

ADMB Getting Started Guide

DRAFT FOR REVIEW

October 2009

Contents

1	What is AD Model Builder?	5
1.1	Features	7
1.2	About this Document	8
1.3	Additional Resources	8
2	Installation	11
2.1	System requirements	11
2.2	Installing ADMB	12
2.2.1	Windows	12
2.2.2	Linux	16
3	Working with AD Model Builder	19
3.1	Opening AD Model Builder	19
3.2	Overview: From question to result	20
4	Creating a program: The template	23
4.1	Data section (and data file)	23
4.2	Parameter section (and initial parameter values file)	28
4.3	Procedure section	32
4.4	Non-required template sections	33
5	Compiling and running a program	39
5.1	Compiling a program	39
5.2	Running a program and command-line options	40
5.3	Errors, debugging, and memory management	41
6	The results: AD Model Builder output files	45
6.1	Parameter estimate file (.par)	46
6.2	Standard deviation report file (.std)	47
6.3	Correlation matrix file (.cor)	47

6.4	User-defined output file (.rep)	48
6.5	Outputting results for R	48
7	Examples	51
7.1	Example 1: Least-squares regression	51
7.1.1	Using sum of squares	52
7.1.2	Using matrix algebra with sum of squares	54
7.1.3	Standard Deviation Report (.std)	55
7.2	Example 2: Nonlinear regression with MLE: Fitting a Von Bertalanffy growth curve to data	57
7.2.1	Using phases and bounds	60
7.2.2	Plotting the results using R	63
7.3	Example 3: A simple fisheries model: estimating parameters and uncertainty	67
7.3.1	Maximum likelihood estimate (fish.tpl)	69
7.3.2	Likelihood Profile and Bayesian posterior analysis . .	72
7.3.3	Report: Profile likelihood report (.plt)	77
7.3.4	Report: Markov Chain Monte Carlo (MCMC) report (.hst)	79
7.4	Example 4: Simulation testing	80
7.4.1	Simulating data: Generating random numbers . . .	80
7.4.2	Simulation testing: Estimating plant yield per pot from pot density	82
8	Useful operators and functions	87
8.1	Useful ADMB Functions	87
8.2	Useful Vector Operations	88
8.3	Useful Matrix Operations	90

Chapter 1

What is AD Model Builder?

AD Model Builder (ADMB) is a free software package designed to help ecologists and others develop and fit complex nonlinear statistical models (Figure 1.1). Offering support for numerous statistical techniques, such as simulation analysis, likelihood profiles, Bayesian posterior analysis using a Markov Chain Monte Carlo (MCMC) algorithm, numerical integration, and mixed-effect modeling, the software is well suited for computationally intensive applications. For example, ADMB has been used to fit complex nonlinear models with thousands of parameters to multiple types of data, as well as to fit nonlinear models with fewer parameters to hundreds of thousands of data points. With ADMB, it is also relatively simple to create and process nonlinear mixed models (i.e., models that contain both fixed and random effects). ADMB combines a flexible mathematical modeling language (built on C++) with a powerful function minimizer (based on Automatic Differentiation) that is substantially faster and more stable than traditional minimizers, which rely on finite difference approximations. Automatic differentiation can make the difference between waiting hours and waiting seconds for a converging model fit, and its use in ADMB provides fast, efficient and robust numerical parameter estimation.

Users interact with the application by providing a simple description of the desired statistical model in an ADMB template. Once the specification is complete (and compiled and run using a few simple commands), ADMB fits the model to the data and reports the results automatically. ADMB does not produce any graphical output, but the output files can be easily brought into standard packages like Excel or R for analysis.

The software's template-based interface allows scientists to specify models without the need for an in-depth knowledge of C or C++ (though some

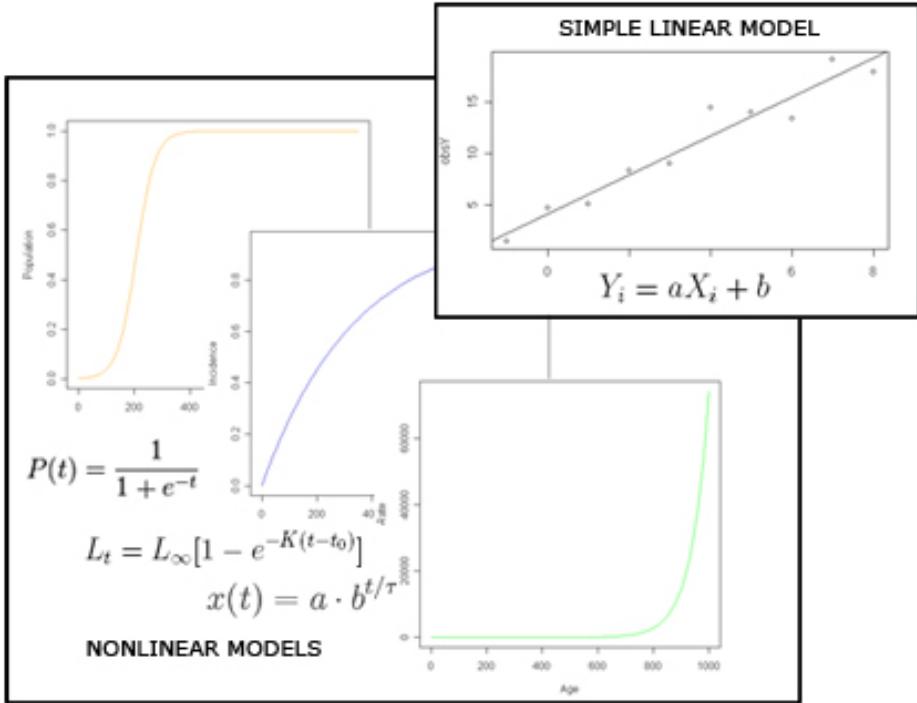


Figure 1.1: Examples of Linear and Nonlinear models

knowledge of C syntax and programming is required, and users must have a coherent model and a data set in mind before creating a template). The template interface also allows users to construct any desired model (rather than selecting one from a set of predefined models, which is standard in other statistical environments, such as R). Note that ADMB users are required to construct a model; the software does not provide a predefined list of standard models to choose from. In addition, experienced C++ programmers can create their own C++ libraries, extending the functionality of ADMB to suit their needs.

The software makes it simple to deal with recurring difficulties in nonlinear modeling, such as restricting parameter values (i.e., setting bounds), optimizing in a stepwise manner (i.e., using phases), and producing standard deviation reports for estimated parameters. ADMB is also very useful in simulation analysis, as the reduced estimation time provided by ADMB can significantly reduce the amount of time required to evaluate the simu-

lation.

ADMB was developed by David Fournier in the early 1990s and is rooted in one of the original C++ implementations of reverse mode automatic differentiation, Fournier's AUTODIF library. For more information about AUTODIF, please see the AUTODIF manual.

1.1 Features

AD Model Builder's features include:

- A flexible template interface, which makes it easy to input and output data from the model, set up the parameters to estimate, and set up an objective function to minimize. Adding additional estimable parameters or converting fixed parameters into estimable parameters is a simple process.
- A set of C++ libraries that can be used in conjunction with user-created libraries to extend ADMB functionality
- An efficient and stable function minimizer that uses automatic differentiation for exact derivatives
- Support for matrix algebra (e.g., vector-matrix arithmetic and vectorized operations for common mathematical functions), which uses less computation time and memory than traditional programmatic loops
- Support for nonlinear, mixed effects models
- Support for simulation analysis
- Likelihood profile generation
- Automated estimates of the variance-covariance matrix, and (optionally) the variance of any model variable
- An MCMC algorithm (Metropolis-Hastings) for Bayesian integration that uses efficient jumping rules based on the estimated variance-covariance matrix
- Support for parallel processing, which allows a single model to be estimated using multiple processes
- Support for bounds to restrict the range of possible parameter values

- Support for using phases to fit a model
- Support for high-dimensional and ragged arrays
- Random number generation
- Random effects parameters (implemented using Laplace’s approximation)
- A “safe mode” compiling option for bounds checking
- Support for Dynamic Link Libraries (DLLs) with other Windows programs (e.g., R, Excel, Visual Basic). For example, an ADMB program can be created and called from R as though it were part of the R language.

1.2 About this Document

This is an introductory guide designed to introduce users to AD Model Builder. This document can be downloaded from <https://code.nceas.ucsb.edu/code/projects/admb-docs/>. Templates and associated data and parameter files for all of the examples can be downloaded from <https://code.nceas.ucsb.edu/code/projects/admb-docs/examples/>

1.3 Additional Resources

To learn more about AD Model Builder, please see the following references:

- **AD Model Builder Manual**
<http://admb-project.googlecode.com/files/admb.pdf>
The AD Model Builder manual contains details about installing and using AD Model Builder for advanced statistical applications. The manual highlights a number of example programs from various fields, including chemical engineering, natural resource modeling, and financial modeling.
- **AUTODIF Library Manual**
<http://admb-project.googlecode.com/files/autodif.pdf>

AUTODIF is fully integrated into AD Model Builder, and users automatically take advantage of its powerful automatic differentiation functionality to lend speed and stability to statistical analyses. This manual contains more information about the AUTODIF C++ libraries, including details about the AUTODIF function minimizing routines, working with random numbers, and supported operations and functions. The manual also includes a number of useful examples.

- **Random effects in AD Model Builder: ADMB-RE user guide**

<http://admb-project.googlecode.com/files/autodif.pdf>

The ADMB-RE user's guide provides a concise introduction to random effects modeling in AD Model Builder. The guide includes background information about random effects modeling as well as a number of useful examples. Additional examples can be found in the online example collection: <http://www.otter-rsch.com/admubre/examples.html>

- **Mailing List**

To participate in discussions about problems and solutions using ADMB, please join the ADMB User Mailing List:

<http://lists.admb-project.org/mailman/listinfo/users>

Archives of past discussions are available at:

<http://lists.admb-project.org/pipermail/users/>

- **Support**

For general questions about the ADMB project, please contact support@admb-project.org. Additional information, including recent ADMB news and links to community resources can be found at:

<http://admb-project.org/community>.

Chapter 2

Installation

To download and install ADMB, please go to the ADMB Download page:
<http://admb-project.org/downloads>.

2.1 System requirements

ADMB runs on both Windows and Linux systems. A Mac version is currently under development—a release candidate is available from the ADMB download page.

Program Requirements:

- A C++ compiler (for Windows: MinGW GCC, Visual C+, or Borland 5.5.1) and the Make application. Be sure to download the correct version of ADMB for your platform and compiler. The Windows installer is bundled with a MinGW compiler and we recommend downloading and installing this version of ADMB if you do not have a compiler installed already.
- A text editor for working with templates. Though you can use Notepad or Wordpad, we recommend that you use a more robust (free) text editor, such as ConTEXT or Crimson (for Windows) or jEdit (for Window, Linux, Mac). These editors support syntax highlighting and other useful coding features. For a list of recommended editors, please see <http://admb-project.org/community/editing-tools>

Hardware Requirements:

- ADMB includes about 4 MB of C++ libraries. Hard disk requirements depend on the C++ compiler used.
- Memory requirements are dependent on size of model.

2.2 Installing ADMB

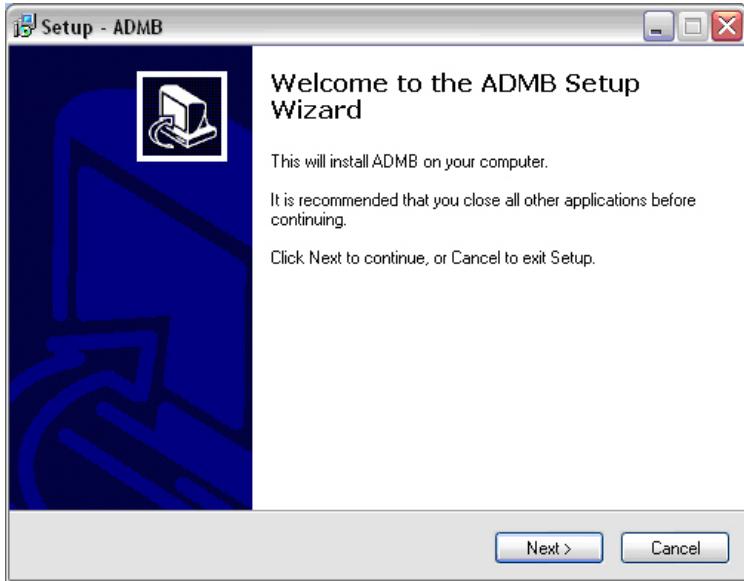
AD Model Builder is available for Windows and Linux, and a beta version is also available for the Mac. Please see the admb-project.org site for more information about downloading and installing ADMB for the Mac.

2.2.1 Windows

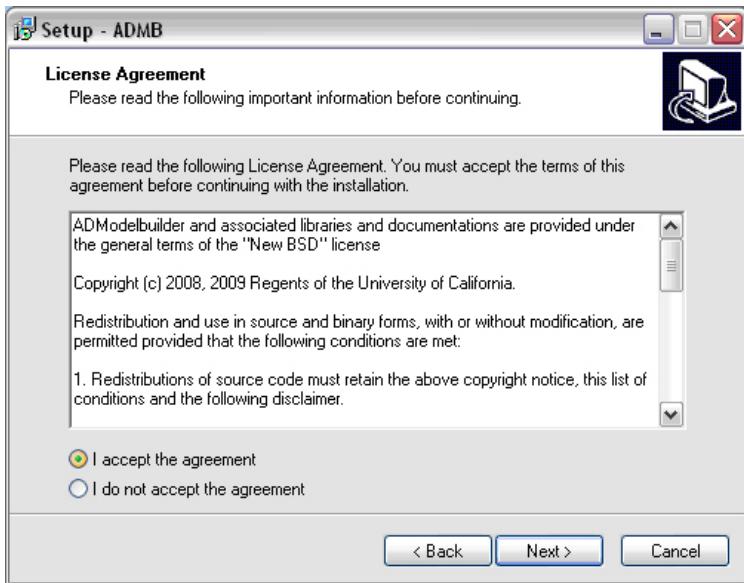
In this section, we provide instructions for downloading and installing ADMB with MinGW, which uses the free gcc and g++ compilers. We recommend that you use this configuration, unless you already like and use the Microsoft or Borland compilers. Note that the installation file includes the MinGW compiler and the Make application, which are required to run ADMB. If you already have a compiler installed, you may wish to install one of the binary versions of ADMB. See the admb-project.org site for more information.

The following instructions are for the recommended ADMB and MinGW installation:

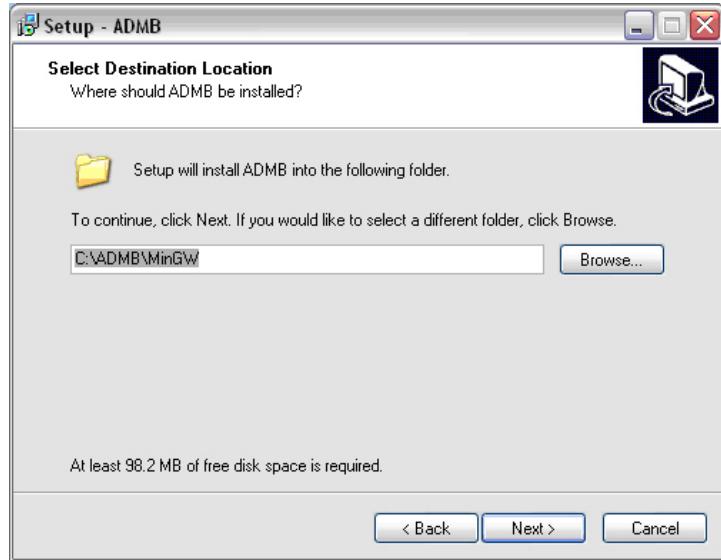
1. Download and save the ADMB-MinGW installer (23.2 MB) that includes the compiler and tools. This version is found under “Window (32 bit only)” on the download page: <http://admb-project.org/downloads>. The installation wizard will install all the necessary tools for running ADMB.
2. Double-click the downloaded installer and click “Run” to open the Setup wizard.



