References

Decker, R. B. Formation of Shock-Spike Events at Quasi-Perpendicular Shocks. Journal of Geophysical Research **88** (1983), 9959-9973.

De Hoffmann, F. and Teller, E. Magneto-Hydrodynamic Shocks. Physical Review **80** (1950), 692-703.

Krasnov, N. F. Aerodynamics 1. Mir Publishers Moscow 1985, 13-16.

Lee, M. A. Coupled Hydrodynamic Wave Excitation and Ion Acceleration at Interplanetary Traveling Shocks. Journal of Geophysical Research **88** (1983), 6109-6119.

Lüst, R. Magneto-Fluid Dynamic Shock Waves. Reviews of Modern Physics **32** (1960), 706-709.

Morrison, P. The Origin of Cosmic Rays. Encyclopedia of Physics **XLVI/1** 1961, 10-38.

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

Parks, G. K. Physics of Space Plasma Addison-Wesley Publishing Company 1991, 2-3.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. Numerical Recipes in C Second Edition Cambridge University Press 1992, 856-857.

Rajamäki, M. and Saarinen, M. Accurate One-Dimensional Computation of Frontal Phenomena by PLIM. Journal of Computational Physics **111** (1994), 62-73.

Richardson, I. G., Cane, H. V. and von Rosenvinge, T. T. MeV Ion Anisotropies at Interplanetary Shocks: Observations from the ISEE-3/ICE Medium Energy Cosmic Ray Experiment. Proceedings of the 21th International Cosmic Ray Conference 1990, 333-336.

Ruffolo, D. Interplanetary Transport of Decay Protons from Solar Flare Neutrons. Astro-

physical 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. Comparative Analysis of Cosmic Rays from Three Types of Solar Flares. Final Report 1996, 1-3.

Ruffolo, D. and Khumlumlert, T. Formation, Propagation, and Decay of Coherent Pulses of Solar Cosmic Rays. Geophysical Research Letters **22** (1995), 2073-2076.

Ruffolo, D. and Khumlumlert, T. Propagation of Coherent Pulses of Solar Cosmic Rays. Proceedings of the 24th International Cosmic Ray Conference 1995.

Zauderer, E. Partial Differential Equations of Applied Mathematics, Second Edition, John Wiley & Sons 1989, 48-58.

Appendix A
Operator Splitting Method

This appendix is derived from Numerical Recipes in C of Press et al(1992). The basic idea of operator splitting or time splitting is that: Suppose we have an initial value problem in form

$$\frac{\partial F}{\partial t} = \mathcal{L}F \tag{1}$$

where $\mathcal{L}$ is some operator not necessarily linear. Suppose that it can be written as a linear sum of m pieces as

$$\mathcal{L}F = \mathcal{L}_1 F + \mathcal{L}_2 F + \ldots + \mathcal{L}_m F \tag{2}$$

Suppose that for each of the pieces, we already know how to update F from timestep n to timestep $n+1$, valid as if that only operator were on the right-hand side:

$$\begin{aligned} \mathcal{L}F &= \mathcal{L}_1 F \\ \mathcal{L}F &= \mathcal{L}_2 F \\ &\ldots \\ \mathcal{L}F &= \mathcal{L}_m F, \end{aligned} \tag{3}$$

we will write these updatings symbolically as

$$\begin{aligned} F^{n+1} &= \mathbf{F_1}(F^n, \Delta t) \\ F^{n+1} &= \mathbf{F_2}(F^n, \Delta t) \\ &\ldots \\ F^{n+1} &= \mathbf{F_m}(F^n, \Delta t), \end{aligned} \tag{4}$$

respectively.

