References

Axford, W. I. Anisotropic diffusion of solar cosmic rays. Planet. Space Science **13** (1965), 1301-1309.

Bieber, J. W. Numerical and analytic solutions of the Boltzmann equation for cosmic ray transport. Ph.D. Thesis, Univ. of Maryland, 1977.

———. and others. Interplanetary pitch angle scattering and coronal transport of solar energetic particles. J. Geophys. Res. **85** (1980), 2313.

Earl, J. A. Diffusion of charged particles in a random magnetic field. Astrophysical Journal **180** (1973), 227.

———. Coherent propagation of charged-particle bunches in random magnetic fields. Astrophysical Journal **188** (1974), 379-397.

———. The diffusive idealization of charged-particle transport in random magnetic fields. Astrophysical Journal **193** (1974), 231-242.

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

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

———. The effects of convection upon charged particles transport in random magnetic fields. Astrophysical Journal **278** (1984), 825.

———. and others. Comparison of three numerical treatments of charged particles transport. Accepted by the Astrophysical Journal (1995).

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

Fisk, L. A. and Axford, W. I. Effect of energy changes on solar cosmic rays. Journal of Geophysical Research, Space Physics **73** (1968), 4396-4399.

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

———-. Propagation of cosmic rays in the solar wind. Rev. Geophys. Space Phys. **9** (1971), 27.

Hamilton, D. C. The radial transport of energetic solar flare particles from 1 to 6 AU. Journal of Geophysical Research **16** (1977), 2157-2169.

Hasselmann, K. and Wibberenz, G. Scattering of charged particles by random electromagnetic fields. Z. Geophys. **34** (1968), 353.

Lupton, J. E. and Stone, E. C. Solar flare particle propagation: comparison of a new analytic solution with spacecraft measurements. Journal of Geophysical Research **78** (1973), 1007-1018.

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

Meyer, P., Parker, E. N. and Simpson, J. A. Solar cosmic rays of February, 1956 and their propagation through interplanetary space. Physical Review **104** (1956), 768-783.

Moraal, H. Observations of the eleven-year cosmic ray modulation cycle. Space Science Reviews **19** (1976), 845-920.

Ng, C. K. Particles in evolving interplanetary magnetic fields. Solar Physics **114** (1987), 165 179.

Ng, C. K. and Gleeson, L. J. The propagation of solar cosmic ray bursts. Solar Physics **20** (1971), 166.

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

Parker, E. N. Dynamics of the interplanetary gas and magnetic fields. Astrophysical Journal **128** (1958), 664-676.

———. The passage of energetic charged particles through interplanetary space. Planet. Space Science **13** (1965), 9-49.

Reames, D. V. and Stone, R. G. The identification of solar ^{3}He-rich events and the study of particle accerelation at the Sun. Astrophysical Journal **216** (1977), 108-122.

Roelof, E.C. Propagation of solar cosmic rays in the interplanetary magnetic field, in Lectures in High Energy Astrophysics ed. H. Ogelman & J. R. Wayland (NASA SP-199) (Washington: NASA) (1969), 111-135.

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

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

Samuel, D. C. and de Boor, C. Elementary numerical analysis and algorithmic approach. Third Edition. International Student Edition 1982, 406-410.

Schulze, B. M., Richter, A. K. and Wibberenz, G. Influence of finite injections and of interplanetary propagation on time-intensity and time-anisotropy profiles of solar cosmic rays. Solar Physics **54** (1977), 207-228.

Simpson, J. A. Elemental and isotropic composition of the galactic cosmic rays. Ann. Rev. Nucl. Part. Sci. **33** (1983).

Tuska, E. B. Charge-sign dependent solar modulation of 1-10 GV cosmic rays. Ph.D. Thesis, Univ. of Delaware, (1990).

Webb, G. M. and Gleeson, L. J. On the equation of transport for cosmic ray

particles in the interplanetary region. AStrophys. and Space Sci. **60** (1979), 335.

Zwickl, R. D. and Roelof, E. C. Interplanetary propagation of < 1-MeV protons in nonimpulsive energetic particle events. Journal of Geophysical Research **86** (July 1981): 5449-5471.

Appendix A
Wind Program For Simulation

```
/* wind.c (t) - July 9th, 1995

    Uses t as the independent variable instead of s.

    Thiranee Khumlumlert and David Ruffolo
    Department of Physics
    Faculty of Science
    Chulalongkorn University
    Bangkok 10330, Thailand

  wind.c - November 12th, 1994

    Passed prints to printout().

  wind.c - October 16th, 1994

    Changed size of g, h, h3 arrays so can be used in neutrons(),
    printout().  Passed p, betasw to printout().

  wind.c - August 10th, 1994

    Changed elements() to include eratio[] in deceleration effect.

  wind.c - April 18th, 1994

    Reset sstep to ensure that zstep=sstep*mustep.

  wind.c - March 15th, 1994
```

Put decel() before stream(). Removed sk[][] and imatrix().

wind.c – February 27th, 1994

Added new arrays for stream(), adapted to new format of allocation routines in nrutil.c. Imported non-integer ZDUMP mechanism from transport.c, modified to start at z=0. New checks on neg, allneg, and ntotal. New use of TINY. Corrected errors in mechanism for fixing negative fluxes.

wind.c – May 8th, 1993

Reworked the variable lprint - set up #define LPRINT

wind.c – March 16th, 1993

Fixed odds and ends. Added dump capability. Added TINY so fluxes within TINY of zero become zero, and thus aren't rejected if they are negative. Changed TOLERANCE to a combination absolute/relative tolerance.

wind.c – December 12th, 1992

Modified for the new version of stream(), slide() eliminated, corr[w] eliminated. a[w][l][u], etc. are now at z=(l-0.5)*zstep again.

wind.c – November 20th, 1992

Calls decel() for deceleration.

wind.c – November 12th, 1992

elements(), step() changed so that matrix elements are defined at 0, zstep, 2*step, ..., length, and a[w][l][u] is the matrix element a[u] at z(lab frame) = l*zstep (same for c).