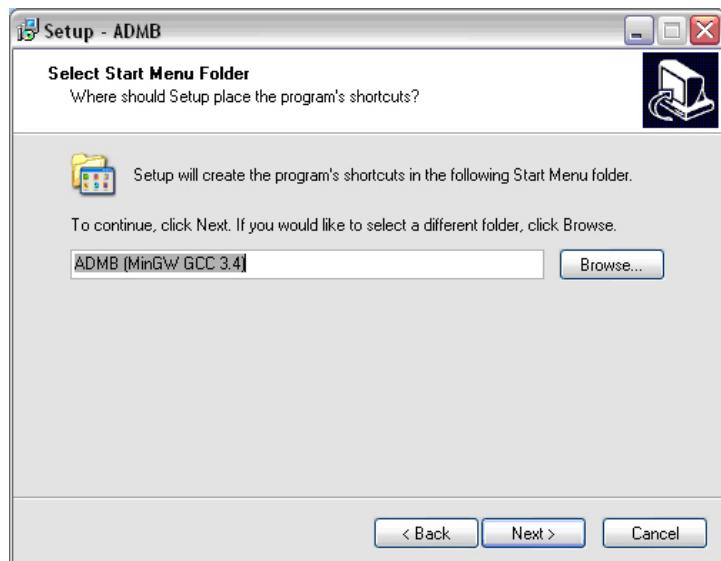
3. Accept the license agreement and click "Next".



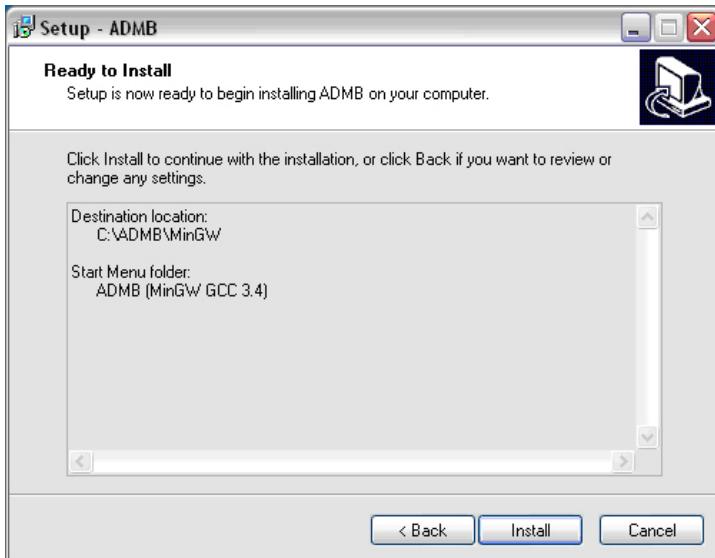
4. Chose to install ADMB into the default directory, "C:\ADMB\MinGW"



5. Chose to create a program shortcut “ADMB (MinGW GCC 3.4)” in the Start Menu Folder.



6. Review the installation settings and click “Install” to install ADMB.



The wizard will alert you when the installation is complete. Click “Finish” to exit the wizard.

7. Check that the installation is working correctly. From the Start menu, select “ADMB (MinGW)” and then “ADMB Command Prompt (MinGW)”



8. In the ADMB Command Prompt window, type “make” to build and run a suite of ADMB tests.



9. If the run is successful, the last two lines of output should appear as below:



If you encounter any problems, please send an exact copy of your error message to “help@nceas.ucsb.edu”. Please include information about your platform, operating system, and which version of ADMB you are using.

2.2.2 Linux

ADMB for Linux systems uses the GNU open source gcc compiler, which is used to build ADMB programs. To install ADMB on a Linux system:

1. Download and extract the ADMB Linux binaries from <http://admb-project.org/downloads>. Note: to determine which gcc you have, type:
`gcc --version`
at the command line.
2. Open a bash shell and navigate to the directory into which you extracted the ADMB code (e.g., `~/admb`):

```
$ cd ~/admb
```

3. Set the ADMB home directory with the following command

```
$ export ADMB_HOME=~/admb
```

4. Add the ADMB bin directory to the \$PATH:

```
$ export PATH=$ADMB_HOME/bin:$PATH
```

5. Navigate to the ADMB Home directory:

```
cd $ADMB_HOME
```

6. Type “make” to build and run a suite of ADMB test analyses.

```
$ make
```

Note that the analyses may take several minutes to compile and run.

If you encounter any problems, please send an exact copy of your error message to “help@nceas.ucsb.edu”. Please include information about your platform, operating system, and which version of ADMB you are using.

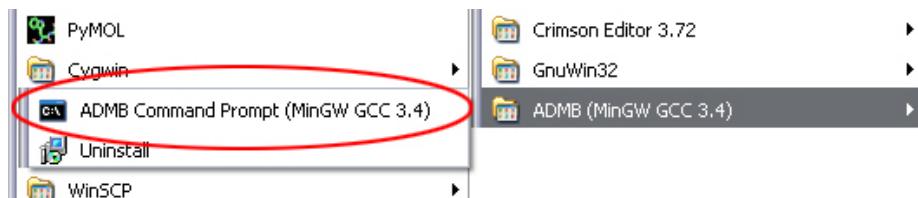
Chapter 3

Working with AD Model Builder

This chapter contains instructions for opening an installed version of AD Model Builder and a high level overview of how the application works.

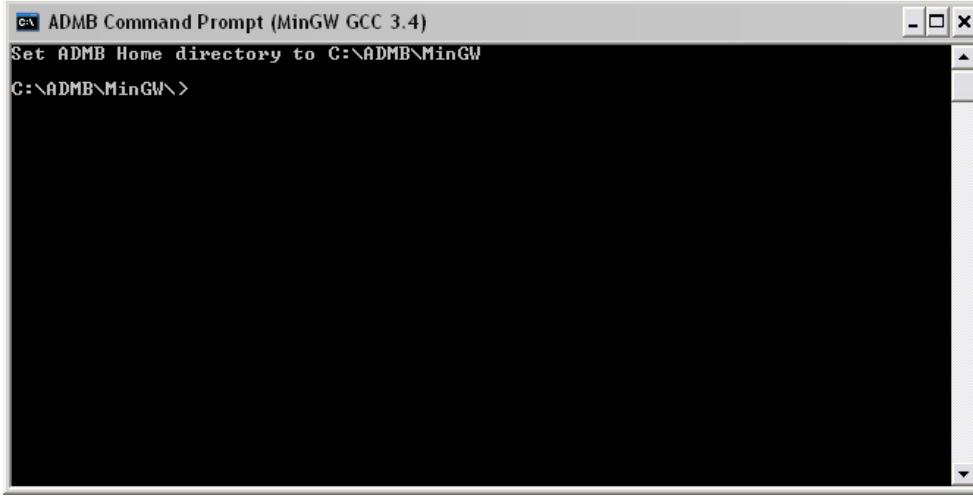
3.1 Opening AD Model Builder

To open AD Model Builder on a Windows system, select ADMB (MinGW GCC 3.4) from the Start menu and then “ADMB Command Prompt (MinGW GCC 3.4)”. Note that the menu items may be named differently depending on the C compiler installed.



On a Linux system, open the ADMB program from the command line.

An ADMB Command Prompt window opens:



By default, you should be in the ADMB MinGW directory. To run a simple example on a Windows system, navigate to the "examples/admb/simple" directory and type "simple" at the command prompt:

A screenshot of a Windows command prompt window titled "ADMB Command Prompt (MinGW GCC 3.4)". The window shows the following sequence of commands:

```
Set ADMB Home directory to C:\ADMB\MinGW
C:\ADMB\MinGW\examples\admb>ls
buscycle chem-eng forest pella-t simple vol
catage finance Makefile robreg truncreg

C:\ADMB\MinGW\examples\admb>cd simple
C:\ADMB\MinGW\examples\admb\simple>simple

Initial statistics: 2 variables; iteration 0; function evaluation 0
Function value 3.6493579e+001; maximum gradient component mag -3.6127e+000
Var Value Gradient !Var Value Gradient !Var Value Gradient
  1  0.00000 -3.61269e+000 :  2  0.00000 -7.27814e-001 :

- final statistics:
2 variables; iteration 7; function evaluation 19
Function value 1.4964e+001; maximum gradient component mag -7.0014e-005
Exit code = 1; converge criter 1.0000e-004
Var Value Gradient !Var Value Gradient !Var Value Gradient
  1  1.90909 -7.00140e-005 :  2  4.07818 -2.08982e-005 :

Estimating row 1 out of 2 for hessian
Estimating row 2 out of 2 for hessian

C:\ADMB\MinGW\examples\admb\simple>
```

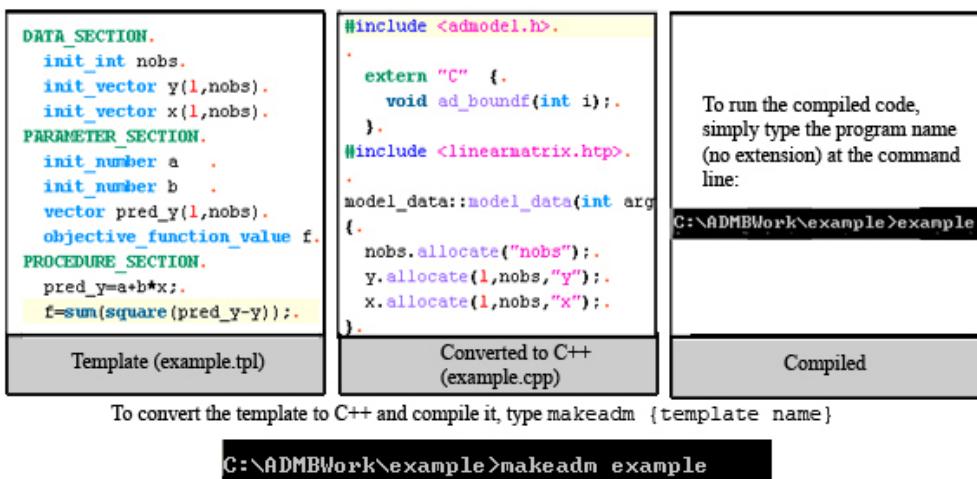
To run the 'simple' program on a Linux system, type './simple' at the command line.

3.2 Overview: From question to result

If you have a data set and a question about it (how well the data fits a model, for example), you are ready to begin using AD Model Builder. By

using command-line tools and a template file created with a standard text editor, users specify an analytical model, read in data and initial parameter values, and compile and execute a program that performs the analysis. AD Model Builder automatically calculates and outputs the results.

The template file is designed to simplify the process of specifying a model. Instead of writing a C or C++ program to carry out the required calculations, users simply supply data and declare the parameters that should be estimated using straightforward ADMB conventions, and write code that expresses the mathematical model. Templates are designated with a .tpl extension. Once finished, the template is converted to true C++ code, compiled, and linked to the ADMB libraries.



Data and initial parameter values are provided via text files (designated with .dat and .pin filename extensions, respectively). By default, AD Model Builder looks for input files that match the program name. For example, when running a program called ‘rabbits.exe’ (or ‘rabbits’, on a Linux system), AD Model Builder automatically looks for data in rabbits.dat and parameter values in rabbits.pin. This default behavior can be overridden with command-line options.

After a program has executed, AD Model Builder writes the parameter estimates, standard errors, a correlation matrix, and any user-specified reports to output files, which are saved in the program directory. Output files are named for the creating program. For a program called rabbits.exe (or rabbits, on a Linux system), the parameter estimates can be found in rab-

bits.par, the correlation matrix in rabbits.cor and any user-specified reports in rabbits.rep.

Chapter 4

Creating a program: The template

The template simplifies the development of statistical analysis by hiding many aspects of C++ programming. Using a template, users must only be familiar with some of the simpler aspects of C or C++ syntax to create and execute an ADMB program (see Section 8 for an overview of useful syntax, functions, and operators).

The template itself is divided into several sections. For now, you need only be concerned with the three required sections: DATA_SECTION, PARAMETER_SECTION, and PROCEDURE_SECTION, which we will look at in more depth in this chapter.

As you work with the template (Figure 4.1), keep in mind that the file will be translated into C++ code and then compiled.

Be careful to adhere to the following syntax requirements:

1. Indent using spaces (not tabs)
2. Template section names should be flush with the file margin
3. Use comments to describe and clarify the template. Comments are designated with a “//” and may be used throughout the template.

4.1 Data section (and data file)

The data section is where you describe the structure of the data used by the model. In general, all observational data read in from the data file and all fixed values (i.e., values that are constant in the model and require no

```

//this program estimates the slope and the
//intercept of a linear model (e.g., y=b+ax)
//data are found in the Linear.dat file,
// and consist of 10 observed x and y values

DATA_SECTION
    init_int nobs
    init_vector y(1,nobs)
    init_vector x(1,nobs)
PARAMETER_SECTION
    init_number a
    init_number b
    objective_function_value f
PROCEDURE_SECTION
    f=0;
    for (int i=1;i<=nobs; i++) //a for loop. Indent nested code for clarity
    {
        f+=pow(y(i)-(b+a*x(i)),2);
    }

```

Figure 4.1: A template file displayed with syntax coloring. Template files have a .tpl filename extension.

derivative calculations) should be defined in the data section. You cannot use a data object in the program until it has been defined.

Data can consist of integers or floating point numbers, which can be grouped into one-dimensional arrays (e.g., a vector of measurements) and two-dimensional arrays (e.g., a list of years with corresponding indices). In addition, floating point numbers can be grouped into three and four-dimensional arrays (Table 4.1). ADMB also supports ragged arrays.

Any data object prefixed with “init_” (e.g., “init_number nobs” or “init_vector biomass(1,nobs)”) will be read in from the data file. Objects prefaced with “init_” are read in from a data file in the order in which they are declared. For example, if you define an integer (e.g., a number of observations) and a vector containing observed population values, you must ensure that the data file contains data in that order (Figure 4.2). Notice that once a data object has been read in, its value can be used to describe a subsequent data object. In the example in Figure 4.2, the number of observations (`nobs`) is used to describe the size of the population-data vector.

Any data type without the `init_` prefix is not initialized from the data file.

Data files should not contain tabs and should include comments to clarify the meaning of the data. Note that comments in data files are designated

<pre>DATA_SECTION init_int nobs init_vector pop(1,nobs)</pre>	<pre># number of observations (nobs) 10 # observed population (pop) 20 24 34 17 20 23 25 18 22 18</pre>
DATA_SECTION of template file (pop.tpl)	Data file (pop.dat).

Figure 4.2: The DATA_SECTION of a template (left) and corresponding data file (right).

with the `#` syntax. If you are creating a .dat file from Excel, we recommend that you save the spreadsheet as a “Formatted text (space delimited)” file (.prn), before converting it to a .dat file.

By default, ADMB will look for a data file with the same name as the program. This default behavior can be overridden by adding a C command to the data section, or by using the ‘-ind’ command-line option when running the program. We will look at examples of both options later in this guide.

