References

Cane, H. V. The evolution of interplanetary shocks. Journal of Geophysical Research **90** (1985), 191-197.

Cane, H. V., McGuire, R. E., and von Rosenvinge, T. T. Two classes of solar energetic particle events associated with impulsive and long-duration soft X-ray flares. Astrophysical Journal **301** (1986), 448-459.

Cliver, E. W., and others. Solar flare nuclear gamma-rays and interplanetary proton events. Astrophysical Journal **343** (1989), 953-970.

Dröge, W., Ruffolo, D., and Klecker, B. Observation of electrons from the decay of solar flare neutrons. Astrophysical Journal **464** (1996), L87-L90.

Earl, J. A. The effect of adiabatic focusing upon charged-particle propagation in random magnetic fields. Astrophysical Journal **205** (1976a), 900-919.

Earl, J. A. Nondiffusive propagation of cosmic rays in the solar system and in extragalactic radio sources. Astrophysical Journal **206** (1976b), 301-311.

Earl, J. A. Charged particle transport calculations on the massively parallel processor. Proc. 20th Int. Cosmic Ray Conf. **10** (1987), 4.1-12.

Earl, J. A., and others. Comparison of three numerical treatments of charged particles transport. Astrophysical Journal **454** (1995), 749-761.

Evenson, P. Particle accelelation on the Sun. Proc. 21st Int. Cosmic Ray Conf. **11** (1990), 152.

Hovestadt, D., and others. Ionic charge state distribution of helium, cabon, oxygen, and iron in an energetic storm paticle enhancement. Astrophysical Journal **258** (1982), L57-L62.

Jokipii, J. R. Cosmic ray propagation. I. Charged particles in a random magnetic field. Astrophysical Journal **146** (1966), 480-487.

Kallenrode, M.-B., and Wibberenz, G. Particle injection following solar flares on 1980 May 28 and June 8: evidence for different injection time histories in impulsive and gradual events? Astrophysical Journal **376** (1991), 787-796.

Kallenrode, M.-B., Wibberenz, G., and Hucke, S. Propagation conditions of relativistic electrons in the inner heliosphere. Astrophysical Journal **394** (1992), 351-356.

Lee, M. A. Coupled hydromagnetic wave excitation and ion acceleration at interplanetary traveling shocks. Journal of Geophysical Research **88** (1983), 6109-6119.

Lee, M. A., and Ryan, J. M. Time-dependent coronal shock acceleration of energetic solar flare particles. Astrophysical Journal **303** (1986), 829-842.

Ma Sung, L. S., and Earl, J. A. Interplanetary propagation of flare-associated energetic particles. Astrophysical Journal **222** (1978), 1080-1096.

Ng, C. K., and Wong, K.-Y. Solar particle propagation under the influence of pitch-angle diffusion and collimation in the interplanetary magnetic field. Proc. 16th Internat. Cosmic Ray Conf **5** (1979), 252-258.

Pallavicini, R., Serio, S., and Vaiana G. S. A survey of soft X-ray limb flare images: the relation between their structure in the corona and other physical parameter. Astrophysical Journal **216** (1977), 108-122.

Palmer, I. D. Transport coefficients of low-energy cosmic rays in interplanetary space. Reviews of Geophysics and Space Physics **20** (1982), 335-351.

Press, W. H., and others. Direction set (Powell's) methods in multidimensions. Numerical Recipes in C (1988), 309-317.

Reames, D. V. Wave generation in the transport of particles from large solar flares. Astrophysical Journal **342** (1989), L51-L53.

Reames, D. V. Acceleration of energetic particles by shock waves from large solar flares. Astrophysical Journal **358** (1990), L63-L67.

Reames, D. V., and Stone, R. G. The identification of solar ^{3}He-rich events and the study of particle acceleration of the Sun. Astrophysical Journal **308** (1986), 902-911.

Reid, G. C. A diffusive model for the initial phase of a solar proton event. Journal of Geophysical Research **69** (1964), 2659-2667.

Ruffolo, D. The interplanetary transport of decay protons from solar flare neutrons. Astrophysical Journal **382** (1991), 688-698.

Ruffolo, D. Effect of adiabatic deceleration on the focused transport of solar cosmic rays. Astrophysical Journal **442** (1995), 861-874.

Ruffolo, D. and Khumlumlert T. Formation, propagation, and decay of coherent pulses of solar cosmic rays. Geophysical Research Letters **22** (1995), 2073-2076.

Appendix A
Program for Finding Coefficients (A, B, t_0)

Program "findcoef.c"

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>
#include <alloc.h>
#include <io.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include <powell.c>

#define     NDIM    3
#define     IMAX    400
#define     SMAX    400
#define     ALLMAX  800
#define     FTOL    1.0e-2

double      C, Tf,A,B,chisq;
int         im,ims,ani;             /* im,ims uses in makeff */
double      sig[ALLMAX], x[ALLMAX], data[ALLMAX], afunc[ALLMAX],
            st[ALLMAX], et[ALLMAX], time[ALLMAX], sim[ALLMAX];
```

```
main(int argc,char *argv[])
{
   double   ftol, fret, *p, **xi;
   double   tempd;
   int      i, j, tempi, iter;
   FILE     *fi, *fp_f, *fp_d, *fp_s, *fp_k;
   double   *yy, tempr;
   int      ndim=NDIM, tempn;
   double   func(), **dmatrix(), *dvector();
   void     powell(), free_dmatrix(), free_dvector(), makefileff();

   printf("\nThis program will find coefficient A,B,C,t0 and chisq");
   printf("\nUses 'counts.prn' for int, 'simulate.dat' for ani");
   printf("\nand 'skeleton.dat'.");
   printf("\nTo use this file type >
          FINDCOEF fitdata fitfunc [fitdata fitfunc]");
   printf("\nRunning...");

   if(argc>3) ani=1; else ani=0;
   fp_d = fopen("in.cof","r");
   fscanf(fp_d,"%lf",&A);
   fscanf(fp_d,"%lf",&B);
   fscanf(fp_d,"%lf",&C);
   fscanf(fp_d,"%lf",&Tf);
   fclose(fp_d);
```

```
fp_d = fopen(argv[1],"r");
for (i=0;i<=IMAX;i++){
  if (fscanf(fp_d,"%lf", &x[i])!=1) break;
  fscanf(fp_d,"%lf", &data[i]);
  fscanf(fp_d,"%lf", &sig[i]);
}
fclose(fp_d);
im = i-1<IMAX ? i-1 : IMAX;
fp_f = fopen(argv[2],"r");
for (i=0;i<=IMAX;i++){
  if (fscanf(fp_d,"%lf", &x[i])!=1) break;
  fscanf(fp_f,"%lf", &afunc[i]);
}
fclose(fp_f);
if (i-1<im) im=i-1;
if (ani==1) {
  fp_d = fopen(argv[3],"r");
  for(i=0;i<=IMAX;i++){
    if (fscanf(fp_d,"%lf", &x[IMAX+i])!=1) break;
    fscanf(fp_d,"%lf", &data[IMAX+i]);
    fscanf(fp_d,"%lf", &sig[IMAX+i]);
  }
  fclose(fp_d);
  if (i-1<im) im=i-1;
  fp_f = fopen(argv[4],"r");
  for (i=0;i<=IMAX;i++){
    if (fscanf(fp_d,"%lf", &x[IMAX+i])!=1) break;
```

