Adjoint code for the Barnov catch equation

Steven Martell

2013-01-10

Abstract

This is a simple example of writing adjoint code in AD Model Builder to numerically solve the Baranov catch equation. A simple simulation model is used to generate simulated catched and relative abundance data with observation error only. The following assessment model is then conditioned on the simulated catch data. Annual fishing mortality rates are based on numerically solving a transcendental catch equation. Adjoint code for the catch equation is then presented. The exact same number of function evaluations and parameter estimates are obtained with both the adjoint code and the non-adjoint code; however, the run time is roughly 30% faster using the adjoint code.

Introduction

The Baranov catch equation

$$C = \frac{NF\left(1 - e^{-M - F}\right)}{M + F} \tag{1}$$

represents the fraction of total mortality that is due to fishing mortality (F). This expression cannot be solved for F analytically. In order to determine what value of F satisfies the above expression for given values of catch (C), abundance (N) and the natural mortality rate (M), a numerical method is required. This is most easily done using Newton's root finding method, where:

$$F_{i+1} = F_i + \frac{(\hat{C} - C)}{\frac{\partial C}{\partial F}} \tag{2}$$

where \hat{C} is the observed catch and C is the predicted catch based on the Baranov catch equation, and the first derivative of the catch equation is given by:

$$\frac{\partial C}{\partial F} = \frac{N\left(1 - \mathrm{e}^{-M - F}\right)}{M + F} + \frac{NF\mathrm{e}^{-M - F}}{M + F} - \frac{NF\left(1 - \mathrm{e}^{-M - F}\right)}{\left(M + F\right)^{2}}.$$

Note that in (1), the abundance N, fishing mortality F, and natural mortality rates are differentiable variables. Therefore, in numerically solving (2), the derivative information for these variables is calculated automatically by ADMB for each iteration. However, the derivative information from the last iteration (the final solution) is all that is acutally required. Althought this example is fairly simple, the number of times this function must be evaluated can be 6-15 times that of a simple analytical problem and this adds considerably to the computational time and memory requirements. The amount of derivative information can be reduced considerably by writing the adjoint code for this function, and computing the derivative only after the numerical solution has been obtained.

The following ADMB FUNCTION can be used to numerically evaluate (2). The arguments of the function are the observed catch, natural mortality, and the abundance. The function returns a dvariable object that corresponds to the instantaneous fishing mortality rate that satisfies (1).

```
FUNCTION dvariable get_f1 (const double& ct, const prevariable& m, const prevariable& nt)
         //use newtons root finding method to calc f.
         dvariable ft;
         ft = (ct/nt);
         dvariable ctmp;
                            // MUST use dvaribles to get the correct derivative info.
         dvariable df_ctmp;
         for (int iter=1; iter \leq=7; iter++)
                 dvariable t1 = (\exp(-m-ft));
                 dvariable t2 = (m+ft);
                 ctmp = (nt * ft * (1. - t1) / t2);
                 df_{-}ctmp = (nt*(1.-t1)/t2 - nt*ft*(1.-t1)/(t2*t2) + nt*ft*t1/t2);
                 ft = ((ctmp-ct)/df_ctmp);
         }
         return (ft);
}
```

Adjoint code

In order to write the adjoint code, the analytical derivatives for each of the respective differentiable variables must be obtained. The partial derivative for the fishing mortality rate with respect to the abundance, obtained through impicit differentiation, is given by:

$$\frac{\partial F}{\partial N} = \frac{F(-M - F + e^{-M - F}M + e^{-M - F}F)}{N(M - e^{-M - F}M + Fe^{-M - F}M + e^{-M - F}F^{2})}.$$
(3)

Similarly, the partial derivative for fishing mortality with respect to natural mortaity is given by:

$$\frac{\partial F}{\partial M} = -\frac{F\left(e^{-M-F}M + e^{-M-F}F - 1 + e^{-M-F}\right)}{M - e^{-M-F}M + Fe^{-M-F}M + e^{-M-F}F^{2}}$$
(4)