Table 4.1: Types of data objects in ADMB programs

Data object	Definition and example
<code>int</code>	An integer. Example: <code>init_int nobs</code> In this example, “nobs” is the name of an integer data object. Because the “init_” prefix is used, the value is read from the data file.
<code>number</code>	Floating point number. Example: <code>init_number temp</code> In this example, “temp” is the name of a floating point number, the value of which is read from the data file.
<code>ivector</code> (cont’d)	One-dimensional array (i.e., vector) of integers. Example: <code>init_ivector years(1,5)</code> The vector “years” has an initial index of 1 and a length of 5 (e.g., “5,4,3,2,1”, where the first element, “5”, has an index of 1, the second element, “4”, of 2, etc.). Often, vector lengths are defined using values that are read from a data file, such as a number of observations. For example:
Continued on next page	

Table 4.1 – continued from previous page

Data object	Definition and example
	<pre>init_ivector biomass(1,nobs)</pre> <p>In this case, the value ‘nobs’ must be defined before the ‘biomass’ ivector is declared. Please note that vector definitions cannot contain spaces.</p> <pre>vector error(1,NGROUPS + 1); //error vector correct(1,NGROUPS+1); //correct</pre>
vector	One-dimensional array (i.e., vector) of numbers. For examples, please see ivector, above.
imatrix	<p>Two-dimensional array (i.e., matrix) of integers. Example:</p> <pre>init_matrix dat(1,nobs,1,2)</pre> <p>In this example, ADMB reads a two-dimensional array of integers (e.g., age and corresponding weight data) from the data file. 1 is the initial index of the first dimension and nobs specifies the length of the first dimension (e.g., the number of observations). 1 and 2 identify the initial index and length of the second dimension (the number of columns of data). See Figure 4.5 at the end of the Chapter for an example. For information about extracting a single column of data from a two-dimensional array, see Figure 4.3.</p>
matrix	Two-dimensional array (i.e., matrix) of numbers. For examples, please see imatrix .
3darray	Three-dimensional array of numbers. See Figure 4.6 at the end of this Chapter for an example.
4darray	Four dimensional array of numbers
Ragged array	See Figure 4.7 at the end of this Chapter for an example

Note: Variable names are case-sensitive and must start with an alphabetical character. We recommend that you choose descriptive, short names (e.g., length, mass, pop, nobs). You may use an underscore if you wish (e.g., fish_mass). Do not use any words reserved by C++ (e.g., catch, if, else).

Tips and Tricks: Extracting a column of data from a data table

Extracting a column of data from a data table (e.g., a matrix) can be done in the template's DATA_SECTION using the `column()` function. Note that any standard C++ command can be used in the DATA_SECTION and/or the PARAMETER_SECTION as long as it is preceded by `!!` and concluded with a semi-colon, e.g.:

```
!!a=column(mytable,1);
```

To use the `column()` function, first read in the data from the data file (Figure 4.3, Step 1). Define a vector to hold each extracted column of data

```
DATA_SECTION
init_int nobs

//declare a two-dimensional array for the
//age/length data
init_matrix fish(1,nobs,1,2) ← STEP 1

//declare vectors to hold length (L) and age (a) data.
// Note that these data will be extracted
// from data that has already been read into ADMB
//(in fish matrix, defined above), and so
// do not require the init_ prefix
vector L(1,nobs) ← STEP 2
vector a(1,nobs) ← STEP 2

//c code used to extract column 1 and column 2
//of the fish data
!!a=column(fish,1); ← STEP 3
!!L=column(fish,2);
```

Template DATA_SECTION (fish.tpl)

```
# This data set contains ten
# fish observations. The age and
# length of each observed fish
# was recorded.

# Number of observations(nobs)
10
# Age and length of each fish (fish)
2 1
2 3
2 4
2 4
3 3
3 4
3 5
4 6
4 9
6 10
```

Data file (fish.dat)

Figure 4.3: Using C code in the DATA_SECTION to extract columns of data.

(Step 2). Because the vector will hold data that has already been read into AD Model Builder, you should not use the `init_` prefix when defining it. Once the vectors have been defined, use the `!!` syntax to insert the C++ command into the template (Step 3). In this example, the value of the '`a`' vector becomes the age data (2,2,2,2,3,3,3,4,4,6) and the value of the '`L`' vector the length data (1,3,4,4,3,4,5,6,9,10).

Note that you can also manipulate data using a LOCAL_CALCS section inside the DATA_SECTION. Standard C++ commands and syntax are used in the LOCAL_CALCS section. For an example, please see the ADMB User

Manual.

4.2 Parameter section (and initial parameter values file)

The PARAMETER_SECTION describes the structure of the model parameters. The section must include a line that defines the variable that will be minimized (the `objective_function_value`). Often, this variable is set to the value of a log-likelihood function, though it can be set to any function that should be minimized. Any temporary variables or variables that are functions of the declared parameters (such as the log of a parameter value) should be defined in this section as well. Note: parameters that are fixed throughout the program should be declared in the DATA_SECTION, not the PARAMETER_SECTION.

```
//this program estimates the slope and the
//intercept of a linear model (e.g., y=b+ax)
//data are found in the Linear.dat file,
// and consist of 10 observed x and y values

DATA_SECTION
    init_int nobs
    init_vector y(1,nobs)
    init_vector x(1,nobs)

PARAMETER_SECTION
    init_number a
    init_number b
    objective_function_value f

PROCEDURE_SECTION
    f=0;
    for (int i=1;i<=nobs;i++) //a for loop. Indent nested code for clarity
    {
        f+=pow(y(i)-(b+a*x(i)),2);
    }
```

Define parameters and any temporary variables used to evaluate them in the PARAMETER_SECTION. You must also define the variable that will be minimized (i.e., the `objective_function_value`). The `objective_function_value` is required.

The preface `init_` indicates that the parameter will be given an initial value and estimated in the minimization procedure. Initial parameter values are specified in a .pin file. The order of the parameter values in the .pin file must match the order in which the parameters are declared in the template file. By default, AD Model Builder will look for parameter values in the .pin file that shares the program name. The default behavior can be overridden with a command line option when the program is run. Note: AD Model

Builder assigns default parameter values if no initial value is specified.

The PARAMETER_SECTION pictured above defines two parameters, **a** and **b**, both of which will be estimated by the program. The objective_function_value (**f**) is the value that will be minimized. In this case, the sum of squares will be minimized. We will look at this example in more detail in Section 7.1.1. For now, it is only important to note the syntax and content of the PARAMETER_SECTION.

By setting bounds on parameter values and/or estimating parameter values in phases (see Tables 4.2 and 4.3 at the end of this section), template designers can make their analyses more efficient and accurate. Bounds on the estimated parameters stop their values from going outside a realistic range. Phases allow for the preliminary estimation of the parameters that have a large influence on the overall results before estimating the parameters that have a small influence on the model. This technique is similar to producing good starting values for the estimation process, which helps streamline the estimation procedure. In addition, model parameters can be fixed at their initial values if necessary.

AD Model Builder will also automatically generate a standard error report or a likelihood profile (when the program is run with the **-lprof** command-line option) for designated parameters. Note that standard error is automatically calculated and reported for all parameters declared with **init_**. Use the prefix **sdreport_** when defining a parameter to generate a standard error report. Use the **likeprof_** prefix to indicate that a likelihood profile should be generated. See Section 7 for examples.

Parameters can be initialized in one of three ways: (1) via a .pin file, (2) in the template's (optional) INITIALIZATION_SECTION, or (3) in the template's PRELIMINARY_CALC_SECTION. If a parameter is not initialized in any of these ways, ADMB will assign it an initial default value—either 0 or (if the parameter is bounded) the midpoint of the defined interval. Because using a .pin file is the most common way to initialize parameter values, we will look at using .pin files here. For more information about initializing parameter values in the PRELIMINARY_CALC_SECTION, please see the User Manual.

The .pin file is much like the data file (.dat), except that it is used to store initial parameter values. The initial parameter values specified in the .pin file must appear in the same order as the declared parameters.

```

PARAMETER_SECTION
    init_number th
    init_vector pop(1,3)
    objective_function_value f

```

Template file (par.tpl)

```

# Initial parameter values

# Value for th parameter
1

# Value for pop parameter
2 2 2

```

Initial parameter value file (par.pin)

Table 4.2: Single Parameters. All parameters are floating point numbers.

Parameter Type	Definition and example
Unbounded, active parameter	The most basic parameter type. Example: <code>init_number a</code> These parameters will be given an initial value and estimated in the minimization procedure. If no other initialization is done in the program or via a .pin file, ADMB initializes the value to zero.
Bounded, active parameter.	<code>init_bounded_number a(0,1)</code> These parameters will be given an initial value and estimated in the minimization procedure. The bounds specify a range of valid values. In this example, ‘a’ can take values between 0 and 1. If no other initialization is done in the program or via a .pin file, ADMB initializes the value to the mid-point of the interval (in this example, .5).
Fixed parameter	<code>init_number a(-1)</code> or <code>init_bounded_number a(0,1,-1)</code> Fixed parameters will not be estimated. To fix a parameter at its initial value, add a ‘-1’ as shown above.
Parameter optimized in phases	<code>init_number a(2)</code> or <code>init_bounded_number b(0,1,3)</code> To estimate parameters in phases, specify the phase in which the parameter should be active. In the above examples, parameter <code>a</code> will be estimated in phase 2 and parameter <code>b</code> will be optimized in phase 3. The value of <code>a</code> will remain fixed in phase 1; the value of <code>b</code> will remain fixed in phases 1 and 2.

Table 4.3: Vectors of Parameters

Parameter Type	Definition and example
Unbounded, active vector of parameters	<pre>init_vector theta(1,3)</pre> <p>These parameters will be given an initial value and estimated in the minimization procedure. If no other initialization is done in the program or via a .pin file, ADMB initializes the value to zero. The above example defines a vector with 3 elements and valid index from 1 to 3.</p>
Bounded, active vector	<pre>init_bounded_vector theta(0,5,-1,3)</pre> <p>These parameters will be given an initial value and estimated in the minimization procedure. The bounds specify a range of valid values. In the above example, the first two numbers in the parenthesis (0,5) describe the vector dimensions. The second two numbers (-1,3) indicate the parameter bounds. If no other initialization is done in the program or via a .pin file, ADMB initializes the value to the mid-point of the interval (in this example, 1).</p>
Fixed vector	<pre>init_vector theta(1,3,-1) or init_bounded_vector theta(0,5,-1,3,-1)</pre> <p>Fixed parameters will not be estimated. To fix a parameter at its initial value, add a '-1' as shown above.</p>
Parameter optimized in phases	<pre>init_vector theta(1,3,2) or init_bounded_vector pop(0,5,-1,3,3)</pre> <p>To estimate parameters in phases, specify the phase in which the parameter should be active. In the above examples, the theta parameter will be estimated in phase 2 and the pop vector will be optimized in phase 3. The value of theta will remain fixed in phase 1; the value of pop will remain fixed in phase 1 and 2.</p>
Parameter vector summing to zero	<pre>init_bounded_dev_vector epsilon(1,5,-10,10,2)</pre> <p>A _dev_ vector will be optimized so that it sums to zero. The epsilon vector defined above has an initial index of 1, a length of 5, bounds between -10 and 10, and will be optimized in phase 2.</p>

4.3 Procedure section

The PROCEDURE_SECTION contains the actual model calculations. The lines in this section are written in C++ and must adhere to C++ syntax. All normal C++ statements can be used in this section (e.g., if-then-else statements) as well as operators, math library functions, and user-defined functions. For a list of useful functions and operators, please see Section 8.

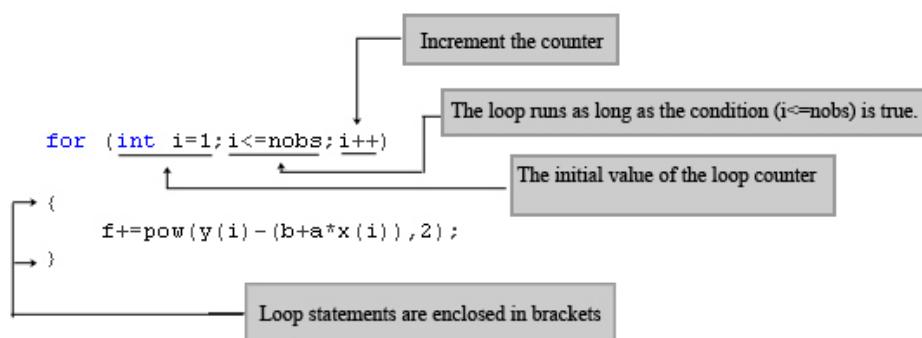
```
//this program estimates the slope and the
//intercept of a linear model (y=b+ax)
//data are found in the Linear.dat file,
//and consist of 10 observed x and y values

DATA_SECTION
    init_int nobs
    init_vector y(1,nobs)
    init_vector x(1,nobs)

PARAMETER_SECTION
    init_number a
    init_number b
    objective_function_value f

PROCEDURE SECTION
    f=0;
    for (int i=1;i<=nobs;i++)
    {
        f+=pow(y(i)-(b+a*x(i)),2);
    }
```

The PROCEDURE_SECTION is written in C++ and must adhere to C++ syntax. Statements must end with a ";"



The code in the displayed PROCEDURE_SECTION uses a for-loop to cycle through each x and y value read in from the data file. Note that the

index for the first item in each vector is 1, not 0. The statement in the loop (`f+=pow(y(i)-(b+a*x(i),2);`) uses the C++ incremental operator (+) to add the squared difference between the observed and the estimated value. Note the use of the `pow(x,y)` function, which raises the first argument, `y(i)-(b+a*x(i))` to the power specified in the second argument, 2. The squared difference is set to the value of the objective function, which is minimized by ADMB.

ADMB users often take advantage of predefined functions (e.g., `sin`, `cos`, `norm`, etc.) in the PROCEDURE_SECTION. To take the log of a parameter, for example, you could use the following line:

```
b=log(parameter);
```

Note that you must define both `parameter` and `b` in the PARAMETER_SECTION before you can use this statement.

For a good list of supported functions, as well as information about creating your own functions, please see the User Manual.

4.4 Non-required template sections

Although only three template sections are required (DATA_SECTION, PARAMETER_SECTION, and PROCEDURE_SECTION), AD Model Builder templates have several optional sections, which you may come across as you are working with AD Model Builder (Figure 4.4).

Use the INITIALIZATION_SECTION to specify initial parameter values. Note that values specified here will be overridden by values specified in a .pin file (if a .pin file is used).

The PRELIMINARY_CALCS_SECTION is often used to manipulate input data—either to convert units (e.g., pounds to kilograms) or to convert data structure (e.g., from matrix to vectors). The C++ statements used in this section are executed only once, unlike the statements in the PROCEDURE_SECTION, which are run once for each iteration of the minimization routine. Instead of using a PRELIMINARY_CALCS_SECTION, users often choose to insert C++ commands into the DATA_SECTION using the !! syntax (see Section 4.1 for an example).

The REPORT_SECTION is used to output user-defined results. Use C++ syntax in this section. For more information and examples, please see Section 6.4.

If you have some C code that you'd like to use, place it in the GLOB-

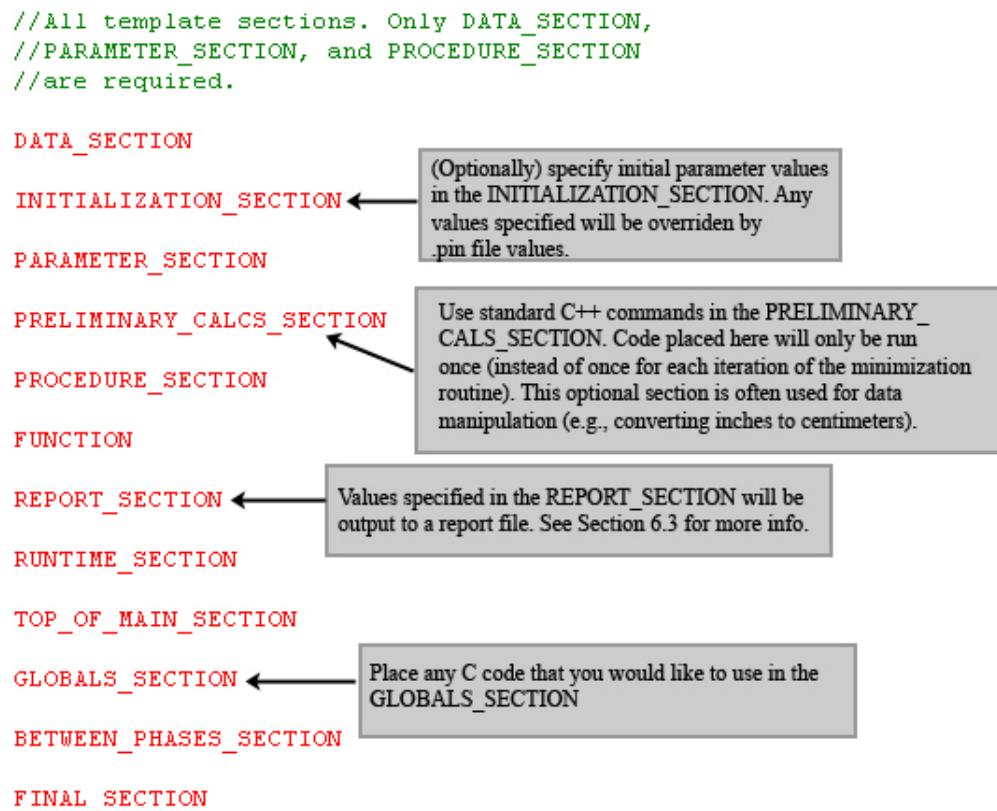


Figure 4.4: The twelve sections of an AD Model Builder template.