```
      fscanf(fp_f,"%lf", &afunc[IMAX+i]);
    }
    fclose(fp_f);
    if (i-1<im) im=i-1;
  }
  fp_k = fopen("skeleton.dat","r");  /* uses in makeff */
  for (i=0;i<=IMAX;i++) {
     if (fscanf(fp_k,"%lf",&st[i])!=1) break;
     fscanf(fp_k,"%lf",&et[i]);
  }
  fclose(fp_k);
  if (i-1<im) im=i-1;
  fp_k = fopen("skeleton.dat","r");  /* uses in makeff */
  for (i=0;i<=IMAX;i++) {
     if (fscanf(fp_k,"%lf",&st[IMAX+i])!=1) break;
     fscanf(fp_k,"%lf",&et[IMAX+i]);
  }
  fclose(fp_k);
  if (i-1<im) im=i-1;
  fp_s = fopen("counts.prn","r");
  for (i=0;i<=SMAX;i++) {
    if (fscanf(fp_s,"%lf",&time[i])!=1) break;
    fscanf(fp_s,"%lf",&sim[i]);
  }
  fclose(fp_s);
  ims = i-1<SMAX ? i-1 : SMAX;
  if (ani==1) {
```

```
    fprintf(fp_d,"%lf\n",C);
    fprintf(fp_d,"%lf\n",p[3]);
    fprintf(fp_d,"%lf\n",chisq);
    fclose(fp_d);
    makefileff(p);

    p  = dvector(1,ndim);
    xi = dmatrix(1,ndim,1,ndim);
    free_dvector(p,1,ndim);
    free_dmatrix(xi,1,ndim,1,ndim);
    printf("\nSuccessful...");
    return(0);
}

/* function.c -- June 18th, 1993
   This is a 2-D function used to test the minimization program.
*/

double func(q)
double   *q;
{
   double  sum, sum1, sum2;
   int     i;
   void    makeff();
   makeff(q);
   sum1=sum2=0;
   for(i=1;i<=im;i++) sum1 += data[i]*afunc[i]/(sig[i]*sig[i]);
```

```
    for(i=1;i<=im;i++) sum2 += afunc[i]*afunc[i]/(sig[i]*sig[i]);
    if (ani==1) {
      for(i=1;i<=im;i++)
        sum1 += data[IMAX+i]*afunc[IMAX+i]/(sig[IMAX+i]*sig[IMAX+i]);
      for(i=1;i<=im;i++)
        sum2 += afunc[IMAX+i]*afunc[IMAX+i]/(sig[IMAX+i]*sig[IMAX+i]);
    }
    C=sum1/sum2;

    sum=0;
    for(i=1;i<=im;i++)
      sum += (data[i]-C*afunc[i])*(data[i]-C*afunc[i])
             /(sig[i]*sig[i]);
    if (ani==1) for(i=1;i<=im;i++)
      sum += (data[IMAX+i]-C*afunc[IMAX+i])*
             (data[IMAX+i]-C*afunc[IMAX+i])/[IMAX+i]*sig[IMAX+i]);
    chisq=sum/(im-NDIM);
    printf("A=%10.7lf,B=%10.7lf,t0=%10.7lf,chisq=%10.7lf\n",
           q[1],q[2],q[3],chisq);
    return(sum);
}

#define   DT      0.1
#define   AC      0.1

void makeff(q)
double  *q;
```

```
{
   double   arg, c[ALLMAX], counts, error, frac;
   double   tprime, value, width, delta;
   double   Cff, t0, timecp;
   int      i, j, n, nn;

   if(q[1]<0) q[1]=-q[1];
   if(q[2]<0) q[2]=-q[2];
   A = q[1]; B = q[2]; t0 = q[3];
   Cff = 1;
   delta = (-log(AC)+2*sqrt(A/B))*B;
   nn =  delta/DT;
   if (nn>2000) nn=2000;
   for (i=0;i<=im;i++) c[i] = 0.0;

  for (j=1;j<=ims;j++) {
     width = time[j] - time[j-1];
     for (n=1;n<=nn;n++) {
tprime = n*DT;
arg = -A/tprime-tprime/B;
if (-arg < 20) { value  = width * DT * Cff/tprime * exp(arg);
  } else value = 0;
   /* Find out which bin s+tprime+offset-DT/2 is in.
      If ...+DT/2 in same bin, add all to c[i].
      Otherwise, split between neighboring bins.
   */
timecp = time[j]+tprime+t0-DT/2.0;
```

```
    if (timecp < et[im]) {
       for (i=0;i<=im && st[i] <= timecp;i++);
       if (i > 0) {
          if (timecp < et[i-1]) { c[i-1] += sim[j-1] * value;
          } else {
     if (timecp < et[i-1]) {
        frac = (et[i-1] - timecp)/DT;
        c[i-1] += frac * sim[j-1] * value;
     }
     if (timecp >= st[i]) {
        frac = (timecp - st[i])/DT;
        c[i] += frac * sim[j-1] * value;
     }
          }
       } else {
          if (timecp >= st[0]) {
                  c[0] += (timecp - st[0])/DT * sim[j-1] * value;
          }
       }
    }
        }
     }
     for (i=0;i<=im;i++) afunc[i]=c[i]/(et[i]-st[i]);

/******   procedure for ani   ******/
    if (ani==1) {
      for (i=0;i<=im;i++) c[IMAX+i] = 0.0;
```

```
  for (j=1;j<=ims;j++) {
     width = time[SMAX+j] - time[SMAX+j-1];
     for (n=1;n<=nn;n++) {
tprime = n*DT;
arg = -A/tprime-tprime/B;
if (-arg < 20) { value  = width * DT * Cff/tprime * exp(arg);
  } else value = 0;
   /* Find out which bin s+tprime+offset-DT/2 is in.
      If ...+DT/2 in same bin, add all to c[i].
      Otherwise, split between neighboring bins.
   */
timecp = time[SMAX+j]+tprime+t0-DT/2.0;
if (timecp < et[IMAX+im]) {
   for (i=0;i<=im && st[IMAX+i] <= timecp;i++);
   if (i > 0) {
      if (timecp < et[IMAX+i-1]) {
 c[IMAX+i-1] += sim[SMAX+j-1] * value;
      } else {
 if (timecp < et[IMAX+i-1]) {
    frac = (et[IMAX+i-1] - timecp)/DT;
    c[IMAX+i-1] += frac * sim[SMAX+j-1] * value;
}
if (timecp >= st[IMAX+i]) {
   frac = (timecp - st[IMAX+i])/DT;
   c[IMAX+i] += frac * sim[SMAX+j-1] * value;
}
     }
```

