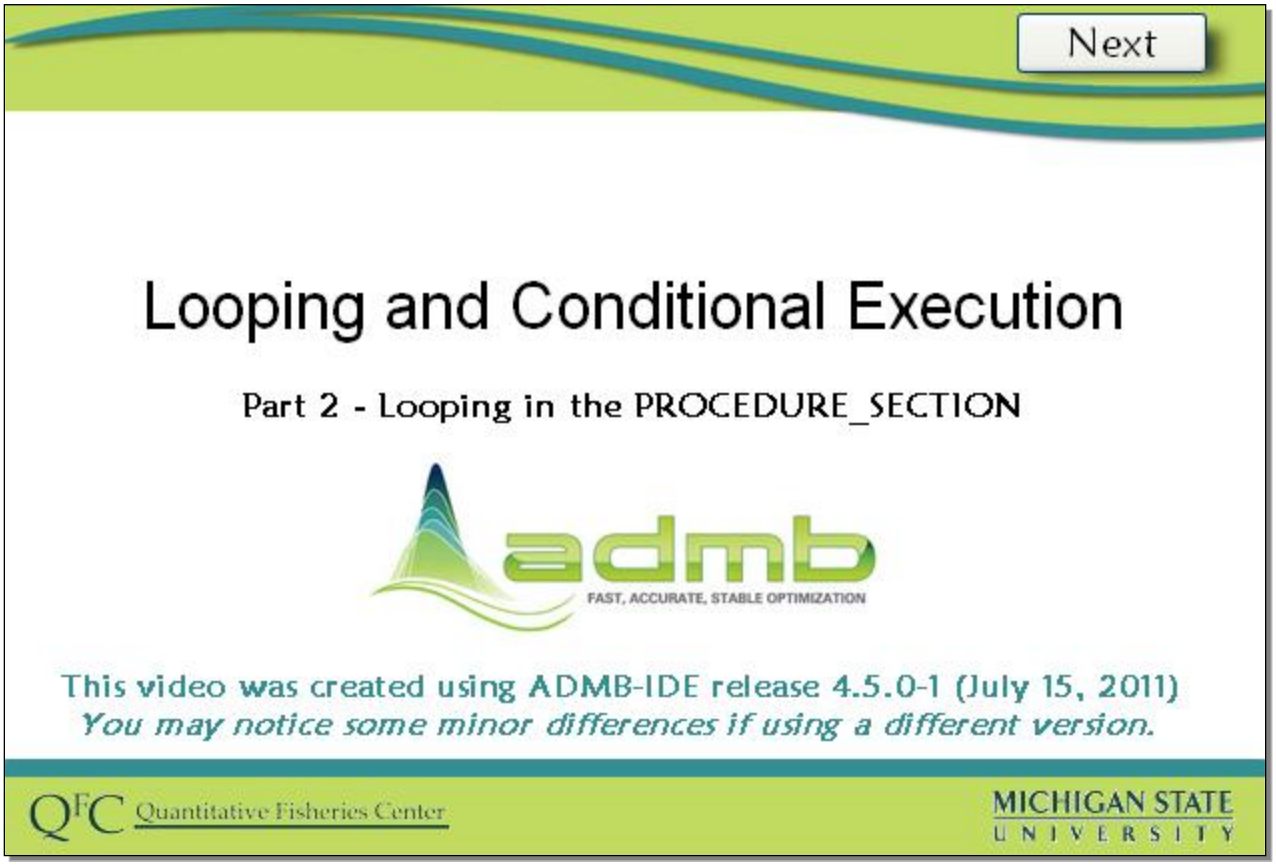
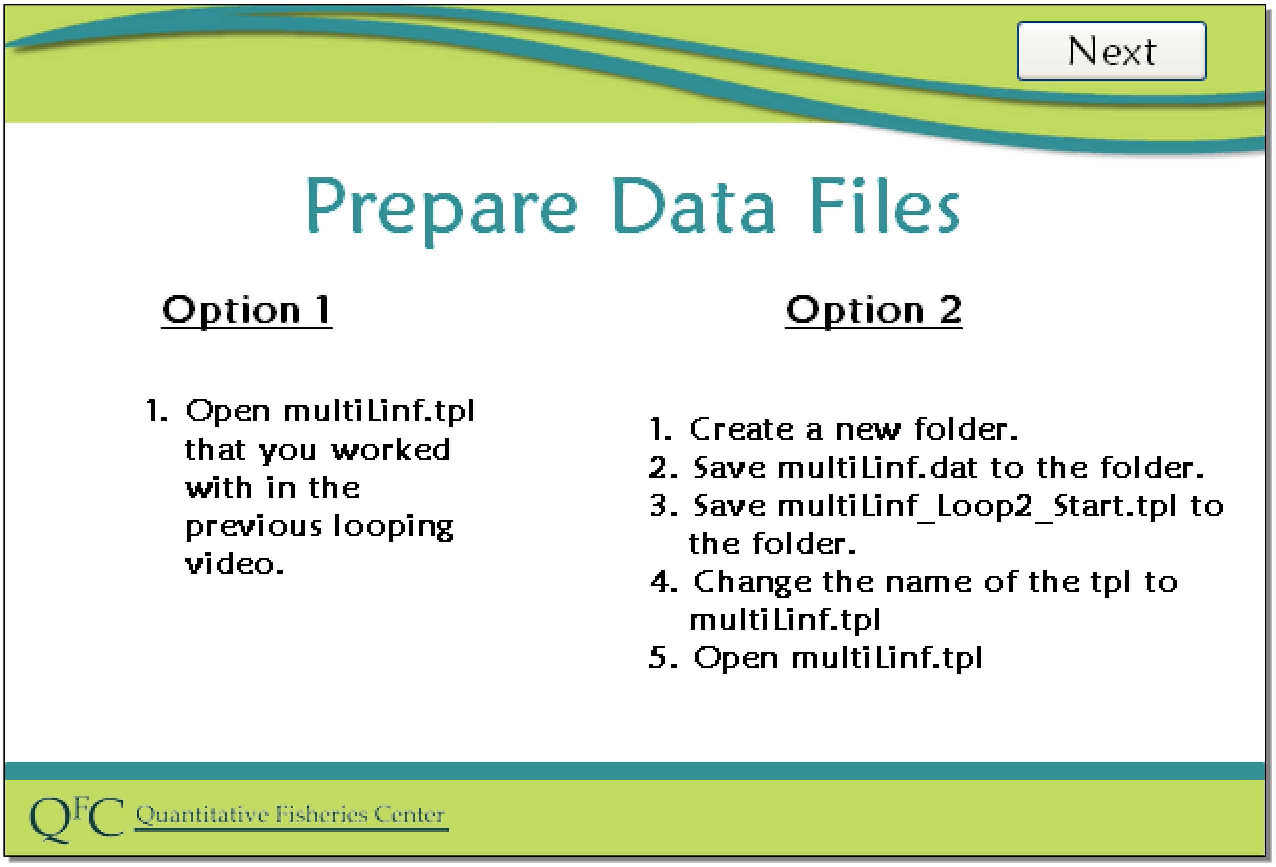


