Adjoint Code for LU Decomposition

Dave Fournier
Otter Research

September 21, 2009

The purpose of this note is to explain how to create the adjoint code for a dvariable type function. The idea is to take a non trivial application and to create the adjoint code without having to analyze the original code very much.

The main difficulty is that of dealing with overwrites of the variables when producing the adjoint code. This is handled by creating a version of the code where the overwritten variables are turned into arrays and the array index is incremented each time the variable is overwritten. As an example we take the LU decomposition of a square matrix.

The code for the dvariable version of ludecomp is based on the description of the LU decomposition given in NR. A major difference is that NR uses one (square) matrix to hold the L and U parts for the decomposition. In modern computers this is disadvantageous because it means that depending on the storage order either the elements of the L or U parts of the matrix will be accessed in a highly non sequential manner in memory necessitating frequent refilling of the on chip cache. For ADMB which stores matrices by row it is advantageous to store the transpose of the U matrix rather than the matrix itself. One can see this from the NR code for the LU decomposition in the main inner loops where almost all the computations which occur are of the form

```
    sum -= alpha(i,k) * beta(k,j)
```

with the inner sum over k. Instead of the U part of the decomposition named beta, we store the transpose of U to be called gamma. The code becomes

```
  sum -= alpha(i,k) * gamma(j,k)
```

For large matrices this can produce a large performance increase. *because?*

To store the LU decomposition in this manner, it is useful to define an appropriate data structure (class), cltudecomp.

```
class cltudecomp
{
  dmatrix L;
  dmatrix U;
  ivector indx;
  double sign;
public:
  void initialize(void)
  {
    indx.initialize();
    sign=0;
    L.initialize();
    U.initialize();
    for (int i=L.indexmin();i<=L.indexmax();i++)
    {
      L(i,i)=1.0;
    }
  }
```

```
    cltudecomp(int l,int u) : indx(l,u)
    {
      ivector iv(l+1,u);
      iv.fill_seqadd(l,1);
      L.allocate(l+1,u,l,iv);
      ivector iv1(l,u);
      iv1.fill_seqadd(l,1);
      U.allocate(l,u,l,iv1);
    }
    dmatrix & get_L(){ return L;}
    dmatrix & get_U(){ return U;}
    ivector & get_index(){ return indx;}
    double & get_sign(){ return sign;}
    // overload () (int,int) to look like Numerical Recipes
    double & operator() (int i,int j){ if (i>j)
                                          return L(i,j);
                                        else
                                          return U(j,i);
                                      }
  };
```

A request from a geezer with old eyes. It is hard to distinguish between 'l' (lower case 'L') and '1' (the integer number) in some type faces. It causes *me no end of trouble.*

A major advantage is to overload the () (int,int) access operator so that accesses to the data structure look identical to those in NR for their storage of the LU decomposition. One might think that having an access operator like this which must evaluate a conditional expression each time the function is called would be inefficient. However it is only used n^2 times while there are n^3 operations in the LU decomposition. *This is a great assistance in writing the adjoint, but is it possible to go back and optimize by accessing L and U directly in "production" code?*

Now we are interested in producing a dvariable version of the LU decomposition in the simplest manner possible to illustrate how easy this is when one follows certain procedures. It turns out that the easiest way is to carry out the LU decomposition twice: first when it is needed in the original calculations, and second when one is doing the calculations necessary for obtaining the derivatives. As mentioned in NR, for numerical stability, is necessary to use partial pivoting. This means that the LU decomposition is actually not done for the matrix itself, but for a matrix obtained by reordering the rows of the original matrix. However when we do the LU decomposition the second time we already know what the reordering is. This enables us to reorder the matrix first and do the LU decomposition without partial pivoting. The result is that the code we use for deriving the derivatives is simpler.

```
// LU decomp wihout partial pivoting
cltudecomp ludecomp(const dmatrix & M)
```

```
{
  int mmin=M.indexmin();
  int mmax=M.indexmax();
  cltudecomp clu(mmin,mmax);

  // get upper and lower parts of LU
  dmatrix & alpha = clu.get_L();
  dmatrix & gamma = clu.get_U(); // gamma is the transpose of beta
  // copy M into alpha and gamma
  for (int i=mmin;i<=mmax;i++)
  {
    for (int j=mmin;j<=mmax;j++)
    {
      clu(i,j)=M(i,j);
    }
  }
  for (int j=mmin;j<=mmax;j++)
  {
    int i;
    for (i=mmin+1;i<j;i++)
    {
      // using subvector here
      clu(i,j)-=alpha(i)(mmin,i-1)*gamma(j)(mmin,i-1);
    }
```

```
      for (i=j;i<=mmax;i++)
      {
        // using subvector here
        if (j>1)
        {
          clu(i,j)-=alpha(i)(mmin,j-1)*gamma(j)(mmin,j-1);
        }
      }
      if (j!=mmax)
      {
        double z= 1.0/gamma(j,j);
        for (i=j+1;i<=mmax;i++)
        {
          alpha(i,j)*=z;
        }
      }
    }
    return clu;
  }
```