```
      } else {
         if (timecp >= st[IMAX]) {
    c[IMAX] += (timecp-st[IMAX+0])/DT*sim[SMAX+j-1]*value;
         }
      }
   }
       }
    }
    for (i=0;i<=im;i++) afunc[IMAX+i]=c[IMAX+i]/(et[IMAX+i]-st[IMAX+i]);
  }
}

void makefileff(q)
double  *q;
{
   FILE     *fp_f;
   double   arg, c[ALLMAX], counts, error, frac;
   double   tprime, value, width, delta, t0, timecp;
   int      i, j, n, nn;

   if(q[1]<0) q[1]=-q[1];
   if(q[2]<0) q[2]=-q[2];
   A = q[1]; B = q[2]; t0 = q[3];
   delta = (-log(AC)+2*sqrt(A/B))*B;
   nn    = delta/DT;
   if (nn>2000) nn=2000;
   for (i=0;i<=im;i++) c[i] = 0.0;
```

```
   for (j=1;j<=ims;j++) {
      width = time[j] - time[j-1];
      for (n=1;n<=nn;n++) {
tprime = n*DT;
arg = -A/tprime-tprime/B;
if (arg<20) { value  = width * DT * C/tprime * exp(arg);
} else value = 0;
   /* Find out which bin s+tprime+offset-DT/2 is in.
      If ...+DT/2 in same bin, add all to c[i].
      Otherwise, split between neighboring bins.
   */
timecp = time[j]+tprime+t0-DT/2.0;
if (timecp < et[im]) {
   for (i=0;i<=im && st[i] <= timecp;i++);
   if (i > 0) {
      if (timecp < et[i-1]) {
 c[i-1] += sim[j-1] * value;
      } else {
 if (timecp < et[i-1]) {
    frac = (et[i-1] - timecp)/DT;
    c[i-1] += frac * sim[j-1] * value;
 }
 if (timecp >= st[i]) {
    frac = (timecp - st[i])/DT;
    c[i] += frac * sim[j-1] * value;
 }
```

```
        }
      } else {
        if (timecp >= st[0]) {
   c[0] += (timecp - st[0])/DT * sim[j-1] * value;
        }
      }
  }
      }
    }
    fp_f = fopen("fitfunc.xxi","w");
    for (i=0;i<=im;i++) fprintf(fp_f,"%12lf %12lf\n",
       (st[i]+et[i])/2.0,c[i]/(et[i]-st[i]));
    fclose(fp_f);

    /******  procedure  for ani  ****/
  if (ani==1) {
    for (i=0;i<=im;i++) c[IMAX+i] = 0.0;
    for (j=1;j<=ims;j++) {
       width = time[SMAX+j] - time[SMAX+j-1];
       for (n=1;n<=nn;n++) {
  tprime = n*DT;
  arg = -A/tprime-tprime/B;
  if (arg<20) { value  = width * DT * C/tprime * exp(arg);
  } else value = 0;
     /* Find out which bin s+tprime+offset-DT/2 is in.
        If ...+DT/2 in same bin, add all to c[IMAX+i].
        Otherwise, split between neighboring bins.
```

```
   */
timecp = time[SMAX+j]+tprime+t0-DT/2.0;
if (timecp < et[IMAX+im]) {
   for (i=0;i<=im && st[IMAX+i] <= timecp;i++);
   if (i > 0) {
      if (timecp < et[IMAX+i-1]) {
c[IMAX+i-1] += sim[SMAX+j-1] * value;
      } else {
if (timecp < et[IMAX+i-1]) {
   frac = (et[IMAX+i-1] - timecp)/DT;
   c[IMAX+i-1] += frac * sim[SMAX+j-1] * value;
}
if (timecp >= st[IMAX+i]) {
   frac = (timecp - st[IMAX+i])/DT;
   c[IMAX+i] += frac * sim[SMAX+j-1] * value;
}
      }
  } else {
      if (timecp >= st[IMAX]) {
c[IMAX] += (timecp-st[IMAX+0])/DT*sim[SMAX+j-1]*value;
      }
  }
}
    }
 }
 fp_f = fopen("fitfunc.xxa","w");
 for (i=0;i<=im;i++) fprintf(fp_f,"%12lf %12lf\n",
```

```
        (st[IMAX+i]+et[IMAX+i])/2.0,c[IMAX+i]/(et[IMAX+i]-st[IMAX+i]));
    fclose(fp_f);
  }
}

float *vector(nl,nh)
int nl,nh;
{
float *v;
v=(float *)malloc((unsigned) (nh-nl+1)*sizeof(float));
if (!v) nrerror("allocation failure in vector()");
return v-nl;
}

long *lvector(nl,nh)
long nl,nh;
{
long *v;
v=(long *)malloc((unsigned) (nh-nl+1)*sizeof(long));
if (!v) nrerror("allocation failure in lvector()");
return v-nl;
}

float **matrix(nrl,nrh,ncl,nch)
int nrl,nrh,ncl,nch;
{
int i;
```

```
float **m;
m=(float **) malloc((unsigned) (nrh-nrl+1)*sizeof(float*));
if (!m) nrerror("allocation failure 1 in matrix()");
m -= nrl;
for(i=nrl;i<=nrh;i++) {
m[i]=(float *) malloc((unsigned) (nch-ncl+1)*sizeof(float));
if (!m[i]) nrerror("allocation failure 2 in matrix()");
m[i] -= ncl;
}
return m;
}

int **imatrix(nrl,nrh,ncl,nch)
int nrl,nrh,ncl,nch;
{
int i;
int **m;
m=(int **) malloc((unsigned) (nrh-nrl+1)*sizeof(int*));
if (!m) nrerror("allocation failure 1 in imatrix()");
m -= nrl;
for(i=nrl;i<=nrh;i++) {
m[i]=(int *) malloc((unsigned) (nch-ncl+1)*sizeof(int));
if (!m[i]) nrerror("allocation failure 2 in imatrix()");
m[i] -= ncl;
}
return m;
}
```