Now, one form of operator splitting would be to get from n to $n+1$ by the following sequence of updatings:

$$\begin{aligned} F^{n+1/m} &= \mathbf{F_1}(F^n, \Delta t) \\ F^{n+2/m} &= \mathbf{F_2}(F^{n+1/m}, \Delta t) \\ &\ldots \\ F^{n+1} &= \mathbf{F_m}(F^{n+(m-1)/m}, \Delta t) \end{aligned} \tag{5}$$

For example, a combined advective-diffusion equation, such as

$$\frac{\partial F}{\partial t} = -v\frac{\partial F}{\partial x} + D\frac{\partial^2 F}{\partial x^2} \tag{6}$$

might use an explicit scheme for the advective term, obtaining

$$F_i^{n+1} = F_i^{n-1} - \frac{v\Delta t}{\Delta z}(F_{i+1}^n - F_{i-1}^n) \tag{7}$$

combined with a Crank-Nicholson or other implicit scheme for the diffusion term, obtaining

$$2(1+\varepsilon)F_i^{n+1} - \varepsilon(F_{i-1}^{n+1} + F_{i+1}^{n+1}) = 2(1-\varepsilon)F_i^n + \varepsilon(F_{i-1}^n + F_{i+1}^n) \tag{8}$$

where $\varepsilon = D\Delta t/(\Delta z)^2$.

then F^{n+1} can be obtained easily from eqs. (7) and (8) by following the sequence of updatings in eq. (5).

Appendix B
Program Usage

This work needs 9 files:

wind.c contains main routine for simulating the transport of cosmic rays. It gets input data from the user then calls routines in other files to do their jobs.

decel.c is used to update $F(t, \mu, z, p)$ for the effects of deceleration.

field.c contain routines that depend on the configuration of the interplanetary magnetic field. In this work magnetic field is constants but different both upstream and downstream regions. It takes into account the change of magnetic field at the shock.

initial.c is used to determine the initial values of the distribution function then send these values to the main routine in wind.c.

inject.c calculates the injection of the particles after the start of simulation.

nrutil.c contains routines to allocate and unallocate memory for variables in running program.

printout.c prints the data that are computed in the program.

stream.c updates F for the streaming and convection of cosmic ray particles. In the thesis stream.c plays important roles in transforming particles across the shock.

These files will be compiled and linked altogether, then the program will be run and it needs input data. The user must key in the following input data:

- initial value of t
- final value of t

- step size
- range for printing
- number of μ points
- length of the simulation region
- number of momentum points
- momentum (in MeV/c)
- particle mass
- solar wind speed divided by c
- scattering mean free path
- scatterng power law index
- whether to print extra diagostic information(0=no, 1=yes)

Appendix C
Source Code for Simulation

```
/*
  s_k_refl.c --- Febuary 24th, 1997

  reflecting at boundary

  s_k_noinf.c -- February 10th, 1997

     Boundary conditions: no flux inflow
     Given f = 1 at one point, we observe its behavior at
     any points before and after crossing the shock.

  s_k_inf.c (v) -- February 5th, 1997

      Boundary conditions: inflow of 1.
```


```
      Songklod Riyavong and David Ruffolo
      Department of Physics
      Faculty of Science
      Chulalongkorn University
      Bangkok 10330, Thailand
```

s_kink.c (v) -- November 18th, 1996

"Stationary" shock at zlength/2. BETASWD and BETASWU are defined as the (constant) solar wind speeds over C *parallel* to the magnetic field.

s_abs.c (t) -- July 6th, 1995

Change s to time.

Change sstep to timestep.

Change Beta[w] to 1/C at r1 and r2.

Add constant C to program by define.

s_abs.c -- March 14th, 1994

Removed mu transformations, occasionally skip extra steps instead.

stream.abs.c -- February 27th, 1994

Reworked the mu transformation to account for differential convection.

stream.abs.c -- July 15th, 1993

Inserted a missing factor of mustep. Reorganized the mu transformation so that the first transformation is explicit, the second is implicit. If the latter generates negative numbers,

explicit, upwind differencing is used instead.

stream.abs.c -- February 8th, 1993

Eliminated slide() and corr[w]. Now working with z defined in a corotating frame.