For the purpose of this example the adjoint code is located in the GLOBALS_SECTION. The adjoint code for this problem differs from the original function above in that the numerical estimation of F is based on double precision numbers, with the exception of the return argument ft. The first half of the get_f function is the same as previously defined, where the solution to equation 1 is obtained using Newton's method. The subsequen lines save the values and positions of the variables that are required to calculate the derivative information. The second to last line of code then calls the $deriv_f$ function to calculate the adjoint code.

```
dvariable get_f(const double& ct, const prevariable& m, const prevariable& nt)
{
    //use newtons root finding method to calc f.
    dvariable ft;
    ft=value(ct/nt);
    double ctmp;
    double df_ctmp;
    for(int iter=1; iter <=17; iter++)
    {</pre>
```

```
double t1 = value(exp(-m-ft));
         double t2 = value(m+ft);
         ctmp=value(nt*ft*(1.-t1)/t2);
         df_{\text{ctmp}} = value(nt*(1.-t1)/t2 - nt*ft*(1.-t1)/(t2*t2) + nt*ft*t1/t2);
         ft = ((ctmp-ct)/df_ctmp);
         // \text{ cout} << \text{iter} << \text{"} \t" << \text{ctmp-ct} << \text{endl};
         if (fabs(ctmp-ct) \le 1.e-14) break;
    }
    // Push variables on stack
    save_identifier_string("place1");
    nt.save_prevariable_value();
    nt.save_prevariable_position();
    m. save_prevariable_value();
    m. save_prevariable_position();
    save_identifier_string("place2");
    ft.save_prevariable_value();
    ft.save_prevariable_position();
    //Call adjoint code to get derivative
    ADJOINT_CODE(deriv_f);
    return (ft);
}
```

The adjoint code is located in the function deriv_f. The first thing that is done in this function is to read the variables off the stack in reverse order. The functions verify_identifier_string are essentially bookmarks that ensure that the variables are popped in the reverse order that they were pused onto the stack. You should use these functions during development to ensure you don't screw up the order.

Once the variables have been popped, the derivative information is then calculated based on the current values. Lastly, the function $save_double_derivative$ is used to store the partial derivatives of F with respect to N and M in the appropriate positions.

```
void deriv_f(void)
{
    //POP variables off stack in reverse order
    prevariable_position ft_pos = restore_prevariable_position();
    double ad_ft=restore_prevariable_value();
    verify_identifier_string("place2");
    prevariable_position m_pos = restore_prevariable_position();
    double ad_m = restore_prevariable_value();
    prevariable_position nt_pos = restore_prevariable_position();
    double ad_nt = restore_prevariable_value();
    verify_identifier_string("place1");
    double dft=restore_prevariable_derivative(ft_pos);
    //cout << "dft \ t" << dft << endl;

// derivatives from implicit differentiation.
    double dfb;
    double t1 = (-ad_m - ad_ft);</pre>
```

