Xamoeba_xpowell_probit.c

```
/**/
/**/
/* xamoeba_xpowell.c  -- C code is from NUMERICAL RECIPES IN C --
 *                       Additional code written by Keith Poole
 *                       September - October 2009
 *
   Amoeba -- Uses Nedler-Mead downhill simplex method to find a minimum/maximum
   of a multi-dimensional function
                  |->amotry->func
Main -> amoeba --|
                  |->func

                  Amoeba initializes the simplex and calls func for each
                  point in the simplex.  The simplex starts at the usual
                  triangluar definition plus the origin -- ndim +1.

                  func is the user defined function being
                  minimized/maximized;

                  Amotry -- (from description in NUMERICAL RECIPES)
                  Extrapolates by a factor, fac, through the face of the
                  simplex across from a high point, tries, and replaces
                  the high point if the new point is better. */
/* Powell Routine:
   Uses Powell's Quadratically Convergent Method to find a minimum/maximum
   of a multi-dimensional function

                  |->func
Main -> powell --|          | ->mnbrak|
                  |->linmin->|         |
                             |->brent->f1dim->func

                  Powell minimizes a function of n variables.  The
                  starting values are in the vector p[1:n] and an n by n
                  matrix of directions -- normally a simplex.

                  func is the user defined function being
                  minimized/maximized;

                  linmin -- Finds the minimum on a line joining p and xi.

                     mnbrak -- Used by linmin.  It finds three points that bracket the
                  minimum.

                     brent -- Given the 3 points from mnbrak, it finds the
                                       minimum

                                       */
/*
 * NUMERICAL DERIVATIVES

   Main -> dfridr -- This computes the first derivatives using simple
                     (f(x+h) - f(x-h))/2h
        -> dfridr2nd -- This computes the 2nd derivatives
                     df/dxdy = (f(x+h,y+h) - f(x+h,y-h) - f(x-h,y+h) + f(x-h,y-h))/(4*h*h)
                        d2f/dx2 = ( f(x+2h) + f(x-2h) - 2f(x) )/4h^2
```

```
*******************************************************************************

Probit estimates                                  Number of obs   =        432
                                                  LR chi2(7)      =     338.29
                                                  Prob > chi2     =     0.0000
Log likelihood = -130.12789                       Pseudo R2       =     0.5652

------------------------------------------------------------------------------
           y |      Coef.   Std. Err.      z    P>|z|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
     black00 |  -.0285973   .0097913    -2.92   0.003    -.0477878   -.0094067
       south |   .7695862   .2545783     3.02   0.003     .2706219    1.268551
  hispanic00 |  -.0089458   .0069163    -1.29   0.196    -.0225015    .0046099
      income |  -.0241489   .0126394    -1.91   0.056    -.0489217    .0006239
     owner00 |   .0235461   .0143687     1.64   0.101    -.0046161    .0517083
     dwnom1n |   2.761974   .2619813    10.54   0.000       2.2485    3.275448
     dwnom2n |   1.136417   .2339829     4.86   0.000     .6778186    1.595015
       _cons |  -.9697998   1.077738    -0.90   0.368    -3.082127    1.142528
------------------------------------------------------------------------------
*/

#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <R_ext/Lapack.h>