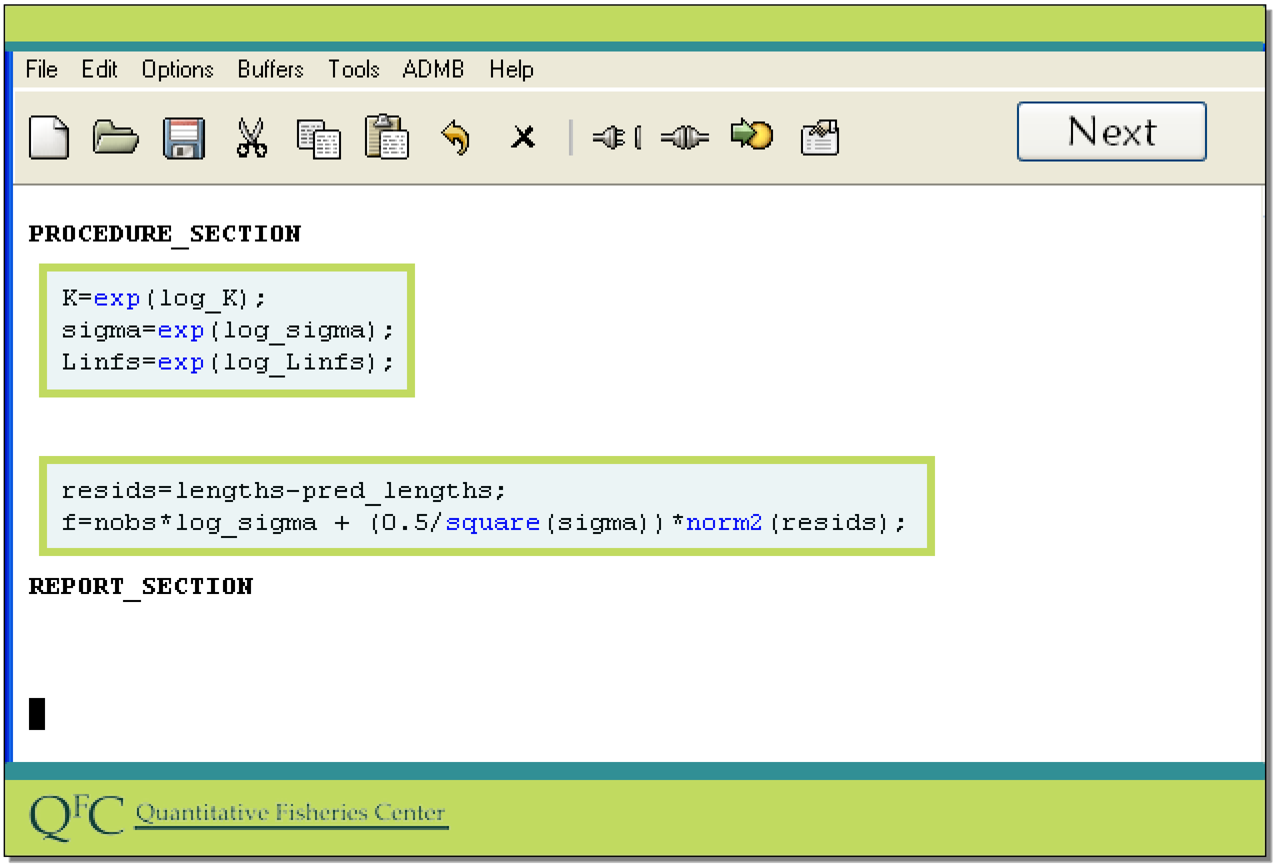
This video, part 2 completes our coverage of loops. For the very simple model we have worked with so far we have not had to deal with how to execute tasks repeatedly in a loop or how to execute tasks depending upon a condition.



Prepare and open your files. Either open the multiLinf.tpl that you worked with in the previous looping video or:

1. Create a new folder.
2. Save multiLinf.dat to the folder.
3. Save multiLinf_Loop2_Start.tpl to the folder.
4. Change the name of the tpl to multiLinf.tpl
5. Open multiLinf.tpl

Click next when you are ready.

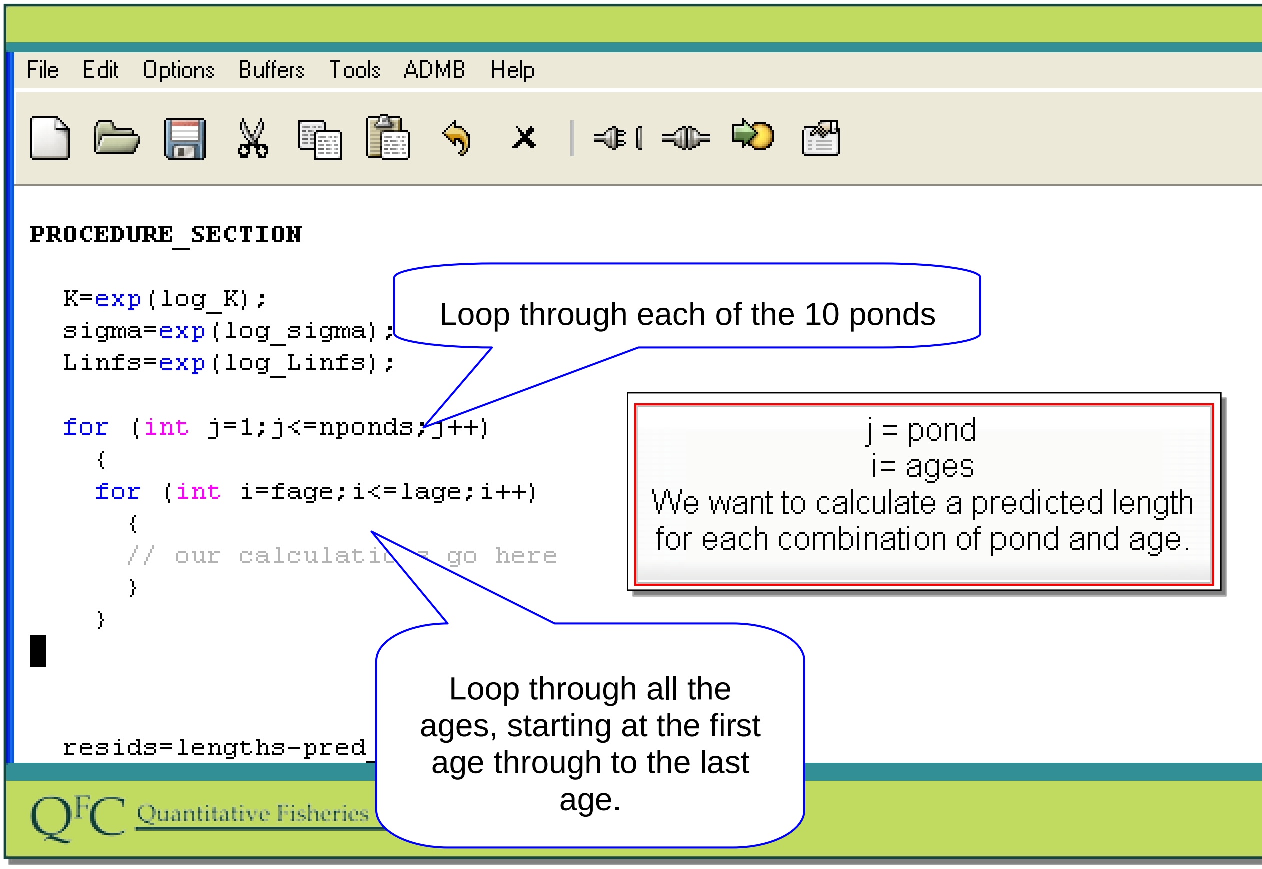


We now go to the procedure section. The first thing that happens here is the usual back-transformation of log-scale parameters. Type these lines and press next to continue.

Slide Code:

```
K=exp(log_K);
sigma=exp(log_sigma);
Linfs=exp(log_Linfs);

resids=lengths-pred_lengths;
f=nobs*log_sigma + (0.5/square(sigma))*norm2(resids);
```

