Random Effects in AD Model Builder

ADMB-RE User Guide

Version 11.1 (2013-05-01)

Hans Skaug & David Fournier

ADMB Foundation, Honolulu.

This is the manual for AD Model Builder with Random Effects (ADMB-RE) version 10.0.

The latest edition of the manual is available at:
`http://admb-project.org/documentation/manuals/admb-user-manuals`

Contents

Preface

A comment about notation:

Important points are emphasized with a star

★ like this.

Please submit all comments and complaints by email to users@admb-project.org.

Chapter 1

Introduction

This document is a user's guide to random-effects modelling in AD Model Builder (ADMB). Random effects are a feature of ADMB, in the same way as profile likelihoods are, but are sufficiently complex to merit a separate user manual. The work on the random-effects "module" (ADMB-RE) started around 2003. The pre-existing part of ADMB (and its evolution) is referred to as "ordinary" ADMB in the following. This manual refers to Version 9.0.x of ADMB (and ADMB-RE).

Before you start with random effects, it is recommended that you have some experience with ordinary ADMB. This manual tries to be self-contained, but it is clearly an advantage if you have written (and successfully run) a few TPL files. Ordinary ADMB is described in the ADMB manual [4], which is available from admb-project.org. If you are new to ADMB, but have experience with C++ (or a similar programming language), you may benefit from taking a look at the quick references in Appendix D.

ADMB-RE is very flexible. The term "random effect" seems to indicate that it only can handle mixed-effect regression type models, but this is very misleading. "Latent variable" would have been a more precise term. It can be argued that ADMB-RE is the most flexible latent variable framework around. All the mixed-model stuff in software packages, such as allowed by R, Stata, SPSS, etc., allow only very specific models to be fit. Also, it is impossible to change the distribution of the random effects if, say, you wanted to do that. The NLMIXED macro in SAS is more flexible, but cannot handle state-space models or models with crossed random effects. WinBUGS is the only exception, and its ability to handle discrete latent variables is a bit more flexible than is ADMB-RE's. However, WinBUGS does all its computations using MCMC exclusively, while ADMB-RE lets the user choose between maximum likelihood estimation (which, in general, is much faster) and MCMC.

An important part of the ADMB-RE documentation is the example collection, which is described in Appendix A. As with the example collections for ADMB and AUTODIF, you will find fully worked examples, including data and code for the model. The examples have been selected to illustrate the various aspects of ADMB-RE, and are frequently referred to throughout this manual.

1.1 Summary of features

Why use AD Model Builder for creating nonlinear random-effects models? The answer consists of three words: "flexibility," "speed," and "accuracy." To illustrate these points. a number of examples comparing ADMB-RE with two existing packages: NLME, which runs on R and Splus, and WinBUGS. In general, NLME is rather fast and it is good for the problems for which it was designed, but it is quite inflexible. What is needed is a tool with at least the computational power of NLME yet the flexibility to deal with arbitrary nonlinear random-effects models. In Section 2.4, we consider a thread from the R user list, where a discussion took place about extending a model to use random effects with a log-normal, rather than normal, distribution. This appeared to be quite difficult. With ADMB-RE, this change takes one line of code. WinBUGS, on the other hand, is very flexible, and many random-effects models can be easily formulated in it. However, it can be very slow. Furthermore, it is necessary to adopt a Bayesian perspective, which may be a problem for some applications. A model which runs 25 times faster under ADMB than under WinBUGS may be found in Section A.1.1.

1.1.1 Model formulation

With ADMB, you can formulate and fit a large class of nonlinear statistical models. With ADMB-RE, you can include random effects in your model. Examples of such models include:

- Generalized linear mixed models (logistic and Poisson regression).
- Nonlinear mixed models (growth curve models, pharmacokinetics).
- State space models (nonlinear Kalman filters).
- Frailty models in survival analysis.
- Nonparametric smoothing.
- Semiparametric modelling.
- Frailty models in survival analysis.
- Bayesian hierarchical models.
- General nonlinear random-effects models (fisheries catch-at-age models).

You formulate the likelihood function in a template file, using a language that resembles C++. The file is compiled into an executable program (on Linux or Windows). The whole C++ language is to your disposal, giving you great flexibility with respect to model formulation.

1.1.2 Computational basis of ADMB-RE

- Hyper-parameters (variance components, etc.) estimated by maximum likelihood.
- Marginal likelihood evaluated by the Laplace approximation, (adaptive) importance sampling or Gauss-Hermite integration.
- Exact derivatives calculated using Automatic Differentiation.
- Sampling from the Bayesian posterior using MCMC (Metropolis-Hastings algorithm).
- Most of the features of ordinary ADMB (matrix arithmetic and standard errors, etc.) are available.
- Sparse matrix libraries, useful for Markov random fields and crossed random effects, are available.

1.1.3 The strengths of ADMB-RE

- *Flexibility:* You can fit a large variety of models within a single framework.
- *Convenience:* Computational details are transparent. Your only responsibility is to formulate the log-likelihood.
- *Computational efficiency:* ADMB-RE is up to 50 times faster than WinBUGS.
- *Robustness:* With exact derivatives, you can fit highly nonlinear models.
- *Convergence diagnostic:* The gradient of the likelihood function provides a clear convergence diagnostic.

1.1.4 Program interface

- *Model formulation*: You fill in a C++-based template using your favorite text editor.
- *Compilation*: You turn your model into an executable program using a C++ compiler (which you need to install separately).
- *Platforms*: Windows, Linux and Mac.

1.1.5 How to obtain ADMB-RE

ADMB-RE is a module for ADMB. Both can be obtained from admb-project.org.

Chapter 2

The Language and the Program

2.1 What is ordinary ADMB?

ADMB is a software package for doing parameter estimation in nonlinear models. It combines a flexible mathematical modelling language (built on C++) with a powerful function minimizer (based on Automatic Differentiation). The following features of ADMB make it very useful for building and fitting nonlinear models to data:

- Vector-matrix arithmetic and vectorized operations for common mathematical functions.
- Reading and writing vector and matrix objects to a file.
- Fitting the model is a stepwise manner (with "phases"), where more and more parameters become active in the minimization.
- Calculating standard deviations of arbitrary functions of the model parameters by the "delta method."
- MCMC sampling around the posterior mode.

To use random effects in ADMB, it is recommended that you have some experience in writing ordinary ADMB programs. In this section, we review, for the benefit of the reader without this experience, the basic constructs of ADMB.

Model fitting with ADMB has three stages: 1) model formulation, 2) compilation and 3) program execution. The model fitting process is typically iterative: after having looked at the output from stage 3, one goes back to stage 1 and modifies some aspect of the program.

2.1.1 Writing an ADMB program

To fit a statistical model to data, we must carry out certain fundamental tasks, such as reading data from file, declaring the set of parameters that should be estimated, and finally

giving a mathematical description of the model. In ADMB, you do all of this by filling in a template, which is an ordinary text file with the file-name extension `.tpl` (and hence the template file is known as the TPL file). You therefore need a text editor—such as *vi* under Linux, or Notepad under Windows—to write the TPL file. The first TPL file to which the reader of the ordinary ADMB manual is exposed is `simple.tpl` (listed in Section 2.3 below). We shall use `simple.tpl` as our generic TPL file, and we shall see that introduction of random effects only requires small changes to the program.

A TPL file is divided into a number of "sections," each representing one of the fundamental tasks mentioned above. See Table 2.1 for the required sections.

Name	Purpose
`DATA_SECTION`	Declare "global" data objects, initialization from file.
`PARAMETER_SECTION`	Declare independent parameters.
`PROCEDURE_SECTION`	Specify model and objective function in C++.

Table 2.1: Required sections.

More details are given when we later look at `simple.tpl`, and a quick reference card is available in Appendix D.

2.1.2 Compiling an ADMB program

After having finished writing `simple.tpl`, we want to convert it into an executable program. This is done in a DOS-window under Windows, and in an ordinary terminal window under Linux. To compile `simple.tpl`, we would, under both platforms, give the command:

```
$ admb -r simple
```

Here, `$` is the command line prompt (which may be a different symbol, or symbols, on your computer), and `-r` is an option telling the program `admb` that your model contains random effects. The program `admb` accepts another option, `-s`, which produces the "safe" (but slower) version of the executable program. The `-s` option should be used in a debugging phase, but it should be skipped when the final production version of the program is generated.

The compilation process really consists of two steps.

In the first step, `simple.tpl` is converted to a C++ program by a preprosessor called `tpl2rem` in the case of ADMB-RE, and `tpl2cpp` in the case of ordinary ADMB (see Appendix D). An error message from `tpl2rem` consists of a single line of text, with a reference to the line in the TPL file where the error occurs. If successful, the first compilation step results in the C++ file `simple.cpp`.

In the second step, `simple.cpp` is compiled and linked using an ordinary C++ compiler (which is not part of ADMB). Error messages during this phase typically consist of long printouts, with references to line numbers in `simple.cpp`. To track down syntax errors, it

may occasionally be useful to look at the content of `simple.cpp`. When you understand what is wrong in `simple.cpp`, you should go back and correct `simple.tpl` and re-enter the command `admb -r simple`. When all errors have been removed, the result will be an executable file, which is called either `simple.exe` under Windows or `simple` under Linux. The compilation process is illustrated in Section D.1.

2.1.3 Running an ADMB-program

The executable program is run in the same window as it was compiled. Note that data are not usually part of the ADMB program (e.g., `simple.tpl`). Instead, data are being read from a file with the file name extension `.dat` (e.g., `simple.dat`). This brings us to the naming convention used by ADMB programs for input and output files: the executable automatically infers file names by adding an extension to its own name. The most important files are listed in Table 2.2. You can use command line options to modify the behavior of the program at

	File name	Contents
Input	`simple.dat`	Data for the analysis
	`simple.pin`	Initial parameter values
Output	`simple.par`	Parameter estimates
	`simple.std`	Standard deviations
	`simple.cor`	Parameter correlations

Table 2.2: File naming convention.

runtime. The available command line options can be listed by typing:

```
$ simple -?
```

(or whatever your executable is called in place of `simple`). The command line options that are specific to ADMB-RE are listed in Appendix C, and are discussed in detail under the various sections. An option you probably will like to use during an experimentation phase is `-est`, which turns off calculation of standard deviations, and hence reduces the running time of the program.

2.1.4 ADMB-IDE: easy and efficient user interface

The graphical user interface to ADMB by Arni Magnusson simplifies the process of building and running the model, especially for the beginner [1]. Among other things, it provides syntax highlighting and links error messages from the C++ compiler to the `.cpp` file.

2.1.5 Initial values

The initial values can be provided in different ways (see the ordinary ADMB manual). Here, we only describe the `.pin` file approach. The `.pin` file should contain legal values (within the bounds) for all the parameters, including the random effects. The values must be given in the same order as the parameters are defined in the `.tpl` file. The easiest way of generating a `.pin` file with the right structure is to first run the program with a `-maxfn 0` option (for this, you do not need a `.pin` file), then copy the resulting `.p01` file into `.pin` file, and then edit it to provide the correct numeric values. More information about what initial values for random effects really mean is given in Section 3.7.

2.2 Why random effects?

Many people are familiar with the method of least squares for parameter estimation. Far fewer know about random effects modeling. The use of random effects requires that we adopt a statistical point of view, where the sum of squares is interpreted as being part of a likelihood function. When data are correlated, the method of least squares is sub-optimal, or even biased. But relax—random effects come to rescue!

The classical motivation of random effects is:

- To create parsimonious and interpretable correlation structures.
- To account for additional variation or overdispersion.

We shall see, however, that random effects are useful in a much wider context, for instance, in non-parametric smoothing.

2.2.1 Statistical prerequisites

To use random effects in ADMB, you must be familiar with the notion of a random variable and, in particular, with the normal distribution. In case you are not, please consult a standard textbook in statistics. The notation $u \sim N(\mu, \sigma^2)$ is used throughout this manual, and means that u has a normal (Gaussian) distribution with expectation μ and variance σ^2. The distribution placed on the random effects is called the "prior," which is a term borrowed from Bayesian statistics.