#include <R_ext/BLAS.h>
//
#define NDIM 8
#define NMAX 10000
#define ITMAX 2000
#define nrowX 432
#define ncolX 8
#define FTOL 0.000000001
#define TOL 0.0002
#define GOLD 1.618034
#define GLIMIT 100.0
#define TINY 1.0e-10
#define CON 1.4
#define BIG 1.0e30
#define NTAB 10
#define SAFE 2.0
#define SHFT(a,b,c,d) (a)=(b);(b)=(c);(c)=(d);
#define SIGN(a,b) ((b) >= 0.0 ? fabs(a) : -fabs(a))
static double maxarg1,maxarg2;
#define FMAX(a,b) (maxarg1=(a),maxarg2=(b),(maxarg1) > (maxarg2) ? (maxarg1) : (maxarg2))
#define CGOLD 0.3819660

typedef struct {
         int counter;
         double xminimum[NDIM];
         double loglikelihood;
} kpsorter;

int structcomparison(const void *v1, const void *v2);

/* Driver for routine powell */

void powell(double pp[], double **xi, int n, double ftol, int *iter, double *fret,
             double (*func)(double []));
void linmin(double pp[], double xi[], int n, double *fret, double (*func)(double []));
void mnbrak(double *ax, double *bx, double *cx, double *fa, double *fb, double *fc,
             double (*func)(double));
double brent(double ax, double bx, double cx, double (*f)(double), double tol,
             double *xmin);
double f1dim(double x);
```

```
void amoeba(double **p, double y[], int ndim, double ftol,
             double (*funk)(double []), int *nfunk);
double amotry(double **p, double y[], double psum[], int ndim,
               double (*funk)(double []), int ihi, double fac);
double dfridr(double (*func)(double []), double x[], double h, double *err, int iiii);
double dfridr2nd(double (*func)(double []), double x[], double h, double *err, int iiii, int
jjjj);
void xsvd(int kpnp, int kpnq, double *, double *, double *, double *);
double keithrules(double x[]);

double func(double x[])
{
        return keithrules(x);
}
int ncom;
double *pcom,*xicom,(*nrfunc)(double []);
static double *Y,*X;
FILE *fp;
FILE *jp;
FILE *kp;

int main(void)
{
         static double *pp;
         int i,info,iter,j,iiii,jjjj,ntrials;
         int nfunc,ndim=NDIM;
         double h,dx,dx2nd,err;
         double fret,**xi;
         double *x,*y,*ppjunk,**p,*firstderv,*secondderv;
         double *XXinv, *xxinvdiag;
         double *u, *lambda, *vt;
         double time1, time2, timedif;
         int *ipiv;
//
         X = (double *) malloc (nrowX*ncolX*sizeof(double));
         Y = (double *) malloc (nrowX*sizeof(double));
         x = (double *) malloc ((NDIM+1)*sizeof(double));
         firstderv = (double *) malloc ((NDIM)*sizeof(double));
         secondderv = (double *) malloc ((NDIM)*(NDIM)*sizeof(double));
         y = (double *) malloc ((NDIM+1+1)*sizeof(double));
         XXinv = (double *) malloc (NDIM*NDIM*sizeof(double));
         ipiv = (int *) malloc (NDIM*sizeof(int));
         xxinvdiag = (double *) malloc (NDIM*sizeof(double));
         u      = (double *) malloc (NDIM*NDIM*sizeof(double));
         lambda = (double *) malloc (NDIM*NDIM*sizeof(double));
         vt     = (double *) malloc (NDIM*NDIM*sizeof(double));
/* Dynamic Allocation of a Matrix: */
/* First, allocate pointers to rows */
/* Second, allocate rows and set pointers to them -- note that p[0] is
 * a pointer to p[0][0:n] -- that is a row -- so the syntax below is
 * p[0][0:nrow*ncol] */
/* Third, form the matrix from step two -- the pointer to the kth row,
 * p[k] is set equal to the memory
 * location of the number of nrows-1 times the number of columns plus
 * the memory location of p[0] */
         p= (double **) malloc((NDIM+1+1)*sizeof(double*));
         p[0]=(double *) malloc((NDIM+1)*(NDIM+1+1)*sizeof(double));
         for (i=0;i<NDIM+1+1;i++)p[i]=p[0]+i*(NDIM+1);
//