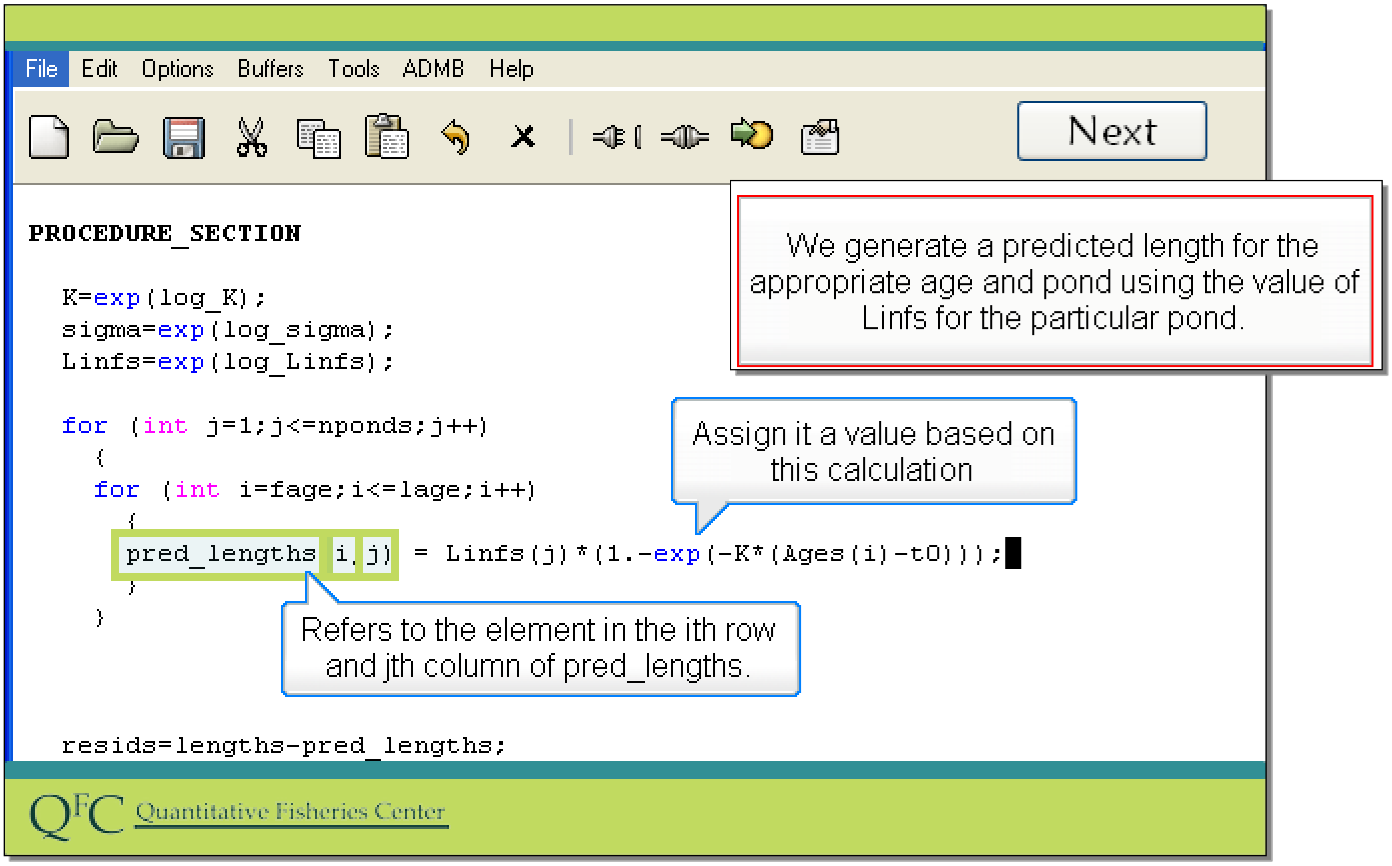


Next, we generate predicted lengths and in the process see that loops can be nested. In this case we will loop over ponds, and will use “j” to indicate which pond. For each pond, we will loop over ages, and use “i” to indicate which age. Our plan is to calculate a predicted length for each combination of pond and age. So we start by setting up the “for loop” structures. The outside loop over j says we will let j take values from 1 to the number of ponds. For each j the inside loop says we will let i take values from the first age through to the last age.

Thus, any statements we put inside the inner set of braces get calculated for every pond and age combination.

Slide Code:

```
for (int j=1;j<=nponds;j++)
 {
  for (int i=fage;i<=lage;i++)
  {
  // our calculations go here
  }
 }
```

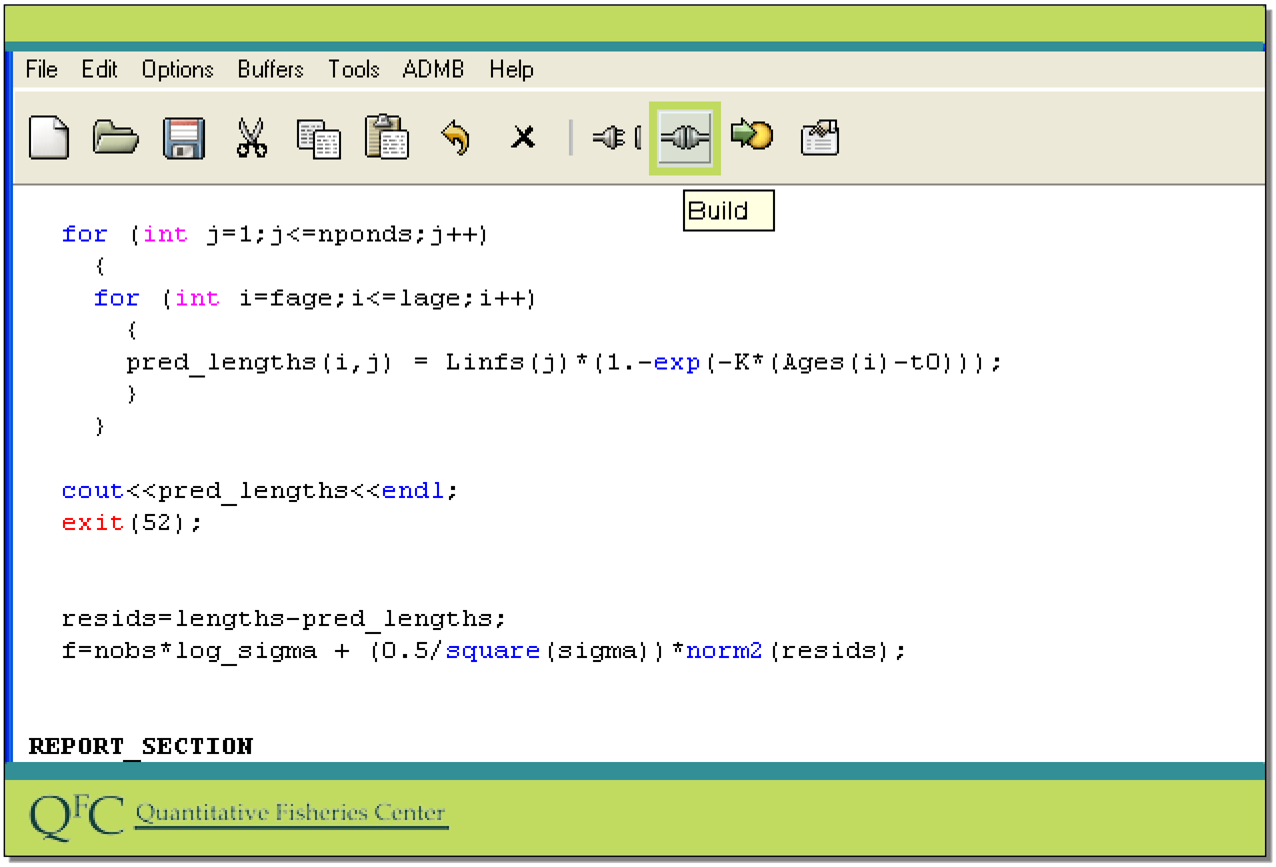


We have just one line of calculation to add there. This generates a predicted length for the appropriate age and pond, using the value of Linfs for that pond referenced by "j" and the value of Ages, referenced by i.

Notice that when we refer to a particular element of a matrix we just follow the matrix name with parentheses and inside we put the row index, i, then a comma, and then the column index, j. So pred_lengths(i,j) refers to the element in the ith row and jth column of pred_lengths. In our code here we are assigning it a value.

