AD Model Builder introduction course

Data input and reporting results

AD Model Builder foundation

anders@nielsensweb.org

DATA_SECTION

- Is where data is read in
- If needed it can be processed a bit before entering the likelihood
- This section is only evaluated ones
- AD Model Builder will not keep track of the derivatives for the quantities declared here
- We have already seen a few examples

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

```
DATA_SECTION
  init_int N
  init_vector Y(1,N)
  init_vector x(1,N)
PARAMETER_SECTION
  init_number a
  init_number b
  init_number logSigma
  sdreport_number sigmasq
  objective_function_value nll
PROCEDURE_SECTION
  sigmasq=exp(2*logSigma);
  nll=0.5*(N*log(2*M_PI*sigmasq)+sum(square(Y-(a+b*x)))/sigmasq);
```

Basic rules

- Lines in the data file starting with a # are comments and ignored by AD Model Builder
- All types in the DATA_SECTION starting with `init_` are initialized from the data file
- E.g:

  init_int

  init_number

  init_ivector

  init_vector

  init_imatrix

  init_matrix

  init_3darray ... init_7darray

  init_adstring
- All types without `init_` can be used for further calculations, but are not initialized from the data file (unless it is done 'manually')

Dimensions set by data

- Notice that the length of the vectors N are also read in
- This is good when we want to make general programs, as no recompilation is needed to run the same program for a data set of different length

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

DATA_SECTION
init_int N
init_vector Y(1,N)
init_vector x(1,N)
```

Data matrix and local one line calculations

- A different way to input data for the same example

```
DATA_SECTION
  init_int N
  init_matrix xy(1,N,1,2)

  vector x(1,N)
  vector Y(1,N)

  !! x = column(xy,1);
  !! Y = column(xy,2);
```

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

- First a matrix is read from the data file
- Then the first column is saved as a vector ‘x’ and the second column as a vector ‘Y’
- The rest of the program is unchanged
- A common use for this feature is transformations like:

```
DATA_SECTION
  init_int N
  init_vector obs(1,N)

  vector logObs(1,N)

  !! logObs = log(obs);
```

LOC_CALCS and random numbers

- For longer calculations LOC_CALCS ... END_CALCS is more convenient than !!

```
DATA_SECTION
  vector X(1,1000);
 LOC_CALCS
  random_number_generator rng(123456);
  X.fill_randn(rng);
  X*=5.0;
  X+=2.0;
 END_CALCS
PARAMETER_SECTION
  init_number logSigma;
  init_number mu;
  sdreport_number sigma;
  objective_function_value nll;
PROCEDURE_SECTION
  sigma=exp(logSigma);
  int N=X.indexmax()-X.indexmin()+1;
  dvariable ss=square(sigma);
  nll=0.5*(N*log(2*M_PI*ss)+sum(square(X-mu))/ss);
```

- This program uses simulated data only, so the data file is not needed, except ...

More on generating random numbers

```
DATA_SECTION
 LOC_CALCS
  random_number_generator rng(123456);
  dvector sample(1,5);

  sample.fill_randu(rng);
  cout<<"Uniform(0,1): "<<sample<<endl;

  sample.fill_randn(rng);
  cout<<"Normal(0,1): "<<sample<<endl;

  sample.fill_randpoisson(1.5,rng);
  cout<<"pois(1.5): "<<sample<<endl;

  sample.fill_randnegbinomial(1.5,2.0,rng);
  cout<<"neg.bin(1.5,2): "<<sample<<endl;

  sample.fill_randcau(rng);
  cout<<"Cauchy: "<<sample<<endl;

  sample.fill_randbi(0.8,rng);
  cout<<"binomial(n=1,p=0.8): "<<sample<<endl;

  dvector p("{.01,.495,.495}");
  sample.fill_multinomial(rng,p);
  cout<<"multinomial(n=1,p=(.01,.495,.495)): "
      <<sample<<endl;

  ad_exit(0);
 END_CALCS
PARAMETER_SECTION
  objective_function_value nll;

PROCEDURE_SECTION
```

```
Uniform(0,1):  0.779837 0.229835 0.0126429
               0.714228 0.654815

Normal(0,1):  -0.325127 1.03682 0.567672
              -0.670345 2.89024

pois(1.5):  2 2 0 1 2

neg.bin(1.5,2):  0 3 1 0 4

Cauchy:  -0.188267 -1.30511 -9.11156
         29.8652 2.83259

binomial(n=1,p=0.8):  1 1 0 0 1

multinomial(n=1,p=(.01,.495,.495)):  3 2 3 2 3
```

Changing input file

- The default is to read from a file named `<modelname>.dat`, where <modelname> is the name of the *.tpl file

- If for some reason we feel like reading from another file we can run the program with the command line option