         /* Dynamically allocate memory to big data structure
          * ntrials is equal to the number of solutions from Amoeba --
          * NDIM+1 plus the number of solutions from Powell -- in this
          * case NDIM+1+10
          */
         ntrials=NDIM+1+NDIM+1+10;
         kpsorter *recordset;
         recordset = (kpsorter *)malloc(ntrials * sizeof(kpsorter));
//
```

```
/* clock() is part of time.h -- returns the implementation's
 * best approximationto the processor time elapsed since the
 * program was invoke, divide by CLOCKS_PER_SEC to get the time
 * in seconds           */
        time1 = (double) clock();           /* get initial time */
        time1 = time1 / CLOCKS_PER_SEC;     /*    in seconds    */

        srand(122);
//
        jp = fopen("c:/docs_c_summer_course/data_optim.txt","w");
        kp = fopen("c:/docs_c_summer_course/data_optim_2.txt","w");
        if((fp = fopen("c:/docs_c_summer_course/bush2000.txt","r"))==NULL)
        {
                printf("\nUnable to open file BUSH2000.TXT: %s\n", strerror(errno));
                exit(EXIT_FAILURE);
        }
        else {

                fprintf(jp," Y and X = \n");
                for(i=0;i<nrowX;i++)
                {
                        fscanf(fp,"%lf",&Y[i]);
                        for(j=0;j<ncolX;j++)
                        {
                                fscanf(fp,"%lf",&X[i+j*nrowX]);
                        }
                        fprintf(jp,"%10d %12.6f", i,Y[i]);
                        for(j=0;j<ncolX;j++)
                        {
                                fprintf(jp,"%12.6f",X[i+j*nrowX]);
                        }
                        fprintf(jp,"\n");
                }
        }
//
//  AMOEBA MINIMIZATION IS DONE FIRST
//
/* if i == (j+1) is true then the value of x[j] = 1.0, otherwise =0.0 */
        for (i=1;i<=NDIM+1;i++) {
                for (j=1;j<=NDIM;j++)
                        x[j]=p[i][j]=(i == (j+1) ? 1.0 : 0.0);
                y[i]=func(x);
        }
        amoeba(p,y,ndim,FTOL,func,&nfunc);
        printf("\nNumber of function evaluations: %3d\n",nfunc);
        fprintf(jp,"\nNumber of function evaluations: %3d\n",nfunc);
        printf("Vertices of final 3-d simplex and\n");
        fprintf(jp,"Vertices of final 3-d simplex and\n");
        printf("function values at the vertices:\n\n");
        fprintf(jp,"function values at the vertices:\n\n");
        printf("%3s %10s %12s %12s %14s\n\n",
               "i","x[i]","y[i]","z[i]","function");
        for (i=1;i<=NDIM+1;i++) {
                printf("%3d ",i);
                fprintf(jp,"%3d ",i);
                for (j=1;j<=NDIM;j++) {
                        fprintf(jp,"%12.6f ",p[i][j]);
//
// store solution in x[]
//
                     x[j]=p[i][j];
                        recordset[i-1].xminimum[j-1]=x[j];
                }
                printf("%12.6f\n",y[i]);
                fprintf(jp,"%12.6f\n",y[i]);
                recordset[i-1].loglikelihood=y[i];
                recordset[i-1].counter=i;
        }
//
```

```
//  POWELL MINIMIZATION IS DONE SECOND -- first, the NDIM+1 (ndim + 1)
//  solutions are checked and, second, 10 random starts are checked
//
          for(iiii=1;iiii <= NDIM+1+10; iiii++)
          {
          pp = (double *) malloc ((NDIM+1)*sizeof(double));
          ppjunk = (double *) malloc ((NDIM+1)*sizeof(double));
          /*
          Get Uniform (-1, +1) Numbers for starts
           */
          pp[0]=0.0;
          for(i=1;i < NDIM+1;i++)
          {
                    if(iiii <= NDIM+1) pp[i] = p[iiii][i];
                    if(iiii > NDIM+1) pp[i] = 2.0*( (double)rand() / ((double)(RAND_MAX)+1))-1.0;