```
float **submatrix(a,oldrl,oldrh,oldcl,oldch,newrl,newcl)
float **a;
int oldrl,oldrh,oldcl,oldch,newrl,newcl;
{
int i,j;
float **m;
m=(float **) malloc((unsigned) (oldrh-oldrl+1)*sizeof(float*));
if (!m) nrerror("allocation failure in submatrix()");
m -= newrl;
for(i=oldrl,j=newrl;i<=oldrh;i++,j++) m[j]=a[i]+oldcl-newcl;
return m;
}


void free_vector(v,nl,nh)
float *v;
int nl,nh;
{
free((char*) (v+nl));
}


void free_lvector(v,nl,nh)
long *v,nl,nh;
{
free((char*) (v+nl));
}
```

```
void free_matrix(m,nrl,nrh,ncl,nch)
float **m;
int nrl,nrh,ncl,nch;
{
int i;
for(i=nrh;i>=nrl;i--) free((char*) (m[i]+ncl));
free((char*) (m+nrl));
}

void free_imatrix(m,nrl,nrh,ncl,nch)
int **m;
int nrl,nrh,ncl,nch;
{
int i;
for(i=nrh;i>=nrl;i--) free((char*) (m[i]+ncl));
free((char*) (m+nrl));
}

void free_submatrix(b,nrl,nrh,ncl,nch)
float **b;
int nrl,nrh,ncl,nch;
{
free((char*) (b+nrl));
}

float **convert_matrix(a,nrl,nrh,ncl,nch)
float *a;
```

```
int nrl,nrh,ncl,nch;
{
int i,j,nrow,ncol;
float **m;
nrow=nrh-nrl+1;
ncol=nch-ncl+1;
m = (float **) malloc((unsigned) (nrow)*sizeof(float*));
if (!m) nrerror("allocation failure in convert_matrix()");
m -= nrl;
for(i=0,j=nrl;i<=nrow-1;i++,j++) m[j]=a+ncol*i-ncl;
return m;
}

void free_convert_matrix(b,nrl,nrh,ncl,nch)
float **b;
int nrl,nrh,ncl,nch;
{
free((char*) (b+nrl));
}
```

Program "powell.c"

```
/*  Copied from "Numerical Recipes in C" */
#include   <math.h>
#include   <stdio.h>
#define   BITMAX 10000
#define   CGOLD 0.3819660
```

```
#define   GLIMIT 100.0
#define   GOLD 1.618034
#define   ITMAX 200
#define   MAX(a,b) ((a) > (b) ? (a) : (b))
#define   SIGN(a,b) ((b) > 0.0 ? fabs(a) : -fabs(a))
#define   SHFT(a,b,c,d) (a)=(b); (b)=(c); (c)=(d);
static    double squarg;
#define   SQR(a) (squarg =(a), squarg*squarg)
#define   TINY 1.0e-20
#define   TOL   1.0e-6
#define   ZEPS 1.0e-10

int ncom=0;
double *pcom=0, *xicom=0, (*nrfunc)();

void powell(p,xi,n,ftol,iter,fret,func)
double p[], **xi,ftol,*fret,(*func)();
int n, *iter;

{
   int i,ibig,j;
   double t, fptt, fp, del;
   double *pt, *ptt, *xit, *dvector();
   void linmin(),nrerror(), free_dvector();
   pt=dvector(1,n);
   ptt=dvector(1,n);
   xit=dvector(1,n);
```

```
*fret=(*func)(p);
for (j=1;j<=n;j++)  pt[j]=p[j];
for (*iter=1 ;; (*iter)++) {
    fp=(*fret);
    ibig=0;
    del=0.0;
    for (i=1;i<=n;i++)  {
       for (j=1;j<=n;j++) xit[j]=xi[j][i];
       fptt=(*fret);
       linmin(p,xit,n,fret,func);
 for (j=1;j<=n;j++) printf("\nx[%d] = %17.12lf",j,p[j]);
 printf("\n");
       if (fabs(fptt-(*fret)) >del) {
           del=fabs(fptt-(*fret));
           ibig=i;
       }
    }
    if (2.0*fabs(fp-(*fret)) <= ftol*(fabs(fp)+fabs(*fret))){
        free_dvector(xit,1,n);
        free_dvector(ptt,1,n);
        free_dvector(pt,1,n);
        return;
    }
if (*iter == ITMAX)
 nrerror("Warning!!! too many iterations in POWELL");
   for (j=1;j<=n;j++)  {
       ptt[j]=2.0*p[j]-pt[j];
```

```
            xit[j]=p[j]-pt[j];
            pt[j]=p[j];
        }
        fptt=(*func)(ptt);
        if (fptt < fp) {
          t=2.0*(fp-2.0*(*fret)+fptt)*SQR(fp-(*fret)-del)
            -del*SQR(fp-fptt);
            if (t < 0.0) {
               linmin(p,xit,n,fret,func);
               for (j=1;j<=n;j++)  xi[j][ibig]=xit[j];
             }
        }
    }
}

void linmin(p,xi,n,fret,func)
double p[], xi[], *fret, (*func)();
int n;

{
   int j;
   double xx, xmin, fx,fb,fa, bx,ax;
   double brent(), f1dim(), *dvector();
   void mnbrak(), free_dvector();

   ncom=n;
   pcom=dvector(1,n);
```

```
    xicom=dvector(1,n);
    nrfunc=func;
    printf("1\t");
    for (j=1;j<=n;j++) {
        pcom[j]=p[j];
        xicom[j]=xi[j];
    }
    ax=0.0;
    xx=1.0;
    bx=2.0;
    mnbrak(&ax,&xx,&bx,&fa,&fx,&fb,f1dim);
    *fret=brent(ax,xx,bx,f1dim,TOL,&xmin);
    for ( j=1;j<=n; j++) {
      xi[j] *=xmin;
      p[j] += xi[j];
    }
    free_dvector(xicom,1,n);
    free_dvector(pcom,1,n);
printf("\np[1]=%10.7lf,p[2]=%10.7lf,p[3]=%10.7lf",p[1],p[2],p[3]);
}

double brent(ax,bx,cx,f,tol,xmin)
double ax,bx,cx,tol,*xmin;
double (*f)();
{
    int iter;
    double a,b,d,etemp,fu,fv,fw,fx,p,q,r,tol1,tol2,u,v,w,x,xm;
```

```
double e=0.0;
void nrerror();
printf("b\t");
a=((ax < cx) ? ax : cx);
b=((ax > cx) ? ax : cx);
x=w=v=bx;
fw=fv=fx=(*f)(x);
for (iter=1;iter <=          BITMAX;iter++) {
    xm=0.5*(a+b);
    tol2=2.0*(tol1=tol*fabs(x)+ZEPS);
    if (fabs(x-xm) <= (tol2-0.5*(b-a)))  {
       *xmin=x;
       return fx;
    }
    if (fabs(e) > tol1) {
         r=(x-w)*(fx-fv);
         q=(x-v)*(fx-fw);
         p=(x-v)*q-(x-w)*r;
         q=2.0*(q-r);
         if(q > 0.0) p =  -p;
         q = fabs(q);
         etemp=e;
         e=d;
   if (fabs(p) >= fabs(0.5*q*etemp)||p<=q*(a-x)||p>=q*(b-x))
         d=CGOLD*(e=(x >= xm ? a-x: b-x));
   else  {
         d=p/q;
```