Slide Code:

```
pred_lengths(i,j) = Linfs(j)*(1.-exp(-K*(Ages(i)-t0)));
```

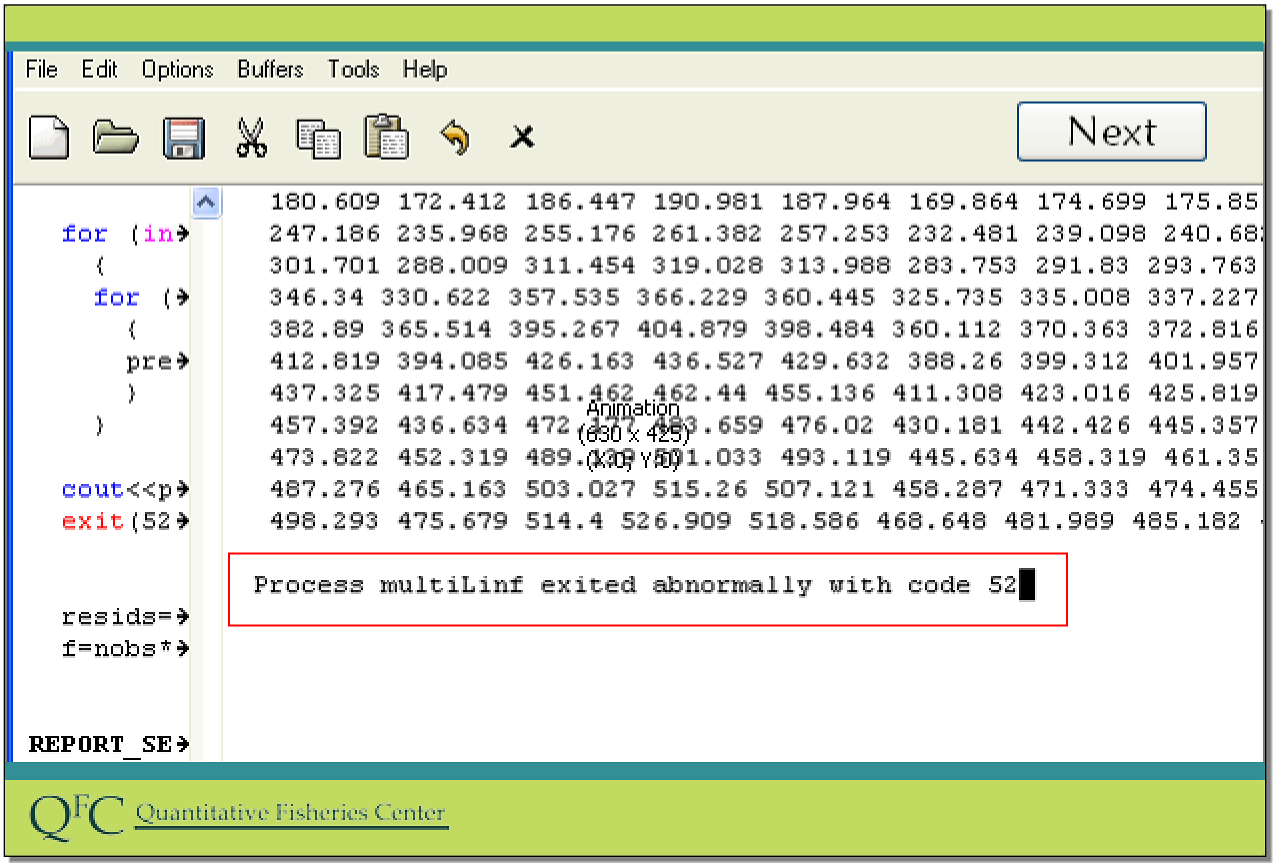


1. We now add a cout and exit statement to check that our code is working.

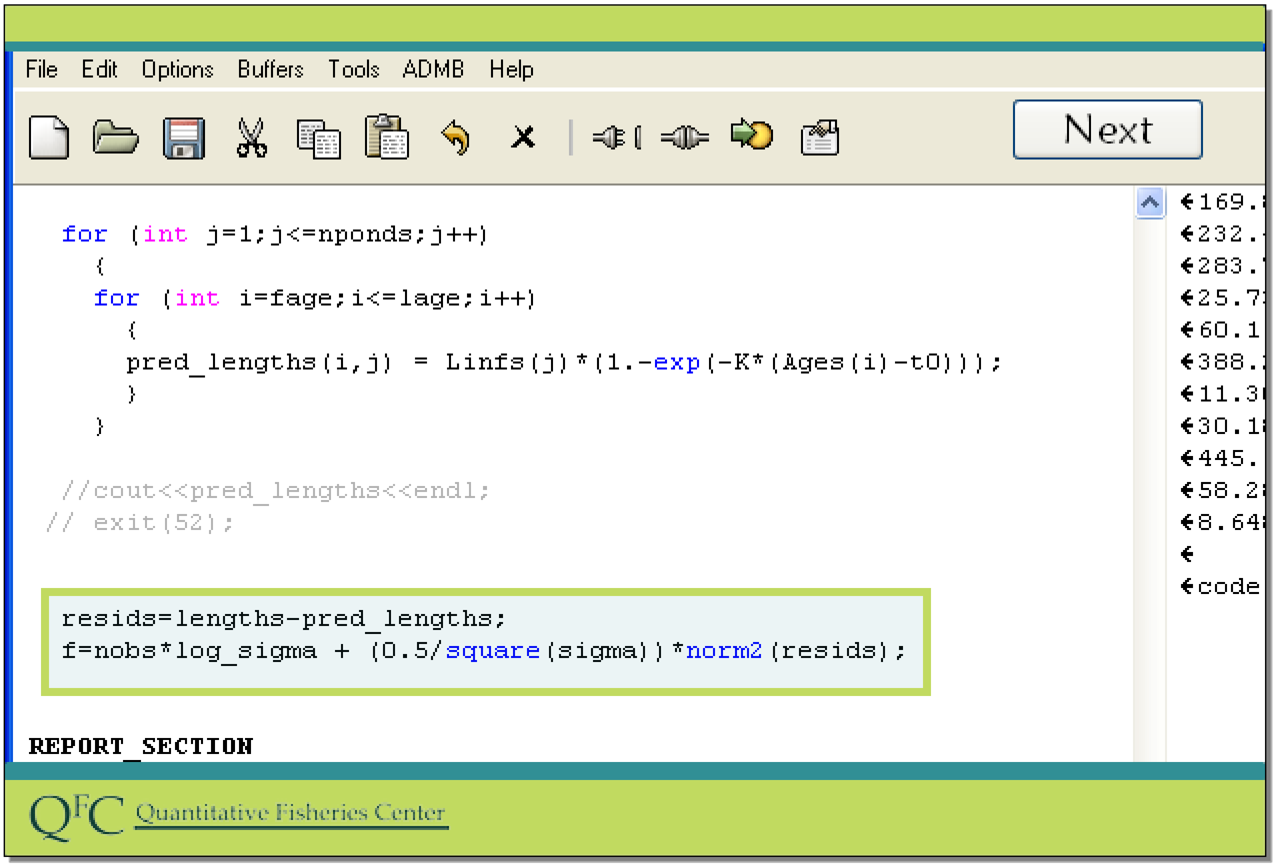
Slide Code:

```
cout<<pred_lengths<<endl;
exit(52);
```

2. We build and run our program.



Indeed the result is predicted lengths that appear to approach an asymptote for each pond or column as age increases down each column. Also, notice that the program exited abnormally with exit code 52 that we just entered.



1. We now need to comment out our cout and exit statements now that we know our program is working.

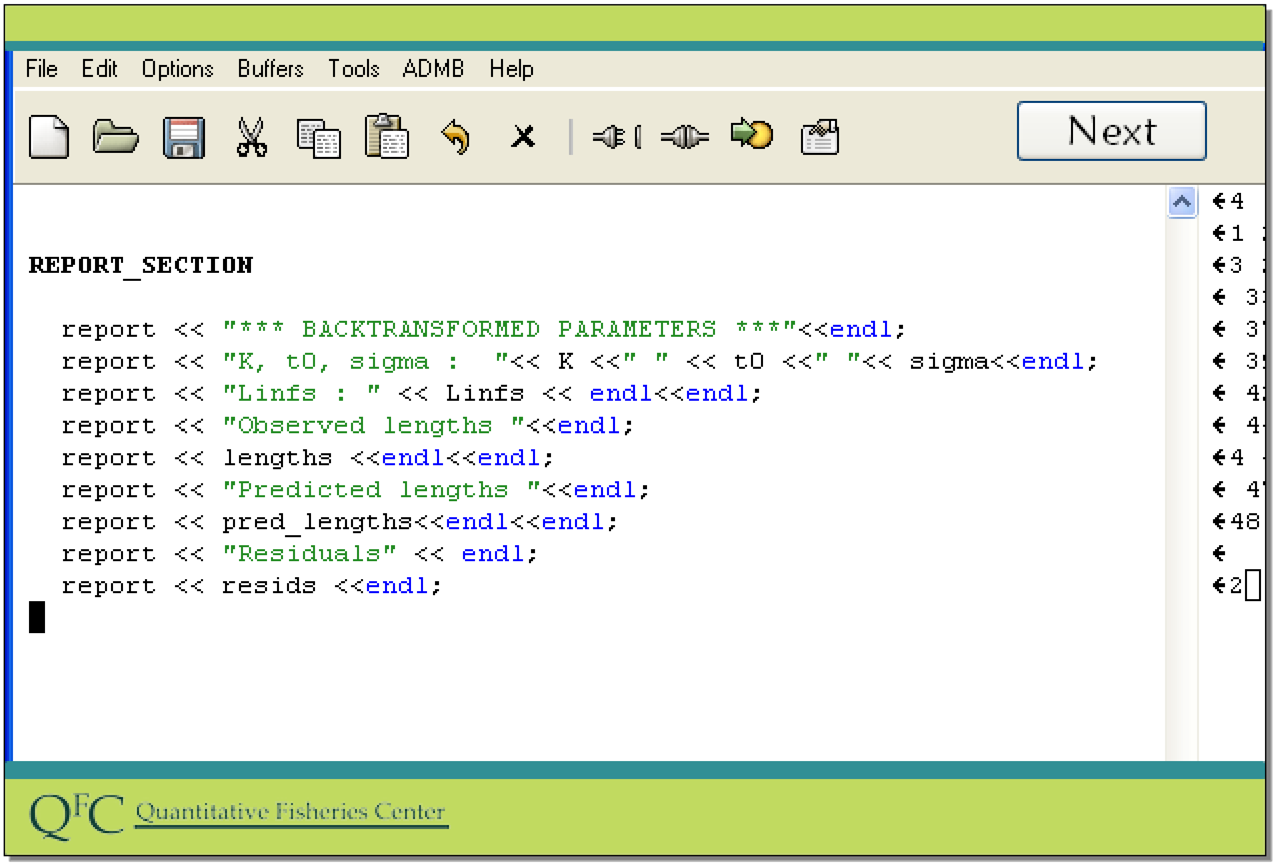
Slide Code:

```
// cout<<pred_lengths<<endl;
// exit(52);
```

2. Our model is completed by including the calculation of residuals, as the difference between predictions and observations, and by including the calculation of the objective function, which is the same form we have used in previous videos assuming a normal distribution for the deviations between observed lengths and the value expected given age and pond. Type these two lines if you are following along and press next to continue.