Now z(lab frame) = (l-1+corr[w])*zstep, so in step() the matrix elements are interpolated as (1-corr[w])*a[w][l-1][u] + corr[w]*a[w][l][u] (same for c).

wind.c – November 6th, 1992

Main program for simulating the transport of solar energetic particles. The possible transport processes are:

streaming
pitch-angle scattering
adiabatic focusing
adiabatic deceleration
injection

Key assumptions include neglecting drifts and diffusion perpendicular to the magnetic field, and calculating adiabatic deceleration as if the solar wind velocity were directed along the magnetic field and of a constant magnitude. Otherwise, the above processes can be easily modified by changing the appropriate subroutines.

Necessary files and subroutines:

wind.c	main, elements, step
decel.c	decel
field.c	dzz, diffcoeff, antideriv, mudot, arclength, radius, cospsi, dsecdz, zenith, decelrate
initial.c	initial
inject.c	inject
nrutil.c	nrerror, dvector, lvector, dmatrix, darray, free_dvector, free_lvector, free_dmatrix, free_darray
printout.c	printout
stream.c	stream
tridag.c	tridag

```
    Variables input from the user:

        starttime   Initial value of t (min.)
        stoptime    Final value of t (min.)
        timestep    Time step (min.)
        prints      Printing interval (AU)
        nmu         Number of mu points
        length      Length of simulation region (AU)
        np          Number of momentum points
        p[1..np]    Momenta (energy units)
        m           Particle mass (energy units)
        betasw      v/c of solar wind
        lambda      Scattering mean free path (AU)
        q           Scattering power law index
        printextra  Print extra diagnostic information? (0/1)

    David Ruffolo and Burin Asavapibhop
    Department of Physics
    Faculty of Science
    Chulalongkorn University
    Bangkok 10330, Thailand
*/

/*15 May 1992   Montien Tienpratip, Wantana Songprakob, Pisit Lilapattana */

#include <math.h>
#include <stdio.h>

#define  C        0.1202
#define  ATOL     1.0e-09  /* Maximum allowable error (absolute)     */
#define  FTOL     1.0e-03  /* Maximum allowable error (fractional)   */
#define  TINY     1.0e-12
#define  LPRINT   500      /* Value of lprint - set to 0 for default */
#define  DUMP     1        /* 0 - DON'T DUMP, 1 - DUMP               */
#define  ZDUMP    2.0      /* Expansion factor for zstep in next run
                              (must be greater than or equal to 1).  */
```

```
double   ***f;
double   ***a, ***c;
double   *aa, *bb, *cc, *an, *bn, *cn, **fs, *g, *h, *h3, *fint, *tosun;

main()
{
   FILE     *fp_dump;
   double   nextprint;
   double   starttime, stoptime, printtime, templeng, timestep, time, v;

   double   *beta, *ke, length, m, *p;
   double   aoverv, lambda, q, betasw, vsw, dzz1;
   double   dumpstep, frac, mustep, z, zdump, *zstep;
   int      nmu, np, printextra, u, w;
   long     *nz, l, ll;

   double   ***darray(), **dmatrix(), *dvector();
   long     *lvector();
   void     free_darray(), free_dvector(), free_lvector(), free_dmatrix();
   void     nrerror();

   double   dzz();
   void     elements(), initial(), inject(), printout(), step();

     /* Input parameters from the user */

   printf("Hello! Welcome to wind.\n");
   printf("Please input the following parameters:\n");
   printf("\nStarting value of s=velocity*time (AU): ");
   scanf("%lf",&starttime);
   printf("\nFinal value of s (AU): ");
   scanf("%lf",&stoptime);
   printf("\ns step (AU): ");
   scanf("%lf",&timestep);
   printf("\ns interval after which to print out data (AU): ");
```

```
scanf("%lf",&printtime);
printf("\nNumber of mu points (must be odd): ");
scanf("%d",&nmu);
printf("\nLength in the z-direction (AU, best if multiple of sstep): ");
scanf("%lf",&length);
printf("\nNumber of momentum (energy) values: ");
scanf("%d",&np);

if (np <= 0) nrerror("wind: np <= 0");

beta  = dvector(1,np);
ke    = dvector(1,np);
p     = dvector(1,np);
tosun = dvector(1,np);
zstep = dvector(1,np);
nz    = lvector(1,np);

for (w=1;w<=np;w++) tosun[w] = 0.0;

printf("\nEnter %d momentum values, from lowest to highest",np);
printf("\n(in energy units): ");
for (w=1;w<=np;w++) {
   printf("\n   p[%d] = ",w);
   scanf("%lf",&p[w]);
}
printf("\nEnter the partice mass (in energy units): ");