ADMB Code

The full baranov.tpl file follows, and there is no input datafile required. The adjoint code is specified in the GLOBALS_SECTION.

```
// | An example of developing adjoint code for the Baranov catch equation.
     Author: Steven Martell
     email:
              stevem@iphc.int
     This is a simple simulation model that first generates catch given ft.
     Then attempts to estimate the model parameters by conditioning the model
     based on the observed catch.
//
//
     The model equations are as follows.
// |
     N_{-}\{t+1\} = N_{-}t * \exp(-Z_{-}t) + R_{-}\{t\}
// |
     R_{-}\{t\}
// |
              = a*N_{t-1}/(1+b*N_{t-1})
// |
     Z_{-}\{t\}
              = M + F_t
// |
             = N_t * F_t * (1 - \exp(-Z_t)) / Z_t
     C_{-}\{t\}
             = q N_t * exp(-0.5 Z_t + epsilon_t)
// |
     y_{t}
//
     F.t is solved for numerically using Newton-Rahpson, or using Pope's approx.
     Uncomment lines 79, 82, or 85 to test the different methods.
// | The ADJoint code is called from line 85.
DATA_SECTION
    int n;
    !! \quad n = 30;
    vector ct(1,n);
    vector yt(1,n);
PARAMETER SECTION
    init_number log_no(1);
    init\_bounded\_number h(0.2,1.0,1);
```

```
init_number log_m(1);
    init_number log_q(1);
     !! \log_{-no} = \log(100);
     !! h = 0.75;
     !! \log_{-m} = \log(0.20);
     !! \log_{q} = \log(0.10);
    objective_function_value f;
    number no;
    number m;
    number reck;
    number a;
    number b;
    number q;
    vector nt(1,n+1);
    vector rt(1,n);
    vector zt(1,n);
    vector ft(1,n);
    vector epsilon(1,n);
PRELIMINARY_CALCS_SECTION
    simulateData();
PROCEDURE_SECTION
    runModel();
    nf++;
FUNCTION \ runModel
    int i;
           = mfexp(log_no);
    no
           = mfexp(log_m);
    \mathbf{m}
           = mfexp(log_q);
    reck = 4.*h/(1.-h);
           = \operatorname{reck} *(1. - \operatorname{mfexp}(-m));
           = (\operatorname{reck} -1.)/\operatorname{no};
    nt(1) = no;
    {\rm rt}\,(1) \; = \; {\rm no}\!*\!(1.-{\rm mfexp}(-\!m)\,)\,;
    // ft = 0.15;
    for (i = 1; i \le n; i ++)
         // Popes approximation.
         // ft(i) = ct(i)/(nt(i)*exp(-0.5*m));
         // Numerical soln for baranov equation w/o the use of ADJoint code.
         // - takes roughly 11.3 millseconds on my computer.
         // ft(i) = get_f1(ct(i), m, nt(i));
         // Numerical soln for Baranov equation with the use of ADJoint code.
         // - takes roughly 8 milliseconds on my computer.
```

```
ft(i) = get_-f(ct(i), m, nt(i));
          zt(i)
                   = m+ft(i);
          rt(i) = a*nt(i)/(1.+b*nt(i));
          nt(i+1) = nt(i)*exp(-zt(i)) + rt(i);
     epsilon = yt(1,n) - elem_prod(q*nt(1,n), exp(-0.5*zt(1,n)));
     f = 0.5*(n-1)*log(norm2(epsilon));
FUNCTION dvariable get_f1(const double& ct, const prevariable& m, const prevariable& nt)
          //use newtons root finding method to calc f.
          dvariable ft;
          ft = (ct/nt);
          dvariable ctmp; // MUST use dvaribles to get the correct derivative info.
          dvariable df_ctmp;
          for (int iter=1; iter \leq=7; iter++)
               dvariable t1 = (\exp(-m-ft));
               dvariable t2 = (m+ft);
               ctmp = (nt * ft * (1. - t1) / t2);
               df_{ctmp} = (nt*(1.-t1)/t2 - nt*ft*(1.-t1)/(t2*t2) + nt*ft*t1/t2);
               ft = ((ctmp-ct)/df_ctmp);
          }
          return (ft);
     }
FUNCTION simulateData
     int i;
     random_number_generator rng(999);
            = mfexp(log_no);
            = mfexp(log_m);
    m
            = mfexp(log_q);
     reck = 4.*h/(1.-h);
            = \operatorname{reck} * (1. - \operatorname{mfexp}(-m));
           = (\operatorname{reck} - 1.) / \operatorname{no};
     nt(1) = no;
     rt(1) = no*(1.-mfexp(-m));
           = 0.15;
     for (i = 1; i \le n; i \leftrightarrow)
     {
          zt(i) = m+ft(i);
          rt(i) = a*nt(i)/(1.+b*nt(i));
          nt(i+1) = nt(i)*exp(-zt(i)) + rt(i);
     \operatorname{ct}(1,n) = \operatorname{value}(\operatorname{elem\_div}(\operatorname{elem\_prod}(\operatorname{nt}(1,n),\operatorname{elem\_prod}(\operatorname{ft},(1.-\operatorname{exp}(-\operatorname{zt})))),\operatorname{zt}));