Slide Code:

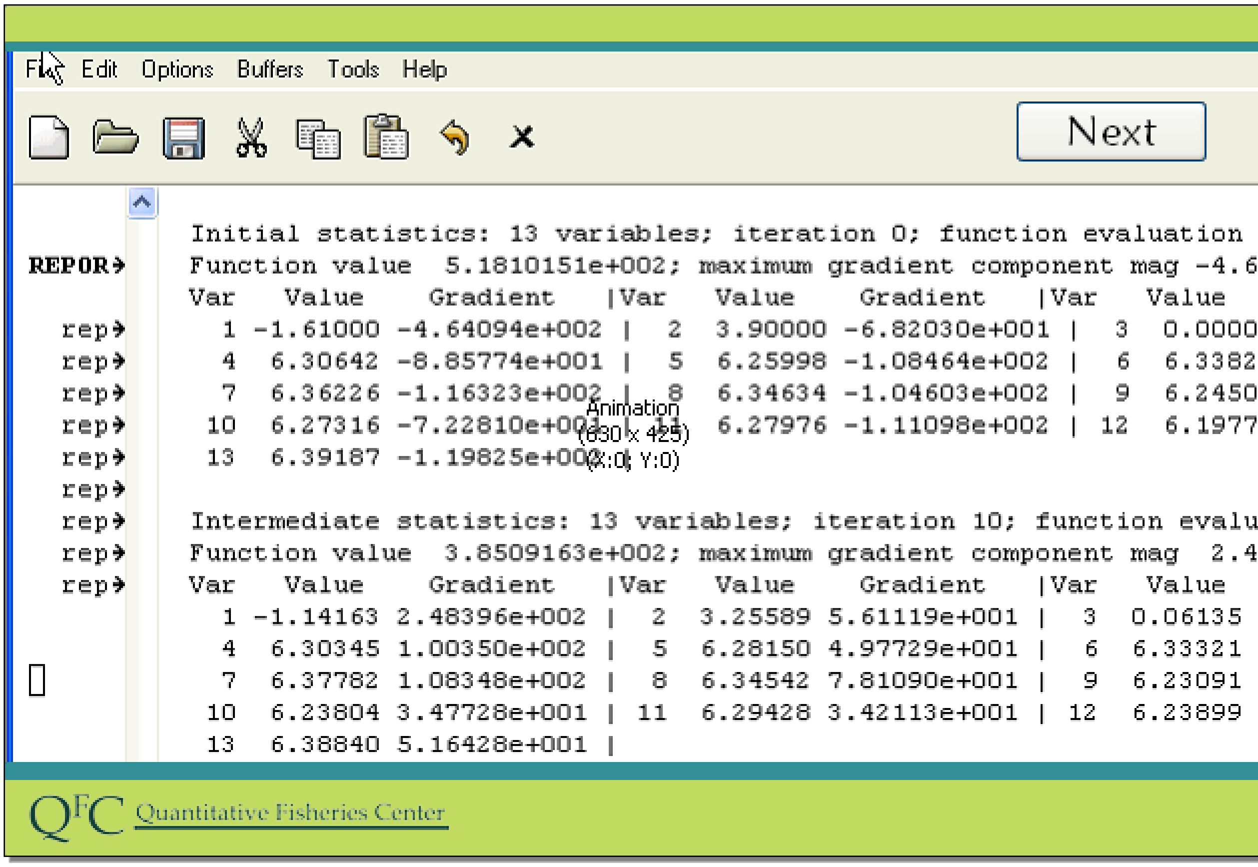
```
resids=lengths-pred_lengths;
f=nobs*log_sigma + (0.5/square(sigma))*norm2(resids);
```



We also include some lines in the report section to output some results of interest. Type in this code and click next to continue.

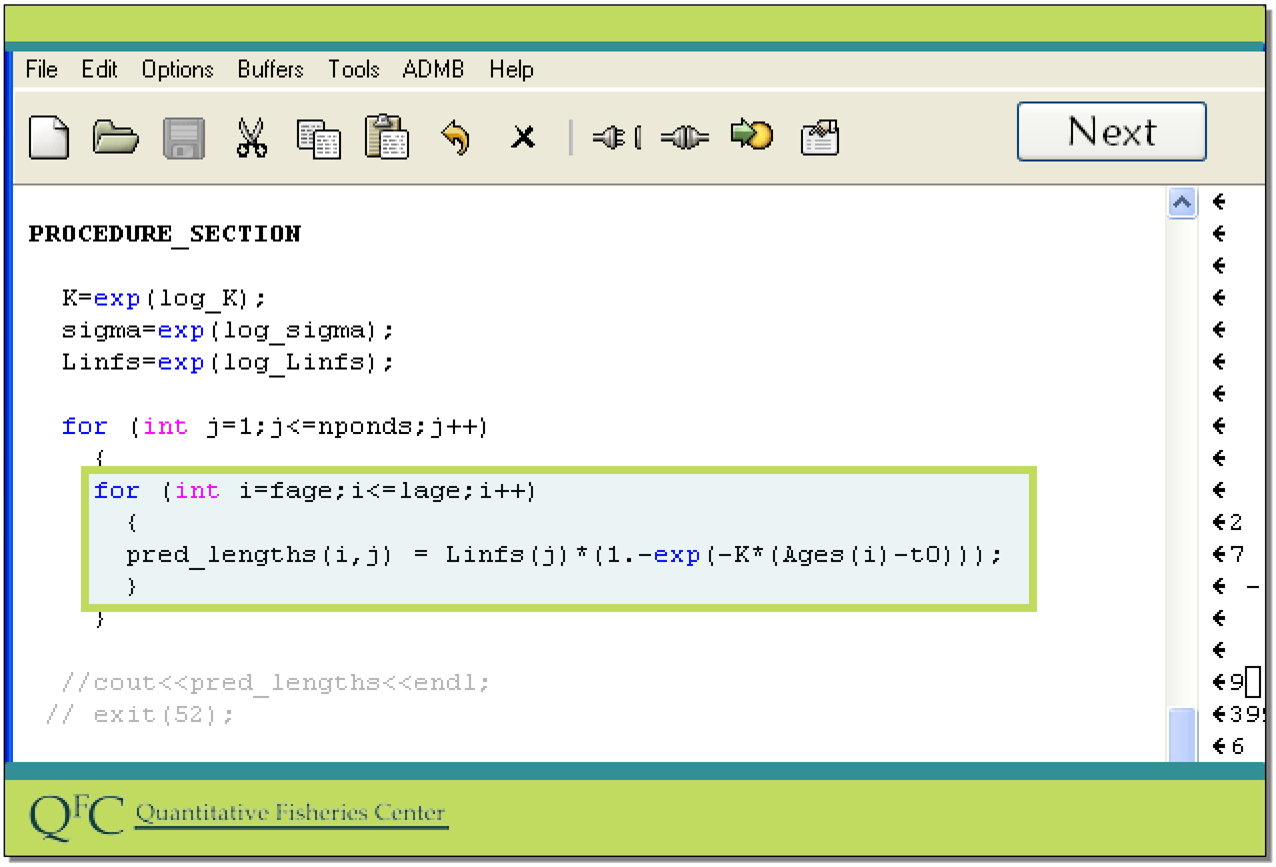
Slide Code:

```
report << "*** BACKTRANSFORMED PARAMETERS ***"<<endl;
report << "K, t0, sigma :  "<< K <<" " << t0 <<" "<< sigma<<endl;
report << "Linfs : " << Linfs << endl<<endl;
report << "Observed lengths "<<endl;
report << lengths <<endl<<endl;
report << "Predicted lengths "<<endl;
report << pred_lengths<<endl<<endl;
report << "Residuals" << endl;
report << resids <<endl;
```