A central concept that originates from generalized linear models is that of a "linear predictor." Let $x_1, \ldots, x_p$ denote observed covariates (explanatory variables), and let $\beta_1, \ldots, \beta_p$ be the corresponding regression parameters to be estimated. Many of the examples in this manual involve a linear predictor $\eta_i = \beta_1 x_{1,i} + \cdots + \beta_p x_{p,i}$, which we also will write in vector form as $\eta = \mathbf{X}\beta$.

2.2.2 Frequentist or Bayesian statistics?

A pragmatic definition of a "frequentist" is a person who prefers to estimate parameters by the method of maximum likelihood. Similarly, a "Bayesian" is a person who uses MCMC techniques to generate samples from the posterior distribution (typically with noninformative priors on hyper-parameters), and from these samples generates some summary statistic, such as the posterior mean. With its `-mcmc` runtime option, ADMB lets you switch freely between the two worlds. The approaches complement each other rather than being competitors. A maximum likelihood fit (point estimate + covariance matrix) is a step-1 analysis. For some purposes, step-1 analysis is sufficient. In other situations, one may want to see posterior distributions for the parameters. In such situations, the established covariance matrix (inverse Hessian of the log-likelihood) is used by ADMB to implement an efficient Metropolis-Hastings algorithm (which you invoke with `-mcmc`).

2.2.3 A simple example

We use the `simple.tpl` example from the ordinary ADMB manual to exemplify the use of random effects. The statistical model underlying this example is the simple linear regression

$$Y_i = ax_i + b + \varepsilon_i, \qquad i = 1, \ldots, n,$$

where Y_i and x_i are the data, a and b are the unknown parameters to be estimated, and $\varepsilon_i \sim N(0, \sigma^2)$ is an error term.

Consider now the situation where we do not observe x_i directly, but rather observe

$$X_i = x_i + e_i,$$

where e_i is a measurement error term. This situation frequently occurs in observational studies, and is known as the "error-in-variables" problem. Assume further that $e_i \sim N(0, \sigma_e^2)$, where σ_e^2 is the measurement error variance. For reasons discussed below, we shall assume that we know the value of σ_e, so we shall pretend that $\sigma_e = 0.5$.

Because x_i is not observed, we model it as a random effect with $x_i \sim N(\mu, \sigma_x^2)$. In ADMB-RE, you are allowed to make such definitions through the new parameter type called `random_effects_vector`. (There is also a `random_effects_matrix`, which allows you to define a matrix of random effects.)

1. Why do we call x_i a "random effect," while we do not use this term for X_i and Y_i (though they clearly are "random")? The point is that X_i and Y_i are observed directly, while x_i is not. The term "random effect" comes from regression analysis, where it means a random regression coefficient. In a more general context. "latent random variable" is probably a better term.

2. The unknown parameters in our model are: a, b, μ, σ, σ_x, and $x_1, \ldots, x_n$. We have agreed to call $x_1, \ldots, x_n$ "random effects." The rest of the parameters are called "hyper-parameters." Note that we place no prior distribution on the hyper-parameters.

3. Random effects are integrated out of the likelihood, while hyper-parameters are estimated by maximum likelihood. This approach is often called "empirical Bayes," and will be considered a frequentist method by most people. There is however nothing preventing you from making it "more Bayesian" by putting priors (penalties) on the hyper-parameters.

4. A statistician will say, "This model is nothing but a bivariate Gaussian distribution for (X, Y), and we don't need any random effects in this situation." This is formally true, because we could work out the covariance matrix of (X, Y) by hand and fit the model using ordinary ADMB. This program would probably run much faster, but it would have taken us longer to write the code without declaring x_i to be of type `random_effects_vector`. However, more important is that random effects can be used also in non-Gaussian (nonlinear) models where we are unable to derive an analytical expression for the distribution of (X, Y).

5. Why didn't we try to estimate σ_e? Well, let us count the parameters in the model: a, b, μ, σ, σ_x, and σ_e. There are six parameters total. We know that the bivariate Gaussian distribution has only five parameters (the means of X and Y and three free parameters in the covariate matrix). Thus, our model is not identifiable if we also try to estimate σ_e. Instead, we pretend that we have estimated σ_e from some external data source. This example illustrates a general point in random effects modelling: you must be careful to make sure that the model is identifiable!

2.3 A code example

Here is the random effects version of `simple.tpl`:

```
DATA_SECTION
 init_int nobs
 init_vector Y(1,nobs)
 init_vector X(1,nobs)

PARAMETER_SECTION
 init_number a
 init_number b
 init_number mu
 vector pred_Y(1,nobs)
 init_bounded_number sigma_Y(0.000001,10)
 init_bounded_number sigma_x(0.000001,10)
 random_effects_vector x(1,nobs)
 objective_function_value f

PROCEDURE_SECTION            // This section is pure C++
```

```
f = 0;
pred_Y=a*x+b;                // Vectorized operations

// Prior part for random effects x
f += -nobs*log(sigma_x) - 0.5*norm2((x-mu)/sigma_x);

// Likelihood part
f += -nobs*log(sigma_Y) - 0.5*norm2((pred_Y-Y)/sigma_Y);
f += -0.5*norm2((X-x)/0.5);

f *= -1; // ADMB does minimization!
```

Comments

1. Everything following `//` is a comment.
2. In the `DATA_SECTION`, variables with a `init_` in front of the data type are read from file.
3. In the `PARAMETER_SECTION`
   - Variables with a `init_` in front of the data type are the hyper-parameters, i.e., the parameters to be estimated by maximum likelihood.
   - `random_effects_vector` defines the random effect vector. (There is also a type called `random_effects_matrix`.) There can be more than one such object, but they must all be defined after the hyper-parameters are—otherwise, you will get an error message from the preprocessor `tpl2rem`.
   - Objects that are neither hyper-parameters nor random effects are ordinary programming variables that can be used in the `PROCEDURE_SECTION`. For instance, we can assign a value to the vector `pred_Y`.
   - The objective function should be defined as the last variable.
4. The `PROCEDURE_SECTION` basically consists of standard C++ code, the primary purpose of which is to calculate the value of the objective function.
   - Variables defined in `DATA_SECTION` and `PARAMETER_SECTION` may be used.
   - Standard C++ functions, as well as special ADMB functions, such as `norm2(x)` (which calculates $\sum x_i^2$), may be used.
   - Often the operations are vectorized, as in the case of `simple.tpl`
   - The objective function should be defined as the last variable.
   - ADMB does minimization, rather than optimization. Thus, the sign of the log-likelihood function `f` is changed in the last line of the code.

2.3.1 Parameter estimation

We learned above that hyper-parameters are estimated by maximum likelihood, but what if we also are interested in the value of the random effects? For this purpose, ADMB-RE offers an "empirical Bayes" approach, which involves fixing the hyper-parameters at their maximum likelihood estimates, and treating the random effects as the parameters of the model. ADMB-RE automatically calculates "maximum posterior" estimates of the random effects for you. Estimates of both hyper-parameters and random effects are written to `simple.par`.

2.4 The flexibility of ADMB-RE

Say that you doubt the distributional assumption $x_i \sim N(\mu, \sigma_x^2)$ made in `simple.tpl`, and that you want to check if a skewed distribution gives a better fit. You could, for instance, take

$$x_i = \mu + \sigma_x \exp(z_i), \qquad z_i \sim N(0,1).$$

Under this model, the standard deviation of x_i is proportional to, but not directly equal to, σ_x. It is easy to make this modification in `simple.tpl`. In the `PARAMETER_SECTION`, we replace the declaration of `x` by

```
vector x(1,nobs)
random_effects_vector z(1,nobs)
```

and in the `PROCEDURE_SECTION` we replace the prior on `x` by

```
f = - 0.5*norm2(z);
x = mu + sigma_x*exp(z);
```

This example shows one of the strengths of ADMB-RE: it is very easy to modify models. In principle, you can implement any random effects model you can think of, but as we shall discuss later, there are limits to the number of random effects you can declare.

Chapter 3

Random Effects Modeling

This chapter describes all ADMB-RE features except those related to "separability," which are dealt with in Chapter 4. Separability, or the Markov property, as it is called in statistics, is a property possessed by many model classes. It allows ADMB-RE to generate more efficient executable programs. However, most ADMB-RE concepts and techniques are better learned and understood without introducing separability. Throughout much of this chapter, we will refer to the program `simple.tpl` from Section 2.3.

3.1 The objective function

As with ordinary ADMB, the user specifies an objective function in terms of data and parameters. However, in ADMB-RE, the objective function must have the interpretation of being a (negative) log-likelihood. One typically has a hierarchical specification of the model, where at the top layer, data are assumed to have a certain probability distribution conditionally on the random effects (and the hyper-parameters), and at the next level, the random effects are assigned a prior distribution (typically, normal). Because conditional probabilities are multiplied to yield the joint distribution of data and random effects, the objective function becomes a sum of (negative) log-likelihood contributions, and the following rule applies:

★ The order in which the different log-likelihood contributions are added to the objective function does not matter.

An addition to this rule is that all programming variables have their values assigned before they enter in a prior or a likelihood expression. WinBUGS users must take care when porting their programs to ADMB, because this is not required in WinBUGS.

The reason why the *negative* log-likelihood is used is that for historical reasons, ADMB does minimization (as opposed to maximization). In complex models, with contributions to the log-likelihood coming from a variety of data sources and random effects priors, it is recommended that you collect the contributions to the objective function using the `-=` operator of C++, i.e.,

```
f -= -nobs*log(sigma_x) - 0.5*norm2((x-mu)/sigma_x);
```

By using `-=` instead of `+=`, you do not have to change the sign of every likelihood expression—which would be a likely source of error. When none of the advanced features of Chapter 4 are used, you are allowed to switch the sign of the objective function at the end of the program:

```
f *= -1; // ADMB does minimization!
```

so that in fact, `f` can hold the value of the log-likelihood until the last line of the program.

It is OK to ignore constant terms ($0.5\log(2\pi)$, for the normal distribution) as we did in `simple.tpl`. This only affects the objective function value, not any other quantity reported in the `.par` and `.std` files (not even the gradient value).

3.2 The random effects distribution (prior)

In `simple.tpl`, we declared $x_1, \ldots, x_n$ to be of type `random_effects_vector`. This statement tells ADMB that $x_1, \ldots, x_n$ should be treated as random effects (i.e., be the targets for the Laplace approximation), but it does not say anything about what distribution the random effects should have. We assumed that $x_i \sim N(\mu, \sigma_x^2)$, and (without saying it explicitly) that the x_is were statistically independent. We know that the corresponding prior contribution to the log-likelihood is

$$-n\log(\sigma_x) - \frac{1}{2\sigma_x^2}\sum_{i=1}(x_i - \mu)^2$$

with ADMB implementation

```
f += -nobs*log(sigma_x) - 0.5*norm2((x-mu)/sigma_x);
```

Both the assumption about independence and normality can be generalized—as we shortly will do—but first we introduce a transformation technique that forms the basis for much of what follows later.

3.2.1 Scaling of random effects

A frequent source of error when writing ADMB-RE programs is that priors get wrongly specified. The following trick can make the code easier to read, and has the additional advantage of being numerically stable for small values of σ_x. From basic probability theory, we know that if $u \sim N(0,1)$, then $x = \sigma_x u + \mu$ will have a $N(\mu, \sigma_x^2)$ distribution. The corresponding ADMB code would be

```
f += - 0.5*norm2(u);
x = sigma_x*u + mu;
```

(This, of course, requires that we change the type of `x` from `random_effects_vector` to `vector`, and that `u` is declared as a `random_effects_vector`.)

The trick here was to start with a $N(0,1)$ distributed random effect `u` and to generate random effects `x` with another distribution. This is a special case of a transformation.

Had we used a non-linear transformation, we would have gotten an `x` with a non-Gaussian distribution. The way we obtain correlated random effects is also transformation based. However, as we shall see in Chapter 4, transformation may "break" the separability of the model, so there are limitations as to what transformations can do for you.