Now before streaming, need to make a transformation (in mu) from the local solar wind frame to the corotating frame. Need to transform back after streaming...

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

stream.abs.c -- November 2nd, 1992

Brought back tosun[w] to keep track of the total density that streams past the inner boundary for a given w.

stream.w.abs.c -- October 29th, 1992

Added a subroutine, slide, to occasionally move f(l) to f(l+1) and to calculate corr[w], so that z in the lab frame is (l-1+corr[w])*zstep...

```
    stream.w.abs.c -- September 29th, 1992

       Modified for the program, wind.

    stream.abs.c -- October 29th, 1990

       A SUBROUTINE FOR transport.

       MOVES f(i,l) TO f(i,l+i).

       ASSUMES ABSORBING BOUNDARY CONDITIONS AT z = 0 AND z = length.
*/

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


#define   C           0.1202  /* speed of light in AU/min. */
#define   BDOVERBU    1.5     /* B downstream / B upstream */
#define   BETASWD     0.0005  /* solar wind speed / C along B downstream
                                                    (z < length/2.0)
#define   BETASWU     0.002   /* solar wind speed / C along B upstream
                                 (z > length/2.0).
                                 Should have BETASWU > BETASWD.      */
#define   GAMMAL      2.0     /* Power-law exponent below p[1]. */
#define   TINY        1e-6
```

```
extern double   ***f, **fs, *tosun;

void      stream(time,timestep,beta,m,np,p,lprint,nz,zstep,nmu,printextra)
double    time, timestep, *beta, m, *p, *zstep;
int       nmu, np, printextra;
long      lprint, *nz;
{
   double   calc_pd(),calc_pu(),mustar1(),refl_pr(),mu_refl();
   double   corr, dpudpd, dpudpr, fpd, fpu, frac, ftemp;
   double   lnp0, lnp1, lower, mu, mucent, mulower, muminus;
   double   munew, muplus=1.0, mustard, mustarr, mustaru, mustep;
   double   muupper, pd, pu, r1, r2, z, expon, pr;
   int      flagm, flagp=1, i, imin, parttrans, u, w, unew;
   long     k, k1, k2, l, l1, lnew;

   double   betaswc(), invert_to_pu(), invert_to_pd(), mu_d(), mu_star();
   double   mu_u();
   double   **temp;

   mustep = 2.0/(double)nmu;

   for (w=1;w<=np;w++) {
      if (w > 1) {
         lnp0 = log(p[w-1]);
```

```
        lnp1 = log(p[w]);
    }

    for(u=1;u<=nmu;u++) for(l=1;l<=nz[w];l++) fs[l][u] = 0.0;

    imin   = -(nmu-1)/2;

    /* constant inflow = 0 */
   /*
    for (u=1,i=imin;u<=nmu;u++,i++){
          if(i < 0)
             for (l=nz[w]+1+i;l<=nz[w];l++) fs[l][u] = 0.0;
          if(i > 0)
             for (l=1;l<=i;l++) fs[l][u] = 0.0;
     }
  */

   for (u=1,i=imin;u<=nmu;u++,i++) {
      mu   = 2.0*i/(double)nmu;
      corr = 1 - mu*mu*beta[w]*beta[w];
      for (l=1;l<=nz[w];l++) {
if (f[w][l][u] > 0.0) {
            z    = (l-nz[w]/2.0-0.5) * zstep[w];
            r1   = (betaswc(z)*C)*((time+timestep)/zstep[w]);
            r2   = (betaswc(z)*C)*(time/zstep[w]);
            k1   = corr * r1 + 0.000001;
            k2   = corr * r2 + 0.000001;
```

```
            k     = k1 - k2;

      lnew = l + i + k;

      /* reflection at left boundary */
      if (lnew < 1) {
fs[1-lnew][nmu+1-u] += f[w][l][u];
      }
      /* reflection at right boundary */
      else if ( lnew > nz[w] ) {
fs[2*nz[w]+1-lnew][nmu+1-u] += f[w][l][u];
      }

                /* If particles pass the inner boundary ("crash into the
                   Sun") then flux is lost and added to tosun[w].
                */

    /* if (lnew < 1) {
tosun[w] += f[w][l][u];*/

                /* If particles cross from upstream to downstream,
                   take special account of changes in mu and p.
                */

    /* } */ else if (l > nz[w]/2 && lnew <= nz[w]/2) {

