts coming from our quarters.

For comparison look at

```
coef(lm(stack.loss ~ qtr, data = dat))
```

```
(Intercept)     qtrqtr2     qtrqtr3     qtrqtr4
         42          -5          -5         -14
```

Why does it look like that when -1 is missing from the `lm()` call? Where did the intercept for quarter 1 go and why are the other intercepts so much smaller?

2.6.1 Seasonal intercepts written in Form 1

Remembering that `lm()` puts models in Form 1, look at the **Z** matrix for Form 1:

```
fit4 = lm(stack.loss ~ -1 + qtr, data = dat)
Z = model.matrix(fit4)
Z[1:4, ]
```

```
  qtrqtr1 qtrqtr2 qtrqtr3 qtrqtr4
1       1       0       0       0
2       0       1       0       0
3       0       0       1       0
4       0       0       0       1
```

Written in Form 1, this model is

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} = \mathbf{Zx} + \mathbf{e} \tag{2.26}$$

Compare to the model that `lm()` is using when the intercept included. What does this model look like written in matrix form?

```
fit5 = lm(stack.loss ~ qtr, data = dat)
Z = model.matrix(fit5)
Z[1:4, ]
```

```
  (Intercept) qtrqtr2 qtrqtr3 qtrqtr4
1           1       0       0       0
2           1       1       0       0
3           1       0       1       0
4           1       0       0       1
```

2.6.2 Seasonal intercepts written in Form 2 {#sec-mlr-season-form2

We do not need to add 1s and 0s to our $\mathbf{Z}$ matrix in Form 2; we just add subscripts to our intercepts like we did when we had north-south intercepts. In this model, we do not have any explanatory variables (except intercept) so our $\mathbf{x}$ is just a 1×1 matrix:

$$\begin{bmatrix} stack.loss_1 \\ stack.loss_2 \\ stack.loss_3 \\ stack.loss_4 \end{bmatrix} = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \end{bmatrix} \begin{bmatrix} 1 \end{bmatrix} + \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \end{bmatrix} = \mathbf{Zx} + \mathbf{e} \tag{2.27}$$

2.7 Seasonal effect plus other explanatory variables*

With our 4 data points, we are limited to estimating 4 parameters. Let's use the full 21 data points so we can estimate some more complex models. We'll add an owner variable and a quarter variable to the stackloss dataset.

```
data(stackloss)
fulldat = stackloss
n = nrow(fulldat)
fulldat = cbind(fulldat, owner = rep(c("sue", "aneesh", "joe"),
    n)[1:n], qtr = paste("qtr", rep(1:4, n)[1:n], sep = ""),
    reg = rep(c("n", "s"), n)[1:n])
```

Let's fit a model where there is only an effect of air flow, but that effect varies by owner and by quarter. We also want a different intercept for each quarter. So if datapoint i is from quarter j on a plant owned by owner k, the model is

$$stack.loss_i = \alpha_j + \beta_{j,k} air_i + e_i \tag{2.28}$$

So there there are 4×3 β's (4 quarters and 3 owners) and 4 α's (4 quarters).

With `lm()`, we fit the model as:

```
fit7 = lm(stack.loss ~ -1 + qtr + Air.Flow:qtr:owner, data = fulldat)
```

Take a look at $\mathbf{Z}$ for Form 1 using `model.matrix(Z)`. It's not shown since it is large:

```
model.matrix(fit7)
```

The $\mathbf{x}$ will be

$$\begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \alpha_4 \\ \beta_{1,a} \\ \beta_{2,a} \\ \beta_{3,a} \\ \dots \end{bmatrix}$$

Take a look at the model matrix that `lm()` is using and make sure you understand how $\mathbf{Zx}$ produces Equation (2.28).

```
Z = model.matrix(fit7)
```

For Form 2, our $\mathbf{Z}$ size doesn't change; number of rows is n (the number data points) and number of columns is 1 (for intercept) plus the number of explanatory variables times n. So in this case, we only have one explanatory variable (air flow) so $\mathbf{Z}$ has 1+21 columns. To allow the intercept to vary by quarter, we use α_1 in the rows of $\mathbf{Z}$ where the data is from

quarter 1, use α_2 where the data is from quarter 2, etc. Similarly we use the appropriate $\beta_{j,k}$ depending on the quarter and owner for that data point.

We could construct $\mathbf{Z}$, $\mathbf{x}$ and $\mathbf{y}$ for Form 2 using

```
y=matrix(fulldat$stack.loss, ncol=1)
x=matrix(c(1,fulldat$Air.Flow),ncol=1)
n=nrow(fulldat)
k=1
Z=matrix(list(0),n,k*n+1)
#give the intercepts names based on qtr
Z[,1]=paste(fulldat$qtr)
#give the betas names based on qtr and owner
diag(Z[1:n,1+1:n])=paste("beta",fulldat$qtr,fulldat$owner,sep=".")
P=MARSS:::convert.model.mat(Z)$free[,,1]
M=kronecker(t(x),diag(n))%*%P
solve(t(M)%*%M)%*%t(M)%*%y
```

Note, the estimates are the same as for `lm()` but are not listed in the same order.

Make sure to look at the $\mathbf{Z}$ and $\mathbf{x}$ for the models and that you understand why they look like they do.

2.8 Models with confounded parameters*

Try adding region as another factor in your model along with quarter and fit with `lm()`:

```
coef(lm(stack.loss ~ -1 + Air.Flow + reg + qtr, data = fulldat))
```

```
  Air.Flow       regn       regs    qtrqtr2    qtrqtr3    qtrqtr4
  1.066524 -49.024320 -44.831760  -3.066094   3.499428         NA
```

The estimate for quarter 1 is gone (actually it was set to 0) and the estimate for quarter 4 is NA. Look at the $\mathbf{Z}$ matrix for Form 1 and see if you can figure out the problem. Try also writing out the model for the 1st plant and you'll see what part of the problem is and why the estimate for quarter 1 is fixed at 0.

```
fit = lm(stack.loss ~ -1 + Air.Flow + reg + qtr, data = fulldat)
Z = model.matrix(fit)
```

But why is the estimate for quarter 4 equal to NA? What if the ordering of north and south regions was different, say 1 through 4 north, 5 through 8 south, 9 through 12 north, etc?

```
fulldat2 = fulldat
fulldat2$reg2 = rep(c("n", "n", "n", "n", "s", "s", "s", "s"),
    3)[1:21]
```

```
fit = lm(stack.loss ~ Air.Flow + reg2 + qtr, data = fulldat2)
coef(fit)
```

```
(Intercept)    Air.Flow       reg2s     qtrqtr2     qtrqtr3     qtrqtr4
-45.6158421   1.0407975  -3.5754722   0.7329027   3.0389763   3.6960928
```

Now an estimate for quarter 4 appears.

The problem is two-fold. First by having both region and quarter intercepts, we created models where 2 intercepts appear for one i model and we cannot estimate both. `lm()` helps us out by setting one of the factor effects to 0. It will chose the first alphabetically. But as we saw with the model where odd numbered plants were north and even numbered were south, we can still have a situation where one of the intercepts is non-identifiable. `lm()` helps us out by alerting us to the problem by setting one to NA.

Once you start writing your own Jags code or using MARSS, you will need to make sure that all your parameters are identifiable. If they are not, your code will simply 'chase its tail'. The code will generally take forever to converge or if you did not try different starting conditions, it may look like it converged but actually the estimates for the confounded parameters are meaningless. So you will need to think carefully about the model you are fitting and consider if there are multiple parameters measuring the same thing (for example 2 intercept parameters).

2.9 Problems

For the homework questions, we will using part of the `airquality` data set in R. Load that as

```
library(datasets)
data(airquality)
#remove any rows with NAs omitted.
airquality=na.omit(airquality)
#make Month a factor (i.e., the Month number is a name rather than a number)
airquality$Month=as.factor(airquality$Month)
#add a region factor
airquality$region = rep(c("north","south"),60)[1:111]
#Only use 5 data points for the homework so you can show the matrices easily
homeworkdat = airquality[1:5,]
```

1. Using Form 1 $\mathbf{y} = \mathbf{Z}\mathbf{x} + \mathbf{e}$, write the model being fit by this command

   ```
   fit = lm(Ozone ~ Wind + Temp, data = homeworkdat)
   ```

2. Build the $\mathbf{y}$ and $\mathbf{Z}$ matrices for the above model in R and solve for $\mathbf{x}$ (the parameters). Show that they match what you get from the first `lm()` call.

3. If you added -1 to your `lm()` call in question 1, what changes in your model?

   ```
   fit = lm(Ozone ~ -1 + Wind + Temp, data = homeworkdat)
   ```

4. Write the model for question 1 in Form 2. Adapt the code from subsection 2.3.4 to solve for the parameters. You will need to construct new Z, y and x in the code.

5. Model the ozone data with only a region (north/south) effect:

   ```
   fit = lm(Ozone ~ -1 + region, data = homeworkdat)
   ```

   Write this model in Form 1b (not Form 1) show that you can solve for the parameter values you get from the `lm()` call.

6. Write the model in question 5 in Form 2. Show that you can solve for the parameter values you get from the `lm()` call. Again to do this, you adapt the code from subsection 2.3.4.

7. Write the model below in Form 2.

   ```
   fit = lm(Ozone ~ Temp:region, data = homeworkdat)
   ```

8. Using the airquality dataset with 111 data points, write the model below in Form 2 and solve for the parametes by adapting code from subsection 2.3.4.

   ```
   fit = lm(Ozone ~ -1 + Temp:region + Month, data = airquality)
   ```

Chapter 3

Basic time-series functions in R

This chapter introduces you to some of the basic functions in R for plotting and analyzing univariate time series data. Many of the things you learn here will be relevant when we start examining multivariate time series as well. We will begin with the creation and plotting of time series objects in R, and then moves on to decomposition, differencing, and correlation (e.g., ACF, PACF) before ending with fitting and simulation of ARMA models.

A script with all the R code in the chapter can be downloaded here.

Data and packages

The chapter uses two data sets described in Section 3.1. The main one is a CO_2 data set: CO2_data.RData. The second is a Northern Hemisphere temperature data set: NHemiTemp_data.RData. The problems use a data set on hourly phytoplankton counts: hourly_phyto.RData. After downloading via the links, load the data

```
load("CO2_data.RData")
load("NHemiTemp_data.RData")
load("hourly_phyto.RData")
```

This chapter uses the **stats** package, which is often loaded by default when you start R, the **MARSS** package and the **forecast** package. The problems use a dataset in the **datasets** package. After installing the packages, if needed, load:

```
library(stats)
library(MARSS)
library(forecast)
library(datasets)
```

3.1 CO_2 and global temperature data set

For this chapter we will use the time series of the atmospheric concentration of CO_2 collected at the Mauna Loa Observatory in Hawai'i. We downloaded this data and created the `CO2` dataframe with the following code:

```
library(RCurl)
## get CO2 data from Mauna Loa observatory
ww1 <- "ftp://aftp.cmdl.noaa.gov/products/"
ww2 <- "trends/co2/co2_mm_mlo.txt"
CO2fulltext <- getURL(paste0(ww1, ww2))
CO2 <- read.table(text = CO2fulltext)[, c(1, 2, 5)]
## assign better column names
colnames(CO2) <- c("year", "month", "ppm")
save(CO2, CO2fulltext, file = "CO2_data.RData")
```

The data file contains some extra information that we don't need, so we only read in a subset of the columns (1, 2 & 5). `CO2fulltext` contains information on the data. Use `cat(CO2fulltext)` to read.

The second data set we will use is Northern Hemisphere land and ocean temperature anomalies from NOAA. We downloaded this data set with the following code:

```
library(RCurl)
ww1 <- "https://www.ncdc.noaa.gov/cag/time-series/"
ww2 <- "global/nhem/land_ocean/p12/12/1880-2014.csv"
Temp <- read.csv(text = getURL(paste0(ww1, ww2)), skip = 4)
save(Temp, file = "NHemiTemp_data.RData")
```

3.2 Time series plots

Time series plots are an excellent way to begin the process of understanding what sort of process might have generated the data of interest. Traditionally, time series have been plotted with the observed data on the y-axis and time on the x-axis. Sequential time points are usually connected with some form of line, but sometimes other plot forms can be a useful way of conveying important information in the time series (e.g., barplots of sea-surface temperature anomolies show nicely the contrasting El Niño and La Niña phenomena).

3.2.1 ts objects and `plot.ts()`

The CO_2 data are stored in R as a `data.frame` object, but we would like to transform the class to a more user-friendly format for dealing with time series. Fortunately, the `ts()` function will do just that, and return an object of class **ts** as well. In addition to the data

themselves, we need to provide `ts()` with 2 pieces of information about the time index for the data.

The first, `frequency`, is a bit of a misnomer because it does not really refer to the number of cycles per unit time, but rather the number of observations/samples per cycle. So, for example, if the data were collected each hour of a day then `frequency=24`.

The second, `start`, specifies the first sample in terms of (*day*, *hour*), (*year*, *month*), etc. So, for example, if the data were collected monthly beginning in November of 1969, then `frequency=12` and `start=c(1969,11)`. If the data were collected annually, then you simply specify `start` as a scalar (e.g., `start=1991`) and omit `frequency` (i.e., R will set `frequency=1` by default).

The Mauna Loa time series is collected monthly and begins in March of 1958, which we can get from the data themselves, and then pass to `ts()`.

```
## create a time series (ts) object from the CO2 data
co2 <- ts(data = CO2$ppm, frequency = 12, start = c(CO2[1, "year"],
    CO2[1, "month"]))
```

Now let's plot the data using `plot.ts()`, which is designed specifically for **ts** objects like the one we just created above. It's nice because we don't need to specify any x-values as they are taken directly from the **ts** object.

```
## plot the ts
plot.ts(co2, ylab = expression(paste("CO"[2], " (ppm)")))
```

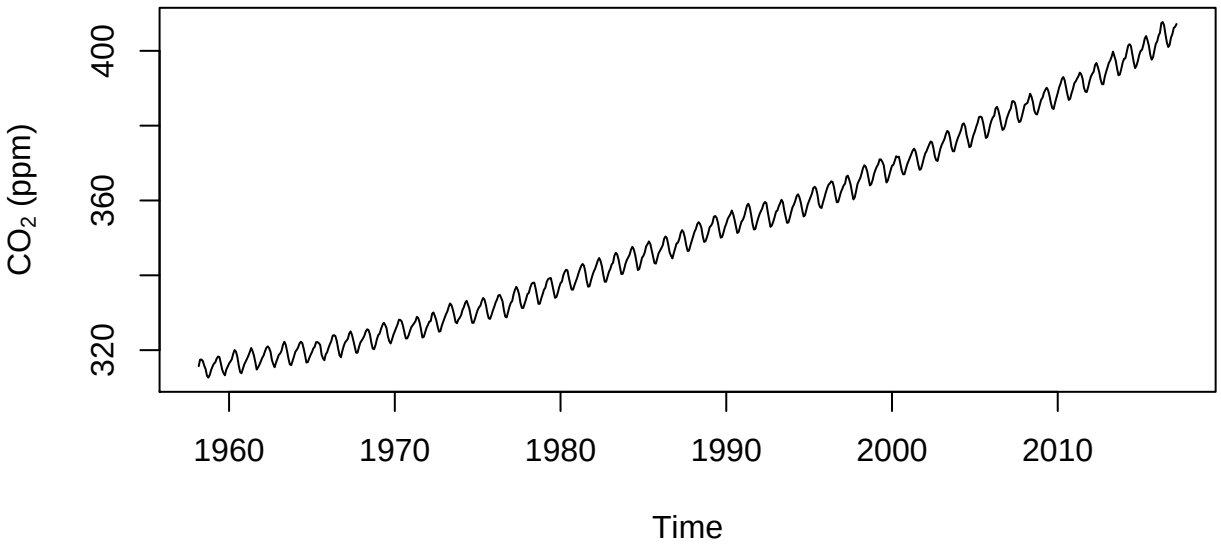


Figure 3.1: Time series of the atmospheric CO_2 concentration at Mauna Loa, Hawai'i measured monthly from March 1958 to present.

Examination of the plotted time series (Figure 3.1) shows 2 obvious features that would violate any assumption of stationarity: 1) an increasing (and perhaps non-linear) trend

over time, and 2) strong seasonal patterns. (Aside: Do you know the causes of these 2 phenomena?)

3.2.2 Combining and plotting multiple ts objects

Before we examine the CO_2 data further, however, let's see a quick example of how you can combine and plot multiple time series together. We'll use the data on monthly mean temperature anomolies for the Northern Hemisphere (`Temp`). First convert `Temp` to a `ts` object.

```
temp.ts <- ts(data = Temp$Value, frequency = 12, start = c(1880,
    1))
```

Before we can plot the two time series together, however, we need to line up their time indices because the temperature data start in January of 1880, but the CO_2 data start in March of 1958. Fortunately, the `ts.intersect()` function makes this really easy once the data have been transformed to **ts** objects by trimming the data to a common time frame. Also, `ts.union()` works in a similar fashion, but it pads one or both series with the appropriate number of NA's. Let's try both.

```
## intersection (only overlapping times)
datI <- ts.intersect(co2, temp.ts)
## dimensions of common-time data
dim(datI)
```

```
[1] 682   2
```

```
## union (all times)
datU <- ts.union(co2, temp.ts)
## dimensions of all-time data
dim(datU)
```

```
[1] 1647    2
```

As you can see, the intersection of the two data sets is much smaller than the union. If you compare them, you will see that the first 938 rows of `datU` contains `NA` in the `co2` column.

It turns out that the regular `plot()` function in R is smart enough to recognize a **ts** object and use the information contained therein appropriately. Here's how to plot the intersection of the two time series together with the y-axes on alternate sides (results are shown in Figure 3.2):

```
## plot the ts
plot(datI, main = "", yax.flip = TRUE)
```

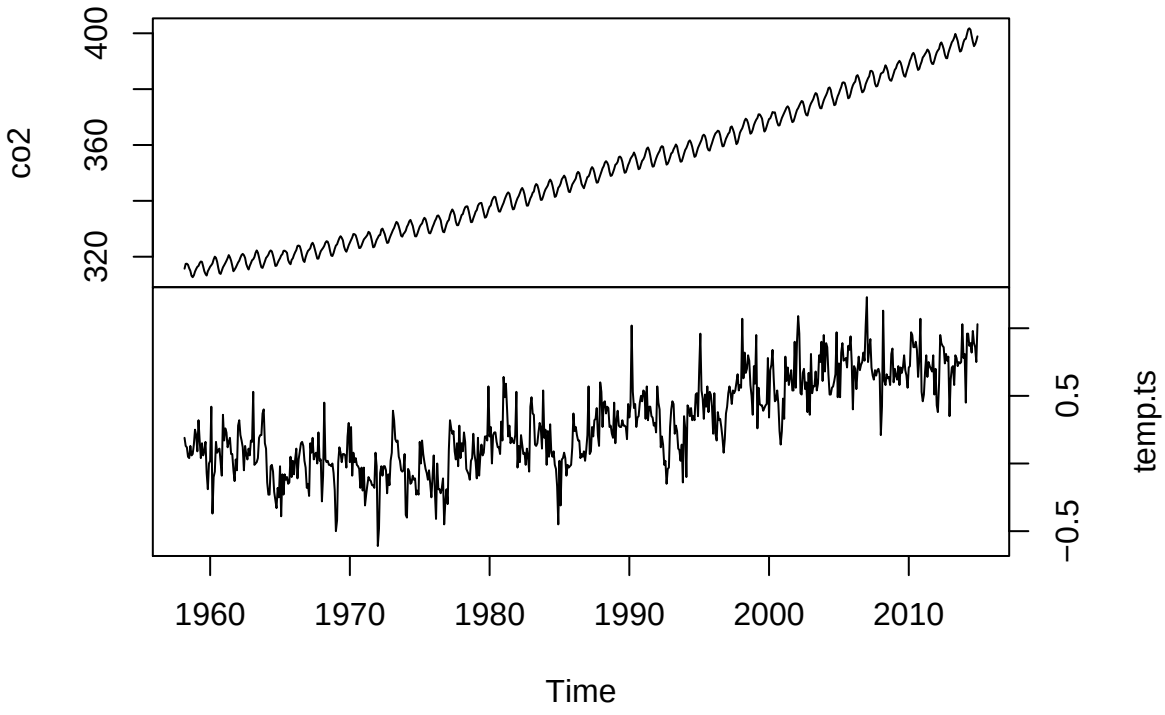


Figure 3.2: Time series of the atmospheric CO_2 concentration at Mauna Loa, Hawai'i (top) and the mean temperature index for the Northern Hemisphere (bottom) measured monthly from March 1958 to present.

3.3 Decomposition of time series

Plotting time series data is an important first step in analyzing their various components. Beyond that, however, we need a more formal means for identifying and removing characteristics such as a trend or seasonal variation. As discussed in lecture, the decomposition model reduces a time series into 3 components: trend, seasonal effects, and random errors. In turn, we aim to model the random errors as some form of stationary process.

Let's begin with a simple, additive decomposition model for a time series x_t

$$x_t = m_t + s_t + e_t, \tag{3.1}$$

where, at time t, m_t is the trend, s_t is the seasonal effect, and e_t is a random error that we generally assume to have zero-mean and to be correlated over time. Thus, by estimating and subtracting both $\{m_t\}$ and $\{s_t\}$ from $\{x_t\}$, we hope to have a time series of stationary residuals $\{e_t\}$.

3.3.1 Estimating trends

In lecture we discussed how linear filters are a common way to estimate trends in time series. One of the most common linear filters is the moving average, which for time lags from $-a$ to a is defined as

$$\hat{m}_t = \sum_{k=-a}^{a} \left(\frac{1}{1+2a} \right) x_{t+k}. \tag{3.2}$$

This model works well for moving windows of odd-numbered lengths, but should be adjusted for even-numbered lengths by adding only $\frac{1}{2}$ of the 2 most extreme lags so that the filtered value at time t lines up with the original observation at time t. So, for example, in a case with monthly data such as the atmospheric CO_2 concentration where a 12-point moving average would be an obvious choice, the linear filter would be

$$\hat{m}_t = \frac{\frac{1}{2}x_{t-6} + x_{t-5} + \cdots + x_{t-1} + x_t + x_{t+1} + \cdots + x_{t+5} + \frac{1}{2}x_{t+6}}{12} \tag{3.3}$$

It is important to note here that our time series of the estimated trend $\{\hat{m}_t\}$ is actually shorter than the observed time series by $2a$ units.

Conveniently, R has the built-in function `filter()` for estimating moving-average (and other) linear filters. In addition to specifying the time series to be filtered, we need to pass in the filter weights (and 2 other arguments we won't worry about here–type `?filter` to get more information). The easiest way to create the filter is with the `rep()` function:

```
## weights for moving avg
fltr <- c(1/2, rep(1, times = 11), 1/2)/12
```

Now let's get our estimate of the trend $\{\hat{m}\}$ with `filter()`} and plot it:

```
## estimate of trend
co2.trend <- filter(co2, filter = fltr, method = "convo", sides = 2)
## plot the trend
plot.ts(co2.trend, ylab = "Trend", cex = 1)
```

The trend is a more-or-less smoothly increasing function over time, the average slope of which does indeed appear to be increasing over time as well (Figure 3.3).

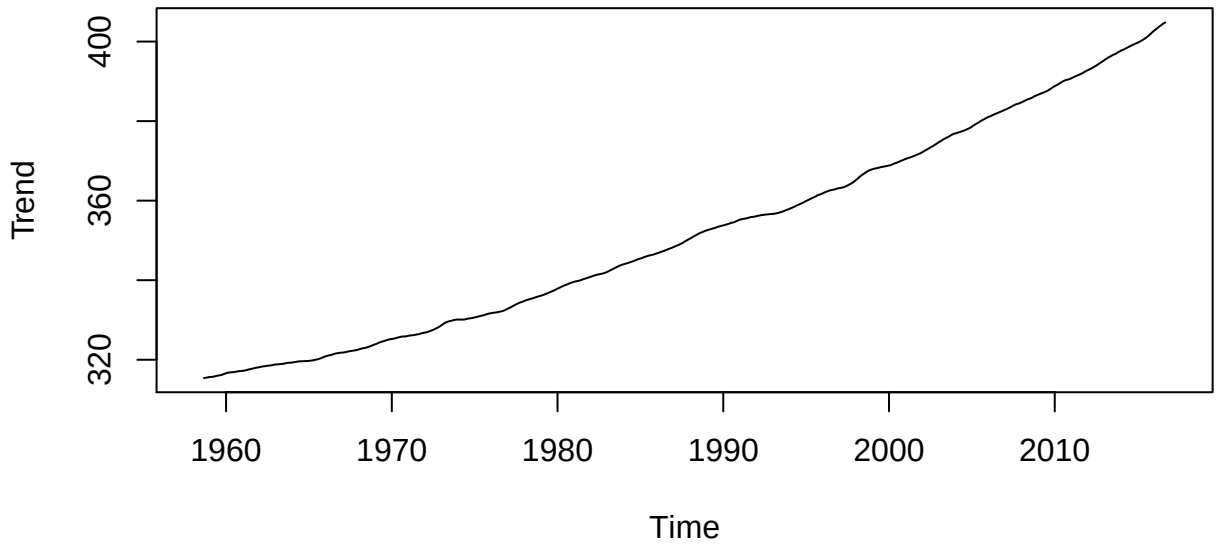


Figure 3.3: Time series of the estimated trend $\{\hat{m}_t\}$ for the atmospheric CO_2 concentration at Mauna Loa, Hawai'i.

3.3.2 Estimating seasonal effects

Once we have an estimate of the trend for time t ($\hat{m}_t$) we can easily obtain an estimate of the seasonal effect at time t ($\hat{s}_t$) by subtraction

$$\hat{s}_t = x_t - \hat{m}_t, \tag{3.4}$$

which is really easy to do in R:

```
## seasonal effect over time
co2.1T <- co2 - co2.trend
```

This estimate of the seasonal effect for each time t also contains the random error e_t, however, which can be seen by plotting the time series and careful comparison of Equations (3.1) and (3.4).

```
## plot the monthly seasonal effects
plot.ts(co2.1T, ylab = "Seasonal effect", xlab = "Month", cex = 1)
```

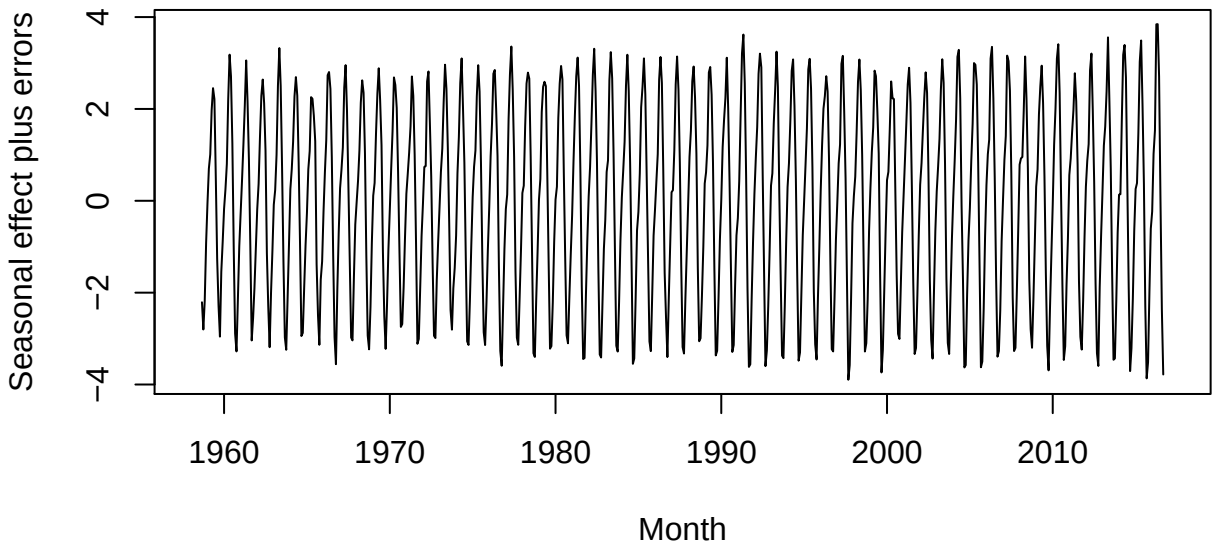


Figure 3.4: Time series of seasonal effects plus random errors for the atmospheric CO_2 concentration at Mauna Loa, Hawai'i, measured monthly from March 1958 to present.

We can obtain the overall seasonal effect by averaging the estimates of $\{\hat{s}_t\}$ for each month and repeating this sequence over all years.

```
## length of ts
ll <- length(co2.1T)
## frequency (ie, 12)
ff <- frequency(co2.1T)
## number of periods (years); %/% is integer division
periods <- ll%/%ff
## index of cumulative month
index <- seq(1, ll, by = ff) - 1
## get mean by month
mm <- numeric(ff)
for (i in 1:ff) {
    mm[i] <- mean(co2.1T[index + i], na.rm = TRUE)
}
## subtract mean to make overall mean=0
mm <- mm - mean(mm)
```

Before we create the entire time series of seasonal effects, let's plot them for each month to see what is happening within a year:

```
## plot the monthly seasonal effects
plot.ts(mm, ylab = "Seasonal effect", xlab = "Month", cex = 1)
```

It looks like, on average, that the CO_2 concentration is highest in spring (March) and lowest in summer (August) (Figure 3.5). (Aside: Do you know why this is?)

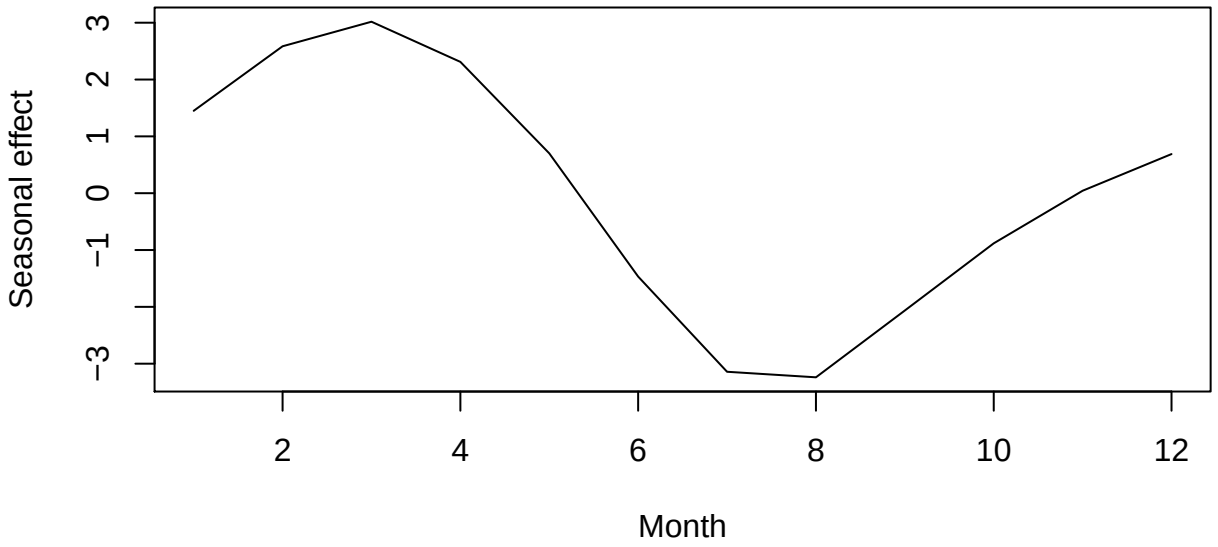


Figure 3.5: Estimated monthly seasonal effects for the atmospheric CO_2 concentration at Mauna Loa, Hawai'i.

Finally, let's create the entire time series of seasonal effects $\{\hat{s}_t\}$:

```
## create ts object for season
co2.seas <- ts(rep(mm, periods + 1)[seq(ll)], start = start(co2.1T),
    frequency = ff)
```

3.3.3 Completing the model

The last step in completing our full decomposition model is obtaining the random errors $\{\hat{e}_t\}$, which we can get via simple subtraction

$$\hat{e}_t = x_t - \hat{m}_t - \hat{s}_t. \tag{3.5}$$

Again, this is really easy in R:

```
## random errors over time
co2.err <- co2 - co2.trend - co2.seas
```

Now that we have all 3 of our model components, let's plot them together with the observed data $\{x_t\}$. The results are shown in Figure 3.6.

```
## plot the obs ts, trend & seasonal effect
plot(cbind(co2, co2.trend, co2.seas, co2.err), main = "", yax.flip = TRUE)
```

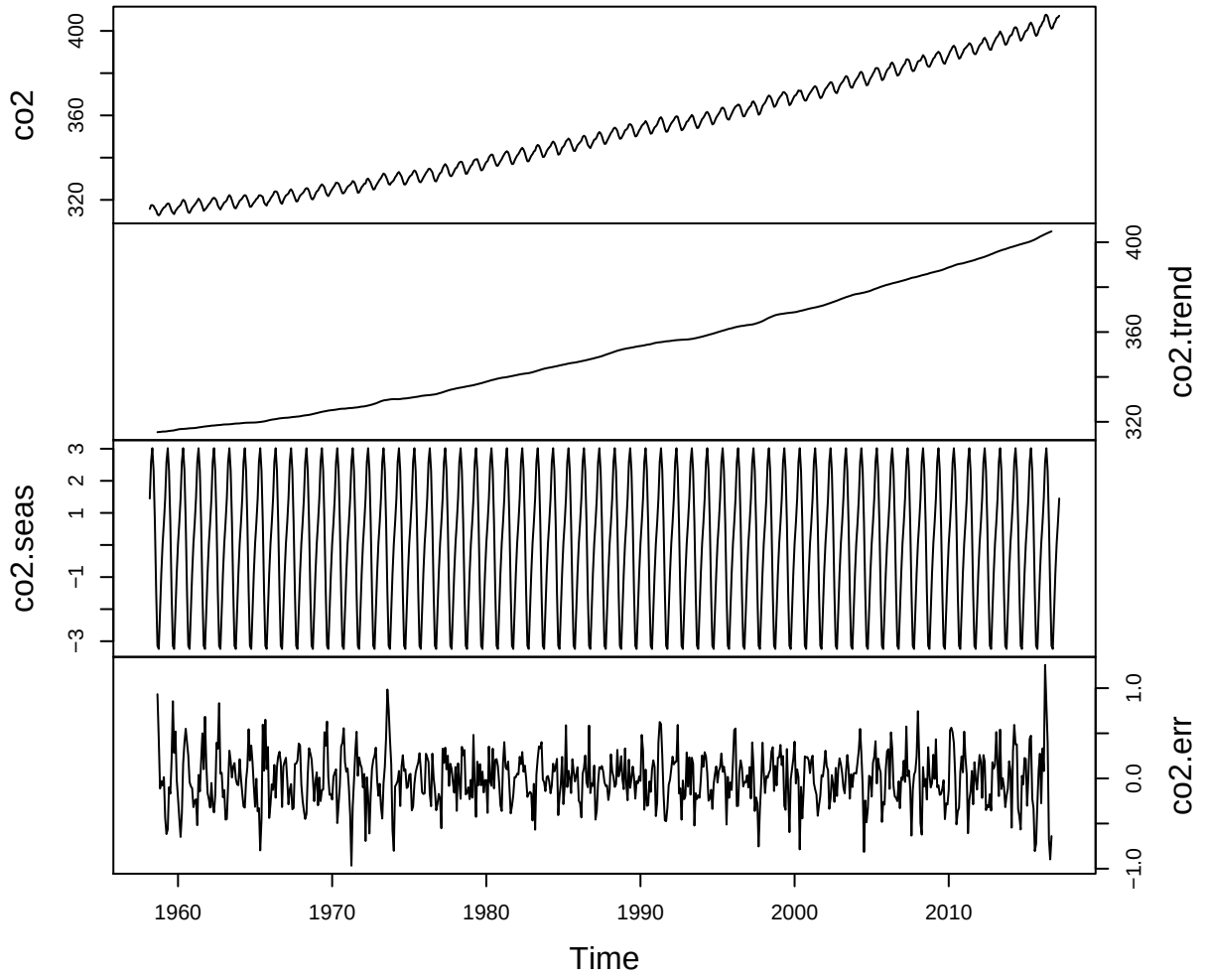


Figure 3.6: Time series of the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i (top) along with the estimated trend, seasonal effects, and random errors.

3.3.4 Using `decompose()` for decomposition

Now that we have seen how to estimate and plot the various components of a classical decomposition model in a piecewise manner, let's see how to do this in one step in R with the function `decompose()`, which accepts a **ts** object as input and returns an object of class **decomposed.ts**.

```
## decomposition of CO2 data
co2.decomp <- decompose(co2)
```

`co2.decomp` is a list with the following elements, which should be familiar by now:

- `x` the observed time series $\{x_t\}$
- `seasonal` time series of estimated seasonal component $\{\hat{s}_t\}$
- `figure` mean seasonal effect (`length(figure) == frequency(x)`)
- `trend` time series of estimated trend $\{\hat{m}_t\}$
- `random` time series of random errors $\{\hat{e}_t\}$
- `type` type of error (`"additive"` or `"multiplicative"`)

We can easily make plots of the output and compare them to those in Figure 3.6:

```
## plot the obs ts, trend & seasonal effect
plot(co2.decomp, yax.flip = TRUE)
```

The results obtained with `decompose()` (Figure 3.7) are identical to those we estimated previously.

Another nice feature of the `decompose()` function is that it can be used for decomposition models with multiplicative (i.e., non-additive) errors (e.g., if the original time series had a seasonal amplitude that increased with time). To do, so pass in the argument `type="multiplicative"`, which is set to `type="additive"` by default.

3.4 Differencing to remove a trend or seasonal effects

An alternative to decomposition for removing trends is differencing. We saw in lecture how the difference operator works and how it can be used to remove linear and nonlinear trends as well as various seasonal features that might be evident in the data. As a reminder, we define the difference operator as

$$\nabla x_t = x_t - x_{t-1}, \tag{3.6}$$

and, more generally, for order d

$$\nabla^d x_t = (1 - \mathbf{B})^d x_t, \tag{3.7}$$

where $\mathbf{B}$ is the backshift operator (i.e., $\mathbf{B}^k x_t = x_{t-k}$ for $k \geq 1$).

So, for example, a random walk is one of the most simple and widely used time series models, but it is not stationary. We can write a random walk model as

$$x_t = x_{t-1} + w_t, \text{ with } w_t \sim \mathrm{N}(0, q). \tag{3.8}$$

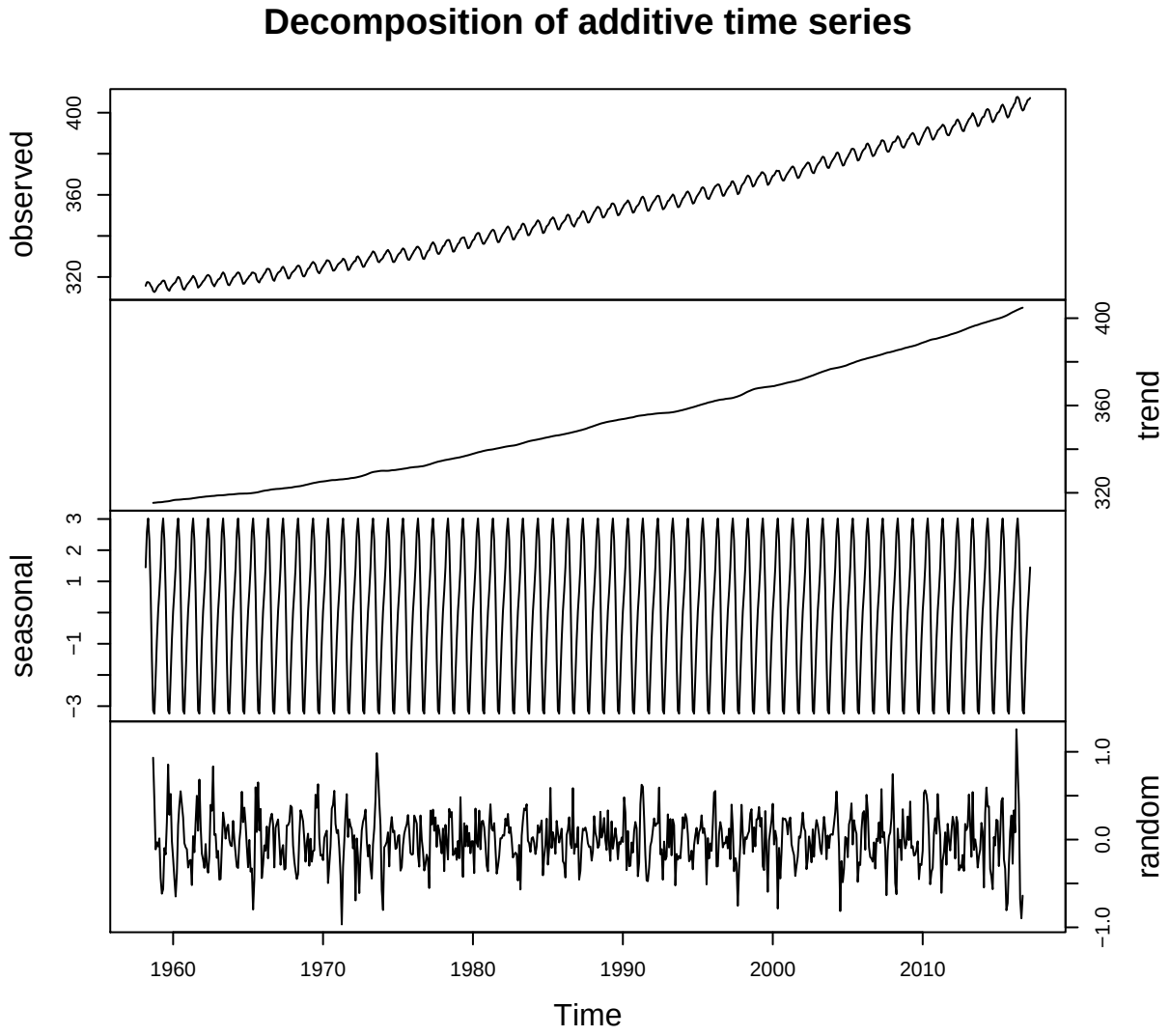


Figure 3.7: Time series of the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i (top) along with the estimated trend, seasonal effects, and random errors obtained with the function `decompose()`.

Applying the difference operator to Equation (3.8) will yield a time series of Gaussian white noise errors $\{w_t\}$:

$$\begin{aligned} \nabla(x_t &= x_{t-1} + w_t) \\ x_t - x_{t-1} &= x_{t-1} - x_{t-1} + w_t \\ x_t - x_{t-1} &= w_t \end{aligned} \tag{3.9}$$

3.4.1 Using the `diff()` function

In R we can use the `diff()` function for differencing a time series, which requires 3 arguments: `x` (the data), `lag` (the lag at which to difference), and `differences` (the order of differencing; d in Equation (3.7)). For example, first-differencing a time series will remove a linear trend (i.e., `differences=1`); twice-differencing will remove a quadratic trend (i.e., `differences=2`). In addition, first-differencing a time series at a lag equal to the period will remove a seasonal trend (e.g., set `lag=12` for monthly data).

Let's use `diff()` to remove the trend and seasonal signal from the CO_2 time series, beginning with the trend. Close inspection of Figure 3.1 would suggest that there is a nonlinear increase in CO_2 concentration over time, so we'll set `differences=2`):

```
## twice-difference the CO2 data
co2.D2 <- diff(co2, differences = 2)
## plot the differenced data
plot(co2.D2, ylab = expression(paste(nabla^2, "CO"[2])))
```

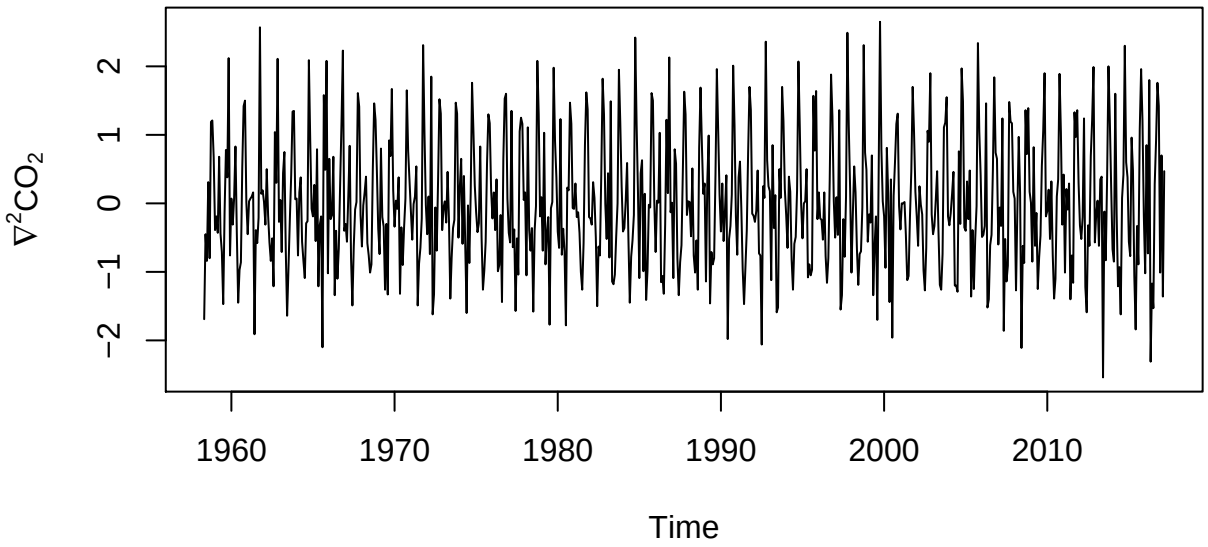


Figure 3.8: Time series of the twice-differenced atmospheric CO_2 concentration at Mauna Loa, Hawai'i.

We were apparently successful in removing the trend, but the seasonal effect still appears obvious (Figure 3.8). Therefore, let's go ahead and difference that series at lag-12 because our data were collected monthly.

```
## difference the differenced CO2 data
co2.D2D12 <- diff(co2.D2, lag = 12)
## plot the newly differenced data
plot(co2.D2D12, ylab = expression(paste(nabla, "(", nabla^2,
    "CO"[2], ")")))
```

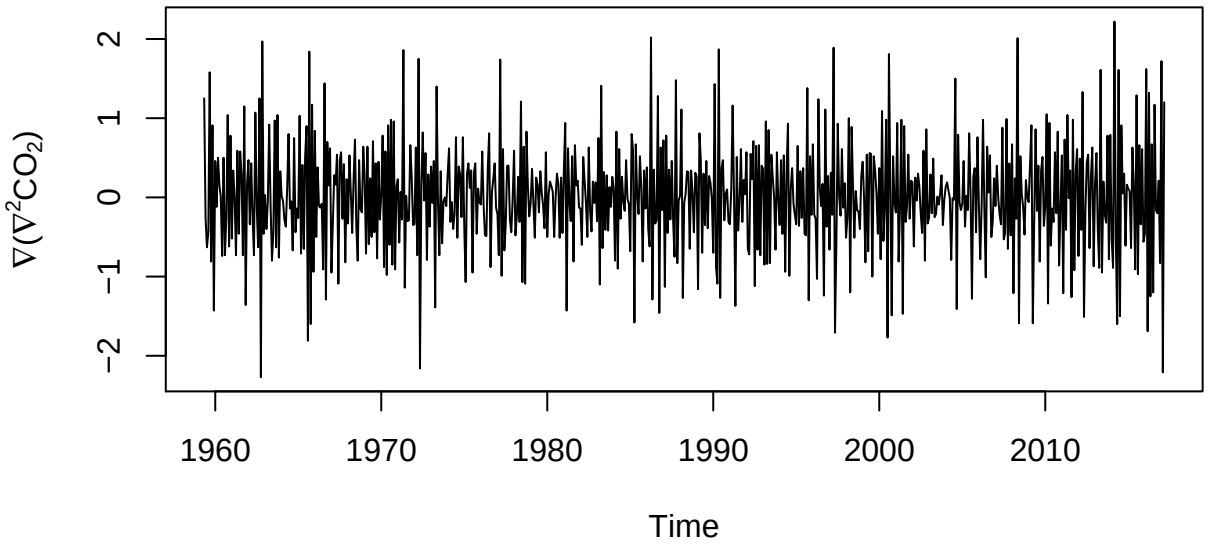


Figure 3.9: Time series of the lag-12 difference of the twice-differenced atmospheric CO_2 concentration at Mauna Loa, Hawai'i.