```
            u=x+d;
            if (u-a < tol2 || b-u < tol2)
               d=SIGN(tol1,xm-x);
        }
    } else {
        d=CGOLD*(e=(x >= xm ? a-x : b-x));
    }
    u=(fabs(d) >= tol1 ? x+d: x+SIGN(tol1,d));
    fu=(*f)(u);
    if (fu <= fx) {
          if (u >= x) a=x; else b=x;
          SHFT(v,w,x,u)
          SHFT(fv,fw,fx,fu)
    } else {
         if (u < x) a=u; else b=u;
         if (fu <= fw || w == x) {
             v=w;
             w=u;
             fv=fw;
             fw=fu;
        } else if (fu <= fv || v == x || v == w) {
               v=u;
               fv=fu;
               }
        }
    }
    nrerror("Warning!!! too many iterations in BRENT");
```

```
    *xmin = x;
    return fx;
}

void mnbrak(ax,bx,cx,fa,fb,fc,func)
double *ax, *bx, *cx, *fa, *fb, *fc;
double (*func)();

/*
    Given a function func, and given distinct initial points
    ax and bx, this routine searches in the downhill direction
    (defined by the function as evaluated at the initial points)
    and returns new points ax, bx, cx which bracket a minimum of
    the minimum of the function. Also returned are the function
    values at the three points, fa, fb and fc.
*/

{
    double ulim,u,r,q,fu,dum;
    *fa=(*func)(*ax);
    *fb=(*func)(*bx);
    if (*fb > *fa) {
        SHFT(dum,*ax,*bx,dum)
        SHFT(dum,*fb,*fa,dum)
    }
    *cx=(*bx)+GOLD*(*bx - *ax);
    *fc=(*func)(*cx);
```

```
while (*fb > *fc) {
    r=(*bx - *ax) * (*fb - *fc);
    q=(*bx - *cx) * (*fb - *fa);
    u=(*bx)-((*bx - *cx) * q - (*bx - *ax) * r)/
        (2.0*SIGN(MAX(fabs(q-r),TINY),q-r));
    ulim=(*bx)+GLIMIT*(*cx - *bx);
    printf("mnbrak: entering big loop...\n");
    if ((*bx - u)*(u - *cx) > 0.0) {
        fu=(*func)(u);
        if (fu < *fc) {
            *ax=(*bx);
            *bx=u;
            *fa=(*fb);
            *fb=fu;
            return;
        } else if (fu > *fb) {
            *cx=u;
            *fc=fu;
            return;
        }
        u=(*cx)+GOLD*(*cx - *bx);
        fu=(*func)(u);
    } else if ((*cx-u)*(u-ulim) > 0.0 ) {
        fu=(*func)(u);
        if (fu < *fc) {
            SHFT(*bx,*cx,u, *cx+GOLD*(*cx - *bx))
            SHFT(*fb,*fc,fu,(*func)(u))
```

```
            }
        } else if ((u - ulim)*(ulim - *cx) >= 0.0) {
            u=ulim;
            fu=(*func)(u);
        } else {
            u=(*cx)+GOLD*(*cx - *bx);
            fu=(*func)(u);
        }
        SHFT(*ax,*bx,*cx,u)
        SHFT(*fa,*fb,*fc,fu)
    }
}


        /* THESE ARE UTILITIES COPIED FROM THE BOOK,     */
        /* NUMERICAL RECIPES IN C                        */
        /* This is the correct version for nrutil.c      */
        /* Last update   01-MAY-2536                     */

#include <stdio.h>

/* #include <malloc.h> -- not understood by VMS at Bartol */
/*      @(#)malloc.h 1.4 88/02/07 SMI; from S5R2 1.2     */

/*
        Constants defining mallopt operations
*/
#define M_MXFAST  1    /* set size of 'small blocks' */
```

```
#define M_NLBLKS  2   /* set num of small blocks in holding block */
#define M_GRAIN   3   /* set rounding factor for small blocks */
#define M_KEEP    4   /* (nop) retain contents of freed blocks */

/*
        malloc information structure
*/
struct mallinfo  {
        int arena;      /* total space in arena */
        int ordblks;    /* number of ordinary blocks */
        int smblks;     /* number of small blocks */
        int hblks;      /* number of holding blocks */
        int hblkhd;     /* space in holding block headers */
        int usmblks;    /* space in small blocks in use */
        int fsmblks;    /* space in free small blocks */
        int uordblks;   /* space in ordinary blocks in use */
        int fordblks;   /* space in free ordinary blocks */
        int keepcost;   /* cost of enabling keep option */

        int mxfast;     /* max size of small blocks */
        int nlblks;     /* number of small blocks in a holding block */
        int grain;      /* small block rounding factor */
        int uordbytes;  /* space (including overhead) allocated */
        int allocated;  /* number of ordinary blocks allocated */
        int treeoverhead; /* bytes used in maintaining the free tree */
};
```

```
extern  int     mallopt();
extern  struct mallinfo mallinfo();