3.3 Correlated random effects

In some situations, you will need correlated random effects, and as part of your problem, you may want to estimate the elements of the covariance matrix. A typical example is mixed regression, where the intercept random effect (u_i) is correlated with the slope random effect (v_i):

$$y_{ij} = (a + u_i) + (b + v_i)\, x_{ij} + \varepsilon_{ij}.$$

(If you are not familiar with the notation, please consult an introductory book on mixed regression, such [9].) In this case, we can define the correlation matrix

$$C = \begin{bmatrix} 1 & \rho \\ \rho & 1 \end{bmatrix},$$

and we want to estimate ρ along with the variances of u_i and v_i. Here, it is trivial to ensure that C is positive-definite, by requiring $-1 < \rho < 1$, but in higher dimensions, this issue requires more careful consideration.

To ensure that C is positive-definite, you can parameterize the problem in terms of the Cholesky factor L, i.e., $C = LL'$, where L is a lower diagonal matrix with positive diagonal elements. There are $q(q-1)/2$) free parameters (the non-zero elements of L) to be estimated, where q is the dimension of C. Since C is a correlation matrix, we must ensure that its diagonal elements are unity. An example with $q = 4$ is

```
PARAMETER_SECTION
  matrix L(1,4,1,4)                 // Cholesky factor
  init_vector a(1,6)                // Free parameters in C
  init_bounded_vector B(1,4,0,10)   // Standard deviations

PROCEDURE_SECTION

  int k=1;
  L(1,1) = 1.0;
  for(i=2;i<=4;i++)
  {
    L(i,i) = 1.0;
    for(j=1;j<=i-1;j++)
      L(i,j) = a(k++);
    L(i)(1,i) /= norm(L(i)(1,i));  // Ensures that C(i,i) = 1
  }
```

Given the Cholesky factor L, we can proceed in different directions. One option is to use the same transformation-of-variable technique as above: Start out with a vector u of independent $N(0,1)$ distributed random effects. Then, the vector

```
x = L*u;
```

has correlation matrix $C = LL'$. Finally, we multiply each component of `x` by the appropriate standard deviation:

```
y = elem_prod(x,sigma);
```

3.3.1 Large structured covariance matrices

In some situations, for instance, in spatial models, q will be large ($q = 100$, say). Then it is better to use the approach outlined in Section 4.5.

3.4 Non-Gaussian random effects

Usually, the random effects will have a Gaussian distribution, but technically speaking, there is nothing preventing you from replacing the normality assumption, such as

```
f -= -nobs*log(sigma_x) - 0.5*norm2((x-mu)/sigma_x);
```

with a log gamma density, say. It can, however, be expected that the Laplace approximation will be less accurate when you move away from normal priors. Hence, you should instead use the transformation trick that we learned earlier, but now with a non-linear transformation. A simple example of this yielding a log-normal prior was given in Section 2.4.

Say you want x to have cumulative distribution function $F(x)$. It is well known that you achieve this by taking $x = F^{-1}(\Phi(u))$, where Φ is the cumulative distribution function of the $N(0,1)$ distribution. For a few common distributions, the composite transformation $F^{-1}(\Phi(u))$ has been coded up for you in ADMB-RE, and all you have to do is:

1. Define a random effect u with a $N(0,1)$ distribution.
2. Transform u into a new random effect x using one of `something_deviate` functions described below.

where `something` is the name of the distribution.

As an example, say we want to obtain a vector `x` of gamma distributed random effects (probability density $x^{a-1}\exp(-x)/\Gamma(a)$). We can then use the code:

```
PARAMETER_SECTION
  init_number a                              // Shape parameter
  init_number lambda                         // Scale parameter
  vector x(1,n)
  random_effects_vector u(1,n)
```

```
  objective_function_value g

PROCEDURE_SECTION
  g -= -0.5*norm2(u);                          // N(0,1) likelihood contr.
  for (i=1;i<=n;i++)
    x(i) = lambda*gamma_deviate(u(i),a);
```

See a full example here.

Similarly, to obtain beta(a, b) distributed random effects, with density $f(x) \propto x^{a-1}(1-x)^{b-1}$, we use:

```
PARAMETER_SECTION
  init_number a
  init_number b

PROCEDURE_SECTION
  g -= -0.5*norm2(u);                           // N(0,1) likelihood contr.
  for (i=1;i<=n;i++)
    x(i) = beta_deviate(u(i),a,b);
```

The function `beta_deviate()` has a fourth (optional) parameter that controls the accuracy of the calculations. To learn more about this, you will have to dig into the source code. You find the code for `beta_deviate()` in the file `df1b2betdev.cpp`. The mechanism for specifying default parameter values are found in the source file `df1b2fun.h`.

A third example is provided by the "robust" normal distribution with probability density

$$f(x) = 0.95\frac{1}{\sqrt{2\pi}}e^{-0.5x^2} + 0.05\frac{1}{c\sqrt{2\pi}}e^{-0.5(x/c)^2}$$

where c is a "robustness" parameter which by default is set to $c = 3$ in `df1b2fun.h`. Note that this is a mixture distribution consisting of 95% $N(0, 1)$ and 5% $N(0, c^2)$. The corresponding ADMB-RE code is

```
PARAMETER_SECTION
  init_number sigma                            // Standard deviations (almost)
  number c

PROCEDURE_SECTION
  g -= - 0.5*norm2(u);  // N(0,1) likelihood contribution from u's
  for (i=1;i<=n;i++)
  {
    x(i) = sigma*robust_normal_mixture_deviate(u(i),c);
  }
```

3.4.1 Can a, b, and c be estimated?

As indicated by the data types used above,

★ a and b are among the parameters that are being estimated.

★ c cannot be estimated.

It would, however, be possible to write a version of `robust_normal_mixture_deviate` where also c and the mixing proportion (fixed at 0.95 here) can be estimated. For this, you need to look into the file `df1b2norlogmix.cpp`. The list of distribution that can be used is likely to be expanded in the future.

3.5 Built-in data likelihoods

In the simple `simple.tpl`, the mathematical expressions for all log-likelihood contributions where written out in full detail. You may have hoped that for the most common probability distributions, there were functions written so that you would not have to remember or look up their log-likelihood expressions. If your density is among those given in Table 3.1, you are lucky. More functions are likely to be implemented over time, and user contributions are welcomed!

We stress that these functions should be only be used for data likelihoods, and in fact, they will not compile if you try to let X be a random effect. So, for instance, if you have observations x_i that are Poisson distributed with expectation μ_i, you would write

```
for (i=1;i<=n;i++)
  f -= log_density_poisson(x(i),mu(i));
```

Note that functions do not accept vector arguments.

Density	Expression	Parameters	Name
Poisson	$\frac{\mu^x}{\Gamma(x+1)}e^{-\mu}$	$\mu > 0$	`log_density_poisson`
Neg. binomial	$\mu = E(X)$, $\tau = \frac{Var(X)}{E(X)}$	$\mu, \tau > 0$	`log_negbinomial_density`

Table 3.1: Distributions that currently can be used as high-level data distributions (for data X) in ADMB-RE. The expression for the negative binomial distribution is omitted, due to its somewhat complicated form. Instead, the parameterization, via the overdispersion coefficient, is given. The interested reader can look at the actual implementation in the source file `df1b2negb.cpp`

.

3.6 Phases

A very useful feature of ADMB is that it allows the model to be fit in different phases. In the first phase, you estimate only a subset of the parameters, with the remaining parameters being fixed at their initial values. In the second phase, more parameters are turned on, and so it goes. The phase in which a parameter becomes active is specified in the declaration of the parameter. By default, a parameter has phase 1. A simple example would be:

```
PARAMETER_SECTION
 init_number a(1)
 random_effects_vector b(1,10,2)
```

where `a` becomes active in phase 1, while `b` is a vector of length 10 that becomes active in phase 2. With random effects, we have the following rule-of-thumb (which may not always apply):

Phase 1 Activate all parameters in the data likelihood, except those related to random effects.

Phase 2 Activate random effects and their standard deviations.

Phase 3 Activate correlation parameters (of random effects).

In complicated models, it may be useful to break Phase 1 into several sub-phases.

During program development, it is often useful to be able to completely switch off a parameter. A parameter is inactivated when given phase -1, as in

```
PARAMETER_SECTION
 init_number c(-1)
```

The parameter is still part of the program, and its value will still be read from the `pin` file, but it does not take part in the optimization (in any phase).

For further details about phases, please consult the section "Carrying out the minimization in a number of phases" in the ADMB manual [4].

3.7 Penalized likelihood and empirical Bayes

The main question we answer in this section is how are random effects estimated, i.e., how are the values that enter the `.par` and `.std` files calculated? Along the way, we will learn a little about how ADMB-RE works internally.

By now, you should be familiar with the statistical interpretation of random effects. Nevertheless, how are they treated internally in ADMB-RE? Since random effects are not observed data, then they have parameter status—but we distinguish them from hyper-parameters. In the marginal likelihood function used internally by ADMB-RE to estimate hyper-parameters, the random effects are "integrated out." The purpose of the integration is to generate the marginal probability distribution for the observed quantities, which are X

and Y in `simple.tpl`. In that example, we could have found an analytical expression for the marginal distribution of (X, Y), because only normal distributions were involved. For other distributions, such as the binomial, no simple expression for the marginal distribution exists, and hence we must rely on ADMB to do the integration. In fact, the core of what ADMB-RE does for you is automatically calculate the marginal likelihood during its effort to estimate the hyper-parameters.

The integration technique used by ADMB-RE is the so-called Laplace approximation [11]. Somewhat simplified, the algorithm involves iterating between the following two steps:

1. The "penalized likelihood" step: maximizing the likelihood with respect to the random effects, while holding the value of the hyper-parameters fixed. In `simple.tpl`, this means doing the maximization w.r.t. `x` only.

2. Updating the value of the hyper-parameters, using the estimates of the random effects obtained in item 1.

The reason for calling the objective function in step 1, a penalized likelihood, is that the prior on the random effects acts as a penalty function.

We can now return to the role of the initial values specified for the random effects in the `.pin` file. Each time step 1 above is performed, these values are used—unless you use the command line option `-noinit`, in which case the previous optimum is used as the starting value.

Empirical Bayes is commonly used to refer to Bayesian estimates of the random effects, with the hyper-parameters fixed at their maximum likelihood estimates. ADMB-RE uses maximum *aposteriori* Bayesian estimates, as evaluated in step 1 above. Posterior expectation is a more commonly used as Bayesian estimator, but it requires additional calculations, and is currently not implemented in ADMB-RE. For more details, see [11].

The classical criticism of empirical Bayes is that the uncertainty about the hyper-parameters is ignored, and hence that the total uncertainty about the random effects is underestimated. ADMB-RE does, however, take this into account and uses the following formula:

$$\mathrm{cov}(u) = -\left[\frac{\partial^2 \log p(u \mid, \mathrm{data}; \theta)}{\partial u \partial u'}\right]^{-1} + \frac{\partial u}{\partial \theta}\mathrm{cov}(\theta)\left(\frac{\partial u}{\partial \theta}\right)' \tag{3.1}$$

where u is the vector of random effect, θ is the vector of hyper-parameters, and $\partial u/\partial\theta$ is the sensitivity of the penalized likelihood estimator on the value of θ. The first term on the r.h.s. is the ordinary Fisher information based variance of u, while the second term accounts for the uncertainty in θ.

3.8 Building a random effects model that works

In all nonlinear parameter estimation problems, there are two possible explanations when your program does not produce meaningful results:

1. The underlying mathematical model is not well-defined, e.g., it may be over-parameterized.

2. You have implemented the model incorrectly, e.g., you have forgotten a minus sign somewhere.

In an early phase of the code development, it may not be clear which of these is causing the problem. With random effects, the two-step iteration scheme described above makes it even more difficult to find the error. We therefore advise you always to check the program on simulated data before you apply it to your real data set. This section gives you a recipe for how to do this.