```
an@ch-pcb-an:~/$./model -ind newdatafile.dat
```

- Or within the DATA_SECTION use a command like

```
DATA_SECTION
  init_int nrowA
  init_int ncolA
  init_matrix A(1,nrowA,1,ncolA)
  !! ad_comm::change_datafile_name("newfile.dat");
  init_int nrowB
  init_int ncolB
  init_matrix B(1,nrowB,1,ncolB)
```

- Then A is read from the default file and B are read from 'newfile.dat'

Ragged arrays

- A special feature is to use integer vectors (ivector) as dimensions to get ‘ragged arrays’

```
DATA_SECTION
  init_int N
  init_ivector startYear(1,N)
  init_ivector endYear(1,N)
  init_matrix A(1,N,startYear,endYear)
```

```
2
1991 1995
2000 1999
23 5 54 12 45 8 23 45 76 32
45 34 32 54 34
```

- Notice that the two rows in ‘A’ does not have the same column index, and that is OK

Checking what is read in

- John once taught me a neat way to check the input, consider this:

```
GLOBALS_SECTION
  #include <fstream.h>
  ofstream clogf("program.log");
  #define TRACE(obj) clogf<<"line "<<__LINE__<<", file "<<__FILE__<<", "<<#obj" =\n " \
                                <<obj<<endl<<endl;

DATA_SECTION
  init_int N
  !! TRACE(N)
  init_ivector startYear(1,N)
  !! TRACE(startYear)
  init_ivector endYear(1,N)
  !! TRACE(endYear)
  init_matrix A(1,N,startYear,endYear)
  !! TRACE(A)
  !! ad_exit(0);
PARAMETER_SECTION
  objective_function_value nll;
PROCEDURE_SECTION
```

- Then the following is placed in the 'program.log', and we can easily check it

```
line 15, file ra.cpp, N =
 2

line 17, file ra.cpp, startYear =
  1991 1995

line 19, file ra.cpp, endYear =
  2000 1999

line 21, file ra.cpp, A =
  23 5 54 12 45 8 23 45 76 32
 45 34 32 54 34
```

Reading standard results into R

- The following R-function is useful for reading the standard ADMB output files

```
read.fit<-function(file){
  #
  # Function to read a basic AD Model Builder fit.
  #
  # Use for instance by:
  #
  #   simple.fit <- read.fit('c:/admb/examples/simple')
  #
  # Then the object 'simple.fit' is a list containing sub-objects
  # 'names', 'est', 'std', 'cor', and 'cov' for all model
  # parameters and sdreport quantities.
  #
  ret<-list()
  parfile<-as.numeric(scan(paste(file,'.par', sep=''),
                      what='', n=16, quiet=TRUE)[c(6,11,16)])
  ret$nopar<-as.integer(parfile[1])
  ret$nlogl<-parfile[2]
  ret$maxgrad<-parfile[3]
  file<-paste(file,'.cor', sep='')
  lin<-readLines(file)
  ret$npar<-length(lin)-2
  ret$logDetHess<-as.numeric(strsplit(lin[1], '=')[[1]][2])
  sublin<-lapply(strsplit(lin[1:ret$npar+2], ' '),function(x)x[x!=''])
  ret$names<-unlist(lapply(sublin,function(x)x[2]))
  ret$est<-as.numeric(unlist(lapply(sublin,function(x)x[3])))
  ret$std<-as.numeric(unlist(lapply(sublin,function(x)x[4])))
  ret$cor<-matrix(NA, ret$npar, ret$npar)
  corvec<-unlist(sapply(1:length(sublin), function(i)sublin[[i]][5:(4+i)]))
  ret$cor[upper.tri(ret$cor, diag=TRUE)]<-as.numeric(corvec)
  ret$cor[lower.tri(ret$cor)] <- t(ret$cor)[lower.tri(ret$cor)]
  ret$cov<-ret$cor*(ret$std%o%ret$std)
  return(ret)
}
```

```
> source("tools.R")
> fit <- read.fit("simplelm")
> fit
$nopar
[1] 3

$nlogl
[1] 17.6406

$maxgrad
[1] 1.0698e-06

$npar
[1] 4

$logDetHess
[1] 8.33061

$names
[1] "a"          "b"          "logSigma" "sigmasq"