ALS_SECTION. Any statements included in this section will be placed at the top of the C++ file that is created from the template and then compiled.

The RUNTIME_SECTION is used to control the behavior of the function minimizer. It is often used to change the stopping criteria during the initial phases of an estimation.

The BETWEEN_PHASES_SECTION contains code that will be executed between estimation phases.

FUNCTION begins the definition of a function or “method” written in C++ code.

Examples of data files and template DATA_SECTION code:

1. Matrix Data (template declaration and data file)

```
DATA_SECTION
    init_int Nobs
    init_int StartYear
    init_int EndYear
    init_matrix Birds(1,Nobs,StartYear,EndYear)
```

Template DATA_SECTION (Birds.tpl)

```
# This data set contains the number of
# birds observed at Painted Post airport.
# Observations were made March 21, June 21,
# and Sept 21 of each study year.

# number of observations (Nobs)
3
# Start Year (StartYear)
1991
# End Year (EndYear)
2000
# Number of birds observed Spring, Summer,
# and Fall of each study year (Birds)

second dimension (year) →
```

first dimension (observations) ↓

```
34 15 67 81 89 45 76 98 45 43
38 54 19 24 34 75 14 75 97 12
74 13 44 38 48 41 85 15 19 72
```

Data file (Birds.dat)

Figure 4.5: An example of matrix data in an ADMB data file (right) and the corresponding template declaration. In the above example, the first dimension is the observations (i.e., the rows in the data set). The second dimension is the years (i.e., the columns in the data set).

2. 3D-Array Data (template declaration and data file)

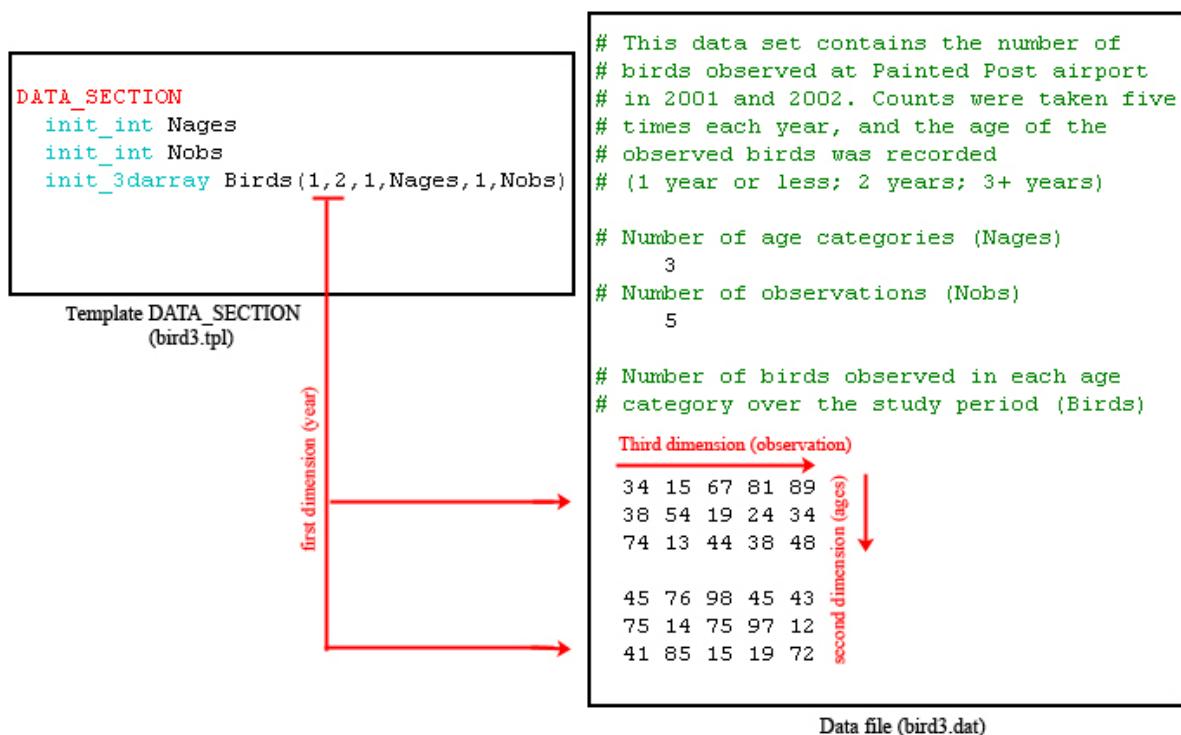


Figure 4.6: Example of 3D-array data and template declaration.

3. Ragged Array Data (template declaration and data file)

```
DATA_SECTION
    init_int Npops
    init_ivector StartYear(1,2)
    init_ivector EndYear(1,2)
    init_matrix Fish(1,Npops,StartYear,EndYear)

Template DATA_SECTION
(fishRag.tpl)

#This file contains catch data for two fish
#populations. The first population was studied
#for 10 years (1991-2000); the second for 5
#(1995-1999). Measurements were taken annually.

# Number of fish populations (Npops)
2

# Start year of each study (Start_Year)
1991 1995

# End year of each study (End_Year)
2000 1999

# Population observations (Fish)
23 5 54 12 45 8 23 45 76 32
45 34 32 54 34
```

Data file (fishRag.dat)

Figure 4.7: Example of ragged-array data and template declaration. In a ragged array, the rows of data have different lengths.

Chapter 5

Compiling and running a program

Once the template is finished, it needs to be translated to C++ and then compiled and linked to the AD Model Builder libraries. All of these steps can be done with a single command (`makeadm`). To run the compiled program, simply type its name at the command line. On Linux systems, use `./programname`. Programs can be run with a number of options, which we will look at in the next sections.

5.1 Compiling a program

To compile a program from the command line, open a command window (on Windows) or a terminal window (on Linux) and navigate to the directory that contains the template. At the command prompt, type:

```
C:\ADMBWork\test> makeadm templatename
```

In the above example, the template is stored in the `C:\ADMBWork\test` directory ('>' is the command prompt). Type the name of the template without its filename extension. For example, if the template file is named "myprogram.tpl", compile it by typing:

```
C:\ADMBWork\test> makeadm myprogram
```

If you are testing or debugging a template, you may wish to use the “safe mode,” which performs bounds checking on all array objects. For example,

safe mode will notify you if an index value exceeds the permissible array bounds (exceeding the bounds can produce calculation errors that may not be detected otherwise). The safe mode is not as fast as the standard mode, and should be used only for debugging—not for a final implementation. To compile a template in safe mode, use a command like:

```
C:\ADMBWork\> makeadm myprogram -s
```

5.2 Running a program and command-line options

Use command-line options to modify the behavior of the program at run-time. To see all available command-line options for a program, type the program name followed by `-?`:

```
> simple -?
```

Table 5.1 contains information about some common command-line options. For a complete list of options and information about each, please see Chapter 12 of the ADMB User Manual.

Table 5.1: Useful command-line options.

Command-line option	Definition and example
<code>-ind <i>datafilename</i></code>	Change the name of the data file used by the program. By default, the program will look for a data file that shares its name. For example, a program named ‘test.exe’ will look for a data file named ‘test.dat’. To use a different data file, e.g., ‘new.dat’, use the following: <code>> test -ind new.dat</code>

Continued on next page

Table 5.1 – continued from previous page

Command-line option	Definition and example
-ainp <i>pinfilename</i>	Change the name of the PIN file used by the program. By default, the program will look for a pin file that shares its name. For example, a program named ‘test.exe’ will look for initial parameter values in a file named ‘test.pin’. To use a different data file, e.g., ‘new.pin’, use the following: <code>> test -ind new.pin</code>
-? -help	Both the ‘-?’ and ‘-help’ options will list all available command-line options.
-est	Only estimate parameters. When this option is used, ADMB will not generate a covariance matrix or standard error report. Because this option saves processing time, you may wish to use it when developing models.
-lprof	Perform likelihood profile calculations for any parameters designated with <code>likeprof_</code> in the parameter section. Results are placed in a .plt file. For an example, please see Section 7.3.3
-mcmc [N]	Perform a Markov Chain Monte Carlo analysis with N simulations. For an example, please see Section 7

5.3 Errors, debugging, and memory management

Syntax errors in AD Model Builder templates can cause troubles when the template is converted to C++ and/or when the converted C+ code is compiled. You may see a message that ADMB cannot translate a template, or that it cannot compile and link the program. Alternatively, the program may build, but generate no output (or odd results) when it is run. You may also receive messages about insufficient memory. In this section, we will look at common types of errors and several troubleshooting techniques.

Many error messages contain a line number or the name of a problematic function or variable. If you are on a Windows system, error messages relate to the translated C++ file (e.g., the .cpp file, not the original .tpl template file). Although you should track down the error in the .cpp file, remember to make the fixes in the original template file.

Some common errors include:

1. Failing to place semi-colons at the end of a line (e.g., in a statement in the PROCEDURE_SECTION or other places where the template uses C++ code)
2. Failing to specify the required objective_function_value in the PARAMETER_SECTION
3. Missing or mis-matched brackets '{}' around loop code or if-statements
4. Typos or inconsistent capitalization in variable names ('Fish' is not the same as 'fish')
5. Errors reading in data (e.g., defining a number variable to hold array data or declaring data variables in an order that differs from the data values in the data file)
6. Failing to specify or incorrectly specifying initial parameter values
7. Using '=' instead of '==' when comparing two values for equality

When troubleshooting, it sometimes helps to comment out sections of code. To comment out code, simply place a “//” before each line:

```
//this code is commented.  
//pred_Y=a*x+b;
```

You can also insert lines of code that print informative messages during runtime. For example, insert a line like the following at the end of the DATA_SECTION to print the value of the last read data object and confirm that it is as expected:

```
!cout<<"Last data object read "<<VARIABLE_NAME<<endl;
```

The above command prints out the text enclosed between quotes as well as the value of the specified variable (*VARIABLE_NAME*). ‘endl’ indicates a carriage return. (Figure 5.1)

You may also find that it is useful to work with a debugger. To use a debugger, load the translated C++ file (.cpp) into a debugging environment, such as MS Developer Studio.

```

DATA SECTION.
init_int nobs;
init_vector Y(1,nobs);
init_vector x(1,nobs);

//troubleshooting technique: add informative messages.
//that are |output during runtime.
!!cout<<"Last data object read in: "<<<endl; ←
.

PARAMETER SECTION.
init_number a .
init_number b .
vector pred Y(1,nobs).

```

Insert a C++ command to output informative messages at runtime. In this example, the program will output the values read into the x vector.

```

C:\ADMBWork\admb\simple>makeadm simple
C:\ADMBWork\admb\simple>simple
Last data object read in: -1 0 1 2 3 4 5 6 7 8
Initial statistics: 2 variables; iteration 0: function evaluation 0
Function value 3.6493579e+001; maximum gradient component mag -3.6127e+000
Var Value Gradient !Var Value Gradient !Var Value Gradient
-1 0.00000 3.6127e-000 1 0 0.00000 3.25914e-001 1

```

Figure 5.1: Use C++ statements to output useful information at runtime.

```

TOP_OF_MAIN_SECTION.
gradient_structure::set_MAX_NVAR_OFFSET(1000); .
gradient_structure::set_NUM_DEPENDENT_VARIABLES(800); .
gradient_structure::set_GRADSTACK_BUFFER_SIZE(100000); .
gradient_structure::set_CMPDIF_BUFFER_SIZE(1000000); ←
arrmbhsize=900000; .

DATA_SECTION.
init_int nobs;
init_vector Y(1,nobs);
init_vector x(1,nobs).

```

To increase the buffer size, increase the values in parenthesis. For more info, please see the AutoDif manual.

Figure 5.2: Increasing the memory buffers.

As the number of estimable parameters in a model increases, the memory required to process the calculations increases as well. If you see an error message that relates to insufficient memory, you may need to increase the memory buffers and/or make your template code more efficient.

To increase the memory buffers, you can either:

1. Increase the buffers using the template's TOP_OF_MAIN_SECTION (Figure 5.2)
2. Increase the buffers using command-line options (-cbs and/or -gbs, for the CMPDIF_BUFFER_SIZE and the GRADSTACK_BUFFER_SIZE, respectively)

For more details about memory buffers and creating efficient code, please see the ADMB Memory Management document:

<http://admb-project.org/community/tutorials-and-examples/memory-management>

Chapter 6

The results: AD Model Builder output files

ADMB outputs results to several useful output files, which are in ASCII format and can be opened and viewed in a standard text editor:

- Parameter Estimate file (.par)
- Standard Deviation file (.std)
- Correlation Matrix file (.cor)
- User-Defined Output file (.rep)

In addition, users can choose to output results to additional, user-defined and formatted files (for use in R, for example) or to the command window at runtime.

Depending on the type of analysis run, AD Model Builder will also create additional output files: a profile likelihood report (.plt); an MCMC posterior distribution report (.hst) and a MCMC algorithm result report (.psv). For more information about these reports, please see Chapter 7.

Table 6.1: ADMB Input and Output Files

File extension	Definition
.tpl	input main model specification
.dat	input data
.pin	input initial parameter values
Continued on next page	

Table 6.1 – continued from previous page

Function	Definition
.cpp	C++ source code (translated from .tpl file)
.htp	C++ header (translated from .tpl file)
.exe	compiled c program
.par	output parameter estimates
.bar	output parameter estimates (binary format)
.std	output parameter standard deviations
.cor	output parameter correlation matrix
.plt	output profile likelihood report
.hst	output MCMC report containing observed distribution
.psv	output the parameter chain from MCMC (binary format)
.rep	output user-generated report
.eva	output the eigenvalues of the Hessian (2nd derivatives of minus the log-likelihood) function. If they are all positive it indicates a minimum.

6.1 Parameter estimate file (.par)

The parameter estimate file contains the parameter estimates, the final objective function value, and the gradient (which should be close to zero) (Figure 6.1). By default, the file shares the name of the program and uses a .par filename extension.

In the parameter estimate file, parameter names are listed above their estimated values, and are set off with a '#'.

```
# Number of parameters = 2  Objective function value = 14.9642 .
Maximum gradient component = 7.00140e-005.
# a:.
1.90909098475.
# b:.
4.07817738582.
```

Figure 6.1: An example of a .par file, which contains two parameter estimates (for a and b).

Note that ADMB also creates a file with a .bar extension (e.g., myTest.bar)

that contains the parameter estimates in a binary file format.

6.2 Standard deviation report file (.std)

The standard deviation report contains the name of each estimated parameter, its estimated value, and its standard deviation (Figure 6.2). The file shares the name of the program and uses a .std filename extension.

```
index  name   value      std dev
1     a  1.9091e+000 1.5547e-001
2     b  4.0782e+000 7.0394e-001
```

Figure 6.2: An example of a standard deviation report file (.std).

6.3 Correlation matrix file (.cor)

By default, ADMB estimates the standard deviations and the correlation matrix for the estimated model parameters (Figure 6.3). These estimates are output to a file with a .cor filename extension.

At the top of the file is the logarithm of the determinant of the hessian. The name of the parameter is followed by its value and standard deviation. The correlation matrix is included after the standard deviation.

```
The logarithm of the determinant of the hessian = 5.33488.
index  name   value      std dev      1      2
1     a  1.9091e+000 1.5547e-001  1.0000.
2     b  4.0782e+000 7.0394e-001 -0.7730  1.0000.
```

Figure 6.3: An example of a correlation matrix file.