Now we have a time series that appears to be random errors without any obvious trend or seasonal components (Figure 3.9).

3.5 Correlation within and among time series

The concepts of covariance and correlation are very important in time series analysis. In particular, we can examine the correlation structure of the original data or random errors from a decomposition model to help us identify possible form(s) of (non)stationary model(s) for the stochastic process.

3.5.1 Autocorrelation function (ACF)

Autocorrelation is the correlation of a variable with itself at differing time lags. Recall from lecture that we defined the sample autocovariance function (ACVF), c_k, for some lag k as

$$c_k = \frac{1}{n} \sum_{t=1}^{n-k} (x_t - \bar{x})(x_{t+k} - \bar{x}) \tag{3.10}$$

Note that the sample autocovariance of $\{x_t\}$ at lag 0, c_0, equals the sample variance of $\{x_t\}$ calculated with a denominator of n. The sample autocorrelation function (ACF) is defined as

$$r_k = \frac{c_k}{c_0} = \text{Cor}(x_t, x_{t+k}) \tag{3.11}$$

Recall also that an approximate 95% confidence interval on the ACF can be estimated by

$$-\frac{1}{n} \pm \frac{2}{\sqrt{n}} \tag{3.12}$$

where n is the number of data points used in the calculation of the ACF.

It is important to remember two things here. First, although the confidence interval is commonly plotted and interpreted as a horizontal line over all time lags, the interval itself actually grows as the lag increases because the number of data points n used to estimate the correlation decreases by 1 for every integer increase in lag. Second, care must be exercised when interpreting the "significance" of the correlation at various lags because we should expect, a priori, that approximately 1 out of every 20 correlations will be significant based on chance alone.

We can use the `acf()` function in R to compute the sample ACF (note that adding the option `type="covariance"` will return the sample auto-covariance (ACVF) instead of the ACF–type `?acf` for details). Calling the function by itself will will automatically produce a correlogram (i.e., a plot of the autocorrelation versus time lag). The argument `lag.max` allows you to set the number of positive and negative lags. Let's try it for the CO_2 data.

```
## correlogram of the CO2 data
acf(co2, lag.max = 36)
```

There are 4 things about Figure 3.10 that are noteworthy:

1. the ACF at lag 0, r_0, equals 1 by default (i.e., the correlation of a time series with itself)–it's plotted as a reference point;
2. the x-axis has decimal values for lags, which is caused by R using the year index as the lag rather than the month;
3. the horizontal blue lines are the approximate 95% CI's; and
4. there is very high autocorrelation even out to lags of 36 months.

As an alternative to the default plots for **acf** objects, let's define a new plot function for **acf** objects with some better features:

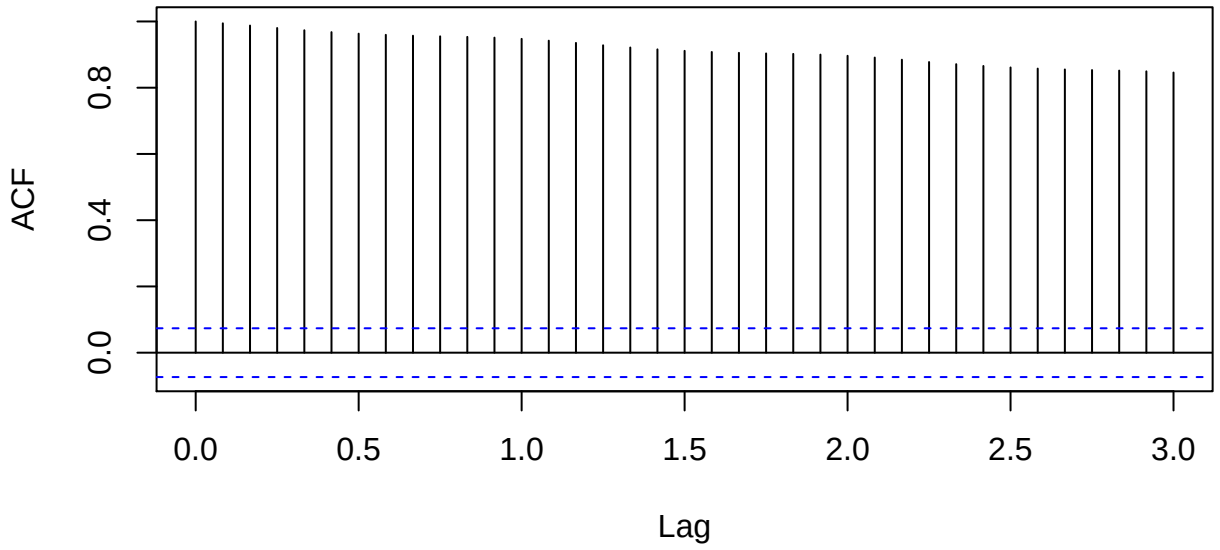
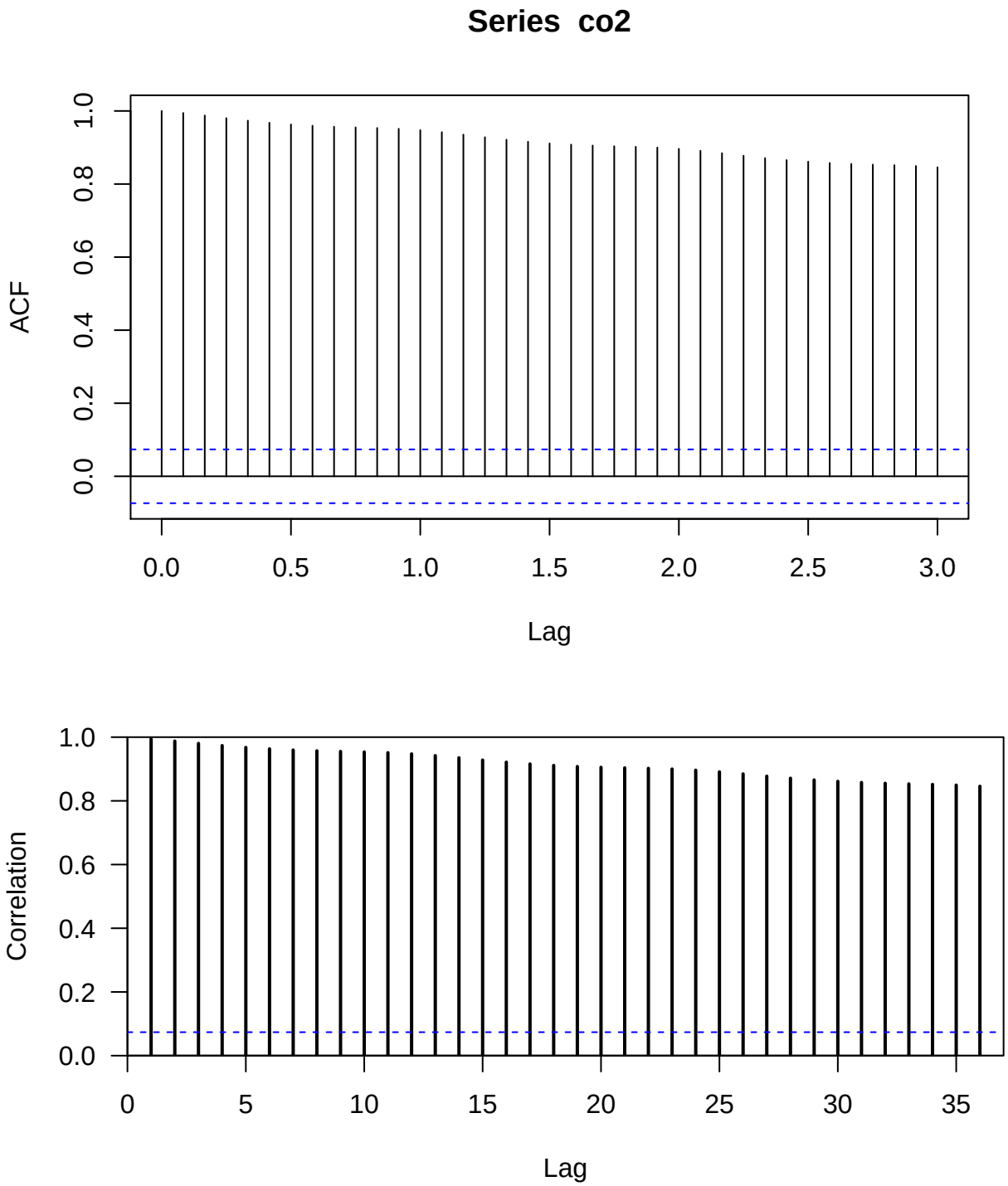


Figure 3.10: Correlogram of the observed atmospheric CO_2 concentration at Mauna Loa, Hawai'i obtained with the function `acf()`.

```
## better ACF plot
plot.acf <- function(ACFobj) {
    rr <- ACFobj$acf[-1]
    kk <- length(rr)
    nn <- ACFobj$n.used
    plot(seq(kk), rr, type = "h", lwd = 2, yaxs = "i", xaxs = "i",
        ylim = c(floor(min(rr)), 1), xlim = c(0, kk + 1), xlab = "Lag",
        ylab = "Correlation", las = 1)
    abline(h = -1/nn + c(-2, 2)/sqrt(nn), lty = "dashed", col = "blue")
    abline(h = 0)
}
```

Now we can assign the result of `acf()` to a variable and then use the information contained therein to plot the correlogram with our new plot function.

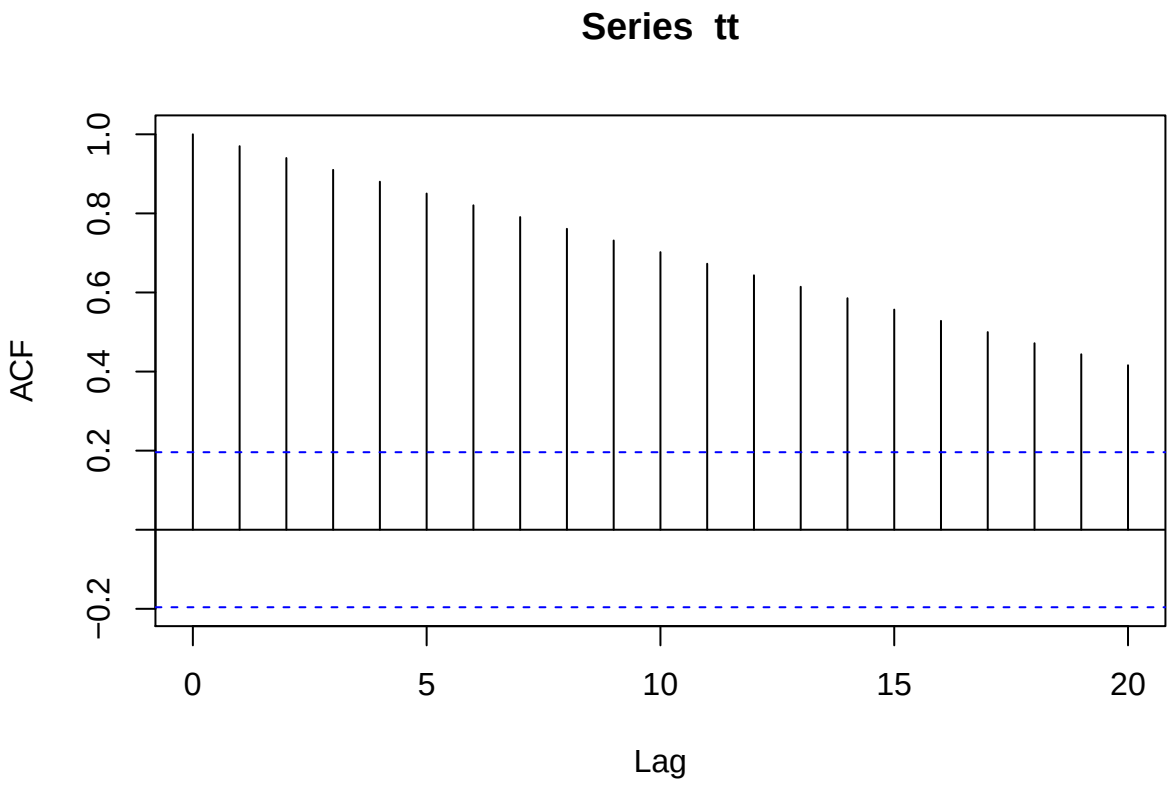
```
## acf of the CO2 data
co2.acf <- acf(co2, lag.max = 36)
## correlogram of the CO2 data
plot.acf(co2.acf)
```

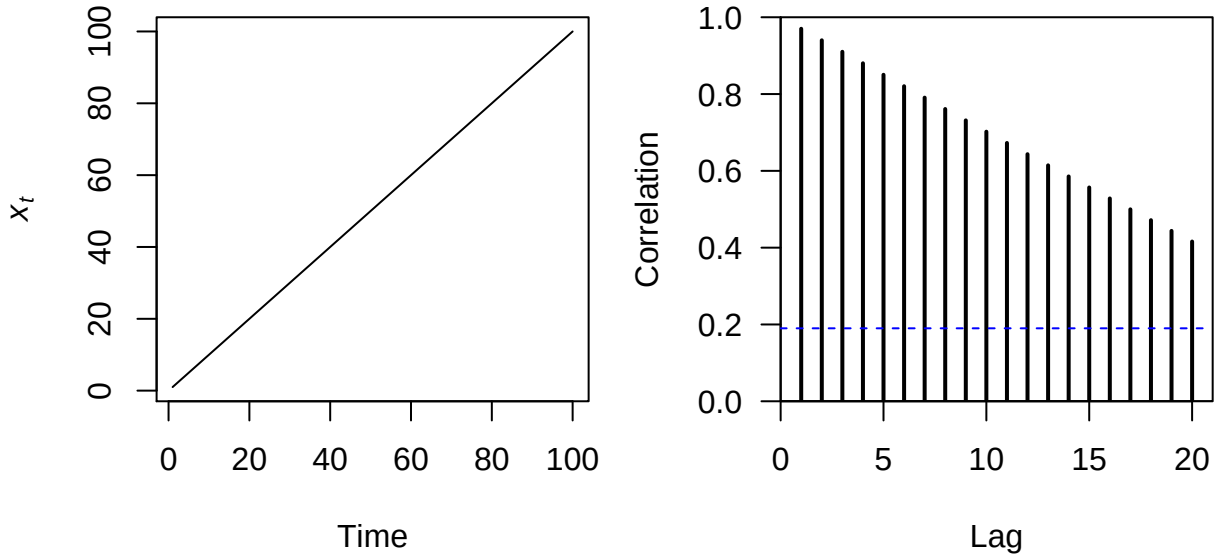


Notice that all of the relevant information is still there (Figure **??**), but now $r_0 = 1$ is not plotted at lag-0 and the lags on the x-axis are displayed correctly as integers.

Before we move on to the PACF, let's look at the ACF for some deterministic time series, which will help you identify interesting properties (e.g., trends, seasonal effects) in a stochastic time series, and account for them in time series models–an important topic in this course. First, let's look at a straight line.

```
## length of ts
nn <- 100
## create straight line
tt <- seq(nn)
## set up plot area
par(mfrow = c(1, 2))
## plot line
plot.ts(tt, ylab = expression(italic(x[t])))
## get ACF
line.acf <- acf(tt, plot = FALSE)
## plot ACF
plot.acf(line.acf)
```

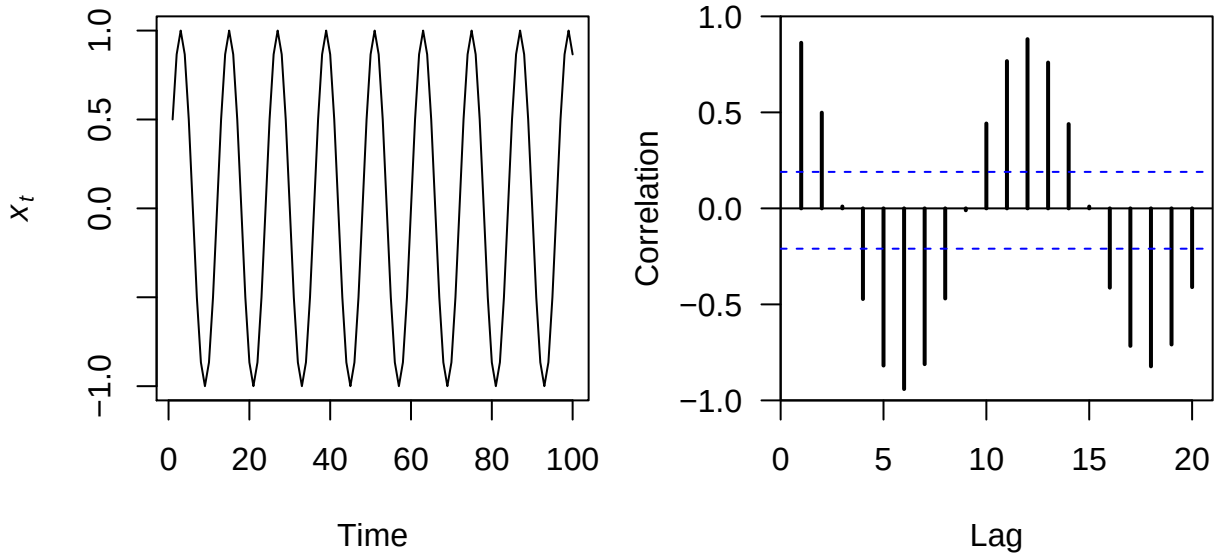




The correlogram for a straight line is itself a linearly decreasing function over time (Figure ??).

Now let's examine the ACF for a sine wave and see what sort of pattern arises.

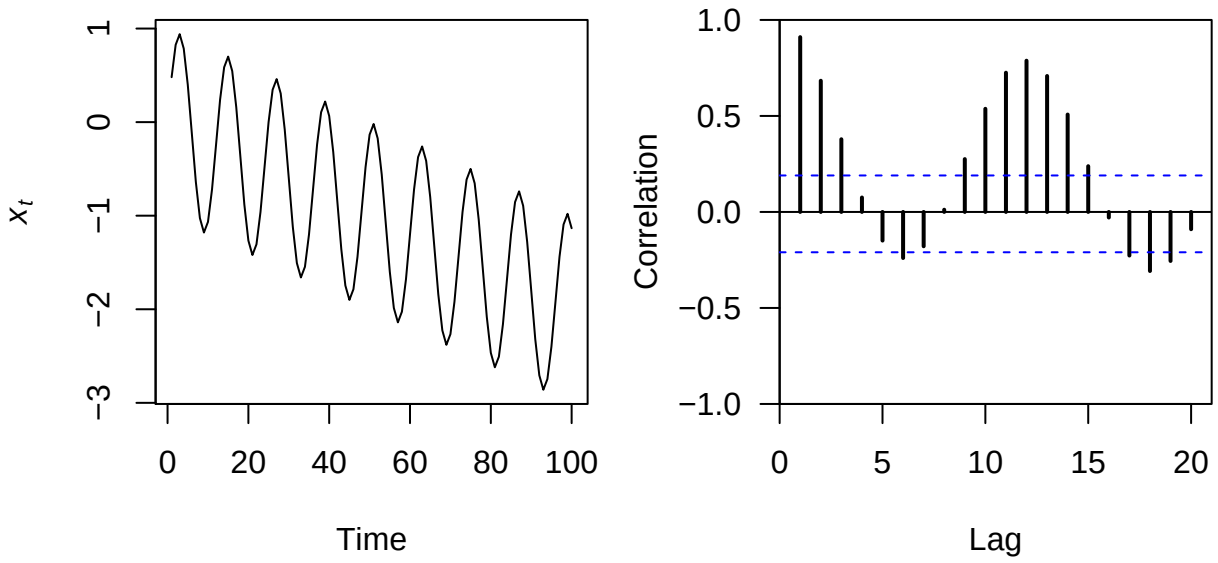
```
## create sine wave
tt <- sin(2 * pi * seq(nn)/12)
## set up plot area
par(mfrow = c(1, 2))
## plot line
plot.ts(tt, ylab = expression(italic(x[t])))
## get ACF
sine.acf <- acf(tt, plot = FALSE)
## plot ACF
plot.acf(sine.acf)
```



Perhaps not surprisingly, the correlogram for a sine wave is itself a sine wave whose amplitude decreases linearly over time (Figure **??**).

Now let's examine the ACF for a sine wave with a linear downward trend and see what sort of patterns arise.

```
## create sine wave with trend
tt <- sin(2 * pi * seq(nn)/12) - seq(nn)/50
## set up plot area
par(mfrow = c(1, 2))
## plot line
plot.ts(tt, ylab = expression(italic(x[t])))
## get ACF
sili.acf <- acf(tt, plot = FALSE)
## plot ACF
plot.acf(sili.acf)
```



The correlogram for a sine wave with a trend is itself a nonsymmetrical sine wave whose amplitude and center decrease over time (Figure **??**).

As we have seen, the ACF is a powerful tool in time series analysis for identifying important features in the data. As we will see later, the ACF is also an important diagnostic tool for helping to select the proper order of p and q in ARMA(p,q) models.

3.5.2 Partial autocorrelation function (PACF)

The partial autocorrelation function (PACF) measures the linear correlation of a series $\{x_t\}$ and a lagged version of itself $\{x_{t+k}\}$ with the linear dependence of $\{x_{t-1}, x_{t-2}, \ldots, x_{t-(k-1)}\}$ removed. Recall from lecture that we define the PACF as

$$f_k = \begin{cases} \text{Cor}(x_1, x_0) = r_1 & \text{if } k = 1; \\ \text{Cor}(x_k - x_k^{k-1}, x_0 - x_0^{k-1}) & \text{if } k \geq 2; \end{cases} \tag{3.13}$$

with

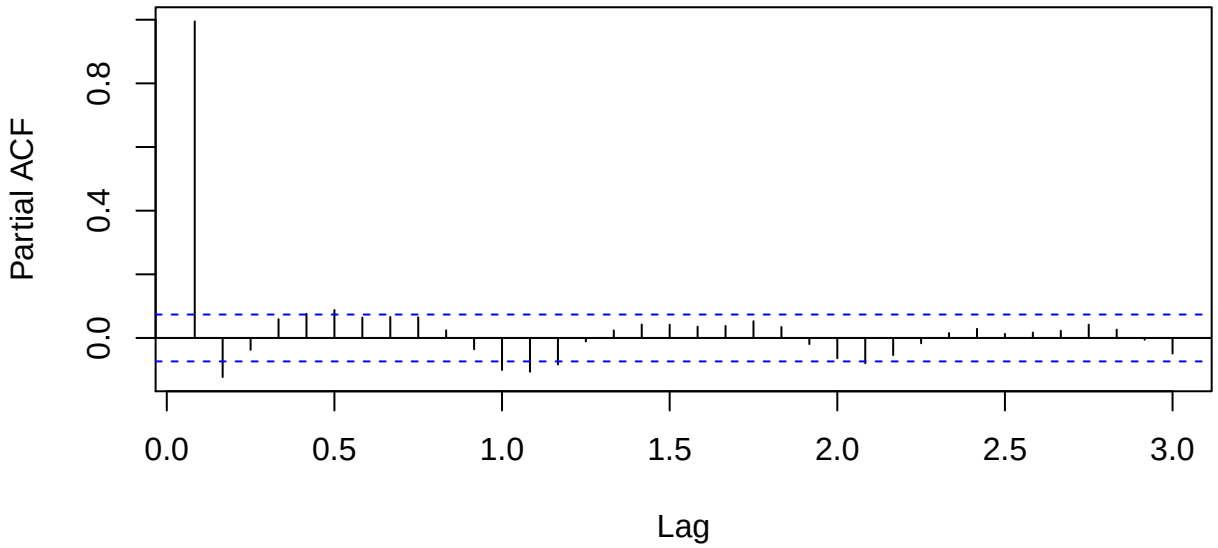
$$x_k^{k-1} = \beta_1 x_{k-1} + \beta_2 x_{k-2} + \cdots + \beta_{k-1} x_1; \tag{3.14a}$$
$$x_0^{k-1} = \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_{k-1} x_{k-1}. \tag{3.14b}$$

It's easy to compute the PACF for a variable in R using the `pacf()` function, which will automatically plot a correlogram when called by itself (similar to `acf()`). Let's look at the PACF for the CO_2 data.

```
## PACF of the CO2 data
pacf(co2, lag.max = 36)
```

The default plot for PACF is a bit better than for ACF, but here is another plotting function that might be useful.

```
## better PACF plot
plot.pacf <- function(PACFobj) {
    rr <- PACFobj$acf
    kk <- length(rr)
    nn <- PACFobj$n.used
    plot(seq(kk), rr, type = "h", lwd = 2, yaxs = "i", xaxs = "i",
        ylim = c(floor(min(rr)), 1), xlim = c(0, kk + 1), xlab = "Lag",
        ylab = "PACF", las = 1)
    abline(h = -1/nn + c(-2, 2)/sqrt(nn), lty = "dashed", col = "blue")
    abline(h = 0)
}
```



Notice in Figure **??** that the partial autocorrelation at lag-1 is very high (it equals the ACF at lag-1), but the other values at lags > 1 are relatively small, unlike what we saw for the ACF. We will discuss this in more detail later on in this lab.

Notice also that the PACF plot again has real-valued indices for the time lag, but it does not include any value for lag-0 because it is impossible to remove any intermediate autocorrelation between t and $t-k$ when $k = 0$, and therefore the PACF does not exist at lag-0. If you would like, you can use the `plot.acf()` function we defined above to plot the PACF estimates because `acf()` and `pacf()` produce identical list structures (results not shown here).

```
## PACF of the CO2 data
co2.pacf <- pacf(co2)
## correlogram of the CO2 data
plot.acf(co2.pacf)
```

As with the ACF, we will see later on how the PACF can also be used to help identify the appropriate order of p and q in ARMA(p,q) models.

3.5.3 Cross-correlation function (CCF)

Often we are interested in looking for relationships between 2 different time series. There are many ways to do this, but a simple method is via examination of their cross-covariance and cross-correlation.

We begin by defining the sample cross-covariance function (CCVF) in a manner similar to the ACVF, in that

$$g_k^{xy} = \frac{1}{n}\sum_{t=1}^{n-k}\left(y_t - \bar{y}\right)\left(x_{t+k} - \bar{x}\right), \tag{3.15}$$

but now we are estimating the correlation between a variable y and a different time-shifted variable x_{t+k}. The sample cross-correlation function (CCF) is then defined analogously to the ACF, such that

$$r_k^{xy} = \frac{g_k^{xy}}{\sqrt{\mathrm{SD}_x \mathrm{SD}_y}}; \tag{3.16}$$

SD_x and SD_y are the sample standard deviations of $\{x_t\}$ and $\{y_t\}$, respectively. It is important to re-iterate here that $r_k^{xy} \neq r_{-k}^{xy}$, but $r_k^{xy} = r_{-k}^{yx}$. Therefore, it is very important to pay particular attention to which variable you call y (i.e., the "response") and which you call x (i.e., the "predictor").

As with the ACF, an approximate 95% confidence interval on the CCF can be estimated by

$$-\frac{1}{n} \pm \frac{2}{\sqrt{n}} \tag{3.17}$$

where n is the number of data points used in the calculation of the CCF, and the same assumptions apply to its interpretation.

Computing the CCF in R is easy with the function `ccf()` and it works just like `acf()`. In fact, `ccf()` is just a "wrapper" function that calls `acf()`. As an example, let's examine the CCF between sunspot activity and number of lynx trapped in Canada as in the classic paper by Moran[1].

To begin, let's get the data, which are conveniently included in the **datasets** package included as part of the base installation of R. Before calculating the CCF, however, we need to find the matching years of data. Again, we'll use the `ts.intersect()` function.

```
## get the matching years of sunspot data
suns <- ts.intersect(lynx, sunspot.year)[, "sunspot.year"]
## get the matching lynx data
lynx <- ts.intersect(lynx, sunspot.year)[, "lynx"]
```

Here are plots of the time series.

```
## plot time series
plot(cbind(suns, lynx), yax.flip = TRUE)
```

It is important to remember which of the 2 variables you call y and x when calling `ccf(x, y, ...)`. In this case, it seems most relevant to treat lynx as the y and sunspots as the x, in which case we are mostly interested in the CCF at negative lags (i.e., when sunspot activity predates inferred lynx abundance). Furthermore, we'll use log-transformed lynx trappings.

```
## CCF of sunspots and lynx
ccf(suns, log(lynx), ylab = "Cross-correlation")
```

[1] Moran, P.A.P. 1949. The statistical analysis of the sunspot and lynx cycles. *J. Anim. Ecol.* 18:115-116

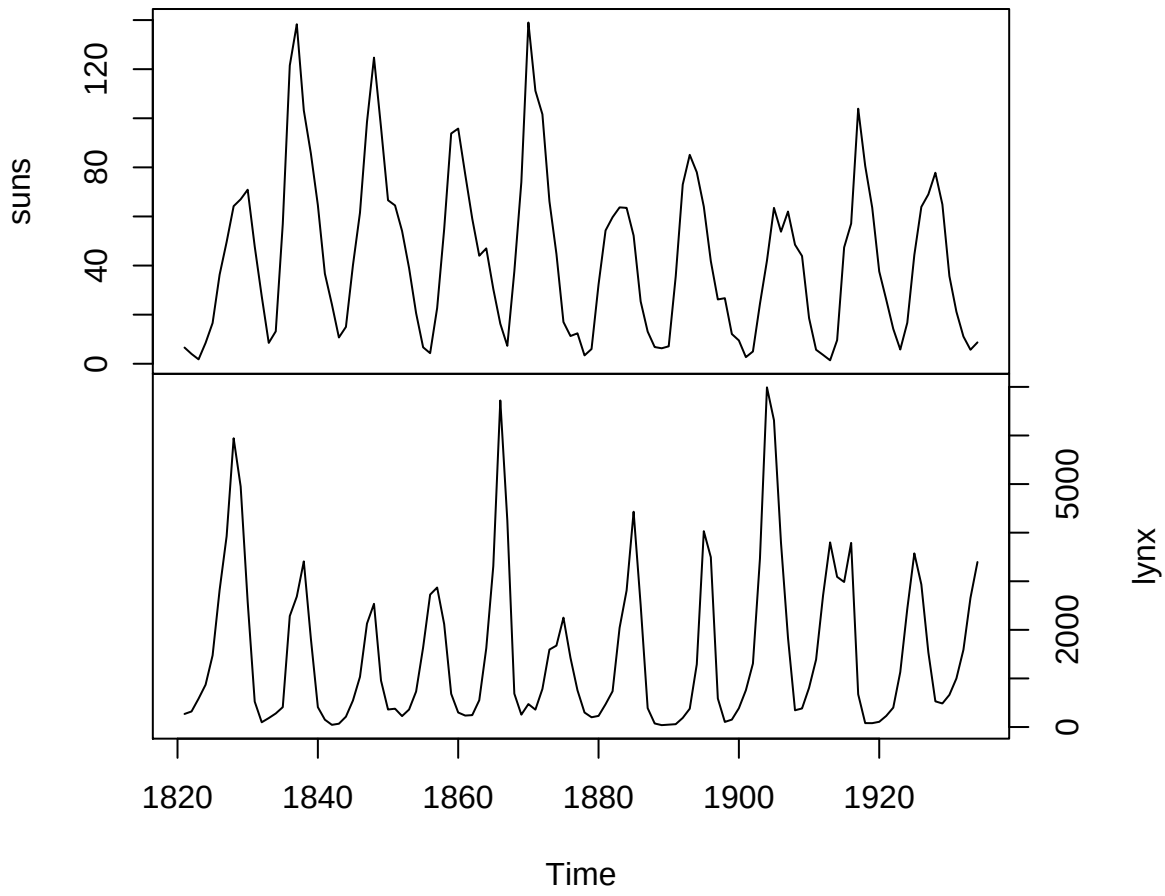


Figure 3.11: Time series of sunspot activity (top) and lynx trappings in Canada (bottom) from 1821-1934.

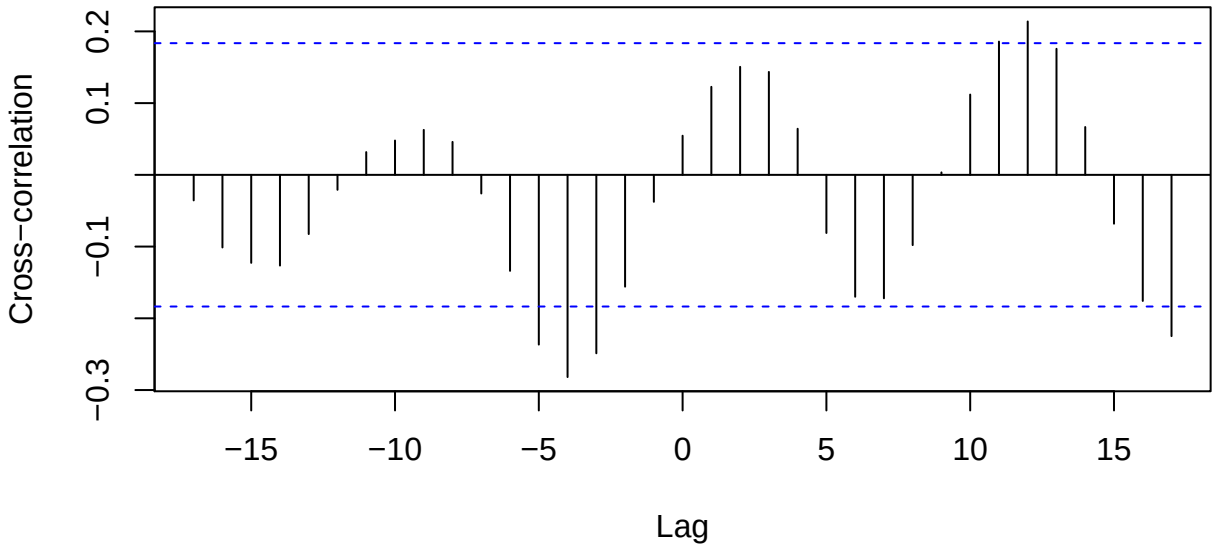


Figure 3.12: CCF for annual sunspot activity and the log of the number of lynx trappings in Canada from 1821-1934.

From Figures 3.11 and 3.12 it looks like lynx numbers are relatively low 3-5 years after high sunspot activity (i.e., significant correlation at lags of -3 to -5).

3.6 White noise (WN)

A time series $\{w_t\}$ is a discrete white noise series (DWN) if the $w_1, w_1, \ldots, w_t$ are independent and identically distributed (IID) with a mean of zero. For most of the examples in this course we will assume that the $w_t \sim \mathrm{N}(0, q)$, and therefore we refer to the time series $\{w_t\}$ as Gaussian white noise. If our time series model has done an adequate job of removing all of the serial autocorrelation in the time series with trends, seasonal effects, etc., then the model residuals ($e_t = y_t - \hat{y}_t$) will be a WN sequence with the following properties for its mean ($\bar{e}$), covariance (c_k), and autocorrelation (r_k):

$$\begin{aligned} \bar{x} &= 0 \\ c_k &= \mathrm{Cov}(e_t, e_{t+k}) = \begin{cases} q & \text{if } k = 0 \\ 0 & \text{if } k \neq 1 \end{cases} \\ r_k &= \mathrm{Cor}(e_t, e_{t+k}) = \begin{cases} 1 & \text{if } k = 0 \\ 0 & \text{if } k \neq 1. \end{cases} \end{aligned} \tag{3.18}$$

3.6.1 Simulating white noise

Simulating WN in R is straightforward with a variety of built-in random number generators for continuous and discrete distributions. Once you know R's abbreviation for the distribution of interest, you add an `r` to the beginning to get the function's name. For example, a Gaussian (or normal) distribution is abbreviated `norm` and so the function is `rnorm()`. All of the random number functions require two things: the number of samples from the distribution and the parameters for the distribution itself (e.g., mean & SD of a normal). Check the help file for the distribution of interest to find out what parameters you must specify (e.g., type `?rnorm` to see the help for a normal distribution).

Here's how to generate 100 samples from a normal distribution with mean of 5 and standard deviation of 0.2, and 50 samples from a Poisson distribution with a rate (λ) of 20.

```
set.seed(123)
## random normal variates
GWN <- rnorm(n = 100, mean = 5, sd = 0.2)
## random Poisson variates
PWN <- rpois(n = 50, lambda = 20)
```

Here are plots of the time series. Notice that on one occasion the same number was drawn twice in a row from the Poisson distribution, which is discrete. That is virtually guaranteed to never happen with a continuous distribution.

```
## set up plot region
par(mfrow = c(1, 2))
## plot normal variates with mean
plot.ts(GWN)
abline(h = 5, col = "blue", lty = "dashed")
## plot Poisson variates with mean
plot.ts(PWN)
abline(h = 20, col = "blue", lty = "dashed")
```

Now let's examine the ACF for the 2 white noise series and see if there is, in fact, zero autocorrelation for lags ≥ 1.

```
## set up plot region
par(mfrow = c(1, 2))
## plot normal variates with mean
acf(GWN, main = "", lag.max = 20)
## plot Poisson variates with mean
acf(PWN, main = "", lag.max = 20)
```

Interestingly, the r_k are all greater than zero in absolute value although they are not statistically different from zero for lags 1-20. This is because we are dealing with a sample of the distributions rather than the entire population of all random variates. As an exercise, try setting `n=1e6` instead of `n=100` or `n=50` in the calls calls above to generate the WN se-

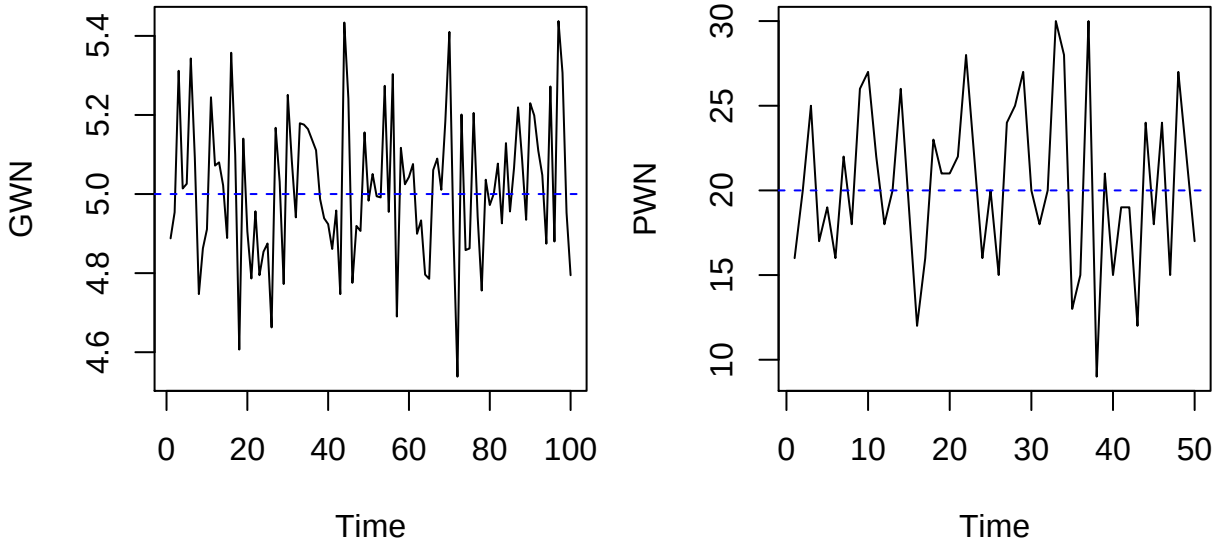


Figure 3.13: Time series plots of simulated Gaussian (left) and Poisson (right) white noise.

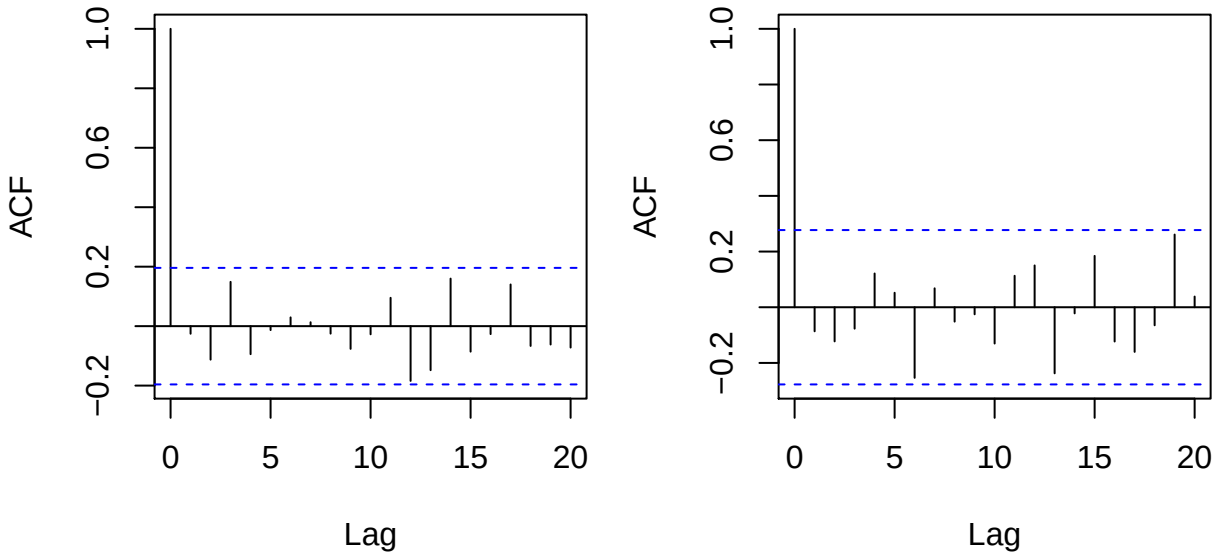


Figure 3.14: ACF's for the simulated Gaussian (left) and Poisson (right) white noise shown in Figure 3.13.

quences and see what effect it has on the estimation of r_k. It is also important to remember, as we discussed earlier, that we should expect that approximately 1 in 20 of the r_k will be statistically greater than zero based on chance alone, especially for relatively small sample sizes, so don't get too excited if you ever come across a case like then when inspecting model residuals.

3.7 Random walks (RW)

Random walks receive considerable attention in time series analyses because of their ability to fit a wide range of data despite their surprising simplicity. In fact, random walks are the most simple non-stationary time series model. A random walk is a time series $\{x_t\}$ where

$$x_t = x_{t-1} + w_t, \tag{3.19}$$

and w_t is a discrete white noise series where all values are independent and identically distributed (IID) with a mean of zero. In practice, we will almost always assume that the w_t are Gaussian white noise, such that $w_t \sim \mathrm{N}(0, q)$. We will see later that a random walk is a special case of an autoregressive model.

3.7.1 Simulating a random walk

Simulating a RW model in R is straightforward with a for loop and the use of `rnorm()` to generate Gaussian errors (type `?rnorm` to see details on the function and its useful relatives `dnorm()` and `pnorm()`). Let's create 100 obs (we'll also set the random number seed so everyone gets the same results).

```
## set random number seed
set.seed(123)
## length of time series
TT <- 100
## initialize {x_t} and {w_t}
xx <- ww <- rnorm(n = TT, mean = 0, sd = 1)
## compute values 2 thru TT
for (t in 2:TT) {
    xx[t] <- xx[t - 1] + ww[t]
}
```

Now let's plot the simulated time series and its ACF.

```
## setup plot area
par(mfrow = c(1, 2))
## plot line
plot.ts(xx, ylab = expression(italic(x[t])))
```

```
## plot ACF
plot.acf(acf(xx, plot = FALSE))
```

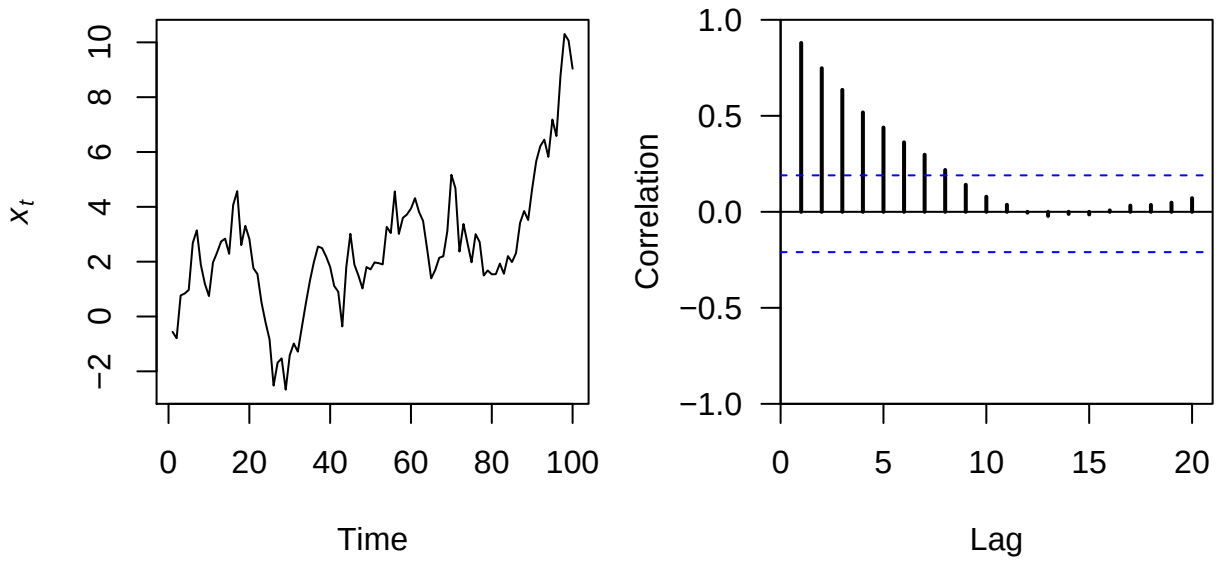


Figure 3.15: Simulated time series of a random walk model (left) and its associated ACF (right).

Perhaps not surprisingly based on their names, autoregressive models such as RW's have a high degree of autocorrelation out to long lags (Figure 3.15).