$est
[1] 4.07820 1.90910 0.34513 1.99420

$std
[1] 0.70394 0.15547 0.22361 0.89184

$cor
       [,1]   [,2] [,3] [,4]
[1,]  1.000 -0.773    0    0
[2,] -0.773  1.000    0    0
[3,]  0.000  0.000    1    1
[4,]  0.000  0.000    1    1

$cov
            [,1]        [,2]       [,3]      [,4]
[1,]  0.49553152 -0.08459832 0.00000000 0.0000000
[2,] -0.08459832  0.02417092 0.00000000 0.0000000
[3,]  0.00000000  0.00000000 0.05000143 0.1994243
[4,]  0.00000000  0.00000000 0.19942434 0.7953786
```

The REPORT_SECTION

- The REPORT_SECTION is for user defined output.
- Any calculated quantity can be written, and formatted as desired
- For instance:

```
REPORT_SECTION
  report<<"Here comes the matrix X<<endl;
  report<<X<<endl;
  report<<"After that the vector y"<<endl;
  report<<y<<endl;
```

- The output is written to the file `<modelname>.rep`

Exercises

Exercise 1: Inputting and using prior information in a Beverton-Holt model

- The Beverton-Holt model can be written (slightly re-parametrized) as:

$$\log R = \log(a) + \log(\mathsf{ssb}) - \log(1 + \exp(\log(b))\mathsf{ssb})$$

- We want to estimate the model parameters $\log(a)$ and $\log(b)$ and have two sources.
- A data set of SSB and $\log(R)$

  http://www.nielsensweb.org/ADMB2/BHex/bh.dat

- Which can be modeled program like

  http://www.nielsensweb.org/ADMB2/BHex/bh.tpl

- The second source is the result of for a similar species in a similar area.

```
The logarithm of the determinant of the hessian = 12.869
index   name         value       std dev       1       2       3
    1   loga      1.8085e+00 1.2725e-01   1.0000
    2   logb     -1.2183e+01 3.2431e-01   0.9278  1.0000
    3   logSigma -1.1332e+00 1.0426e-01   0.0000  0.0000  1.0000
```

- Modify the program to use this prior information on $\log(a)$ and $\log(b)$

Solution:

```
GLOBALS_SECTION
  #include <fstream.h>
  ofstream clogf("program.log");
  #define TRACE(obj) clogf<<"line "<<__LINE__<<", file "<<__FILE__<<", "<<#obj" =\n " \
                         <<obj<<endl<<endl;

DATA_SECTION
  init_int nR
  init_int nC
  init_matrix obs(1,nR,1,nC)
  vector ssb(1,nR)
  !! ssb=column(obs,1);
  vector logR(1,nR)
  !! logR=column(obs,2);

  !! ad_comm::change_datafile_name("prior.cor");

  !! // Dirty trick to wind fast forward over the first two lines
  !! // Alternertively the two lines could have been commented out with #
  !! for(int i=1; i<=18; ++i){
  init_adstring tmp;
  !!}

  init_int idxA
  init_adstring nameA
  init_number estA
  init_number sdA
  init_number corrAA

  init_int idxB
  init_adstring nameB
  init_number estB
  init_number sdB
  init_number corrBA
  init_number corrBB

  vector meanAB(1,2)
  matrix covAB(1,2,1,2)
 LOC_CALCS
  meanAB(1)=estA;
  meanAB(2)=estB;
  covAB(1,1)=square(sdA);
  covAB(1,2)=sdA*sdB*corrBA;
  covAB(2,1)=covAB(1,2);
  covAB(2,2)=square(sdB);
```

```
    TRACE(meanAB);
    TRACE(covAB);
   END_CALCS

  PARAMETER_SECTION
    init_number loga;
    init_number logb;
    init_number logSigma;
    number sigmaSq;
    vector pred(1,nR);
    vector vecAB(1,2);

    objective_function_value nll;

  PROCEDURE_SECTION
    sigmaSq=exp(2.0*logSigma);
    pred=loga+log(ssb)-log(1+exp(logb)*ssb);
    nll=0.5*(nR*log(2*M_PI*sigmaSq)+sum(square(logR-pred))/sigmaSq);

    vecAB(1)=loga;
    vecAB(2)=logb;
    dvar_vector diff=vecAB-meanAB;
    nll+=0.5*(log(2.0*M_PI)*2.0+log(det(covAB))+diff*inv(covAB)*diff);
```