  double f1dim(x)
  double x;
    {
        int j;
        double f, *xt, *dvector();
        void free_dvector();
        xt= dvector(1,ncom);
        for (j=1;j<=ncom;j++) xt[j]=pcom[j]+x*xicom[j];
        f=(*nrfunc)(xt);
        free_dvector(xt,1,ncom);
        return f;
   }
```

Appendix B

Direction Set (Powell's) Methods in Multidimensions [1]

We know how to minimize a function of one variable. If we start at a point P in N-dimensional space, and proced from there in some vector direction n, then any function of N varible $f(P)$ can be minimized along the line n by our one-dimensional methods. One can dream up various multidimensional minimization methods which consist of sequences of such line minimizations. Different methods will differ only by now, at each stage, they choose the next direction n to try. All such methods presume the existence of a "black-box" subalgorithm, which we might call **linmin** (given as an explicit routine at the end of this section), whose definition can be taken for now as

> **linmin**: Given as input the vectors P and n, and the function f, find the scalar λ that minimizes $f(P + \lambda n)$. Replace P by $P + \lambda n$. Replace n by λn. Done

All the minimization methods in this section and in the two sections following fall under this general schema of successive line minimizations. In this section we consider a class of methods whose choise of successive directions does not involve explicit computation of the function's gradient; the next two section do require gradient calculations. You will note that we need not specify whether **linmin** uses gradient information or not. That choice is up to you, and its

[1] From Press et al. (1988)

optimization depends on your particular function. You would be crazy, however, to use gradients in linmin and not use them in the choice of directions, since in this latter role they can drastically reduce the total computational burden.

But what if, in your application, calculation of the gradient is out of the quiestion. You might first think of this simple method: Take the unit vector $e_1, e_2, ...e_N$ as a *set of directions*. Using **linmin**, move along the first direction to its minimum, then *from there* along the second direction to *its* minimum, and so on, cycling through the whole set of directions as many times as necessary, until the function stops decreasing.

This dumb method is actually not too bad for many functions. Even more interesting is why it *is* bad, i.e. very inefficient, for some other functions. Consider a function of two dimensions whose contour map (level lines) happens to define a long, narrow valley at some angle to the coordinate basis vectors (see Figure 1). Then the only way "down the length of the vally" going along the basis vectors at each is by a series of many tiny steps. More generally, in N dimensions, if the function's second derivatives are much larger in magnitude in some directions than in others, then many cycles through all N'basis vectors will be required in order to get anywhere. This condition is not all that unusual; by Murphy's Law, you should count on it.

Obviously what we need is a better set of directions than the e_i's. All *direction set methods* consist of prescriptions for updating the set of directions as the method peoceeds, attempting to come up with a set which either (i) includes some very good directions that will take us far along narrow valleys, or else (more subtly) (ii) includes some number of "non-interferin" directions with the special property that minimization along one is not "spoiled" by subseqyent minimization along another, so that interminable cycling through the set of directions can be avoided.

Conjugate Directions

This concept of "non-interfering" directions, more conventionally called conjugate directions, is worth making mathematically explicit.

First, note that if we minimize a function along some direction u, then the gradient of the function must be perpendicular to u at the line minimum; if not, then there would still be a nonzero direction derivative along u.

Next take some particular point P as the origin of the coordinate system with coordinates x. Then any function f can be approximated by its Taylor series

$$\begin{aligned} f(x) &= f(p) + \sum_i \frac{\partial f}{\partial x_i} x_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2 f}{\partial x_i \partial x_j} x_i x_j + \dots \\ &\approx c - b \cdot x + \frac{1}{2} x \cdot A \cdot x \end{aligned} \tag{1}$$

where

$$c \equiv f(P) \quad b \equiv -\nabla f|_P \quad [A]_{i,j} \equiv \frac{\partial^2 f}{\partial x_i \partial x_j}|_P \tag{2}$$

The matrix A whose components are the second partial derivative matrix of the function is called the *Hessian matrix* of the function at P.

In the approximation of (1), the gradient of f is easily calculated as

$$\nabla f = A \cdot x - b \tag{3}$$

(This implies that the gradient will vanish - the function will be at an extremum - at a value of x obtained by solving $A \cdot x = b$.)

How does the gradient ∇f *change* as we move along some derection? Evidently

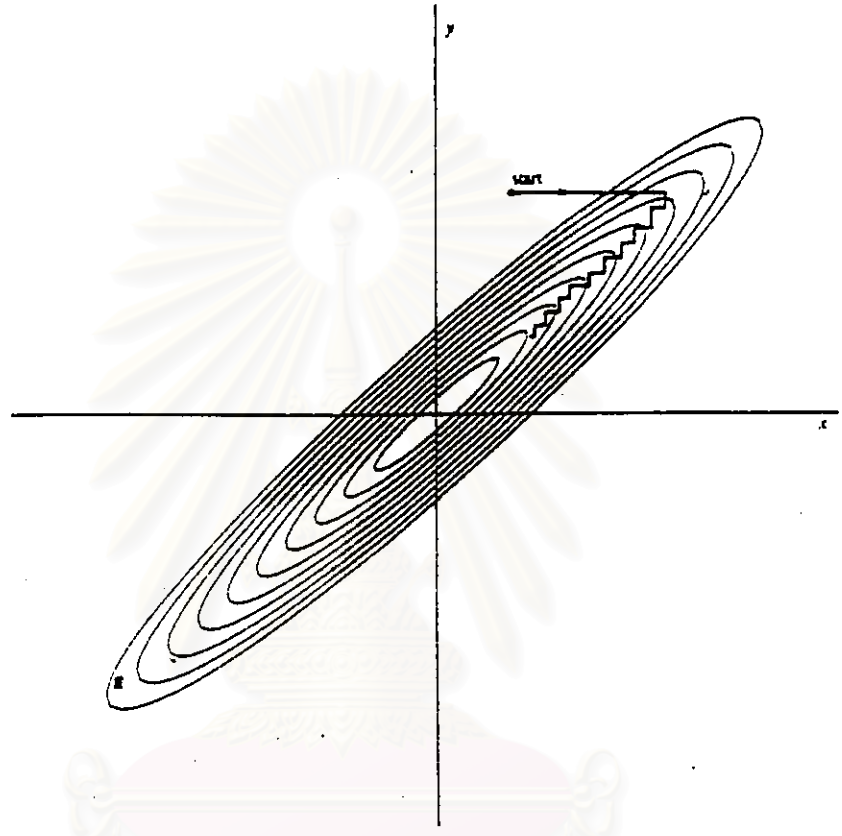

Figure B.1: Successive minimizations along coordinate directions in a long, narrow "valley" (show as contour lines). Unless the valley is optimally oriented, this is extremely inefficient, taking many tiny steps to get to the minimum, crossing abd re-crossing the principal axis.

$$\delta(\nabla f) = A \cdot (\delta x) \tag{4}$$

Suppose that we have moves along some direction u to a minimum and now propose to move along some new direction v. The condition that motion along v not *spoil* our minimization along u is just that the gradient stay perpendicular to u, i.e. that the change in the grade in the gradient be perpendicular to u. By equation (4) this is just

$$0 = u \cdot \delta(\nabla f) = u \cdot A \cdot v \tag{5}$$

When (5) holds for two vectors u and v, they said to be *conjugate*. When the relation holds pairwise for all members of a set of vectors, they are said to be a conjugate set. If you do successive line minimization of a function along a conjugate set of directions, then you don't need redo any of those directions (unless, of course, you spoil things by minimizing along a direction that they are *not* conjugate to).

A triumph for a direction set method is to come up with a set of N linearly independent, mutually conjugate directions. Then, one pass of N line minimizations will put it exactly at the minimum of a quadeatic from like (1). For functions f which are not exactly quadratic forms, it won't be exactly at the minimum; but repeated cycles of N line minimizations will in due course converge *quagratically* to the minimum.

Powell's Quadratically Convergent Method

Powell first discoved a direction set method which does produce N mutually conjugate directions. Here is hoe it goes: Initialize the set of directions u_i to the basis vectors,

$$u_i = e_i \qquad i = 1, \ldots, N \tag{6}$$

Now repeat the following sequence of steps ("basis procedure") until your function stops decreasing:

1. Save your starting position as P_0.
2. For $i = 1, \ldots, N$, move P_{i-1} to the minimum along direction u_i and call this point P_i.
3. For $i = 1, \ldots, N-1$, set $u_i \leftarrow u_{i+1}$.
4. Set $u_N \leftarrow P_N - P_0$.
5. Move P_N to the minimum along direction u_N and call this point P_0.

Powell, in 1964, showed that, for a quadratic from like (1), k iterations of the above basis procedure produce a set of directions u_i whose last k members are mutually conjugate. Therefore, N iterations of the basic procedure, amounting to $N(N+1)$ line minimizations in all, will exactly minimize a quadratic from. Brent (1973) give proofs of these statements in accessible form.

Unfortuanately, there is a problem with Powell's quadraticlly convergent algirithm. The procedure of throwing away, at each stage, u_1 in favor of $P_N - P_0$ tends to produce sets of directions that "fold up on each other" and become linearly dependent. Once this happens, then the procedure finds the minimum of the function f only over a subspace of the full N-dimensional case: in other words, it gives the wrong answer. Therefore, the algorithm must not be used in the form given above.

There are a number of ways to fix up the problem of linear dependence in Powell's algorithm, among them:

1. You can reinitialize the set of directions u_i to the basis vectors e_i after every N or $N+1$ iterations of the basic procedure. This peoduces a serviceable method. which we commend to you if quadratic convergence is important for your application (i.e. if ypur functions are close to quadratic forms and if you desire hjgh accuracy).

2. Brent point out that the set if directions can equally well be reset to the columns of any orthogonal matrix. Rather than throw away the information on conjugate directions already built up, he resets the direction set to calculated principal directions of the matrix A (which he gives a produre for determining). The calculation is essentially a singular value decomposition algorithm. Brent has a number of other cute tricks up his sleeve, and his modification of Powell's method is probably the best presently known. Consult his book for a detailed description and listing of the program. Unfortunately it is rather too elaborate for us to include here.

3. You can give up the property of quadratic convergence in favor of a more heuristic scheme (due to Powell) which tries to find a few good directions along narrow valley instead of N necessarily conjugate directions. This is the method which we now implement. (It is also the version of Powell's method given in Acton, from which parts of the following discussion are drawn.)

Powell's Mehtod Discarding the Direction of Largest Decrease

The fox and the grapes: Now that we are going to give up the property of quadratic concergence. was it so important after all? That depends on the function that you are minimizing. Some applications produce functions with long. twidty valleys. Quadratic convergence is of no particular advantage to a program which must slalom down the legrth of a valley floor that twists one way

and another (and another, and another, . . . - ther are N dimensions!). Along the long direction, a quadratically convergent method is trying to extrapolate to the minimum of a parabola which just isn't (yet) there; while the conjugacy of the $N-1$ transverse directions keeps getting spoiled by the twists.

Sooner or later, however, we do arrive at an approximately ellipsoidal minimum (cf. equation 1 when b, the gradient, is zero). Then, depending on how much accuracy we require, a method with quadratic convergence can save us several times N^2 extra line minimizations, since quadratic convergence *doubles* the number of signigicant figures at each iteration.

The basic idea of our now-modified Powell's method is still to take $P_N - P_0$ as a new direction: it is, after all, the average direction moved after trying all N possible directions. For a valley whose long direction is twisting slowly, this direction is likely to give us a good run along the new long direction. The change is to discard the old direction along which the function f made its *largest decrease*. This seems paradoxical, since that direction was the *best* of the previous iteration. However, it is also likely to be a major component of the new direction that we are adding, so dropping it gives us the best chance of avoiding a buildup of linear dependence.

There are a couple of exceptions to this basic idea. Sometimes it is better *not* to add a new direction at all. Define

$$f_0 = f(P_0) \quad f_N = f(P_N) \quad f_E = f(2P_N - P_0) \tag{7}$$

Here f_E is the function value at an "extrapolated" point somewhat further along the proposed new direction. Also define Δf to be the magnitude of thelargest decrese along one particular direction of the present basis procedure iteration. (Δf is a positive number.) Then:

1. If $f_E \geq f_0$, then keep the old set of directions for the next basic procedure, because the average direction $P_N - P_0$ is all played out.

2. If $2(f_0 - 2f_N + f_E)[(f_0 - f_N) - \Delta f]^2 \geq (f_0 - f_E)^2 \Delta f$, then keep the old set of directions for the next basic procedure, because either (i) the decrease along the average direction was not primarily due to any single direction's decrease, or (ii) there is a substantial second derivative along the average direction and we seem to be near to the bottom of its minimum.

The following routine implements Powell's method in the version just described. In the routine, **xi** is the matrix whose columns are the set of directions n_i; otherwise the correspondence of notation should be self-evident.