3.7.2 Alternative formulation of a random walk

As an aside, let's use an alternative formulation of a random walk model to see an even shorter way to simulate an RW in R. Based on our definition of a random walk in Equation (3.19), it is easy to see that

$$\begin{aligned} x_t &= x_{t-1} + w_t \\ x_{t-1} &= x_{t-2} + w_{t-1} \\ x_{t-2} &= x_{t-3} + w_{t-2} \\ &\vdots \end{aligned} \tag{3.20}$$

Therefore, if we substitute $x_{t-2} + w_{t-1}$ for x_{t-1} in the first equation, and then $x_{t-3} + w_{t-2}$ for x_{t-2}, and so on in a recursive manner, we get

$$x_t = w_t + w_{t-1} + w_{t-2} + \cdots + w_{t-\infty} + x_{t-\infty}. \tag{3.21}$$

In practice, however, the time series will not start an infinite time ago, but rather at some $t = 1$, in which case we can write

$$\begin{aligned} x_t &= w_1 + w_2 + \cdots + w_t \\ &= \sum_{t=1}^{T} w_t. \end{aligned} \tag{3.22}$$

From Equation (3.22) it is easy to see that the value of an RW process at time step t is the sum of all the random errors up through time t. Therefore, in R we can easily simulate a realization from an RW process using the `cumsum(x)` function, which does cumulative summation of the vector `x` over its entire length. If we use the same errors as before, we should get the same results.

```
## simulate RW
x2 <- cumsum(ww)
```

Let's plot both time series to see if it worked.

```
## setup plot area
par(mfrow = c(1, 2))
## plot 1st RW
plot.ts(xx, ylab = expression(italic(x[t])))
## plot 2nd RW
plot.ts(x2, ylab = expression(italic(x[t])))
```

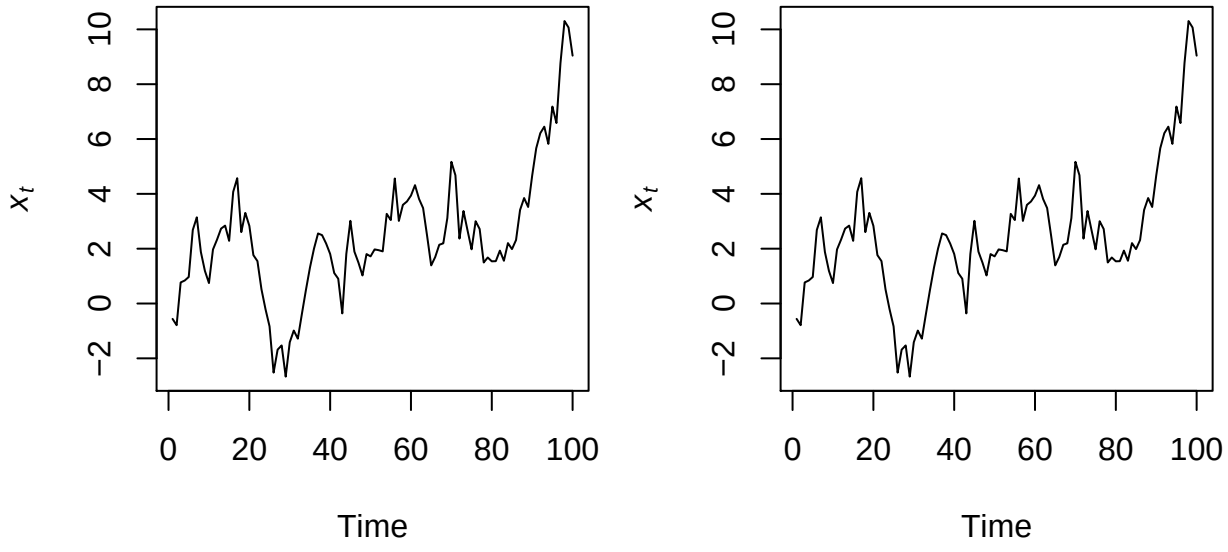


Figure 3.16: Time series of the same random walk model formulated as Equation (3.19) and simulated via a for loop (left), and as Equation (3.22) and simulated via `cumsum()` (right).

Indeed, both methods of generating a RW time series appear to be equivalent.

3.8 Autoregressive (AR) models

Autoregressive models of order p, abbreviated AR(p), are commonly used in time series analyses. In particular, AR(1) models (and their multivariate extensions) see considerable use in ecology as we will see later in the course. Recall from lecture that an AR(p) model is written as

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \cdots + \phi_p x_{t-p} + w_t, \tag{3.23}$$

where $\{w_t\}$ is a white noise sequence with zero mean and some variance σ^2. For our purposes we usually assume that $w_t \sim \mathrm{N}(0, q)$. Note that the random walk in Equation (3.19) is a special case of an AR(1) model where $\phi_1 = 1$ and $\phi_k = 0$ for $k \geq 2$.

3.8.1 Simulating an AR(p) process

Although we could simulate an AR(p) process in R using a for loop just as we did for a random walk, it's much easier with the function `arima.sim()`, which works for all forms and subsets of ARIMA models. To do so, remember that the AR in ARIMA stands for "autoregressive", the I for "integrated", and the MA for "moving-average"; we specify the order of ARIMA models as p, d, q. So, for example, we would specify an AR(2) model as ARIMA(2,0,0), or an MA(1) model as ARIMA(0,0,1). If we had an ARMA(3,1) model that we applied to data that had been twice-differenced, then we would have an ARIMA(3,2,1) model.

`arima.sim()` will accept many arguments, but we are interested primarily in two of them: `n` and `model` (type `?arima.sim` to learn more). The former simply indicates the length of desired time series, but the latter is more complex. Specifically, `model` is a list with the following elements:

- `order` a vector of length 3 containing the ARIMA(p, d, q) order
- `ar` a vector of length p containing the AR(p) coefficients
- `ma` a vector of length q containing the MA(q) coefficients
- `sd` a scalar indicating the std dev of the Gaussian errors

Note that you can omit the `ma` element entirely if you have an AR(p) model, or omit the `ar` element if you have an MA(q) model. If you omit the `sd` element, `arima.sim()` will assume you want normally distributed errors with SD = 1. Also note that you can pass `arima.sim()` your own time series of random errors or the name of a function that will generate the errors (e.g., you could use `rpois()` if you wanted a model with Poisson errors). Type `?arima.sim` for more details.

Let's begin by simulating some AR(1) models and comparing their behavior. First, let's choose models with contrasting AR coefficients. Recall that in order for an AR(1) model to be stationary, $\phi < |1|$, so we'll try 0.1 and 0.9. We'll again set the random number seed so we will get the same answers.

```
set.seed(456)
## list description for AR(1) model with small coef
AR.sm <- list(order = c(1, 0, 0), ar = 0.1, sd = 0.1)
## list description for AR(1) model with large coef
AR.lg <- list(order = c(1, 0, 0), ar = 0.9, sd = 0.1)
## simulate AR(1)
AR1.sm <- arima.sim(n = 50, model = AR.sm)
AR1.lg <- arima.sim(n = 50, model = AR.lg)
```

Now let's plot the 2 simulated series.

```
## setup plot region
par(mfrow = c(1, 2))
## get y-limits for common plots
ylm <- c(min(AR1.sm, AR1.lg), max(AR1.sm, AR1.lg))
## plot the ts
plot.ts(AR1.sm, ylim = ylm, ylab = expression(italic(x)[italic(t)]),
    main = expression(paste(phi, " = 0.1")))
plot.ts(AR1.lg, ylim = ylm, ylab = expression(italic(x)[italic(t)]),
    main = expression(paste(phi, " = 0.9")))
```

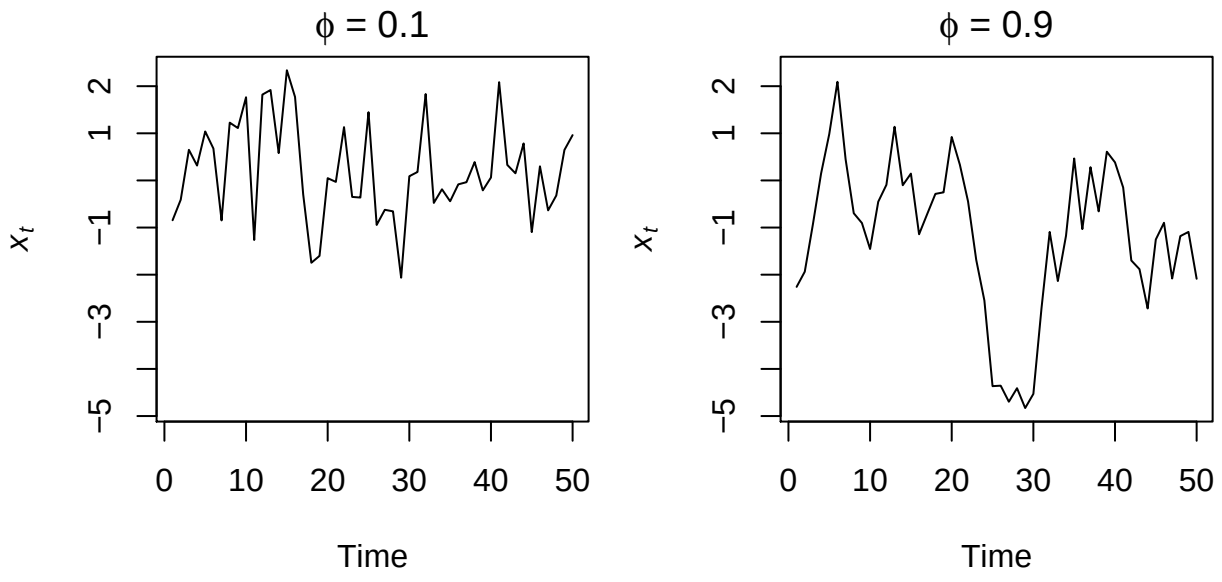


Figure 3.17: Time series of simulated AR(1) processes with $\phi = 0.1$ (left) and $\phi = 0.9$ (right).

What do you notice about the two plots in Figure 3.17? It looks like the time series with the smaller AR coefficient is more "choppy" and seems to stay closer to 0 whereas the time series with the larger AR coefficient appears to wander around more. Remember that as the coefficient in an AR(1) model goes to 0, the model approaches a WN sequence, which is stationary in both the mean and variance. As the coefficient goes to 1, however, the model

approaches a random walk, which is not stationary in either the mean or variance.

Next, let's generate two AR(1) models that have the same magnitude coeficient, but opposite signs, and compare their behavior.

```
set.seed(123)
## list description for AR(1) model with small coef
AR.pos <- list(order = c(1, 0, 0), ar = 0.5, sd = 0.1)
## list description for AR(1) model with large coef
AR.neg <- list(order = c(1, 0, 0), ar = -0.5, sd = 0.1)
## simulate AR(1)
AR1.pos <- arima.sim(n = 50, model = AR.pos)
AR1.neg <- arima.sim(n = 50, model = AR.neg)
```

OK, let's plot the 2 simulated series.

```
## setup plot region
par(mfrow = c(1, 2))
## get y-limits for common plots
ylm <- c(min(AR1.pos, AR1.neg), max(AR1.pos, AR1.neg))
## plot the ts
plot.ts(AR1.pos, ylim = ylm, ylab = expression(italic(x)[italic(t)]),
    main = expression(paste(phi[1], " = 0.5")))
plot.ts(AR1.neg, ylab = expression(italic(x)[italic(t)]), main = expression(paste(phi[1
    " = -0.5")))
```

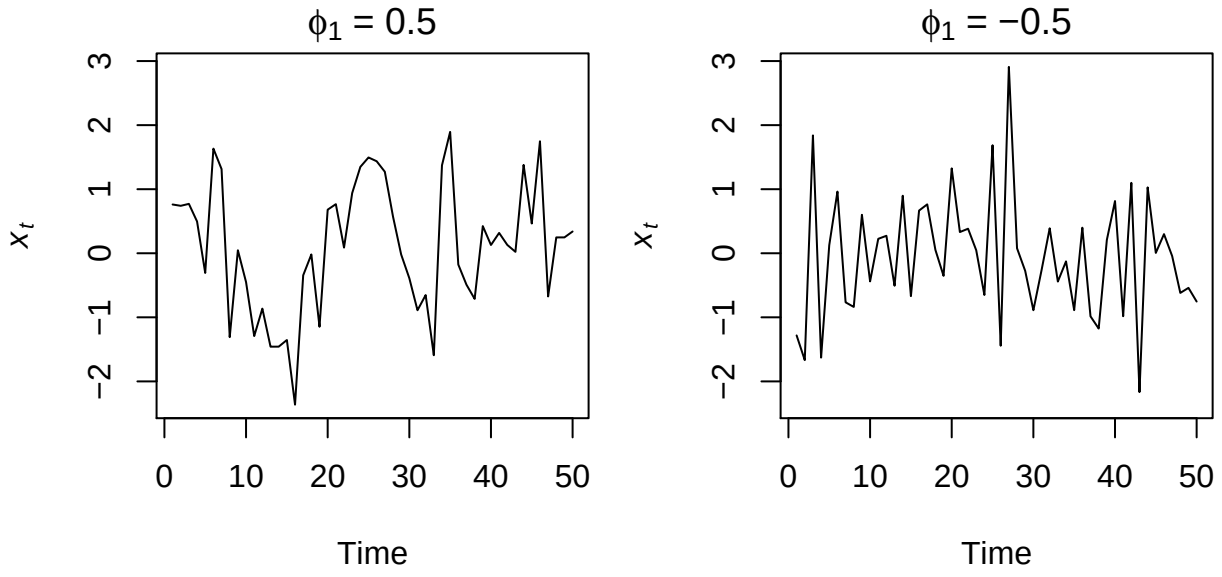


Figure 3.18: Time series of simulated AR(1) processes with $\phi_1 = 0.5$ (left) and $\phi_1 = -0.5$ (right).

Now it appears like both time series vary around the mean by about the same amount, but the model with the negative coefficient produces a much more "sawtooth" time series. It

turns out that any AR(1) model with $-1 < \phi < 0$ will exhibit the 2-point oscillation you see here.

We can simulate higher order AR(p) models in the same manner, but care must be exercised when choosing a set of coefficients that result in a stationary model or else `arima.sim()` will fail and report an error. For example, an AR(2) model with both coefficients equal to 0.5 is not stationary, and therefore this function call will not work:

```
arima.sim(n = 100, model = list(order(2, 0, 0), ar = c(0.5, 0.5)))
```

If you try, R will respond that the "`'ar' part of model is not stationary`".

3.8.2 Correlation structure of AR(p) processes

Let's review what we learned in lecture about the general behavior of the ACF and PACF for AR(p) models. To do so, we'll simulate four stationary AR(p) models of increasing order p and then examine their ACF's and PACF's. Let's use a really big n so as to make them "pure", which will provide a much better estimate of the correlation structure.

```
set.seed(123)
## the 4 AR coefficients
ARp <- c(0.7, 0.2, -0.1, -0.3)
## empty list for storing models
AR.mods <- list()
## loop over orders of p
for (p in 1:4) {
    ## assume SD=1, so not specified
    AR.mods[[p]] <- arima.sim(n = 10000, list(ar = ARp[1:p]))
}
```

Now that we have our four AR(p) models, lets look at plots of the time series, ACF's, and PACF's.

```
## set up plot region
par(mfrow = c(4, 3))
## loop over orders of p
for (p in 1:4) {
    plot.ts(AR.mods[[p]][1:50], ylab = paste("AR(", p, ")", sep = ""))
    acf(AR.mods[[p]], lag.max = 12)
    pacf(AR.mods[[p]], lag.max = 12, ylab = "PACF")
}
```

As we saw in lecture and is evident from our examples shown in Figure 3.19, the ACF for an AR(p) process tails off toward zero very slowly, but the PACF goes to zero for lags $> p$. This is an important diagnostic tool when trying to identify the order of p in ARMA(p, q) models.

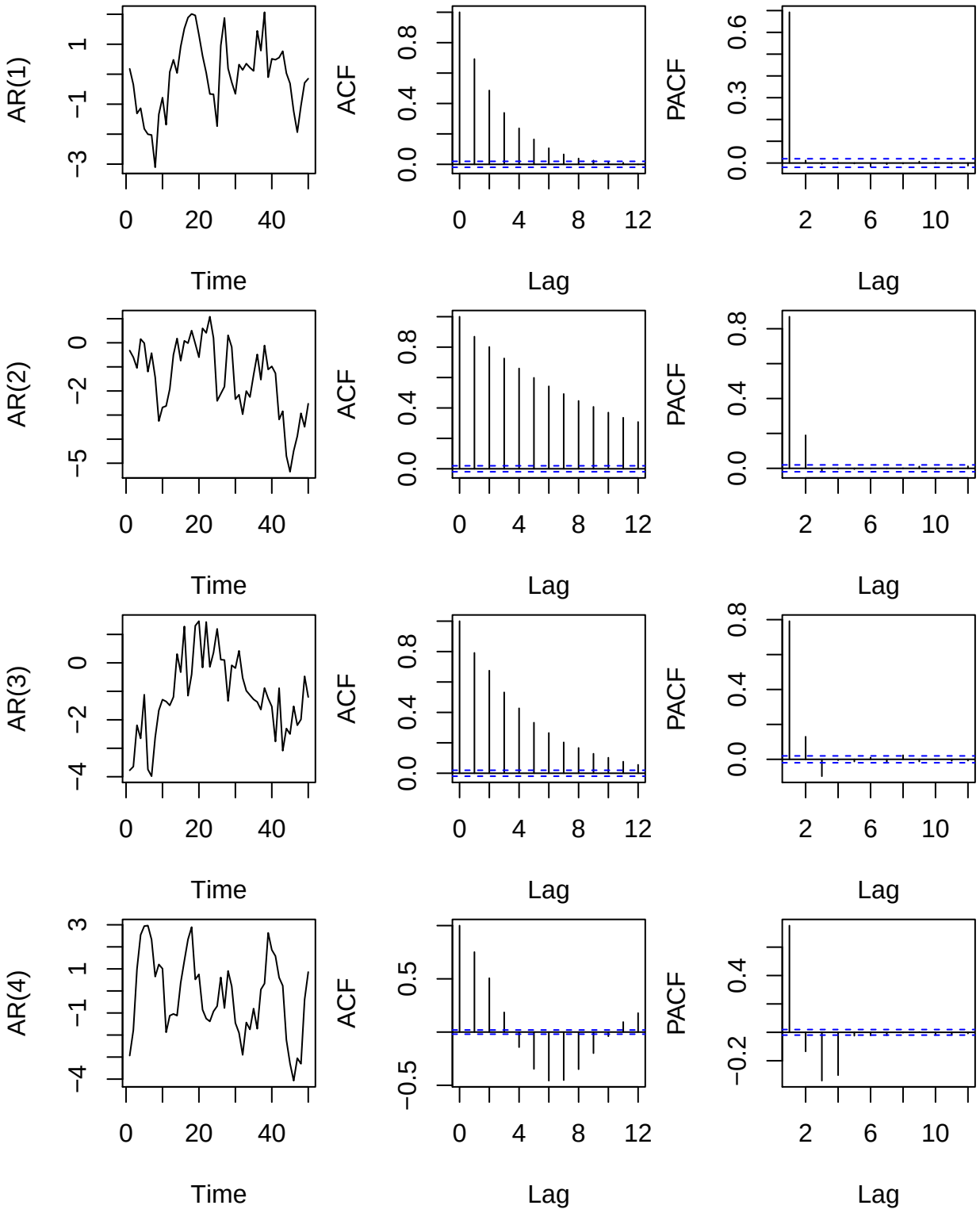


Figure 3.19: Time series of simulated AR(p) processes (left column) of increasing orders from 1-4 (rows) with their associated ACF's (center column) and PACF's (right column). Note that only the first 50 values of x_t are plotted.

3.9 Moving-average (MA) models

A moving-averge process of order q, or MA(q), is a weighted sum of the current random error plus the q most recent errors, and can be written as

$$x_t = w_t + \theta_1 w_{t-1} + \theta_2 w_{t-2} + \cdots + \theta_q w_{t-q}, \tag{3.24}$$

where $\{w_t\}$ is a white noise sequence with zero mean and some variance σ^2; for our purposes we usually assume that $w_t \sim \mathrm{N}(0, q)$. Of particular note is that because MA processes are finite sums of stationary errors, they themselves are stationary.

Of interest to us are so-called "invertible" MA processes that can be expressed as an infinite AR process with no error term. The term invertible comes from the inversion of the backshift operator ($\mathbf{B}$) that we discussed in class (i.e., $\mathbf{B}x_t = x_{t-1}$). So, for example, an MA(1) process with $\theta < |1|$ is invertible because it can be written using the backshift operator as

$$\begin{aligned}
x_t &= w_t - \theta w_{t-1} \\
x_t &= w_t - \theta \mathbf{B} w_t \\
x_t &= (1 - \theta \mathbf{B}) w_t, \\
&\Downarrow \\
w_t &= \frac{1}{(1 - \theta \mathbf{B})} x_t \\
w_t &= (1 + \theta \mathbf{B} + \theta^2 \mathbf{B}^2 + \theta^3 \mathbf{B}^3 + \dots) x_t \\
w_t &= x_t + \theta x_{t-1} + \theta^2 x_{t-2} + \theta^3 x_{t-3} + \dots
\end{aligned} \tag{3.25}$$

3.9.1 Simulating an MA(q) process

We can simulate MA(q) processes just as we did for AR(p) processes using `arima.sim()`. Here are 3 different ones with contrasting θ's:

```
set.seed(123)
## list description for MA(1) model with small coef
MA.sm <- list(order = c(0, 0, 1), ma = 0.2, sd = 0.1)
## list description for MA(1) model with large coef
MA.lg <- list(order = c(0, 0, 1), ma = 0.8, sd = 0.1)
## list description for MA(1) model with large coef
MA.neg <- list(order = c(0, 0, 1), ma = -0.5, sd = 0.1)
## simulate MA(1)
MA1.sm <- arima.sim(n = 50, model = MA.sm)
MA1.lg <- arima.sim(n = 50, model = MA.lg)
MA1.neg <- arima.sim(n = 50, model = MA.neg)
```

with their associated plots.

```
## setup plot region
par(mfrow = c(1, 3))
## plot the ts
plot.ts(MA1.sm, ylab = expression(italic(x)[italic(t)]), main = expression(paste(theta,
    " = 0.2")))
plot.ts(MA1.lg, ylab = expression(italic(x)[italic(t)]), main = expression(paste(theta,
    " = 0.8")))
plot.ts(MA1.neg, ylab = expression(italic(x)[italic(t)]), main = expression(paste(theta
    " = -0.5")))
```

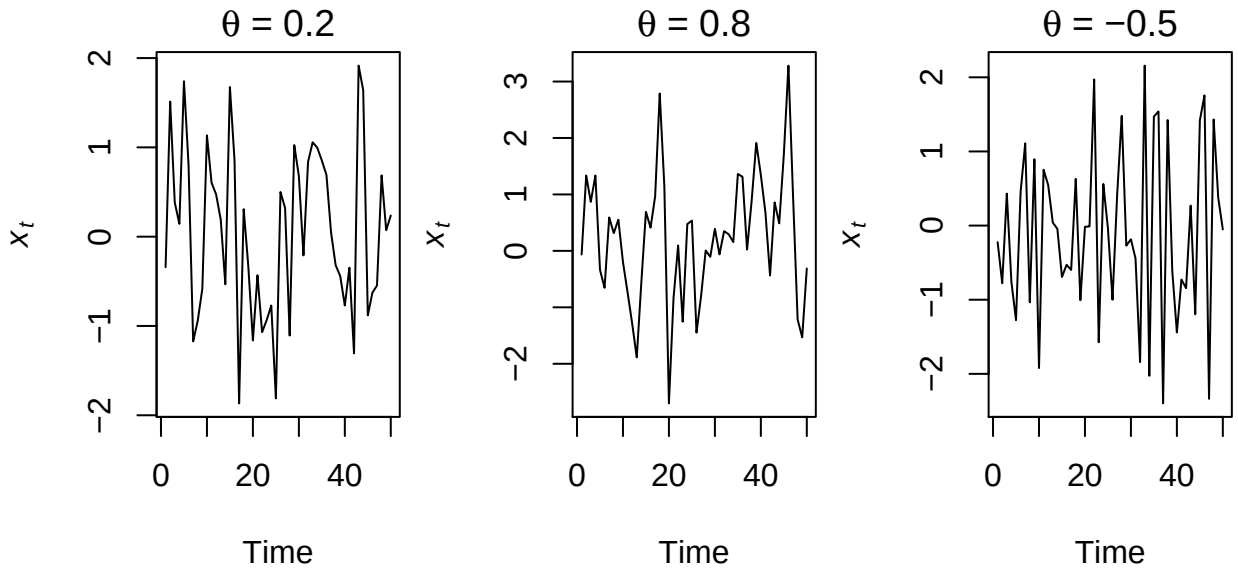


Figure 3.20: Time series of simulated MA(1) processes with $\theta = 0.2$ (left), $\theta = 0.8$ (middle), and $\theta = -0.5$ (right).

In contrast to AR(1) processes, MA(1) models do not exhibit radically different behavior with changing θ. This should not be too surprising given that they are simply linear combinations of white noise.

3.9.2 Correlation structure of MA(q) processes

We saw in lecture and above how the ACF and PACF have distinctive features for AR(p) models, and they do for MA(q) models as well. Here are examples of four MA(q) processes. As before, we'll use a really big n so as to make them "pure", which will provide a much better estimate of the correlation structure.

```
set.seed(123)
## the 4 MA coefficients
MAq <- c(0.7, 0.2, -0.1, -0.3)
## empty list for storing models
```

```
MA.mods <- list()
## loop over orders of q
for (q in 1:4) {
    ## assume SD=1, so not specified
    MA.mods[[q]] <- arima.sim(n = 1000, list(ma = MAq[1:q]))
}
```

Now that we have our four MA(q) models, lets look at plots of the time series, ACF's, and PACF's.

```
## set up plot region
par(mfrow = c(4, 3))
## loop over orders of q
for (q in 1:4) {
    plot.ts(MA.mods[[q]][1:50], ylab = paste("MA(", q, ")", sep = ""))
    acf(MA.mods[[q]], lag.max = 12)
    pacf(MA.mods[[q]], lag.max = 12, ylab = "PACF")
}
```

Note very little qualitative difference in the realizations of the four MA(q) processes (Figure 3.21). As we saw in lecture and is evident from our examples here, however, the ACF for an MA(q) process goes to zero for lags $> q$, but the PACF tails off toward zero very slowly. This is an important diagnostic tool when trying to identify the order of q in ARMA(p, q) models.

3.10 Autoregressive moving-average (ARMA) models

ARMA(p, q) models have a rich history in the time series literature, but they are not nearly as common in ecology as plain AR(p) models. As we discussed in lecture, both the ACF and PACF are important tools when trying to identify the appropriate order of p and q. Here we will see how to simulate time series from AR(p), MA(q), and ARMA(p, q) processes, as well as fit time series models to data based on insights gathered from the ACF and PACF.

We can write an ARMA(p, q) as a mixture of AR(p) and MA(q) models, such that

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + \cdots + \phi_p x_{t-p} + w_t + \theta w_{t-1} + \theta_2 w_{t-2} + \cdots + \theta_q x_{t-q}, \tag{3.26}$$

and the w_t are white noise.

3.10.1 Fitting ARMA(p, q) models with `arima()`

We have already seen how to simulate AR(p) and MA(q) models with `arima.sim()`; the same concepts apply to ARMA(p, q) models and therefore we will not do that here. Instead,

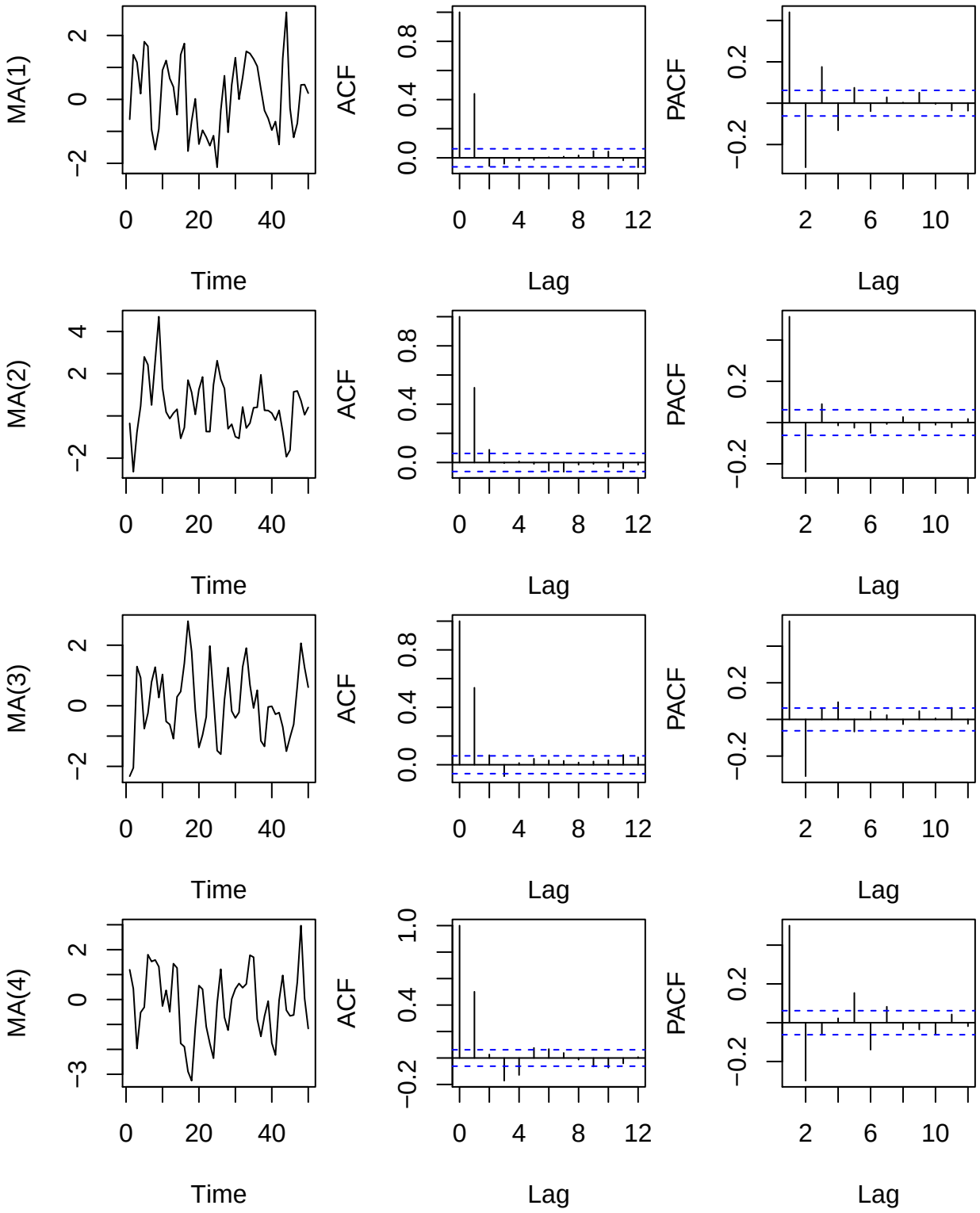


Figure 3.21: Time series of simulated MA(q) processes (left column) of increasing orders from 1-4 (rows) with their associated ACF's (center column) and PACF's (right column). Note that only the first 50 values of x_t are plotted.

we will move on to fitting ARMA(p, q) models when we only have a realization of the process (i.e., data) and do not know the underlying parameters that generated it.

The function `arima()` accepts a number of arguments, but two of them are most important:

- `x` a univariate time series
- `order` a vector of length 3 specifying the order of ARIMA(p,d,q) model

In addition, note that by default `arima()` will estimate an underlying mean of the time series unless $d > 0$. For example, an AR(1) process with mean μ would be written

$$x_t = \mu + \phi(x_{t-1} - \mu) + w_t. \tag{3.27}$$

If you know for a fact that the time series data have a mean of zero (e.g., you already subtracted the mean from them), you should include the argument `include.mean=FALSE`, which is set to `TRUE` by default. Note that ignoring and not estimating a mean in ARMA(p, q) models when one exists will bias the estimates of all other parameters.

Let's see an example of how `arima()` works. First we'll simulate an ARMA(2,2) model and then estimate the parameters to see how well we can recover them. In addition, we'll add in a constant to create a non-zero mean, which `arima()` reports as `intercept` in its output.

```
set.seed(123)
## ARMA(2,2) description for arim.sim()
ARMA22 <- list(order = c(2, 0, 2), ar = c(-0.7, 0.2), ma = c(0.7,
    0.2))
## mean of process
mu <- 5
## simulated process (+ mean)
ARMA.sim <- arima.sim(n = 10000, model = ARMA22) + mu
## estimate parameters
arima(x = ARMA.sim, order = c(2, 0, 2))
```

```
Call:
arima(x = ARMA.sim, order = c(2, 0, 2))

Coefficients:
          ar1     ar2     ma1     ma2  intercept
      -0.7079  0.1924  0.6912  0.2001     4.9975
s.e.   0.0291  0.0284  0.0289  0.0236     0.0125

sigma^2 estimated as 0.9972:  log likelihood = -14175.92,  aic = 28363.84
```

It looks like we were pretty good at estimating the true parameters, but our sample size was admittedly quite large; the estimate of the variance of the process errors is reported as `sigmâ2` below the other coefficients. As an exercise, try decreasing the length of time series

in the `arima.sim()` call above from 10,000 to something like 100 and see what effect it has on the parameter estimates.

3.10.2 Searching over model orders

In an ideal situation, you could examine the ACF and PACF of the time series of interest and immediately decipher what orders of p and q must have generated the data, but that doesn't always work in practice. Instead, we are often left with the task of searching over several possible model forms and seeing which of them provides the most parsimonious fit to the data. There are two easy ways to do this for ARIMA models in R. The first is to write a little script that loops ove the possible dimensions of p and q. Let's try that for the process we simulated above and search over orders of p and q from 0-3 (it will take a few moments to run and will likely report an error about a "`possible convergence problem`", which you can ignore).

```
## empty list to store model fits
ARMA.res <- list()
## set counter
cc <- 1
## loop over AR
for (p in 0:3) {
    ## loop over MA
    for (q in 0:3) {
        ARMA.res[[cc]] <- arima(x = ARMA.sim, order = c(p, 0,
            q))
        cc <- cc + 1
    }
}
```

```
Warning in arima(x = ARMA.sim, order = c(p, 0, q)): possible convergence
problem: optim gave code = 1
```

```
## get AIC values for model evaluation
ARMA.AIC <- sapply(ARMA.res, function(x) x$aic)
## model with lowest AIC is the best
ARMA.res[[which(ARMA.AIC == min(ARMA.AIC))]]
```

```

Call:
arima(x = ARMA.sim, order = c(p, 0, q))

Coefficients:
          ar1     ar2     ma1     ma2  intercept
      -0.7079  0.1924  0.6912  0.2001     4.9975
s.e.   0.0291  0.0284  0.0289  0.0236     0.0125
```

```
sigma^2 estimated as 0.9972:  log likelihood = -14175.92,  aic = 28363.84
```

It looks like our search worked, so let's look at the other method for fitting ARIMA models. The `auto.arima()` function in the **forecast** package will conduct an automatic search over all possible orders of ARIMA models that you specify. For details, type `?auto.arima` after loading the package. Let's repeat our search using the same criteria.

```
## find best ARMA(p,q) model
auto.arima(ARMA.sim, start.p = 0, max.p = 3, start.q = 0, max.q = 3)
```

```
Series: ARMA.sim
ARIMA(2,0,2) with non-zero mean

Coefficients:
          ar1     ar2     ma1     ma2    mean
      -0.7079  0.1924  0.6912  0.2001  4.9975
s.e.   0.0291  0.0284  0.0289  0.0236  0.0125

sigma^2 estimated as 0.9977:  log likelihood=-14175.92
AIC=28363.84   AICc=28363.84   BIC=28407.1
```

We get the same results with an increase in speed and less coding, which is nice. If you want to see the form for each of the models checked by `auto.arima()` and their associated AIC values, include the argument `trace=1`.

3.11 Problems

We have seen how to do a variety of introductory time series analyses with R. Now it is your turn to apply the information you learned here and in lecture to complete some analyses. You have been asked by a colleague to help analyze some time series data she collected as part of an experiment on the effects of light and nutrients on the population dynamics of phytoplankton. Specifically, after controlling for differences in light and temperature, she wants to know if the natural log of population density can be modeled with some form of ARMA(p, q) model.

The data are expressed as the number of cells per milliliter recorded every hour for one week beginning at 8:00 AM on December 1, 2014; you can download the data here: hourly_phyto.RData.

Use the information above to do the following:

1. Convert `pDat`, which is a **data.frame** object, into a **ts** object. This bit of code might be useful to get you started:

```
## what day of 2014 is Dec 1st?
dBegin <- as.Date("2014-12-01")
dayOfYear <- (dBegin - as.Date("2014-01-01") + 1)
```

2. Plot the time series of phytoplankton density and provide a brief description of any notable features.

3. Although you do not have the actual measurements for the specific temperature and light regimes used in the experiment, you have been informed that they follow a regular light/dark period with accompanying warm/cool temperatures. Thus, estimating a fixed seasonal effect is justifiable. Also, the instrumentation is precise enough to preclude any systematic change in measurements over time (i.e., you can assume $m_t = 0$ for all t). Obtain the time series of the estimated log-density of phytoplankton absent any hourly effects caused by variation in temperature or light. (Hint: You will need to do some decomposition.)

4. Use diagnostic tools to identify the possible order(s) of ARMA model(s) that most likely describes the log of population density for this particular experiment. Note that at this point you should be focusing your analysis on the results obtained in Question 3.

5. Use some form of search to identify what form of ARMA(p, q) model best describes the log of population density for this particular experiment. Use what you learned in Question 4 to inform possible orders of p and q. (Hint: if you use `auto.arima()`, include the additional argument `seasonal=FALSE`)

6. Write out the best model in the form of Equation (3.26) using the underscore notation to refer to subscripts (e.g., write `x_t` for x_t). You can round any parameters/coefficients to the nearest hundreth. (*Hint*: if the mean of the time series is not zero, refer to Eqn 1.27 in the lab handout).

Chapter 4

Univariate state-space models

This lab show you how to fit some basic univariate state-space models using MARSS(). This will also introduce you to the idea of writing AR models in state-space form.

A script with all the R code in the chapter can be downloaded here.

Data and packages

All the data used in the chapter are in the **MARSS** package. The other required packages are **stats** (normally loaded by default when starting R), **datasets** and **forecast**. To run the JAGS code example, you will also need JAGS installed and the **R2jags** and **coda** R packages. Install the packages, if needed, and load:

```
library(stats)
library(MARSS)
library(forecast)
library(datasets)
library(R2jags)
library(coda)
```

4.1 Fitting a state-space model with MARSS

The **MARSS** package fits multivariate auto-regressive models of this form:

$$\begin{aligned} \mathbf{x}_t &= \mathbf{B}\mathbf{x}_{t-1} + \mathbf{u} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \text{N}(0, \mathbf{Q}) \\ \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \text{N}(0, \mathbf{R}) \\ \mathbf{x}_0 &= \boldsymbol{\mu} \end{aligned} \tag{4.1}$$

To fit your time-series model with the **MARSS** package, you need to put your model into the form above. The $\mathbf{B}$, $\mathbf{Z}$, $\mathbf{u}$, $\mathbf{a}$, $\mathbf{Q}$, $\mathbf{R}$ and $\boldsymbol{\mu}$ are parameters that are (potentially) estimated. The $\mathbf{y}$ are your data. The $\mathbf{x}$ are the hidden state(s). Everything in bold is a matrix; if it is a small bolded letter, it is a matrix with 1 column.

Important: this state-space model, $\mathbf{y}$ is always the data and $\mathbf{x}$ is a hidden random walk estimated from the data.

A basic `MARSS()` call looks like `fit=MARSS(y, model=list(...))`. The argument `model` tells the function what form the parameters take. The list has the elements with the names: `B`, `U`, `Q`, etc. The names correspond to the parameters with the same names in Equation (4.1) except that $\boldsymbol{\mu}$ is called `x0`. `tinitx` indicates whether the initial $\mathbf{x}$ is specified at $t = 0$ so $\mathbf{x}_0$ or $t = 1$ so $\mathbf{x}_1$.

Here's an example. Let's say we want to fit a univariate AR(1) model observed with error. Here is that model:

$$\begin{aligned} x_t &= bx_{t-1} + w_t \text{ where } \mathbf{w}_t \sim \text{N}(0, q) \\ y_t &= x_t + v_t \text{ where } v_t \sim \text{N}(0, r) \\ x_0 &= \mu \end{aligned} \tag{4.2}$$

To fit this with `MARSS()`, we need to write Equation (4.2) as Equation (4.1). Equation (4.1) is in MATRIX form. In the model list, the parameters must be written EXACTLY like they would be written for Equation (4.1). For example, `1` is the number 1 in R. It is not a matrix:

```
class(1)
```

```
[1] "numeric"
```

If you need a 1 (or 0) in your model, you need to pass in the parameter as a 1×1 matrix: `matrix(1)`.

With that mind, our model list for Equation (4.2) is:

```
mod.list = list(B = matrix(1), U = matrix(0), Q = matrix("q"),
    Z = matrix(1), A = matrix(0), R = matrix("r"), x0 = matrix("mu"),
    tinitx = 0)
```

We can simulate some AR(1) plus error data like so

```
q = 0.1
r = 0.1
n = 100
y = cumsum(rnorm(n, 0, sqrt(q))) + rnorm(n, 0, sqrt(r))
```

And then fit with `MARSS()` using `mod.list` above:

```
fit = MARSS(y, model = mod.list)
```

```
Success! abstol and log-log tests passed at 26 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 26 iterations.
Log-likelihood: -57.26285
AIC: 120.5257   AICc: 120.7757

      Estimate
R.r     0.0534
Q.q     0.0933
x0.mu   0.4067

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

If we wanted to fix $q = 0.1$, then $\mathbf{Q} = [0.1]$ (a 1×1 matrix with 0.1). We just change `mod.list$Q` and re-fit:

```
mod.list$Q = matrix(0.1)
fit = MARSS(y, model = mod.list)
```

4.2 Examples using the Nile river data

We will use the data from the Nile River (Figure 4.1). We will fit different flow models to the data and compare the models with AIC.

```
library(datasets)
dat = as.vector(Nile)
```

4.2.1 Flat level model

We will start by modeling these data as a simple average river flow with variability around some level μ.

$$y_t = \mu + v_t \text{ where } v_t \sim \text{N}(0, r) \tag{4.3}$$

where y_t is the river flow volume at year t.

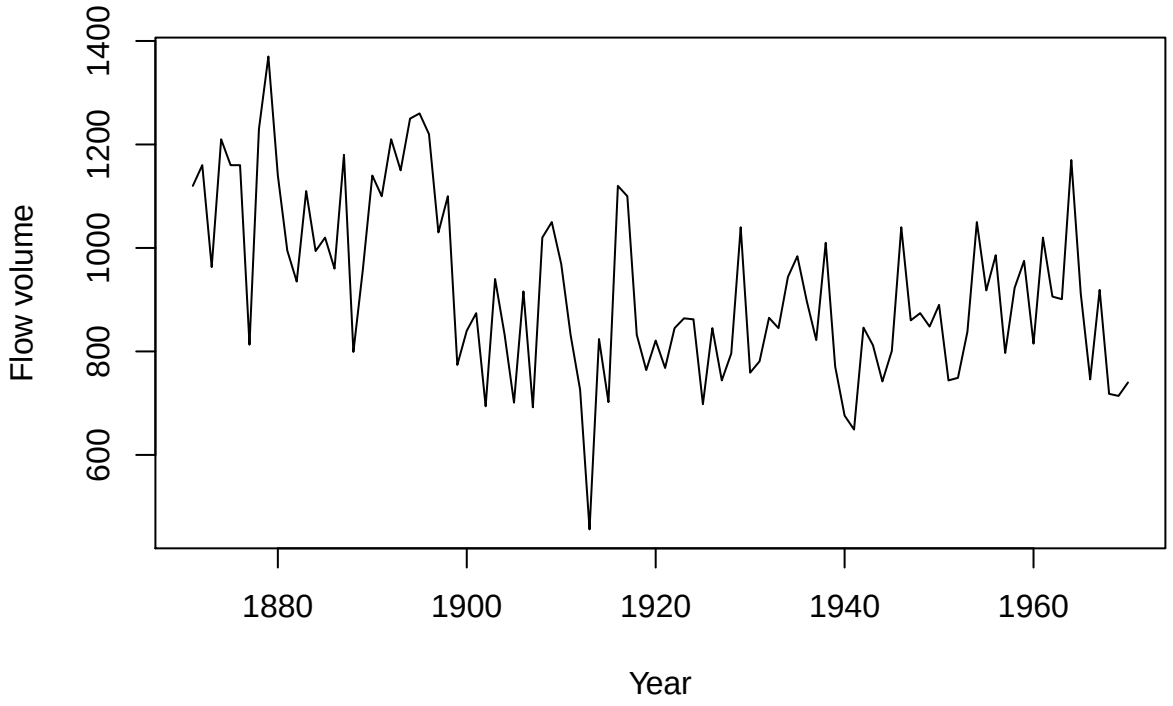


Figure 4.1: The Nile River flow volume 1871 to 1970 (`Nile` dataset in R).

We can write this model as a univariate state-space model as follows. We use x_t to model the average flow level. y_t is just an observation of this flat x_t. Work through x_1, x_2, ... starting from x_0 to convince yourself that x_t will always equal μ.

$$\begin{aligned} x_t &= 1 \times x_{t-1} + 0 + w_t \text{ where } w_t \sim \text{N}(0, 0) \\ y_t &= 1 \times x_t + 0 + v_t \text{ where } v_t \sim \text{N}(0, r) \\ x_0 &= \mu \end{aligned} \tag{4.4}$$

The model is specified as a list as follows:

```
mod.nile.0 = list(B = matrix(1), U = matrix(0), Q = matrix(0),
    Z = matrix(1), A = matrix(0), R = matrix("r"), x0 = matrix("mu"),
    tinitx = 0)
```

We then fit the model:

```
kem.0 = MARSS(dat, model = mod.nile.0)
```

Output not shown, but here are the estimates and AICc.

```
c(coef(kem.0, type = "vector"), LL = kem.0$logLik, AICc = kem.0$AICc)
```

```
       R.r      x0.mu         LL       AICc
28351.5675   919.3500  -654.5157  1313.1552
```

4.2.2 Linear trend in flow model

Figure 4.2 shows the fit for the flat average river flow model. Looking at the data, we might expect that a declining average river flow would be better. In MARSS form, that model would be:

$$\begin{aligned} x_t &= 1 \times x_{t-1} + u + w_t \text{ where } w_t \sim \text{N}(0, 0) \\ y_t &= 1 \times x_t + 0 + v_t \text{ where } v_t \sim \text{N}(0, r) \\ x_0 &= \mu \end{aligned} \tag{4.5}$$

where u is now the average per-year decline in river flow volume. The model is specified as follows:

```
mod.nile.1 = list(B = matrix(1), U = matrix("u"), Q = matrix(0),
    Z = matrix(1), A = matrix(0), R = matrix("r"), x0 = matrix("mu"),
    tinitx = 0)
```

We then fit the model:

```
kem.1 = MARSS(dat, model = mod.nile.1)
```

Here are the estimates, log-likelihood and AICc:

```
c(coef(kem.1, type = "vector"), LL = kem.1$logLik, AICc = kem.1$AICc)
```

```
         R.r          U.u         x0.mu           LL         AICc
22213.595453    -2.692106  1054.935067  -642.315910  1290.881821
```

Figure 4.2 shows the fits for the two models with deterministic models (flat and declining) for mean river flow along with their AICc values (smaller AICc is better). The AICc for the model with a declining river flow is lower by over 20 (which is a lot).

4.2.3 Stochastic level model

Looking at the flow levels, we might suspect that a model that allows the average flow to change would model the data better and we might suspect that there have been sudden, and anomalous, changes in the river flow level. 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 is year t is a function of average river flow in year $t-1$.

$$\begin{aligned} x_t &= x_{t-1} + w_t \text{ where } w_t \sim \text{N}(0, q) \\ y_t &= x_t + v_t \text{ where } v_t \sim \text{N}(0, r) \\ x_0 &= \mu \end{aligned} \tag{4.6}$$

As before, y_t is the river flow volume at year t. x_t is the mean level. The model is specified as:

```
mod.nile.2 = list(B = matrix(1), U = matrix(0), Q = matrix("q"),
    Z = matrix(1), A = matrix(0), R = matrix("r"), x0 = matrix("mu"),
    tinitx = 0)
```

We could also use the text shortcuts to specify the model. Because $\mathbf{R}$ and $\mathbf{Q}$ are 1×1 matrices, "unconstrained", "diagonal and unequal", "diagonal and equal" and "equalvarcov" will all lead to a 1×1 matrix with one estimated element. For $\mathbf{a}$ and $\mathbf{u}$, the following shortcut could be used:

```
A = U = "zero"
```

Because $\mathbf{x}_0$ is 1×1, it could be specified as "unequal", "equal" or "unconstrained".