scanf("%lf",&m);
printf("\nSolar wind velocity divided by c: ");
scanf("%lf",&betasw);
printf("\nScattering mean free path, lambda (AU): ");
scanf("%lf",&lambda);
printf("\nScattering power-law index, q : ");
scanf("%lf",&q);
printf("\nDo you want me to print extra diagnostic information? ");
printf("\n(enter 1 for yes, 0 for no) ");
scanf("%d",&printextra);
```

```
  /* Calculating nz, ke, beta, and vsw. */

for (w=1;w<=np;w++) {
   ke[w] = sqrt(p[w]*p[w]+m*m) - m;
   beta[w] = p[w] / (ke[w]+m);
   if (printextra)
      printf("\nke[%2d] = %12lf,   beta[%2d] = %12lf",w,ke[w],w,beta[w]);
}
printf("\n");

vsw = betasw * C;

   /* Calculating nz, mustep, zstep; length is now an integral
      multiple of zstep.  Reset sstep to ensure that zstep=mustep*sstep.
   */

mustep = 2.0 / (double)nmu;
zstep[1] = mustep * beta[1] * C * timestep;
nz[1] = length / zstep[1] + 0.5;
if (nz[1] == 0) nrerror("wind: nz[1] == 0");
printf("\ninput w=1: nz[1](double)=%lf, nz[1](int)=%ld, length=%lf\n",
 length/zstep[1],nz[1],nz[1]*zstep[1]);
length = templeng = nz[1] * zstep[1];
for (w=2;w<=np;w++) {
   zstep[w] = mustep * beta[w] * C * timestep;
   nz[w] = templeng / zstep[w] + 0.999;
   if (nz[w] == 0) nrerror("wind: nz[w] == 0");
   templeng = nz[w] * zstep[w];
   printf("\ninput w=%d: nz[%d](int)=%ld, templeng=%lf\n",
    w,w,nz[w],nz[w]*zstep[w]);
}

printf("\nmustep = %lf\n",mustep);
for (w=1;w<=np;w++) printf("\nzstep[%d] = %lf\n",w,zstep[w]);
```

```
/* Echoing */

printf("\nYou input the following parameters:\n");
printf("\nStarting time (min.)   : %12lf",starttime);
printf("\nFinal time (min.)      : %12lf",stoptime);
printf("\nTime step (min.)       : %12lf",timestep);
printf("\nPrint interval (min.)  : %12lf",printtime);
printf("\nNumber of mu points    : %5d",nmu);
printf("\nLength *               : %12lf\n",length);

for (w=1,w<=np;w++) printf("\np[%d] = %12lf",w,p[w]);

printf("\n\nParticle mass          : %12lf",m);
printf("\nSolar wind velocity/c  : %12lf",betasw);
printf("\nLambda (in AU)         : %12lf",lambda);
printf("\nq                      : %12lf",q);
printf("\nPrintextra             : %5d\n",printextra);
printf("\n* reset length to be an integral multiple of");
printf("\n  zstep[1]=mustep*beta[1]*C*timestep\n");

/* Idiot Proofing */

if (stoptime < starttime) nrerror("wind: stoptime < starttime");
if (timestep <= 0) nrerror("wind: timestep <= 0");
if (printtime < timestep) nrerror("wind: printtime < timestep");
if (nmu % 2 == 0 || nmu <= 0) nrerror("wind: nmu is bad");
if (length <= 0) nrerror("wind: length <= 0");
if (p[1] <= 0) nrerror("wind: p[1] <= 0");
for (w=2;w<=np;w++) if (p[w] <= p[w-1]) nrerror("wind: p's reversed");
if (m <= 0) nrerror("wind: m <= 0");
if (betasw < 0 || betasw >= 1) nrerror("wind: betasw is bad");
if (lambda <= 0) nrerror("wind: lambda <= 0");
if (q < 0 || q >= 2) nrerror("wind: q is bad");

/* Defining arrays */
```

```
f    = darray(1,np,1,nz,1,nmu);
a    = darray(1,np,1,nz,1,nmu);
c    = darray(1,np,1,nz,1,nmu);

if (printextra) printf("\n Now we are here step1 \n");

aa   = dvector(1,nmu);
bb   = dvector(1,nmu);
cc   = dvector(1,nmu);
an   = dvector(1,nmu);
bn   = dvector(1,nmu);
cn   = dvector(1,nmu);
g    = dvector(0,nmu+1>np-1 ? nmu+1 : np-1);
h    = dvector(0,nmu+1);
h3   = dvector(0,nmu+1);
fint = dvector(0,nmu+1);
fs   = dmatrix(1,nz[1],1,nmu);

if (printextra) printf("\n Now we are here step2 \n");
dzz1 = dzz(mustep,q);
aoverv = 3.0 * dzz1 / lambda;
printf("\n q = %lf, dzz = %lf, lambda = %lf, aoverv = %lf\n",q,dzz1,
   lambda,aoverv);

initial(np,p,nz,zstep,mustep,nmu);
printf("\n\n time = %lf\n",starttime);
printout(starttime,printtime,beta,np,p,nz,zstep,nmu,betasw);
elements(timestep,beta,np,nz,zstep,mustep,nmu,aoverv,q,vsw,printextra);
nextprint = starttime + printtime;

  /* FOR EACH TIME STEP:
       INJECT NEW FLUX (IF NECESSARY).
       CALCULATE f AT THE NEW TIME STEP, ACCORDING TO THE
       TRANSPORT EQUATION.
       PRINT OUT THE DATA (IF NECESSARY).
  */
```

```
for (time=starttime;time+timestep<=stoptime+timestep/2;time+=timestep) {
   inject(time,timestep,beta,np,nz,zstep,mustep,printextra);
   step(time,timestep,beta,m,np,p,nz,zstep,nmu,vsw,printextra);
   if (time+timestep >= nextprint-0.01*timestep) {
      printf("\n\n time = %lf\n",time+timestep);