The first thing you should do after having finished the TPL file is to check that the penalized likelihood step is working correctly. In ADMB, it is very easy to switch from a random-effects version of the program to a penalized-likelihood version. In `simple.tpl`, we would simply redefine the random effects vector `x` to be of type `init_vector`. The parameters would then be a, b, μ, σ, σ_x, and $x_1,\ldots,$ x_n. It is not recommended, or even possible, to estimate all of these simultaneously, so you should fix σ_x (by giving it a phase -1) at some reasonable value. The actual value at which you fix σ_x is not critically important, and you could even try a range of σ_x values. In larger models, there will be more than one parameter that needs to be fixed. We recommend the following scheme:

1. Write a simulation program (in R, S-Plus, Matlab, or some other program) that generates data from the random effects model (using some reasonable values for the parameters) and writes to `simple.dat`.

2. Fit the penalized likelihood program with σ_x (or the equivalent parameters) fixed at the value used to simulate data.

3. Compare the estimated parameters with the parameter values used to simulate data. In particular, you should plot the estimated $x_1,\ldots,x_n$ against the simulated random effects. The plotted points should center around a straight line. If they do (to some degree of approximation), you most likely have got a correct formulation of the penalized likelihood.

If your program passes this test, you are ready to test the random effects version of the program. You redefine `x` to be of type `random_effects_vector`, free up σ_x, and apply your program again to the same simulated data set. If the program produces meaningful estimates of the hyper-parameters, you most likely have implemented your model correctly, and you are ready to move on to your real data!

With random effects, it often happens that the maximum likelihood estimate of a variance component is zero ($\sigma_x = 0$). Parameters bouncing against the boundaries usually makes one feel uncomfortable, but with random effects, the interpretation of $\sigma_x = 0$ is clear and unproblematic. All it really means is that data do not support a random effect, and the natural consequence is to remove (or inactivate) $x_1,\ldots,x_n$, together with the corresponding prior (and hence σ_x), from the model.

3.9 MCMC

There are two different MCMC methods built into ADMB-RE: `-mcmc` and `-mcmc2`. Both are based on the Metropolis-Hastings algorithm. The former generates a Markov chain on the hyper-parameters only, while the latter generates a chain on the joint vector of hyper-parameters and random effects. (Some sort of rejection sampling could be used with `-mcmc` to generate values also for the random effects, but this is currently not implemented). The advantages of `-mcmc` are:

- Because there typically is a small number of hyper-parameters, but a large number of random effects, it is much easier to judge convergence of the chain generated by `-mcmc` than that generated by `-mcmc2`.
- The `-mcmc` chain mixes faster than the `-mcmc2` chain.

The disadvantage of the `-mcmc` option is that it is slow, because it relies on evaluation of the marginal likelihood by the Laplace approximation. It is recommended that you run both `-mcmc` and `-mcmc2` (separately), to verify that they yield the same posterior for the hyper-parameters.

3.10 Importance sampling

The Laplace approximation may be inaccurate in some situations. The accuracy may be improved by adding an importance sampling step. This is done in ADMB-RE by using the command line argument `-is N seed`, where `N` is the sample size in the importance sampling and `seed` (optional) is used to initialize the random number generator. Increasing `N` will give better accuracy, at the cost of a longer run time. As a rule-of-thumb, you should start with $\texttt{N} = 100$, and increase `N` stepwise by a factor of 2 until the parameter estimates stabilize.

By running the model with different seeds, you can check the Monte Carlo error in your estimates, and possibly average across the different runs to decrease the Monte Carlo error. Replacing the `-is N seed` option with an `-isb N seed` one gives you a "balanced" sample, which in general, should reduce the Monte Carlo error.

For large values of `N`, the option `-is N seed` will require a lot of memory, and you will see that huge temporary files are produced during the execution of the program. The option `-isf 5` will split the calculations relating to importance sampling into 5 (you can replace the 5 with any number you like) batches. In combination with the techniques discussed in Section 3.12.1, this should reduce the storage requirements. An example of a command line is:

```
lessafre -isb 1000 9811 -isf 20 -cbs 50000000 -gbs 50000000
```

The `-is` option can also be used as a diagnostic tool for checking the accuracy of the Laplace approximation. If you add the `-isdiag` (print importance sampling), the importance sampling weights will be printed at the end of the optimization process. If these weights do

not vary much, the Laplace approximation is probably doing well. On the other hand, if a single weight dominates the others by several orders of magnitude, you are in trouble, and it is likely that even `-is N` with a large value of `N` is not going to help you out. In such situations, reformulating the model, with the aim of making the log-likelihood closer to a quadratic function in the random effects, is the way to go. See also the following section.

3.11 REML (Restricted maximum likelihood)

It is well known that maximum likelihood estimators of variance parameters can be downwards biased. The biases arise from estimation of one or more mean-related parameters. The simplest example of a REML estimator is the ordinary sample variance

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2,$$

where the divisor $(n-1)$, rather than the n that occurs for the maximum likelihood estimator, accounts for the fact that we have estimated a single mean parameter.

There are many ways of deriving the REML correction, but in the current context, the most natural explanation is that we integrate the likelihood function (*note:* not the log-likelihood) with respect to the mean parameters—β, say. This is achieved in ADMB-RE by defining β as being of type `random_effects_vector`, but without specifying a distribution/prior for the parameters. It should be noted that the only thing that the `random_effects_vector` statement tells ADMB-RE is that the likelihood function should be integrated with respect to β. In linear-Gaussian models, the Laplace approximation is exact, and hence this approach yields exact REML estimates. In nonlinear models, the notion of REML is more difficult, but REML-like corrections are still being used. For linear-Gaussian models, the REML likelihood is available in closed form. Also, many linear models can be fitted with standard software packages. It is typically much simpler to formulate a hierarchical model with explicit latent variables. As mentioned, the Laplace approximation is exact for Gaussian models, so it does not matter what way you do it.

An example of such a model is found here. To make the executable program run efficiently, the command line options `-nr 1 -sparse` should be used for linear models. Also, note that REML estimates can be obtained, as explained in Section 3.11.

3.12 Improving performance

In this section, we discuss certain mechanisms you can use to make an ADMB-RE program run faster and more smoothly.

3.12.1 Reducing the size of temporary files

When ADMB needs more temporary storage than is available in the allocated memory buffers, it starts producing temporary files. Since writing to disk is much slower than accessing memory, it is important to reduce the size of temporary files as much as possible. There are several parameters (such as `arrmblsize`) built into ADMB that regulate how large of memory buffers an ADMB program allocates at startup. With random effects, the memory requirements increase dramatically, and ADMB-RE deals with this by producing (when needed) six temporary files. (See Table 3.2.): The table also shows the command line arguments you can use to

File name	Command line option
`f1b2list1`	`-l1 N`
`f1b2list12`	`-l2 N`
`f1b2list13`	`-l3 N`
`nf1b2list1`	`-nl1 N`
`nf1b2list12`	`-nl2 N`
`nf1b2list13`	`-nl3 N`

Table 3.2: Temporary file command line options.

manually set the size (determined by `N`) of the different memory buffers.

When you see any of these files starting to grow, you should kill your application and restart it with the appropriate command line options. In addition to the options shown above, there is `-ndb N`, which splits the computations into N chunks. This effectively reduces the memory requirements by a factor of N—at the cost of a somewhat longer run time. N must be a divisor of the total number of random effects in the model, so that it is possible to split the job into N equally large parts. The `-ndb` option can be used in combination with the `-l` and `-nl` options listed above. The following rule-of-thumb for setting N in `-ndb N` can be used: if there are a total of m random effects in the model, one should choose N such that $m/N \approx 50$. For most of the models in the example collection (Chapter 3),this choice of N prevents any temporary files being created.

Consider this model as an example. It contains only about 60 random effects, but does rather heavy computations with these. As a consequence, large temporary files are generated. The following command line

```
$ ./union -l1 10000000 -l2 100000000 -l3 10000000 -nl1 10000000
```

takes away the temporary files, but requires 80Mb of memory. The command line

```
$ ./union -est -ndb 5 -l1 10000000
```

also runs without temporary files, requires only 20Mb of memory, but runs three times slower.

Finally, a warning about the use of these command line options. If you allocate too much memory, your application will crash, and you will (should) get a meaningful error message. You should monitor the memory use of your application (using "Task Manager" under Windows, and the command `top` under Linux) to ensure that you do not exceed the available memory on your computer.

3.12.2 Exploiting special model structure

If your model has special structure, such as grouped or nested random effects, state-space structure, crossed random effects or a general Markov strucure you will benefit greatly from using the techniques described in Section 4 below. In this case the memory options in Table 3.2 are less relevant (although sometimes useful), and instead the memory use can be controlled with the classical AD Model Builder command line options `-cbs`, `-gbs` etc.

3.12.3 Limited memory Newton optimization

The penalized likelihood step (Section 3.7), which forms a crucial part of the algorithm used by ADMB to estimate hyper-parameters, is by default conducted using a quasi-Newton optimization algorithm. If the number of random effects is large—as it typically is for separable models—it may be more efficient to use a "limited memory quasi-Newton" optimization algorithm. This is done using the command line argument `-ilmn N`, where `N` is the number of steps to keep. Typically, $\texttt{N} = 5$ is a good choice.

Chapter 4

Exploiting special structure (Separability)

A model is said to be "separable" if the likelihood can be written as a product of terms, each involving only a small number of random effects. Not all models are separable, and for small toy examples (less than 50 random effects, say), we do not need to care about separability. You need to care about separability both to reduce memory requirements and computation time. Examples of separable models are

- Grouped or nested random effects
- State-space models
- Crossed random effects
- Latent Markov random fields

The presence of separability allows ADMB to calculate the "Hessian" matrix very efficiently.The Hessian H is defined as the (negative) Fisher information matrix (inverse covariance matrix) of the posterior distribution of the random effects , and is a key component of the Laplace approximation.

How do we inform ADMB-RE that the model is separable? We define `SEPARABLE_FUNCTION`s in the `PROCEDURE_SECTION` to specify the individual terms in the product that defines the likelihood function. Typically, a `SEPARABLE_FUNCTION` is invoked many times, with a small subset of the random effects each time.

For separable models the Hessian is a sparse matrix which means that it contains mostly zeros. Sparsity can be exploited by ADMB when manipulating the matrix H, such as calculating its determinant. The actual sparsity pattern depends on the model type:

- *Grouped or nested random effects:* H is block diagonal.
- *State-space models:* H is a banded matrix with a narrow band.

- *Crossed random effects:* unstructured sparsity pattern.
- *Latent Markov random fields:* often banded, but with a wide band.

For block diagonal and banded H, ADMB-RE automatically will detect the structure from the `SEPARABLE_FUNCTION` specification, and will print out a message such as:

```
Block diagonal Hessian (Block size = 3)
```

at the beginning of the phase when the random effects are becoming active parameters. For general sparsity pattern the command line option `-shess` can be used to invoke the sparse matrix libraries for manipulation of the matrix H.

4.1 The first example

A simple example is the one-way variance component model

$$y_{ij} = \mu + \sigma_u u_i + \varepsilon_{ij}, \qquad i = 1, \ldots, q, \quad j = 1, \ldots, n_i$$

where $u_i \sim N(0,1)$ is a random effect and $\varepsilon_{ij} \sim N(0, \sigma^2)$ is an error term. The straightforward (non-separable) implementation of this model (shown only in part) is

```
PARAMETER_SECTION
  random_effects_vector u(1,q)

PROCEDURE_SECTION
  for(i=1;i<=q;i++)
  {
    g -= -0.5*square(u(i));
    for(j=1;j<=n(i);j++)
      g -= -log(sigma) - 0.5*square((y(i,j)-mu-sigma_u*u(i))/sigma);
  }
```

The efficient (separable) implementation of this model is

```
PROCEDURE_SECTION
  for(i=1;i<=q;i++)
    g_cluster(i,u(i),mu,sigma,sigma_u);

SEPARABLE_FUNCTION void g_cluster(int i, const dvariable& u,...)
  g -= -0.5*square(u);
  for(int j=1;j<=n(i);j++)
    g -= -log(sigma) - 0.5*square((y(i,j)-mu-sigma_u*u)/sigma);
```

where (due to lack of space in this document) we've replaced the rest of the argument list with ellipsis (...).