```
kem.2 = MARSS(dat, model = mod.nile.2)
```

Here are the estimates, log-likelihood and AICc:

```
c(coef(kem.2, type = "vector"), LL = kem.2$logLik, AICc = kem.2$AICc)
```

```
       R.r        Q.q      x0.mu         LL       AICc
15065.6121  1425.0030  1111.6338  -637.7631  1281.7762
```

4.2.4 Stochastic level model with drift

We can add a drift to term to our random walk; the u in the process model (x) is the drift term. This causes the random walk to tend to trend up or down.

$$\begin{aligned} x_t &= x_{t-1} + u + w_t \text{ where } w_t \sim \text{N}(0, q) \\ y_t &= x_t + v_t \text{ where } v_t \sim \text{N}(0, r) \\ x_0 &= \mu \end{aligned} \tag{4.7}$$

The model is then specified by changing `U` to indicate that a u is estimated:

```
mod.nile.3 = list(B = matrix(1), U = matrix("u"), Q = matrix("q"),
    Z = matrix(1), A = matrix(0), R = matrix("r"), x0 = matrix("mu"),
    tinitx = 0)
```

```
kem.3 = MARSS(dat, model = mod.nile.3)
```

Here are the estimates, log-likelihood and AICc:

```
c(coef(kem.3, type = "vector"), LL = kem.3$logLik, AICc = kem.3$AICc)
```

```
         R.r          U.u          Q.q        x0.mu           LL
15585.278194    -3.248793  1088.987455  1124.044484  -637.302692
        AICc
 1283.026436
```

Figure 4.2 shows all the models along with their AICc values.

4.3 The StructTS function

The `StructTS` function in R will also fit the stochastic level model:

```
fit.sts = StructTS(dat, type = "level")
fit.sts
```

```

Call:
StructTS(x = dat, type = "level")

Variances:
  level  epsilon
   1469    15099
```

The estimates from `StructTS()` will be different (though similar) from `MARSS()` because `StructTS()` uses $x_1 = y_1$, that is the hidden state at $t = 1$ is fixed to be the data at $t = 1$.

That is fine if you have a long data set, but would be disastrous for the short data sets typical in fisheries and ecology.

`StructTS()` is much, much faster for long time series. The example in `?StructTS` is pretty much instantaneous with `StructTS()` but takes minutes with the EM algorithm that is the default in `MARSS()`. With the BFGS algorithm, it is much closer to `StructTS()`:

```
trees <- window(treering, start = 0)
fitts = StructTS(trees, type = "level")
fitem = MARSS(as.vector(trees), mod.nile.2)
fitbf = MARSS(as.vector(trees), mod.nile.2, method = "BFGS")
```

Note that `mod.nile.2` specifies a univariate stochastic level model so we can use it just fine with other univariate data sets.

In addition, `fitted(fit.sts)` where `fit.sts` is a fit from `StructTS()` is very different than `fit.marss$states` from `MARSS()`.

```
t = 10
fitted(fit.sts)[t]
```

```
[1] 1162.904
```

is the expected value of y_{t+1} (in this case y_{11} since we set $t = 10$) given the data up to y_t (in this case, up to y_{10}). It is called the one-step ahead prediction.
We are not going to use the one-step ahead predictions unless we are forecasting or doing cross-validation.

Typically, when we analyze fisheries and ecological data, we want to know the estimate of the state, the x_t, given ALL the data. For example, we might need an estimate of the population size in year 1990 given a time series of counts from 1930 to 2015. We don't want to use only the data up to 1989; we want to use all the information. `fit.marss$states` from `MARSS()` is the expected value of x_t given all the data. For the stochastic level model, that is equal to the expected value of y_t given all the data except y_t.

If you needed the one-step predictions from `MARSS()`, you can get them from the Kalman filter output:

```
kf = print(kem.2, what = "kfs")
kf$xtt1[1, t]
```

Passing in `what="kfs"` returns the Kalman filter/smoother output. The expected value of x_t conditioned on y_1 to y_{t-1} is in `kf$xtt1`. The expected value of x_t conditioned on all the data is in `kf$xtT`.

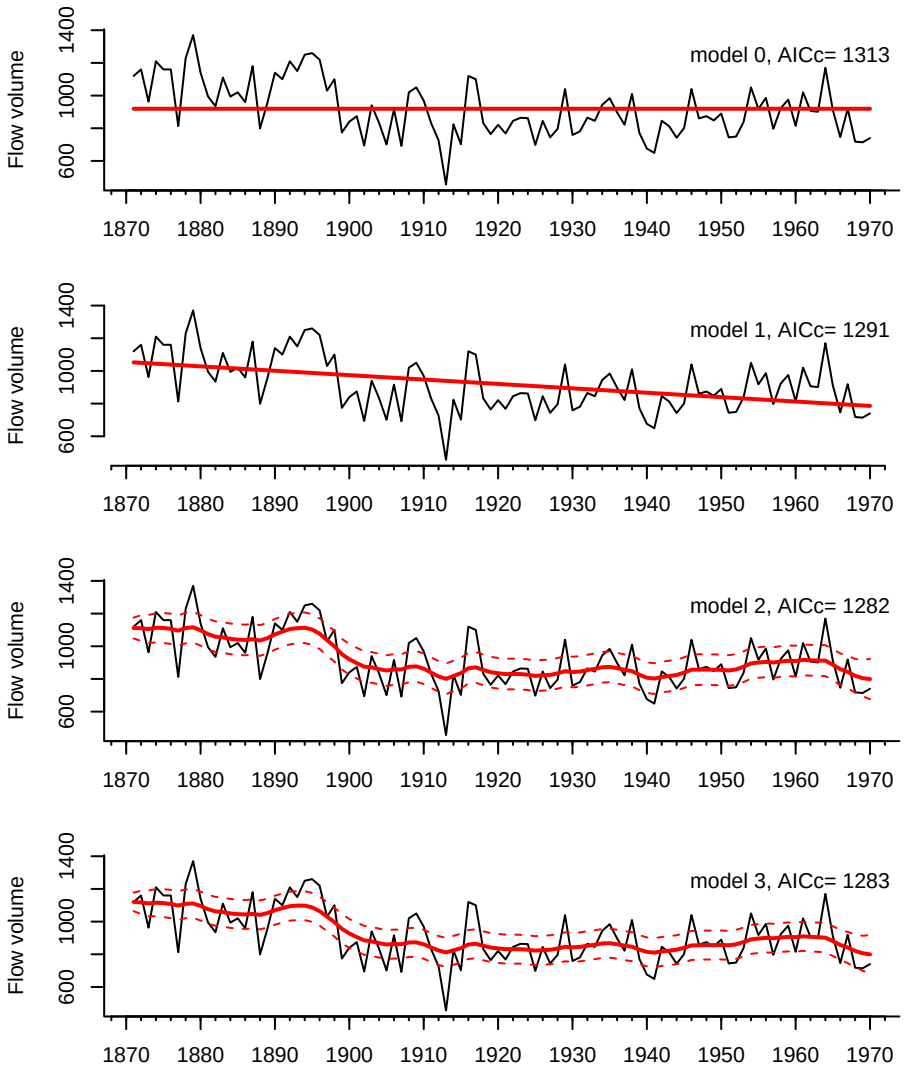


Figure 4.2: The Nile River flow volume with the model estimated flow rates (solid lines). The bottom model is a stochastic level model, meaning there isn't one level line. Rather the level line is a distribution that has a mean and standard deviation. The solid state line in the bottom plots is the mean of the stochastic level and the 2 standard deviations are shown. The other two models are deterministic level models so the state is not stochastic and does not have a standard deviation.

4.4 Comparing models with AIC and model weights

To get the AIC or AICc values for a model fit from a MARSS fit, use `fit$AIC` or `fit$AICc`. The log-likelihood is in `fit$logLik` and the number of estimated parameters in `fit$num.params`. For fits from other functions, try `AIC(fit)` or look at the function documentation.

Let's put the AICc values 3 Nile models together:

```
nile.aic = c(kem.0$AICc, kem.1$AICc, kem.2$AICc, kem.3$AICc)
```

Then we calculate the AICc minus the minus AICc in our model set and compute the model weights. ΔAIC is the AIC values minus the minimum AIC value in your model set.

```
delAIC = nile.aic - min(nile.aic)
relLik = exp(-0.5 * delAIC)
aicweight = relLik/sum(relLik)
```

And this leads to our model weights table:

```
aic.table = data.frame(AICc = nile.aic, delAIC = delAIC, relLik = relLik,
    weight = aicweight)
rownames(aic.table) = c("flat level", "linear trend", "stoc level",
    "stoc level w drift")
```

Here the table is printed using `round()` to limit the number of digits shown.

```
round(aic.table, digits = 3)
```

```
                     AICc delAIC relLik weight
flat level       1313.155 31.379  0.000  0.000
linear trend     1290.882  9.106  0.011  0.007
stoc level       1281.776  0.000  1.000  0.647
stoc level w drift 1283.026  1.250  0.535  0.346
```

One thing to keep in mind when comparing models within a set of models is that the model set needs to include at least one model that can fit the data reasonably well. `Reasonably well' means the model can put a fitted line through the data. Can't all models do that? Definitely, not. For example, the flat-level model cannot put a fitted line through the Nile River data. It is simply impossible. The straight trend model also cannot put a fitted line through the flow data. So if our model set only included flat-level and straight trend, then we might have said that the straight trend model is`best' even though it is just the better of two bad models.

4.5 Basic diagnostics

The first diagnostic that you do with any statistical analysis is check that your residuals correspond to your assumed error structure. We have two types of errors in a univariate state-space model: process errors, the w_t, and observation errors, the v_t.
They should not have a temporal trend. To get the residuals from most types of fits in R, you can use `residuals(fit)`. `MARSS()` calls the v_t, "model residuals", and the w_t "state residuals". We can plot these using the following code (Figure 4.3).

```
par(mfrow = c(1, 2))
resids = residuals(kem.0)
plot(resids$model.residuals[1, ], ylab = "model residual", xlab = "",
    main = "flat level")
abline(h = 0)
plot(resids$state.residuals[1, ], ylab = "state residual", xlab = "",
    main = "flat level")
abline(h = 0)
```

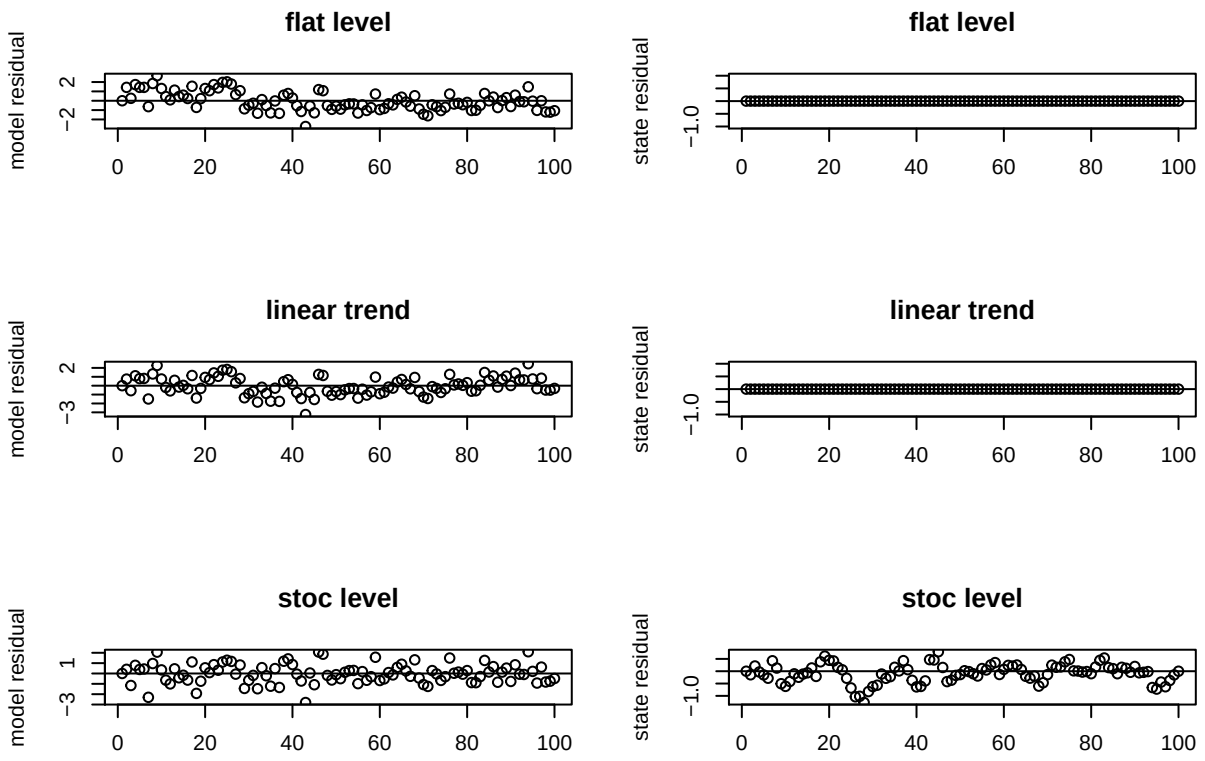


Figure 4.3: The model and state residuals for the first 3 models.

The residuals should also not be autocorrelated in time. We can check the autocorrelation with the function `acf()`. We won't do this for the state residuals for the flat level or linear

trends since for those models $w_t = 0$. The autocorrelation plots are shown in Figure 4.4. The stochastic level model looks the best in that its model residuals (the v_t) are fine but the state model still has problems. Clearly the state is not a simple random walk. This is not surprising. The Aswan Low Dam was completed in 1902 and changed the mean flow. The Aswan High Dam was completed in 1970 and also affected the flow. You can see these perturbations in Figure 4.1.

```
par(mfrow = c(2, 2))
resids = residuals(kem.0)
acf(resids$model.residuals[1, ], main = "flat level v(t)")
resids = residuals(kem.1)
acf(resids$model.residuals[1, ], main = "linear trend v(t)")
resids = residuals(kem.2)
acf(resids$model.residuals[1, ], main = "stoc level v(t)")
acf(resids$state.residuals[1, ], main = "stoc level w(t)")
```

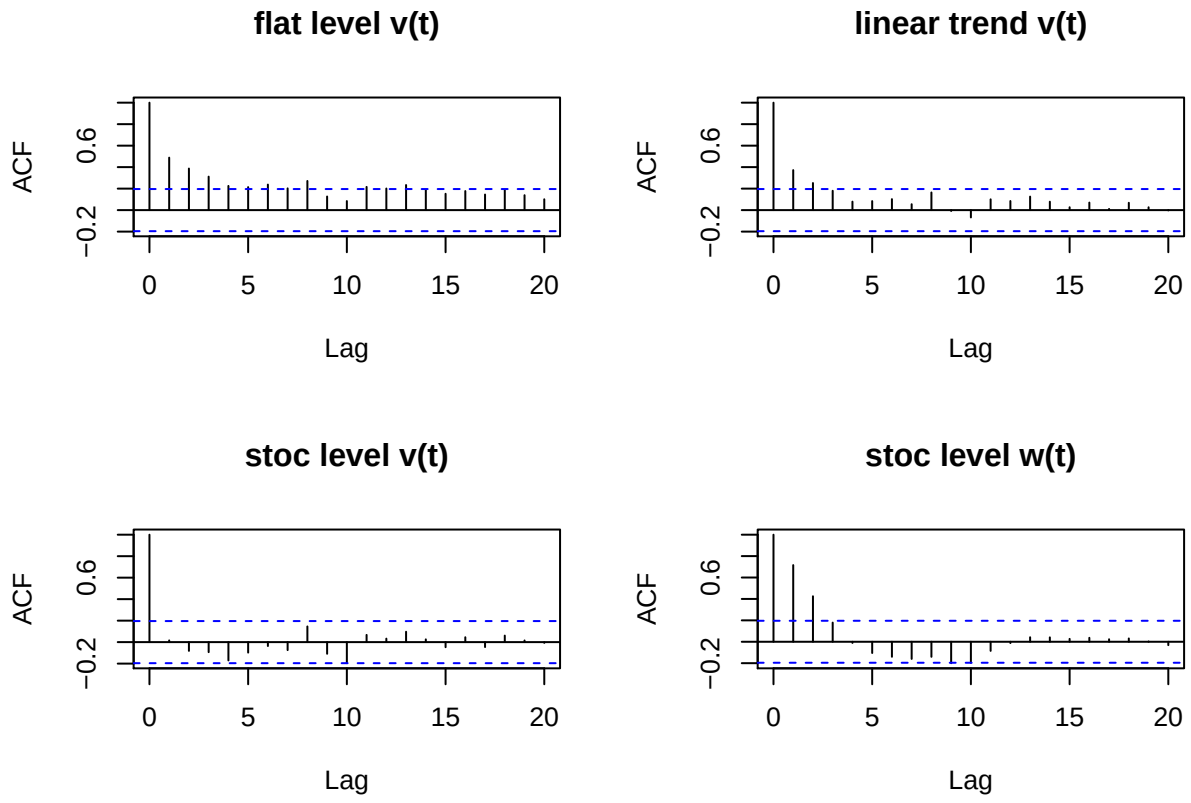


Figure 4.4: The model and state residual acfs for the 3 models.

4.6 Fitting a univariate AR(1) state-space model with JAGS

Here we show how to fit model 3, Equation (4.7), with JAGS. This section requires that you have JAGS installed and the **R2jags** and **coda** R packages loaded.

The first step is to write the model for JAGS to a file (filename in `model.loc`):

```
model.loc = "ss_model.txt"
jagsscript = cat("
   model {
   # priors on parameters
   mu ~ dnorm(Y1, 1/(Y1*100)); # normal mean = 0, sd = 1/sqrt(0.01)
   tau.q ~ dgamma(0.001,0.001); # This is inverse gamma
   sd.q <- 1/sqrt(tau.q); # sd is treated as derived parameter
   tau.r ~ dgamma(0.001,0.001); # This is inverse gamma
   sd.r <- 1/sqrt(tau.r); # sd is treated as derived parameter
   u ~ dnorm(0, 0.01);

   # Because init X is specified at t=0
   X0 <- mu
   X[1] ~ dnorm(X0+u,tau.q);
   Y[1] ~ dnorm(X[1], tau.r);

   for(i in 2:N) {
   predX[i] <- X[i-1]+u;
   X[i] ~ dnorm(predX[i],tau.q); # Process variation
   Y[i] ~ dnorm(X[i], tau.r); # Observation variation
   }
   }
   ",
    file = model.loc)
```

Next we specify the data (and any other input) that the JAGS code needs. In this case, we need to pass in `dat` and the number of time steps since that is used in the for loop. We also specify the parameters that we want to monitor. We need to specify at least one, but we will monitor all of them so we can plot them after fitting. Note, that the hidden state is a parameter in the Bayesian context (but not in the maximum likelihood context).

```
jags.data = list(Y = dat, N = length(dat), Y1 = dat[1])
jags.params = c("sd.q", "sd.r", "X", "mu", "u")
```

Now we can fit the model:

```
mod_ss = jags(jags.data, parameters.to.save = jags.params, model.file = model.loc,
    n.chains = 3, n.burnin = 5000, n.thin = 1, n.iter = 10000,
```

```
    DIC = TRUE)
```

We can then show the posteriors along with the MLEs from MARSS on top (Figure 4.5) using the code below.

```
attach.jags(mod_ss)
par(mfrow = c(2, 2))
hist(mu)
abline(v = coef(kem.3)$x0, col = "red")
hist(u)
abline(v = coef(kem.3)$U, col = "red")
hist(log(sd.q^2))
abline(v = log(coef(kem.3)$Q), col = "red")
hist(log(sd.r^2))
abline(v = log(coef(kem.3)$R), col = "red")
```

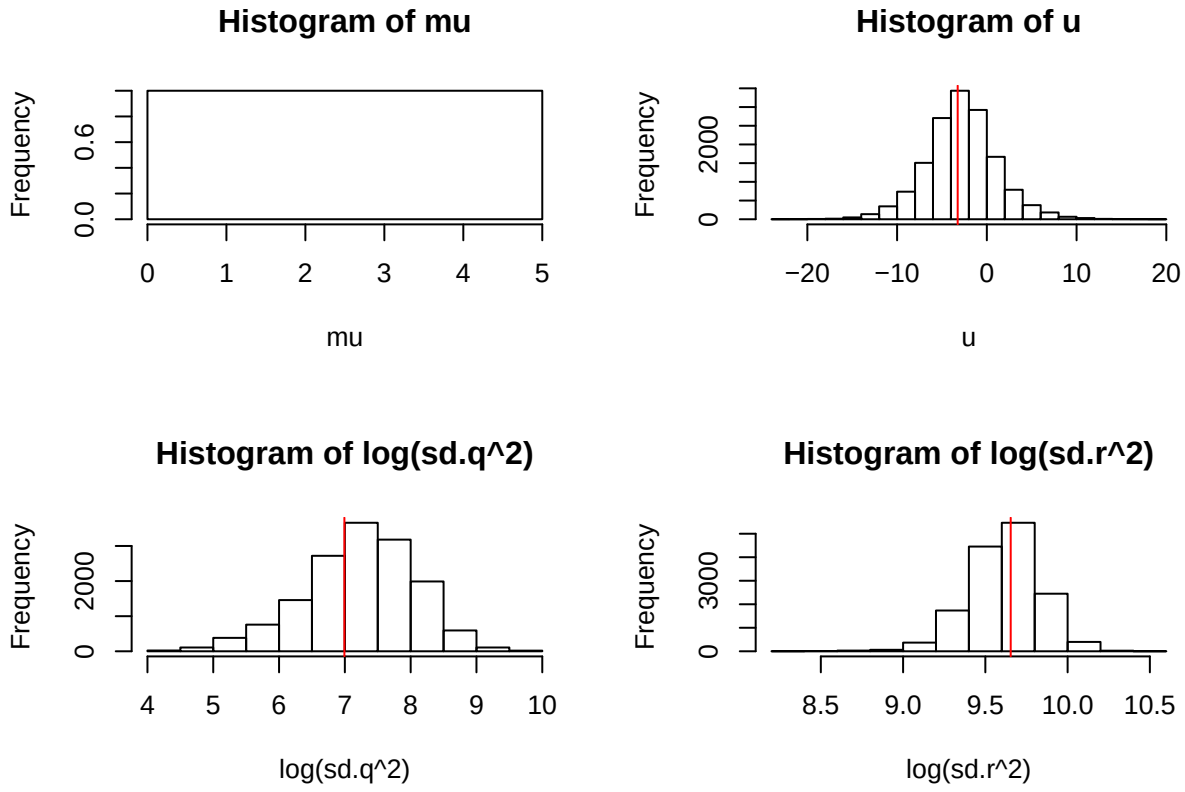


Figure 4.5: The posteriors for model 3 with MLE estimates from `MARSS()` shown in red.

```
detach.jags()
```

To plot the estimated states (Figure 4.6), we write a helper function:

```
plotModelOutput = function(jagsmodel, Y) {
    attach.jags(jagsmodel)
    x = seq(1, length(Y))
    XPred = cbind(apply(X, 2, quantile, 0.025), apply(X, 2, mean),
        apply(X, 2, quantile, 0.975))
    ylims = c(min(c(Y, XPred), na.rm = TRUE), max(c(Y, XPred),
        na.rm = TRUE))
    plot(Y, col = "white", ylim = ylims, xlab = "", ylab = "State predictions")
    polygon(c(x, rev(x)), c(XPred[, 1], rev(XPred[, 3])), col = "grey70",
        border = NA)
    lines(XPred[, 2])
    points(Y)
}
```

```
plotModelOutput(mod_ss, dat)
```

```
The following object is masked _by_ .GlobalEnv:

    mu
```

```
lines(kem.3$states[1, ], col = "red")
lines(1.96 * kem.3$states.se[1, ] + kem.3$states[1, ], col = "red",
    lty = 2)
lines(-1.96 * kem.3$states.se[1, ] + kem.3$states[1, ], col = "red",
    lty = 2)
title("State estimate and data from\nJAGS (black) versus MARSS (red)")
```

4.7 A random walk model of animal movement

A simple random walk model of movement with drift (directional movement) but no correlation is

$$x_{1,t} = x_{1,t-1} + u_1 + w_{1,t}, \quad w_{1,t} \sim \mathrm{N}(0, \sigma_1^2) \tag{4.8}$$
$$x_{2,t} = x_{2,t-1} + u_2 + w_{2,t}, \quad w_{2,t} \sim \mathrm{N}(0, \sigma_2^2) \tag{4.9}$$

where $x_{1,t}$ is the location at time t along one axis (here, longitude) and $x_{2,t}$ is for another, generally orthogonal, axis (in here, latitude). The parameter u_1 is the rate of longitudinal movement and u_2 is the rate of latitudinal movement. We add errors to our observations of location:

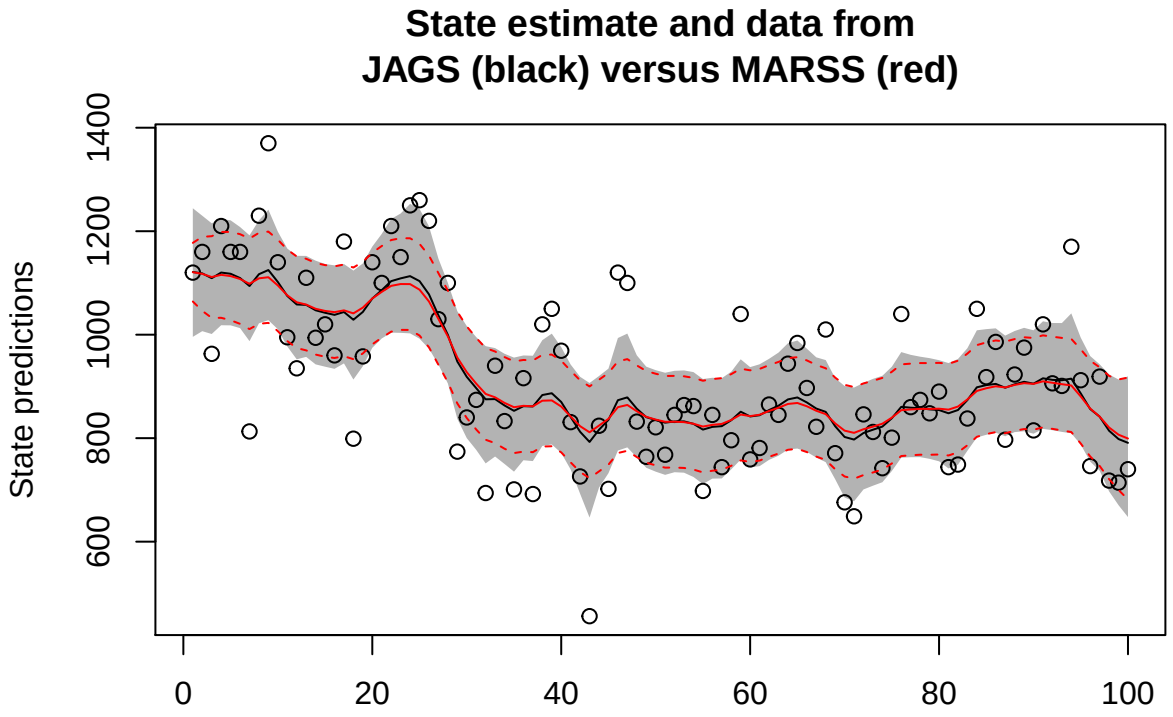


Figure 4.6: The estimated states from the Bayesian fit along with 95% credible intervals (black and grey) with the MLE states and 95% condidence intervals in red.

$$y_{1,t} = x_{1,t} + v_{1,t}, \quad v_{1,t} \sim \text{N}(0, \eta_1^2) \tag{4.10}$$
$$y_{2,t} = x_{2,t} + v_{2,t}, \quad v_{2,t} \sim \text{N}(0, \eta_2^2), \tag{4.11}$$

This model is comprised of two separate univariate state-space models. Note that y_1 depends only on x_1 and y_2 depends only on x_2. There are no actual interactions between these two univariate models. However, we can write the model down in the form of a multivariate model using diagonal variance-covariance matrices and a diagonal design ($\mathbf{Z}$) matrix. Because the variance-covariance matrices and $\mathbf{Z}$ are diagonal, the x_1:y_1 and x_2:y_2 processes will be independent as intended. Here are Equations (4.9) and (4.11) written as a MARSS model (in matrix form):

$$\begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} = \begin{bmatrix} x_{1,t-1} \\ x_{2,t-1} \end{bmatrix} + \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} + \begin{bmatrix} w_{1,t} \\ w_{2,t} \end{bmatrix}, \ \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix} \sigma_1^2 & 0 \\ 0 & \sigma_2^2 \end{bmatrix}\right) \tag{4.12}$$

$$\begin{bmatrix} y_{1,t} \\ y_{2,t} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_{1,t} \\ x_{2,t} \end{bmatrix} + \begin{bmatrix} v_{1,t} \\ v_{2,t} \end{bmatrix}, \ \mathbf{v}_t \sim \text{MVN}\left(0, \begin{bmatrix} \eta_1^2 & 0 \\ 0 & \eta_2^2 \end{bmatrix}\right) \tag{4.13}$$

The variance-covariance matrix for $\mathbf{w}_t$ is a diagonal matrix with unequal variances, σ_1^2 and σ_2^2. The variance-covariance matrix for $\mathbf{v}_t$ is a diagonal matrix with unequal variances, η_1^2 and η_2^2. We can write this succinctly as

$$\mathbf{x}_t = \mathbf{x}_{t-1} + \mathbf{u} + \mathbf{w}_t, \quad \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \tag{4.14}$$
$$\mathbf{y}_t = \mathbf{x}_t + \mathbf{v}_t, \quad \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}). \tag{4.15}$$

4.8 Problems

1. Write the equations for each of these models: ARIMA(0,0,0), ARIMA(0,1,0), ARIMA(1,0,0), ARIMA(0,0,1), ARIMA(1,0,1). Read the help file for the `Arima()` function (in the **forecast** package) if you are fuzzy on the arima notation.

2. The **MARSS** package includes a data set of sharp-tailed grouse in Washington. Load the data to use as follows:

```
library(MARSS)
dat = log(grouse[, 2])
```

   Consider these two models for the data:

   - Model 1 random walk with no drift observed with no error
   - Model 2 random walk with drift observed with no error

   Written as a univariate state-space model, model 1 is

$$\begin{aligned} x_t &= x_{t-1} + w_t \text{ where } w_t \sim \text{N}(0, q) \\ x_0 &= a \\ y_t &= x_t \end{aligned} \tag{4.16}$$

   Model 2 is almost identical except with u added

$$\begin{aligned} x_t &= x_{t-1} + u + w_t \text{ where } w_t \sim \text{N}(0, q) \\ x_0 &= a \\ y_t &= x_t \end{aligned} \tag{4.17}$$

   y is the log grouse count in year t.

   a. Plot the data. The year is in column 1 of `grouse`.

   b. Fit each model using `MARSS()`.

   c. Which one appears better supported given AICc?

   d. Load the **forecast** package. Use `?auto.arima` to learn what it does. Then use `auto.arima(dat)` to fit the data. Next run `auto.arima(dat, trace=TRUE)` to see all the ARIMA models that the function compared. Note, ARIMA(0,1,0) is a random walk with b=1. ARIMA(0,1,0) with drift would be a random walk (b=1) with drift (with u).

   e. Is the difference in the AICc values between a random walk with and without drift comparable between MARSS() and auto.arima()?

Note when using `auto.arima()`, an AR(1) model of the following form will be fit (notice the b): $x_t = bx_{t-1} + w_t$. `auto.arima()` refers to this model $x_t = x_{t-1} + w_t$, which is also AR(1) but with $b = 1$, as ARIMA(0,1,0). This says that the first difference of the data (that's the 1 in the middle) is a ARMA(0,0) process (the 0s in the 1st and 3rd spots). So ARIMA(0,1,0) means this: $x_t - x_{t-1} = w_t$.

3. Create a random walk with drift time series using `cumsum()` and `rnorm()`. Look at the `rnorm()` help file (`?rnorm`) to make sure you know what the arguments to the `rnorm()` are.

```
dat = cumsum(rnorm(100, 0.1, 1))
```

   a. What is the order of this random walk written as ARIMA(p, d, q)? "what is the order" means "what is p, d, and q. Model"order" is how `arima()` and `Arima()` specify arima models.

   b. Fit that model using `Arima()` in the **forecast** package. You'll need to specify the arguments `order` and `include.drift`. Use `?Arima` to review what that function does if needed.

   c. Write out the equation for this random walk as a univariate state-space model. Notice that there is no observation error, but still write this as a state-space model.

   d. Fit that model with `MARSS()`.

   e. How are the two estimates from `Arima()` and `MARSS()` different?

4. The first-difference of `dat` used in the previous problem is:

```
diff.dat = diff(dat)
```

   Use `?diff` to check what the `diff()` function does.

   a. If x_t denotes a time series. What is the first difference of x? What is the second difference?

   b. What is the `x` model for `diff.dat`? Look at your answer to part (a) and the answer to part (e).

   c. Fit `diff.dat` using `Arima()`. You'll need to change the arguments `order` and `include.mean`.

   d. Fit with `MARSS()`. You will need to write the model for `diff.dat` as a state-space model. If you've done this right, the estimated parameters using `Arima()` and `MARSS()` will now be the same.

   This question should clue you into the fact that `Arima()` is not exactly fitting Equation (4.1). It's very similar, but not quite written that way. By the way, Equation (4.1) is how structural time series observed with error are written (state-space models). To recover the estimates that a function like `arima()` or `Arima()` returns, you need to write your state-space model in a specific way (as seen above).

5. `Arima()` will also fit what it calls an "AR(1) with drift". An AR(1) with drift is NOT this model:

$$x_t = bx_{t-1} + u + w_t \text{ where } w_t \sim \text{N}(0, q) \tag{4.18}$$

In the population dynamics literature, this equation is called the Gompertz model and is a type of density-dependent population model.

   a. Write R code to simulate Equation (4.18). Make b less than 1 and greater than 0. Set u and x_0 to whatever you want. You can use a for loop.

   b. Plot the trajectories and show that this model does not "drift" upward or downward. It fluctuates about a mean value.

   c. Hold b constant and change u. How do the trajectories change?

   d. Hold u constant and change b. Make sure to use a b close to 1 and another close to 0. How do the trajectories change?

   e. Do 2 simulations each with the same w_t. In one simulation, set $u = 1$ and in the other $u = 2$. For both simulations, set $x_1 = u/(1-b)$. You can set b to whatever you want as long as $0 < b < 1$. Plot the 2 trajectories on the same plot. What is different?

We will fit what `Arima()` calls "AR(1) with drift" models in the chapter on MARSS models with covariates.

6. The **MARSS** package includes a data set of gray whales. Load the data to use as follows:

```
library(MARSS)
dat = log(graywhales[, 2])
```

Fit a random walk with drift model observed with error to the data:

$$\begin{aligned} x_t &= x_{t-1} + u + w_t \text{ where } w_t \sim \text{N}(0, q) \\ y_t &= x_t + v_t \text{ where } v_t \sim \text{N}(0, r) \\ x_0 &= a \end{aligned} \tag{4.19}$$

y is the whale count in year t. x is interpreted as the 'true' unknown population size that we are trying to estimate.

   a. Fit this model with `MARSS()`

   b. Plot the estimated x as a line with the actual counts added as points. x is in `fit$states`. It is a matrix. To plot using `plot()`, you will need to change it to a vector using `as.vector()` or `fit$states[1,]`

c. Simulate 1000 sample gray whale populstion trajectories (the x in your model) using the estimated u and q starting at the estimated x in 1997. You can do this with a couple for loops or write something terse with `cumsum()` and `apply()`.

d. Using these simulated trajectories, what is your estimate of the probability that the grey whale population will be above 50,000 graywhales in 2007?

e. What kind(s) of uncertainty does your estimate above NOT include?

7. Fit the following models to the graywhales data using MARSS(). Assume $b = 1$.

- Model 1 Process error only model with drift
- Model 2 Process error only model without drift
- Model 3 Process error with drift and observation error with observation error variance fixed = 0.05.
- Model 4 Process error with drift and observation error with observation error variance estimated.

a. Compute the AICc's for each model and likelihood or deviance (-2 * log likelihood). Where to find these? Try `names(fit)`. `logLik()` is the standard R function to return log-likelihood from fits.

b. Calculate a table of ΔAICc values and AICc weights.

c. Show the acf of the model and state residuals for the best model. You will need a vector of the residuals to do this. If `fit` is the fit from a fit call like `fit = MARSS(dat)`, you get the residuals using this code:

```
residuals(fit)$state.residuals[1, ]
residuals(fit)$model.residuals[1, ]
```

Do the acf's suggest any problems?

8. Evaluate the predictive accuracy of forecasts using the **forecast** package using the `airmiles` dataset. Load the data to use as follows:

```
library(forecast)
dat = log(airmiles)
n = length(dat)
training.dat = dat[1:(n - 3)]
test.dat = dat[(n - 2):n]
```

This will prepare the training data and set aside the last 3 data points for validation.

a. Fit the following four models using `Arima()`: ARIMA(0,0,0), ARIMA(1,0,0), ARIMA(0,0,1), ARIMA(1,0,1).

b. Use `forecast()` to make 3 step ahead forecasts from each.

c. Calculate the MASE statistic for each using the `accuracy()` function in the **forecast** package. Type `?accuracy` to learn how to use this function.

d. Present the results in a table.

e. Which model is best supported based on the MASE statistic?

9. The WhaleNet Archive of STOP Data has movement data on loggerhead turtles on the east coast of the US from ARGOS tags. The **MARSS** package `loggerheadNoisy` dataset is lat/lot data on eight individuals, however we have corrupted this data severely by adding random errors in order to create a "bad tag" problem (very noisy). Use `head(loggerheadNoisy)` to get an idea of the data. Then load the data on one turtle, MaryLee. MARSS needs time across the columns to you need to use transpose the data (as shown).

```
turtlename = "MaryLee"
dat = loggerheadNoisy[which(loggerheadNoisy$turtle == turtlename),
    5:6]
dat = t(dat)
```

a. Plot MaryLee's locations (as a line not dots). Put the latitude locations on the y-axis and the longitude on the y-axis. You can use `rownames(dat)` to see which is in which row. You can just use `plot()` for the homework. But if you want, you can look at the MARSS Manual chapter on animal movement to see how to plot the turtle locations on a map using the **maps** package.

b. Analyze the data with a state-space model (movement observed with error) using

```
fit0 = MARSS(dat)
```

Look at the output from the above MARSS call. What is the meaning of the parameters output from MARSS in terms of turtle movement? What exactly is the u estimate for example? Look at the data and think about the model you fit.

c. What assumption did the default MARSS model make about observation error and process error? What does that assumption mean in terms of how steps in the N-S and E-W directions are related? What does that assumption mean in terms of our assumption about the latitudal and longitudinal observation errors?

d. Does MaryLee move faster in the latitude direction versus longitude direction?

e. Add MaryLee's estimated "true" positions to your plot of her locations. You can use `lines(x, y, col="red")` (with x and y replaced with your x and y data). The true position is the "state". This is in the states element of an output from MARSS `fit0$states`.

f. Fit the following models with different assumptions regarding the movement in the lat/lon direction:

- Lat/lon movements are independent but the variance is the same

- Lat/lon movements are correlated and lat/lon variances are different
- Lat/lon movements are correlated and the lat/lon variances are the same.

You only need to change `Q` specification. Your MARSS call will now look like the following with `...` replaced with your `Q` specification.

```
fit1 = MARSS(dat, list(Q = ...))
```

g. Plot your state residuals (true location residuals). What are the problems? Discuss in reference to your plot of the location data. Here is how to get state residuals from `MARSS()` output:

```
resids = residuals(fit0)$state.residuals
```

The lon residuals are in row 1 and lat residuals are in row 2 (same order as the data).

Chapter 5

Multivariate state-space models without covariates

This lab show you how to fit multivariate state-space models using the **MARSS** package. This chapter is an example which uses model selection to test different population structures in west coast harbor seals. See Holmes et al. (2014) for a fuller version of this examples. Chapter **??** shows an example of a multivariate state-space model with covariates.

A script with all the R code in the chapter can be downloaded here.

Data and packages

All the data used in the chapter are in the **MARSS** package. For most examples, we will use the `MARSS()` function to fit models via maximum-likelihood. We also show how to fit a Bayesian model using JAGS. For this section you will need the **R2jags** and **coda** packages. To run the JAGS code, you will also need JAGS installed. See Chapter 8 for more details on JAGS.

```
library(MARSS)
library(R2jags)
library(coda)
```

5.1 Overview

As discussed in Chapter 4, the **MARSS** package fits multivariate state-space models in this form:

$$\begin{aligned} \mathbf{x}_t &= \mathbf{B}\mathbf{x}_{t-1} + \mathbf{u} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \text{N}(0, \mathbf{Q}) \\ \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \text{N}(0, \mathbf{R}) \\ \mathbf{x}_0 &= \boldsymbol{\mu} \end{aligned} \tag{5.1}$$

where each of the bolded terms are matrices. Those that are bolded and small (not capitalized) have one column only, so are column matrices.

To fit a multivariate time-series model with the **MARSS** package, you need to first determine the size and structure of each of the parameter matrices: $\mathbf{B}$, $\mathbf{u}$, $\mathbf{Q}$, $\mathbf{Z}$, $\mathbf{a}$, $\mathbf{R}$ and $\boldsymbol{\mu}$. This requires first writing down your model in matrix form. We will illustarte this with a series of models for the temporal population dynamics of West coast harbor seals.

5.2 West coast harbor seals counts

In this example, we will use multivariate state-space models to combine surveys from four survey regions to estimate the average long-term population growth rate and the year-to-year variability in that population growth rate.

We have five regions (or sites) where harbor seals were censused from 1978-1999 while hauled out of land[1]. During the period of this dataset, harbor seals were recovering steadily after having been reduced to low levels by hunting prior to protection. We will assume that the underlying population process is a stochastic exponential growth process with mean rates of increase that were not changing through 1978-1999.

The survey methodologies were consistent throughout the 20 years of the data but we do not know what fraction of the population that each region represents nor do we know the observation-error variance for each region. Given differences between the numbers of haul-outs in each region, the observation errors may be quite different. The regions have had different levels of sampling; the best sampled region has only 4 years missing while the worst has over half the years missing (Figure 5.1).

5.2.1 Load the harbor seal data

The harbor seal data are included in the **MARSS** package as matrix with years in column 1 and the logged counts in the other columns. Let's look at the first few years of data:

```
print(harborSealWA[1:8, ], digits = 3)
```

```
     Year  SJF  SJI EBays PSnd  HC
[1,] 1978 6.03 6.75  6.63 5.82 6.6
[2,] 1979   NA   NA    NA   NA  NA
```

[1] Jeffries et al. 2003. Trends and status of harbor seals in Washington State: 1978-1999. Journal of Wildlife Management 67(1):208–219

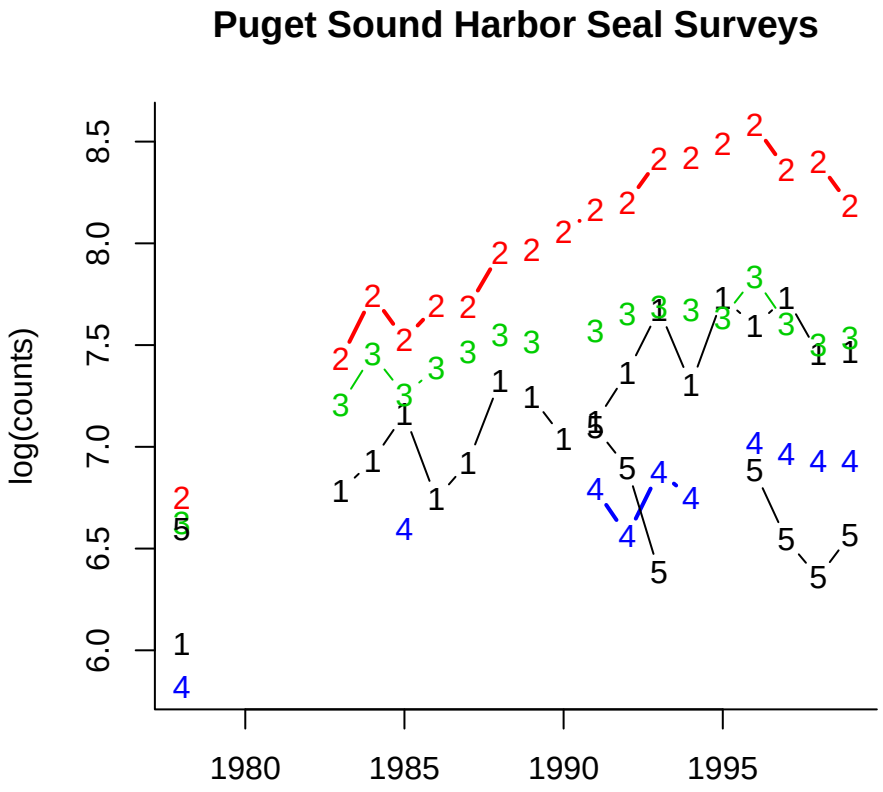


Figure 5.1: Plot of the of the count data from the five harbor seal regions (Jeffries et al. 2003). The numbers on each line denote the different regions: 1) Strait of Juan de Fuca (SJF), 2) San Juan Islands (SJI), 2) Eastern Bays (EBays), 4) Puget Sound (PSnd), and 5) Hood Canal (HC). Each region is an index of the total harbor seal population in each region.