```
#include <math.h>
#define  ITMAX 200    % Maximum allowed iterations.
static   float sqrarg;
#define  SQR(a) (sqrarg=(a),sqrarg*sqrarg)

void powell(p,xi,n,ftol,iter,fret,func)
float p[],**xi,ftol,*fret,(*func)();
int   n,*iter;
Minimization of a function func of n variables. Input consists of
an initial starting point p[1..n]; an initial matrix
xi[1..n][1..n] whose columns contain the initial set of directions
(usually the n unit vectors); and ftol, the fractional tolerance
in the function value such that failure to decrease by more than
this amount on one iteration signals doneness. On output, p is set
to the best point found, xi is the then-current derection set,
fret is the retuened function value at p, and iter is the number
```

```
of iterations taken. The routine linmin is used.
{
   int i,ibig,j;
   double t,fptt,fp,del;
   double *pt,*ptt,*xit,*dvector();
   void linmin(),nrerror(),free_dvector();

   pt=dvector(1,n);
   ptt=dvector(1,n);
   xit=dvector(1,n);
   *fret=(*func)(p);
   for (j=1;j<=n;j++)  pt[j]=p[j];  % Save the initial point.
   for (*iter=1;; (*iter)++) {
       fp=(*fret);
       ibig=0;
       del=0.0;
       for (i=1;i<=n;i++)  {
          for (j=1;j<=n;j++) xit[j]=xi[j][i];
          fptt=(*fret);
          linmin(p,xit,n,fret,func);
          if (fabs(fptt-(*fret)) >del) {
              del=fabs(fptt-(*fret));
              ibig=i;
          }
       }
       if (2.0*fabs(fp-(*fret)) <= ftol*(fabs(fp)+fabs(*fret))){
           free_dvector(xit,1,n);
```