It is the function call `g_cluster(i,u(i),mu,sigma,sigma_u)` that enables ADMB-RE to identify that the posterior distribution (??) factors (over i):

$$p(u \mid y) \propto \prod_{i=1}^{q} \left\{ \prod_{j=1}^{n_i} p\big(u_i \mid y_{ij}\big) \right\}$$

and hence that the Hessian is block diagonal (with block size 1). Knowing that the Hessian is block diagonal enables ADMB-RE to do a series of univariate Laplace approximations, rather than a single Laplace approximation in full dimension q. It should then be possible to fit models where q is in the order of thousands, but this clearly depends on the complexity of the function `g_cluster`.

The following rules apply:

- ★ The argument list in the definition of the `SEPARABLE_FUNCTION` *should not* be broken into several lines of text in the TPL file. This is often tempting, as the line typically gets long, but it results in an error message from `tpl2rem`.

- ★ Objects defined in the `PARAMETER_SECTION` *must* be passed as arguments to `g_cluster`. There is one exception: the objective function `g` is a global object, and does not need to be an argument. Temporary/programming variables should be defined locally within the `SEPARABLE_FUNCTION`.

- ★ Objects defined in the `DATA_SECTION` *should not* be passed as arguments to `g_cluster`, as they are also global objects.

The data types that currently can be passed as arguments to a `SEPARABLE_FUNCTION` are:

```
int
const dvariable&
const dvar_vector&
const dvar_matrix&
```

with an example being:

```
SEPARABLE_FUNCTION void f(int i, const dvariable& a, const dvar_vector& beta)
```

The qualifier `const` is required for the latter two data types, and signals to the C++ compiler that the value of the variable is not going to be changed by the function. You may also come across the type `const prevariable&`, which has the same meaning as `const dvariable&`.

There are other rules that have to be obeyed:

- ★ No calculations of the log-likelihood, except calling `SEPARABLE_FUNCTION`, are allowed in `PROCEDURE_SECTION`. Hence, the only allowed use of parameters defined in `PARAMETER_SECTION` is to pass them as arguments to `SEPARABLE_FUNCTION`s. However, evaluation of `sdreport` numbers during the `sd_phase`, as well as MCMC calculations, are allowed.

This rule implies that all the action has to take place inside the SEPARABLE_FUNCTIONs. To minimize the number of parameters that have be passed as arguments, the following programming practice is recommended when using SEPARABLE_FUNCTIONs:

★ The PARAMETER_SECTION should contain definitions only of the independent parameters (those variables whose type has a init_ prefix) and random effects, i.e., no temporary programming variables.

All temporary variables should be defined locally in the SEPARABLE_FUNCTION, as shown here:

```
SEPARABLE_FUNCTION void prior(const dvariable& log_s, const dvariable& u)
  dvariable sigma_u = exp(log_s);
  g -= -log_s - 0.5*square(u(i)/sigma_u);
```

See a full example here. The orange model has block size 1.

4.2 Nested or clustered random effects: block diagonal *H*

In the above model, there was no hierarchical structure among the latent random variables (the us). A more complicated example is provided by the following model:

$$y_{ijk} = \sigma_v v_i + \sigma_u u_{ij} + \varepsilon_{ijk}, \qquad i = 1, \ldots, q, \quad j = 1, \ldots, m, \quad k = 1, \ldots, n_{ij},$$

where the random effects v_i and u_{ij} are independent $N(0,1)$ distributed, and $\varepsilon_{ijk} \sim N(0, \sigma^2)$ is still the error term. One often says that the us are nested within the vs.

Another perspective is that the data can be split into independent clusters. For $i_1 \neq i_2$, y_{i_1jk} and y_{i_2jk} are statistically independent, so that the likelihood factors are at the outer nesting level (i).

To exploit this, we use the SEPARABLE_FUNCTION as follows:

```
PARAMETER_SECTION
  random_effects_vector v(1,q)
  random_effects_matrix u(1,q,1,m)

PROCEDURE_SECTION
  for(i=1;i<=q;i++)
    g_cluster(v(i),u(i),sigma,sigma_u,sigma_v,i);
```

Each element of v and each row (u(i)) of the matrix u are passed only once to the separable function g_cluster. This is the criterion ADMB uses to detect the block diagonal Hessian structure. Note that v(i) is passed as a single value while u(i) is passed as a vector to the SEPARABLE_FUNCTION as follows:

```
SEPARABLE_FUNCTION void g_cluster(const dvariable& v,const dvar_vector& u,...)
  g -= -0.5*square(v);
  g -= -0.5*norm2(u);

  for(int j=1;j<=m;j++)
    for(int k=1;k<=n(i,j);k++)
      g -= -log(sigma) - 0.5*square((y(i,j,k)
                 -sigma_v*v - sigma_u*u(j))/sigma);
```

★ For a model to be detected as "Block diagonal Hessian," each latent variable should be passed *exactly once* as an argument to a `SEPARABLE_FUNCTION`.

To ensure that you have not broken this rule, you should look for an message like this at run time:

```
Block diagonal Hessian (Block size = 3)
```

The "block size" is the number of random effects in each call to the `SEPARABLE_FUNCTION`, which in this case is one `v(i)` and a vector `u(i)` of length two. It is possible that the groups or clusters (as indexed by i, in this case) are of different size. Then, the "Block diagonal Hessian" that is printed is an average.

The program could have improperly been structured as follows:

```
PARAMETER_SECTION
  random_effects_vector v(1,q)
  random_effects_matrix u(1,q,1,m)

PROCEDURE_SECTION
  for(i=1;i<=q;i++)
    for(j=1;j<=m;j++)
      g_cluster(v(i),u(i,j),sigma,sigma_u,sigma_u,i);
```

but this would not be detected by ADMB-RE as a clustered model (because `v(i)` is passed multiple times), and hence ADMB-RE will not be able to take advantage of block diagonal Hessiand, as indicated by the absence of runtime message

```
Block diagonal Hessian (Block size = 3).
```

4.2.1 Gauss-Hermite quadrature

In the situation where the model is separable of type "Block diagonal Hessian," (see Section 4), Gauss-Hermite quadrature is available as an option to improve upon the Laplace approximation. It is invoked with the command line option `-gh N`, where `N` is the number of quadrature points determining the accuracy of the integral approximation. For a block size of 1, the default choice should be $N = 10$ or greater, but for larger block sizes the

computational and memory requirements very quickly limits the range of N. If N is chosen too large ADMB-RE will crash witout giving a meaningful error message. To avoid ADMB-RE creating large temporary files the command line options `-cbs` and `-gbs` can be used.

The `-gh N` option should be preferred over importance sampling (`-is`).

4.3 State-space models: banded $\boldsymbol{H}$

A simple state space model is

$$\begin{aligned} y_i &= u_i + \epsilon_i, \\ u_i &= \rho u_{i-1} + e_i, \end{aligned}$$

where $e_i \sim N(0, \sigma^2)$ is an innovation term. The log-likelihood contribution coming from the state vector $(u_1, \ldots, u_n)$ is

$$\sum_{i=2}^{n} \log\left(\frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(u_i - \rho u_{i-1})^2}{2\sigma^2}\right]\right),$$

where $(u_1, \ldots, u_n)$ is the state vector. To make ADMB-RE exploit this special structure, we write a `SEPARABLE_FUNCTION` named `g_conditional`, which implements the individual terms in the above sum. This function would then be invoked as follows

```
for(i=2;i<=n;i++)
  g_conditional(u(i),u(i-1),rho,sigma);
```

See a full example here.

Above, we have looked at a model with a univariate state vector. For multivariate state vectors, as in

$$\begin{aligned} y_i &= u_i + v_i + \epsilon_i, \\ u_i &= \rho_1 u_{i-1} + e_i, \\ v_i &= \rho_2 v_{i-1} + d_i, \end{aligned}$$

we would merge the u and v vectors into a single vector $(u_1, v_1, u_2, v_2, \ldots, u_n, v_n)$, and define

```
random_effects_vector u(1,m)
```

where $m = 2n$. The call to the `SEPARABLE_FUNCTION` would now look like

```
for(i=2;i<=n;i++)
 g_conditional(u(2*(i-2)+1),u(2*(i-2)+2),u(2*(i-2)+3),u(2*(i-2)+4),...);
```

where the ellipsis $(\ldots)$ denotes the arguments ρ_1, ρ_2, σ_e, and σ_d.

4.4 Crossed random effects: sparse $\boldsymbol{H}$

The simplest instance of a crossed random effects model is

$$y_k = \sigma_u u_{i(k)} + \sigma_v v_{j(k)} + \varepsilon_k, \qquad i = 1, \ldots n,$$

where $u_1, \ldots, u_N$ and $v_1, \ldots, v_M$ are random effects, and where $i(k) \in \{1, N\}$ and $j(k) \in \{1, M\}$ are index maps. The ys sharing either a u or a v will be dependent, and in general, no complete factoring of the likelihood will be possible. However, it is still important to exploit the fact that the us and vs only enter the likelihood through pairs $(u_{i(k)}, v_{j(k)})$. Here is the code for the crossed model:

```
for (k=1;k<=n;k++)
 log_lik(k,u(i(k)),v(j(k)),mu,s,s_u,s_v);

SEPARABLE_FUNCTION void log_lik(int k, const dvariable& u,...)
 g -= -log(s) - 0.5*square((y(k)-(mu + s_u*u + s_v*v))/s);
```

If only a small proportion of all the possible combinations of u_i and v_j actually occurs in the data, then the posterior covariance matrix of $(u_1, \ldots, u_N, v_1, \ldots, v_M)$ will be sparse. When an executable program produced by ADMB-RE is invoked with the `-shess` command line option, sparse matrix calculations are used.

This is useful not only for crossed models. Here are a few other applications:

- For the nested random effects model, as explained in Section 4.2.
- For REML estimation. Recall that REML estimates are obtained by making a fixed effect random, but with no prior distribution. For the nested models in Section 4.2, and the models with state-space structure of Section 4.3, when using REML, ADMB-RE will detect the cluster or time series structure of the likelihood. (This has to do with the implementation of ADMB-RE, not the model itself.) However, the posterior covariance will still be sparse, and the use of `-shess` is advantageous.

4.5 Gaussian priors and quadratic penalties

In most models, the prior for the random effect will be Gaussian. In some situations, such as in spatial statistics, all the individual components of the random effects vector will be jointly correlated. ADMB contains a special feature (the `normal_prior` keyword) for dealing efficiently with such models. The construct used to declaring a correlated Gaussian prior is

```
random_effects_vector u(1,n)
normal_prior S(u);
```

The first of these lines is an ordinary declaration of a random effects vector. The second line tells ADMB that `u` has a multivariate Gaussian distribution with zero expectation and

covariance matrix `S`, i.e., the probability density of $\mathbf{u}$ is

$$h(\mathbf{u}) = (2\pi)^{-\dim(S)/2} \det(S)^{-1/2} \exp\left(-\tfrac{1}{2}\mathbf{u}' S^{-1} u\right).$$

Here, S is allowed to depend on the hyper-parameters of the model. The part of the code where `S` gets assigned its value must be placed in a `SEPARABLE_FUNCTION`.

★ The log-prior $\log(h(\mathbf{u}))$ is automatically subtracted from the objective function. Therefore, the objective function must hold the negative log-likelihood when using the `normal_prior`.

See a full example here.

Appendix A

Example Collection

This section contains various examples of how to use ADMB-RE. Some of these has been referred to earlier in the manual. The examples are grouped according to their "Hessian type" (see Section 4). At the end of each example, you will find a Files section containing links to webpages, where both program code and data can be downloaded.

A.1 Non-separable models

This section contains models that do not use any of the separability stuff. Sections A.1.2 and A.1.3 illustrate how to use splines as non-parametric components. This is currenlty a very popular technique, and fits very nicely into the random effects framework [10]. All the models except the first are, in fact, separable, but for illustrative purposes (the code becomes easier to read), this has been ignored.