                mustaru = mustar1(m,p[w],&mustard,&mustarr);
```

```
                printf("mustaru=%lf\n",mustaru);

                   /* Case where all particles in [mu-mustep/2.0,
                      mu+mustep/2.0] are transmitted, or some
                      are transmitted and some are reflected.

                      This loop will handle the transmitted flux.
                   */

  if (mu - mustep/2.0 < mustaru) {
                   parttrans = mu + mustep/2.0 > mustaru;
                   if (parttrans) {
                      mucent = (mustaru + mu - mustep/2.0) / 2.0;
                   } else {
                      mucent = mu;
                   }
                   pd = calc_pd(m,p[w],mucent,&dpudpd);
                   printf("\n pd = %lf",pd);

                   if (pd < p[w] - TINY)
                      nrerror("stream: anti-acceleration");

                   /* Find fpu, the z-interpolated value of f at (t,z,pu) */
                   /* printf("\n pu = %lf p[w] = %lf",pu,p[w]); */

    if (w > 1) {
printf("\n I am in loop w>1 for transmission");
```

```
li = z/zstep[w-1] + nz[w-1]/2.0 + 0.5;
frac = z/zstep[w-1] + nz[w-1]/2.0 + 0.5 - (double)li;

if (li < 1 || li+1 > nz[w+1]) {   /* cannot interpolate */
   lower = 0.0;
} else {                          /* can interpolate */
   if (f[w-1][li][u] == 0.0 || f[w-1][li+1][u] == 0.0) {
      lower = 0.0;
   } else {
      lower = (1.0-frac)*f[w-1][li][u]
            + frac*f[w-1][li+1][u];
   }
}

   /* fpu is F(p[w],mucent,zold) * pow(2.0-pd/p[w],expon),
      where pd is the momentum downstream
      corresponding to p[w] upstream.
      ftemp is fpu*d(pu)/d(pd).  This will be split into
      different mu-cells according to the fraction
      of the interval [mu_u - mustep/2,mu_u + mustep/2]
      which maps into different cells.
   */

if (lower <= 0.0) {
   if (printextra && l%lprint==1 && w==np)
      printf("\nNote: f[%d][%d][%d] = %le, lower = 0",
         w,l,u,f[w][l][u]);
```

```
                fpu = f[w][l][u];
            } else {
                expon = (log(f[w][l][u])-log(lower))
                        / (lnpi-lnp0);
                fpu = f[w][l][u] * pow(2.0-pd/p[w],expon);
            }

            /* Otherwise, w=1 -> extrapolate from lower p assuming
               f \propto p^-GAMMAL.
            */

        } else {

            /* fpu is F(p_u,mucent,zold), where p_u is the momentum
               which will change to p_d on the downstream side.
               ftemp is fpu*d(p_u)/d(p_d).  This will be split into
               different mu-cells according to the fraction
               of the interval [mu_u - mustep/2,mu_u + mustep/2]
               which maps into different cells.
            */

fpu = f[w][l][u] * pow(2.0-pd/p[w],-GAMMAL);
printf("\n I am out loop w>1 for transmission");
printf("\n total transmitted fpu = %lf",fpu);
                }
                printf("dpudpd=%lf\n",dpudpd);
                ftemp = fpu * dpudpd;
```

```
            printf("\n fpu for transmitt= %lf",fpu);
    printf("\n ftemp for transmitt = %lf",ftemp);


                        /* Will loop over all possible unew and munew
                           grid points on the downstream side, and
                           see how much of ftemp falls in each cell.

                           muminus is munew - mustep/2.0 (lower cell boundary),
                              transformed back upstream
                           muplus is munew + mustep/2.0 (upper cell boundary)
                              transformed back upstream

                           flagm -> 1 if muminus within original cell
                           flagp -> 1 if muplus within original cell

                           original cell means from mu-mustep/2.0 to muupper