```
[3,] 1980   NA   NA    NA   NA  NA
[4,] 1981   NA   NA    NA   NA  NA
[5,] 1982   NA   NA    NA   NA  NA
[6,] 1983 6.78 7.43  7.21   NA  NA
[7,] 1984 6.93 7.74  7.45   NA  NA
[8,] 1985 7.16 7.53  7.26 6.60  NA
```

We are going to leave out Hood Canal (HC) since that region is somewhat isolated from the others and experiencing very different conditions due to hypoxic events and periodic intense killer whale predation. We will set up the data as follows:

```
years = harborSealWA[, "Year"]
dat = harborSealWA[, !(colnames(harborSealWA) %in% c("Year",
    "HC"))]
dat = t(dat)  #transpose to have years across columns
colnames(dat) = years
n = nrow(dat) - 1
```

5.3 A single well-mixed population

When we are looking at data over a large geographic region, we might make the assumption that the different census regions are measuring a single population if we think animals are moving sufficiently such that the whole area (multiple regions together) is "well-mixed". We write a model of the total population abundance for this case as:

$$n_t = \exp(u + w_t) n_{t-1}, \tag{5.2}$$

where n_t is the total count in year t, u is the mean population growth rate, and w_t is the deviation from that average in year t. We then take the log of both sides and write the model in log space:

$$x_t = x_{t-1} + u + w_t, \text{ where } w_t \sim \text{N}(0, q) \tag{5.3}$$

$x_t = \log n_t$. When there is one effective population, there is one x, therefore $\mathbf{x}_t$ is a 1×1 matrix. This is our **state** model and x is called the "state". This is just the jargon used in this type of model (state-space model) for the hidden state that you are estimating from the data. "Hidden" means that you observe this state with error.

5.3.1 The observation process

We assume that all four regional time series are observations of this one population trajectory but they are scaled up or down relative to that trajectory. In effect, we think of each regional

survey as an index of the total population. With this model, we do not think the regions represent independent subpopulations but rather independent observations of one population. Our model for the data, $\mathbf{y}_t = \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t$, is written as:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}_t = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} x_t + \begin{bmatrix} 0 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}_t \tag{5.4}$$

Each y_i is the observed time series of counts for a different region. The a's are the bias between the regional sample and the total population. $\mathbf{Z}$ specifies which observation time series, y_i, is associated with which population trajectory, x_j. In this case, $\mathbf{Z}$ is a matrix with 1 column since each region is an observation of the one population trajectory.

We allow that each region could have a unique observation variance and that the observation errors are independent between regions. We assume that the observations errors on log(counts) are normal and thus the errors on (counts) are log-normal. The assumption of normality is not unreasonable since these regional counts are the sum of counts across multiple haul-outs. We specify independent observation errors with different variances by specifying that $\mathbf{v} \sim \text{MVN}(0, \mathbf{R})$, where

$$\mathbf{R} = \begin{bmatrix} r_1 & 0 & 0 & 0 \\ 0 & r_2 & 0 & 0 \\ 0 & 0 & r_3 & 0 \\ 0 & 0 & 0 & r_4 \end{bmatrix} \tag{5.5}$$

This is a diagonal matrix with unequal variances. The shortcut for this structure in `MARSS()` is `"diagonal and unequal"`.

5.3.2 Fitting the model

We need to write the model in the form of Equation (5.1) with each parameter written as a matrix. The observation model (Equation (5.4)) is already in matrix form. Let's write the state model in matrix form too:

$$[x]_t = [1][x]_{t-1} + [u] + [w]_t, \text{ where } [w]_t \sim \text{N}(0, [q]) \tag{5.6}$$

It is very simple since all terms are 1×1 matrices.

To fit our model with `MARSS()`, we set up a list which precisely describes the size and structure of each parameter matrix. Fixed values in a matrix are designated with their numeric value and estimated values are given a character name and put in quotes. Our model list for a single well-mixed population is:

```
mod.list.0 = list(B = matrix(1), U = matrix("u"), Q = matrix("q"),
    Z = matrix(1, 4, 1), A = "scaling", R = "diagonal and unequal",
    x0 = matrix("mu"), tinitx = 0)
```

and fit:

```
fit.0 = MARSS(dat, model = mod.list.0)
```

```
Success! abstol and log-log tests passed at 32 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 32 iterations.
Log-likelihood: 21.62931
AIC: -23.25863   AICc: -19.02786

                Estimate
A.SJI            0.79583
A.EBays          0.27528
A.PSnd          -0.54335
R.(SJF,SJF)      0.02883
R.(SJI,SJI)      0.03063
R.(EBays,EBays)  0.01661
R.(PSnd,PSnd)    0.01168
U.u              0.05537
Q.q              0.00642
x0.mu            6.22810

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

We already discussed that the short-cut `"diagonal and unequal"` means a diagonal matrix with each diagonal element having a different value. The short-cut `"scaling"` means the form of $\mathbf{a}$ in Equation (5.4) with one value set to 0 and the rest estimated. You should run the code in the list to make sure you see that each parameter in the list has the same form as in our mathematical equation for the model.

5.3.3 Model residuals

The model fits fine but look at the model residuals (Figure 5.2). They have problems.

```
par(mfrow = c(2, 2))
resids = residuals(fit.0)
for (i in 1:4) {
    plot(resids$model.residuals[i, ], ylab = "model residuals",
        xlab = "")
    abline(h = 0)
    title(rownames(dat)[i])
}
```

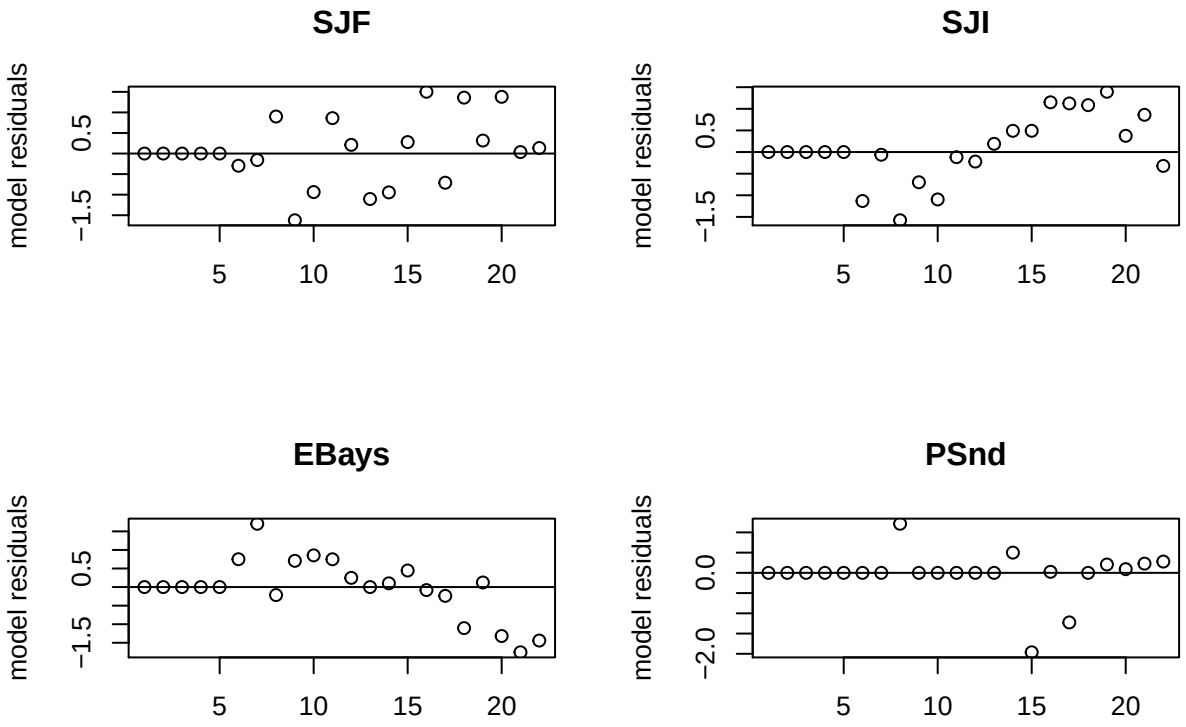


Figure 5.2: The model residuals for the first model. SJI and EBays do not look good.

5.4 Four subpopulations with temporally uncorrelated errors

The model for one well-mixed population was not very good. Another reasonable assumption is that the different census regions are measuring four different temporally independent subpopulations. We write a model of the log subpopulation abundances for this case as:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}_{t-1} + \begin{bmatrix} u \\ u \\ u \\ u \end{bmatrix} + \begin{bmatrix} w \\ w \\ w \\ w \end{bmatrix}_t$$
$$\text{where } \mathbf{w}_t \sim \text{MVN}\left(0, \begin{bmatrix} q & 0 & 0 & 0 \\ 0 & q & 0 & 0 \\ 0 & 0 & q & 0 \\ 0 & 0 & 0 & q \end{bmatrix}\right) \tag{5.7}$$
$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}_0 = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \mu_3 \\ \mu_4 \end{bmatrix}_t$$

The **Q** matrix is diagonal with one variance value. This means that the process variance (variance in year-to-year population growth rates) is independent (good and bad years are not correlated) but the level of variability is the same across regions. We made the **u** matrix with one u value. This means that we assume the population growth rates are the same across regions.

Notice that we set the **B** matrix equal to a diagonal matrix with 1 on the diagonal. This is the "identity" matrix and it is like a 1 but for matrices. We do not need **B** for our model, but `MARSS()` requires a value.

5.4.1 The observation process

In this model, each survey is an observation of a different x:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}_t + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \end{bmatrix}_t \tag{5.8}$$

No a's can be estimated since we do not have multiple observations of a given x time series. Our **R** matrix doesn't change; the observation errors are still assumed to the independent with different variances.

Notice that our **Z** matrix changed. **Z** is specifying which y_i goes to which x_j. The one we have specified means that y_1 is observing x_1, y_2 observes x_2, etc. We could have set up **Z** like so

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{5.9}$$

This would mean that y_1 observes x_2, y_2 observes x_1, y_3 observes x_4, and y_4 observes x_3. Which x goes to which y is arbitrary; we need to make sure it is one-to-one. We will stay with **Z** as an identity matrix since y_i observing x_i makes it easier to remember which x goes with which y.

5.4.2 Fitting the model

We set up the model list for `MARSS()` as:

```
mod.list.1 = list(B = "identity", U = "equal", Q = "diagonal and equal",
    Z = "identity", A = "scaling", R = "diagonal and unequal",
    x0 = "unequal", tinitx = 0)
```

We introduced a few more short-cuts. `"equal"` means all the values in the matrix are the same. `"diagonal and equal"` means that the matrix is diagonal with one value on the diagonal. `"unequal"` means that all values in the matrix are different.

We can then fit our model for 4 subpopulations as:

```
fit.1 = MARSS(dat, model = mod.list.1)
```

5.5 Four subpopulations with temporally correlated errors

Another reasonable assumption is that the different census regions are measuring different subpopulations but that the year-to-year population growth rates are correlated (good and bad year coincide). The only parameter that changes is the **Q** matrix:

$$\mathbf{Q} = \begin{bmatrix} q & c & c & c \\ c & q & c & c \\ c & c & q & c \\ c & c & c & q \end{bmatrix} \tag{5.10}$$

This **Q** matrix structure means that the process variance (variance in year-to-year population growth rates) is the same across regions and the covariance in year-to-year population growth rates is also the same across regions.

5.5.1 Fitting the model

Set up the model list for `MARSS()` as:

```
mod.list.2 = mod.list.1
mod.list.2$Q = "equalvarcov"
```

`"equalvarcov"` is a shortcut for the matrix form in Equation (5.10).

Fit the model with:

```
fit.2 = MARSS(dat, model = mod.list.2)
```

Results are not shown, but here are the AICc. This last model is much better:

```
c(fit.0$AICc, fit.1$AICc, fit.2$AICc)
```

```
[1] -19.02786 -22.20194 -41.00511
```

5.5.2 Model residuals

Look at the model residuals (Figure 5.3). They are also much better.

Figure 5.4 shows the estimated states for each region using this code:

```
par(mfrow = c(2, 2))
for (i in 1:4) {
    plot(years, fit.2$states[i, ], ylab = "log subpopulation estimate",
        xlab = "", type = "l")
    lines(years, fit.2$states[i, ] - 1.96 * fit.2$states.se[i,
        ], type = "l", lwd = 1, lty = 2, col = "red")
    lines(years, fit.2$states[i, ] + 1.96 * fit.2$states.se[i,
        ], type = "l", lwd = 1, lty = 2, col = "red")
    title(rownames(dat)[i])
}
```

5.6 Using MARSS models to study spatial structure

For our next example, we will use MARSS models to test hypotheses about the population structure of harbor seals on the west coast. For this example, we will evaluate the support for different population structures (numbers of subpopulations) using different **Z**s to specify

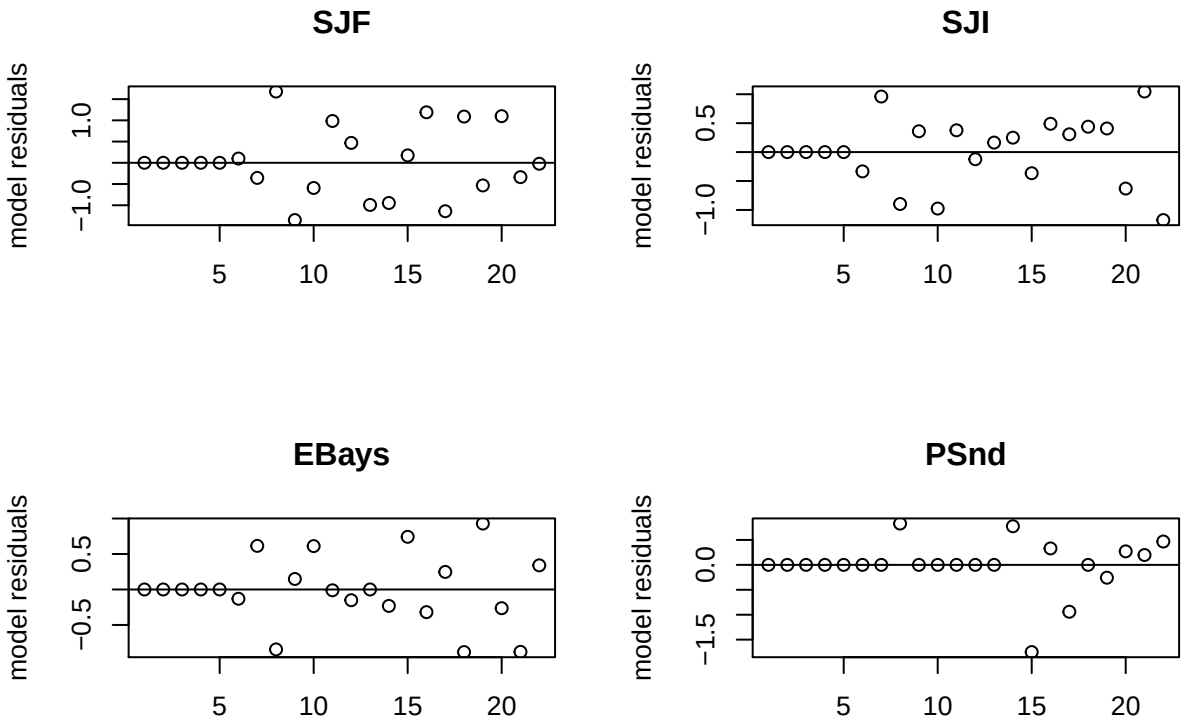


Figure 5.3: The model residuals for the model with four temporally correlated subpopulations.

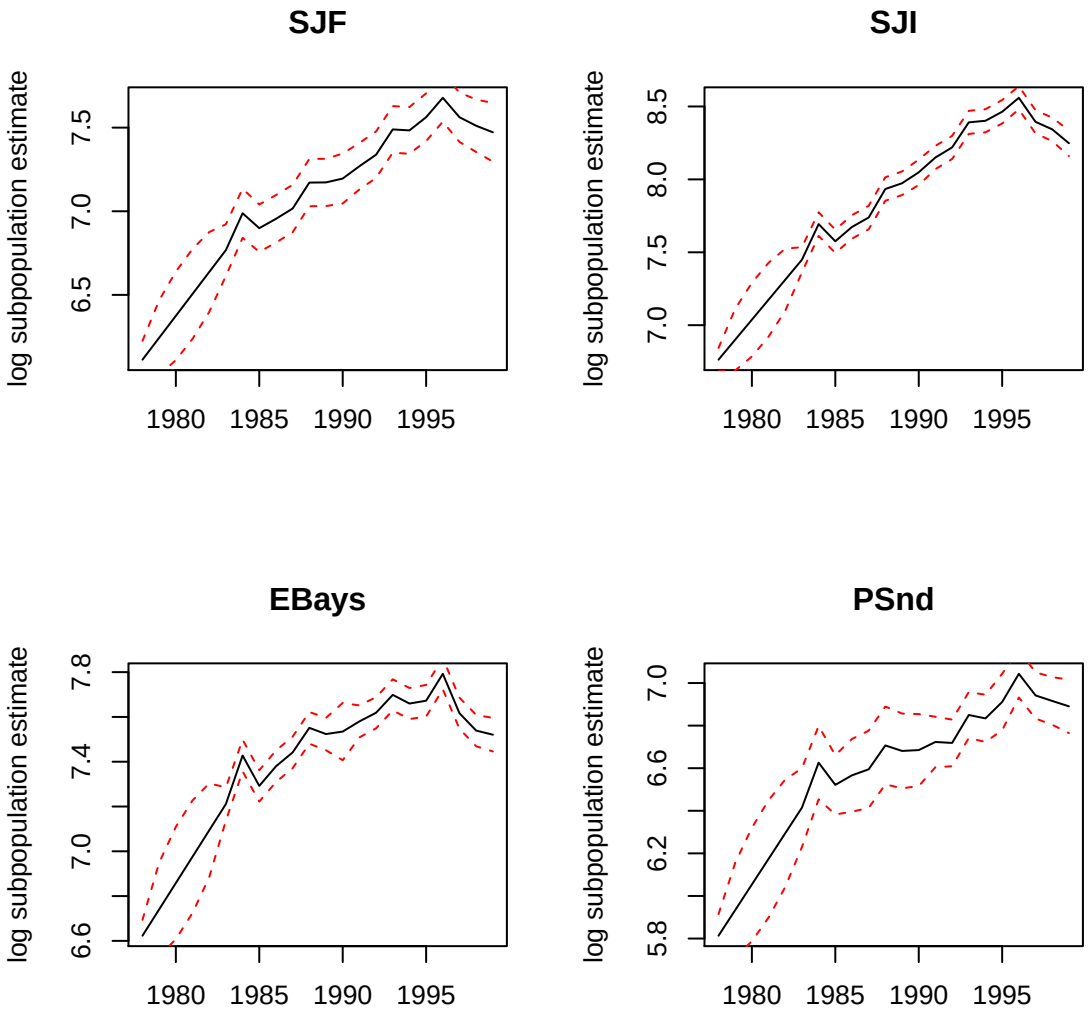


Figure 5.4: Plot of the estimate of log harbor seals in each region. The 95% confidence intervals on the population estimates are the dashed lines. These are not the confidence intervals on the observations, and the observations (the numbers) will not fall between the confidence interval lines.

how survey regions map onto subpopulations. We will assume correlated process errors with the same magnitude of process variance and covariance. We will assume independent observations errors with equal variances at each site. We could do unequal variances but it takes a long time to fit so for this example, the observation variances are set equal.

The dataset we will use is `harborSeal`, a 29-year dataset of abundance indices for 12 regions along the U.S. west coast between 1975-2004 (Figure 5.5).

We start by setting up our data matrix. We will leave off Hood Canal.

```
years = harborSeal[, "Year"]
good = !(colnames(harborSeal) %in% c("Year", "HoodCanal"))
sealData = t(harborSeal[, good])
```

5.7 Hypotheses regarding spatial structure

We will evaluate the data support for the following hypotheses about the population structure:

- H1: `stock` 3 subpopulations defined by management units
- H2: `coast+PS` 2 subpopulations defined by coastal versus WA inland
- H3: `N+S` 2 subpopulations defined by north and south split in the middle of Oregon
- H4:`NC+strait+PS+SC` 4 subpopulations defined by N coastal, S coastal, SJF+Georgia Strait, and Puget Sound
- H5: `panmictic` All regions are part of the same panmictic population
- H6: `site` Each of the 11 regions is a subpopulation

These hypotheses translate to these $\mathbf{Z}$ matrices (H6 not shown; it is an identity matrix):

	$H1$	$H2$	$H4$	$H5$
	pnw ps ca	coast pc	nc is ps sc	pan
Coastal Estuaries	$\begin{bmatrix}1&0&0\end{bmatrix}$	$\begin{bmatrix}1&0\end{bmatrix}$	$\begin{bmatrix}1&0&0&0\end{bmatrix}$	$\begin{bmatrix}1\end{bmatrix}$
Olympic Peninsula	$\begin{bmatrix}1&0&0\end{bmatrix}$	$\begin{bmatrix}1&0\end{bmatrix}$	$\begin{bmatrix}1&0&0&0\end{bmatrix}$	$\begin{bmatrix}1\end{bmatrix}$
Str. Juan de Fuca	$\begin{bmatrix}0&1&0\end{bmatrix}$	$\begin{bmatrix}0&1\end{bmatrix}$	$\begin{bmatrix}0&1&0&0\end{bmatrix}$	$\begin{bmatrix}1\end{bmatrix}$
San Juan Islands	$\begin{bmatrix}0&1&0\end{bmatrix}$	$\begin{bmatrix}0&1\end{bmatrix}$	$\begin{bmatrix}0&1&0&0\end{bmatrix}$	$\begin{bmatrix}1\end{bmatrix}$
Eastern Bays	$\begin{bmatrix}0&1&0\end{bmatrix}$	$\begin{bmatrix}0&1\end{bmatrix}$	$\begin{bmatrix}0&0&1&0\end{bmatrix}$	$\begin{bmatrix}1\end{bmatrix}$
Puget Sound	$\begin{bmatrix}0&1&0\end{bmatrix}$	$\begin{bmatrix}0&1\end{bmatrix}$	$\begin{bmatrix}0&0&1&0\end{bmatrix}$	$\begin{bmatrix}1\end{bmatrix}$
CA Mainland	$\begin{bmatrix}0&0&1\end{bmatrix}$	$\begin{bmatrix}1&0\end{bmatrix}$	$\begin{bmatrix}0&0&0&1\end{bmatrix}$	$\begin{bmatrix}1\end{bmatrix}$
CA Channel Islands	$\begin{bmatrix}0&0&1\end{bmatrix}$	$\begin{bmatrix}1&0\end{bmatrix}$	$\begin{bmatrix}0&0&0&1\end{bmatrix}$	$\begin{bmatrix}1\end{bmatrix}$
OR North Coast	$\begin{bmatrix}1&0&0\end{bmatrix}$	$\begin{bmatrix}1&0\end{bmatrix}$	$\begin{bmatrix}1&0&0&0\end{bmatrix}$	$\begin{bmatrix}1\end{bmatrix}$
OR South Coast	$\begin{bmatrix}1&0&0\end{bmatrix}$	$\begin{bmatrix}1&0\end{bmatrix}$	$\begin{bmatrix}0&0&0&1\end{bmatrix}$	$\begin{bmatrix}1\end{bmatrix}$
Georgia Strait	$\begin{bmatrix}0&1&0\end{bmatrix}$	$\begin{bmatrix}0&1\end{bmatrix}$	$\begin{bmatrix}0&1&0&0\end{bmatrix}$	$\begin{bmatrix}1\end{bmatrix}$

To tell `MARSS()` the form of $\mathbf{Z}$, we construct the same matrix in R. For example, for hypotheses 1, we can write:

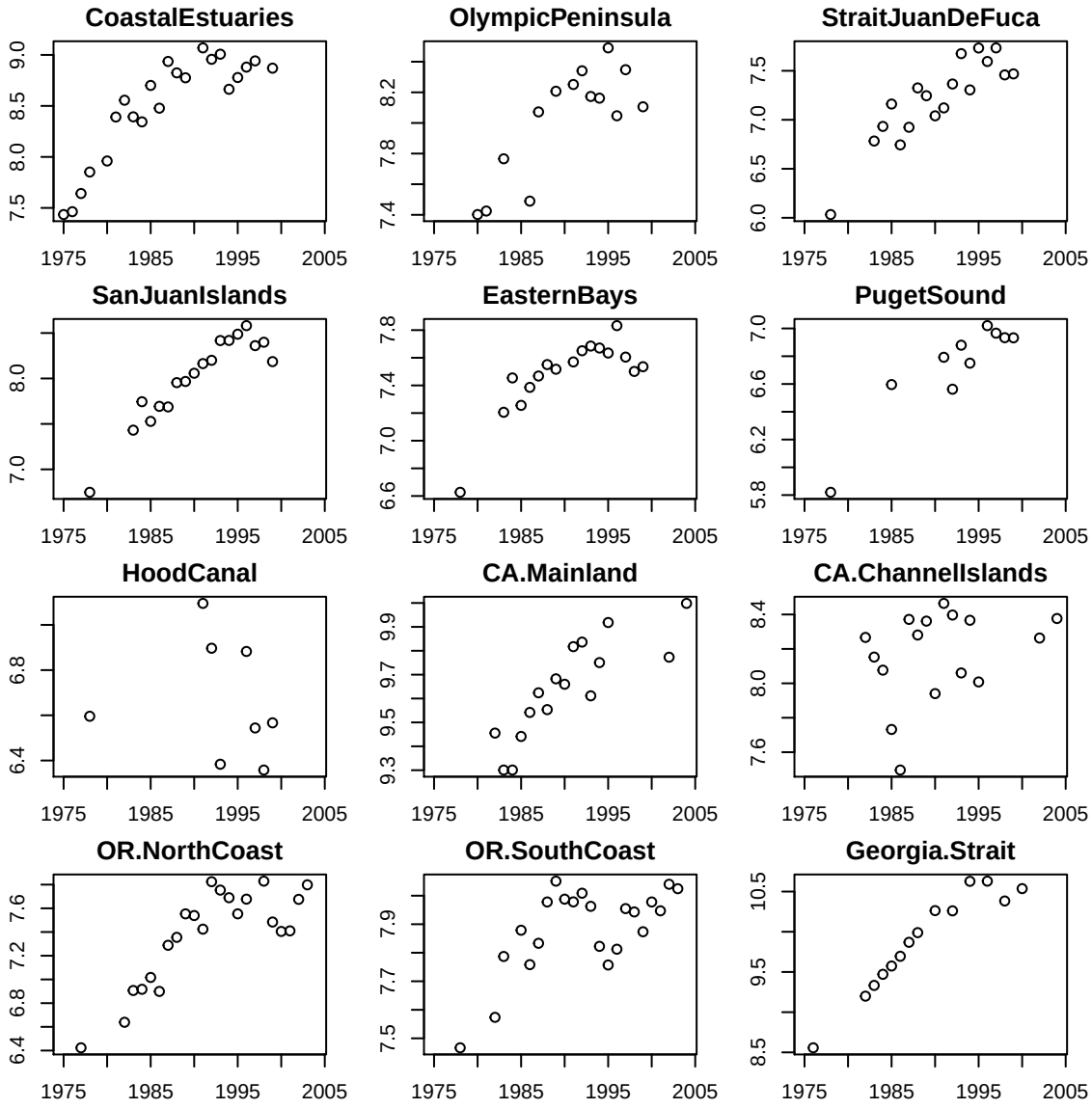


Figure 5.5: Plot of log counts at each survey region in the harborSeal dataset. Each region is an index of the harbor seal abundance in that region.

```
Z.model=matrix(0,11,3)
Z.model[c(1,2,9,10),1]=1  #which elements in col 1 are 1
Z.model[c(3:6,11),2]=1  #which elements in col 2 are 1
Z.model[7:8,3]=1  #which elements in col 3 are 1
```

Or we can use a short-cut by specifying **Z** as a factor that has the name of the subpopulation associated with each row in **y**. For hypothesis 1, this is

```
Z1 = factor(c("pnw", "pnw", rep("ps", 4), "ca", "ca", "pnw",
    "pnw", "ps"))
```

Notice it is 11 elements in length; one element for each row of data.

5.8 Set up the hypotheses as different models

Only the **Z** matrices change for our model. We will set up a base model list used for all models.

```
mod.list = list(
B = "identity",
U = "unequal",
Q = "equalvarcov",
Z = "placeholder",
A = "scaling",
R = "diagonal and equal",
x0 = "unequal",
tinitx = 0 )
```

Then we set up the **Z** matrices using the factor short-cut.

```
Z.models = list(
H1=factor(c("pnw","pnw",rep("ps",4),"ca","ca","pnw","pnw","ps")),
H2=factor(c(rep("coast",2),rep("ps",4),rep("coast",4),"ps")),
H3=factor(c(rep("N",6),"S","S","N","S","N")),
H4=factor(c("nc","nc","is","is","ps","ps","sc","sc","nc","sc","is")),
H5=factor(rep("pan",11)),
H6=factor(1:11) #site
)
names(Z.models)=
     c("stock","coast+PS","N+S","NC+strait+PS+SC","panmictic","site")
```

5.8.1 Fit the models

We loop through the models, fit and store the results:

```
out.tab = NULL
fits = list()
for (i in 1:length(Z.models)) {
    mod.list$Z = Z.models[[i]]
    fit = MARSS(sealData, model = mod.list, silent = TRUE, control = list(maxit = 1000))
    out = data.frame(H = names(Z.models)[i], logLik = fit$logLik,
        AICc = fit$AICc, num.param = fit$num.params, m = length(unique(Z.models[[i]])),
        num.iter = fit$numIter, converged = !fit$convergence)
    out.tab = rbind(out.tab, out)
    fits = c(fits, list(fit))
}
```

We will use AICc and AIC weights to summarize the data support for the different hypotheses. First we will sort the fits based on AICc:

```
min.AICc = order(out.tab$AICc)
out.tab.1 = out.tab[min.AICc, ]
```

Next we add the ΔAICc values by subtracting the lowest AICc:

```
out.tab.1 = cbind(out.tab.1, delta.AICc = out.tab.1$AICc - out.tab.1$AICc[1])
```

Relative likelihood is defined as $\exp(-\Delta\text{AICc}/2)$.

```
out.tab.1 = cbind(out.tab.1, rel.like = exp(-1 * out.tab.1$delta.AICc/2))
```

The AIC weight for a model is its relative likelihood divided by the sum of all the relative likelihoods.

```
out.tab.1 = cbind(out.tab.1, AIC.weight = out.tab.1$rel.like/sum(out.tab.1$rel.like))
```

Let's look at the model weights (`out.tab.1`):

	H	delta.AICc	AIC.weight	converged
NC+strait+PS+SC		0.00	0.979	TRUE
site		7.65	0.021	TRUE
N+S		36.97	0.000	TRUE
stock		47.02	0.000	TRUE
coast+PS		48.78	0.000	TRUE
panmictic		71.67	0.000	TRUE

5.9 Multivariate state-space models with JAGS

Here we show you how to fit a MARSS model for the harbor seal data using JAGS. We will focus on four time series from inland Washington and set up the data as follows:

```
sites = c("SJF", "SJI", "EBays", "PSnd")
Y = harborSealWA[, sites]
```

We will fit the model with four temporally independent subpopulations with the same population growth rate (u) and year-to-year variance (q). This is the model in Section 5.4.

5.10 Writing the model in JAGS

The first step is to write this model in JAGS. See Chapter 8 for more information on and examples of JAGS models.

```
jagsscript = cat("
model {
   U ~ dnorm(0, 0.01);
   tauQ~dgamma(0.001,0.001);
   Q <- 1/tauQ;

   # Estimate the initial state vector of population abundances
   for(i in 1:nSites) {
      X[1,i] ~ dnorm(3,0.01); # vague normal prior
   }

   # Autoregressive process for remaining years
   for(i in 2:nYears) {
      for(j in 1:nSites) {
         predX[i,j] <- X[i-1,j] + U;
         X[i,j] ~ dnorm(predX[i,j], tauQ);
      }
   }

   # Observation model
   # The Rs are different in each site
   for(i in 1:nSites) {
     tauR[i]~dgamma(0.001,0.001);
     R[i] <- 1/tauR[i];
   }
   for(i in 1:nYears) {
     for(j in 1:nSites) {
       Y[i,j] ~ dnorm(X[i,j],tauR[j]);
     }
   }
}
```

```
",
    file = "marss-jags.txt")
```

Then we write the data list, parameter list, and pass the model to the `jags()` function:

```
jags.data = list(Y = Y, nSites = dim(Y)[2], nYears = dim(Y)[1])  # named list
jags.params = c("X", "U", "Q")
model.loc = "marss-jags.txt"  # name of the txt file
mod_1 = jags(jags.data, parameters.to.save = jags.params, model.file = model.loc,
    n.chains = 3, n.burnin = 5000, n.thin = 1, n.iter = 10000,
    DIC = TRUE)
```

5.11 Plot the posteriors for the estimated states

We can plot any of the variables we chose to return to R in the `jags.params` list. Let's focus on the `X`. When we look at the dimension of the `X`, we can use the `apply()` function to calculate the means and 95 percent CIs of the estimated states.

```
# attach.jags attaches the jags.params to our workspace
attach.jags(mod_1)
means = apply(X, c(2, 3), mean)
upperCI = apply(X, c(2, 3), quantile, 0.975)
lowerCI = apply(X, c(2, 3), quantile, 0.025)
par(mfrow = c(2, 2))
nYears = dim(Y)[1]
for (i in 1:dim(means)[2]) {
    plot(means[, i], lwd = 3, ylim = range(c(lowerCI[, i], upperCI[,
        i])), type = "n", main = colnames(Y)[i], ylab = "log abundance",
        xlab = "time step")
    polygon(c(1:nYears, nYears:1, 1), c(upperCI[, i], rev(lowerCI[,
        i]), upperCI[1, i]), col = "skyblue", lty = 0)
    lines(means[, i], lwd = 3)
}
```

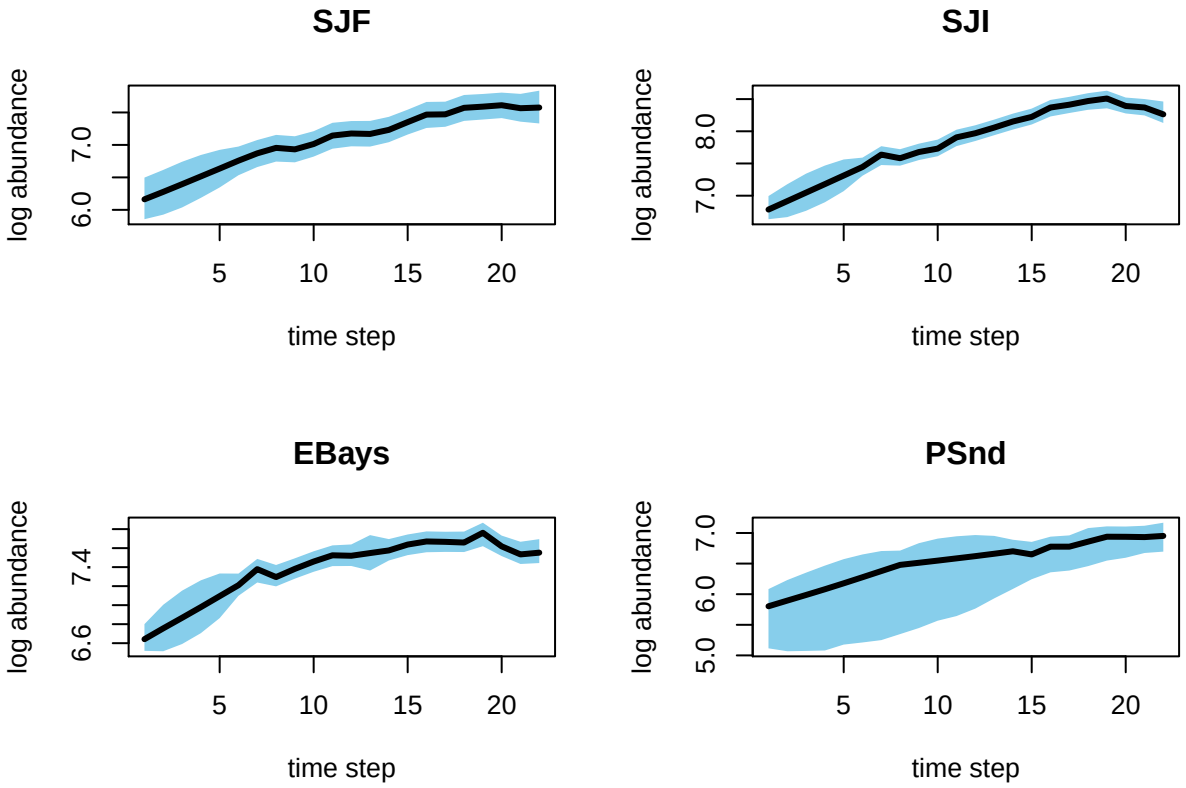


Figure 5.6: Plot of the posterior means and credible intervals for the estimated states.

Chapter 6

Dynamic linear models

Here we will use MARSS to analyze dynamic linear models (DLMs), wherein the parameters in a regression model are treated as time-varying. DLMs are used commonly in econometrics, but have received less attention in the ecological literature (c.f. Lamon et al., 1998; Scheuerell and Williams, 2005). Our treatment of DLMs is rather cursory—we direct the reader to excellent textbooks by Pole et al. (1994) and Petris et al. (2009) for more in-depth treatments of DLMs. The former focuses on Bayesian estimation whereas the latter addresses both likelihood-based and Bayesian estimation methods.

A script with all the R code in the chapter can be downloaded here.

Data

Most of the data used in the chapter are from the **MARSS** package. Install the package, if needed, and load:

```
library(MARSS)
```

The problem set uses an additional data set on spawners and recruits: KvichakSockeye.RData.

6.1 Overview

We begin our description of DLMs with a static regression model, wherein the i^{th} observation is a linear function of an intercept, predictor variable(s), and a random error term. For example, if we had one predictor variable (F), we could write the model as

$$y_i = \alpha + \beta F_i + v_i, \tag{6.1}$$

where the α is the intercept, β is the regression slope, F_i is the predictor variable matched to the i^{th} observation (y_i), and $v_i \sim \mathrm{N}(0, r)$. It is important to note here that there is no implicit ordering of the index i. That is, we could shuffle any/all of the (y_i, F_i) pairs in our dataset with no effect on our ability to estimate the model parameters.

We can write the model in Equation (6.1) using vector notation, such that

$$\begin{aligned} y_i &= \begin{pmatrix} 1 & F_i \end{pmatrix} \times \begin{pmatrix} \alpha \\ \beta \end{pmatrix} + v_i \\ &= \mathbf{F}_i^\top \boldsymbol{\theta} + v_i, \end{aligned} \tag{6.2}$$

and $\mathbf{F}_i^\top = (1, F_i)$ and $\boldsymbol{\theta} = (\alpha, \beta)^\top$.

In a DLM, however, the regression parameters are *dynamic* in that they "evolve" over time. For a single observation at time t, we can write

$$y_t = \mathbf{F}_t^\top \boldsymbol{\theta}_t + v_t, \tag{6.3}$$

where $\mathbf{F}_t$ is a column vector of regression variables at time t, $\boldsymbol{\theta}_t$ is a column vector of regression parameters at time t and $v_t \sim \mathrm{N}(0, r)$. This formulation presents two features that distinguish it from Equation (6.2). First, the observed data are explicitly time ordered (i.e., $\mathbf{y} = \{y_1, y_2, y_3, \ldots, y_T\}$), which means we expect them to contain implicit information. Second, the relationship between the observed datum and the predictor variables are unique at every time t (i.e., $\boldsymbol{\theta} = \{\boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \boldsymbol{\theta}_3, \ldots, \boldsymbol{\theta}_T\}$).

However, closer examination of Equation (6.3) reveals an apparent complication for parameter estimation. With only one datum at each time step t, we could, at best, estimate only one regression parameter, and even then, the 1:1 correspondence between data and parameters would preclude any estimation of parameter uncertainty. To address this shortcoming, we return to the time ordering of model parameters. Rather than assume the regression parameters are independent from one time step to another, we instead model them as an autoregressive process where

$$\boldsymbol{\theta}_t = \mathbf{G}_t \boldsymbol{\theta}_{t-1} + \mathbf{w}_t, \tag{6.4}$$

$\mathbf{G}_t$ is the parameter "evolution" matrix, and $\mathbf{w}_t$ is a vector of process errors, such that $\mathbf{w}_t \sim \mathrm{MVN}(\mathbf{0}, \mathbf{Q})$. The elements of $\mathbf{G}_t$ may be known and fixed a priori, or unknown and estimated from the data. Although we allow for $\mathbf{G}_t$ to be time-varying, we will typically assume that it is time invariant.

The idea is that the evolution matrix $\mathbf{G}_t$ deterministically maps the parameter space from one time step to the next, so the parameters at time t are temporally related to those before and after. However, the process is corrupted by stochastic error, which amounts to a degradation of information over time. If the diagonal elements of $\mathbf{Q}$ are relatively large,

then the parameters can vary widely from t to $t+1$. If $\mathbf{Q} = \mathbf{0}$, then $\boldsymbol{\theta}_1 = \boldsymbol{\theta}_2 = \boldsymbol{\theta}_T$ and we are back to the static model in Equation (6.1).

6.2 Example of a univariate DLM

Let's consider an example from the literature. Scheuerell and Williams (2005) used a DLM to examine the relationship between marine survival of Chinook salmon and an index of ocean upwelling strength along the west coast of the USA. Upwelling brings cool, nutrient-rich waters from the deep ocean to shallower coastal areas. Scheuerell & Williams hypothesized that stronger upwelling in April should create better growing conditions for phytoplankton, which would then translate into more zooplankton. In turn, juvenile salmon ("smolts") entering the ocean in May and June should find better foraging opportunities. Thus, for smolts entering the ocean in year t,

$$survival_t = \alpha_t + \beta_t F_t + v_t \text{ with } v_t \sim \text{N}(0, r), \tag{6.5}$$

and F_t is the coastal upwelling index (cubic meters of seawater per second per 100 m of coastline) for the month of April in year t.

Both the intercept and slope are time varying, so

$$\alpha_t = \alpha_{t-1} + w_t^{(\alpha)} \text{ with } w_t^{(\alpha)} \sim \text{N}(0, q_\alpha); \text{ and} \tag{6.6}$$
$$\beta_t = \beta_{t-1} + w_t^{(\beta)} \text{ with } w_t^{(\beta)} \sim \text{N}(0, q_\beta). \tag{6.7}$$

If we define $\boldsymbol{\theta}_t = (\alpha_t, \beta_t)^\top$, $\mathbf{G}_t = \mathbf{I} \; \forall \; t$, $\mathbf{w}_t = (w_t^{(1)}, w_t^{(2)})^\top$, and $\mathbf{Q} = \text{diag}(q_1, q_2)$, we get Equation (6.4). If we define $y_t = survival_t$ and $\mathbf{F}_t = (1, F_t)^\top$, we can write out the full univariate DLM as a state-space model with the following form:

$$\begin{gathered} \boldsymbol{\theta}_t = \mathbf{G}_t \boldsymbol{\theta}_{t-1} + \mathbf{w}_t \text{ with } \mathbf{w}_t \sim \text{MVN}(\mathbf{0}, \mathbf{Q}); \\ y_t = \mathbf{F}_t^\top \boldsymbol{\theta}_t + v_t \text{ with } v_t \sim \text{N}(0, r); \\ \boldsymbol{\theta}_0 \sim \text{MVN}(\boldsymbol{\pi}_0, \boldsymbol{\Lambda}_0). \end{gathered} \tag{6.8}$$

Equation (6.8) is, not surprisingly, equivalent to our standard MARSS model:

$$\begin{gathered} \mathbf{x}_t = \mathbf{B}_t \mathbf{x}_{t-1} + \mathbf{u}_t + \mathbf{C}_t \mathbf{c}_t + \mathbf{w}_t \text{ with } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}_t); \\ \mathbf{y}_t = \mathbf{Z}_t \mathbf{x}_t + \mathbf{a}_t + \mathbf{D}_t \mathbf{d}_t + \mathbf{v}_t \text{ with } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}_t); \\ \mathbf{x}_0 \sim \text{MVN}(\boldsymbol{\pi}, \boldsymbol{\Lambda}); \end{gathered} \tag{6.9}$$

where $\mathbf{x}_t = \boldsymbol{\theta}_t$, $\mathbf{B}_t = \mathbf{G}_t$, $\mathbf{u}_t = \mathbf{C}_t = \mathbf{c}_t = \mathbf{0}$, $\mathbf{y}_t = y_t$ (i.e., $\mathbf{y}_t$ is 1 x 1), $\mathbf{Z}_t = \mathbf{F}_t^\top$, $\mathbf{a}_t = \mathbf{D}_t = \mathbf{d}_t = \mathbf{0}$, and $\mathbf{R}_t = r$ (i.e., $\mathbf{R}_t$ is 1 x 1).

6.3 Fitting a univariate DLM with `MARSS()`

Now let's go ahead and analyze the DLM specified in Equations (6.5)–(6.8). We begin by loading the data set (which is in the **MARSS** package). The data set has 3 columns for 1) the year the salmon smolts migrated to the ocean (`year`), 2) logit-transformed survival [1] (`logit.s`), and 3) the coastal upwelling index for April (`CUI.apr`). There are 42 years of data (1964–2005).

```
## load the data
data(SalmonSurvCUI)
## get time indices
years = SalmonSurvCUI[, 1]
## number of years of data
TT = length(years)
## get response data: logit(survival)
dat = matrix(SalmonSurvCUI[, 2], nrow = 1)
```

As we have seen in other case studies, standardizing our covariate(s) to have zero-mean and unit-variance can be helpful in model fitting and interpretation. In this case, it's a good idea because the variance of `CUI.apr` is orders of magnitude greater than `logit.s`.

```
## get regressor variable
CUI = SalmonSurvCUI[, 3]
## z-score the CUI
CUI.z = matrix((CUI - mean(CUI))/sqrt(var(CUI)), nrow = 1)
## number of regr params (slope + intercept)
m = dim(CUI.z)[1] + 1
```

Plots of logit-transformed survival and the z-scored April upwelling index are shown in Figure 6.1.

Next, we need to set up the appropriate matrices and vectors for MARSS. Let's begin with those for the process equation because they are straightforward.

```
## for process eqn
B = diag(m)  ## 2x2; Identity
U = matrix(0, nrow = m, ncol = 1)  ## 2x1; both elements = 0
Q = matrix(list(0), m, m)  ## 2x2; all 0 for now
diag(Q) = c("q.alpha", "q.beta")  ## 2x2; diag = (q1,q2)
```

Defining the correct form for the observation model is a little more tricky, however, because of how we model the effect(s) of explanatory variables. In a DLM, we need to use $\mathbf{Z}_t$ (instead of $\mathbf{d}_t$) as the matrix of known regressors/drivers that affect $\mathbf{y}_t$, and $\mathbf{x}_t$ (instead of $\mathbf{D}_t$) as the

[1] Survival in the original context was defined as the proportion of juveniles that survive to adulthood. Thus, we use the logit function, defined as $logit(p) = log_e(p/[1-p])$, to map survival from the open interval (0,1) onto the interval $(-\infty, \infty)$, which allows us to meet our assumption of normally distributed observation errors.