A.1.1 Mixed-logistic regression: a WinBUGS comparison

Mixed regression models will usually have a block diagonal Hessian due to grouping/clustering of the data. The present model was deliberately chosen not to be separable, in order to pose a computational challenge to both ADMB-RE and WinBUGS.

Model description

Let $\mathbf{y} = (y_1, \ldots, y_n)$ be a vector of dichotomous observations ($y_i \in \{0, 1\}$), and let $\mathbf{u} = (u_1, \ldots, u_q)$ be a vector of independent random effects, each with Gaussian distribution (expectation 0 and variance σ^2). Define the success probability $\pi_i = \Pr(y_i = 1)$. The following relationship between π_i and explanatory variables (contained in matrices $\mathbf{X}$ and $\mathbf{Z}$) is assumed:

$$\log\left(\frac{\pi_i}{1-\pi_i}\right) = \mathbf{X}_i\beta + \mathbf{Z}_i\mathbf{u},$$

where $\mathbf{X}_i$ and $\mathbf{Z}_i$ are the i^{th} rows of the known covariates matrices $\mathbf{X}$ $(n \times p)$ and $\mathbf{Z}$ $(n \times q)$, respectively, and β is a p-vector of regression parameters. Thus, the vector of fixed-effects is $\theta = (\beta, \log \sigma)$.

Results

The goal here is to compare computation times with WinBUGS on a simulated data set. For this purpose, we use $n = 200$, $p = 5$, $q = 30$, and values of the the hyper-parameters, as shown in the Table A.1. The matrices $\mathbf{X}$ and $\mathbf{Z}$ were generated randomly with each element uniformly distributed on $[-2, 2]$. As start values for both AD Model Builder and BUGS, we used $\beta_{\text{init},j} = -1$ and $\sigma_{\text{init}} = 4.5$. In BUGS, we used a uniform $[-10, 10]$ prior on β_j and a standard (in the BUGS literature) noninformative gamma prior on $\tau = \sigma^{-2}$. In AD Model Builder, the parameter bounds $\beta_j \in [-10, 10]$ and $\log \sigma \in [-5, 3]$ were used in the optimization process. On the simulated data set, AD Model Builder used 27 seconds to

	β_1	β_2	β_3	β_4	β_5	σ
True values	0.0000	0.0000	0.0000	0.0000	0.0000	0.1000
ADMB-RE	0.0300	-0.0700	0.0800	0.0800	-0.1100	0.1700
Std. dev.	0.1500	0.1500	0.1500	0.1400	0.1600	0.0500
WinBUGS	0.0390	-0.0787	0.0773	0.0840	-0.1041	0.1862

Table A.1: True values

converge to the optimum of likelihood surface. On the same data set, we first ran WinBUGS (Version 1.4) for 5,000 iterations. The recommended convergence diagnostic in WinBUGS is the Gelman-Rubin plot (see the help files available from the menus in WinBUGS), which require that two Markov chains are run in parallel. From the Gelman-Rubin plot, it was clear that convergence appeared after approximately 2,000 iterations. The time taken by WinBUGS to generate the first 2,000 was approximately 700 seconds.

See the files here.

A.1.2 Generalized additive models (GAMs)

Model description

A very useful generalization of the ordinary multiple regression

$$y_i = \mu + \beta_1 x_{1,i} + \cdots + \beta_p x_{p,i} + \varepsilon_i,$$

is the class of additive model

$$y_i = \mu + f_1(x_{1,i}) + \cdots + f_p(x_{p,i}) + \varepsilon_i. \tag{A.1}$$

Here, the f_j are "nonparametric" components that can be modelled by penalized splines. When this generalization is carried over to generalized linear models, and we arrive at the class of GAMs [6]. From a computational perspective, penalized splines are equivalent to random effects, and thus GAMs fall naturally into the domain of ADMB-RE.

For each component f_j in equation (A.1), we construct a design matrix $\mathbf{X}$ such that $f_j(x_{i,j}) = \mathbf{X}^{(i)}\mathbf{u}$, where $\mathbf{X}^{(i)}$ is the i^{th} row of $\mathbf{X}$ and $\mathbf{u}$ is a coefficient vector. We use the R-function `splineDesign` (from the `splines` library) to construct a design matrix $\mathbf{X}$. To avoid overfitting, we add a first-order difference penalty [3]

$$-\lambda^2 \sum_{k=2} (u_k - u_{k-1})^2 , \tag{A.2}$$

to the ordinary GLM log-likelihood, where λ is a smoothing parameter to be estimated. By viewing $\mathbf{u}$ as a random effects vector with the above Gaussian prior, and by taking λ as a hyper-parameter, it becomes clear that GAM's are naturally handled in ADMB-RE.

Implementation details

- A computationally more efficient implementation is obtained by moving λ from the penalty term to the design matrix, i.e., $f_j(x_{i,j}) = \lambda^{-1}\mathbf{X}^{(i)}\mathbf{u}$.

- Since equation (A.2) does not penalize the mean of $\mathbf{u}$, we impose the restriction that $\sum_{k=1} u_k = 0$ (see `union.tpl` for details). Without this restriction, the model would be over-parameterized, since we already have an overall mean μ in equation (A.1).

- To speed up computations, the parameter μ (and other regression parameters) should be given "phase 1" in ADMB, while the λs and the $\mathbf{u}$s should be given "phase 2."

The Wage-union data

The data, which are available from Statlib, contain information for each of 534 workers about whether they are members ($y_i = 1$) of a workers union or are not ($y_i = 0$). We study the probability of membership as a function of six covariates. Expressed in the notation used by the R (S-Plus) function `gam`, the model is:

```
union ~race + sex + south + s(wage) + s(age) + s(ed), family=binomial
```

Here, `s()` denotes a spline functions with 20 knots each. For `wage`, a cubic spline is used, while for `age` and `ed`, quadratic splines are used. The total number of random effects that arise from the three corresponding $\mathbf{u}$ vectors is 64. Figure A.1 shows the estimated nonparametric components of the model. The time taken to fit the model was 165 seconds.

Extensions

- The linear predictor may be a mix of ordinary regression terms ($f_j(x) = \beta_j x$) and nonparametric terms. ADMB-RE offers a unified approach to fitting such models, in which the smoothing parameters λ_j and the regression parameters β_j are estimated simultaneously.
- It is straightforward in ADMB-RE to add "ordinary" random effects to the model, for instance, to accommodate for correlation within groups of observations, as in [8].

See the files here.

A.1.3 Semi-parametric estimation of mean and variance

Model description

An assumption underlying the ordinary regression

$$y_i = a + bx_i + \varepsilon_i'$$

is that all observations have the same variance, i.e., $\text{Var}(\varepsilon_i') = \sigma^2$. This assumption does not always hold, e.g., for the data shown in the upper panel of Figure A.2. This example is taken from [10].

It is clear that the variance increases to the right (for large values of x). It is also clear that the mean of y is not a linear function of x. We thus fit the model

$$y_i = f(x_i) + \sigma(x_i)\varepsilon_i,$$

where $\varepsilon_i \sim N(0,1)$, and $f(x)$ and $\sigma(x)$ are modelled nonparametrically. We take f to be a penalized spline. To ensure that $\sigma(x) > 0$, we model $\log[\sigma(x)]$, rather than $\sigma(x)$, as a spline function. For f, we use a cubic spline (20 knots) with a second-order difference penalty

$$-\lambda^2 \sum_{k=3}^{20} (u_j - 2u_{j-1} + u_{j-2})^2,$$

while we take $\log[\sigma(x)]$ to be a linear spline (20 knots) with the first-order difference penalty (see equation (A.2)).

Implementation details

Details on how to implement spline components are given in Example A.1.2.

- Parameters associated with f should be given "phase 1" in ADMB, while those associated with σ should be given "phase 2." The reason is that in order to estimate the variation, one first needs to have fitted the mean part.

- In order to estimate the variation function, one first needs to have fitted the mean part. Parameters associated with f should thus be given "phase 1" in ADMB, while those associated with σ should be given "phase 2."

See the files here.

A.1.4 Weibull regression in survival analysis

Model description

A typical setting in survival analysis is that we observe the time point t at which the death of a patient occurs. Patients may leave the study (for some reason) before they die. In this case, the survival time is said to be "censored," and t refers to the time point when the patient left the study. The indicator variable δ is used to indicate whether t refers to the death of the patient ($\delta = 1$) or to a censoring event ($\delta = 0$). The key quantity in modelling the probability distribution of t is the hazard function $h(t)$, which measures the instantaneous death rate at time t. We also define the cumulative hazard function $\Lambda(t) = \int_0^t h(s)\,\mathrm{ds}$, implicitly assuming that the study started at time $t = 0$. The log-likelihood contribution from our patient is $\delta \log(h(t)) - H(t)$. A commonly used model for $h(t)$ is Cox's proportional hazard model, in which the hazard rate for the i^{th} patient is assumed to be on the form

$$h_i t = h_0(t) \exp(\eta_i), \qquad i = 1, \ldots n.$$

Here, $h_0(t)$ is the "baseline" hazard function (common to all patients) and $\eta_i = \mathbf{X}_i\beta$, where $\mathbf{X}_i$ is a covariate vector specific to the i^{th} patient and β is a vector of regression parameters. In this example, we shall assume that the baseline hazard belongs to the Weibull family: $h_0(t) = rt^{r-1}$ for $r > 0$.

In the collection of examples following the distribution of WinBUGS, this model is used to analyse a data set on times to kidney infection for a set of $n = 38$ patients (see *Kidney: Weibull regression with random effects* in the Examples list at The Bugs Project). The data set contains two observations per patient (the time to first and second recurrence of infection). In addition, there are three covariates: *age* (continuous), *sex* (dichotomous), and *type of disease* (categorical, four levels). There is also an individual-specific random effect $u_i \sim N(0, \sigma^2)$. Thus, the linear predictor becomes

$$\eta_i = \beta_0 + \beta_{\text{sex}} \cdot \text{sex}_i + \beta_{\text{age}} \cdot \text{age}_i + \beta_{\text{D}}\,\mathbf{x}_i + u_i,$$

where $\beta_{\text{D}} = (\beta_1, \beta_2, \beta_3)$ and $\mathbf{x}_i$ is a dummy vector coding for the disease type. Parameter estimates are shown in Table A.2. See the files here.

A.2 Block-diagonal Hessian

This section contains models with grouped or nested random effects.

	β_0	β_{age}	β_1	β_2	β_3	β_{sex}	r	σ
ADMB-RE	-4.3440	0.0030	0.1208	0.6058	-1.1423	-1.8767	1.1624	0.5617
Std. dev.	0.8720	0.0137	0.5008	0.5011	0.7729	0.4754	0.1626	0.2970
BUGS	-4.6000	0.0030	0.1329	0.6444	-1.1680	-1.9380	1.2150	0.6374
Std. dev.	0.8962	0.0148	0.5393	0.5301	0.8335	0.4854	0.1623	0.3570

Table A.2: Parameter estimates for Weibull regression with random effects.

A.2.1 Nonlinear mixed models: an NLME comparison

Model description

The orange tree growth data was used by [9, Ch.8.2] to illustrate how a logistic growth curve model with random effects can be fit with the S-Plus function `nlme`. The data contain measurements made at seven occasions for each of five orange trees. See Table A.3. The

t_{ij}	Time point when the j^{th} measurement was made on tree i.
y_{ij}	Trunk circumference of tree i when measured at time point t_{ij}.

Table A.3: Orange tree data.

following logistic model is used:

$$y_{ij} = \frac{\phi_1 + u_i}{1 + \exp\left[-\left(t_{ij} - \phi_2\right)/\phi_3\right]} + \varepsilon_{ij},$$

where (ϕ_1, ϕ_2, ϕ_3) are hyper-parameters, and $u_i \sim N(0, \sigma_u^2)$ is a random effect, and $\varepsilon_{ij} \sim N(0, \sigma^2)$ is the residual noise term.