      printout(time+timestep,printtime,beta,np,p,nz,zstep,nmu,betasw);
      nextprint += printtime;
   }
}

   /* "DUMP" OUT f FOR SELECTED z'S, SO THAT THE RUN MAY BE CONTINUED. */

if (DUMP) {
   fp_dump = fopen("dump.dat","w");

   printf("\n\nNow dumping f(mu,z) for a later run...\n");
   zdump = ZDUMP;
   if (zdump < 1.0) zdump = 1.0;
   printf("nz = %ld, zdump = %lf\n",nz,zdump);
   for (w=1;w<=np;w++) {
      dumpstep = zstep[w] * zdump;
      for (ll=1,z=0.5*dumpstep;z<=length-zstep[w]/2.0;
       ll++,z=(ll-0.5)*dumpstep) {
         l    = z/zstep[w] + 0.5;
         frac = z/zstep[w] + 0.5 - (double)l;
         for (u=1;u<=nmu;u++) fprintf(fp_dump,"%12le\n",
            (1.0-frac)*f[w][l][u]+frac*f[w][l+1][u]);
      }
   }

   fclose(fp_dump);
}


printf("\nWow! I have finished my work!\n");
```

```
    free_dvector(beta,1);
    free_dvector(ke,1);
    free_dvector(p,1);
    free_dvector(tosun,1),
    free_dvector(zstep,1);

    free_dvector(aa,1);
    free_dvector(bb,1);
    free_dvector(cc,1);
    free_dvector(an,1);
    free_dvector(bn,1);
    free_dvector(cn,1);
    free_dvector(g,0);
    free_dvector(h,0);
    free_dvector(h3,0);
    free_dvector(fint,0);
    free_dmatrix(fs,1,nz[1],1),

    free_darray(f,1,np,1,nz,1);
    free_darray(a,1,np,1,nz,1);
    free_darray(c,1,np,1,nz,1);
    free_lvector(nz,1),
}

/* elements

     A SUBROUTINE FOR transport.
     CALCULATES ELEMENTS OF MATRICES USED IN step.
*/

void     elements(timestep,beta,np,nz,zstep,mustep,nn  ,aoverv,q,vsw,printextra)
double   timestep, *beta, *zstep, mustep, aoverv, q, vsw;
int      np, nmu, printextra,
long     *nz;
{
```

```
double   advplus, diffplus, *eratio, mu, muplus, r, v, z;
int      u, w;
long     l, lprint;

double   diffcoeff(), mudot(), radius(), cospsi();

double   *dvector();
void     free_dvector();

eratio = dvector(1,nmu);

for (w=1;w<=np;w++) {
   v = beta[w] * C;

     /* FIRST, CALCULATE THE SOLAR RADIUS AT EACH POINT. */

   for (l=1;l<=nz[w];l++) {
      z = zstep[w]*(l-0.5);
      r = radius(z);

        /* CALCULATE MATRICES FOR PITCH-ANGLE SCATTERING AND FOCUSING

           IMPLICIT SCATTERING: THE MATRIX IS ONLY NONZERO ALONG
                                THE THREE CENTRAL DIAGONALS.

                                THE (u,u-1) ELEMENT IS a[u].
                                THE (u,u)   ELEMENT IS b[u].
                                THE (u,u+1) ELEMENT IS c[u].

           EXPLICIT SCATTERING: THE MATRIX IS ONLY NONZERO ALONG
                                THE THREE CENTRAL DIAGONALS.

                                THE (u,u-1) ELEMENT IS -a[u].
                                THE (u,u)   ELEMENT IS  d[u] = 2 - b[u].
                                THE (u,u+1) ELEMENT IS -c[u].
        */
```

```
    /* IF THERE IS NO PITCH-ANGLE SCATTERING OR FOCUSING, EACH
       ITERATION MATRIX IS THE UNIT MATRIX.

       Set the vector eratio[u] = E'/E = 1-mu*vsw*v*sec(psi)/C^2,
       which is needed in the diffusion term.
    */

for (u=1;u<=nmu;u++) {
    a[w][l][u] = 0.0;
    c[w][l][u] = 0.0;
    mu = -1 + (u-0.5)*mustep;
    eratio[u] = 1 - mu*v*vsw/(C*C*cospsi(z));
}

muplus  = -1.0 + mustep;
for (u=1;u<=nmu-1;u++,muplus+=mustep) {

       /* PITCH-ANGLE SCATTERING */

       /* The "4" below appears because each explicit
          or implicit substep accounts for timestep/4.
          After four substeps (explicit and implicit,
          before and after streaming), they account for
          the full timestep.

          During step(), the size of the substep and
          the number of substeps will be repeatedly
          changed by a factor of 2 until the resulting
          f converges to within TOLERANCE.

          Output of diffcoeff() and mudot() is multiplied
          by v here to avoid changing those routines from
          the "s" versions.  Thus, these outputs are to
          be interpreted as the rates per distance traveled.
       */
```

```
            diffplus = (timestep/(4*mustep*mustep))
          * v * diffcoeff(muplus,mustep,aoverv,q,r);

          a[w][l][u+1] -= diffplus * eratio[u];
          c[w][l][u]   -= diffplus * eratio[u+1];

            /* ADIABATIC FOCUSING */

            advplus = (timestep/(8*mustep)) * v * mudot(v,muplus,r,vsw);

          a[w][l][u+1] -= advplus;