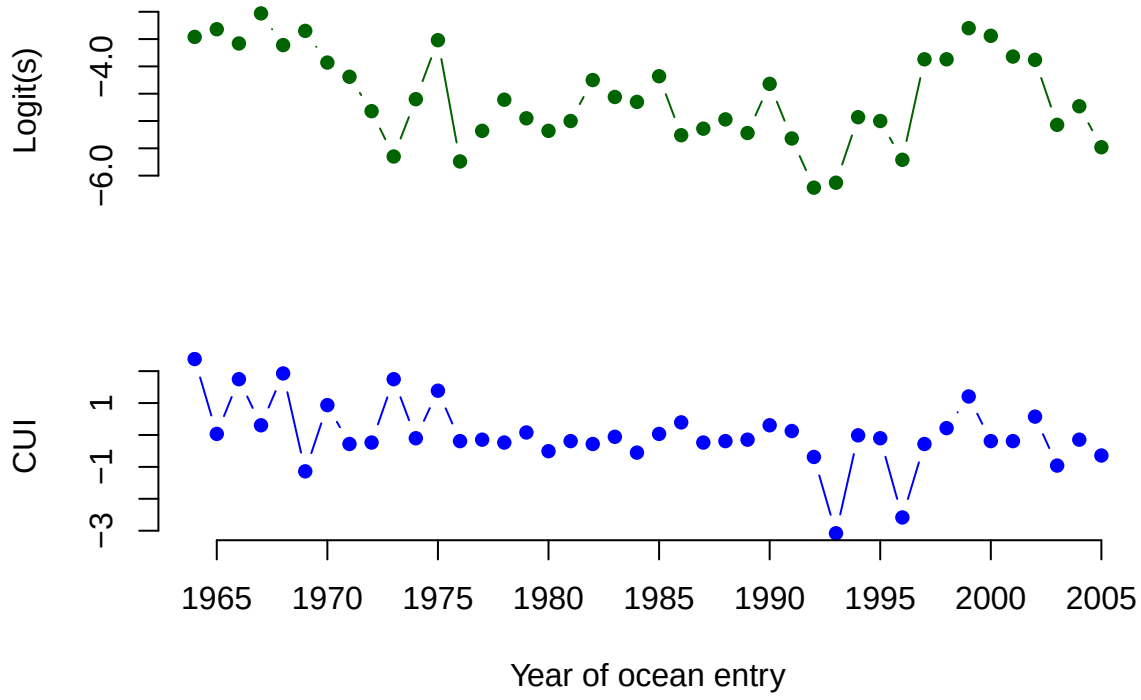


Figure 6.1: Time series of logit-transformed marine survival estimates for Snake River spring/summer Chinook salmon (top) and z-scores of the coastal upwelling index at 45N 125W (bottom). The x-axis indicates the year that the salmon smolts entered the ocean.

regression parameters. Therefore, we need to set $\mathbf{Z}_t$ equal to an n x m x T array, where n is the number of response variables (= 1; y_t is univariate), m is the number of regression parameters (= intercept + slope = 2), and T is the length of the time series (= 42).

```
## for observation eqn
Z = array(NA, c(1, m, TT))  ## NxMxT; empty for now
Z[1, 1, ] = rep(1, TT)  ## Nx1; 1's for intercept
Z[1, 2, ] = CUI.z  ## Nx1; regr variable
A = matrix(0)  ## 1x1; scalar = 0
R = matrix("r")  ## 1x1; scalar = r
```

Lastly, we need to define our lists of initial starting values and model matrices/vectors.

```
## only need starting values for regr parameters
inits.list = list(x0 = matrix(c(0, 0), nrow = m))
## list of model matrices & vectors
mod.list = list(B = B, U = U, Q = Q, Z = Z, A = A, R = R)
```

And now we can fit our DLM with MARSS.

```
## fit univariate DLM
dlm1 = MARSS(dat, inits = inits.list, model = mod.list)
```

```
Success! abstol and log-log tests passed at 115 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 115 iterations.
Log-likelihood: -40.03813
AIC: 90.07627   AICc: 91.74293

          Estimate
R.r        0.15708
Q.q.alpha  0.11264
Q.q.beta   0.00564
x0.X1     -3.34023
x0.X2     -0.05388

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

Notice that the MARSS output does not list any estimates of the regression parameters themselves. Why not? Remember that in a DLM the matrix of states ($\mathbf{x}$) contains the estimates of the regression parameters ($\boldsymbol{\theta}$). Therefore, we need to look in `dlm1$states` for the MLEs of the regression parameters, and in `dlm1$states.se` for their standard errors.

Time series of the estimated intercept and slope are shown in Figure 6.2. It appears as though the intercept is much more dynamic than the slope, as indicated by a much larger estimate of process variance for the former (`Q.q1`). In fact, although the effect of April upwelling appears to be increasing over time, it doesn't really become important as an explanatory variable until about 1990 when the approximate 95% confidence interval for the slope no longer overlaps zero.

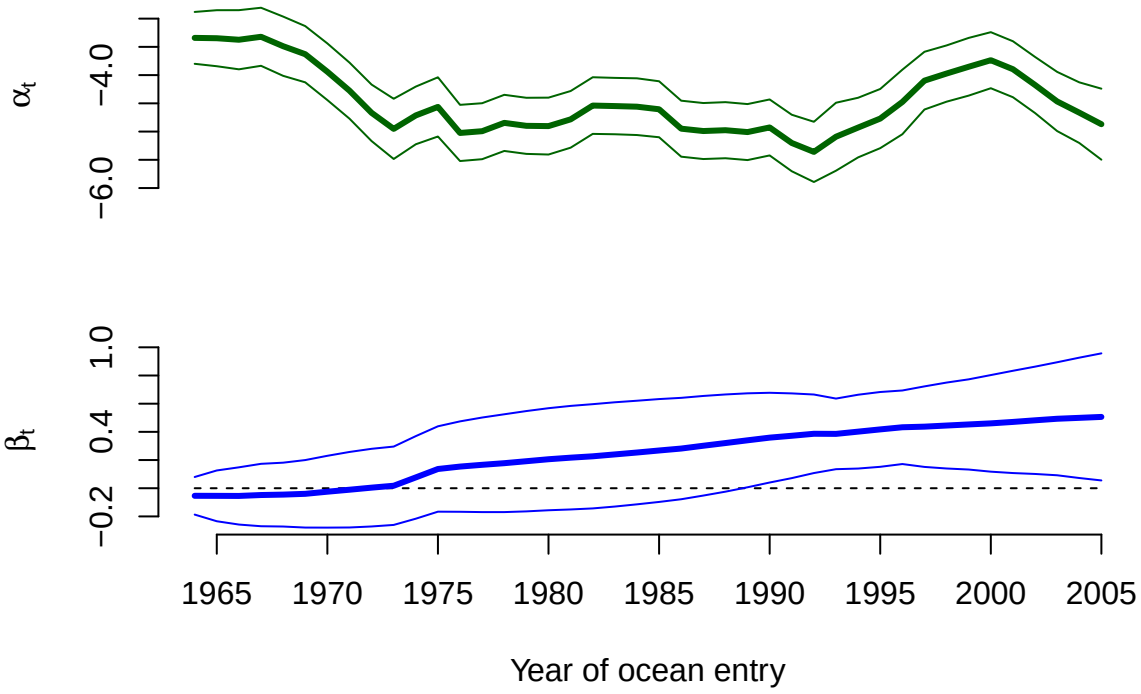


Figure 6.2: Time series of estimated mean states (thick lines) for the intercept (top) and slope (bottom) parameters from the univariate DLM specified by Equations (6.5)–(6.8). Thin lines denote the mean ± 2 standard deviations.

6.4 Forecasting with a univariate DLM

Scheuerell and Williams (2005) were interested in how well upwelling could be used to actually *forecast* expected survival of salmon, so let's look at how well our model does in that context. To do so, we need the predictive distributions for the regression parameters and observation.

Beginning with our definition for the distribution of the parameters at time $t = 0$, $\boldsymbol{\theta}_0 \sim \text{MVN}(\boldsymbol{\pi}_0, \boldsymbol{\Lambda}_0)$ in Equation (6.8), we write

$$\boldsymbol{\theta}_{t-1}|y_{1:t-1} \sim \text{MVN}(\boldsymbol{\pi}_{t-1}, \boldsymbol{\Lambda}_{t-1}) \tag{6.10}$$

to indicate the distribution of $\boldsymbol{\theta}$ at time $t-1$ conditioned on the observed data through time $t-1$ (i.e., $y_{1:t-1}$). Then, we can write the one-step ahead predictive distribution for $\boldsymbol{\theta}_t$ given $y_{1:t-1}$ as

$$\begin{aligned} \boldsymbol{\theta}_t | y_{1:t-1} &\sim \text{MVN}(\boldsymbol{\eta}_t, \boldsymbol{\Phi}_t), \text{ where} \\ \boldsymbol{\eta}_t &= \mathbf{G}_t \boldsymbol{\pi}_{t-1}, \text{ and} \\ \boldsymbol{\Phi}_t &= \mathbf{G}_t \boldsymbol{\Lambda}_{t-1} \mathbf{G}_t^\top + \mathbf{Q}. \end{aligned} \tag{6.11}$$

Consequently, the one-step ahead predictive distribution for the observation at time t given $y_{1:t-1}$ is

$$\begin{aligned} y_t | y_{1:t-1} &\sim \text{N}(\zeta_t, \Psi_t), \text{ where} \\ \zeta_t &= \mathbf{F}_t \boldsymbol{\eta}_t, \text{ and} \\ \Psi_t &= \mathbf{F}_t \Phi_t \mathbf{F}_t^\top + \mathbf{R}. \end{aligned} \tag{6.12}$$

6.4.1 Forecasting a univariate DLM with MARSS

Working from Equation (6.12), we can now use `MARSS()` to compute the expected value of the forecast at time t ($\text{E}[y_t | y_{1:t-1}] = \zeta_t$), and its variance ($\text{var}[y_t | y_{1:t-1}] = \Psi_t$). For the expectation, we need $\mathbf{F}_t \boldsymbol{\eta}_t$. Recall that $\mathbf{F}_t$ is our $1 \times m$ matrix of explanatory variables at time t ($\mathbf{F}_t$ is called $\mathbf{Z}_t$ in MARSS notation). The one-step ahead forecasts of the parameters at time t ($\boldsymbol{\eta}_t$) are calculated as part of the Kalman filter algorithm—they are termed $\tilde{x}_t^{t-1}$ in `MARSS()` notation and stored as `xtt1` in the list produced by the `MARSSkfss()` function.

```
## get list of Kalman filter output
kf.out = MARSSkfss(dlm1)
## forecasts of regr parameters; 2xT matrix
eta = kf.out$xtt1
## ts of E(forecasts)
fore.mean = vector()
for (t in 1:TT) {
    fore.mean[t] = Z[, , t] %*% eta[, t, drop = F]
}
```

For the variance of the forecasts, we need $\mathbf{F}_t \Phi_t \mathbf{F}_t^\top + \mathbf{R}$. As with the mean, $\mathbf{F}_t \equiv \mathbf{Z}_t$. The variances of the one-step ahead forecasts of the parameters at time t (Φ_t) are also calculated as part of the Kalman filter algorithm—they are stored as `Vtt1` in the list produced by the `MARSSkfss()` function. Lastly, the observation variance $\mathbf{R}$ is part of the standard MARSS output.

```
## variance of regr parameters; 1x2xT array
Phi = kf.out$Vtt1
## obs variance; 1x1 matrix
R.est = coef(dlm1, type = "matrix")$R
## ts of Var(forecasts)
fore.var = vector()
for (t in 1:TT) {
    tZ = matrix(Z[, , t], m, 1)  ## transpose of Z
    fore.var[t] = Z[, , t] %*% Phi[, , t] %*% tZ + R.est
}
```

Plots of the model mean forecasts with their estimated uncertainty are shown in Figure 6.3. Nearly all of the observed values fell within the approximate prediction interval. Notice that we have a forecasted value for the first year of the time series (1964), which may seem at odds with our notion of forecasting at time t based on data available only through time $t-1$. In this case, however, MARSS is actually estimating the states at $t = 0$ ($\boldsymbol{\theta}_0$), which allows us to compute a forecast for the first time point.

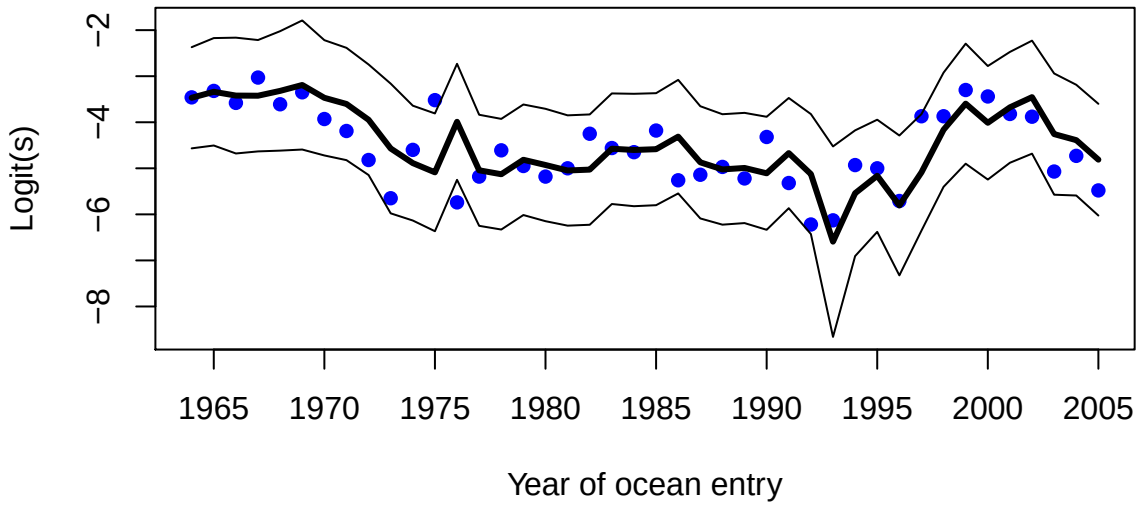


Figure 6.3: Time series of logit-transformed survival data (blue dots) and model mean forecasts (thick line). Thin lines denote the approximate 95% prediction intervals.

Although our model forecasts look reasonable in logit-space, it is worthwhile to examine how well they look when the survival data and forecasts are back-transformed onto the interval [0,1] (Figure 6.4). In that case, the accuracy does not seem to be affected, but the precision appears much worse, especially during the early and late portions of the time series when survival is changing rapidly.

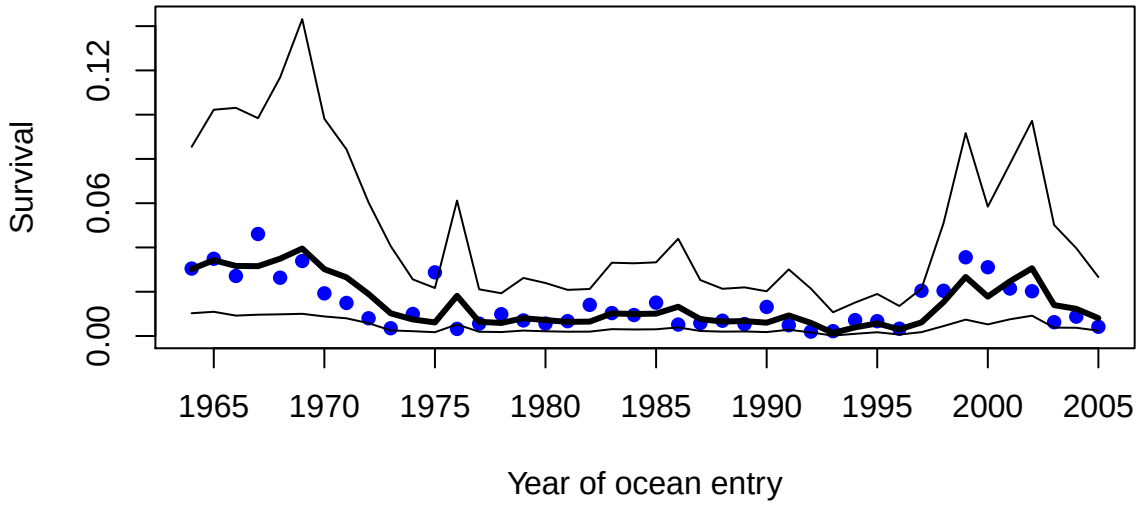


Figure 6.4: Time series of survival data (blue dots) and model mean forecasts (thick line). Thin lines denote the approximate 95% prediction intervals.

6.4.2 DLM forecast diagnostics

As with other time series models, evaluation of a DLM should include some model diagnostics. In a forecasting context, we are often interested in the forecast errors, which are simply the observed data minus the forecasts ($e_t = y_t - \zeta_t$). In particular, the following assumptions should hold true for e_t:

1. $e_t \sim \text{N}(0, \sigma^2)$; 2. $\text{cov}(e_t, e_{t-k}) = 0$.

In the literature on state-space models, the set of e_t are commonly referred to as "innovations". `MARSS()` calculates the innovations as part of the Kalman filter algorithm—they are stored as `Innov` in the list produced by the `MARSSkfss()` function.

```
## forecast errors
innov = kf.out$Innov
```

Let's see if our innovations meet the model assumptions. Beginning with (1), we can use a Q-Q plot to see whether the innovations are normally distributed with a mean of zero. We'll use the `qqnorm()` function to plot the quantiles of the innovations on the y-axis versus the theoretical quantiles from a Normal distribution on the x-axis. If the 2 distributions are similar, the points should fall on the line defined by $y = x$.

```
## Q-Q plot of innovations
qqnorm(t(innov), main = "", pch = 16, col = "blue")
## add y=x line for easier interpretation
qqline(t(innov))
```

The Q-Q plot (Figure 6.5) indicates that the innovations appear to be more-or-less normally

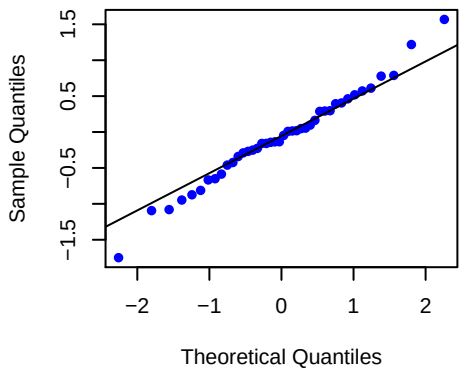


Figure 6.5: Q-Q plot of the forecast errors (innovations) for the DLM specified in Equations (6.5)–(6.8).

distributed (i.e., most points fall on the line). Furthermore, it looks like the mean of the innovations is about 0, but we should use a more reliable test than simple visual inspection. We can formally test whether the mean of the innovations is significantly different from 0 by using a one-sample t-test. based on a null hypothesis of $\mathrm{E}(e_t) = 0$. To do so, we will use the function `t.test()` and base our inference on a significance value of $\alpha = 0.05$.

```
## p-value for t-test of H0: E(innov) = 0
t.test(t(innov), mu = 0)$p.value
```

```
[1] 0.4840901
```

The p-value $>> 0.05$ so we cannot reject the null hypothesis that $\mathrm{E}(e_t) = 0$.

Moving on to assumption (2), we can use the sample autocorrelation function (ACF) to examine whether the innovations covary with a time-lagged version of themselves. Using the `acf()` function, we can compute and plot the correlations of e_t and e_{t-k} for various values of k. Assumption (2) will be met if none of the correlation coefficients exceed the 95% confidence intervals defined by $\pm z_{0.975}/\sqrt{n}$.

```
## plot ACF of innovations
acf(t(innov), lag.max = 10)
```

The ACF plot (Figure 6.6) shows no significant autocorrelation in the innovations at lags 1–10, so it looks like both of our model assumptions have indeed been met.

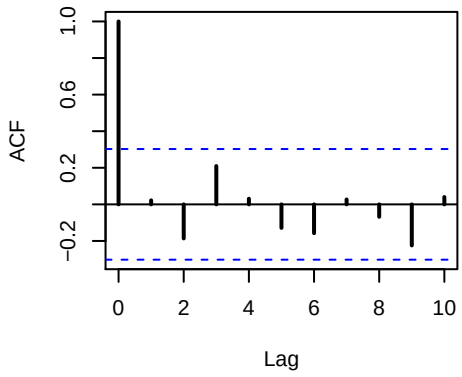


Figure 6.6: Autocorrelation plot of the forecast errors (innovations) for the DLM specified in Equations (6.5)–(6.8). Horizontal blue lines define the upper and lower 95% confidence intervals.

6.5 Homework discussion and data

For the homework this week we will use a DLM to examine some of the time-varying properties of the spawner-recruit relationship for Pacific salmon. Much work has been done on this topic, particularly by Randall Peterman and his students and post-docs at Simon Fraser University. To do so, researchers commonly use a Ricker model because of its relatively simple form, such that the number of recruits (offspring) born in year t (R_t) from the number of spawners (parents) (S_t) is

$$R_t = aS_t e^{-bS+v_t}. \tag{6.13}$$

The parameter a determines the maximum reproductive rate in the absence of any density-dependent effects (the slope of the curve at the origin), b is the strength of density dependence, and $v_t \sim N(0, \sigma)$. In practice, the model is typically log-transformed so as to make it linear with respect to the predictor variable S_t, such that

$$\begin{aligned} \log(R_t) &= \log(a) + \log(S_t) - bS_t + v_t \\ \log(R_t) - \log(S_t) &= \log(a) - bS_t + v_t \\ \log(R_t/S_t) &= \log(a) - bS_t + v_t. \end{aligned} \tag{6.14}$$

Substituting $y_t = \log(R_t/S_t)$, $x_t = S_t$, and $\alpha = \log(a)$ yields a simple linear regression model with intercept α and slope b.

Unfortunately, however, residuals from this simple model typically show high-autocorrelation due to common environmental conditions that affect overlaping generations. Therefore, to correct for this and allow for an index of stock productivity that controls for any density-dependent effects, the model may be re-witten as

$$\begin{aligned} \log(R_t/S_t) &= \alpha_t - bS_t + v_t, \\ \alpha_t &= \alpha_{t-1} + w_t, \end{aligned} \tag{6.15}$$

and $w_t \sim N(0, q)$. By treating the brood-year specific productivity as a random walk, we allow it to vary, but in an autocorrelated manner so that consecutive years are not independent from one another.

More recently, interest has grown in using covariates (*e.g.*, sea-surface temperature) to explain the interannual variability in productivity. In that case, we can can write the model as

$$\log(R_t/S_t) = \alpha + \delta_t X_t - bS_t + v_t. \tag{6.16}$$

In this case we are estimating some base-level productivity (α) plus the time-varying effect of some covariate X_t (δ_t).

6.5.1 Spawner-recruit data

The data come from a large public database begun by Ransom Myers many years ago. If you are interested, you can find lots of time series of spawning-stock, recruitment, and harvest for a variety of fishes around the globe. Here is the website:
http://ram.biology.dal.ca/~myers/about_site.html

For this exercise, we will use spawner-recruit data for sockeye salmon (*Oncorhynchus nerka*) from the Kvichak River in SW Alaska that span the years 1952-1989. In addition, we'll examine the potential effects of the Pacific Decadal Oscillation (PDO) during the salmon's first year in the ocean, which is widely believed to be a "bottleneck" to survival.

Download the Rdata file for Kvichak River from here: KvichakSockeye.RData. Then load the Rdata file:

```
load("KvichakSockeye.RData")
```

The data are a dataframe with columns for brood year (`brood.yr`), number of spawners (`Sp`), number of recruits (`Rec`) and PDO at year $t-2$ (`PDO.t2`) and $t-3$ (`PDO.t3`).

```
## head of data file
head(SRdata)
```

```
  brood.yr   Sp   Rec PDO.t2 PDO.t3
1     1952 5970 17310  -0.61  -0.61
2     1953  320   520  -1.48  -2.66
3     1954  240   750  -2.05  -1.26
4     1955  250  1280   0.01   0.11
5     1956 9443 39036   0.86   0.37
6     1957 2843  4091  -0.25   0.29
```

6.6 Problems

Use the information and data in the previous section to answer the following questions. Note that if any model is not converging, then you will need to increase the `maxit` parameter in the `control` argument/list that gets passed to `MARSS()`. For example, you might try `control=list(maxit=2000)`.

1. Begin by fitting a reduced form of Equation (6.15) that includes only a time-varying level (α_t) and observation error (v_t). That is,

$$\begin{aligned} \log(R_t) &= \alpha_t + \log(S_t) + v_t \\ \log(R_t/S_t) &= \alpha_t + v_t \end{aligned}$$

This model assumes no density-dependent survival in that the number of recruits is an ascending function of spawners. Plot the ts of α_t and note the AICc for this model. Also plot appropriate model diagnostics.

2. Fit the full model specified by Equation (6.15). For this model, obtain the time series of α_t, which is an estimate of the stock productivity in the absence of density-dependent effects. How do these estimates of productivity compare to those from the previous question? Plot the ts of α_t and note the AICc for this model. Also plot appropriate model diagnostics. (*Hint*: If you don't want a parameter to vary with time, what does that say about its process variance?)

3. Fit the model specified by Equation (6.16) with the summer PDO index as the covariate (`PDO.t2`). What is the mean level of productivity? Plot the ts of δ_t and note the AICc for this model. Also plot appropriate model diagnostics.

4. Fit the model specified by Equation (6.16) with the winter PDO index as the covariate (`PDO.t3`). What is the mean level of productivity? Plot the ts of δ_t and note the AICc for this model. Also plot appropriate model diagnostics.

5. Based on AICc, which of the models above is the most parsimonius? Is it well behaved (*i.e.*, are the model assumptions met)? Plot the model forecasts for the best model. Is this a good forecast model?

Chapter 7

Dynamic Factor Analysis

Here we will use the **MARSS** package to do Dynamic Factor Analysis (DFA), which allows us to look for a set of common underlying processes among a relatively large set of time series (Zuur et al., 2003). There have been a number of recent applications of DFA to ecological questions surrounding Pacific salmon (Stachura et al., 2014; Jorgensen et al., 2016; Ohlberger et al., 2016) and stream temperatures (Lisi et al., 2015). For a more in-depth treatment of potential applications of MARSS models for DFA, see Chapter 9 in the MARSS User's Guide.

A script with all the R code in the chapter can be downloaded here.

Data and packages

All the data used in the chapter are in the **MARSS** package. Install the package, if needed, and load to run the code in the chapter.

```
library(MARSS)
```

7.1 Introduction

DFA is conceptually different than what we have been doing in the previous applications. Here we are trying to explain temporal variation in a set of n observed time series using linear combinations of a set of m hidden random walks, where $m << n$. A DFA model is a type of MARSS model with the following structure:

$$\begin{aligned} \mathbf{x}_t &= \mathbf{x}_{t-1} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \\ \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}) \end{aligned} \tag{7.1}$$

This equation should look rather familiar as it is exactly the same form we used for estimating varying number of processes from a set of observations in Lesson II. The difference with DFA

is that rather than fixing the elements within $\mathbf{Z}$ at 1 or 0 to indicate whether an observation does or does not correspond to a trend, we will instead estimate them as "loadings" on each of the states/processes.

7.2 Example of a DFA model

The general idea is that the observations $\mathbf{y}$ are modeled as a linear combination of hidden processes $\mathbf{x}$ and factor loadings $\mathbf{Z}$ plus some offsets $\mathbf{a}$. Imagine a case where we had a data set with five observed time series ($n = 5$) and we want to fit a model with three hidden processes ($m = 3$). If we write out our DFA model in MARSS matrix form, the process model would look like this:

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}_t \tag{7.2}$$

And the observation equation would look like

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}_t = \begin{bmatrix} z_{11} & z_{12} & z_{13} \\ z_{21} & z_{22} & z_{23} \\ z_{31} & z_{32} & z_{33} \\ z_{41} & z_{42} & z_{43} \\ z_{51} & z_{52} & z_{53} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_t + \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}_t . \tag{7.3}$$

The process errors would be

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} q_{11} & q_{12} & q_{13} \\ q_{12} & q_{22} & q_{23} \\ q_{13} & q_{23} & q_{33} \end{bmatrix} \right). \tag{7.4}$$

And the observation errors would be

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & r_{15} \\ r_{12} & r_{22} & r_{23} & r_{24} & r_{25} \\ r_{13} & r_{23} & r_{33} & r_{34} & r_{35} \\ r_{14} & r_{24} & r_{34} & r_{44} & r_{45} \\ r_{15} & r_{25} & r_{35} & r_{45} & r_{55} \end{bmatrix} \right) \tag{7.5}$$

7.3 Constraining a DFA model

If $\mathbf{Z}$, $\mathbf{a}$, and $\mathbf{Q}$ are not constrained, however, then the DFA model above is unidentifiable. Nevertheless, we can use the following parameter constraints to make the model identifiable:

- in the first $m-1$ rows of $\mathbf{Z}$, the z-value in the j-th column and i-th row is set to zero if $j > i$;
- $\mathbf{a}$ is constrained so that the first m values are set to zero; and
- $\mathbf{Q}$ is set equal to the identity matrix $\mathbf{I}_m$.

Using these constraints, the process equation for the DFA model above becomes

$$\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_t = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_{t-1} + \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}_t \tag{7.6}$$

and the observation equation becomes

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}_t = \begin{bmatrix} z_{11} & 0 & 0 \\ z_{21} & z_{22} & 0 \\ z_{31} & z_{32} & z_{33} \\ z_{41} & z_{42} & z_{43} \\ z_{51} & z_{52} & z_{53} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}_t + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}_t . \tag{7.7}$$

The process errors would then become

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \right), \tag{7.8}$$

but the observation errors would stay the same, such that

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{bmatrix}_t \sim \text{MVN}\left(\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \begin{bmatrix} r_{11} & r_{12} & r_{13} & r_{14} & r_{15} \\ r_{12} & r_{22} & r_{23} & r_{24} & r_{25} \\ r_{13} & r_{23} & r_{33} & r_{34} & r_{35} \\ r_{14} & r_{24} & r_{34} & r_{44} & r_{45} \\ r_{15} & r_{25} & r_{35} & r_{45} & r_{55} \end{bmatrix} \right). \tag{7.9}$$

7.4 Different error structures

The example observation equation we used above had what we refer to as an "unconstrained" variance-covariance matrix $\mathbf{R}$ wherein all of the parameters are unique. In certain applica-

tions, however, we may want to change our assumptions about the forms for $\mathbf{R}$. For example, we might have good reason to believe that all of the observations have different error variances and they were independent of one another (e.g., different methods were used for sampling), in which case

$$\mathbf{R} = \begin{bmatrix} r_1 & 0 & 0 & 0 & 0 \\ 0 & r_2 & 0 & 0 & 0 \\ 0 & 0 & r_3 & 0 & 0 \\ 0 & 0 & 0 & r_4 & 0 \\ 0 & 0 & 0 & 0 & r_5 \end{bmatrix}.$$

Alternatively, we might have a situation where all of the observation errors had the same variance r, but they were not independent from one another. In that case we would have to include a covariance parameter k, such that

$$\mathbf{R} = \begin{bmatrix} r & k & k & k & k \\ k & r & k & k & k \\ k & k & r & k & k \\ k & k & k & r & k \\ k & k & k & k & r \end{bmatrix}.$$

Any of these options for $\mathbf{R}$ (and other custom options as well) are available to us in a DFA model, just as they were in the MARSS models used in previous chapters.

7.5 Lake Washington phytoplankton data

For this exercise, we will use the Lake Washington phytoplankton data contained in the **MARSS** package. Let's begin by reading in the monthly values for all of the data, including metabolism, chemistry, and climate.

```
## load the data (there are 3 datasets contained here)
data(lakeWAplankton)
## we want lakeWAplanktonTrans, which has been transformed so
## the 0s are replaced with NAs and the data z-scored
all_dat <- lakeWAplanktonTrans
## use only the 10 years from 1980-1989
yr_frst <- 1980
yr_last <- 1989
plank_dat <- all_dat[all_dat[, "Year"] >= yr_frst & all_dat[,
    "Year"] <= yr_last, ]
## create vector of phytoplankton group names
phytoplankton <- c("Cryptomonas", "Diatoms", "Greens", "Unicells",
    "Other.algae")
```

```
## get only the phytoplankton
dat_1980 <- plank_dat[, phytoplankton]
```

Next, we transpose the data matrix and calculate the number of time series and their length.

```
## transpose data so time goes across columns
dat_1980 <- t(dat_1980)
## get number of time series
N_ts <- dim(dat_1980)[1]
## get length of time series
TT <- dim(dat_1980)[2]
```

It will be easier to estimate the real parameters of interest if we de-mean the data, so let's do that.

```
y_bar <- apply(dat_1980, 1, mean, na.rm = TRUE)
dat <- dat_1980 - y_bar
rownames(dat) <- rownames(dat_1980)
```

7.5.1 Plots of the data

Here are time series plots of all five phytoplankton functional groups.

```
spp <- rownames(dat_1980)
clr <- c("brown", "blue", "darkgreen", "darkred", "purple")
cnt <- 1
par(mfrow = c(N_ts, 1), mai = c(0.5, 0.7, 0.1, 0.1), omi = c(0,
    0, 0, 0))
for (i in spp) {
    plot(dat[i, ], xlab = "", ylab = "Abundance index", bty = "L",
        xaxt = "n", pch = 16, col = clr[cnt], type = "b")
    axis(1, 12 * (0:dim(dat_1980)[2]) + 1, yr_frst + 0:dim(dat_1980)[2])
    title(i)
    cnt <- cnt + 1
}
```

7.6 Fitting DFA models with the MARSS package

The **MARSS** package is designed to work with the fully specified matrix form of the multivariate state-space model we wrote out in Sec 3. Thus, we will need to create a model list with forms for each of the vectors and matrices. Note that even though some of the model elements are scalars and vectors, we will need to specify everything as a matrix (or array for time series of matrices).

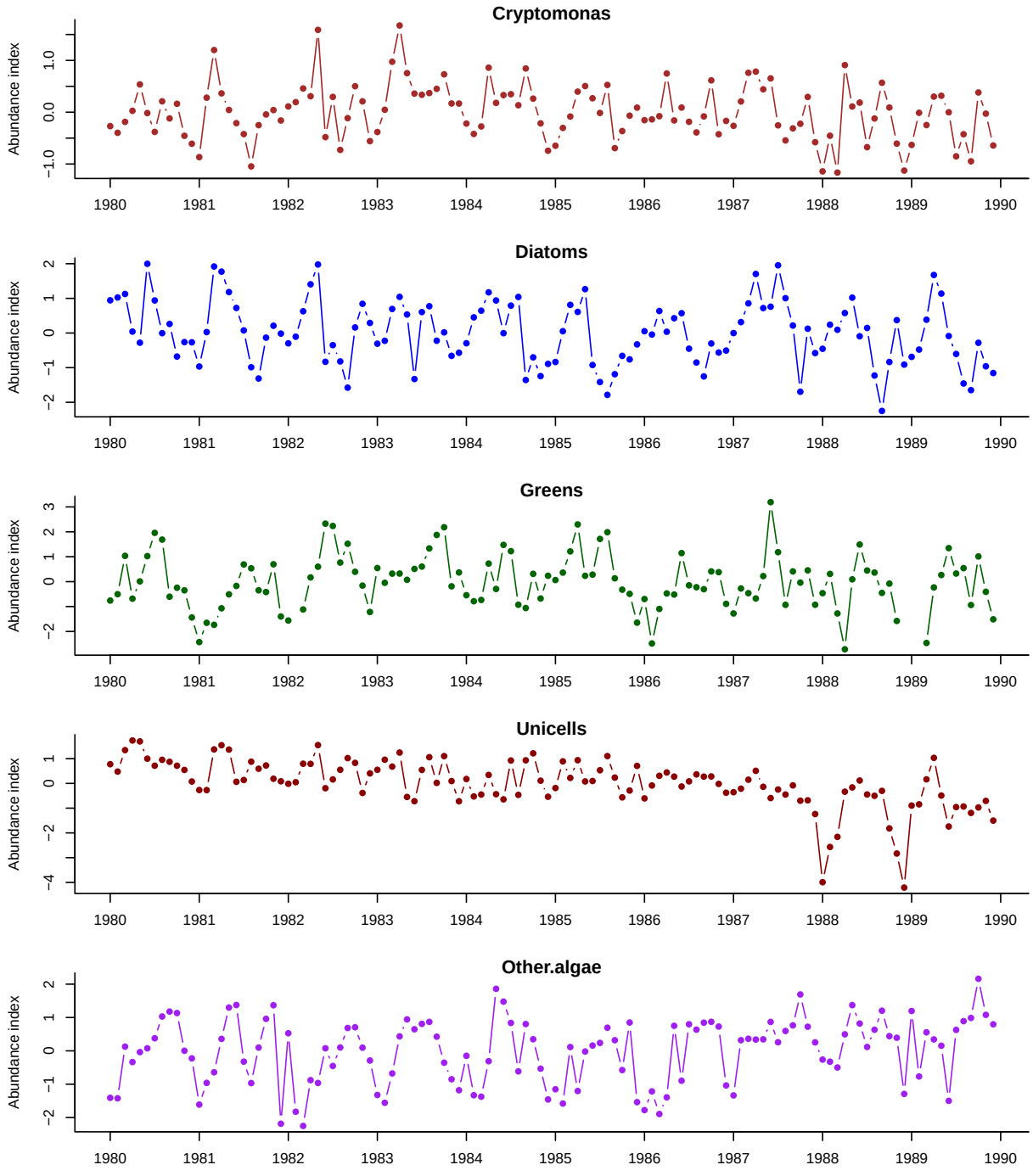


Figure 7.1: Demeaned time series of Lake Washington phytoplankton.

Notice that the code below uses some of the **MARSS** shortcuts for specifying forms of vectors and matrices. We will also use the `matrix(list(),nrow,ncol)` trick we learned previously.

7.6.1 The process model

We need to specify the explicit form for all of the vectors and matrices in the full form of the MARSS model we defined in Sec 3.1. Note that we do not have to specify anything for the states ($\mathbf{x}$) – those are elements that `MARSS` will identify and estimate itself based on our definitions of the other vectors and matrices.

```
## number of processes
mm <- 3
## 'BB' is identity: 1's along the diagonal & 0's elsewhere
BB <- "identity"  # diag(mm)
## 'uu' is a column vector of 0's
uu <- "zero"  # matrix(0,mm,1)
## 'CC' and 'cc' are for covariates
CC <- "zero"  # matrix(0,mm,1)
cc <- "zero"  # matrix(0,1,wk_last)
## 'QQ' is identity
QQ <- "identity"  # diag(mm)
```

7.6.2 The observation model

Here we will fit the DFA model above where we have `R N_ts` observed time series and we want 3 hidden states. Now we need to set up the observation model for `MARSS`. Here are the vectors and matrices for our first model where each nutrient follows its own process. Recall that we will need to set the elements in the upper R corner of $\mathbf{Z}$ to 0. We will assume that the observation errors have different variances and they are independent of one another.

```
## 'ZZ' is loadings matrix
Z_vals <- list("z11", 0, 0, "z21", "z22", 0, "z31", "z32", "z33",
    "z41", "z42", "z43", "z51", "z52", "z53")
ZZ <- matrix(Z_vals, nrow = N_ts, ncol = 3, byrow = TRUE)
ZZ
```

```
     [,1]  [,2]  [,3]
[1,] "z11" 0     0
[2,] "z21" "z22" 0
[3,] "z31" "z32" "z33"
[4,] "z41" "z42" "z43"
[5,] "z51" "z52" "z53"
```

```
## 'aa' is the offset/scaling
aa <- "zero"
## 'DD' and 'd' are for covariates
DD <- "zero"  # matrix(0,mm,1)
dd <- "zero"  # matrix(0,1,wk_last)
## 'RR' is var-cov matrix for obs errors
RR <- "diagonal and unequal"
```

7.6.3 Fit the model in MARSS

Now it's time to fit our first DLM. To do so, we need to create a three lists that we will need to pass to the `MARSS()` function:

1. A list of specifications for the model's vectors and matrices;
2. A list of any initial values – `MARSS` will pick its own otherwise;
3. A list of control parameters for the `MARSS()` function.

```
## list with specifications for model vectors/matrices
mod_list <- list(B = BB, U = uu, C = CC, c = cc, Q = QQ, Z = ZZ,
    A = aa, D = DD, d = dd, R = RR)
## list with model inits
init_list <- list(x0 = matrix(rep(0, mm), mm, 1))
## list with model control parameters
con_list <- list(maxit = 3000, allow.degen = TRUE)
```

Now we can fit the model.

```
## fit MARSS
dfa_1 <- MARSS(y = dat, model = mod_list, inits = init_list,
    control = con_list)
```

```
Success! abstol and log-log tests passed at 246 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 246 iterations.
Log-likelihood: -692.9795
AIC: 1425.959   AICc: 1427.42

                          Estimate
Z.z11                       0.2738
Z.z21                       0.4487
```

```
Z.z31                          0.3170
Z.z41                          0.4107
Z.z51                          0.2553
Z.z22                          0.3608
Z.z32                         -0.3690
Z.z42                         -0.0990
Z.z52                         -0.3793
Z.z33                          0.0185
Z.z43                         -0.1404
Z.z53                          0.1317
R.(Cryptomonas,Cryptomonas)    0.1638
R.(Diatoms,Diatoms)            0.2913
R.(Greens,Greens)              0.8621
R.(Unicells,Unicells)          0.3080
R.(Other.algae,Other.algae)    0.5000
x0.X1                          0.2218
x0.X2                          1.8155
x0.X3                         -4.8097

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

7.7 Interpreting the MARSS output

By now the `MARSS()` output should look familiar. The first 12 parameter estimates `Z.z##` are the loadings of each observed time series on the 3 hidden states. The next 5 estimates `R.(,)` are the variances of the observation errors ($v_{i,t}$). The last 3 values, `x0.X#`, are the estimates of the initial states at $t = 0$.

Recall that the estimates of the processes themselves (i.e., $\mathbf{x}$) are contained in one of the list elements in our fitted `MARSS` object. Specifically, they are in `mod_fit$states`, and their respective standard errors are in `mod_fit$states.se`. For the names of all of the other objects, type `names(dfa_1)`.

7.8 Rotating trends and loadings

Before proceeding further, we need to address the constraints we placed on the DFA model in Sec 2.2. In particular, we arbitrarily constrained $\mathbf{Z}$ in such a way to choose only one of these solutions, but fortunately the different solutions are equivalent, and they can be related to each other by a rotation matrix $\mathbf{H}$. Let $\mathbf{H}$ be any $m \times m$ non-singular matrix. The following are then equivalent DFA models:

$$\begin{aligned} \mathbf{x}_t &= \mathbf{x}_{t-1} + \mathbf{w}_t \\ \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \end{aligned} \tag{7.10}$$

and

$$\begin{aligned} \mathbf{H}\mathbf{x}_t &= \mathbf{H}\mathbf{x}_{t-1} + \mathbf{H}\mathbf{w}_t \\ \mathbf{y}_t &= \mathbf{Z}\mathbf{H}^{-1}\mathbf{x}_t + \mathbf{a} + \mathbf{v}_t \end{aligned}. \tag{7.11}$$

There are many ways of doing factor rotations, but a common method is the "varimax"" rotation, which seeks a rotation matrix $\mathbf{H}$ that creates the largest difference between the loadings in $\mathbf{Z}$. For example, imagine that row 3 in our estimated $\mathbf{Z}$ matrix was (0.2, 0.2, 0.2). That would mean that green algae were a mixture of equal parts of processes 1, 2, and 3. If instead row 3 was (0.8, 0.1, 0.05), this would make our interpretation of the model fits easier because we could say that green algae followed the first process most closely. The varimax rotation would find the $\mathbf{H}$ matrix that makes the rows in $\mathbf{Z}$ more like (0.8, 0.1, 0.05) and less like (0.2, 0.2, 0.2).

The varimax rotation is easy to compute because R has a built in function for this: `varimax()`. Interestingly, the function returns the inverse of $\mathbf{H}$, which we need anyway.

```
## get the estimated ZZ
Z_est <- coef(dfa_1, type = "matrix")$Z
## get the inverse of the rotation matrix
H_inv <- varimax(Z_est)$rotmat
```

We can now rotate both $\mathbf{Z}$ and $\mathbf{x}$.

```
## rotate factor loadings
Z_rot = Z_est %*% H_inv
## rotate processes
proc_rot = solve(H_inv) %*% dfa_1$states
```

7.9 Estimated states and loadings

Here are plots of the three hidden processes (left column) and the loadings for each of phytoplankton groups (right column).

```
ylbl <- phytoplankton
w_ts <- seq(dim(dat)[2])
layout(matrix(c(1, 2, 3, 4, 5, 6), mm, 2), widths = c(2, 1))
## par(mfcol=c(mm,2), mai=c(0.5,0.5,0.5,0.1), omi=c(0,0,0,0))
par(mai = c(0.5, 0.5, 0.5, 0.1), omi = c(0, 0, 0, 0))
## plot the processes
for (i in 1:mm) {
```

```
    ylm <- c(-1, 1) * max(abs(proc_rot[i, ]))
    ## set up plot area
    plot(w_ts, proc_rot[i, ], type = "n", bty = "L", ylim = ylm,
        xlab = "", ylab = "", xaxt = "n")
    ## draw zero-line
    abline(h = 0, col = "gray")
    ## plot trend line
    lines(w_ts, proc_rot[i, ], lwd = 2)
    lines(w_ts, proc_rot[i, ], lwd = 2)
    ## add panel labels
    mtext(paste("State", i), side = 3, line = 0.5)
    axis(1, 12 * (0:dim(dat_1980)[2]) + 1, yr_frst + 0:dim(dat_1980)[2])
}
## plot the loadings
minZ <- 0
ylm <- c(-1, 1) * max(abs(Z_rot))
for (i in 1:mm) {
    plot(c(1:N_ts)[abs(Z_rot[, i]) > minZ], as.vector(Z_rot[abs(Z_rot[,
        i]) > minZ, i]), type = "h", lwd = 2, xlab = "", ylab = "",
        xaxt = "n", ylim = ylm, xlim = c(0.5, N_ts + 0.5), col = clr)
    for (j in 1:N_ts) {
        if (Z_rot[j, i] > minZ) {
            text(j, -0.03, ylbl[j], srt = 90, adj = 1, cex = 1.2,
                col = clr[j])
        }
        if (Z_rot[j, i] < -minZ) {
            text(j, 0.03, ylbl[j], srt = 90, adj = 0, cex = 1.2,
                col = clr[j])
        }
        abline(h = 0, lwd = 1.5, col = "gray")
    }
    mtext(paste("Factor loadings on state", i), side = 3, line = 0.5)
}
```

It looks like there are strong seasonal cycles in the data, but there is some indication of a phase difference between some of the groups. We can use `ccf()` to investigate further.

```
par(mai = c(0.9, 0.9, 0.1, 0.1))
ccf(proc_rot[1, ], proc_rot[2, ], lag.max = 12, main = "")
```

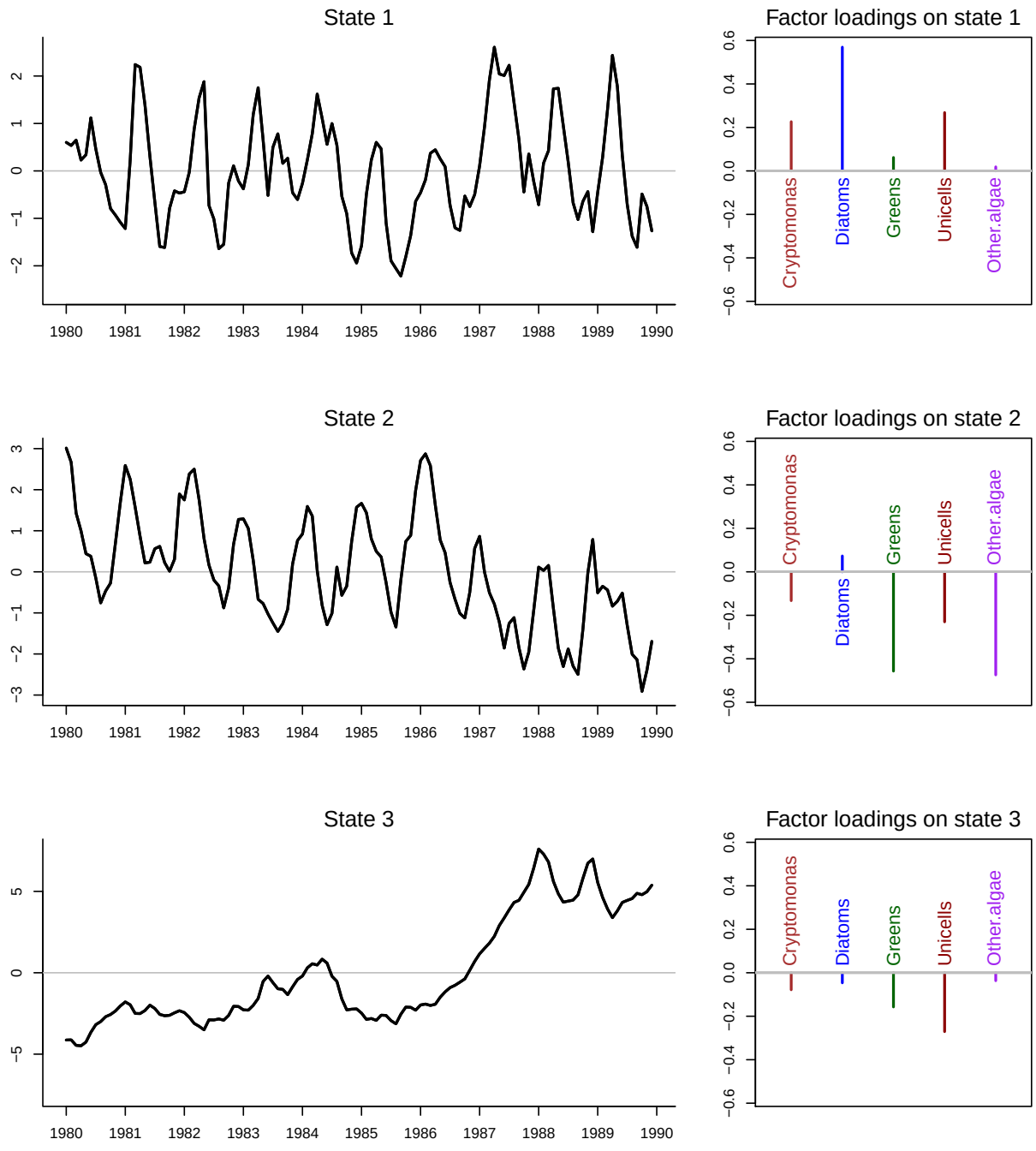


Figure 7.2: Estimated states from the DFA model.