                    ppjunk[i] = pp[i];
          }
//
/* Dynamic Allocation of a Matrix: */
/* First, allocate pointers to rows */
/* Second, allocate rows and set pointers to them -- note that p[0] is
 * a pointer to p[0][0:n] -- that is a row -- so the syntax below is
 * p[0][0:nrow*ncol] */
/* Third, form the matrix from step two -- the pointer to the kth row,
 * p[k] is set equal to the memory
 * location of the number of nrows-1 times the number of columns plus
 * the memory location of p[0] */
          xi= (double **) malloc((NDIM+1)*sizeof(double*));
          xi[0]=(double *) malloc((NDIM+1)*(NDIM+1)*sizeof(double));
          for (i=0;i<NDIM+1;i++)xi[i]=xi[0]+i*(NDIM+1);
//
          for (i=1;i<=NDIM;i++)
                    for (j=1;j<=NDIM;j++)
                              xi[i][j]=(i == j ? 1.0 : 0.0);
/* if i == j is true then the value of x[i][j] = 1.0, otherwise =0.0 */
          powell(pp,xi,NDIM,FTOL,&iter,&fret,func);
          fprintf(jp,"Iterations: %3d\n\n",iter);
//        printf("Iterations: %3d\n\n",iter);
          fprintf(jp,"Minimum found at: \n");
//        printf("Minimum found at: \n");
          for (i=1;i<=NDIM;i++) {
                    fprintf(jp,"%5d %12.6f %12.6f\n",i, pp[i], ppjunk[i]);
//                  printf("%5d %12.6f %12.6f\n",i, pp[i], ppjunk[i]);
                    recordset[iiii-1+NDIM+1].xminimum[i-1]=pp[i];
          }
          fprintf(jp,"\n\n%5d Minimum function value = %12.6f \n\n",iiii,fret);
          printf("%5d Minimum function value = %12.6f \n",iiii,fret);
          recordset[iiii-1+NDIM+1].loglikelihood=fret;
          recordset[iiii-1+NDIM+1].counter=iiii+NDIM+1;
          free(xi);
          free(pp);
          free(ppjunk);
}
          qsort(recordset, ntrials, sizeof(kpsorter), structcomparison);
          for(i=0;i<ntrials;i++)
          {
//
// Pass Best Solution to Numerical Derivatives
//
                    if(i == 0){
                              for(j=1;j<=NDIM;j++)
                    {
                                   x[j]=recordset[i].xminimum[j-1];
                             }
                    }
                    fprintf(kp,"%5d %5d  %14.6f", i,
recordset[i].counter,recordset[i].loglikelihood);
                    for(j=0;j<NDIM;j++)
                    {
```

```
                        fprintf(kp,"%12.6f",recordset[i].xminimum[j]);
                }
                fprintf(kp,"\n");
        }
        fprintf(kp,"\n\n");
        h = 0.01;
// 1st derivatives
        for(iiii=1;iiii<=NDIM;iiii++){
                h = 0.01;
                dx=dfridr(func,x,h,&err,iiii);
                fprintf(jp,"dfridr    = %5d %5d %12.6f %12.6f %12.6f\n",iiii,iiii,dx,h,err);
                firstderv[iiii-1]=dx;
// 2nd derivatives
                for(jjjj=1;jjjj<=NDIM;jjjj++){
                        h = 0.01;
                        dx2nd=dfridr2nd(func,x,h,&err,iiii,jjjj);
                        fprintf(jp,"dfrid2nd  = %5d %5d %12.6f %12.6f
%12.6f\n",iiii,jjjj,dx2nd,h,err);
                        secondderv[iiii-1 + (jjjj-1)*NDIM]=dx2nd;
                }
        }
        fprintf(jp,"\n\nFirst Derivatives \n\n");
        for(iiii=0;iiii<NDIM;iiii++){
                fprintf(jp," %5d %12.6f %12.6f\n",iiii,firstderv[iiii],x[iiii+1]);
        }
        fprintf(jp,"\n\nSecond Derivatives \n\n");
        for(iiii=0;iiii<NDIM;iiii++){
                for(jjjj=0;jjjj<NDIM;jjjj++){
                        fprintf(jp,"%14.6f",secondderv[iiii + (jjjj)*NDIM]);
                }
                fprintf(jp,"\n");
        }
/* Call Singular Value Decomposition Routine to look at the rank
 * the Hessian*/
/*  Clock the SVD Routine*/

        time2 = (double) clock();            /* get initial time */