          c[w][l][u]   += advplus;
        }
      }
    }

  /* IF printextra IS 1, PRINT OUT SELECTED MATRIX ELEMENTS. */

  if (printextra) {
    for (w=1;w<=np;w++) {
      if (w==1 || w==np) {
        printf("\nMatrix Elements: w = %d\n",w);
        lprint = LPRINT;
        if (lprint <= 0 || lprint >= nz[w]) lprint = nz[w]/4;
        for (l=1;l<=nz[w];l+=lprint) {
          printf("\n\tl = %ld \n",l);
          for (u=1;u<=nmu;u++)
          printf("a[%3d]=%10lf, c[%3d]=%10lf\n",u,a[w][l][u],u,c[w][l][u]);
          printf("\n");
        }
      }
    }
  }

  free_dvector(eratio,1);
```

```
}

/* step

      A SUBROUTINE FOR transport.

      REMINDER OF NOTATION:

        z = ARCLENGTH ALONG MAGNETIC FIELD

        mu = COSINE OF THE PITCH ANGLE

        w = INDEX FOR v-GRID POINTS, FROM 1 TO np

        l = INDEX FOR z-GRID POINTS, FROM 1 TO nz[w]
            z = (l-0.5) * zstep[w]

        u = INDEX FOR mu-GRID POINTS, FROM 1 TO nmu
            mu = (u-0.5) * mustep - 1.0

        i = INDEX FOR mu-GRID POINTS, FROM -(nmu-1)/2 TO +(nmu-1)/2
            mu = i * mustep

      THIS ROUTINE USES THE ALTERNATING DIRECTION IMPLICIT METHOD,
      BUT INSTEAD OF DIFFERENCING IN THE z DIRECTION, EACH GRID
      POINT IS MOVED FROM l TO l + i, WHICH CORRESPONDS TO MOVING
      z TO z + i*zstep = z + mu*timestep (zstep=mustep*timestep).
*/

void      step(time,timestep,beta,m,np,p,nz,zstep,nmu,vsw,printextra)
double    time, timestep, *beta, m, *p, *zstep, vsw;
int       np, nmu, printextra;
long      *nz;
{
   double    *coarse, *fine, *fl, *swap;
   double    average, asym, betasw;
```

```
int     abcneg, allneg, allzero, bad, loop, n, neg, ntotal, u, w;
long    l, lprint;

void    decel(), stream();
void    nrerror(), tridag();

/* A NOTE ON POINTER USAGE: THERE ARE 5 ARRAYS USED TO
   STORE f VALUES:

   f[1..np][1..nz][1..nmu] - MAIN STORAGE OF f(mu,z,v)
   h[1..nmu] — AN ESTIMATE OF THE UPDATED f(mu), FOR A FIXED z, v
   h3[1..nmu] —
   g[1..nmu] — STORAGE FOR SWAPPING WITH h AND h3
   fint[1..nmu] - Temporary storage for new estimate, which
                  will be rejected if any elements are negative.

   IN ADDITION, FOUR MORE POINTERS HAVE BEEN DEFINED, WHICH
   ARE ASSIGNED TO PARTS OF THE FIXED STORAGE:

   fl = f[w][l] - POINTS TO f(mu) FOR z = (i-.5)*zstep, v(w)
   coarse —— POINTS TO THE COARSER ESTIMATE (h OR h3)
   fine ——— FINER
   swap ——— FOR SWAPPING coarse AND fine

   aa[u], bb[u], and cc[u] contain matrix elements for a fixed
   z and v.  an[u], bn[u], and cn[u] are the same, but without
   focusing - they are used when focusing leads to negative
   fluxes.
*/

/* EXECUTE ONE STEP:

   1) EXPLICIT SCATTERING AND FOCUSING OF f(u) FOR EACH l
      (EXPLICITLY IN mu).

   2) IMPLICIT SCATTERING AND FOCUSING OF f(u) FOR EACH l
```

```
        (IMPLICITLY IN mu)

      3) PERFORM THE ABOVE 3 TIMES AS MANY TIMES, FOR ONE THIRD
         THE sstep, USING SMALLER STEPS UNTIL THE DIFFERENCE IS
         BELOW "TOLERANCE".

      4) MOVE f(l,u) TO f(l+i,u).

      5) REPEAT STEPS 1, 2, AND 3.
   */

if (printextra) printf("\n NUI IN STEP");
for (loop=1;loop<=2;loop++) {
   if (printextra) printf("  loop #%d\n",loop);
   for (w=1;w<=np,w++) {

      lprint = LPRINT;
      if (lprint <= 0 || lprint >= nz[w]) lprint = nz[w]/4;

         /* Perform streaming IN BETWEEN THE
            TWO PITCH-ANGLE DIFFERENCING STEPS
         */

      if (loop == 2) {
         betasw = vsw / C;
         stream(time,timestep,beta,w,lprint,nz[w],zstep[w],nmu,betasw,printextra);
      }

      for (l=1;l<=nz[w];l++) {
         fl  = f[w][l];

            /* FIND OUT IF ALL f![u]'S ARE ZERO (or below TINY). */

         allzero = 1;
         for (u=1;allzero && u<=nmu;u++) {
            if (fl[u] < 0) {
```

```
        printf("time = %lf, w = %d, l = %ld, u = %d\n",time,w,l,u);
        nrerror("step: neg # at start of step");
    }
    allzero = fl[u] < TINY;
}
if (allzero && printextra && (w==1 || w==np) && l % lprint == 1)
    printf("\nl = %12ld: all entries are 0.0",l);

    /* ONLY PROCEED IF allzero IS FALSE. */

if (!allzero) {

      /* Initialization for the loop. Note that the ntotal=1
         result is first compared with the current f - if the
         difference is below TOLERANCE, this value is