Results

Parameter estimates are shown in Table A.4. The difference between the estimates ob-

	ϕ_1	ϕ_2	ϕ_3	σ	σ_u
ADMB-RE	192.1	727.9	348.1	7.843	31.65
Std. dev.	15.658	35.249	27.08	1.013	10.26
`nlme`	191.0	722.6	344.2	7.846	31.48

Table A.4: Parameter estimates.

tained with ADMB-RE and `nlme` is small. The difference is caused by the fact that the two approaches use different approximations to the likelihood function. (ADMB-RE uses the Laplace approximation, and for `nlme`, the reader is referred to [9, Ch. 7].)

The computation time for ADMB was 0.58 seconds, while the computation time for `nlme` (running under S-Plus 6.1) was 1.6 seconds.

See the files here.

A.2.2 Pharmacokinetics: an NLME comparison

Model description

The "one-compartment open model" is commonly used in pharmacokinetics. It can be described as follows. A patient receives a dose D of some substance at time t_d. The concentration c_t at a later time point t is governed by the equation

$$c_t = \frac{D}{V} \exp\left[-\frac{Cl}{V}(t - t_d)\right]$$

where V and Cl are parameters (the so-called "Volume of concentration" and the "Clearance"). Doses given at different time points contribute additively to c_t. [9, Ch. 6.4] fitted this model to a data set using the S-Plus routine `nlme`. The linear predictor used by [9, p. 300] is:

$$\log(V) = \beta_1 + \beta_2 Wt + u_V,$$
$$\log(Cl) = \beta_3 + \beta_4 Wt + u_{Cl},$$

where Wt is a continuous covariate, while $u_V \sim N(0, \sigma_V^2)$ and $u_{Cl} \sim N(0, \sigma_{Cl}^2)$ are random effects. The model specification is completed by the requirement that the observed concentration y in the patient is related to the true concentration by $y = c_t + \varepsilon$, where $\varepsilon \sim N(0, \sigma^2)$ is a measurement error term.

Results

Estimates of hyper-parameters are shown in Table A.5. The differences between the estimates

	β_1	β_2	β_3	β_4	σ	σ_V	σ_{Cl}
ADMB-RE	-5.99	0.622	-0.471	0.532	2.72	0.171	0.227
Std. Dev	0.13	0.076	0.067	0.040	0.23	0.024	0.054
`nlme`	-5.96	0.620	-0.485	0.532	2.73	0.173	0.216

Table A.5: Hyper-parameter estimates: pharmacokinetics.

obtained with ADMB-RE and `nlme` are caused by the fact that the two methods use different

approximations of the likelihood function. ADMB-RE uses the Laplace approximation, while the method used by `nlme` is described in [9, Ch. 7].

The time taken to fit the model by ADMB-RE was 17 seconds, while the computation time for `nlme` (under S-Plus 6.1) was 7 seconds.

See the files here.

A.2.3 Frequency weighting in ADMB-RE

Model description

Let X_i be binomially distributed with parameters $N = 2$ and p_i, and further assume that

$$p_i = \frac{\exp(\mu + u_i)}{1 + \exp(\mu + u_i)}, \tag{A.3}$$

where μ is a parameter and $u_i \sim N(0, \sigma^2)$ is a random effect. Assuming independence, the log-likelihood function for the parameter $\theta = (\mu, \sigma)$ can be written as

$$l(\theta) = \sum_{i=1}^{n} \log\big[p(x_i; \theta)\big]. \tag{A.4}$$

In ADMB-RE, $p(x_i; \theta)$ is approximated using the Laplace approximation. However, since x_i only can take the values 0, 1, and 2, we can rewrite the log-likelihood as

$$l(\theta) = \sum_{j=0}^{2} n_j \log\big[p(j; \theta)\big], \tag{A.5}$$

where n_j is the number x_i being equal to j. Still, the Laplace approximation must be used to approximate $p(j; \theta)$, but now only for $j = 0, 1, 2$, as opposed to $j = 1, \ldots n$, as above. For large n, this can give large savings.

To implement the log-likelihood (A.5) in ADMB-RE, you must organize your code into a `SEPARABLE_FUNCTION` (see the section "Nested models" in the ADMB-RE manual). Then you should do the following:

- Formulate the objective function in the weighted form (A.5).
- Include the statement

  ```
  !! set_multinomial_weights(w);
  ```

  in the `PARAMETER_SECTION`. The variable `w` is a vector (with indexes starting at 1) containing the weights, so in our case, $w = (n_0, n_1, n_2)$.

See the files here.

A.2.4 Ordinal-logistic regression

Model description

In this model, the response variable y takes on values from the ordered set $\{y^{(s)}, s = 1, \ldots, S-1\}$, where $y^{(1)} < y^{(2)} < \cdots < y^{(S)}$. For $s = 1, \ldots, S-1$, define $P_s = P(y \leq y^{(s)})$ and $\kappa_s = \log[P_s/(1-P_s)]$. To allow κ_s to depend on covariates specific to the i^{th} observation $(i = 1, \ldots, n)$, we introduce a disturbance η_i of κ_s:

$$P(y_i \leq y^{(s)}) = \frac{\exp(\kappa_s - \eta_i)}{1 + \exp(\kappa_s - \eta_i)}, \qquad s = 1, \ldots, S-1.$$

with

$$\eta_i = \mathbf{X}_i \beta + u_{j_i},$$

where $\mathbf{X}_i$ and β play the sample role, as in earlier examples. The u_j $(j = 1, \ldots, q)$ are independent $N(0, \sigma^2)$ variables, and j_i is the latent variable class of individual i.

See the files here.

A.3 Banded Hessian (state-space)

Here are some examples of state-space models.

A.3.1 Stochastic volatility models in finance

Model description

Stochastic volatility models are used in mathematical finance to describe the evolution of asset returns, which typically exhibit changing variances over time. As an illustration, we use a time series of daily pound/dollar exchange rates $\{z_t\}$ from the period 01/10/81 to 28/6/85, previously analyzed by [5]. The series of interest are the daily mean-corrected returns $\{y_t\}$, given by the transformation

$$y_t = \log z_t - \log z_{t-1} - n^{-1} \sum_{i=1}^{n} (\log z_t - \log z_{t-1}).$$

The stochastic volatility model allows the variance of y_t to vary smoothly with time. This is achieved by assuming that $yt \sim N(\mu, \sigma_t^2)$, where $\sigma_t^2 = \exp(\mu_x + x_t)$. The smoothly varying component x_t follows the autoregression

$$x_t = \beta x_{t-1} + \varepsilon_t, \qquad \varepsilon_t \sim N(0, \sigma^2).$$

The vector of hyper-parameters is for this model is thus $(\beta, \sigma, \mu, \mu_x)$.

See the files here.

A.3.2 A discrete valued time series: the polio data set

Model description

[12] analyzed a time series of monthly numbers of poliomyelitis cases during the period 1970–1983 in the U.S. We make a comparison to the performance of the Monte Carlo Newton-Raphson method, as reported in [7]. We adopt their model formulation.

Let y_i denote the number of polio cases in the i^{th} period ($i = 1, \ldots, 168$). It is assumed that the distribution of y_i is governed by a latent stationary AR(1) process $\{u_i\}$ satisfying

$$u_i = \rho u_{i-1} + \varepsilon_i,$$

where the $\varepsilon_i \sim N(0, \sigma^2)$. To account for trend and seasonality, the following covariate vector is introduced:

$$\mathbf{x}_i = \left(1, \frac{i}{1000}, \cos\left(\frac{2\pi}{12}i\right), \sin\left(\frac{2\pi}{12}i\right), \cos\left(\frac{2\pi}{6}i\right), \sin\left(\frac{2\pi}{6}i\right)\right).$$

Conditionally on the latent process $\{u_i\}$, the counts y_i are independently Poisson distributed with intensity

$$\lambda_i = \exp(\mathbf{x}_i{}'\beta + u_i).$$

Results

Estimates of hyper-parameters are shown in Table A.6.

	β_1	β_2	β_3	β_4	β_5	β_6	ρ	σ
ADMB-RE	0.242	-3.81	0.162	-0.482	0.413	-0.0109	0.627	0.538
Std. dev.	0.270	2.76	0.150	0.160	0.130	0.1300	0.190	0.150
[7]	0.244	-3.82	0.162	-0.478	0.413	-0.0109	0.665	0.519

Table A.6: Hyper-parameter estimates: polio data set.

We note that the standard deviation is large for several regression parameters. The ADMB-RE estimates (which are based on the Laplace approximation) are very similar to the exact maximum likelihood estimates, as obtained with the method of [7].

See the files here.

A.4 Generally sparse Hessian

A.4.1 Multilevel Rasch model

The multilevel Rasch model can be implemented using random effects in ADMB. As an example, we use data on the responses of 2042 soldiers to a total of 19 items (questions),

taken from [2] This illustrates the use of crossed random effects in ADMB. Furthermore, it is shown how the model easily can be generalized in ADMB. These more general models cannot be fitted with standard GLMM software, such as "lmer" in R.

See the files here.

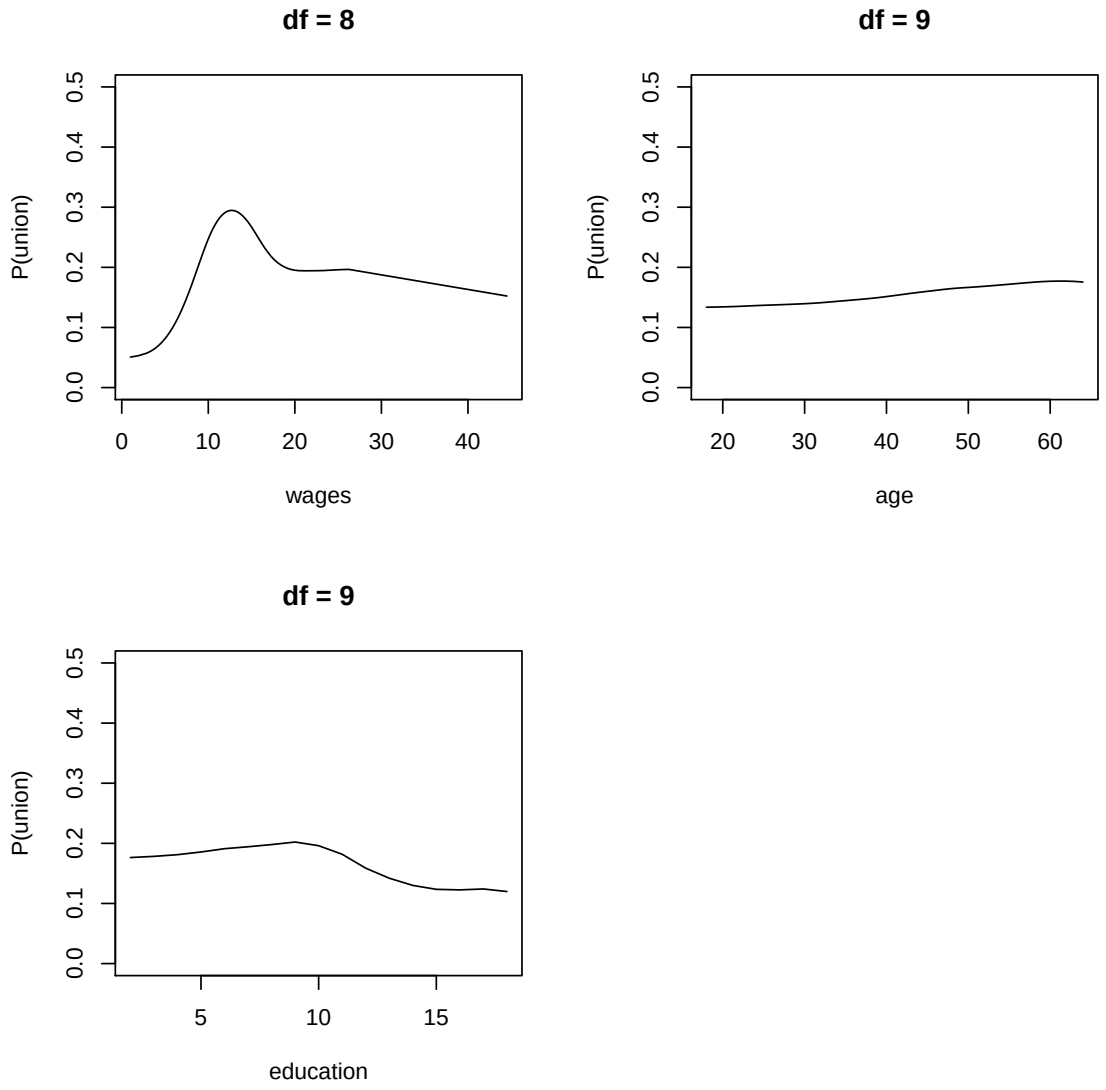


Figure A.1: Probability of membership as a function of covariates. In each plot, the remaining covariates are fixed at their sample means. The effective degrees of freedom (df) are also given [6].