6.4 User-defined output file (.rep)

The report file (.rep) is used for user-defined output. Any calculated quantity can be written to this report and formatted as desired via the REPORT_SECTION of the template. Any variables specified in the REPORT_SECTION will be output (Figure 6.4).

```
DATA_SECTION.  
    init_int nobs.  
    init_vector y(1,nobs).  
    init_vector x(1,nobs).  
PARAMETER_SECTION.  
    init_number a .  
    init_number b .  
    objective_function_value f.  
PROCEDURE_SECTION.  
    f=0.;  
    for (int i=1;i<=nobs;i++) .  
    {.  
        f+=pow(y(i)-(b+a*x(i)),2);.  
    }.  
REPORT_SECTION. ←  
    report<<"x data" << endl;.  
    report<<x<<endl;.  
    report<<"y data" << endl;.  
    report<<y<<endl;.  
    report<<"value of a" << endl;.  
    report<<a<<endl;.
```

Use the REPORT_SECTION to output values to the .rep file. You must use C++ syntax. In this example, the program outputs an identifying line of text (e.g., "x data") followed by a carriage return ('endl'), and then the value of the variable (x).

Figure 6.4: Using the template's REPORT_SECTION to output values to the .rep file.

6.5 Outputting results for R

A number of useful tools have been created to help users work with ADMB and R (<http://www.r-project.org/>). Many of these tools have been posted to the Community section of the admb-project.org site:
<http://admb-project.org/community/related-software/r>

Packages include scapeMCMC, an R package for plotting multipanel MCMC diagnostic plots; scape, an R package for plotting fisheries stock assessment data and model fit; ADMB2R, which is used to read ADMB output directly into R; and PBSadmb, which is used to organize and run

48

ADMB models from R.

A useful R-function that can read the contents of an ADMB report file and store the contents in the form of a list object, is also included. For an example using the R-function, please see Section [7.2.2](#).

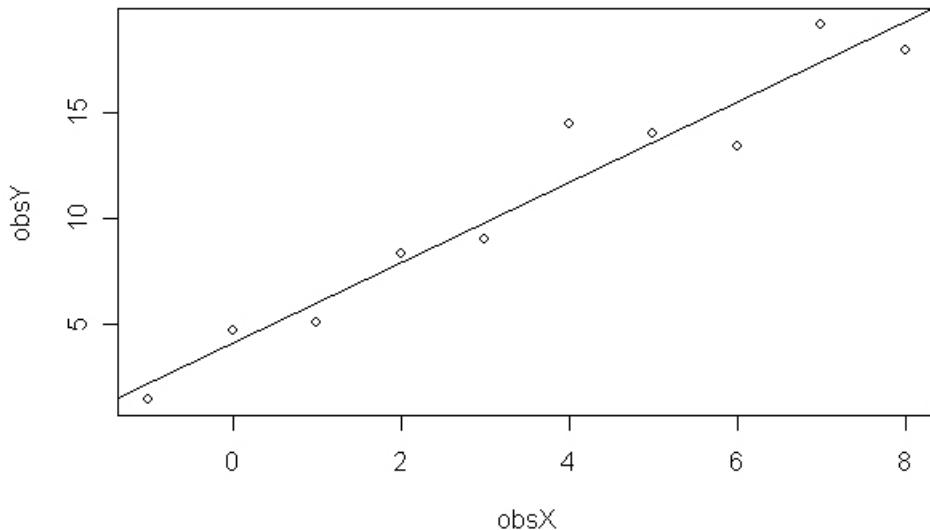
Chapter 7

Examples

7.1 Example 1: Least-squares regression

The regression line displayed below is the ‘best fit’ line through a series of graphed data points. The line shows the relationship between the X and Y data, and can be used to predict the value of the dependent variable (Y) based on the value of the independent variable (X).

Simple linear regression



The relationship between X and Y is linear and can be represented as:

$$Y_i = aX_i + b \quad (7.1)$$

where **a** and **b** are constants (i.e., parameters) representing, respectively, the slope and Y-intercept of the regression line. To estimate the parameters, we can minimize the sum-of-squared differences between the observed and predicted Y values. In other words, the slope and intercept of the regression line is determined by minimizing the following:

$$\sum_{i=1}^n (Y_i - (ax_i + b))^2 \quad (7.2)$$

Throughout this documentation, we refer to the function that will be minimized as the “objective function.”

In this section, we will look at how to create a template to perform a simple linear regression: how to read in the X,Y data points, how to initialize the required parameters (**a** and **b**), how to calculate the predicted Y values, and how to specify the objective function for performing the regression. The first example uses a for-loop to perform the analysis; the second uses the more efficient matrix algebra supported by AD Model Builder.

7.1.1 Using sum of squares

In this example, we will determine the relationship between the distance from the tide line (X) and the weight of algae collected from a six-inch square plot (Y). The data set consists of ten observations, each collected from a different point along the beach. The ADMB **data file** looks like this:

```
# number of observations
10
# observed Y values
1.4 4.7 5.1 8.3 9.0 14.5 14.0 13.4 19.2 18
# observed x values
-1 0 1 2 3 4 5 6 7 8
```

The template ([linear.tpl](#)) for performing the analysis (we'll go through it step-by-step in a moment) looks like this:

```
//this program estimates the slope and the
//intercept of a linear model (y=b+ax)
//data are found in the Linear.dat file,
//and consist of 10 observed x and y values

DATA_SECTION
    init_int nobs
    init_vector y(1,nobs)
    init_vector x(1,nobs)
PARAMETER_SECTION
    init_number a
    init_number b
    objective_function_value f
PROCEDURE_SECTION
    f=0;
    for (int i=1;i<=nobs;i++)
    {
        f+=pow(y(i)-(b+a*x(i)),2);
    }
```

The first thing the template must do is read in the data; this is done in the DATA_SECTION. Note that the order of declared variables must correspond to the order of data in the data file:

```
DATA_SECTION
    init_int nobs
    init_vector y(1,nobs)
    init_vector x(1,nobs)
```

The above template code creates an integer `nobs`, which is initialized to the number of observations specified in the data file (10). The vectors `y` and `x` are also created, and receive the `y` and `x` values, respectively. Note that once declared, a variable can be used by subsequent lines of template code. For example, `nobs` is used when specifying the length of the `x` and `y` vectors.

Parameters are declared in the PARAMETER_SECTION:

```
PARAMETER_SECTION
    init_number a
    init_number b
    objective_function_value f
```

Here, `a` and `b` are declared as numbers. If no initial value for the parameters is supplied—as is the case in this example—ADMB will automatically assign the parameters a default value. Every template must have an `objective_function_value`, as well. In this case, the objective function has been defined as `f`. The ADMB code for minimizing the sum of squares is specified in the PROCEDURE_SECTION:

```
PROCEDURE_SECTION
  f=0;
  for (int i=1;i<=nobs;i++)
  {
    f+=pow(y(i)-(b+a*x(i)),2);
  }
```

The procedure code uses a for-loop to “cycle through” all ten observed values. Notice that the objective function is specified using the `pow()` function, which is used to raise the first argument `(y(i)-(b+a*x(i))` to the power of the second argument (2). The ‘`f+=`’ syntax specifies addition—each time the loop iterates, the new squared difference is added to the cumulative sum. The code in the procedure section is iterated until AD Model Builder determines the values of `b` and `a` that minimize the function. Note that at the beginning of each iteration of the minimization routine, the value of `f` is reset to 0.

Click to see the [data file](#) and [template](#) file used in this example.

7.1.2 Using matrix algebra with sum of squares

Matrix algebra, which is more efficient than the for-loop used in the previous example, can be used in the regression. You need only make a few small changes to the code:

In addition to the `a` and `b` parameters and the objective function, the PARAMETER_SECTION must also define a vector to contain predicted Y values:

```
PARAMETER_SECTION
  init_number a
  init_number b
  vector pred_y(1,nobs)
objective_function_value f
```

And the PROCEDURE SECTION must be updated to use Matrix operations:

```
PROCEDURE_SECTION  
    pred_y=a+b*x;  
    f=sum(square(pred_y-y));
```

When the code is executed, AD Model Builder will calculate the predicted Y values and store them in the `pred_y` vector; then the minimization is performed using matrix algebra. Note that the `square()` function is used to square its argument (`pred_y-y`) and that the `sum()` function is used to calculate the sum of the squares.

The full template (`linearmatrix.tpl`) looks like this:

```
DATA_SECTION  
    init_int nobs  
    init_vector y(1,nobs)  
    init_vector x(1,nobs)  
PARAMETER_SECTION  
    init_number a  
    init_number b  
    vector pred_y(1,nobs)  
    objective_function_value f  
PROCEDURE_SECTION  
    pred_y=a+b*x;  
    f=sum(square(pred_y-y));
```

Click to see the [data file](#) and [template](#) used in this example.

7.1.3 Standard Deviation Report (.std)

Process error, measurement error, and model specification error are three important sources of uncertainty in models. Process uncertainty arises because biological processes vary; we may assume that birth rate is constant, but it likely varies slightly from year to year in a way we can't predict with certainty. Observational uncertainty arises when we collect data: we may count fifty rabbits, but we can't be certain that we've accounted for the entire population. Additionally, model specification error can arise because of imprecision in estimated model parameters, which affects our ability to make predictions using the model.

Prediction uncertainty can be explored by declaring any predicted quantities of interest as `sdreport_` variables in the PARAMETER section. Their estimated standard deviations and correlations will then be computed and reported in the `.std` and `.cor` output files.

The standard error is automatically calculated and reported for all parameters declared with `init_` (e.g., `a` and `b`). To generate a standard error report for a derived variable (in our example, the derived variable would be the estimated weight of algae, `pred_y`), declare the quantity of interest (`pred_y`) as a `sdreport_vector` in the parameter section:

```
PARAMETER_SECTION
    init_number a
    init_number b
    sdreport_vector pred_y(1,nobs)
    objective_function_value f
```

The standard deviation information will be reported in the `*.std` and `*.cor` files. The first few lines of the `simplematrix.std` error report generated by the above example are:

index	name	value	std dev
1	a	4.0782e+000	3.5248e-001
2	b	1.9091e+000	7.7850e-002
3	pred_y	2.1691e+000	4.1560e-001
4	pred_y	4.0782e+000	3.5248e-001
5	pred_y	5.9873e+000	2.9644e-001
...			

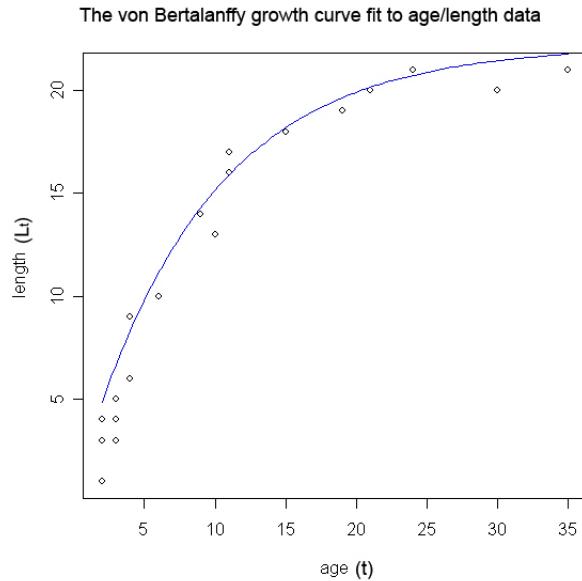
The first few lines of the `.cor` report are:

```
The logarithm of the determinant of the hessian = 8.10168
index name value std dev 1 2 3 4
1 a 4.0782e+000 3.5248e-001 1.0000
2 b 1.9091e+000 7.7850e-002 -0.7730 1.0000
3 pred_y 2.1691e+000 4.1560e-001 0.9929 -0.8429 1.0000
4 pred_y 4.0782e+000 3.5248e-001 1.0000 -0.7730 0.9929 1.0000
...
```

Additionally, uncertainty for any parameter estimates or predicted quantities can be further explored using profile likelihood and MCMC (see Section 7.3.3 for more details).

7.2 Example 2: Nonlinear regression with MLE: Fitting a Von Bertalanffy growth curve to data

The von Bertalanffy growth function predicts the size of a fish as a function of its age. Many fish species have growth-patterns that conform to this model, though the individual curves will be flatter or steeper depending on the values of the estimated parameters for each species' data set.



The form of the growth curve is assumed to be:

$$L_t = L_\infty [1 - e^{-K(t-t_0)}] \quad (7.3)$$

where L_∞ is the mean length of the oldest fish (i.e., the maximum length for the species) and K dictates the shape of the curve—how quickly a fish's length approaches the maximum value. For example, a species with a life-span of one year might have a high K -value (the length approaches the maximum length very rapidly), while a species that lives twenty years might have

a much lower K-value (and a flatter curve). The t_0 parameter adjusts the curve to account for the initial size at birth (which is usually not zero at age zero).

In this section, we will look at how to estimate the L_∞ , K , and t_0 parameters as well as the standard deviation using non-linear regression. In this case, we will use maximum likelihood estimation instead of the Least Squares Estimate (LSE) used in the previous example to calculate the most likely parameter values. To use this method, we must minimize the negative log-likelihood of the density function for the normal distribution. Note that we will also use parameter bounds and phases to ensure that the regression generates the best possible fit to the data.

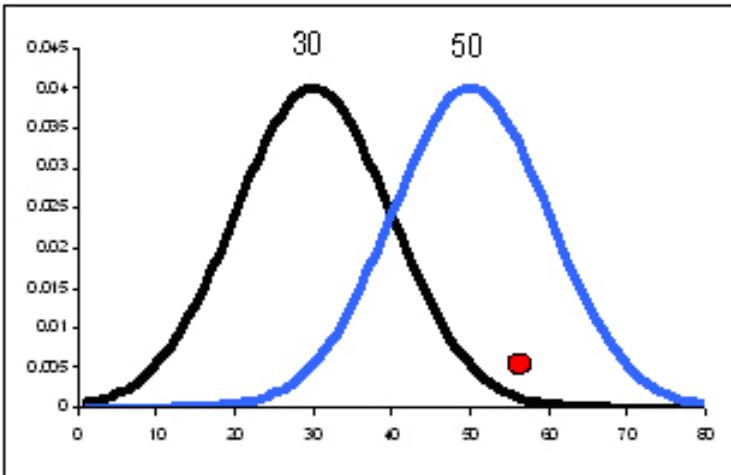
A detailed discussion of maximum likelihood estimation is beyond the scope of this document. However, we will provide a brief introduction to the technique, which is a standard approach to parameter estimation and has several benefits over LSE: it allows us to determine confidence bounds on parameters, and it is a prerequisite to many other statistical methods, including Bayesian methods and modeling random effects.

After we have collected data and come up with a hypothesis about it (i.e., the von Bertalanffy model described above), we need to evaluate the model to determine its “goodness of fit.” When we use a LSE method (as we did in the first example), we attempt to best describe the data by minimizing the difference between the observed and predicted values. When we use a likelihood method, we instead ask the question: given my data, how likely are the various hypotheses about the population from which it came? In other words, maximum likelihood methods identify the parameter values that are most likely to have produced the data rather than the ones that most accurately describe the data sample in terms of how well it fits a model.

Likelihood may remind you of a related and perhaps more familiar concept, probability. With probabilities, we know the population (e.g., a bowl containing 20 red balls and 10 blue balls), and can calculate the probability of any given sample (e.g., 2 red balls and 1 blue ball, or 1 red ball and 2 blue balls) based on that knowledge. With likelihood, we have a sample (our data), and must come to the most likely conclusion about the population from which the sample has been drawn (e.g., what is the mostly likely conclusion we can make about the general population based on our sample of 2 red balls and 1 blue ball?).

The likelihood of the data, given a hypothesis, is assumed to be proportional to the probability. In fact, when we evaluate a likelihood, we use a probability density distribution to identify the parameter vector that best

fits the model to the data. The desired probability distribution makes the observed data “most likely.”