                              in the case of partial transmission
                        */

                      muupper = mu+mustep/2.0;
                      if (parttrans) {
/*                          ftemp *= (mustaru-mu+mustep/2.0) / mustep; */
                         muupper = mustaru;
printf("\n fraction = %lf",(mustaru-mu+mustep/2.0) / mustep);
printf("\n fraction of ftemp transmitted = %lf", ftemp);
                      }
```

```
                muminus = -1;
                flagm = muminus >= mu-mustep/2.0
                        && muminus < muupper;
                for (unew=1;unew<=nmu &&
                 (munew=(unew-0.5)*mustep-1.0)-mustep/2.0<=mustard;
                 unew++,muminus=muplus,flagm=flagp) {

                   if (munew+mustep/2.0 >= mustard) {
                      flagp = (muplus = mustaru)
                            >= mu-mustep/2.0 && muplus < muupper;
                   } else {
                      flagp = (muplus = mu_u(m,pd,munew+mustep/2.0))
                            >= mu-mustep/2.0 && muplus < muupper;
                   }
                   printf("unew=%d, muplus=%lf\n",unew,muplus);
                   if (muminus < mu-mustep/2.0
                     && muplus >= muupper) {
                      fs[lnew][unew] += ftemp;
if (lnew>=12 && lnew<=14) printf("UD1 fs[%ld][%d] = %lf\n",lnew,unew,fs[lnew][unew]);
                      break;
                   } else if (!flagm && flagp) {
                      fs[lnew][unew] += ftemp * (muplus-(mu-mustep/2.0))
                                              / mustep;
if (lnew>=12 && lnew<=14) printf("UD2 fs[%ld][%d] = %lf\n",lnew,unew,fs[lnew][unew]);
                   } else if (flagm && flagp) {
                      fs[lnew][unew] += ftemp * (muplus-muminus)
                                              / mustep;
```

```
if (lnew>=12 && lnew<=14) printf("UD3 fs[%ld][%d] = %lf\n",lnew,unew,fs[lnew][unew]);
                    } else if (flagm && !flagp) {
                        fs[lnew][unew] += ftemp * (muupper-muminus)
                                                / mustep;
if (lnew>=12 && lnew<=14) printf("UD4 fs[%ld][%d] = %lf\n",lnew,unew,fs[lnew][unew]);
                        break;
                    }
                }
            }

            /* Case where all particles in [mu-mustep/2.0,
               mu+mustep/2.0] are reflected, or some
               are transmitted and some are reflected.

               This loop will handle the reflected flux.
            */

            if( mu+mustep/2.0 > mustaru ) {

                /* Reflection: lnew = nz[w] + 1 - lnew. */

                lnew = nz[w] + 1 - lnew;
                parttrans = mu-mustep/2.0 < mustaru;

                /* partially, not completely transmitted */

                if( parttrans ) {
```

```
mucent = (mu+mustep/2.0+mustaru)/2.0;
printf("\n partial reflection");
                    } else {
mucent = mu;
printf("\n total reflection");
                    }

                    pr = refl_pr(m,p[w],mucent,&dpudpr);
                    printf("\n pr = %lf",pr);
                    /* Find lower. */

    if (w > 1) {
printf("\n I am in loop w>1 for reflection");
                      l1 = z/zstep[w-1] + nz[w-1]/2.0 + 0.5;
                      frac = z/zstep[w-1] + nz[w-1]/2.0 + 0.5 - (double)l1;

                      if (l1 < 1 || l1+1 > nz[w+1]) {   /* cannot interpolate */
                        lower = 0.0;
                      } else {                          /* can interpolate */
                        if (f[w-1][l1][u] == 0.0 || f[w-1][l1+1][u] == 0.0) {
                          lower = 0.0;
                        } else {
                          lower = (1.0-frac)*f[w-1][l1][u]