     yt(1,n) = value(elem_prod(q*nt(1,n),exp(-0.5*zt(1,n))));
     // Add observation errors
     double sigma = 1.e-3;
     dvector obsError(1,n);
```

```
obsError.fill_randn(rng);
            = elem_prod(yt, exp(sigma*obsError-0.5*sigma*sigma));
REPORT_SECTION
   REPORT(no);
   REPORT(h);
   REPORT(m);
   REPORT(q);
   REPORT(ft);
   REPORT(nt);
TOP_OF_MAIN_SECTION
    time(&start);
    arrmblsize = 50000000;
    gradient_structure::set_GRADSTACK_BUFFER_SIZE(1.e7);
    gradient_structure::set_CMPDIF_BUFFER_SIZE(1.e7);
    gradient_structure::set_MAX_NVAR_OFFSET(5000);
    \verb|gradient_structure|:: set_NUM_DEPENDENT_VARIABLES (5000);
GLOBALS_SECTION
    /**
    \def REPORT(object)
    Prints name and value of \a object on ADMB report % of stream file.
    */
    #undef REPORT
    #define REPORT(object) report << #object "\n" << object << endl;
   #undef COUT
   #define COUT(object) cout<<fixed<<#object "\n"<<object<<endl;</pre>
   #include <iostream>
    #include <iomanip>
    using namespace std;
   #include <admodel.h>
    #include <time.h>
    #include <statsLib.h>
    time_t start, finish;
    long hour, minute, second;
    double elapsed_time;
    static int nf=0;
    adtimer runtime;
    ADJoint code for solving the Baranov Catch equation.
    void deriv_f(void);
    dvariable get_f(const double& ct, const prevariable& m, const prevariable& nt)
        //use newtons root finding method to calc f.
        dvariable ft;
        ft=value(ct/nt);
        double ctmp;
```

```
double df_ctmp;
    for (int iter=1; iter \leq=17; iter++)
         double t1 = value(exp(-m-ft));
         double t2 = value(m+ft);
         ctmp=value(nt*ft*(1.-t1)/t2);
         df_{\text{ctmp}} = value(nt*(1.-t1)/t2 - nt*ft*(1.-t1)/(t2*t2) + nt*ft*t1/t2);
         ft = ((ctmp-ct)/df_ctmp);
         // \text{ cout} << \text{iter} << \text{"} \t" << \text{ctmp-ct} << \text{endl};
         if(fabs(ctmp-ct) \le 1.e-14) break;
    }
    // Push variables on stack
    save_identifier_string("place1");
    nt.save_prevariable_value();
    nt.save_prevariable_position();
    m. save_prevariable_value();
    m. save_prevariable_position();
    save_identifier_string("place2");
    ft.save_prevariable_value();
    ft.save_prevariable_position();
    //Call adjoint code to get derivative
    ADJOINT_CODE(deriv_f);
    return (ft);
}
void deriv_f (void)
    //POP variables off stack in reverse order
    prevariable_position ft_pos = restore_prevariable_position();
    double ad_ft=restore_prevariable_value();
    verify_identifier_string("place2");
    prevariable_position m_pos = restore_prevariable_position();
    double ad_m = restore_prevariable_value();
    prevariable_position nt_pos = restore_prevariable_position();
    double ad_nt = restore_prevariable_value();
    verify_identifier_string("place1");
    double dft=restore_prevariable_derivative(ft_pos);
    //\operatorname{cout} << \operatorname{"dft} \setminus \operatorname{t"} << \operatorname{dft} << \operatorname{endl};
    //derivatives from implicit differentiation.
    double dfb;
    double t1 = (-ad_m - ad_ft);
    double t2 = \exp(t1);
    double t8 = (ad_ft * ad_ft);
    dfb = (-ad_ft * (-0.1e1 + t2) * t1 / ad_nt
         /((-ad_m + ad_ft * ad_m + t8) * t2 + ad_m))*dft;
    // cout << "dfb \ t" << dfb << endl;
```

```
double dfm;
      double t3 = \exp(t1);
      //t8 = F * F;
      dfm = (-ad_ft * (-0.1e1 + (ad_m + ad_ft + 0.1e1) * t3)
         /((-ad_m + ad_ft * ad_m + t8) * t3 + ad_m))*dft;
      // cout << "dfm \ t" << dfm << endl;
      save_double_derivative(dfb, nt_pos);
      save_double_derivative(dfm, m_pos);
   }
FINAL_SECTION
   time(&finish);
   elapsed_time=difftime(finish, start);
   hour=long(elapsed_time)/3600;
   minute=long (elapsed_time)%3600/60;
   second=(long(elapsed_time)%3600)%60;
   cout << "--Start time: "<< ctime(&start) << endl;</pre>
   cout << "--Finish time: "<< ctime(&finish)<< endl;</pre>
   cout <<"—Runtime: ";
   cout << "--Runtime in milliseconds: ";
   cout << runtime.get_elapsed_time_and_reset() << endl;</pre>
```