Two normal probability density distributions, one with a mean of 30, the other, a mean of 50. It is more likely that the data (the red dot) come from the distribution with a mean of 50 than one with a mean of 30, and even more likely that the sample comes from a distribution with an even higher mean.

Which density distribution should be used (e.g., normal, binomial, etc.) for an analysis depends on the nature of uncertainty in the model. For example, if the data are categorical, the uncertainty may be described by a multinomial distribution. Data that deviate from their average in a way that follows a normal distribution can be described by the normal distribution.

Because likelihoods are often very small numbers, we use the logarithm of the likelihood (the “log-likelihood”) when comparing hypotheses. Traditionally, we choose to minimize the negative log-likelihood (rather than maximize the log-likelihood), though either approach would provide the same result.

Likelihood for the normal distribution:

$$\mathcal{L}(\Theta | data) = \frac{1}{\sqrt{2\Pi\sigma^2}} \exp\left[-\frac{(O - P)^2}{2\sigma^2}\right] \quad (7.4)$$

Negative log-likelihood:

$$-\ln \mathcal{L}(\Theta | data) = 0.5 \ln(2\Pi) + \ln(\sigma) + \frac{(O - P)^2}{2\sigma^2} \quad (7.5)$$

Negative log-likelihood without constants, σ known:

$$-\ln \mathcal{L}(\Theta | data) = \frac{(O - P)^2}{2\sigma^2} \quad (7.6)$$

For a more thorough introduction to maximum likelihood methods, we recommend *The Ecological Detective* by Ray Hilborn and Marc Mangel.

7.2.1 Using phases and bounds

The data used in this example consist of twenty observations of age and length data. Observed ages range from 2 to 35 years. To see the data file formatted for AD Model builder ([vonb.dat](#)), click here.

Initial parameter values are supplied in a .pin file named [vonb.pin](#):

```
# t0 (the size at birth is close to zero)
0
# Linf (L∞, the maximum size is close to the maximum observed size)
21
# K (the shape of the curve)
0.1
# Standard Deviation
1
```

Note that if no initial parameter values are specified, AD Model Builder will supply default initial values. The complete template ([vonb.tpl](#)) for performing the regression is:

```

DATA_SECTION
init_int nobs
init_matrix dat(1,nobs,1,2)
vector L(1,nobs)
vector a(1,nobs)
!!a=column(dat,1);
!!L=column(dat,2);

PARAMETER_SECTION
init_number t0
init_number Linf
init_number k
init_bounded_number sd(0.01,10,2)
vector Lpred(1,nobs)
objective_function_value f

PROCEDURE_SECTION
Lpred=Linf*(1-mfexp(-k*(a-t0)));
f=nobs*log(sd)+sum(0.5*square((log(L)-log(Lpred))/sd));

```

The data are read in and extracted with the code in the DATA_SECTION:

```

DATA_SECTION
init_int nobs //reads number of observations from data file
init_matrix dat(1,nobs,1,2) //reads age length data from data file
vector L(1,nobs) //creates a vector L with 20 elements.
vector a(1,nobs) //creates a vector a with 20 elements.
!!a=column(dat,1); //extracts the first column (age) from dat matrix
!!L=column(dat,2); //extracts the second column (length) from dat matrix

```

The three model parameters: `t0`, `Linf`, `k`, as well as the standard deviation, `sd`, are defined at the top of the PARAMETER_SECTION. Default values are read and assigned from the .pin file. Note that the `sd` parameter is defined as a bounded number, which means that its values will be constrained to fall between a lower and an upper bound, in this case, between .01 and 10.

Placing bounds on variables ensures that their MLEs fall within a reasonable range. Note that AD Model Builder does not “know” the limits of a reasonable range unless you specify them. For example, the software might identify that a parameter value that minimizes the negative log-likelihood function is “-10”, which is mathematically valid, but not sensible in a situation where the value represents the length of a fish.

```

PARAMETER_SECTION
    init_number t0
    init_number Linf
    init_number k
    init_bounded_number sd(0.01,10,2)
    vector Lpred(1,nobs)
    objective_function_value f

```

The third parenthetical value after the `sd` variable (the 2), represents the phase in which the parameter will be estimated. AD Model Builder supports “phased optimization,” in which additional parameters can be added and optimized in a series of steps. In each phase, the parameters estimated in the previous phase and the parameters activated in the current phase are all estimated. The values from the previous phase are used as “initial values.” You will likely wish to estimate influential parameters in the early phases to avoid unrealistic parameter space. Relatively well known parameters can be fixed until later phases. In this example, the standard deviation is not estimated until the other model parameters have been “almost” estimated. In other words, the analysis is performed in two phases: first the `t0`, `Linf`, and `k` parameters are estimated (holding `sd` constant at its initial value), and then the estimated values are adjusted as the standard deviation is estimated in a second phase.

In general, the last number in the declaration of an initial parameter, if present, determines the number of the phase in which that parameter becomes active. If no number is given, the parameter becomes active in phase 1. If you wished the `sd` parameter to be active in phase one, for example, you could define it as: `init_bounded_number sd(0.01,10)`. Alternatively, if you wished to activate the `t0` parameter in the second phase instead of the first, you could define it as `init_number t0(2)`.

In addition to the parameters, an `Lpred` vector that will be used in the PROCEDURE SECTION, as well as the (required) objective function, `f`, are defined in the PARAMETER SECTION.

The PROCEDURE SECTION contains two lines. The first uses the model and each set of parameter values generated and tested by AD Model Builder to generate predicted length values. The `mfexp()` function raises the number `e` to the power specified as the argument `(-k*(a-t0))`. Once the predicted values have been generated, they are plugged into the negative log-likelihood function in the second line, which is evaluated for each vector of parameter values. AD Model Builder minimizes this objective function and then returns the results in the output files.

```

PROCEDURE_SECTION
Lpred=Linf*(1-mfexp(-k*(a-t0)));
f=nobs*log(sd)+sum(0.5*square((log(L)-log(Lpred))/sd));

```

The resulting .par file contains the following estimates:

```

# Number of parameters = 4 Objective function value = -14.8033 Maximum
gradient component = 3.31725e-007
# t0:
0.929195941003
# Linf:
22.1727271642
# k:
0.113188254800
# sd:
0.289336477562

```

7.2.2 Plotting the results using R

A simple way to plot ADMB results using R is to use the R-function written by Steve Martell (inspired by some earlier code developed by George Watters). The function reads the contents of a report file (or any output file) and stores the contents as an R list-object. The function is capable of reading single variables, vectors, and 2-D arrays (including ragged arrays).

Note that the output files must conform to the following formatting (variable name, then new line, then values):

```

agedata
2 2 2 2 3 3 3 4 4 6 9 10 11 11 15 19 21 24 30 35
lengthdata
1 3 4 4 3 4 5 6 9 10 14 13 16 17 18 19 20 21 20 21
t0
0.929196
k
0.113188
sd
0.289336
Linf

```

22.1727

Formatting and outputting a customized report is accomplished with a few lines in the template's REPORT_SECTION:

```
REPORT_SECTION
  report<<"agedata"<<endl<<a<<endl;
  report<<"lengthdata"<<endl<<L<<endl;
  report<<"t0"<<endl<<t0<<endl;
  report<<"k"<<endl<<k<<endl;
  report<<"sd"<<endl<<sd<<endl;
  report<<"Linf"<<endl<<Linf<<endl;
```

The above template code generates the output file in the required format and saves it as *template-name.rep* (e.g., vonbR.rep). The complete template, modified to format output for use with the R-function, is [here](#).

To use the R-function to “import the data” into R:

1. Copy and paste the following R-code into a text file and save it as “reptoRlist.R”.

```
reptoRlist <- function(fn) {
  ifile <- scan(fn, what="character", flush=TRUE,
    blank.lines.skip=FALSE, quiet=TRUE)
  idx <- sapply(as.double(ifile), is.na)
  vnam <- ifile[idx] #list names
  nv <- length(vnam) #number of objects
  A <- list()
  ir <- 0
  for (i in 1:nv) {
    ir <- match(vnam[i], ifile)
    if (i!=nv) {
      irr <- match(vnam[i+1], ifile)
    } else {
      irr <- length(ifile)+1 #next row
    }
    dum <- NA
```

```

if (irr-ir==2) {
  dum <- as.double(scan(fn, skip=ir, nlines=1,
    quiet=TRUE, what=""))
}
if (irr-ir>2) {
  dum <- as.matrix(read.table(fn, skip=ir,
    nrow=irr-ir-1, fill=TRUE))
}
# Logical test to ensure dealing with numbers
if (is.numeric(dum)) {
  A[[vnam[i]]] <- dum
}
}
return(A)
}

```

2. Load the function into the R environment. To do so, open the R-environment and type:

```
source(file.choose())
```

Then, using the drop-down menu that appears, navigate to the location where the `reptoRlist.R` function is saved and select it.

3. Use the `reptoRlist` function to read the data into R by typing the following line into the R-environment:

```
A=reptoRlist("filename")
```

Where `filename` is the name of your data file. For example, on a Windows system, the path might look something like:

```
A=reptoRlist("C:/ADMB/vonb/vonbR.rep")
```

4. Once the file has been read into R, the data will be available in the ‘A’ list object. At the R command-prompt, type ‘A’ to view the data:

```

R Console
> A
$agedata
[1]  2  2  2  2  3  3  3  4  4  6  9 10 11 11 15 19 21 24 30 35

$lengthdata
[1]  1  3  4  4  3  4  5  6  9 10 14 13 16 17 18 19 20 21 20 21

$t0
[1] 0.929196

$k
[1] 0.113188

$sd
[1] 0.289336

$Linf
[1] 22.1727

>

```

5. To access a value, use the syntax

`A$variablename`

where *variablename* is the name of the desired value. To access the vector of ages, use:

`A$agedata`

To plot the data and regression line, you could use the following R code:

```

plot(A$agedata,
      A$lengthdata,
      xlab='Age (t)',
      ylab='Length (Lt)')

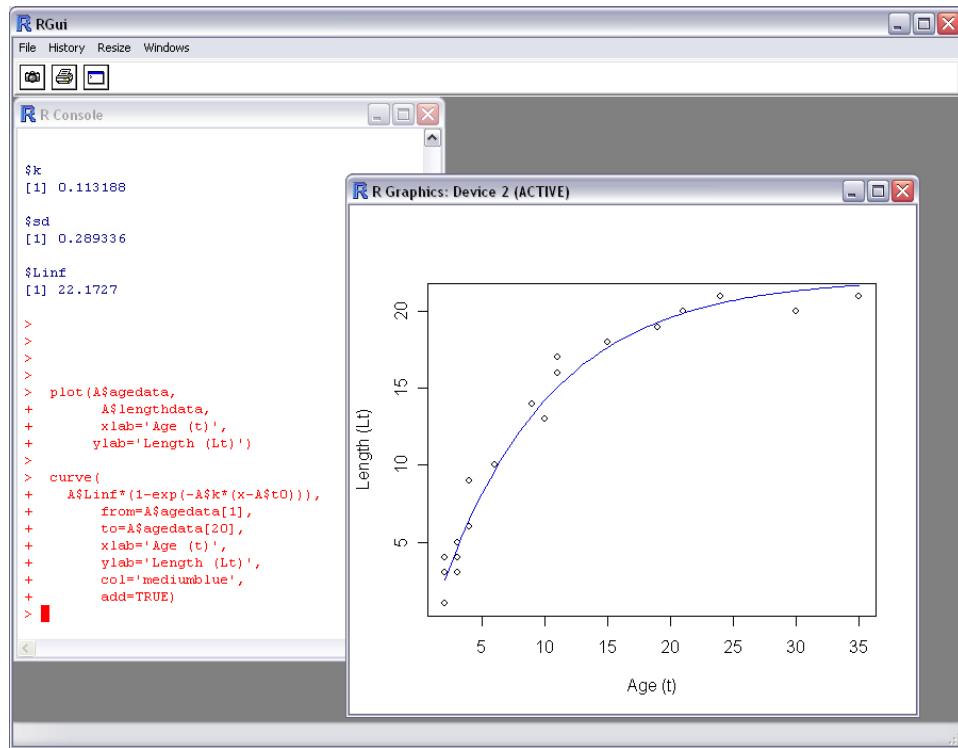
curve(
      A$Linf*(1-exp(-A$k*(x-A$t0))),
      from=A$agedata[1],
      to=A$agedata[20],
      xlab='Age (t)',

```

```

ylab='Length (Lt)',
col='mediumblue',
add=TRUE)

```



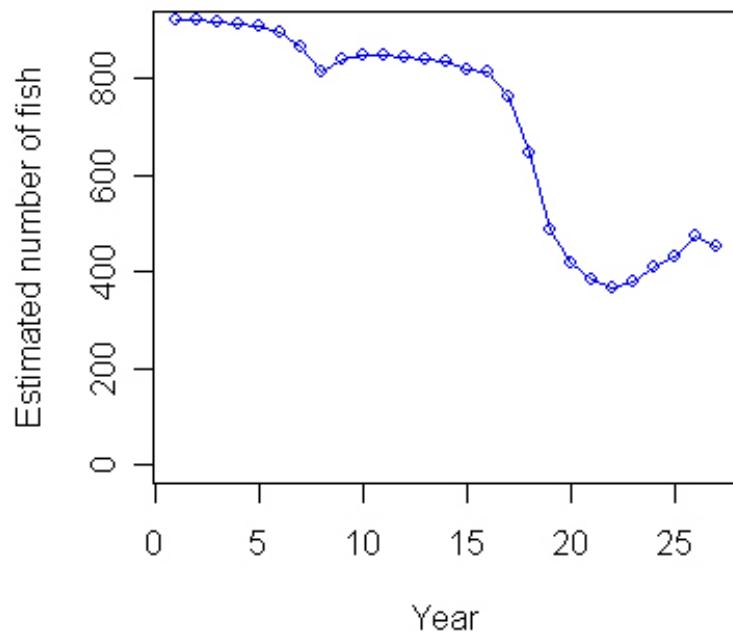
7.3 Example 3: A simple fisheries model: estimating parameters and uncertainty

The simplified fisheries model examined in this section uses maximum likelihood methods to estimate parameter values used to estimate a fish population. The MLE is a prerequisite for the other methods covered in this section: a profile likelihood and a Bayesian analysis that incorporates previous results into the analysis using an MCMC algorithm. In addition to talking a bit about these techniques and how they are accomplished using AD Model Builder, we will also look at several types of related results files that are output by the program.

A bit about the model

Fishery managers have the tough task of estimating the abundance of creatures that travel underwater, where they can't be easily counted. As a result, a number of useful alternative measurements for understanding and managing populations have been developed. One such index is the Catch Per Unit Effort (CPUE), which is assumed to be proportional to abundance. In this model, the number of fish (N_t) in each study year is estimated using known survival rates (S), catch data (C_t), and (an estimated) recruitment rate (R). The estimated population is multiplied by a proportionality constant (q) so that it can be compared to the known CPUE data. Maximum likelihood methods are then used to estimate the values of the model's two unknown parameters: the recruitment rate (R) and the proportionality constant (q). Ultimately, using the estimated R parameter with equations 7.7 and 7.8 in the model, we can look at the changes in fish abundance over the study period.

Estimated Fish Population Over Study Period



The number of fish at time zero is described as

$$N_0 = \frac{R}{(1 - S)} \quad (7.7)$$

where R is recruitment and S is survival. The survival rate is known; the recruitment will be estimated in the regression.

The number of fish in each subsequent year is then assumed to be

$$N_{t+1} = N_t S + R - C_t \quad (7.8)$$

where C_t is the catch rate. The catch rate for each time period is available in the data set.

The value of N is then adjusted to be equivalent to the CPUE index

$$I_t = qN_t \exp[\varepsilon_t] \quad (7.9)$$

where I is the CPUE index, q is a constant of proportionality estimated by the regression, and ε_t is a measure of error.

