Creating Custom Libraries for use in ADMB

By Steve Martell

Often I find that I'm repeatedly writing the same code in my ADMB template files for various projects; for example, the negative log-likelihood for the normal distribution. Wouldn't it be nice to just call a function such as `dnorm(x,mu,sd)` and have it return the negative log density. Many such functions have already been implemented in the ADMB library (e.g., `posfun`). The purpose of this article is to show you how to develop your own personal libraries that contain your own custom functions. Building custom libraries is a great way to organize your code, make the code more readable, and reduce the probability of making a coding error in future projects. Custom functions from these libraries can be included in your ADMB project using `#include <library_name>` statement in the `GLOBALS_SECTION` of your ADMB template. When you compile the ADMB project the functions written in the library are included in the project. There are two options for creating your own custom libraries for use in ADMB: 1) is to create a static-library application which involves the creation of a header and cpp files and compiling these into a binary library, or 2) create a new cpp file with various functions to include into your ADMB project. The second option is by far the simplest option as it does not require any header files and compiling static-libraries. The remainder of this article will focus on the much simpler second option.

To create your own custom library all you need to do is create a new file (e.g., "mylib.cpp" and be sure that this file is saved in the project root folder or the ADMB/lib folder), and at the top of that file be sure to include the admodel header file (`#inlcude <admodel.h>`). Following the include statements, write your own custom functions that return or modify variables. For example, I have a `stats.cxx` file saved in my ADMB Lib directory where I have compiled over the years many statistical functions (many of which are adopted from R) to carry out routine negative log-likelihood calculations for various statistical distributions. An example of the `stats.cxx` file containing two functions for computing the negative logdensity for a normal distribution is shown in the following code. Notice that you may overload functions, by declaring more than one function using the same name in the same scope. The declarations of the functions must differ from each other either the number of arguments and/or the type of arguments. When you call an overloaded function, the correct function is selected by comparing the argument list with the variable list.

```
#include <admodel.h>
dvariable dnorm(const double& x, const dvariable& mu, const dvariable& std)
{
       double pi=3.14159265358979323844;
       return(0.5*log(2.*pi)+log(std)+0.5*square(x-mu)/(std*std));
}
dvariable dnorm(const dvector& x, const dvar_vector& mu, const dvar_vector& std)
{
       double pi=3.14159265358979323844;
       int n=size_count(x);
       dvar_vector var=square(std);
       dvar_vector SS=square(x-mu);
       return(0.5*n*log(2.*pi)+sum(log(std))+0.5*sum(elem_div(SS,var)));
}
```

If you're not familiar with C++ code, you might find the above code a bit challenging at first. The above two functions return a single differentiable variables (`dvariable dnorm(...)`), require three arguments (`x,mu,std`) where the first argument is defined as a data-type variable/vector and the second and third arguments are differentiable variables (`dvariable` or `dvar_vector`). All three of these arguments are given "read-only" access to the variables via pointers (e.g., `const double& x`) which is faster than creating copy of the object (e.g., `double x`) for the argument list. The second function also declares two differentiable vectors on the fly (e.g., `dvar_vector`); the scope of these two vectors is only within the `dnorm` function. You can also write your own function within the ADMB template file, and copy the corresponding C++ code from the cpp file that is produced by `tpl2cpp.exe` and remove the class member name (`model_parameters::`) from the function. — Continued on page 4 —

Creating Custom Libraries for use in ADMB continued from page 3

Note also you may need to add the `RETURN_ARRAYS_INCREMENT()` and `RETURN_ARRAYS_DECREMENT()` statements if your function returns a variable object (see 3-7 in the Autodiff manual for more information). To use this function within my ADMB project I first must include the file in the `GLOBALS_SECTION` then call the `dnorm` function in the `PROCEDURE_SECTION`, e.g.,

```
GLOBALS_SECTION
  #include <stats.cxx> //include functions from my custom library
PROCEDURE_SECTION
  dvar_vector plen = linf*(1.-exp(-k*(age-to)));
  dvar_vector stdev = cv*plen;
  f = dnorm(len,plen,stdev); //neg loglike for normal density.
```

where x is a data vector, `mu` and `std` are differentiable vectors.
The following is a simple example of fitting a von Bertalanffy growth model to some length-age data that uses the custom library `stats.cxx` and calls the `dnorm` function.

```
DATA_SECTION
  init_int nobs;
  init_vector age(1,nobs);
  init_vector len(1,nobs);
PARAMETER_SECTION
  init_number linf;
  init_number k;
  init_number to;
  init_number cv;
  !! linf = 100.;
  !! k = 0.2;
  !! to = -0.5;
  !! cv= 0.1;
  objective_function_value f;
PROCEDURE_SECTION
  dvar_vector plen = linf*(1.-exp(-k*(age-to)));
  dvar_vector stdev = cv*plen;
  f = dnorm(len,plen,stdev); //neg loglike for normal density.
GLOBALS_SECTION
  #include <stats.cxx> //custom library that contains the dnorm function
———————— Example data ————————————————
# nobs
20
# age
1 2 2 3 3 3 3 3 4 6 5 5 7 4 4 6 7 7 7 9
# length
27 41 43 49 51 53 55 57 57 59 65 67 69 71 71 73 75 83 85 101
```