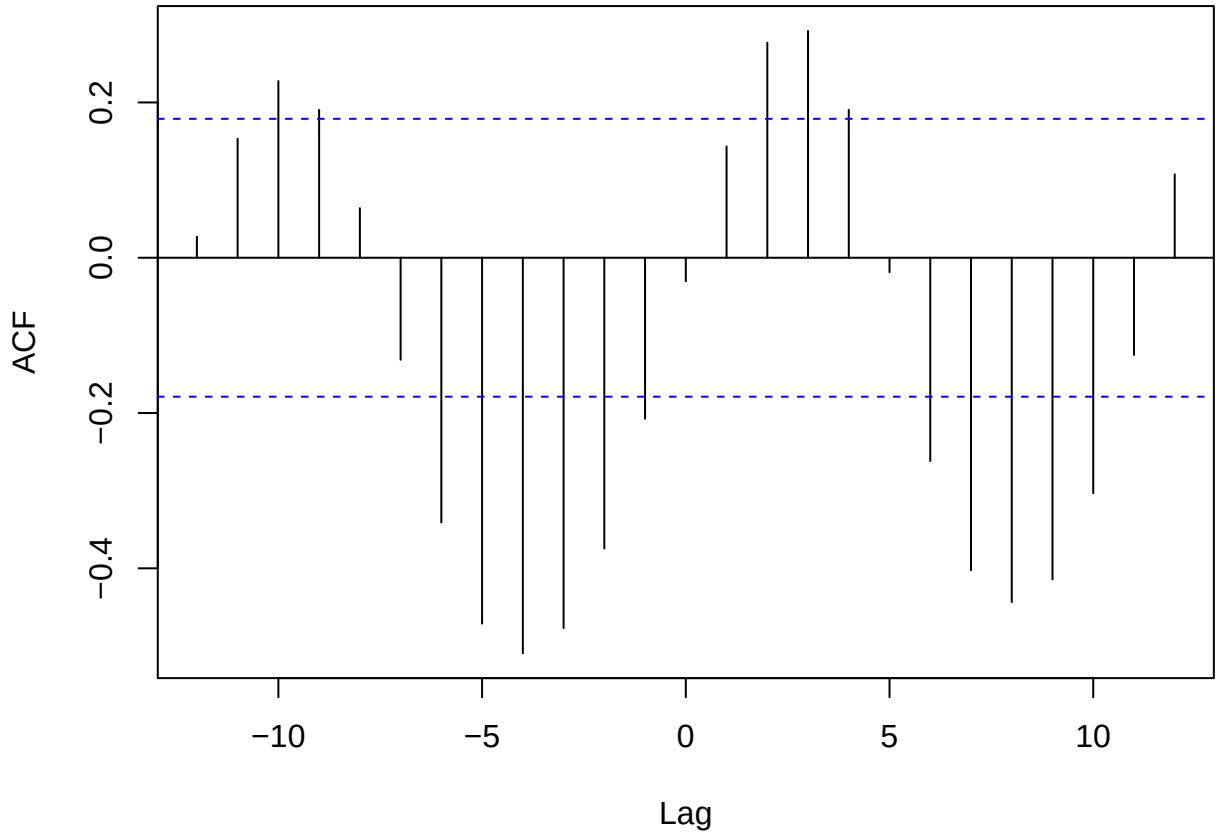


Figure 7.3: Cross-correlation plot of the two rotations.

7.10 Plotting the data and model fits

We can plot the fits for our DFA model along with the data. The following function will return the fitted values ± (1-α)% confidence intervals.

```
get_DFA_fits <- function(MLEobj, dd = NULL, alpha = 0.05) {
    ## empty list for results
    fits <- list()
    ## extra stuff for var() calcs
    Ey <- MARSS:::MARSShatyt(MLEobj)
    ## model params
    ZZ <- coef(MLEobj, type = "matrix")$Z
    ## number of obs ts
    nn <- dim(Ey$ytT)[1]
    ## number of time steps
    TT <- dim(Ey$ytT)[2]
    ## get the inverse of the rotation matrix
    H_inv <- varimax(ZZ)$rotmat
    ## check for covars
    if (!is.null(dd)) {
        DD <- coef(MLEobj, type = "matrix")$D
        ## model expectation
        fits$ex <- ZZ %*% H_inv %*% MLEobj$states + DD %*% dd
    } else {
        ## model expectation
        fits$ex <- ZZ %*% H_inv %*% MLEobj$states
    }
    ## Var in model fits
    VtT <- MARSSkfss(MLEobj)$VtT
    VV <- NULL
    for (tt in 1:TT) {
        RZVZ <- coef(MLEobj, type = "matrix")$R - ZZ %*% VtT[,
            , tt] %*% t(ZZ)
        SS <- Ey$yxtT[, , tt] - Ey$ytT[, tt, drop = FALSE] %*%
            t(MLEobj$states[, tt, drop = FALSE])
        VV <- cbind(VV, diag(RZVZ + SS %*% t(ZZ) + ZZ %*% t(SS)))
    }
    SE <- sqrt(VV)
    ## upper & lower (1-alpha)% CI
    fits$up <- qnorm(1 - alpha/2) * SE + fits$ex
    fits$lo <- qnorm(alpha/2) * SE + fits$ex
    return(fits)
}
```

Here are time series of the five phytoplankton groups (points) with the mean of the DFA fits

(black line) and the 95% confidence intervals (gray lines).

```
## get model fits & CI's
mod_fit <- get_DFA_fits(dfa_1)
## plot the fits
ylbl <- phytoplankton
par(mfrow = c(N_ts, 1), mai = c(0.5, 0.7, 0.1, 0.1), omi = c(0,
    0, 0, 0))
for (i in 1:N_ts) {
    up <- mod_fit$up[i, ]
    mn <- mod_fit$ex[i, ]
    lo <- mod_fit$lo[i, ]
    plot(w_ts, mn, xlab = "", ylab = ylbl[i], xaxt = "n", type = "n",
        cex.lab = 1.2, ylim = c(min(lo), max(up)))
    axis(1, 12 * (0:dim(dat_1980)[2]) + 1, yr_frst + 0:dim(dat_1980)[2])
    points(w_ts, dat[i, ], pch = 16, col = clr[i])
    lines(w_ts, up, col = "darkgray")
    lines(w_ts, mn, col = "black", lwd = 2)
    lines(w_ts, lo, col = "darkgray")
}
```

7.11 Covariates in DFA models

It is standard to add covariates to the analysis so that one removes known important drivers. The DFA with covariates is written:

$$\begin{aligned} \mathbf{x}_t &= \mathbf{x}_{t-1} + \mathbf{w}_t \text{ where } \mathbf{w}_t \sim \text{MVN}(0, \mathbf{Q}) \\ \mathbf{y}_t &= \mathbf{Z}\mathbf{x}_t + \mathbf{a} + \mathbf{D}\mathbf{d}_t + \mathbf{v}_t \text{ where } \mathbf{v}_t \sim \text{MVN}(0, \mathbf{R}) \end{aligned} \tag{7.12}$$

where the $q \times 1$ vector $\mathbf{d}_t$ contains the covariate(s) at time t, and the $n \times q$ matrix $\mathbf{D}$ contains the effect(s) of the covariate(s) on the observations. Using `form="dfa"` and `covariates=<covariate name(s)>`, we can easily add covariates to our DFA, but this means that the covariates are input, not data, and there can be no missing values (see Chapter 6 in the **MARSS** User Guide for how to include covariates with missing values).

7.12 Example from Lake Washington

The Lake Washington dataset has two environmental covariates that we might expect to have effects on phytoplankton growth, and hence, abundance: temperature (`Temp`) and total phosphorous (`TP`). We need the covariate inputs to have the same number of time steps as the variate data, and thus we limit the covariate data to the years 1980-1994 also.

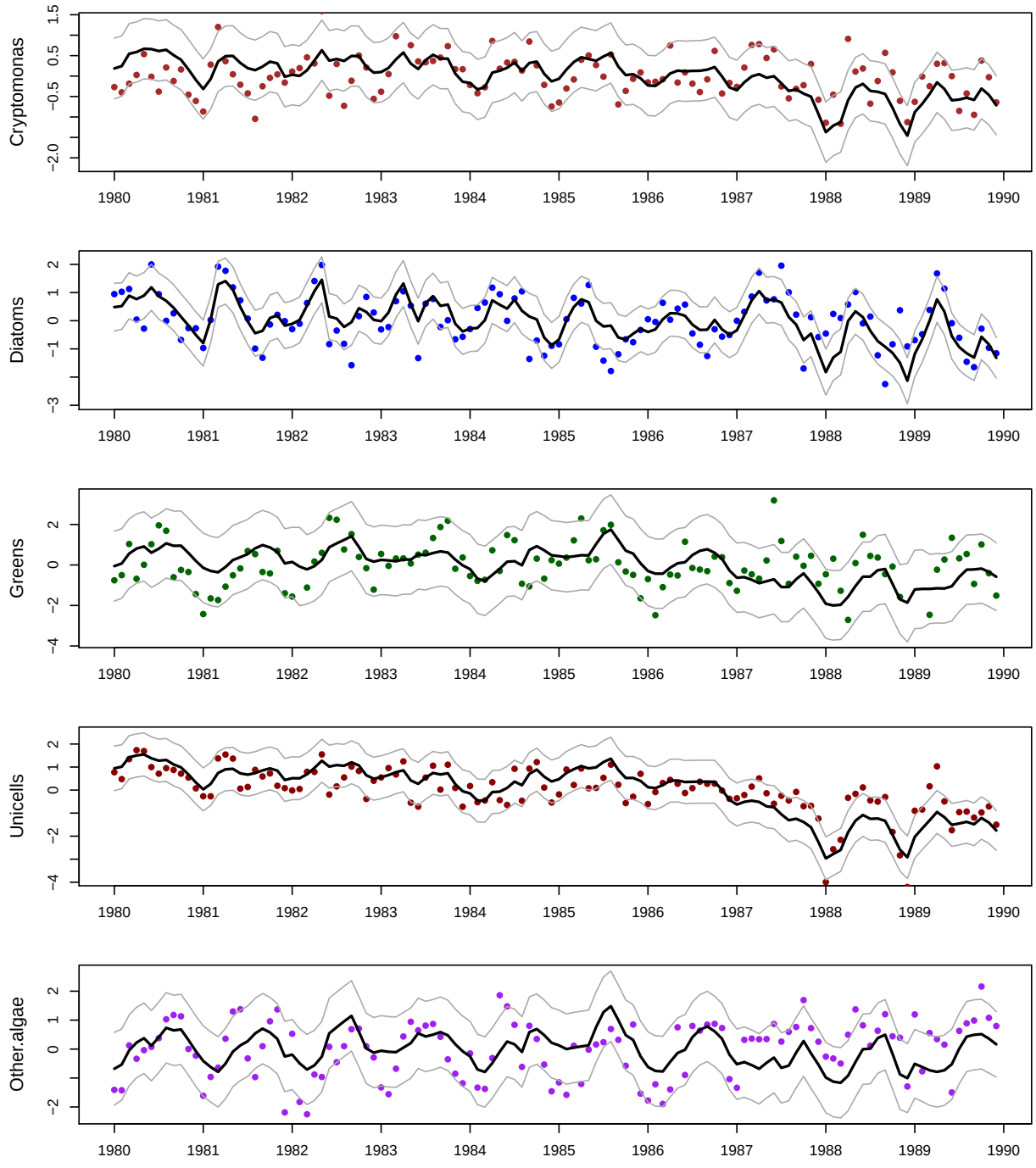


Figure 7.4: Data and fits from the DFA model.

```
temp <- t(plank_dat[, "Temp", drop = FALSE])
TP <- t(plank_dat[, "TP", drop = FALSE])
```

We will now fit three different models that each add covariate effects (i.e., `Temp`, `TP`, `Temp` and `TP`) to our existing model above where $m = 3$ and $\mathbf{R}$ is `"diagonal and unequal"`.

```
mod_list = list(m = 3, R = "diagonal and unequal")
dfa_temp <- MARSS(dat, model = mod_list, form = "dfa", z.score = FALSE,
    control = con_list, covariates = temp)
dfa_TP <- MARSS(dat, model = mod_list, form = "dfa", z.score = FALSE,
    control = con_list, covariates = TP)
dfa_both <- MARSS(dat, model = mod_list, form = "dfa", z.score = FALSE,
    control = con_list, covariates = rbind(temp, TP))
```

Next we can compare whether the addition of the covariates improves the model fit.

```
print(cbind(model = c("no covars", "Temp", "TP", "Temp & TP"),
    AICc = round(c(dfa_1$AICc, dfa_temp$AICc, dfa_TP$AICc, dfa_both$AICc))),
    quote = FALSE)
```

```
     model     AICc
[1,] no covars 1427
[2,] Temp      1356
[3,] TP        1414
[4,] Temp & TP 1362
```

This suggests that adding temperature or phosphorus to the model, either alone or in combination with one another, does seem to improve overall model fit. If we were truly interested in assessing the "best" model structure that includes covariates, however, we should examine all combinations of 1-4 trends and different structures for $\mathbf{R}$.

Now let's try to fit a model with a dummy variable for season, and see how that does.

```
cos_t <- cos(2 * pi * seq(TT)/12)
sin_t <- sin(2 * pi * seq(TT)/12)
dd <- rbind(cos_t, sin_t)
dfa_seas <- MARSS(dat_1980, model = mod_list, form = "dfa", z.score = TRUE,
    control = con_list, covariates = dd)
```

```
Success! abstol and log-log tests passed at 384 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 384 iterations.
Log-likelihood: -713.8464
```

```
AIC: 1481.693   AICc: 1484.355

                             Estimate
Z.11                          0.49562
Z.21                          0.27206
Z.31                          0.03354
Z.41                          0.51692
Z.51                          0.18981
Z.22                          0.05290
Z.32                         -0.08042
Z.42                          0.06336
Z.52                          0.06157
Z.33                          0.02383
Z.43                          0.19506
Z.53                         -0.10800
R.(Cryptomonas,Cryptomonas)   0.51583
R.(Diatoms,Diatoms)           0.53296
R.(Greens,Greens)             0.60329
R.(Unicells,Unicells)         0.19787
R.(Other.algae,Other.algae)   0.52977
D.(Cryptomonas,cos_t)        -0.43973
D.(Diatoms,cos_t)            -0.44836
D.(Greens,cos_t)             -0.66003
D.(Unicells,cos_t)           -0.34898
D.(Other.algae,cos_t)        -0.42773
D.(Cryptomonas,sin_t)         0.23672
D.(Diatoms,sin_t)             0.72062
D.(Greens,sin_t)             -0.46019
D.(Unicells,sin_t)           -0.00873
D.(Other.algae,sin_t)        -0.64228

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

```
dfa_seas$AICc
```

```
[1] 1484.355
```

The model with a dummy seasonal factor does much better than the covariate models, but still not as well as the model with only 3 trends. The model fits for the seasonal effects model are shown below.

```
## get model fits & CI's
mod_fit <- get_DFA_fits(dfa_seas, dd = dd)
## plot the fits
ylbl <- phytoplankton
```

```
par(mfrow = c(N_ts, 1), mai = c(0.5, 0.7, 0.1, 0.1), omi = c(0,
    0, 0, 0))
for (i in 1:N_ts) {
    up <- mod_fit$up[i, ]
    mn <- mod_fit$ex[i, ]
    lo <- mod_fit$lo[i, ]
    plot(w_ts, mn, xlab = "", ylab = ylbl[i], xaxt = "n", type = "n",
        cex.lab = 1.2, ylim = c(min(lo), max(up)))
    axis(1, 12 * (0:dim(dat_1980)[2]) + 1, yr_frst + 0:dim(dat_1980)[2])
    points(w_ts, dat[i, ], pch = 16, col = clr[i])
    lines(w_ts, up, col = "darkgray")
    lines(w_ts, mn, col = "black", lwd = 2)
    lines(w_ts, lo, col = "darkgray")
}
```

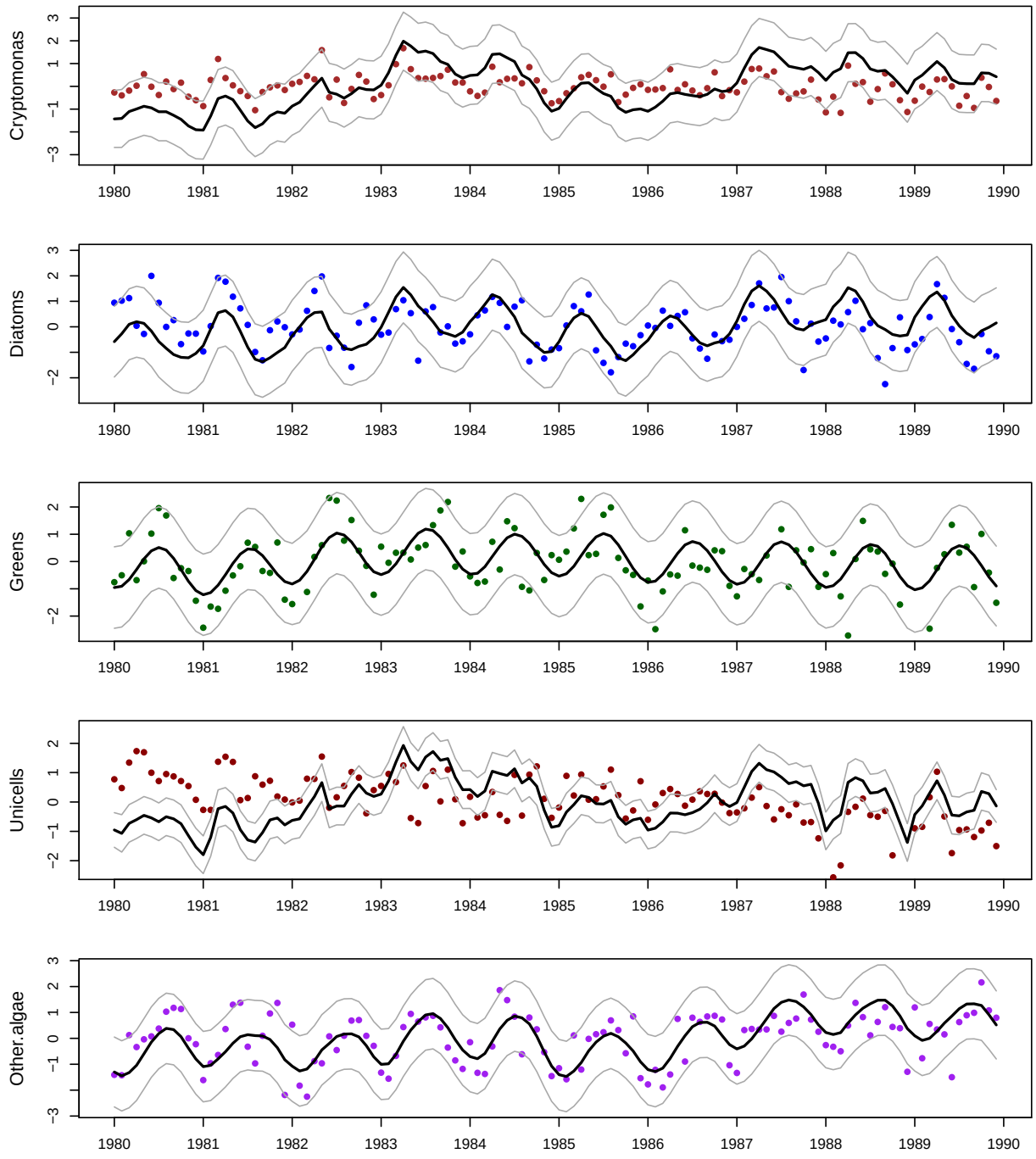


Figure 7.5: (#fig:dfa-plot_dfa_temp_fits)Data and model fits for the DFA with covariates.

7.13 Problems

For your homework this week, we will continue to investigate common trends in the Lake Washington plankton data.

1. Fit other DFA models to the phytoplankton data with varying numbers of trends from 1-4 (we fit a 3-trend model above). Do not include any covariates in these models. Using `R="diagonal and unequal"` for the observation errors, which of the DFA models has the most support from the data?

   Plot the model states and loadings as in Section 7.9. Describe the general patterns in the states and the ways the different taxa load onto those trends.

   Also plot the the model fits as in Section 7.10. Do they reasonable? Are there any particular problems or outliers?

2. How does the best model from Question 1 compare to a DFA model with the same number of trends, but with `R="unconstrained"`?

   Plot the model states and loadings as in Section 7.9. Describe the general patterns in the states and the ways the different taxa load onto those trends.

   Also plot the the model fits as in Section 7.10. Do they reasonable? Are there any particular problems or outliers?

3. Fit a DFA model that includes temperature as a covariate and 3 trends (as in Section 7.12), but with`R="unconstrained"`? How does this model compare to the model with `R="diagonal and unequal"`? How does it compare to the model in Question 2?

   Plot the model states and loadings as in Section 7.9. Describe the general patterns in the states and the ways the different taxa load onto those trends.

Also plot the the model fits as in Section 7.10. Do they reasonable? Are there any particular problems or outliers?

Chapter 8

JAGS for Bayesian time-series analysis

In this lab, we will work through using Bayesian methods to estimate parameters in time series models. There are a variety of software tools to do time-series analysis using Bayesian methods. R lists a number of packages available on the R Cran TimeSeries task view.

Software to implement more complicated models is also available, and many of you are probably familiar with these options (AD Model Builder and Template Model Builder, WinBUGS, OpenBUGS, JAGS, Stan, to name a few). In this chapter, we will show you how to write state-space models in JAGS and fit these models.

After updating to the latest version of R, install JAGS for your operating platform using the instructions here. Click on JAGS, then the most recent folder, then the platform of your machine. You will also need the **coda** and **R2jags** packages.

```
library(coda)
library(R2jags)
```

8.1 The airquality dataset

For data for this lab, we will include a dataset on air quality in New York. We will load the data and create a couple new variables for future use. For the majority of our models, we are going to treat wind speed as the response variable for our time-series models.

```
data(airquality)
Wind = airquality$Wind  # wind speed
Temp = airquality$Temp  # air temperature
N = dim(airquality)[1]  # number of data points
```

8.2 Linear regression with no covariates

We will start with the simplest time series model possible: linear regression with only an intercept, so that the predicted values of all observations are the same. There are several ways we can write this equation. First, the predicted values can be written as $E[y_t] = \mu$. Assuming that the residuals are normally distributed, the model linking our predictions to observed data are written as

$$y_t = \mu + e_t, e_t \sim \text{N}(0, \sigma^2) \tag{8.1}$$

An equivalent way to think about this model is that instead of the residuals as normally distributed with mean zero, we can think of the data y as being normally distributed with a mean of the intercept, and the same residual standard deviation:

$$y \sim \text{N}(E[y_t], \sigma^2) \tag{8.2}$$

Remember that in linear regression models, the residual error is interpreted as independent and identically distributed observation error.

To run the JAGS model, we will need to start by writing the model in JAGS notation. For our linear regression model, one way to construct the model is

```
# 1. LINEAR REGRESSION with no covariates no covariates, so
# intercept only. The parameters are mean 'mu' and
# precision/variance parameter 'tau.obs'

model.loc = "lm_intercept.txt"  # name of the txt file
jagsscript = cat("
model {
   # priors on parameters
   mu ~ dnorm(0, 0.01); # mean = 0, sd = 1/sqrt(0.01)
   tau.obs ~ dgamma(0.001,0.001); # This is inverse gamma
   sd.obs <- 1/sqrt(tau.obs); # sd is treated as derived parameter

    for(i in 1:N) {
      Y[i] ~ dnorm(mu, tau.obs);
   }
}
",
    file = model.loc)
```

A couple things to notice: JAGS is not vectorized so we need to use for loops (instead of matrix multiplication) and the `dnorm` notation means that we assume that value (on the left) is normally distributed around a particular mean with a particular precision (1 over the square root of the variance).

The model can briefly be summarized as follows: there are 2 parameters in the model (the mean and variance of the observation error). JAGS is a bit funny in that instead of giving a normal distribution the standard deviation or variance, you pass in the precision (1/variance), so our prior on μ is pretty vague. The precision receives a gamma prior, which is equivalent to the variance receiving an inverse gamma prior (fairly common for standard Bayesian regression models). We will treat the standard deviation as derived (if we know the variance or precision, which we are estimating, we automatically know the standard deviation). Finally, we write a model for the data y_t (`Y[i]`). Again we use the `dnorm` distribution to say that the data are normally distributed (equivalent to our likelihood).

The function from the **R2jags** package that we actually use to run the model is `jags()`. There is a parallel version of the function called `jags.parallel()` which is useful for larger, more complex models. The details of both can be found with `?jags` or `?jags.parallel`.

To actually run the model, we need to create several new objects, representing (1) a list of data that we will pass to JAGS, (2) a vector of parameters that we want to monitor in JAGS and have returned back to R, and (3) the name of our text file that contains the JAGS model we wrote above. With those three things, we can call the `jags()` function.

```
jags.data = list(Y = Wind, N = N)  # named list of inputs
jags.params = c("sd.obs", "mu")  # parameters to be monitored
mod_lm_intercept = jags(jags.data, parameters.to.save = jags.params,
    model.file = model.loc, n.chains = 3, n.burnin = 5000, n.thin = 1,
    n.iter = 10000, DIC = TRUE)
```

Notice that the `jags()` function contains a number of other important arguments. In general, larger is better for all arguments: we want to run multiple MCMC chains (maybe 3 or more), and have a burn-in of at least 5000. The total number of samples after the burn-in period is n.iter-n.burnin, which in this case is 5000 samples. Because we are doing this with 3 MCMC chains, and the thinning rate equals 1 (meaning we are saving every sample), we will retain a total of 1500 posterior samples for each parameter.

The saved object storing our model diagnostics can be accessed directly, and includes some useful summary output.

```
mod_lm_intercept
```

```
Inference for Bugs model at "lm_intercept.txt", fit using jags,
 3 chains, each with 10000 iterations (first 5000 discarded)
 n.sims = 15000 iterations saved
         mu.vect sd.vect    2.5%     25%     50%     75%   97.5%  Rhat
mu         9.950   0.283   9.390   9.761   9.947  10.143  10.506 1.001
sd.obs     3.536   0.204   3.161   3.396   3.526   3.665   3.967 1.001
deviance 820.524   1.995 818.590 819.108 819.893 821.315 825.825 1.001
         n.eff
mu       15000
sd.obs   14000
deviance 11000
```

```
For each parameter, n.eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor (at convergence, Rhat=1).

DIC info (using the rule, pD = var(deviance)/2)
pD = 2.0 and DIC = 822.5
DIC is an estimate of expected predictive error (lower deviance is better).
```

The last 2 columns in the summary contain `Rhat` (which we want to be close to 1.0), and `neff` (the effective sample size of each set of posterior draws). To examine the output more closely, we can pull all of the results directly into R,

```
attach.jags(mod_lm_intercept)
```

```
The following object is masked _by_ .GlobalEnv:

    mu
```

Attaching the **R2jags** object allows us to work with the named parameters directly in R. For example, we could make a histogram of the posterior distributions of the parameters `mu` and `sd.obs` with the following code,

```
# Now we can make plots of posterior values
par(mfrow = c(2, 1))
hist(mu, 40, col = "grey", xlab = "Mean", main = "")
hist(sd.obs, 40, col = "grey", xlab = expression(sigma[obs]),
    main = "")
```

Finally, we can run some useful diagnostics from the **coda** package on this model output. We have written a small function to make the creation of mcmc lists (an argument required for many of the diagnostics). The function

```
createMcmcList = function(jagsmodel) {
    McmcArray = as.array(jagsmodel$BUGSoutput$sims.array)
    McmcList = vector("list", length = dim(McmcArray)[2])
    for (i in 1:length(McmcList)) McmcList[[i]] = as.mcmc(McmcArray[,
        i, ])
    McmcList = mcmc.list(McmcList)
    return(McmcList)
}
```

Creating the MCMC list preserves the random samples generated from each chain and allows you to extract the samples for a given parameter (such as μ) from any chain you want. To extract μ from the first chain, for example, you could use the following code. Because `createMcmcList()` returns a list of **mcmc** objects, we can summarize and plot these directly. Figure 8.2 shows the plot from `plot(myList[[1]])`.

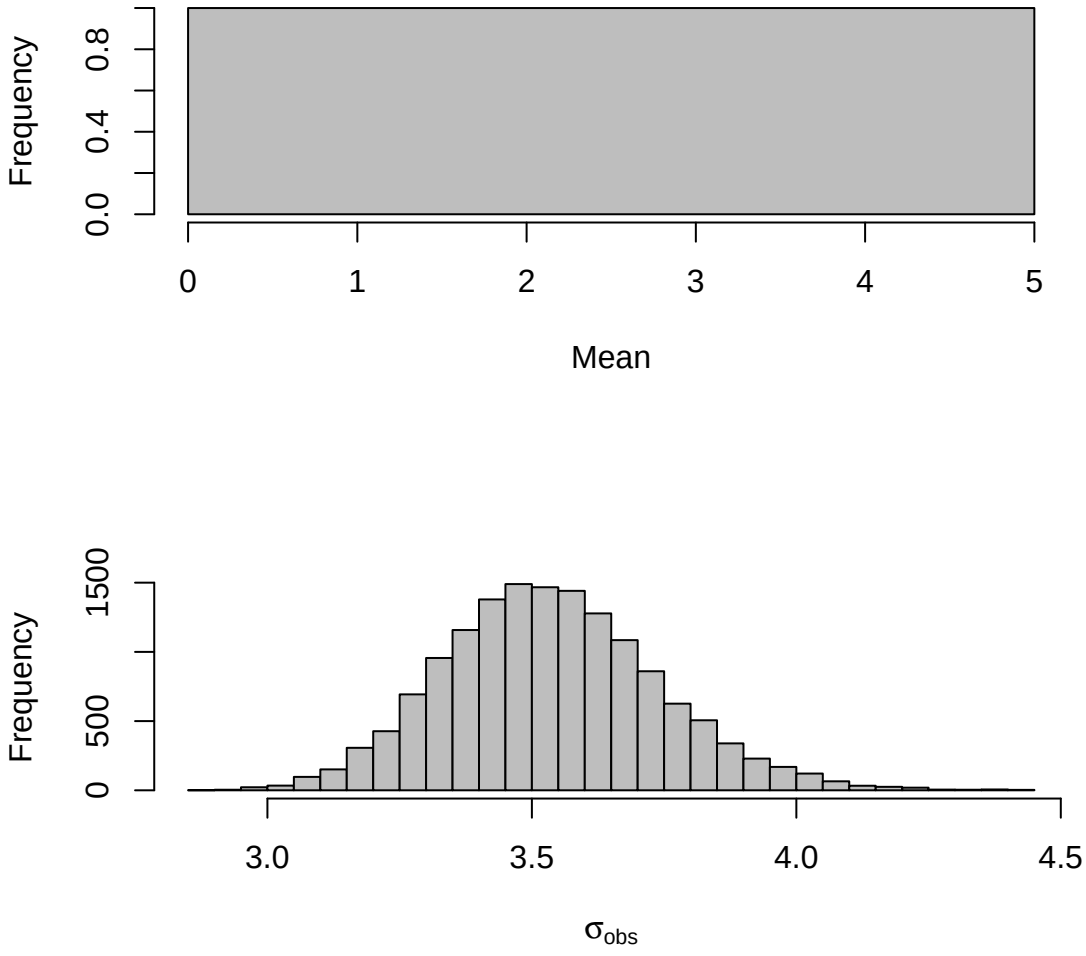
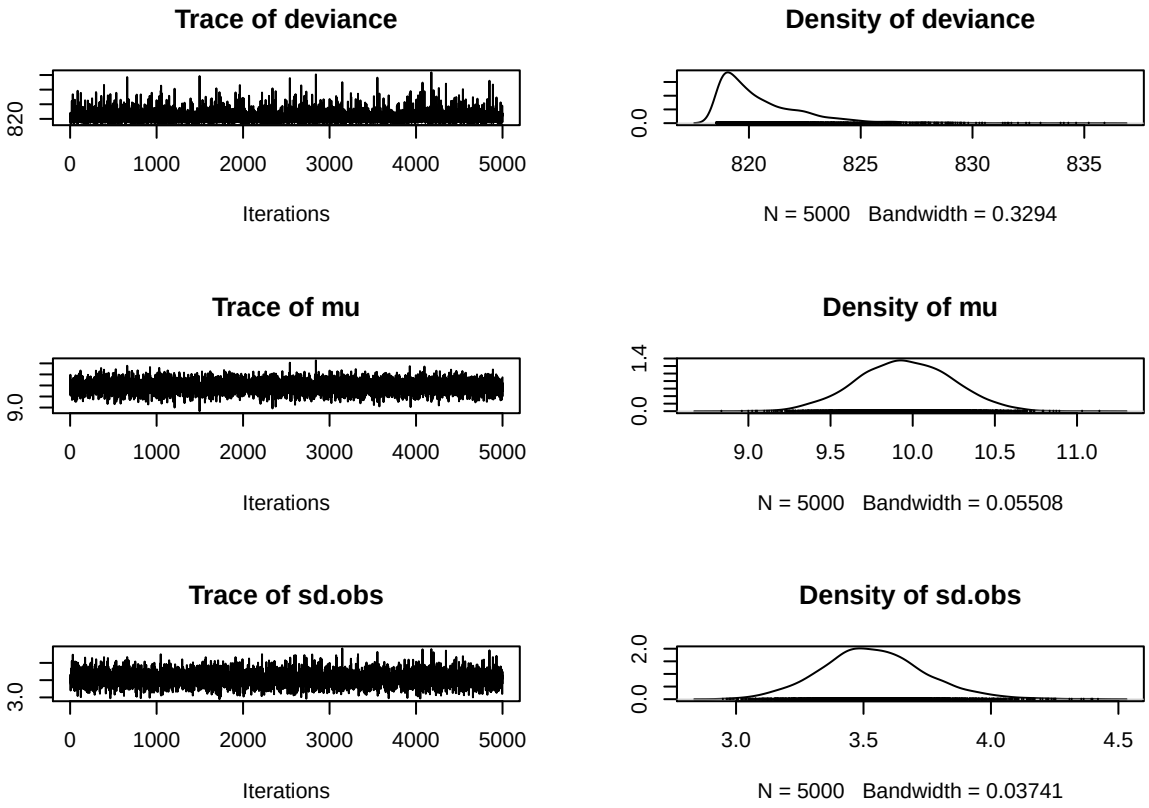


Figure 8.1: Plot of the posteriors for the linear regression model.

```
myList = createMcmcList(mod_lm_intercept)
summary(myList[[1]])
```

```
Iterations = 1:5000
Thinning interval = 1
Number of chains = 1
Sample size per chain = 5000

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

            Mean     SD Naive SE Time-series SE
deviance 820.541 2.0350 0.028779       0.030093
mu         9.952 0.2854 0.004037       0.004037
sd.obs     3.539 0.2041 0.002886       0.002886

2. Quantiles for each variable:

            2.5%     25%     50%     75%   97.5%
deviance 818.587 819.099 819.888 821.386 826.111
mu         9.389   9.754   9.949  10.151  10.508
sd.obs     3.158   3.406   3.529   3.666   3.968
```

```
plot(myList[[1]])
```



For more quantitative diagnostics of MCMC convergence, we can rely on the **coda** package in R. There are several useful statistics available, including the Gelman-Rubin diagnostic (for one or several chains), autocorrelation diagnostics (similar to the ACF you calculated above), the Geweke diagnostic, and Heidelberger-Welch test of stationarity.

```
# Run the majority of the diagnostics that CODA() offers
library(coda)
gelmanDiags = gelman.diag(createMcmcList(mod_lm_intercept), multivariate = F)
autocorDiags = autocorr.diag(createMcmcList(mod_lm_intercept))
gewekeDiags = geweke.diag(createMcmcList(mod_lm_intercept))
heidelDiags = heidel.diag(createMcmcList(mod_lm_intercept))
```

8.3 Regression with autocorrelated errors

In our first model, the errors were independent in time. We are going to modify this to model autocorrelated errors. Autocorrelated errors are widely used in ecology and other fields – for a greater discussion, see Morris and Doak (2002) Quantitative Conservation Biology. To make the deviations autocorrelated, we start by defining the deviation in the first time step, $e_1 = Y_1 - u$. The expectation of y_t in each time step is then written as

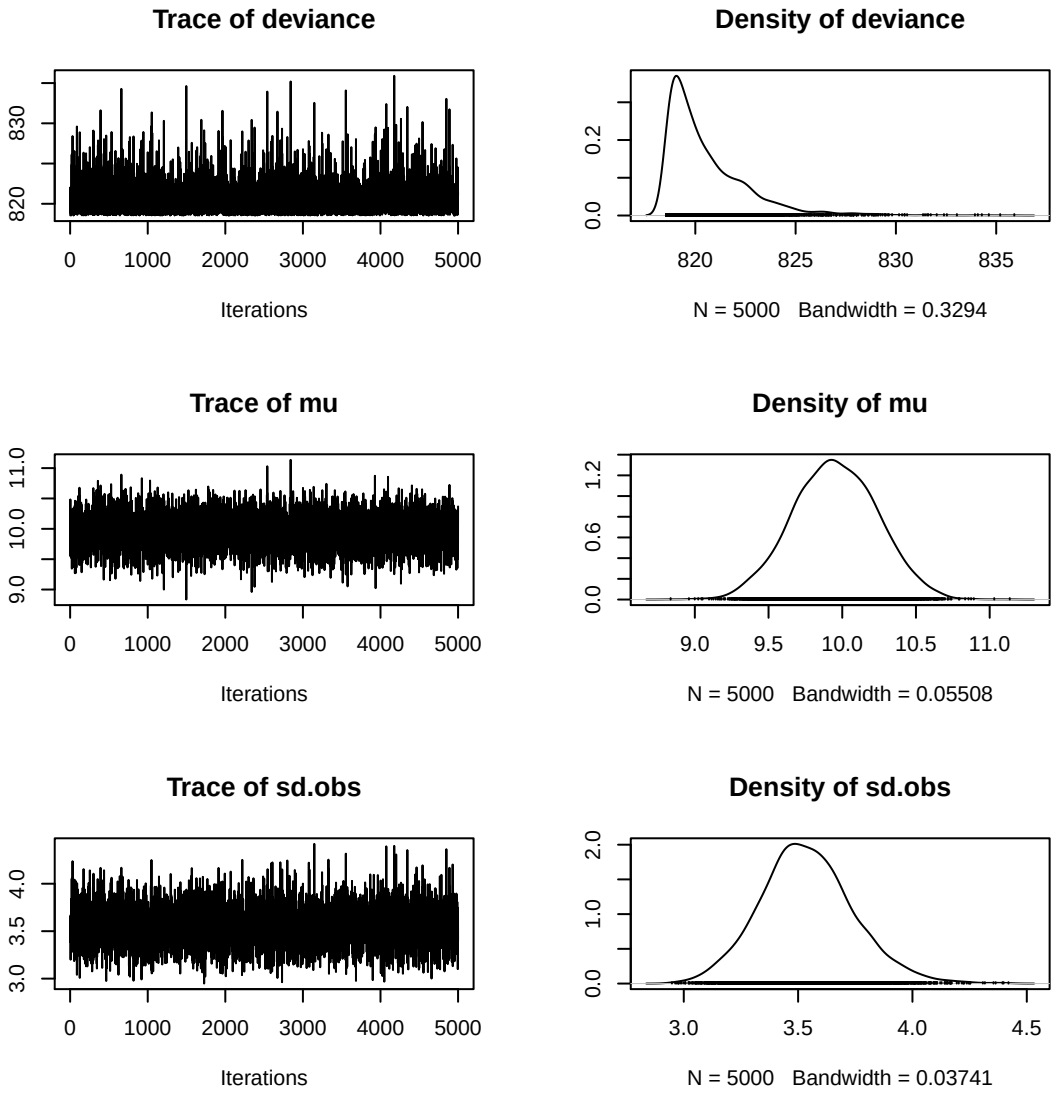


Figure 8.2: Plot of an object output from `creatMcmcList`.

$$E[y_t] = \mu + \phi * e_{t-1} \tag{8.3}$$

In addition to affecting the expectation, the correlation parameter ϕ also affects the variance of the errors, so that

$$\sigma^2 = \psi^2 \left(1 - \phi^2\right) \tag{8.4}$$

Like in our first model, we assume that the data follow a normal likelihood (or equivalently that the residuals are normally distributed), $y_t = E[y_t] + e_t$, or $y_t \sim \mathrm{N}(E[y_t], \sigma^2)$. Thus, it is possible to express the subsequent deviations as $e_t = y_t - E[y_t]$, or equivalently as $e_t = y_t - \mu - \phi \times e_{t-1}$. The JAGS script for this model is:

```
# 2. LINEAR REGRESSION WITH AUTOCORRELATED ERRORS no
# covariates, so intercept only.

model.loc = ("lmcor_intercept.txt")
jagsscript = cat("
model {
   # priors on parameters
   mu ~ dnorm(0, 0.01);
   tau.obs ~ dgamma(0.001,0.001);
   sd.obs <- 1/sqrt(tau.obs);
   phi ~ dunif(-1,1);
   tau.cor <- tau.obs * (1-phi*phi); # Var = sigma2 * (1-rho^2)

   epsilon[1] <- Y[1] - mu;
   predY[1] <- mu; # initial value
   for(i in 2:N) {
      predY[i] <- mu + phi * epsilon[i-1];
      Y[i] ~ dnorm(predY[i], tau.cor);
      epsilon[i] <- (Y[i] - mu) - phi*epsilon[i-1];
   }
}
",
    file = model.loc)
```

Notice several subtle changes from the simpler first model: (1) we are estimating the autocorrelation parameter ϕ, which is assigned a Uniform(-1, 1) prior, (2) we model the residual variance as a function of the autocorrelation, and (3) we allow the autocorrelation to affect the predicted values `predY`. One other change we can make is to add `predY` to the list of parameters we want returned to R.