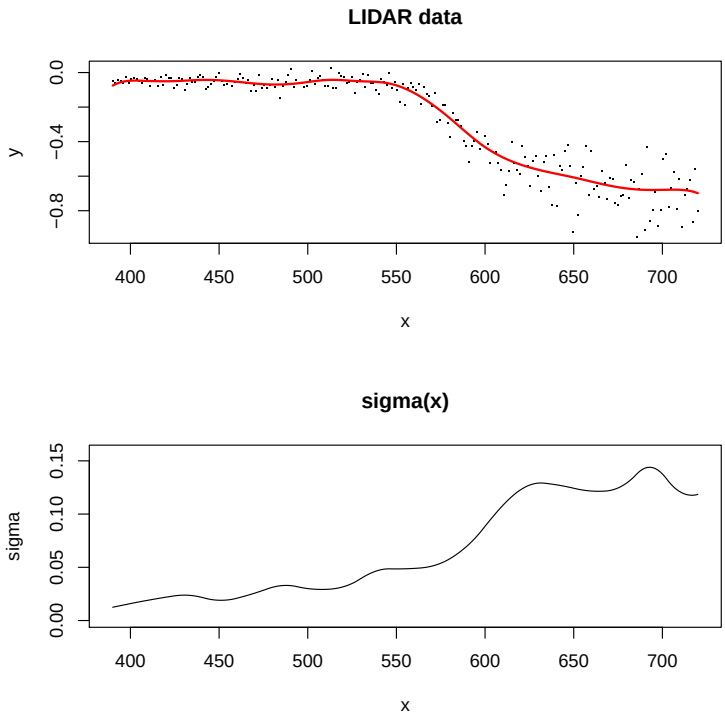


Figure A.2: LIDAR data (upper panel) used by [10], with fitted mean. Fitted standard deviation is shown in the lower panel.

Appendix B

Differences between ADMB and ADMB-RE?

- Profile likelihoods are now implemented also in random effects models, but with the limitation that the `likeprof_number` can only depend on parameters, not random effects.
- Certain functions, especially for matrix operations, have not been implemented.
- The assignment operator for `dvariable` behaves differently. The code

  ```
  dvariable y = 1;
  dvariable x = y;
  ```

  will make `x` and `y` point to the same memory location (shallow copy) in ADMB-RE. Hence, changing the value of `x` automatically changes `y`. Under ADMB, on the other hand, `x` and `y` will refer to different memory locations (deep copy). If you want to perform a deep copy in ADMB-RE you should write:

  ```
  dvariable y = 1;
  dvariable x;
  x = y;
  ```

  For vector and matrix objects ADMB and ADMB-RE behave identically in that a shallow copy is used.

Appendix C

Command Line options

A list of command line options accepted by ADMB programs can be obtained using the command line option `-?`, for instance,

```
$ simple -?
```

Those options that are specific to ADMB-RE are printed after line the "Random effects options if applicable." See Table C.1. The options in the last section (the sections are separated by horizontal bars) are not printed, but can still be used (see earlier).

Option	Explanation
`-nr N`	maximum number of Newton-Raphson steps
`-imaxfn N`	maximum number of evals in quasi-Newton inner optimization
`-is N`	set importance sampling size to `N` for random effects
`-isf N`	set importance sampling size funnel blocks to `N` for random effects
`-isdiag`	print importance sampling diagnostics
`-hybrid`	do hybrid Monte Carlo version of MCMC
`-hbf`	set the hybrid bounded flag for bounded parameters
`-hyeps`	mean step size for hybrid Monte Carlo
`-hynstep`	number of steps for hybrid Monte Carlo
`-noinit`	do not initialize random effects before inner optimzation
`-ndi N`	set maximum number of separable calls
`-ndb N`	set number of blocks for derivatives for random effects (reduces temporary file sizes)
`-ddnr`	use high-precision Newton-Raphson for inner optimization for banded Hessian case *only*, even if implemented
`-nrdbg`	verbose reporting for debugging Newton-Raphson
`-mm N`	do minimax optimization
`-shess`	use sparse Hessian structure inner optimzation
`-l1 N`	set the size of buffer `f1b2list1` to `N`
`-l2 N`	set the size of buffer `f1b2list12` to `N`
`-l3 N`	set the size of buffer `f1b2list13` to `N`
`-nl1 N`	set the size of buffer `nf1b2list1` to `N`
`-nl2 N`	set the size of buffer `nf1b2list12` to `N`
`-nl3 N`	set the size of buffer `nf1b2list13` to `N`

Table C.1: Command line options.

Appendix D

Quick References

D.1 Compiling ADMB programs

To compile `model.tpl` in a DOS/Linux terminal window, type

```
admb [-s] [-re] model
```

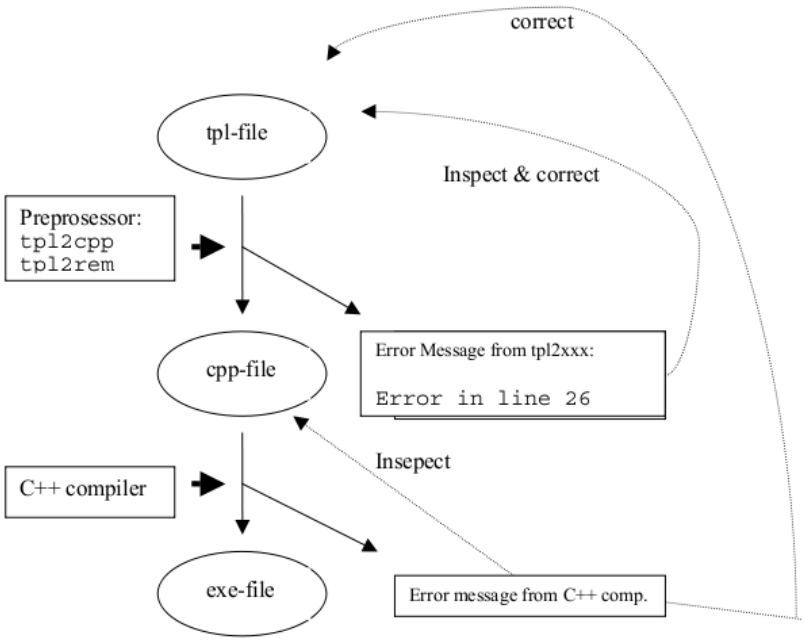
where the options

`-s` yields the "safe" version of the EXE file

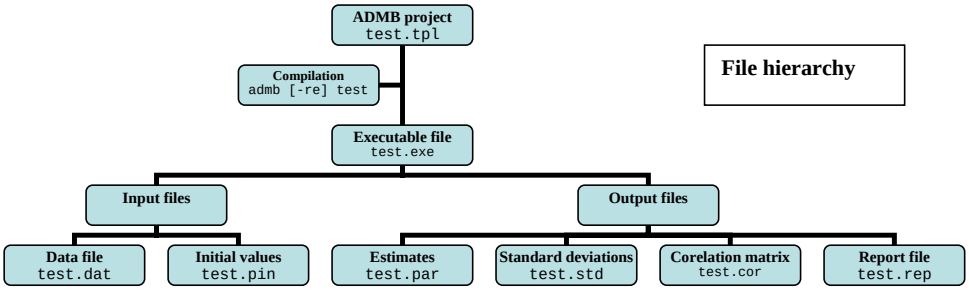
`-re` is used to invoke the random effect module

There are two stages of compilation:

- Proprocessor: `tpl2cpp` or `tpl2rem`
- C++ compiler (Borland, Visual C++, etc.)



The ADMB primer



TPL file

```
DATA_SECTION
  init_int n

  init_number a
  init_vector y(2,n)
  init_matrix z(1,n,2,10)

  int M
  vector x(1,n)

PARAMETER_SECTION
  init_number alpha
  init_bounded_number beta(0,a,2)

  init_bounded_vector mu(3,8,0,a,3)

  random_effects_vector u(1,n,2)

  objective_function_value g

PROCEDURE_SECTION
  g = 0.5*norm2(u);

  for(int i=2;i<=n;i++)
  {
    tmp = beta*(y(i)-u(i)-alpha);
    g += log(beta)
         + 0.5*square(tmp/beta);
```

Two blanks indentation of each line.

Declaration of data objects, and programming variables derived from these.

`init` here means "read from dat-file".

Valid index range: 2,…,n

Row range: 1,…,n
Col. range: 2,…, 10

`init` here means independent variable, i.e. parameter to be estimated. Starting value read from pin-file.

Declaration of independent parameters, random effects and variables derived from these.

Bounds: 0 < beta < a

Valid indices: 3,…,8
Bounds: 0 < mu(i) < a
Activated in phase 3

Parameter beta activated in phase 2,

Vector of random effects. Valid index range: 1,…,n. Activated in phase 2.

Assigns value to the log-likelihood function, in terms of data, parameters and random effects.
Pure C++ code.

Objective function / negative log-likelihood

ADMB function that calculates the squared norm of the vector u.

Contribution from N(0,1) variables placed on random effects

Temporary variable defined in `PARAMETER SECTION`

Likelihood contribution from y(i), corresponding to assumption that y(i) is normally distributed with mean u(i)-alpha and standard deviation beta.

C++ incremental operator

References

[1] ADMB Foundation. ADMB-IDE: Easy and efficient user interface. *ADMB Foundation Newsletter*, 1(3):1–2, July 2009. 2-3

[2] H. Doran, D. Bates, P. Bliese, and M. Dowling. Estimating the multilevel Rasch model: With the lme4 package. *Journal of Statistical Software*, 20(2):1–17, 2007. A-11

[3] P. Eilers and B. Marx. Flexible smoothing with B-splines and penalties. *Statistical Science*, 11:89–121, 1996. A-3

[4] D. Fournier. *An Introduction to AD Model Builder*. ADMB Foundation, Honolulu, 2011. 1-1, 3-7

[5] A.C. Harvey, E. Ruiz, and N. Shephard. Multivariate stochastic variance models. *Review of Economic Studies*, 61:247–264, 1994. A-9

[6] T.J. Hastie and R.J. Tibshirani. *Generalized Additive Models*, volume 43 of *Monographs on Statistics and Applied Probability*. Chapman & Hall, London, 1990. A-3, A-12

[7] A. Y. C. Kuk and Y. W. Cheng. Pointwise and functional approximations in Monte Carlo maximum likelihood estimation. *Statistics and Computing*, 9:91–99, 1999. A-10

[8] X. Lin and D. Zhang. Inference in generalized additive mixed models by using smoothing splines. *Journal of the Royal Statistical Society of Britain*, 61:381–400, 1999. A-4

[9] José C. Pinheiro and Douglas M. Bates. *Mixed-Effects Models in S and S-PLUS*. Statistics and Computing. Springer, 2000. 3-3, A-6, A-7, A-8

[10] David Ruppert, M.P. Wand, and R.J. Carroll. *Semiparametric Regression*. Cambridge University Press, 2003. A-1, A-4, A-13

[11] H. Skaug and D. Fournier. Automatic approximation of the marginal likelihood in non-Gaussian hierarchical models. *Computational Statistics & Data Analysis*, 56:699–709, 2006. 3-8

[12] S. L. Zeger. A regression-model for time-series of counts. *Biometrika*, 75:621–629, 1988. A-10

Index