        time2 = time2 / CLOCKS_PER_SEC;      /*    in seconds    */
        xsvd(NDIM,NDIM,secondderv,u,lambda,vt);
        printf("Singular Values\n");
        for(j=0;j<NDIM;j++)
        {
                printf("%5d %16.6f\n",j,lambda[j]);
        }
        timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time2;
        printf("SVD took %12.3f seconds\n", timedif);
        fprintf(jp,"SVD took %12.3f seconds\n", timedif);
        fprintf(kp,"SVD took %12.3f seconds\n", timedif);
/*
   Check to make certain the Hessian is non-singular
*/
        if(lambda[NDIM-1]<=TOL){
                fprintf(jp," Error: Hessian Matrix is Singular!");
                fprintf(kp," Error: Hessian Matrix is Singular!");
                printf(" Error: Hessian Matrix is Singular!");
        }
       if(lambda[NDIM-1]>TOL){
/* Initialize the Identity matrix */
        for(i=0;i<NDIM;i++)
        {
                for(j=0;j<NDIM;j++)
                {
                        XXinv[i+j*NDIM]=0.0;
                        if(i == j)XXinv[i+j*NDIM]=1.0;
                }
        }
        dgesv_(&ndim,&ndim,secondderv,&ndim,ipiv,XXinv,&ndim,&info);
        fprintf(kp,"\n\nInverse Hessian = \n");
        for(i=0;i<NDIM;i++)
```

```
        {
                for(j=0;j<NDIM;j++)
                {
                        fprintf(kp,"%12.6f",XXinv[i+j*NDIM]);
/* Save Diagonal of Inverse Hessian */
                        if(i == j)xxinvdiag[i] = XXinv[i+j*NDIM];
                }
                fprintf(kp,"\n");
        }
        fprintf(kp,"\n\nFirst Derivatives, Coefficients, and Standard Errors \n\n");
        for(iiii=0;iiii<NDIM;iiii++){
                fprintf(kp," %5d %12.6f %12.6f
%12.6f\n",iiii,firstderv[iiii],x[iiii+1],sqrt(xxinvdiag[iiii]));
        }
        }
        free(Y);
        free(X);
        free(x);
        free(firstderv);
        free(secondderv);
        free(y);
        free(XXinv);
        free(ipiv);
        free(xxinvdiag);
        free(u);
        free(lambda);
        free(vt);
        free(p);
//
        timedif = ( ((double) clock()) / CLOCKS_PER_SEC) - time1;
        printf("The total elapsed time of the program is %12.3f seconds\n", timedif);
        fprintf(jp,"\nThe total elapsed time of the program is %12.3f seconds\n", timedif);
        fprintf(kp,"\nThe total elapsed time of the program is %12.3f seconds\n", timedif);

        fclose(jp);
        fclose(fp);
        return 0;
}
```

Etc. etc. etc.

```
double keithrules(double x[])
{
        int i, j, k1, k0, l1, l0;
        double sum=0;
        double phi=0.0;
        double xphi=0.0;
        double sumsquared=0;
        k1=0;
        k0=0;
        for(i=0;i<nrowX;i++)
        {
                sum=0.0;
                for(j=0;j<ncolX;j++)
                {
                        sum=sum+X[i+j*nrowX]*x[j+1];
                }
                phi = (erf(fabs(sum)/sqrt(2.0)))/2.0 + 0.5;
                xphi = phi;
                if(sum < 0.0)phi=1.0-xphi;
                xphi = phi;
```

```
                if(xphi > 0.99999999)phi=0.9999999;
                if(xphi < 0.00000001)phi=0.0000001;
// Voted for Bush
                if(Y[i] >= 50.0){
                        l1=1;
                        l0=0;
                        k1=k1+1;
                        sumsquared = sumsquared + log(phi);
                }
// Voted for Gore
                if(Y[i] < 50.0){
                        l1=0;
                        l0=1;
                        k0=k0+1;
                        sumsquared = sumsquared + log(1.0 - phi);
                }
        }
//      printf("%lf\n",-sumsquared);
        return -sumsquared;

}
```