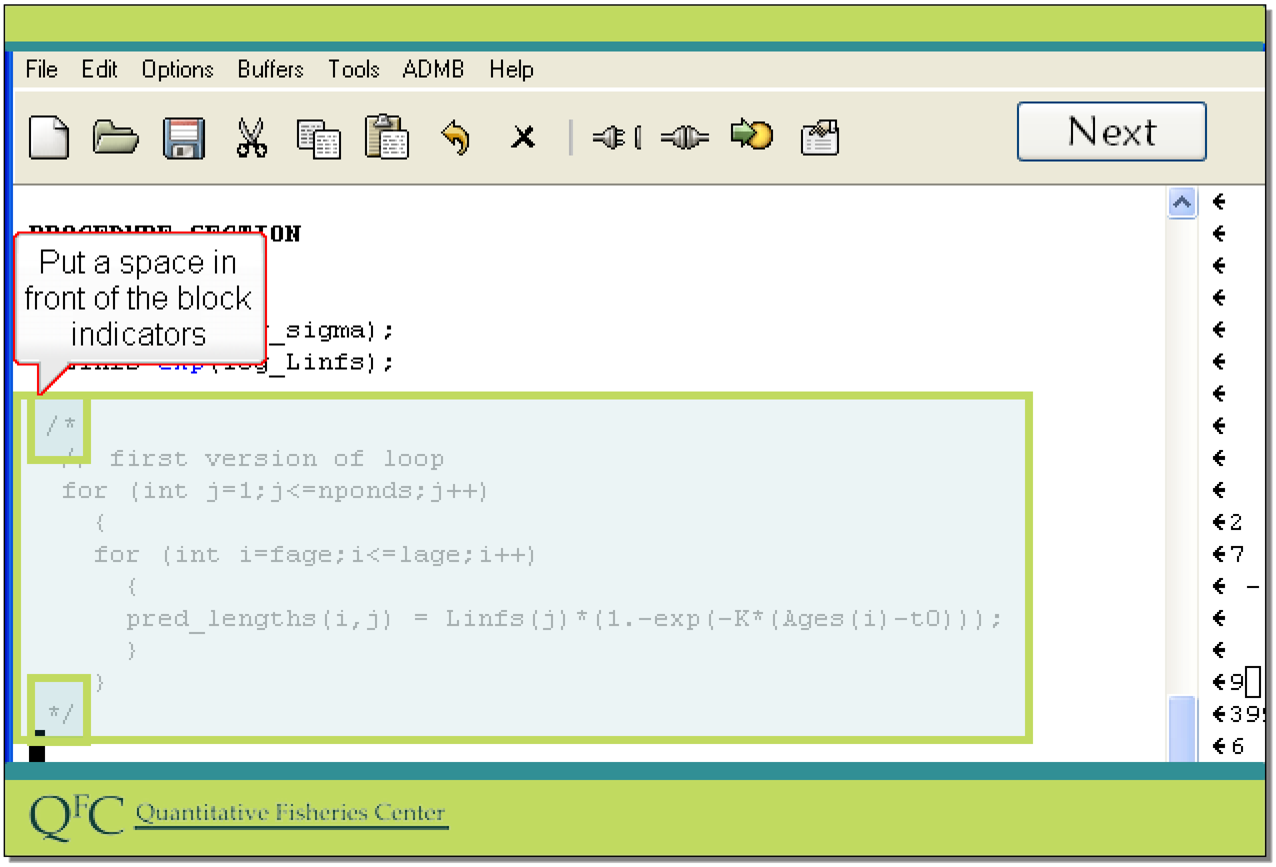


We now build and run the program. It appears to converge correctly.

A quick look at the report file suggests a reasonable set of results based both on the pattern of predicted values with age and the residuals. Go to ADMB then down to view report. Notice the report file has our text that we just entered in the REPORT_SECTION.



Our intent in this video up to this point was to illustrate the use of loops, and not necessarily to code things in the most efficient way. Often loops can be avoided in admb by using higher level functions and these tend to do calculations in a more efficient way. For example, we could replace the inner loop of the code we just wrote to generate predicted values for all the ages for a pond at once.



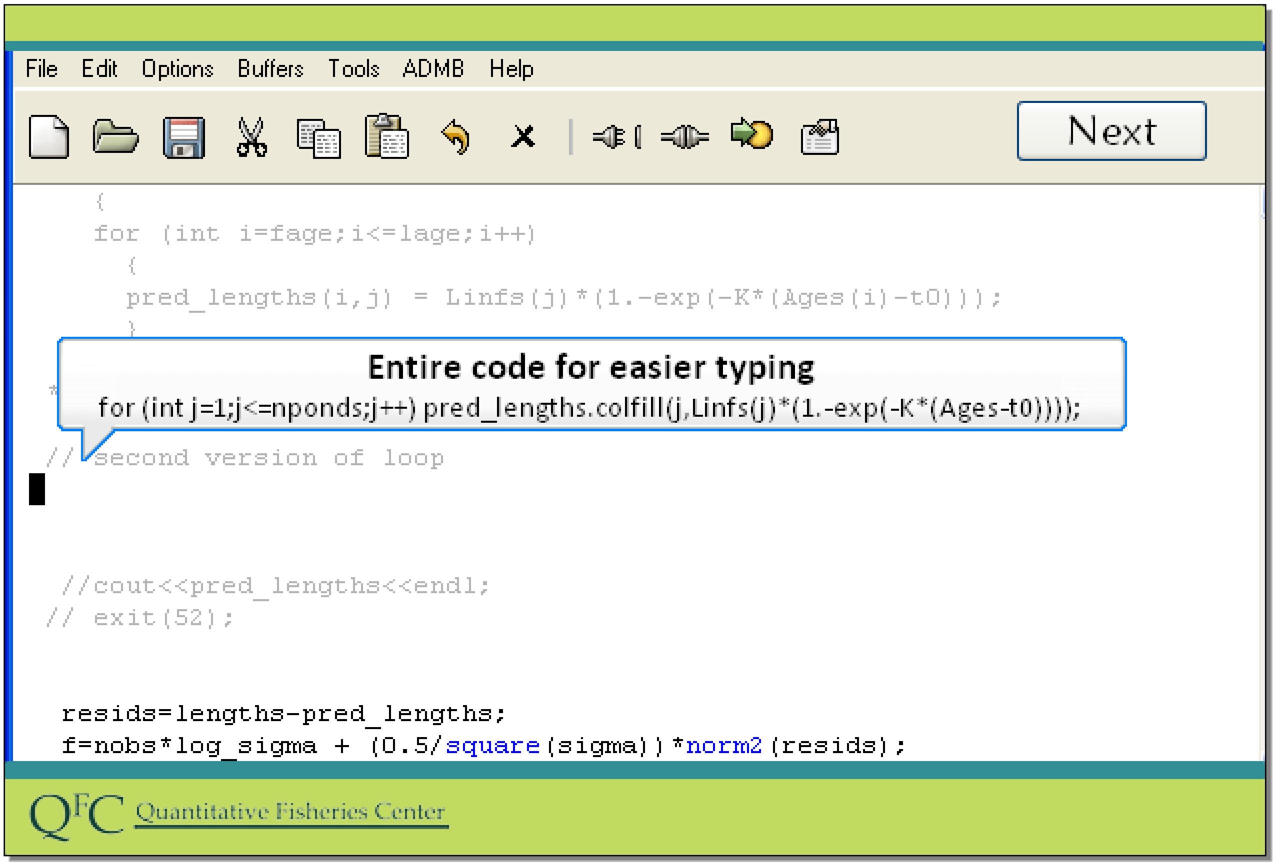
We will demonstrate this new loop, but first, since we are going to have two versions of the loop let's add a comment saying our original code is version one and then enclose the entire double loop and its comment with indicators turning it all into non executable comment code.

To create the block of comments we start with slash asterisk and end with star asterisk. These indicators of the start and end of the block should have one space before them. Such commenting out of blocks of lines is often useful.

Slide Code:

```
 /*
// first version of loop
for (int j=1;j<=nponds;j++)
  {
   for (int i=fage;i<=lage;i++)
   {
   pred_lengths(i,j) = Linfs(j)*(1.-exp(-K*(Ages(i)-t0)));
   }
  }
*/
```