The error is assumed to be normally distributed with a mean of 0 and a standard deviation σ , which is known

$$\varepsilon_t \sim N(0, \sigma^2) \quad (7.10)$$

The form of the negative log-likelihood that we will minimize (without constants, σ known) is:

$$-\ln \mathcal{L} = \frac{(\ln[I_t] - \ln[qN_t])^2}{2\sigma^2} \quad (7.11)$$

7.3.1 Maximum likelihood estimate (fish.tpl)

In this example, the data includes CPUE and catch data for each of the 26 study years, as well as the survival rate, which is assumed to be constant. Click here to see the [data file](#). The parameters to be estimated are R (the recruitment rate) and q (the proportionality constant). Initial parameter values for both R and q are provided in the [.pin](#) file: 300 and .001, respectively.

The AD Model Builder template ([fish.tpl](#)) that will calculate the estimated fish populations for each study year uses maximum likelihood to estimate the q and R parameters is:

```

DATA_SECTION
    init_number S
    init_number sigma
    init_int Nyears
    init_vector C(1,Nyears)
    init_int Nobs
    init_matrix CPUE(1,Nobs,1,2)
PARAMETER_SECTION
    init_number R
    init_number q
    vector N(1,Nyears+1)
    objective_function_value f
PROCEDURE_SECTION
    f=0;
    N(1)=R/(1-S);
    for(int year=1;year<=Nyyears;year++)
    {
        N(year+1)=S*N(year)+R-C(year);
    }
    for(int obs=1;obs<=Nobs;obs++)
    {
        f+= 0.5*square((log(CPUE(obs,2))-log(N(CPUE(obs,1))*q)) / sigma);
    }
REPORT_SECTION
    report<<N<<endl;

```

The data is read in via the DATA_SECTION. No transformation is required.

```

DATA_SECTION
    init_number S //the survival rate (0.7)
    init_number sigma //the standard error (0.4)
    init_int Nyears //the number of years in the study (26)
    init_vector C(1,Nyears) //the catch data for each year
    init_int Nobs //the number of CPUE observations (26)
    init_matrix CPUE(1,Nobs,1,2) //the study year and CPUE data

```

The two estimated parameters (R and q), as well as a vector, N , that will contain the estimated abundance values, are defined in the PARAMETER_SECTION. The objective function, f , is also defined.

```

PARAMETER_SECTION
    init_number R
    init_number q
    vector N(1,Nyears+1)
    objective_function_value f

```

The PROCEDURE_SECTION contains all the C code needed to calculate the estimated abundance values and to perform the maximum likelihood analysis:

```

PROCEDURE_SECTION
[1]     f=0;
[2]     N(1)=R/ (1-S);
[3]     for(int year=1;year<=Nyears;year++)
[4]     {
[5]         N(year+1)=S*N(year)+R-C(year);
[6]     }
[7]     for(int obs=1;obs<=Nobs;obs++)
[8]     {
[9]         f+= 0.5*square((log(CPUE(obs,2))-log(N(CPUE(obs,1))*q)) / sigma);
[10]

```

Line [1] sets the objective function (f) to zero so that each iteration of the minimization routine will start with a “clean slate.” The code in line [2] uses Equation 7.7 to estimate the abundance for the initial year. Lines 3-6 use a for-loop to calculate the abundance for the subsequent years (using Equation 7.8). Lines 7-10 also use a for-loop, this time to loop through each year and minimize the negative log-likelihood function. Because we know σ , we can minimize the negative log-likelihood function of the form 7.11. Note that $CPUE(obs, 1)$ is equal to the first column of data in the CPUE matrix, which simply contains the year indices that run from 1 to 26.

Compiling the template and executing the program produces the following results (fish.par):

```

# R:
272.858848131
# q:
0.000821746937564

```

The estimated fish population for each year (N) is output in the REPORT_SECTION and will appear in the fish.rep file.

7.3.2 Likelihood Profile and Bayesian posterior analysis

Bayesian analyses are useful for a number of reasons: they can be used to include information from previous studies, estimate uncertainty, and calculate the probabilities of alternative hypotheses. When we perform a Bayesian analysis, we generate a “posterior distribution” that describes the probability assigned to each possible hypothesis in light of the collected data.

Bayesian analysis is based on Bayes’ theorem:

$$Pr\{B|A\} = \frac{Pr\{A, B\}}{Pr\{A\}} = \frac{Pr\{A|B\}Pr\{B\}}{Pr\{A\}} \quad (7.12)$$

From Bayes’ theorem, we can draw the conclusion that the probabilities in the posterior distribution (e.g., the probability of each hypothesis given the data) are proportional to the product of each likelihood and prior probability. (For a more thorough look at Bayes’ theorem and how it is used, please see *The Ecological Detective* by Hilborn and Mangel.)

$$Pr\{H_1|data\} = \frac{\mathcal{L}\{H_1, data\}Pr\{H_1\}}{Pr\{data\}} \quad (7.13)$$

Note that when using Bayesian analysis, we must provide “priors”—the prior probability of each parameter, which doesn’t take into account the new observations. The prior probability can be specified with a probability density distribution (e.g., a normal distribution with a mean that represents the most probable value of the parameter, and a standard deviation that represents our prior uncertainty in this parameter). Often, we have no prior information about a parameter, and in that case, we attempt to use an “uninformative prior” that will not affect the estimation of the quantities of interest. Two standard ‘uninformative’ priors are uniform and uniform on the log scale.

In this example, we will again estimate q and R , only this time we will provide priors for each parameter. The prior R is assumed to be normally distributed with a mean and standard deviation supplied (`Rmean` and `Rsigma`), and the prior q is uniformly distributed. Note that alternative values for a log-uniform scale are included in the comments in the data file. To see

how assumptions about a parameter distribution affect the analysis, we recommend that you try running the example using both the uniform and log-uniform priors for q .

[Click here](#) to view the [data file](#) and the [.pin](#) file.

Although we are estimating both q and R , we are ultimately most interested in R because we are using it to generate our fish population estimates. In order to look more closely at R and determine how likely each estimated value is, the template also generates a likelihood profile for the parameter. The likelihood profile allows us to examine how the model's likelihood changes as parameter values are changed, and to generate confidence intervals on estimated parameters. Once AD Model Builder has generated the likelihood profile, it can also generate a Bayesian posterior distribution using a Markov Chain Monte Carlo (MCMC) algorithm. The MCMC analysis is easily initiated and customized by the user with a few command-line options.

[Click here](#) to see the complete template ([bayes.tpl](#)) used in the analysis. Notice that the DATA_SECTION now contains the new prior data for R and q :

```
DATA SECTION
  init_number Rmean //The mean of the R-prior distribution (300)
  init_number Rsigma //the standard deviation of the R-prior distribution (100)
  init_int qtype //the distribution describing q-prior (0,uniform)
  init_number qlb //The lower-bound of possible q values (0.000001)
  init_number qub // The upper-bound of possible q values (1)
  init_number S //the survival rate (0.7)
  init_number sigma //the standard error (.4)
  init_int Nyears //the number of study years (26)
  init_vector C(1,Nyears) //the catch data for each study year
  init_int Nobs //the number of CPUE indexes (26)
  init_matrix CPUE(1,Nobs,1,2) //the CPUE data for each of the 26 years
```

The R and q parameters are initialized in the PARAMETER_SECTION. Note that they are both specified as bounded numbers with lower and upper bounds to constrain the range of possible values to a realistic interval. $qtemp$ is used in the PROCEDURE_SECTION to convert the q distribution from uniform to log-uniform if the log-uniform scale is specified.

Note that AD Model Builder will automatically generate a likelihood profile for any parameter defined as “likeprof_number” in the PARAMETER_SECTION when the -lprof command-line option is used during runtime (e.g., `bayesfish -lprof`). The likelihood profile variable must be defined in the PARAMETER_SECTION and set equal to the parameter of

interest in the PROCEDURE_SECTION (we'll look at that in a moment). As in the previous example, the vector *N* is defined and used to store population estimates generated from the model. The objective function, *f*, is also defined.

```
PARAMETER_SECTION
    init_bounded_number R(1,1000)
    init_bounded_number q(qlb,qub)
    number qtemp
    likeprof_number Rtemp
    vector N(1,Nyears+1)
    objective_function_value f
```

The PROCEDURE_SECTION is much like the one used in the previous example with a couple additions: on line [2], *Rtemp=R*; is used to assign the likeprof_number defined in the PARAMETER_SECTION to the parameter of interest (*R*) so that AD Model Builder knows the parameter for which to generate the likelihood profile. Lines [8] and [9] are used to transform the value of *q* from uniform to log-uniform, depending on the value of *qtype*. In our data set, the value of *qtype* is specified as 0, so no transformation will be performed. Finally, line [14] is used to calculate the maximum likelihood estimate of *R* and its posterior distribution based on current estimates and prior values.

```
PROCEDURE_SECTION
[1]   f=0;
[2]   Rtemp=R;
[3]   N(1)=R/(1-S);
[4]   for(int year=1;year<=Nyears;year++)
[5]   {
[6]     N(year+1)=S*N(year)+R-C(year);
[7]   }
[8]   if (qtype==0) qtemp=q;
[9]   else qtemp=exp(q);
[10]  for(int obs=1;obs<=Nobs;obs++)
[11]  {
[12]    f+= 0.5*square((log(CPUE(obs,2))-log(N(CPUE(obs,1))*qtemp)) / sigma);
[13]  }
[14]  f+=0.5*square((R-Rmean) / (Rsigma));
```

Compile the template with the command: `makeadm bayesfish`, where `bayesfish` is the name of the template without its .tpl extension. To run the program and generate a likelihood profile report, use the `-lprof` option at the command line:

```
C:\ADMB\MinGW>bayesfish -lprof
```

When the `-lprof` option is used, AD Model Builder will generate a `.plt` file that contains the likelihood profile results for any parameter designated as a `likeprof_number` in the template. The report is named after the estimated parameter. In this case, the file will be named “`Rtemp.plt`”. Note that in this example, the likelihood report for `Rtemp` will include the effects of the prior information. To see a likelihood report that does not include prior information, you could define `Rtemp` as a likelihood profile variable in the previous example (`fish.tpl`). An example of a likelihood profile report is included in Section [7.3.3](#).

If you would like to adjust the number or size of steps used in the likelihood profile calculation, add the following lines to the template after the `PARAMETER_SECTION`:

```
PRELIMINARY_CALCS_SECTION
  Rtemp.set_stepnumber(10); //default value is 8
  Rtemp.set_stepsize(0.1); // default value is 0.5.
  //Note: the stepsize is in standard deviation units.
```

In addition to the likelihood profile, AD Model Builder will also generate the standard output files (`.par`, `.cor`, `.std`). The parameter estimates contained in the `bayesfish.par` file are:

```
# R:
276.553534814
# q:
0.000805865729783
```

To create a posterior probability distribution using AD Model Builder’s MCMC algorithm, simply run the program with the `-mcmc` command-line option:

```
bayesfish -mcmc 10000 -mcsave 100
```

The `-mcmc` option initiates the MCMC algorithm (AD Model Builder uses the Metropolis-Hastings algorithm). The number following the `-mcmc` option (10000, in this case) specifies the number of samples (i.e., sets of parameter values) to take when the model is run. The second option, `-mcsave` 100, tells AD Model Builder how often to save the samples. In this case, ADMB will save the result of every 100th simulation, essentially “thinning” the chain and reducing auto-correlation. To save every value, use `-mcsave 1`

After the MCMC algorithm has run, the saved results can be used by running the model again with the `-mceval` option:

```
bayesfish -mceval
```

The `-mceval` option will evaluate a user-supplied function specified in the template once for every saved simulation value. The function `mceval_phase()` can be used in the template as a “switch” that is activated when AD Model Builder is performing an `-mceval` operation:

```
if (mceval_phase())
{
    ofstream out("posterior.dat",ios::app);
    out<<R<<" "<<q<<" "<< N<<endl;
    out.close();
}
```

The above code instructs AD Model Builder to create an output file named “posterior.dat” and to write the values of `R`, `q`, and `N` to the file when a user evaluates the results generated by the MCMC analysis.

Note: the results generated by the MCMC algorithm are saved in a binary file `root.psv` (e.g., `bayesfish.psv`). For information about converting this file into ASCII, please see the User Manual. The `.psv` report can also be read into R with the following lines of R-code:

```
filen <- file("your.psv", "rb")
nopar <- readBin(filen, what=integer(), n=1)
mcmc <- readBin(filen, what=numeric(), n=nopar * 10000)
mcmc <- matrix(mcmc, byrow=TRUE, ncol=nopar)
```

```

R Console

> filen <- file("C:/ADMBWork/manual/Example3_Fish/bayes.psv", "rb")
> nopar <- readBin(filen, what=integer(), n=1)
> mcmc <- readBin(filen, what=numeric(), n=nopar * 10000)
> mcmc <- matrix(mcmc, byrow=TRUE, ncol=nopar)
> mcmc
     [,1]      [,2]
[1,] 276.5535 0.0008058657
[2,] 260.6408 0.0009231368
[3,] 227.0858 0.0011224753
[4,] 281.9727 0.0008043407
[5,] 318.0835 0.0005499933
[6,] 245.9336 0.0009641961
[7,] 303.7369 0.0007647302
[8,] 270.2589 0.0008363575
[9,] 306.1241 0.0007720200
[10,] 309.3638 0.0007293926
[11,] 265.0659 0.0008602725
[12,] 273.9041 0.0007430795
[13,] 339.9104 0.0005358713
[14,] 288.7475 0.0007247757
[15,] 262.3541 0.0009356574
[16,] 245.5393 0.0010527074
[17,] 278.0209 0.0007812575
[18,] 274.4290 0.0007635835
[19,] 303.1165 0.0006188994

```

7.3.3 Report: Profile likelihood report (.plt)

The profile likelihood report contains the value and corresponding probability for the profiled variable (in this example, Rtemp). The report also contains lower and upper bounds that correspond to 90%, 95% and 97.5% confidence limits, as well as either lower or upper bounds for one-sided confidence limits. At the bottom of the report is the normal approximation of the probabilities and confidence levels obtained from the parameter values and their covariances. The following report is (an abridged version of) the report generated by the bayesfish example in this chapter.

```

Rtemp:
Profile likelihood
. .
250.752    0.00678057
253.468    0.00717952
256.184    0.00757846
258.9      0.00797741
261.616    0.00837635

```

264.332	0.00877529
267.048	0.00917424
268.406	0.00922056
269.764	0.00926688
271.122	0.0093132
272.48	0.00935952
273.838	0.00940584
275.196	0.00945216
276.554	0.00949848
. . .	

Minimum width confidence limits:

significance level	lower bound	upper bound
0.9	223.593	379.097
0.95	219.52	409.438
0.975	205.094	430.002

One sided confidence limits for the profile likelihood:

The probability is 0.9 that Rtemp is greater than 247.276
 The probability is 0.95 that Rtemp is greater than 236.936
 The probability is 0.975 that Rtemp is greater than 228.573

The probability is 0.9 that Rtemp is less than 369.234
 The probability is 0.95 that Rtemp is less than 396.955
 The probability is 0.975 that Rtemp is less than 419.396

Normal approximation

. . .	
261.616	0.00925701
264.332	0.00956935
267.048	0.00988169
268.406	0.00992652
269.764	0.00997134
271.122	0.0100162
272.48	0.010061
273.838	0.0101058
275.196	0.0101506
276.554	0.0101955
277.911	0.0101506

```

279.269      0.0101058
280.627      0.010061
281.985      0.0100162
283.343      0.00997134
. . .
Minimum width confidence limits:
significance level   lower bound   upper bound
                           0.9          204.167     344.451
                           0.95         192.361     356.733
                           0.975        181.645     366.178

```

One sided confidence limits for the Normal approximation:

The probability is	0.9 that Rtemp is greater than 226.407
The probability is	0.95 that Rtemp is greater than 212.012
The probability is	0.975 that Rtemp is greater than 200.051
The probability is	0.9 that Rtemp is less than 332.121
The probability is	0.95 that Rtemp is less than 347.48
The probability is	0.975 that Rtemp is less than 357.086

7.3.4 Report: Markov Chain Monte Carlo (MCMC) report (.hst)

The .hst report contains information about the MCMC analysis: the sample sizes (specified with the `-mcmc` command-line option), the step size scaling factor, the step sizes, and information about the posterior probability distribution (e.g., the mean, standard deviation, and lower and upper bounds).