         accepted. Otherwise, ntotal is tripled and a and c
         are decreased by a factor of 2 until the result
         converges to within TOLERANCE. Bomb out if ntotal
         > 1000.
      */

    for (u=1;u<=nmu;u++) {
      aa[u] = a[w][l][u];
      cc[u] = c[w][l][u];
    }
    an[1] = 0.0;
    for (u=1;u<=nmu-1;u++) {
      an[u+1] = (aa[u+1] + cc[u]) / 2.0;
      cn[u]   = (aa[u+1] + cc[u]) / 2.0;
    }
    cn[nmu] = 0.0;
    bb[1]   = 1.0 - aa[2];
    bn[1]   = 1.0 - an[2];
    for (u=2;u<=nmu-1;u++) bb[u] = 1.0 - aa[u+1] - cc[u-1];
    for (u=2,u<=nmu-1;u++) bn[u] = 1.0 - an[u+1] - cn[u-1];
    bb[nmu] = 1.0 - cc[nmu-1];
```

```
bn[nmu] = 1.0 - cn[nmu-1];
abcneg  = 1;

fine   = h;
coarse = h3;
for (u=1;u<=nmu;u++) coarse[u] = fl[u];
ntotal = 1;

  /* DO STEPS 1) AND 2) ntotal TIMES. */

for (;;) {

    /* Check if -an, 2-bn, -cn < 0.  If so,
       immediately jump to increase ntotal.
    */

  if (abcneg) for (u=1,abcneg=0;!abcneg && u<=nmu;u++)
    abcneg = an[u] > 0 || 2.0-bn[u] < 0 || cn[u] > 0;

  if (!abcneg) {
    if (printextra && (w==1 || w==np) && l % lprint == 1)
      printf("\nl=%12ld (%d)",l,ntotal);
    for (u=1;u<=nmu;u++) fine[u] = fl[u];

    for (n=1,allneg=1;n<=ntotal;n++) {

        /* EXPLICIT DIFFERENCING IN mu   fine -> g.  */

      g[1] = (2.0-bb[1])*fine[1]-cc[1]*fine[2];
      for (u=2;u<=nmu-1;u++) g[u] = -aa[u]*fine[u-1]
        +(2.0-bb[u])*fine[u]-cc[u]*fine[u+1];
      g[nmu] = -aa[nmu]*fine[nmu-1]+(2.0-bb[nmu])*fine[nmu];

        /* IMPLICIT DIFFERENCING IN mu.

           tridag SOLVES THE MATRIX EQUATION M(i,j) x(j) =
```

```
      y(i), FOR A TRIDIAGONAL M.  THE FIRST 3 ARGUMENTS
      POINT TO MATRIX ELEMENTS (TO M(u,u-1)'S,
      M(u,u)'S, AND M(u,u+1)'S), THE FOURTH ARGUMENT
      POINTS TO y, THE FIFTH POINTS TO x, AND THE LAST
      IS THE DIMENSION OF THE MATRICES AND VECTORS.

      g -> fint.
   */

tridag(aa,bb,cc,g,fint,nmu);

   /* Set fluxes between -TINY and zero to zero.
      Check for negative numbers.  If they appear,
      use matrix elements with the effect of focusing
      removed (an[u], bn[u], and cn[u]).  Bomb out
      if negative numbers persist.  Do not allow
      the iteration to end if each substep yielded
      negative numbers.
   */

for (u=1;u<=nmu;u++)
   if (-TINY < fint[u] && fint[u] < 0) fint[u] = 0.0;
for (u=1,neg=0;!neg && u<=nmu;u++) neg = fint[u] < 0.0;
if (!neg) allneg=0;

if (printextra && (w==1 || w==np) && l%lprint==1
    &&  neg) printf(" -");
if (printextra && (w==1 || w==np) && l%lprint==1
    && !neg) printf(" +");

if (!neg) {
   for (u=1;u<=nmu;u++) fine[u] = fint[u];
} else {
   g[1] = (2.0-bn[1])*fine[1]-cn[1]*fine[2];
   for (u=2;u<=nmu-1;u++) g[u] = -an[u]*fine[u-1]
      +(2.0-bn[u])*fine[u]-cn[u]*fine[u+1];
```

```
        g[nmu] = -an[nmu]*fine[nmu-1]
           +(2.0-bn[nmu])*fine[nmu];

        tridag(an,bn,cn,g,fine,nmu);

        for (u=1,neg=0;!neg && u<=nmu;u++)
           neg = fine[u] < 0.0;
        if (neg) {
           printf("time = %lf, w = %d, l = %ld\n",time,w,l);
           nrerror("step: neg after special treatment");
        }
      }
    }

    average = 0.0;
    for (u=1;u<=nmu;u++) average += fine[u];
    average /= nmu;

    for (u=1,bad=0;!bad && u<=nmu;u++)
      bad = fabs(coarse[u] - fine[u]) > ATOL
        && fabs(coarse[u] - fine[u]) > FTOL*average;
    if (!bad && !allneg) break;

    swap   = fine;
    fine   = coarse;
    coarse = swap;
  }

  ntotal *= 2;
  if (ntotal > 1000) {
    printf("Bombing out! ");
    printf("time=%lf, w=%d, l=%ld\n",time,w,l);
    for (u=1,u<=nmu;u++) printf("fl[%d]=%lf\n",u,fl[u]);
    nrerror("step: ntotal > 1000");
  }
  for (u=1;u<=nmu;u++) {
```

```
                aa[u] /= 2.0;
                an[u] /= 2.0;
                bb[u]  = 1.0 + (bb[u] - 1.0) / 2.0;
                bn[u]  = 1.0 + (bn[u] - 1.0) / 2.0;
                cc[u] /= 2.0;
                cn[u] /= 2.0;
              }
            }

              /* MOVE BEST ESTIMATE, fine[u], TO fl[u] */

            for (u=1;u<=nmu;u++) fl[u] = fine[u];
          }
        }
      }

      /* Deceleration for all w values at once, in between the
         two pitch-angle differencing steps.
      */

    if (loop == 1) {
      decel(timestep,np,p,lprint,nz,zstep,nmu,vsw,printextra);
    }
  }
}
```