Slide 25 - Slide 25

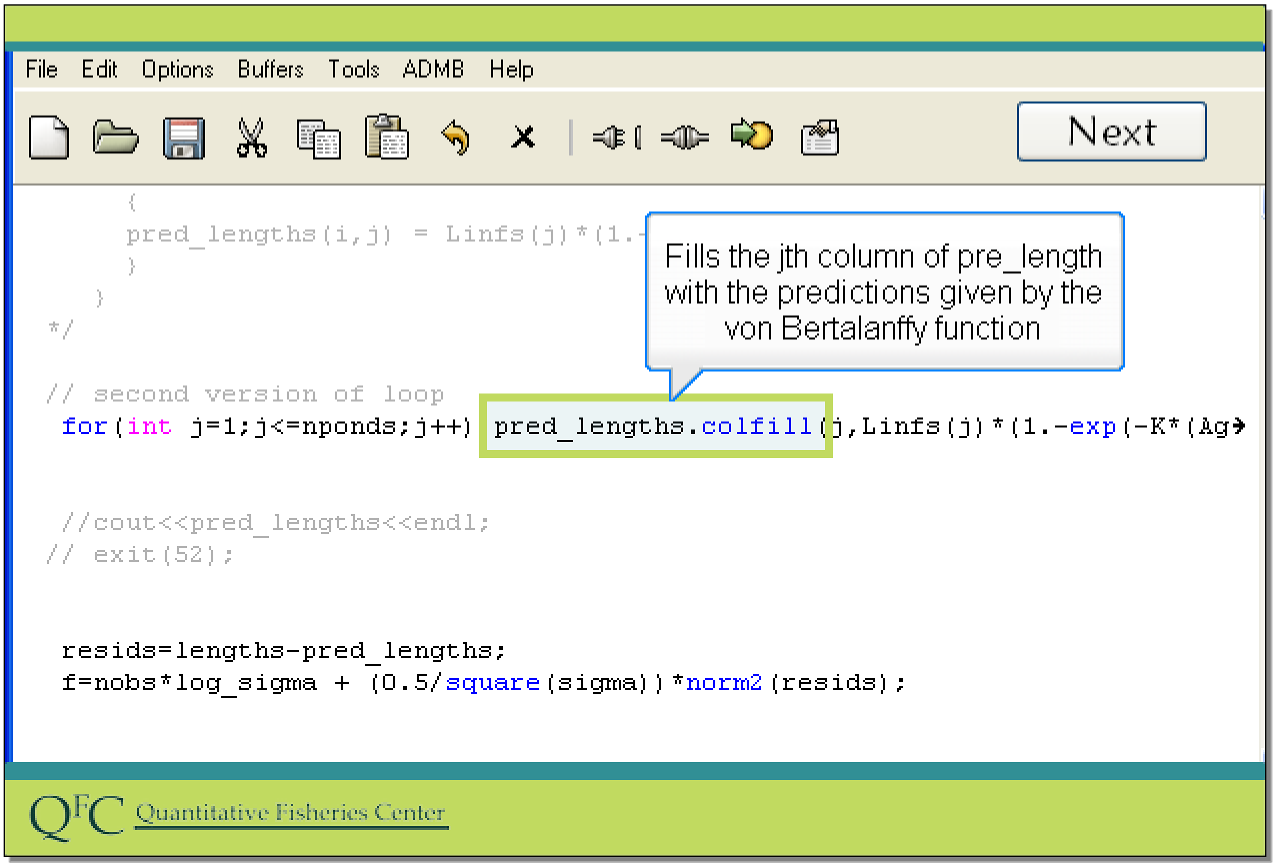


Slide notes

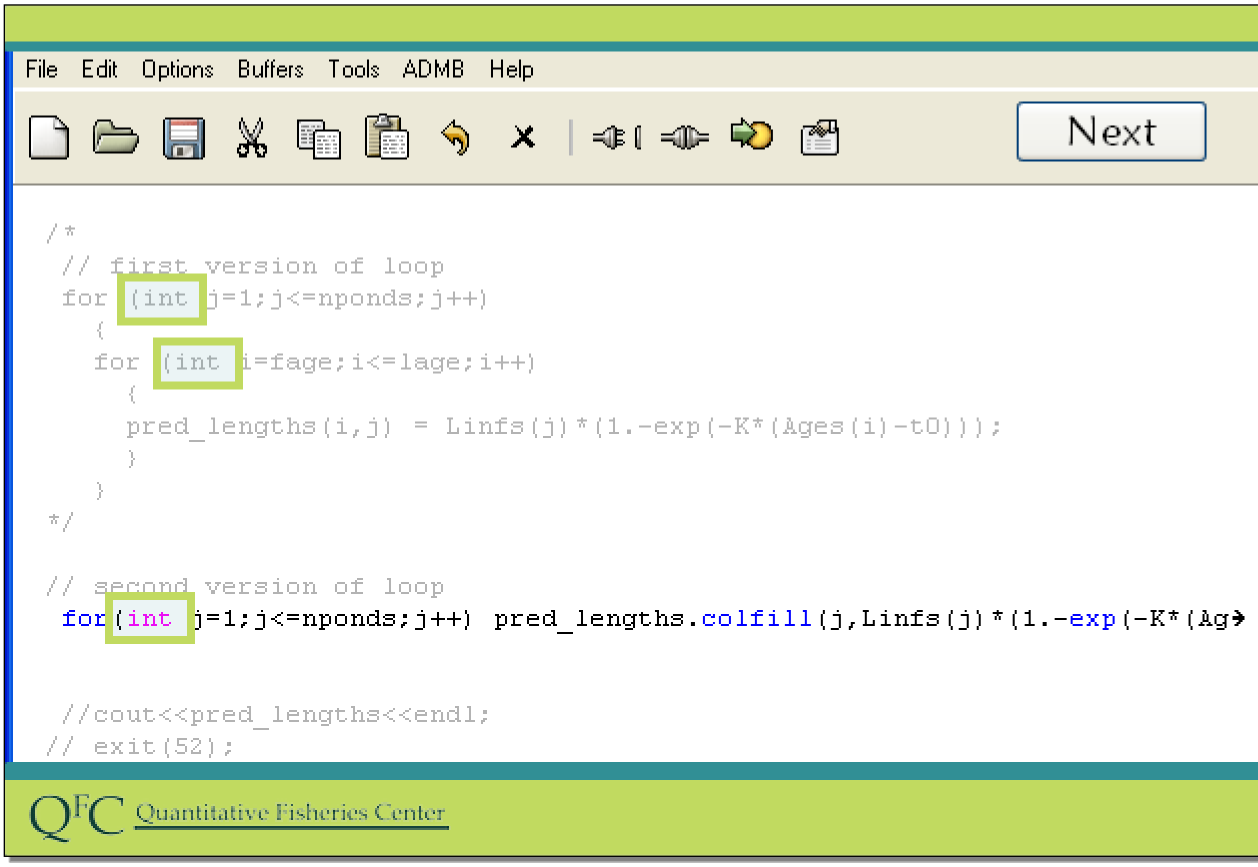
1. Let's add a comment indicating we are about to write the second version of the loop
2. Here we will type the entire code from the first double loop on one line. Finish typing this code then click next to continue.

Slide Code:

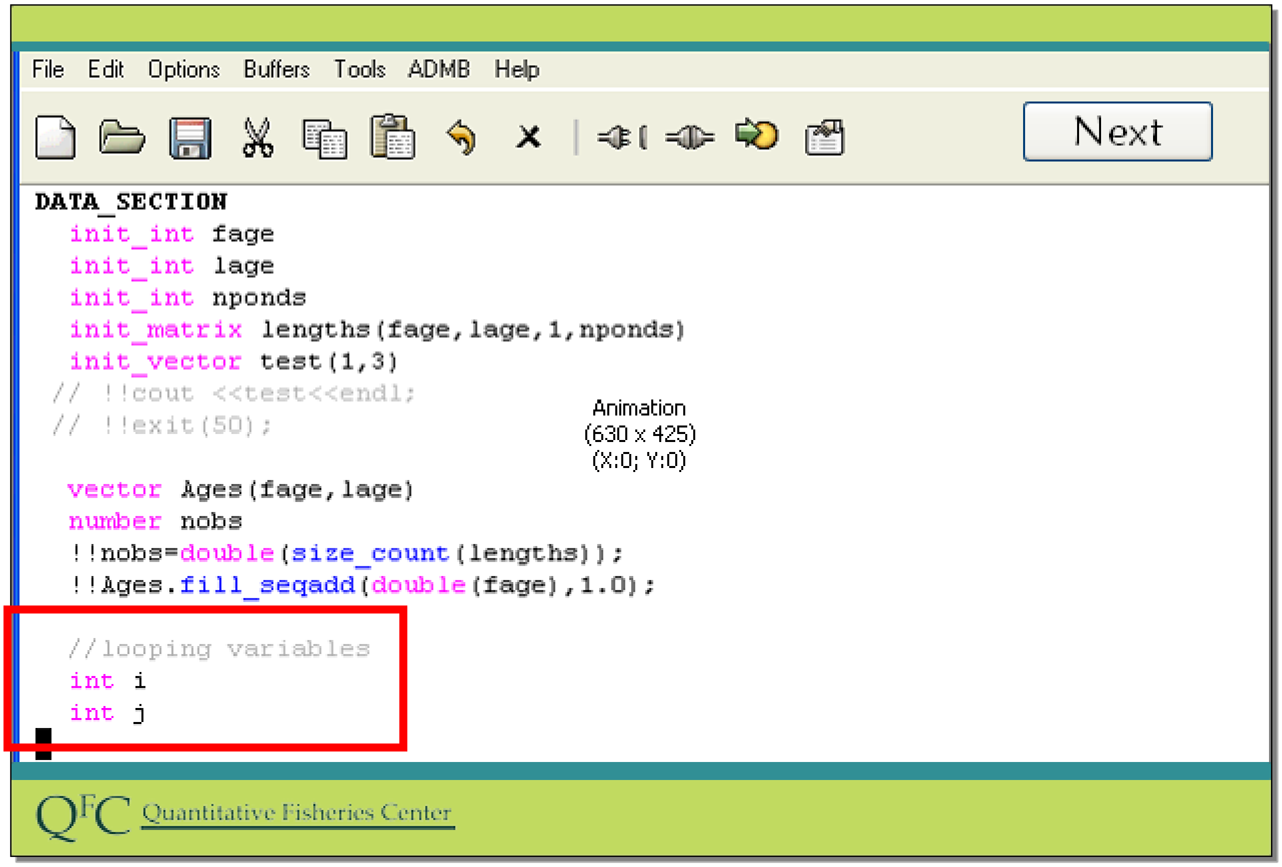
```
// second version of loop
  for( j=1;j<=nponds;j++) pred_lengths.colfill(j,Linfs(j)*(1.-exp(-K*(Ages-t0))));
```



Here we use pred_lengths dot colfill to fill the jth column of pred_lengths with the predictions given by the von Bertalanffy function. The von Bertalanffy function used here is based on single numbers for the parameters and a vector of ages so it produces a vector of predictions for the jth pond. You can read more about using colfill to fill columns of a matrix in the admb manual.