```
            free_dvector(ptt,1,n);
            free_dvector(pt,1,n);
            return;
        }
    if (*iter == ITMAX)
      nrerror("Warning!!! too many iterations in POWELL");
        for (j=1;j<=n;j++) {
            ptt[j]=2.0*p[j]-pt[j];
            xit[j]=p[j]-pt[j];
            pt[j]=p[j];
        }
        fptt=(*func)(ptt);
        if (fptt < fp) {
          t=2.0*(fp-2.0*(*fret)+fptt)*SQR(fp-(*fret)-del)
            -del*SQR(fp-fptt);
            if (t < 0.0) {
              linmin(p,xit,n,fret,func);
              for (j=1;j<=n;j++)  xi[j][ibig]=xit[j];
            }
        }
    }
}
```

Implementation of Line Minimization

Make no mistake, there is a *right* way to implement **linmin**: It is to use the *methods* of one-dimensional minimization, but to rewrite the programs

of those sections so that their bookkeeping is done on vector-valued points P (all lying along a given direction n) rather than scalar-valued abscissas x. That straighrforward task produces long routines densely populated with "

```
for(k=1;k<=n;k++)
```

" loops.

We do not have space to include such routines in this book. Our **linmin**, which works just fine, is instead a kind of bookkeeping swindle. It constructs an "artificial" function of one variable called **f1dim**, which is the value of your function, say, **func**, along the line going through the point p in the direction **xi**. **linmin** calls our familiar one-dimensional routines **mnbrak** and **brent** and instructs them to minimize **f1dim**. **linmin** communicates with **f1dim** "over the head" of **mnbrak** and **brent**, through global (external) varibles. That is also how it passes to **f1dim** a pointer to your user-supplied function.

The only thing inefficient about **linmin** is this: Its use as an interface between a multidimensional minimization strategy and a one-dimensional minimization routine results in some unnecessary copying of vectors from hither to you and back again. That should not normally be a signficant addition to the overall comoutation burden, but we cannot disguise its inelegance.

```
#define TOL 2.0e-4
int     ncom=0;
float   *pcom=0,*xicom=0,(*nrfunc)();

void linmin(p,xi,n,fret,func)
double p[], xi[], *fret, (*func)();
int n;
```

```
{
    int j;
    double xx, xmin, fx,fb,fa, bx,ax;
    double brent(), f1dim(), *dvector();
    void mnbrak(), free_dvector();

    ncom=n;
    pcom=dvector(1,n);
    xicom=dvector(1,n);
    nrfunc=func;
    printf("1\t");
    for (j=1;j<=n;j++) {
        pcom[j]=p[j];
        xicom[j]=xi[j];
    }
    ax=0.0;
    xx=1.0;
    bx=2.0;
    mnbrak(&ax,&xx,&bx,&fa,&fx,&fb,f1dim);
    *fret=brent(ax,xx,bx,f1dim,tol,&xmin);
    for ( j=1;j<=n; j++) {
      xi[j] *=xmin;
      p[j] += xi[j];
    }
    free_dvector(xicom,1,n);
    free_dvector(pcom,1,n);
}
```

Appendix C

FLARE DATA SHEET (Example of ISEE-3/ULEWAT)

Computational Astrophysics Group กลุ่มดาราศาสตร์เชิงคอมพิวเตอร์

Department of Physics ภาควิชาฟิสิกส์

Faculty of Science คณะวิทยาศาสตร์

Chulalongkorn University จุฬาลงกรณ์มหาวิทยาลัย

Flare date (year month day): 1978 / 09 / 23 Day of year (DOY): 266

Hα data: 0940 UT at +35 ° (+ N, - S) and +50 ° (+ W, - E)

corrected to +50 ° (+ W, - E)

X-ray flux decay time: min. (source: Cliver et al. 1989)

Solar wind condition: [ ] nearly constant speed, density [ ] slowly varying speed, density

[ ] other: describe

Solar wind speed: 370 km/s -----> $\beta_{sw} = v_{sw} / 299{,}790$ km/s = 0.001234

Magnetic field condidtion: [/] relatively steady direction, magnitude

[ ] slowly varying direction, magnitude

[ ] other: describe

Latitude of mag. field: 22.5 ° (+ N, - S) ---> θ_B = 90° - (latitude) = 67.5 °

Long. of mag. field: 110 ° (+ N, - S) ---> φ_B = (long.) - 180° or 360° = -70 °

Approximate φ_B as [ ] +22.5° [ ] -22.5° [/] -67.5° [ ] -112.5° [ ] other

$\varphi_{Earth} \approx [(DOY - 2)/365]*360°$ = 260.38 °, $r_{Earth} = 1$ AU / (1 + 0.0017 cos φ_{Earth}) = 0.99972 AU

$r_{s/c}$ = 0.9897 AU (for ISEE-3 before mid-1982, subtract 0.010 AU)

Corrected flare longitude = (flare longitude) - (s/c longitude) = ° (+ W, - E; enter at top)

Date is in period starting (month-date) -> Earth's solar latitude (in degrees; + N, - S):

2-9		->	-7	6-3	or	12-4 ->	0
4-2	or	1-25 ->	-6	6-11	or	11-26 ->	+1
4-17	or	1-15 ->	-5	6-19	or	11-18 ->	+2
4-28	or	1-5 ->	-4	6-28	or	11-10 ->	+3
5-8	or	12-28 ->	-3	7-7	or	10-31 ->	+4
5-17	or	12-20 ->	-2	7-17	or	10-20 ->	+5
5-25	or	12-12 ->	-1	7-28	or	10-5 ->	+6

Solar latitude of Earth = +7 ° (+ N, - S) 8-13 -> +7

Solar latitude of s/c = +7 ° (+ N, - S; for ISEE-3 before mid-1982, same as for Earth)

$R = v_{sw} / \Omega \cos$ (s/c latitude) = (v_{sw} / km per s) (0.002291 AU) / cos (s/c latitude) = 0.854 AU

(The above formula assumes a sidereal solar rotation period of 24.92 days corresponding to a synodic period of 26.75 days; Bai 1987)

Cosmic ray particle species: e Mass (MeV/c^2): 0.511 (0.511 for e^-, 938.27 for p)

Energy values (MeV): 0.180, 0.250, 0.610, 1.100, 3.000, 1.500, 15.000, 30.00, 60.000

Momentum values (MeV/c): 0.465, 0.564, 0.998, 1.528, 3.470, 7.995, 15.503, 30.507, 60.509

{Use (p in MeV/c) = $\sqrt{[(E \text{ in MeV})^2 + (m \text{ in MeV}/c^2)^2]}$ }

Spectrum (please indicate source or reference): _____

_____

(e.g., energies 1-4: 10900 * (p / 0.5 MeV per c)$^{-5.57}$;

energies 5-9: 0.0265 * (p / 15 MeV per c)$^{-2.54}$; from Moses et al. 1989)

Curriculum Vitae

Mr Wiwat Youngdee was born on October 25, 1972 in Nakornrachasima Province. He received his B.Sc. degree in physics from Khon Kaen University in 1993. He received support from the Institute for the Promotion of Teaching Science and Technology for both his undergraduate and graduate studies.