Appendix B
Program For Fitting

```
/* least2 - July 22, 1995

   Modified as follows:

   Requires user input of the background level (in same units as fitdata.dat)
   Background level is subtracted from data (y) - unc. of back. is neglected

   Note: user should delete any lines where the data are zero or suspected
   to be purely background.
*/

#include <stdio.h>
#include <math.h>
static   double sqrarg;
#define  SQR(a) (sqrarg=(a),sqrarg*sqrarg)
#define  SWAP(a,b) {double temp=(a); (a)=(b); (b)=temp;}

double   **afunc;
main()
{
   FILE     *fp_d, *fp_f;
   double   *a, back, chisq, **covar,
   double   *x, *y, *sig;
/*    static int      lista[MA+1]=LISTA; */
   int      i, j, k, *lista, ma, mmfit, ndata;

   double   *dvector(), **dmatrix();
   int      *ivector();
```

```
    void      free_dvector(), free_dmatrix(), free_ivector(), lfit();

    printf("\nNumber of x,y: ",ndata);
    scanf("%d",&ndata);
    printf("\nNumber of fitting functions: ",ma);
    scanf("%d",&ma);
    printf("\nBackground level (same units as data): ",back);
    scanf("%lf",&back);

    covar = dmatrix(1,ma,1,ma);
    afunc = dmatrix(1,ma,1,ndata);

printf("Reading fitfunc.dat...ndata=%d,ma=%d\n",ndata,ma);
    fp_f = fopen("fitfunc.dat","r");
    for (i=1;i<=ndata;i++) {
       for(k=1;k<=ma;k++){
            /*
                printf(" input afunc[%d][%d] = ",k,i);
*/
                fscanf(fp_f,"%lf",&afunc[k][i]);
       printf("afunc = %lf\n",afunc[k][i]);
       }
    }
    fclose(fp_f);

    x     = dvector(1,ndata);
    y     = dvector(1,ndata);
    sig   = dvector(1,ndata);
    a     = dvector(1,ma);

printf("Reading fitdata.dat...");
    fp_d = fopen("fitdata.dat","r");
    for(i=1,i<=ndata,i++){
/*      printf("input x[%d] = ",i); */
       fscanf(fp_d,"%lf", &x[i]);
```

```
/*      printf("input y[%d] = ",i); */
        fscanf(fp_d,"%lf", &y[i]);
        y[i] -= back;
/*      printf("input sig[%d] = ",i); */
        fscanf(fp_d,"%lf", &sig[i]);
    }
    fclose(fp_d);

    lista = ivector(1,ma);
    for (k=1;k<=ma;k++) lista[k] = k;
    mmfit = ma;

printf("mmfit=%d\n",mmfit);
    lfit(x, y, sig, ndata, a, ma, lista, mmfit, covar, &chisq);

    free_dvector(x,1);
    free_dvector(y,1);
    free_dvector(sig,1);
    free_dvector(a,1);
    free_ivector(lista,1);
    free_dmatrix(afunc,1,ma,1);
    free_dmatrix(covar,1,ma,1);
}

void    lfit(x, y, sig, ndata, a, ma, lista, mmfit, covar, chisq)
int     ndata, ma,  mmfit, *lista;
double  *x, *y, *sig, *a, **covar, *chisq;
{
    int     k, kk, j, ihit, i;
    double  ym, wt, sum, sig2i, **beta;
    void    gaussj(), covsrt(), nrerror(), free_dmatrix();
    double  **dmatrix();

printf("Entering lfit; mmfit = %d...\n",mmfit);
    beta  = dmatrix(1,ma,1,1);
    kk    = mmfit+1;
```

```
    for(j=1;j<=ma;j++) {
      ihit = 0;
      for(k=1;k<=mmfit;k++) {
          if(lista[k] == j)  ihit++;
/* printf("mmfit=%d, ihit=%d, k=%d\n",mmfit,ihit,k); */
      }
      if(ihit == 0) {
          lista[kk++] = j;
printf("ihit=%d\n",ihit);
      }
      else if(ihit>1)  nrerror("Bad LISTA permutation in LFIT-1");
    }
    if(kk != (ma+1))  nrerror("Bad LISTA permutation in LFIT-2");
    for(j=1;j<=mmfit;j++) {
      for(k=1;k<=mmfit;k++)  covar[j][k]=0.0;
      beta[j][1] = 0.0;
    }
printf("A\n");
    for(i=1;i<=ndata;i++) {
      ym = y[i];
      if(mmfit < ma)
          for(j=(mmfit+1);j<=ma;j++)
            ym -= a[lista[j]]*afunc[lista[j]][i];
      sig2i = 1.0/SQR(sig[i]);
      for(j=1;j<=mmfit;j++) {
          wt = afunc[lista[j]][i]*sig2i;
          for(k=1;k<=j;k++)
            covar[j][k] += wt*afunc[lista[k]][i];
            beta[j][1] += ym*wt;
      }
    }
printf("B\n");
    if(mmfit > 1)
      for(j=2;j<=mmfit;j++)
          for(k=1;k<=j-1;k++)
            covar[k][j] = covar[j][k];
```

```
printf("C\n");
    printf("ma=%d\n",ma);
   for(i=1;i<=ma;i++){
       for(j=1;j<=ma;j++){
            printf("%lf",covar[i][j]);
       }
       printf("\n");
   }
   gaussj(covar,mmfit,beta 1);
printf("D\n");
   for(j=1;j<=mmfit;j++) {
      a[lista[j]] = beta[j][1];
      printf("\n a[lista[%d]] = %le",j,a[lista[j]]);
   }
   *chisq = 0.0;
   for(i=1;i<=ndata;i++) {