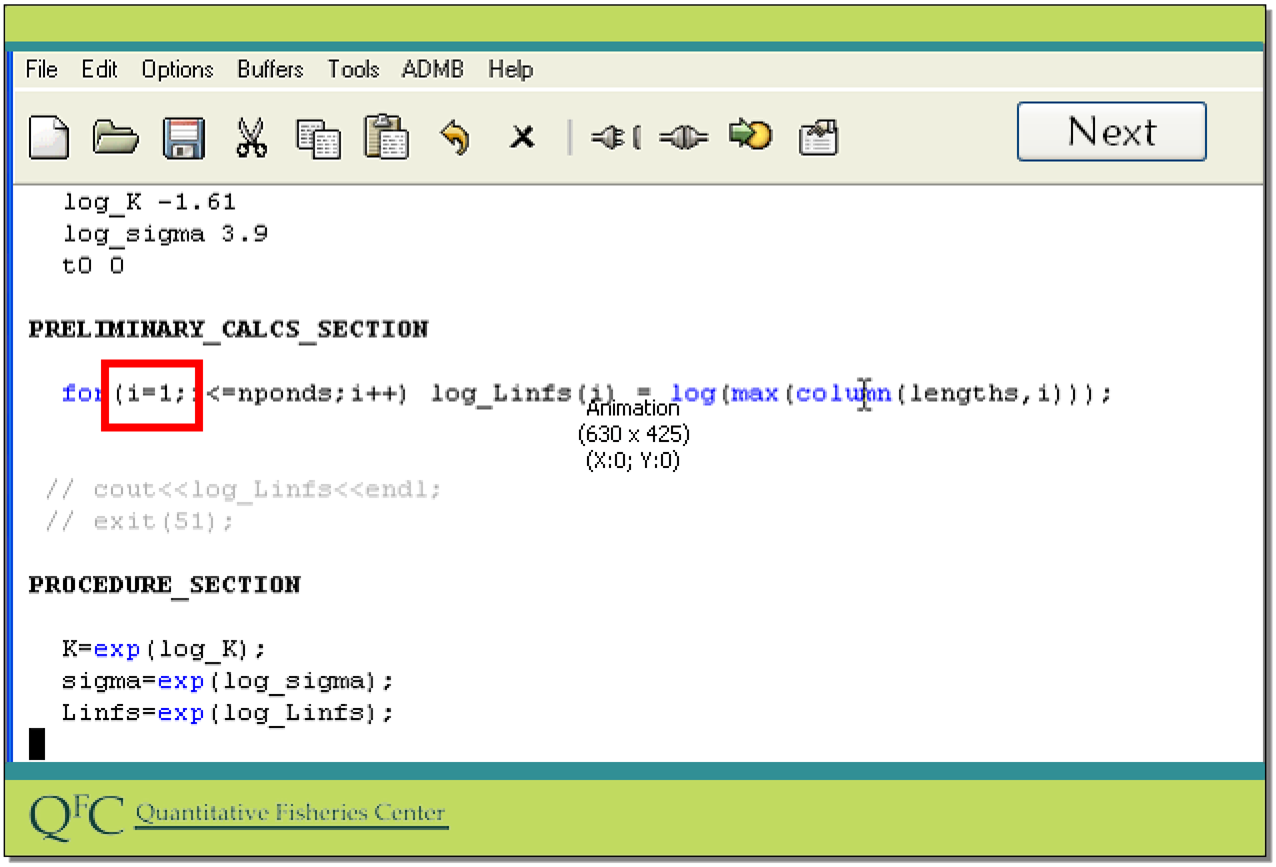
In our loops up to now, we have defined the integers we are using as indices in our loops at the start of our loop. An alternative approach would be to define them one time in the data section. We then would not have to repeatedly use int to define them each time we do a loop. Besides saving on typing this can make your tpl more portable. While defining the index variables for each for loop is in accord with ANSI C++ standards, which are followed by the gnu C++ compiler packaged with admb-IDE, not all compilers have followed the standards. Doing it the way we had been showing you, there are some compilers that will remember that your looping variables exist outside the loops and so when you redefine them for another loop an error message will result. This is becoming less of a problem, but we will demonstrate how to adjust your code so it will work on compilers that both follow the standard and those that do not.



First we will define the looping variables in data section.

Slide Code:

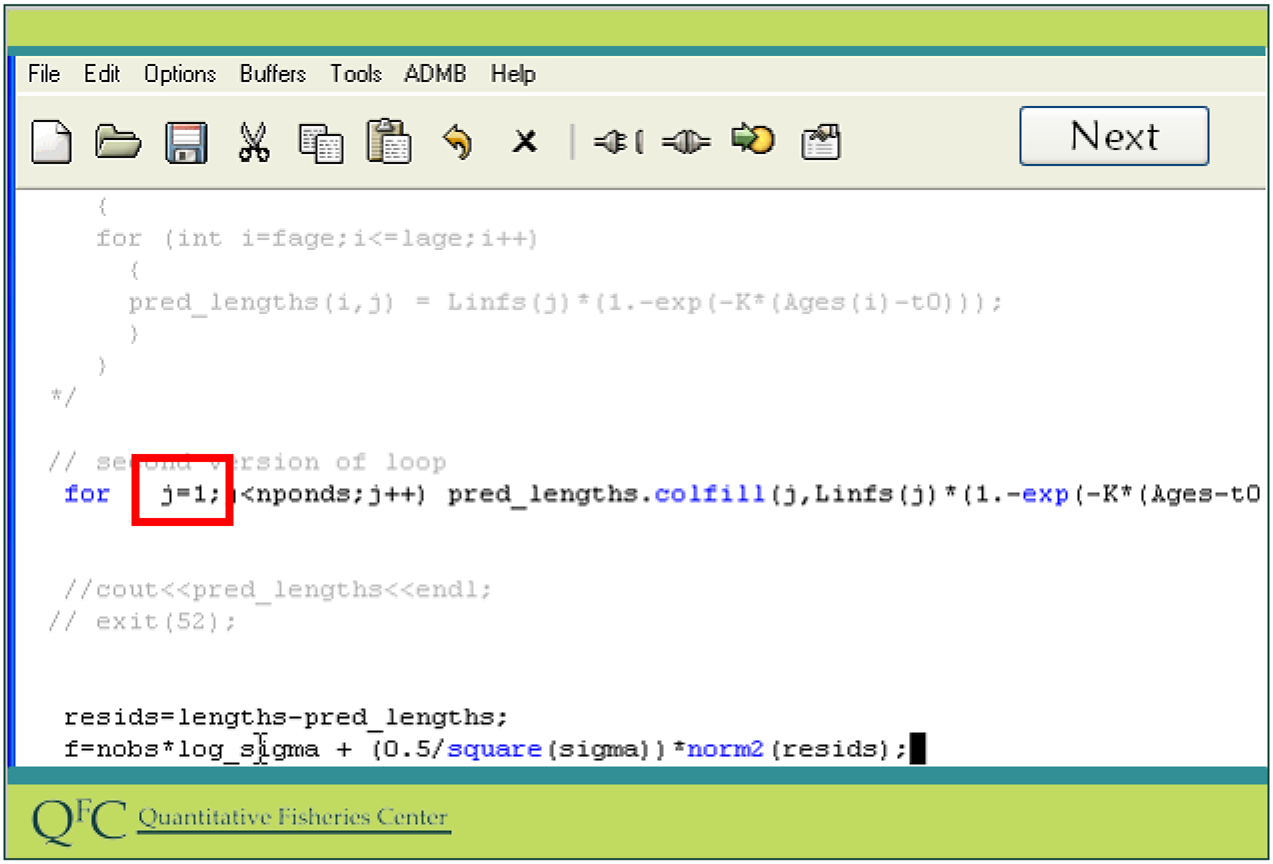
```
// looping variables
 int i
 int j
```



We need to modify our code in the PRELIMINARY_CALCS_SECTION

Slide Code:

Remove int in front of i=1;



and in the PROCEDURE_SECTION. Now your code will work on compilers that follow the standard and those that do not.

Slide Code:

Remove int in front of j=1;

This ends our videos on looping. Next we will cover the use of conditional statements.