For each simulated parameter, a range of values (with step sizes reported in the "step sizes" section of the .hst file) and their simulated posterior probabilities is reported. Plotting the first column (parameter values) on the x-axis and the second column (simulated probabilities) on the y-axis can be a convenient way to make a visualization of the posterior probability distribution.

```

# samples sizes
100000
# step size scaling factor
1.2
# step sizes

```

```

14.7402
# means
310.04
# standard devs
117.995
# lower bounds
-17
# upper bounds
21
#number of parameters
3
#current parameter values for mcmc restart
0.803384 351.456 -7.83507
#random nmber seed
1648724408
#Rtemp
. . .
177.379 0.000232697
192.119 0.000357526
. . .

```

7.4 Example 4: Simulation testing

In this section, we will look at how to generate random numbers in ADMB, and how to use them to simulate data sets that can be used to test the fit of a model.

7.4.1 Simulating data: Generating random numbers

Random numbers can be generated with the `random_number_generator` class:

```
random_number_generator rng(n);
```

where `n` is the seed that initializes the random number generator.

The following script demonstrates how to generate a sample of five random values from each of the uniform, normal, poisson, negative binomial, standard Cauchy, binomial, and multinomial distributions. The random seed (`rng`) is defined on the first line of the LOCAL_CALCS section (123456). The template directs ADMB to write the output (displayed below the script) to the screen. Note that when you use LOCAL_CALS and

END_CALCS in the DATA_SECTION, you must indent the LOCAL_CALCS identifier one space.

```

DATA_SECTION
LOC_CALCS
    random_number_generator rng(123456);
    dvector sample(1,5);
    //generates a uniformly distributed random number
    sample.fill_randu(rng);
    cout<<"Uniform(0,1): "<<sample<<endl;

    //generates a normally distributed random
    //number with mean 0 and sd 1
    sample.fill_randn(rng);

    //generates a poisson with mean of 1.5
    sample.fill_randpoisson(1.5,rng);
    cout<<"pois(1.5): "<<sample<<endl;

    //generates a neg binomial with mean of 1.5 and overdispersion 2.0
    sample.fill_randnegbinomial(1.5,2.0,rng);
    cout<<"neg.bin(1.5,2): "<<sample<<endl;

    //generates a standard Cauchy distributed random number
    sample.fill_randcau(rng);
    cout<<"Cauchy: "<<sample<<endl;

    //generates random binomials with probability 0.8
    sample.fill_randbi(0.8,rng);
    cout<<"binomial(n=1,p=0.8): "<<sample<<endl;

    //generates multinomial distributed random number.
    dvector p("(.01,.495,.495)");
    sample.fill_multinomial(rng,p);
    cout<<"multinomial(n=1,p=(.01,.495,.495)): "
    <<sample<<endl;
    ad_exit(0);
END_CALCS
PARAMETER_SECTION
    objective function value nll;

```

following The script produces the below output:

```

Uniform(0,1): 0.779837 0.229835 0.0126429
0.714228 0.654815
Normal(0,1): -0.325127 1.03682 0.567672
-0.670345 2.89024
pois(1.5): 2 2 0 1 2
neg.bin(1.5,2): 0 3 1 0 4

```

```

Cauchy: -0.188267 -1.30511 -9.11156
29.8652 2.83259
binomial(n=1,p=0.8): 1 1 0 0 1
multinomial(n=1,p=(.01,.495,.495)): 3 2 3 2 3

```

In the next section, we will look at an example that uses the random_number_generator to generate a vector of random values used in a simulation.

7.4.2 Simulation testing: Estimating plant yield per pot from pot density

In this example, we will create a randomly generated data set to help evaluate the fit of a model. First, however, we must fit an actual data set to the model and estimate the corresponding parameter values. In this case, we'll look at a model that describes the relationship between the density of plants per pot and the plant yield per pot, which is assumed to be:

$$\log(Y_i) = -\log(\alpha + \beta * D_i) + \varepsilon_i \quad (7.14)$$

where D_i is the observed density, Y_i is the yield, and $\varepsilon_i \sim N(0, \sigma^2)$

The data set used in this example consists of ten observations of pot density and corresponding yield data. Click here to see the data file, [pot-density.dat](#).

The following template ([potdensity.tpl](#)) uses maximum likelihood to estimate α , β and σ .

```

DATA_SECTION
    init_int N
    init_vector density(1,N)
    init_vector yield(1,N)
    vector logYield(1,N)
    !! logYield=log(yield);

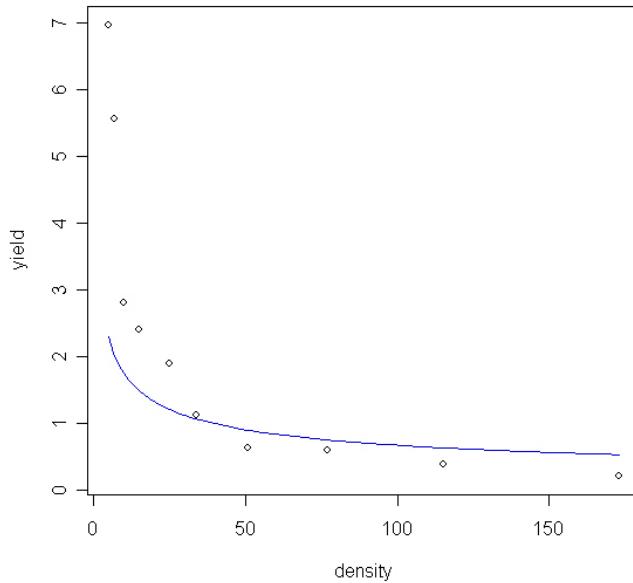
PARAMETER_SECTION
    init_number logA
    init_number logB
    init_number logSigma
    sdreport_number a
    sdreport_number b
    sdreport_number sigma
    vector pred(1,N)
    number ss
    objective_function_value f

PROCEDURE_SECTION
    b=exp(logB);
    a=exp(logA)-b*min(density);
    sigma=exp(logSigma);
    ss=square(sigma);
    pred=-log(a+b*density);
    f=0.5*(N*log(2*M_PI*ss)+sum(square(logYield-pred))/ss);

```

The program returns the following parameter estimates:

index	name	value	std dev
1	logA	-1.9044e+000	1.0966e-001
2	logB	-3.6793e+000	6.7797e-002
3	logSigma	-1.9246e+000	2.2361e-001
4	a	2.2708e-002	2.1107e-002
5	b	2.5241e-002	1.7113e-003
6	sigma	1.4594e-001	3.2633e-002



Using the estimated parameters, we can now generate a random set of values for the dependent variable (*yield*) using the `random_number_generator` function. The only part of the template that changes is the DATA_SECTION:

```
DATA_SECTION
    init_int N
    init_vector density(1,N)
    init_vector yield(1,N)
    vector logYield(1,N)
    // Replace with simulated data
    !! random_number_generator rng(123456);
    !! logYield.fill_rndn(rng);
    !! logYield*= .15; // assumed sd
    !! logYield+= -log(.022+.025*density);
```

The `random_number_generator` uses the specified seed, 123456, to generate a vector of normally distributed values between 0 and 1, which is assigned to the vector `logYield`. Each random value is then multiplied by the standard deviation (.15) and then added to the estimated yield ($\log(Y_i)$) that is calculated using the parameter estimates generated for `a` and `b`. The program can now use the randomly adjusted yield values when it calculates

the parameter estimates, and we can compare the results from the original and the simulated output. Compiling and running the simulation program produces the following results:

index	name	value	std dev
1	logA	-1.8823e+000	8.1064e-002
2	logB	-3.6725e+000	5.0495e-002
3	logSigma	-2.2184e+000	2.2361e-001
4	a	2.5178e-002	1.5895e-002
5	b	2.5412e-002	1.2832e-003
6	sigma	1.0878e-001	2.4324e-002

Change the random seed several times to generate several different sets of random data. Click here to see the [template](#) and [data file](#) used for the simulation.

Chapter 8

Useful operators and functions

Note: This reference was originally compiled by C.V. Minite-Vera in 2000 and expanded for this guide.

8.1 Useful ADMB Functions

Table 8.1 contains some useful functions that can be used in ADMB templates.

Table 8.1: Useful ADMB functions

Function	Definition
<code>active(Par)</code>	Returns a binary: “true” (1) if the parameter <i>Par</i> is active in the current phase and “false” (0) otherwise.
<code>Bolinha.initialize()</code>	Initializes the object <i>bolinha</i> . When an object is initialized, all elements are set equal to zero.
<code>column(matrix, index)</code>	Extracts the indexed column from the matrix. For an example, please see Figure 4.3.
<code>extract_row(matrix, index)</code>	Extracts the indexed row from the matrix.

Continued on next page

Table 8.1 – continued from previous page

Function	Definition
<code>current_phase()</code>	Returns an integer that is the value of the current phase.
<code>exp(x)</code>	Returns e raised to the power x .
<code>last_phase()</code>	Returns a binary: “true” (1) if the current phase is the last phase and “false” (0) otherwise.
<code>log(x)</code>	Returns the natural log of x . For an example, see 7.2.1 .
<code>mceval_phase()</code>	Returns a binary: “true” (1) if the current phase is the mceval phase and “false” (0) otherwise. For an example, see 7.3.2 .
<code>mfexp(x)</code>	Returns the exponential of x , element by element. x can be real or complex.
<code>posfun($x, \text{eps}, \text{pen}$)</code>	The <code>posfun()</code> function is used to ensure that a value remains positive. The syntax is $y = \text{posfun}(x, \text{eps}, \text{pen})$, where $y \leq x$ and $y \geq \text{eps}$ and pen is a penalty function whose value is $0.01 * \text{square}(x - \text{eps})$. x and pen are dvariables. Before calling <code>posfun()</code> , set <code>pen = 0</code> .
<code>pow($base, \text{exp}$)</code>	Returns the $base$ raised to the power exp . For an example, see Section 7.1.1
<code>regression(x)</code>	Calculates the log-likelihood function of the nonlinear least-squares regression. For an example of the <code>regression</code> and <code>robust_regression</code> functions, please see Chapter One of the ADMB User Manual.
<code>sqrt(x)</code>	Returns the square root of x .
<code>square(x)</code>	Returns the square of x . For an example, see Section 7.1.2

8.2 Useful Vector Operations

Table [8.2](#) contains some common vector operations.

Let V, X, Y be vectors and v_i, w_i, y_i be the elements of the

vectors

Let M, N, A be matrices and m_{if} , m_{ij} , a_{ij} be the elements of the matrices

Let nu be a scalar (number)

Note: In cases where there are several ways to code an operation in ADMB, the more efficient ADMB code is indicated in blue beneath the less efficient code.

Table 8.2: Useful Vector Operations

Vector Operation	Mathematical Notation	ADMB code
Addition		
vector + vector	$vi = xi + yi$	<code>V=X+Y</code> <code>X+=Y</code>
scalar + vector	$vi = xi + nu$	<code>V = X + nu</code> <code>X+=nu</code>
sum of vector elements	$nu = \sum ivi$	<code>nu=sum(V)</code>
Subtraction		
vector - vector	$vi = xi - yi$	<code>V=X-Y</code> <code>X-=Y</code>
vector - scalar	$vi = xi - nu$	<code>V=X-nu</code> <code>X-=nu</code>
Multiplication		
scalar * vector	$xi = nu * yi$	<code>X=nu*Y</code> <code>Y*=nu</code>
vector * scalar	$xi = yi * nu$	<code>X=Y*nu</code>
vector * matrix	$xj = \sum_{i=1}^n yi * mij$	<code>X=Y*M</code>
matrix * vector	$xi = \sum_{j=1}^n mij * yi$	<code>X=M*Y</code>
Division		
vector / scalar	$yi = xi/nu$	<code>Y=X/nu</code> <code>X/=nu</code>
scalar / vector	$yi = nu/xi$	<code>Y=nu/X</code>
Continued on next page		

Table 8.2 – continued from previous page

Vector Operation	Mathematical Notation	ADMB code
Element-wise (e-w) operations (vectors must have the same dimensions)		
e-w multiplication	$vi = xi * yi$	<code>V=elem_prod(X,Y)</code>
e-w division	$vi = xi / yi$	<code>V=elem_div(X,Y)</code>
Other Operations		
Concatenation	$X = (x1, x2, x3)$ $Y = (y1, y2)$ $V = (x1, x2, x3, y1, y2)$	<code>V=X&Y</code>
Dot Product	$nu = \sum xi * yi$	<code>nu=X*Y</code>
Maximum Value		<code>nu=max(V)</code>
Minimum Value		<code>nu=min(V)</code>
Norm	$nu = \sqrt{(\sum_{i=1}^n vi^2)}$	<code>nu=norm(V)</code>
Norm Square	$nu = \sum_{i=1}^n vi^2$	<code>nu=norm2(V)</code>
Outer Product	$mij = xi * yi$	<code>M=outer_prod(X,Y)</code>

8.3 Useful Matrix Operations

Table 8.3 contains some common matrix operations.

Let V, X, Y be vectors and vi, wi, yi be the elements of the vectors

Let M, N, A be matrices and mif, mij, aij be the elements of the matrices

Let nu be a scalar (number)

Note: In cases where there are several ways to code an operation in ADMB, the more efficient ADMB code is indicated in blue beneath the less efficient code.

Table 8.3: Useful Matrix Operations

Matrix Operation	Mathematical Notation	ADMB code
Addition		
matrix + matrix	$a_{ij} = m_{ij} + n_{ij}$	<code>A=M+N</code>
scalar + matrix	$a_{ij} = m_{ij} + nu$	<code>A=M+nu</code> <code>M+=nu</code>
Subtraction		
matrix - scalar	$a_{ij} = m_{ij} - nu$	<code>A=M-nu</code> <code>M-=nu</code>
matrix - matrix	$a_{ij} = m_{ij} - n_{ij}$	<code>A=M-N</code> <code>M-=N</code>
Multiplication		
scalar * matrix	$m_{ij} = nu * a_{ij}$	<code>A=nu*M</code> <code>M*=nu</code>
vector * matrix	$x_j = \sum_{i=1}^n y_i * m_{ij}$	<code>X=Y*M</code>
matrix * vector	$x_i = \sum_{j=1}^n m_{ij} * y_i$	<code>X=M*Y</code>
matrix * matrix	$a_{if} = \sum_{k=1}^n m_{ik} * n_{kj}$	<code>A=M*N</code>
Division		
matrix / scalar	$m_{ij} = a_{ij}/nu$	<code>M=N/nu</code>
Element-wise (e-w) operations (matrices must have the same dimensions)		
e-w multiplication	$m_{ij} = a_{ij} * n_{ij}$	<code>M=elem_prod(A,N)</code>
e-w division	$m_{ij} = a_{ij}/n_{ij}$	<code>M=elem_div(A,N)</code>
Other Operations		
column sum	$y_j = \sum_{i=1}^n m_{ij}$	<code>Y=colsum(M)</code>
determinant of symmetric matrix		<code>nu=det(M)</code>
eigenvalues of a symmetric matrix		<code>V=eigenvalues(M)</code>
eigenvectors of a symmetric matrix		<code>N=eigenvectors(M)</code>
Continued on next page		

Table 8.3 – continued from previous page

Matrix Operation	Mathematical Notation	ADMB code
identity matrix function		M=identity_matrix(int min, int max)
inverse of a symmetric matrix		N=inv(M)
log of the determinant		nu=ln_det(M, sgn) (sgn is an integer)
norm	$nu = \sqrt{\sum_{i=1}^n \sum_{j=1}^m mij^2}$	nu=norm(M)
norm square	$nu = \sum_{i=1}^n \sum_{j=1}^m mij^2$	nu=norm2(M)
row sum	$xi = \sum_{j=1}^n mij$	nu=rowsum(M)