      for(sum=0,j=1;j<=ma;j++)  sum += a[j]*afunc[j][i];
      *chisq += SQR((y[i]-sum)/sig[i]);
   }
   printf("\n chi_square = %lf ",*chisq);

         /* print covar[][] */

  printf("\n\n");
  for(i=1;i<=ma;i++){
     for(j=1;j<=ma;j++){
          printf("%12lf    ",covar[i][j]);
     }
     printf("\n");
  }

  covsrt(covar,ma,lista,mmfit);

         /* print covar[][] */

  printf("\n\n");
```

```
    for(i=1;i<=ma;i++){
        for(j=1;j<=ma;j++){
                printf("%12le    ",covar[i][j]);
        }
        printf("\n");
    }

    free_dmatrix(beta,1,ma,1,1);
printf("E\n");
}

void    covsrt(covar,ma,lista,mmfit)
double  **covar;
int     ma , lista[], mmfit ;
{
    int i, j;
    double swap;

    for(j=1;j<=ma;j++)
        for(i=j+1;i<=ma;i++)  covar[i][j] = 0.0;
    for(i=1;i<=mmfit;i++)
        for(j=i+1;j<=mmfit;j++) {
                if( lista[j] > lista[i] )
                    covar[lista[j]][lista[i]] = covar[i][j];
                else
                    covar[lista[i]][lista[j]] = covar[i][j];
        }
    swap = covar[1][1];
    for(j=1;j<=ma;j++) {
        covar[1][j] = covar[j][j];
        covar[j][j] = 0.0;
    }
    covar[lista[1]][lista[1]] = swap;
    for(j=2;j<=mmfit;j++)  covar[lista[j]][lista[j]] = covar[1][j];
    for(j=2;j<=ma;j++)
        for(i=1;i<=j-1;i++)  covar[i][j] = covar[j][i];
```

```
}

void  nrerror(error_text)
char  error_text[];
{
    void  exit();

    fprintf(stderr,"Numerical Recipes run-time error...\n");
    fprintf(stderr,"%s\n",error_text);
    fprintf(stderr,"...now exiting to system...\n");
    exit(1);
}

void   free_dvector(v,nl)
double *v;
int    nl;
{
   free((char*) (v+nl));
}

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

void   gaussj(a,n,b,m)
double **a, **b;
int    n, m;
{
   int    *indxc,*indxr,*ipiv;
   int    i,icol,irow,j,k,l,ll,*ivector();
```

```
void   nrerror(), free_ivector();
double big, pivinv, dum,

indxc = ivector(1,n);
indxr = ivector(1,n);
ipiv  = ivector(1,n);
for(j=1;j<=n;j++)  ipiv[j] = 0;
for(i=1;i<=n;i++)  {
   big = 0.0;
   for(j=1;j<=n;j++)
         if(ipiv[j] !=1)
            for(k=1;k<=n;k++) {
               if(ipiv[k]==0) {
                     if(fabs(a[j][k]) >= big) {
                        big = fabs(a[j][k]);
                        irow=j;
                        icol=k;
                     }
               } else if (ipiv[k] > 1) nrerror("GAUSSJ: Singular Matrix-1");
            }
         ++(ipiv[icol]);
         if (irow != icol) {
            for(l=1;l<=n;l++)  SWAP(a[irow][l],a[icol][l])
            for(l=1;l<=m;l++)  SWAP(b[irow][l],b[icol][l])
         }
         indxr[i]=irow;
         indxc[i]=icol;
         if (a[icol][icol] == 0.0) nrerror("GAUSSJ: Singular Matrix-2");
         pivinv=1.0/a[icol][icol];
         a[icol][icol] = 1.0;
         for(l=1;l<=n;l++) a[icol][l] *= pivinv;
         for(l=1;l<=m;l++) b[icol][l] *= pivinv;
         for(ll=1;ll<=n;ll++)
            if (ll != icol) {
               dum = a[ll][icol];
               a[ll][icol] = 0;
```

```
                for(l=1,l<=n;l++) a[ll][l] -= a[icol][l]*dum;
                for(l=1;l<=m;l++) b[ll][l] -= b[icol][l]*dum;
            }
        }
        for(l=n;l>=1;l--) {
            if (indxr[l] != indxc[l])
                for(k=1;k<=n;k++)
                    SWAP(a[k][indxr[l]],a[k][indxc[l]]);
        }

            /* print covar[][] */

    printf("\n\n");
    for(i=1;i<=n;i++){
        for(j=1;j<=n,j++){
            printf("%12lf    ",a[i][j]);
        }
        printf("\n");
    }
        free_ivector(ipiv,1,n);
        free_ivector(indxr,1,n);
        free_ivector(indxc,1,n);
}

void free_ivector(v,nl)
int  *v,nl;
{
    free((char*) (v+nl));
}

double *dvector(nl,nh)
int nl,nh;
{
    double *v;
    v = (double *)malloc((unsigned) (nh-nl+1)*sizeof(double));
    if (!v) nrerror("allocation failure in dvector()");
```

```
    return v-nl;
}

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

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

Curriculum Vitae

Miss Thiranee Khumlumlert was born on December 27, 1970 in Bangkok. She received her B.Sc. (2nd Class Honor) degree in physics from Naresuan University in 1992.