```
jags.data = list(Y = Wind, N = N)
jags.params = c("sd.obs", "predY", "mu", "phi")
mod_lmcor_intercept = jags(jags.data, parameters.to.save = jags.params,
```

```
    model.file = model.loc, n.chains = 3, n.burnin = 5000, n.thin = 1,
    n.iter = 10000, DIC = TRUE)
```

For some models, we may be interested in examining the posterior fits to data. You can make this plot yourself, but we have also put together a simple function whose arguments are one of our fitted models and the raw data. The function is:

```
plotModelOutput = function(jagsmodel, Y) {
    # attach the model
    attach.jags(jagsmodel)
    x = seq(1, length(Y))
    summaryPredictions = cbind(apply(predY, 2, quantile, 0.025),
        apply(predY, 2, mean), apply(predY, 2, quantile, 0.975))
    plot(Y, col = "white", ylim = c(min(c(Y, summaryPredictions)),
        max(c(Y, summaryPredictions))), xlab = "", ylab = "95% CIs of predictions and da
        main = paste("JAGS results:", jagsmodel$model.file))
    polygon(c(x, rev(x)), c(summaryPredictions[, 1], rev(summaryPredictions[,
        3])), col = "grey70", border = NA)
    lines(summaryPredictions[, 2])
    points(Y)
}
```

We can use the function to plot the predicted posterior mean with 95% CIs, as well as the raw data. For example, try

```
plotModelOutput(mod_lmcor_intercept, Wind)
```

```
The following object is masked _by_ .GlobalEnv:

    mu
```

8.4 Random walk time series model

All of the previous three models can be interpreted as observation error models. Switching gears, we can alternatively model error in the state of nature, creating process error models. A simple process error model that many of you may have seen before is the random walk model. In this model, the assumption is that the true state of nature (or latent states) are measured perfectly. Thus, all uncertainty is originating from process variation (for ecological problems, this is often interpreted as environmental variation). For this simple model, we will assume that our process of interest (in this case, daily wind speed) exhibits no daily trend, but behaves as a random walk.

$$E[y_t] = y_{t-1} + e_{t-1} \tag{8.5}$$

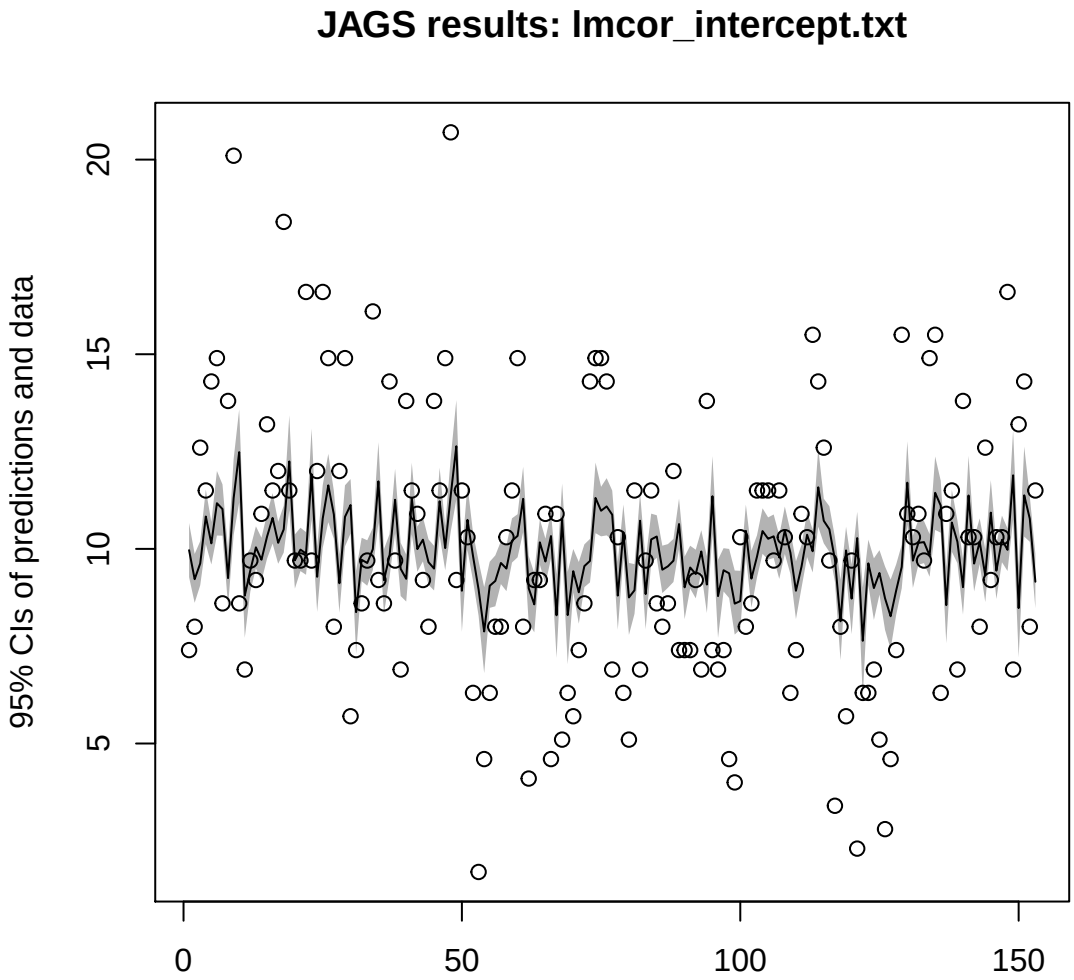


Figure 8.3: Predicted posterior mean with 95% CIs

And the $e_t \sim N(0, \sigma^2)$. Remember back to the autocorrelated model (or MA(1) models) that we assumed that the errors e_t followed a random walk. In contrast, the AR(1) model assumes that the errors are independent, but that the state of nature follows a random walk. The JAGS random walk model and R script to run it is below:

```
# 3. AR(1) MODEL WITH NO ESTIMATED AR COEFFICIENT = RANDOM
# WALK no covariates. The model is y[t] ~ Normal(y[n-1],
# sigma) for we will call the precision tau.pro Note too that
# we have to define predY[1]
model.loc = ("rw_intercept.txt")
jagsscript = cat("
model {
   mu ~ dnorm(0, 0.01);
   tau.pro ~ dgamma(0.001,0.001);
   sd.pro <- 1/sqrt(tau.pro);

   predY[1] <- mu; # initial value
   for(i in 2:N) {
      predY[i] <- Y[i-1];
      Y[i] ~ dnorm(predY[i], tau.pro);
   }
}
",
    file = model.loc)

jags.data = list(Y = Wind, N = N)
jags.params = c("sd.pro", "predY", "mu")
mod_rw_intercept = jags(jags.data, parameters.to.save = jags.params,
    model.file = model.loc, n.chains = 3, n.burnin = 5000, n.thin = 1,
    n.iter = 10000, DIC = TRUE)
```

8.5 Autoregressive AR(1) time series models

A variation of the random walk model described previously is the autoregressive time-series model of order 1, AR(1). This model introduces a coefficient, which we will call ϕ. The parameter ϕ controls the degree to which the random walk reverts to the mean—when $\phi = 1$, the model is identical to the random walk, but at smaller values, the model will revert back to the mean (which in this case is zero). Also, ϕ can take on negative values, which we will discuss more in future lectures. The math to describe the AR(1) time series model is:

$$E[y_t] = \phi * y_{t-1} + e_{t-1} \tag{8.6}$$

The JAGS random walk model and R script to run the AR(1) model is below:

```
# 4. AR(1) MODEL WITH AND ESTIMATED AR COEFFICIENT We're
# introducting a new AR coefficient 'phi', so the model is
# y[t] ~ N(mu + phi*y[n-1], sigma^2)

model.loc = ("ar1_intercept.txt")
jagsscript = cat("
model {
   mu ~ dnorm(0, 0.01);
   tau.pro ~ dgamma(0.001,0.001);
   sd.pro <- 1/sqrt(tau.pro);
   phi ~ dnorm(0, 1);

   predY[1] <- Y[1];
   for(i in 2:N) {
      predY[i] <- mu + phi * Y[i-1];
      Y[i] ~ dnorm(predY[i], tau.pro);
   }
}
",
    file = model.loc)

jags.data = list(Y = Wind, N = N)
jags.params = c("sd.pro", "predY", "mu", "phi")
mod_ar1_intercept = jags(jags.data, parameters.to.save = jags.params,
    model.file = model.loc, n.chains = 3, n.burnin = 5000, n.thin = 1,
    n.iter = 10000, DIC = TRUE)
```

8.6 Univariate state space model

At this point, we have fit models with observation or process error, but we have not tried to estimate both simultaneously. We will do so here, and introduce some new notation to describe the process model and observation model. We use the notation x_t to denote the latent state or state of nature (which is unobserved) at time t and y_t to denote the observed data. For introductory purposes, we will make the process model autoregressive (similar to our AR(1) model),

$$x_t = \phi * x_{t-1} + e_{t-1}; e_{t-1} \sim \text{N}(0, q) \tag{8.7}$$

For the process model, there are a number of ways to parameterize the first state (x_1), and we will talk about this more in the class. For the sake of this model, we will place a vague weakly informative prior on x_1: $x_1 \sim \text{N}(0, 0.01)$. Second, we need to construct an observation model

linking the estimate unseen states of nature x_t to the data y_t. For simplicitly, we will assume that the observation errors are indepdendent and identically distributed, with no observation component. Mathematically, this model is

$$y_t \sim \mathrm{N}(x_t, r) \tag{8.8}$$

In the two above models, q is the process variance and r is the observation error variance. The JAGS code will use the standard deviation (square root) of these. The code to produce and fit this model is below:

```
# 5. MAKE THE SS MODEL a univariate random walk no
# covariates.

model.loc = ("ss_model.txt")
jagsscript = cat("
model {
   # priors on parameters
   mu ~ dnorm(0, 0.01);
   tau.pro ~ dgamma(0.001,0.001);
   sd.q <- 1/sqrt(tau.pro);
   tau.obs ~ dgamma(0.001,0.001);
   sd.r <- 1/sqrt(tau.obs);
   phi ~ dnorm(0,1);

   X[1] <- mu;
   predY[1] <- X[1];
   Y[1] ~ dnorm(X[1], tau.obs);

   for(i in 2:N) {
      predX[i] <- phi*X[i-1];
      X[i] ~ dnorm(predX[i],tau.pro); # Process variation
      predY[i] <- X[i];
      Y[i] ~ dnorm(X[i], tau.obs); # Observation variation
   }
}
",
    file = model.loc)

jags.data = list(Y = Wind, N = N)
jags.params = c("sd.q", "sd.r", "predY", "mu")
mod_ss = jags(jags.data, parameters.to.save = jags.params, model.file = model.loc,
    n.chains = 3, n.burnin = 5000, n.thin = 1, n.iter = 10000,
    DIC = TRUE)
```

8.6.1 Including covariates

Returning to the first example of regression with the intercept only, we will introduce `Temp` as the covariate explaining our response variable `Wind`. Note that to include the covariate, we (1) modify the JAGS script to include a new coefficient—in this case `beta`, (2) update the predictive equation to include the effects of the new covariate, and (3) we include the new covariate in our named data list.

```
# 6. Include some covariates in a linear regression Use
# temperature as a predictor of wind

model.loc = ("lm.txt")
jagsscript = cat("
model {
   mu ~ dnorm(0, 0.01);
   beta ~ dnorm(0,0.01);
   tau.obs ~ dgamma(0.001,0.001);
   sd.obs <- 1/sqrt(tau.obs);

   for(i in 1:N) {
      predY[i] <- mu + C[i]*beta;
      Y[i] ~ dnorm(predY[i], tau.obs);
   }
}
",
    file = model.loc)

jags.data = list(Y = Wind, N = N, C = Temp)
jags.params = c("sd.obs", "predY", "mu", "beta")
mod_lm = jags(jags.data, parameters.to.save = jags.params, model.file = model.loc,
    n.chains = 3, n.burnin = 5000, n.thin = 1, n.iter = 10000,
    DIC = TRUE)
```

8.7 Forecasting with JAGS models

There are a number of different approaches to using Bayesian time-series models to perform forecasting. One approach might be to fit a model, and use those posterior distributions to forecast as a secondary step (say within R). A more streamlined approach is to do this within the JAGS code itself. We can take advantage of the fact that JAGS allows you to include NAs in the response variable (but never in the predictors). Let's use the same Wind dataset, and the univariate state-space model described above to forecast three time steps into the future. We can do this by including 3 more NAs in the dataset, and incrementing the variable `N` by 3.

```
jags.data = list(Y = c(Wind, NA, NA, NA), N = (N + 3))
jags.params = c("sd.q", "sd.r", "predY", "mu")
model.loc = ("ss_model.txt")
mod_ss_forecast = jags(jags.data, parameters.to.save = jags.params,
    model.file = model.loc, n.chains = 3, n.burnin = 5000, n.thin = 1,
    n.iter = 10000, DIC = TRUE)
```

We can inspect the fitted model object, and see that `predY` contains the 3 new predictions for the forecasts from this model.

8.8 Problems

1. Fit the intercept only model from section 8.2. Set the burn-in to 3, and when the model completes, plot the time series of the parameter `mu` for the first MCMC chain.
   a. Based on your visual inspection, has the MCMC chain convered?
   b. What is the ACF of the first MCMC chain?
2. Increase the MCMC burn-in for the model in question 1 to a value that you think is reasonable. After the model has converged, calculate the Gelman-Rubin diagnostic for the fitted model object.
3. Compare the results of the `plotModelOutput()` function for the intercept only model from section 8.2. You will to add "predY" to your JAGS model and to the list of parameters to monitor, and re-run the model.
4. Modify the random walk model without drift from section 8.4 to a random walk model with drift. The equation for this model is

$$E[y_t] = y_{t-1} + \mu + e_{t-1}$$

   where μ is interpreted as the average daily trend in wind speed. What might be a reasonable prior on μ?
5. Plot the posterior distribution of ϕ for the AR(1) model in section 8.5. Can this parameter be well estimated for this dataset?
6. Plot the posteriors for the process and observation variances (not standard deviation) for the univariate state-space model in section 8.6. Which is larger for this dataset?
7. Add the effect of temperature to the AR(1) model in section 8.5. Plot the posterior for `beta` and compare to the posterior for `beta` from the model in section 8.6.1.
8. Plot the fitted values from the model in section 8.7, including the forecasts, with the 95% credible intervals for each data point.
9. The following is a dataset from the Upper Skagit River (Puget Sound, 1952-2005) on salmon spawners and recruits:

```
Spawners = c(2662, 1806, 1707, 1339, 1686, 2220, 3121, 5028,
    9263, 4567, 1850, 3353, 2836, 3961, 4624, 3262, 3898, 3039,
    5966, 5931, 7346, 4911, 3116, 3185, 5590, 2485, 2987, 3829,
    4921, 2348, 1932, 3151, 2306, 1686, 4584, 2635, 2339, 1454,
    3705, 1510, 1331, 942, 884, 666, 1521, 409, 2388, 1043, 3262,
    2606, 4866, 1161, 3070, 3320)
Recruits = c(12741, 15618, 23675, 37710, 62260, 32725, 8659,
    28101, 17054, 29885, 33047, 20059, 35192, 11006, 48154, 35829,
    46231, 32405, 20782, 21340, 58392, 21553, 27528, 28246, 35163,
```

```
    15419, 16276, 32946, 11075, 16909, 22359, 8022, 16445, 2912,
    17642, 2929, 7554, 3047, 3488, 577, 4511, 1478, 3283, 1633,
    8536, 7019, 3947, 2789, 4606, 3545, 4421, 1289, 6416, 3647)
logRS = log(Recruits/Spawners)
```

a. Fit the following Ricker model to these data using the following linear form of this model with normally distributed errors:

$$log(R_t/S_t) = a + b \times S_t + e_t, \text{ where } e_t \sim \text{N}(0, \sigma^2)$$

You will recognize that this form is exactly the same as linear regression, with independent errors (very similar to the intercept only model of Wind we fit in section 8.2).

b. Within the constraints of the Ricker model, think about other ways you might want to treat the errors. The basic model described above has independent errors that are not correlated in time. Approaches to analyzing this dataset might involve

- modeling the errors as independent (as described above)
- modeling the errors as autocorrelated
- fitting a state-space model, with independent or correlated process errors

Fit each of these models, and compare their performance (either using their predictive ability, or forecasting ability).

Chapter 9

Stan for Bayesian time-series analysis

For this lab, we will use Stan for fitting models. These examples are primarily drawn from the Stan manual and previous code from this class. For running the code in this chapter, you will need to download and install **RStan** for your operating system: follow the instructions here.

A script with all the R code in the chapter can be downloaded here.

9.1 Stan packages and chapter data sets

You will need the **statss** package we have written for fitting state-space time-series models with Stan. This is hosted on Github safs-timeseries. Install using the **devtools** package.

```
library(devtools)
devtools::install_github("eric-ward/safs-timeseries/statss")
```

In addition, you will need the **rstan** and **datasets** packages. After installing, if needed, load the packages:

```
library(statss)
library(rstan)
library(datasets)
```

Once you have Stan and **rstan** installed, optimize Stan on your machine:

```
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())
```

For this lab, we will use a data set on airquality in New York from the **datasets** package. Load the data and create a couple new variables for future use.

```
data(airquality)
Wind = airquality$Wind  # wind speed
Temp = airquality$Temp  # air temperature
```

9.2 Linear regression

We'll start with the simplest time series model possible: linear regression with only an intercept, so that the predicted values of all observations are the same. There are several ways we can write this equation. First, the predicted values can be written as $E[Y_t] = \beta x$, where $x = 1$. Assuming that the residuals are normally distributed, the model linking our predictions to observed data is written as

$$y_t = \beta x + e_t, e_t \sim N(0, \sigma), x = 1$$

An equivalent way to think about this model is that instead of the residuals as normally distributed with mean zero, we can think of the data y_t as being drawn from a normal distribution with a mean of the intercept, and the same residual standard deviation:

$$Y_t \sim N(E[Y_t], \sigma)$$

Remember that in linear regression models, the residual error is interpreted as independent and identically distributed observation error.

To run this model using our package, we'll need to specify the response and predictor variables. The covariate matrix with an intercept only is a matrix of 1s. To double check, you could always look at

```
x = model.matrix(lm(Temp ~ 1))
```

Fitting the model using our function is done with this code,

```
lm_intercept = fit_stan(y = as.numeric(Temp), x = rep(1, length(Temp)),
    model_name = "regression")
```

Coarse summaries of `stanfit` objects can be examined by typing one of the following

```
lm_intercept
# this is huge
summary(lm_intercept)
```

But to get more detailed output for each parameter, you have to use the `extract()` function,

```
pars = extract(lm_intercept)
names(pars)
```

```
[1] "beta"    "sigma"   "pred"    "log_lik" "lp__"
```

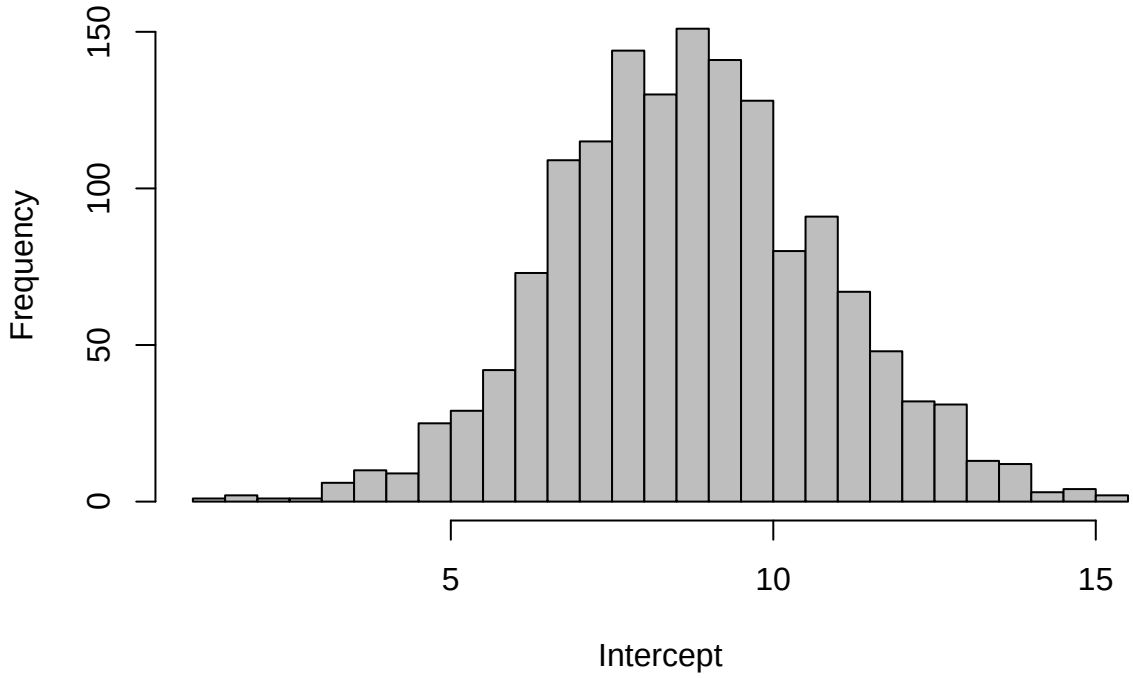
`extract()` will return the draws from the posterior for your parameters and any derived variables specified in your stan code. In this case, our model is

$$y_t = \beta \times 1 + e_t, e_t \sim N(0, \sigma)$$

so our estimated parameters are β and σ. Our stan code computed the derived variables: predicted y_t which is $\hat{y}_t = \beta \times 1$ and the log-likelihood. lp___ is the log posterior which is automatically returned.

We can then make basic plots or summaries of each of these parameters,

```
hist(pars$beta, 40, col = "grey", xlab = "Intercept", main = "")
```




```
quantile(pars$beta, c(0.025, 0.5, 0.975))
```

```
     2.5%       50%     97.5%
 4.614887  8.677668 12.915321
```

One of the other useful things we can do is look at the predicted values of our model ($\hat{y}_t = \beta \times 1$) and overlay the data. The predicted values are pars$pred.

```
plot(apply(pars$pred, 2, mean), main = "Predicted values", lwd = 2,
    ylab = "Wind", ylim = c(min(pars$pred), max(pars$pred)),
    type = "l")
lines(apply(pars$pred, 2, quantile, 0.025))
```

```
lines(apply(pars$pred, 2, quantile, 0.975))
points(Wind, col = "red")
```

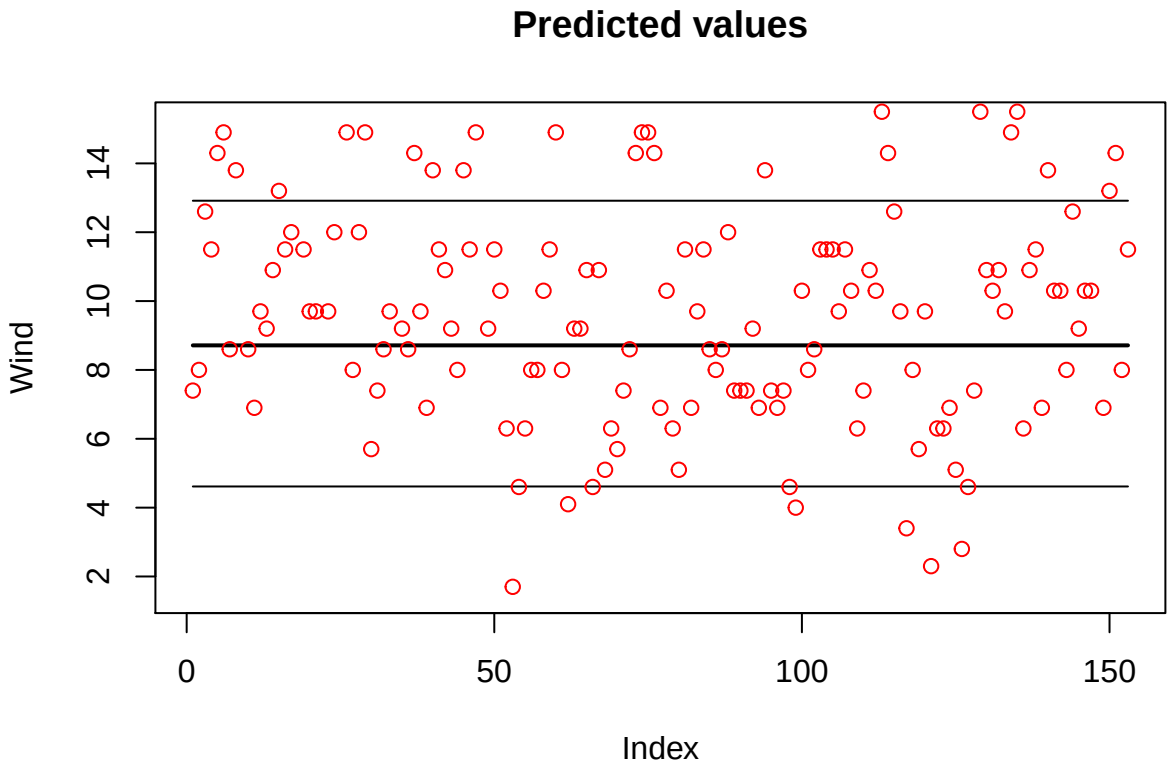


Figure 9.1: Data and predicted values for the linear regression model.

9.2.1 Burn-in and thinning

To illustrate the effects of the burn-in/warmup period and thinning, we can re-run the above model, but for just 1 MCMC chain (the default is 3).

```
lm_intercept = fit_stan(y = Temp, x = rep(1, length(Temp)), model_name = "regression",
    mcmc_list = list(n_mcmc = 1000, n_burn = 1, n_chain = 1,
        n_thin = 1))
```

Here is a plot of the time series of `beta` with one chain and no burn-in. Based on visual inspection, when does the chain converge?

```
pars = extract(lm_intercept)
plot(pars$beta)
```

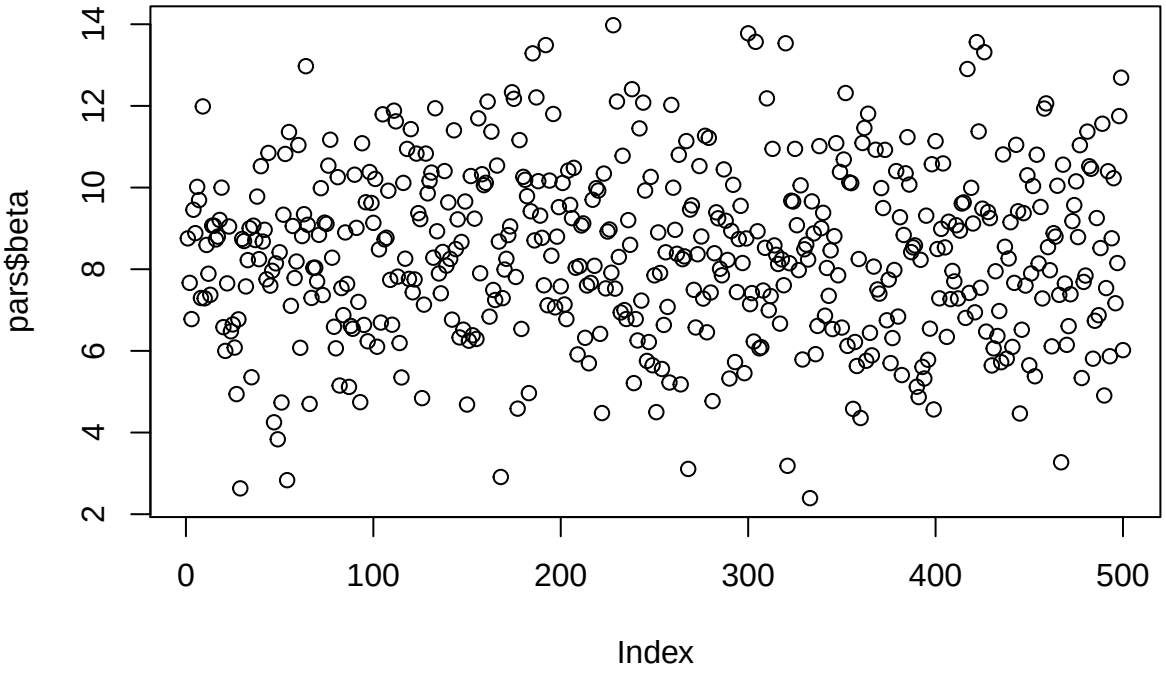


Figure 9.2: A time series of our posterior draws using one chain and no burn-in.

9.3 Linear regression with correlated errors

In our first model, the errors were independent in time. We're going to modify this to model autocorrelated errors. Autocorrelated errors are widely used in ecology and other fields – for a greater discussion, see Morris and Doak (2002) Quantitative Conservation Biology. To make the errors autocorrelated, we start by defining the error in the first time step, $e_1 = y_1 - \beta$. The expectation of Y_t in each time step is then written as

$$E[Y_t] = \beta + \phi e_{t-1}$$

In addition to affecting the expectation, the correlation parameter ϕ also affects the variance of the errors, so that

$$\sigma^2 = \psi^2 \left(1 - \phi^2\right)$$

Like in our first model, we assume that the data follows a normal likelihood (or equivalently that the residuals are normally distributed), $y_t = E[Y_t] + e_t$, or $Y_t \sim N(E[Y_t], \sigma)$. Thus, it is possible to express the subsequent deviations as $e_t = y_t - E[Y_t]$, or equivalently as $e_t = y_t - \beta - \phi e_{t-1}$.

We can fit this regression with autocorrelated errors by changing the model name to 'regression_cor'

```
lm_intercept_cor = fit_stan(y = Temp, x = rep(1, length(Temp)),
    model_name = "regression_cor", mcmc_list = list(n_mcmc = 1000,
        n_burn = 1, n_chain = 1, n_thin = 1))
```

9.4 Random walk model

All of the previous three models can be interpreted as observation error models. Switching gears, we can alternatively model error in the state of nature, creating process error models. A simple process error model that many of you may have seen before is the random walk model. In this model, the assumption is that the true state of nature (or latent states) are measured perfectly. Thus, all uncertainty is originating from process variation (for ecological problems, this is often interpreted as environmental variation). For this simple model, we'll assume that our process of interest (in this case, daily wind speed) exhibits no daily trend, but behaves as a random walk.

$$y_t = y_{t-1} + e_t$$

And the $e_t \sim N(0, \sigma)$. Remember back to the autocorrelated model (or MA(1) models) that we assumed that the errors e_t followed a random walk. In contrast, this model assumes that the errors are independent, but that the state of nature follows a random walk. Note also that this model as written doesn't include a drift term (this can be turned on / off using the `est_drift` argument).

We can fit the random walk model using argument `model_name = 'rw'` passed to the `fit_stan()` function.

```
rw = fit_stan(y = Temp, est_drift = FALSE, model_name = "rw")
```

9.5 Autoregressive models

A variation of the random walk model described previously is the autoregressive time-series model of order 1, AR(1). This model is essentially the same as the random walk model but it introduces an estimated coefficient, which we'll call ϕ. The parameter ϕ controls the degree to which the random walk reverts to the mean – when $\phi = 1$, the model is identical to the random walk, but at smaller values, the model will revert back to the mean (which in this case is zero). Also, ϕ can take on negative values, which we'll discuss more in future lectures. The math to describe the AR(1) model is:

$$y_t = \phi y_{t-1} + e_t$$

.

The `fit_stan()` function can fit higher order AR models, but for now we just want to fit an AR(1) model and make a histogram of phi.

```
ar1 = fit_stan(y = Temp, x = matrix(1, nrow = length(Temp), ncol = 1),
    model_name = "ar", est_drift = FALSE, P = 1)
```

9.6 Univariate state-space models

At this point, we've fit models with observation or process error, but we haven't tried to estimate both simultaneously. We will do so here, and introduce some new notation to describe the process model and observation model. We use the notation x_t to denote the latent state or state of nature (which is unobserved) at time t and y_t to denote the observed data. For introductory purposes, we'll make the process model autoregressive (similar to our AR(1) model),

$$x_t = \phi x_{t-1} + e_t, e_t \sim N(0, q)$$

For the process model, there are a number of ways to parameterize the first 'state', and we'll talk about this more in the class, but for the sake of this model, we'll place a vague weakly informative prior on x_1, $x_1 \sim N(0, 0.01)$.Second, we need to construct an observation model linking the estimate unseen states of nature x_t to the data y_t. For simplicitly, we'll assume that the observation errors are indepdendent and identically distributed, with no observation component. Mathematically, this model is

$$Y_t \sim N(x_t, r)$$

In the two above models, we'll refer to q as the standard deviation of the process variance and r as the standard deviation of the observation error variance

We can fit the state-space AR(1) and random walk models using the `fit_stan()` function:

```
ss_ar = fit_stan(y = Temp, est_drift = FALSE, model_name = "ss_ar")
ss_rw = fit_stan(y = Temp, est_drift = FALSE, model_name = "ss_rw")
```

9.7 Dynamic factor analysis

First load the plankton dataset from the MARSS package.

```
library(MARSS)
data(lakeWAplankton)
# we want lakeWAplanktonTrans, which has been transformed so
# the 0s are replaced with NAs and the data z-scored
dat = lakeWAplanktonTrans
# use only the 10 years from 1980-1989
plankdat = dat[dat[, "Year"] >= 1980 & dat[, "Year"] < 1990,
    ]
# create vector of phytoplankton group names
phytoplankton = c("Cryptomonas", "Diatoms", "Greens", "Unicells",
    "Other.algae")
# get only the phytoplankton
dat.spp.1980 = t(plankdat[, phytoplankton])
# z-score the data since we subsetted time
dat.spp.1980 = dat.spp.1980 - apply(dat.spp.1980, 1, mean, na.rm = TRUE)
dat.spp.1980 = dat.spp.1980/sqrt(apply(dat.spp.1980, 1, var,
    na.rm = TRUE))
# check our z-score
apply(dat.spp.1980, 1, mean, na.rm = TRUE)
```

```
  Cryptomonas       Diatoms        Greens      Unicells   Other.algae
 4.951913e-17 -1.337183e-17  3.737694e-18 -5.276451e-18  4.365269e-18
```

```
apply(dat.spp.1980, 1, var, na.rm = TRUE)
```

```
Cryptomonas     Diatoms      Greens    Unicells Other.algae
          1           1           1           1           1
```

Plot the data.

```
# make into ts since easier to plot
dat.ts = ts(t(dat.spp.1980), frequency = 12, start = c(1980,
    1))
par(mfrow = c(3, 2), mar = c(2, 2, 2, 2))
```

```
for (i in 1:5) plot(dat.ts[, i], type = "b", main = colnames(dat.ts)[i],
    col = "blue", pch = 16)
```

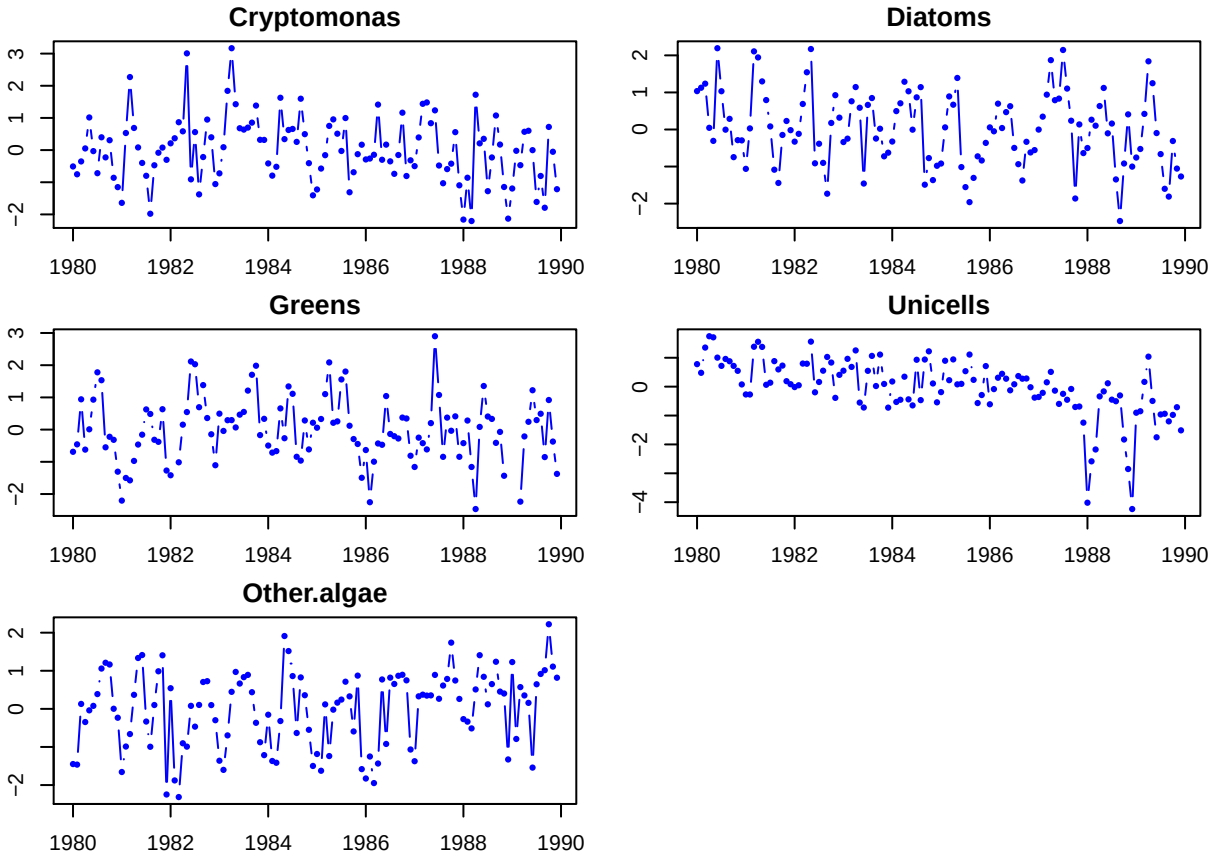


Figure 9.3: Phytoplankton data.

Run a 3 trend model on these data.

```
mod_3 = fit_dfa(y = dat.spp.1980, num_trends = 3)
```

Rotate the estimated trends and look at what it produces.

```
rot = rotate_trends(mod_3)
names(rot)
```

```
[1] "Z_rot"        "trends"       "Z_rot_mean"   "trends_mean"
[5] "trends_lower" "trends_upper"
```

Plot the estimate of the trends.

```
matplot(t(rot$trends_mean), type = "l", lwd = 2, ylab = "mean trend")
```

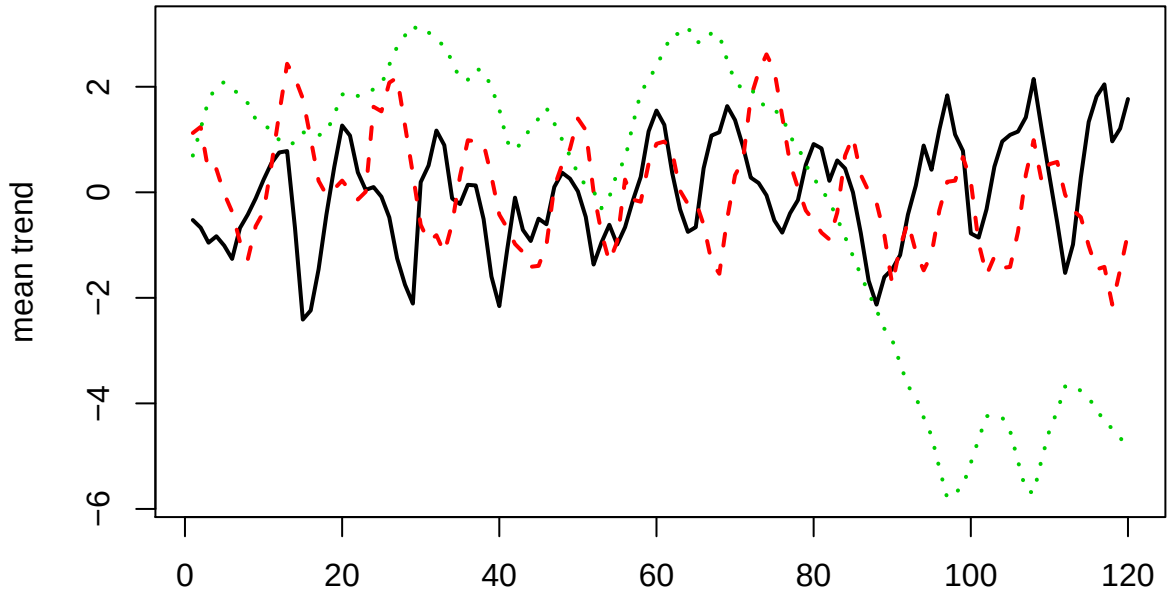


Figure 9.4: Trends.

9.7.1 Using leave one out cross-validation to select models

We will fit multiple DFA with different numbers of trends and use leave one out (LOO) cross-validation to choose the best model.

```
mod_1 = fit_dfa(y = dat.spp.1980, num_trends = 1)
mod_2 = fit_dfa(y = dat.spp.1980, num_trends = 2)
mod_3 = fit_dfa(y = dat.spp.1980, num_trends = 3)
mod_4 = fit_dfa(y = dat.spp.1980, num_trends = 4)
mod_5 = fit_dfa(y = dat.spp.1980, num_trends = 5)
```

We will compute the Leave One Out Information Criterion (LOOIC) using the loo package. Like AIC, lower is better.

```
library(loo)
loo(extract_log_lik(mod_1))$looic
```

```
[1] 1598.7
```

Table of the LOOIC values:

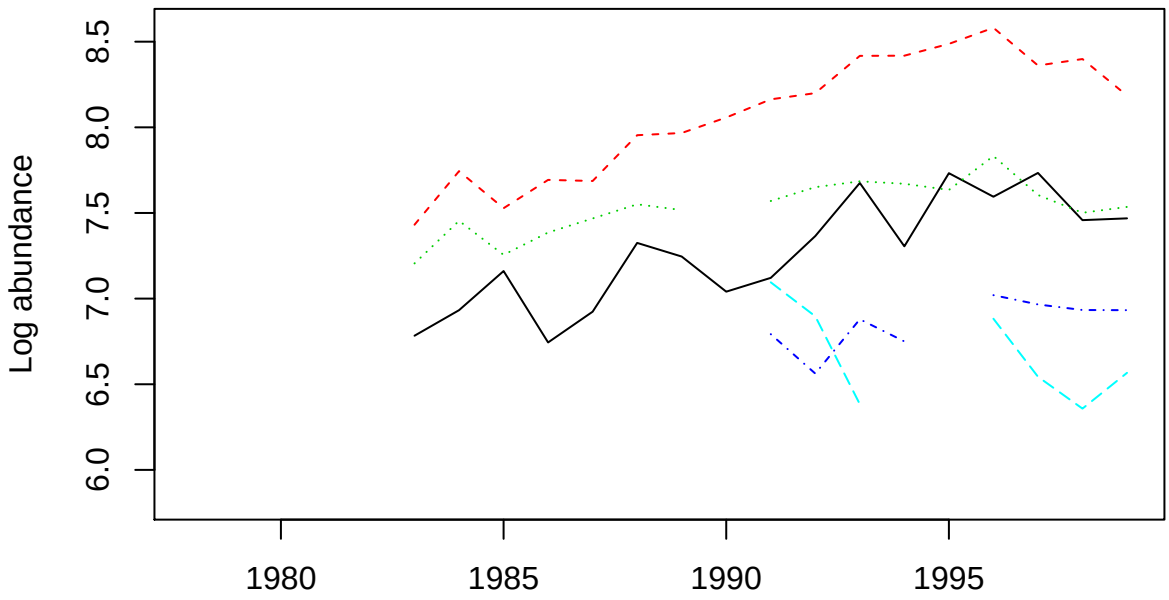
```
r   looics = c(loo(extract_log_lik(mod_1))$looic, loo(extract_log_lik(mod_2))$looic,
loo(extract_log_lik(mod_3))$looic, loo(extract_log_lik(mod_4))$looic,
loo(extract_log_lik(mod_5))$looic)   looic.table = data.frame(trends = 1:5,
LOOIC = looics)   looic.table
```

```
trends    LOOIC   1      1 1598.700   2      2 1530.603   3      3 1471.334
4      4 1454.646   5      5 1447.238
```

9.8 Uncertainty intervals on states

We will look at the effect of missing data on the uncertainty intervals on estimates states using a DFA on the harbor seal dataset.

```
data("harborSealWA")
# the first column is year
matplot(harborSealWA[, 1], harborSealWA[, -1], type = "l", ylab = "Log abundance",
    xlab = "")
```



Assume they are all observing a single trend.

```
seal.mod = fit_dfa(y = t(harborSealWA[, -1]), num_trends = 1)
```

```
pars = extract(seal.mod)
```

```
pred_mean = c(apply(pars$x, c(2, 3), mean))
pred_lo = c(apply(pars$x, c(2, 3), quantile, 0.025))
pred_hi = c(apply(pars$x, c(2, 3), quantile, 0.975))

plot(pred_mean, type = "l", lwd = 3, ylim = range(c(pred_mean,
    pred_lo, pred_hi)), main = "Trend")
lines(pred_lo)
lines(pred_hi)
```

9.9 Problems

1. By adapting the code in Section 9.2, fit a regression model that includes the intercept and a slope, modeling the effect of Wind. What is the mean wind effect you estimate?

2. Using the results from the linear regression model fit with no burn-in (Section 9.2.1),

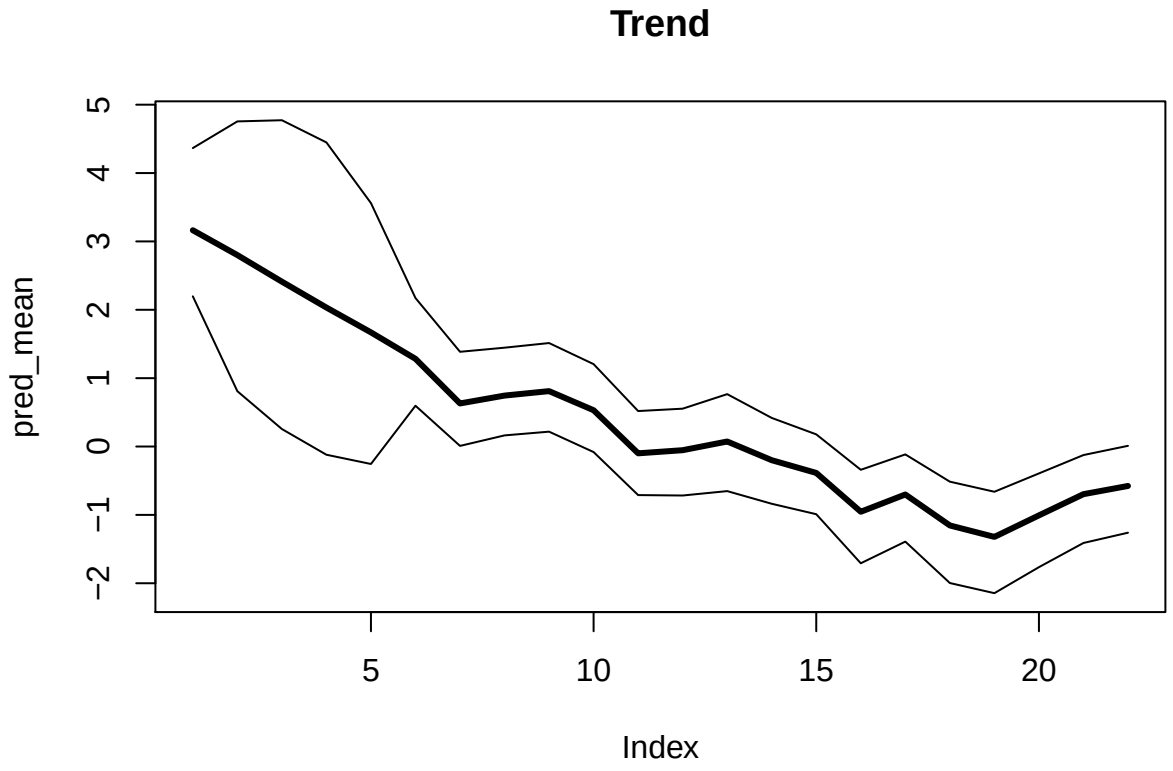


Figure 9.5: Estimated states and 95% credible intervals.

calculate the ACF of the `beta` time series using `acf()`. Would thinning more be appropriate? How much?

3. Using the fit of the random walk model to the temperature data (Section 9.4), plot the predicted values (states) and 95% CIs.
4. To see the effect of this increased flexibility in estimating the autocorrelation, make a plot of the predictions from the AR(1) model (Section 9.5 and the RW model (9.4).
5. Fit the univariate state-space model (Section 9.6) with and without the autoregressive parameter ϕ and compare the estimated process and observation error variances. Recall that AR(1) without the ϕ parameter is a random walk.

Bibliography

Holmes, E. E., Ward, E. J., and Scheuerell, M. D. (2014). Analysis of multivariate time-series using the marss package. Technical report, Northwest Fisheries Science Center, Seattle, WA.

Jorgensen, J. C., Ward, E. J., Scheuerell, M. D., and Zabel, R. W. (2016). Assessing spatial covariance among time series of abundance. Ecology and Evolution, 6:2472–2485.

Lamon, E. I., Carpenter, S., and Stow, C. (1998). Forecasting pcb concentrations in lake michigan salmonids: a dynamic linear model approach. Ecological Applications, 8:659–668.

Lisi, P. J., Schindler, D. E., Cline, T. J., Scheuerell, M. D., and Walsh, P. B. (2015). Watershed geomorphology and snowmelt control stream thermal sensitivity to air temperature. Geophysical Research Letters, 42(9):3380–3388.

Ohlberger, J., Scheuerell, M. D., and Schindler, D. E. (2016). Population coherence and environmental impacts across spatial scales: a case study of Chinook salmon. Ecosphere, 7:e01333.

Petris, G., Petrone, S., and Campagnoli, P. (2009). Dynamic Linear Models with R. Use R! Springer.

Pole, A., West, M., and Harrison, J. (1994). Applied Bayesian forecasting and time series analysis. Chapman and Hall, New York.

Scheuerell, M. D. and Williams, J. G. (2005). Forecasting climate induced changes in the survival of snake river spring/summer chinook salmon (oncorhynchus tshawytscha). Fisheries Oceanography, 14(6):448–457.

Stachura, M. M., Mantua, N. J., and Scheuerell, M. D. (2014). Oceanographic influences on patterns in North Pacific salmon abundance. Canadian Journal of Fisheries and Aquatic Sciences, 71(2):226–235.

Zuur, A. F., Fryer, R. J., Jolliffe, I. T., Dekker, R., and Beukema, J. J. (2003). Estimating common trends in multivariate time series using dynamic factor analysis. Environmetrics, 14(7):665–685.