Using the cltudecomp object clu, we can access the LU either via its constituents alpha and gamma or in its entirety clu as in the line of code

```
clu(i,j)-=alpha(i)(mmin,i-1)*gamma(j)(mmin,i-1);
```

This greatly simplifies coding up the algorithm based on its description in

NR (at least in my opinion). For the moment assume that the components of clu do not get overwritten. Then the adjoint code for the derivatives corresponding to the line of code

```
//clu(i,j)-=alpha(i)(mmin,i-1)*gamma(j)(mmin,i-1);
```

is

```
dfalpha(i)(mmin,i-1)-=dfclu(i,j)*gamma(j)(mmin,i-1);
dfgamma(j)(mmin,i-1)-=dfclu(i,j)*alpha(i)(mmin,i-1);
```

However the components of clu do get overwritten. In order to facilitate the writing of valid adjoint code we wish to make it appear that they do not. The first thing is to determine how many times they get overwritten. We can keep track of this by adding another array.

```
 cltudecomp clu2(mmin,mmax);

// LU decomp wihout partial pivoting
cltudecomp ludecomp(const dmatrix & M)
{
  int mmin=M.indexmin();
  int mmax=M.indexmax();
  cltudecomp clu(mmin,mmax);
  cltudecomp clu2(mmin,mmax);


  // get upper and lower parts of LU
```

```
dmatrix & alpha = clu.get_L();
dmatrix & gamma = clu.get_U(); // gamma is the transpose of beta
dmatrix & alpha2 = clu2.get_L();
// copy M into alpha and gamma
for (int i=mmin;i<=mmax;i++)
{
  for (int j=mmin;j<=mmax;j++)
  {
    clu(i,j)=M(i,j);
    clu2(i,j)=1.0;
  }
}
for (int j=mmin;j<=mmax;j++)
{
  int i;
  for (i=mmin+1;i<j;i++)
  {
    // using subvector here
    clu(i,j)-=alpha(i)(mmin,i-1)*gamma(j)(mmin,i-1);
    clu2(i,j)+=1.0;
  }
  for (i=j;i<=mmax;i++)
  {
    // using subvector here
```

```
        if (j>1)
        {
          clu(i,j)-=alpha(i)(mmin,j-1)*gamma(j)(mmin,j-1);
          clu2(i,j)+=1.0;
        }
      }
      if (j!=mmax)
      {
        double z= 1.0/gamma(j,j);
        for (i=j+1;i<=mmax;i++)
        {
          alpha(i,j)*=z;
          alpha2(i,j)+=1;
        }
      }
    }
    cout << alpha2 << endl;
    return clu;
  }
```

It is evident that alpha gets written to at most three times, while gamma gets written to at most two times. We modify the class cltudecomp to produce the class `cltudecomp_for_adjoint` where the matrices L and U are now three dimensional arrays to accommodate the overwriting.

```
class cltudecomp_for_adjoint
{
  dmatrix_for_adjoint L;
  dmatrix_for_adjoint U;
  imatrix iL;
  imatrix iU;
  ivector indx;
  double sign;
public:
  void initialize(void)
  {
    indx.initialize();
    sign=0;
    L.initialize();
    U.initialize();
    for (int i=L.indexmin();i<=L.indexmax();i++)
    {
      L(i,i)=1.0;
    }
  }

  cltudecomp_for_adjoint(int l,int u,int n,int m) : indx(l,u)
  {
    ivector iv(l+1,u);
    iv.fill_seqadd(l,1);
    L.allocate(l+1,u,l,iv,1,n);
    ivector iv1(l,u);
    iv1.fill_seqadd(l,1);
    U.allocate(l,u,l,iv1,1,m);
  }
  dmatrix & get_L(){ return L;}
  dmatrix & get_U(){ return U;}
  ivector & get_index(){ return indx;}
  double & get_sign(){ return sign;}
  // overload () (int,int) to look like Numerical Recipes
  double & operator() (int i,int j){ if (i>j)
                                       return L(i,j);
                                     else
                                       return U(j,i);
                                   }
};
```