                                + frac*f[w-1][l1+1][u];
                        }
                      }
```

```
            /* fpu is F(p[w],mucent,zold) * pow(2.0-pd/p[w],expon),
               where pd is the momentum downstream
               corresponding to p[w] upstream.
               ftemp is fpu*d(pu)/d(pd).  This will be split into
               different mu-cells according to the fraction
               of the interval [mu_u - mustep/2,mu_u + mustep/2]
               which maps into different cells.
            */

         if (lower <= 0.0) {
            if (printextra && l%lprint==1 && w==np)
               printf("\nNote: f[%d][%d][%d] = %le, lower = 0",
                  w,l,u,f[w][l][u]);
            fpu = f[w][l][u];
         } else {
            expon = (log(f[w][l][u])-log(lower))
                     / (lnp1-lnp0);
            fpu = f[w][l][u] * pow(2.0-pr/p[w],expon);
         }

            /* Otherwise, w=1 -> extrapolate from lower p assuming
               f \propto p^-GAMMAL.
            */

      } else {

         /* fpu is F(p_u,mucent,zold), where p_u is the momentum
```

```
                    which will change to p_d on the downstream side.
                    ftemp is fpu*d(p_u)/d(p_d).  This will be split into
                    different mu-cells according to the fraction
                    of the interval [mu_u - mustep/2,mu_u + mustep/2]
                    which maps into different cells.
                  */
printf("\n I am out loop w>1 for reflection");
   fpu = f[w][l][u] * pow(2.0-pr/p[w],-GAMMAL);
printf("\n total reflection fpu = %lf",fpu);
printf("\n partial reflection fpu before = %lf",pow(2.0-pr/p[w],-GAMMAL));
printf("\n partial reflection fpu = %lf",fpu);
                  }

                  printf("dpudpr=%lf\n",dpudpr);
    ftemp = fpu * dpudpr;
    printf("\n ftemp for reflection = %lf",ftemp);

                /* Will loop over all possible unew and munew
                   grid points reflected upstream, and
                   see how much of ftemp falls in each cell.

                   muminus is munew - mustep/2.0 (lower cell boundary),
                      transformed back upstream
                   muplus is munew + mustep/2.0 (upper cell boundary)
                      transformed back upstream

                  flagm -> 1 if muminus within original cell
```

```
                    flagp -> 1 if muplus within original cell

                    original cell means from mulower to mu + mustep/2.0
                       in the case of partial transmission
                  */

                 mulower = mu-mustep/2.0;
                 if (parttrans) {
/*                     ftemp *= (mu+mustep/2.0-mustaru) / mustep; */
printf("\n fraction = %lf",(mu+mustep/2.0-mustaru) / mustep);
printf("\n fraction of ftemp for reflection = %lf",ftemp);
                    mulower = mustaru;
                 }

                 muminus = mu_refl(m,pr,-mustep/2.0);
                 flagm = muminus >= mulower
                         && muminus < mu+mustep/2.0;
                 for (unew=(nmu+1)/2;unew<=nmu &&
                  (munew=(unew-0.5)*mustep-1.0)-mustep/2.0<=mustarr;
                  unew++,muminus=muplus,flagm=flagp) {

                   if (munew+mustep/2.0 >= mustarr) {
                     flagp = (muplus = mustaru)
                           >= mulower && muplus < mu+mustep/2.0;
                   } else {
                     flagp = (muplus = mu_refl(m,pr,munew+mustep/2.0))
                           >= mulower && muplus < mu+mustep/2.0;
```

```
                    }

                       /* Note that muminus > muplus ! */

                    printf("unew=%d, muplus=%lf\n",unew,muplus);
                    if (muplus < mulower
                      && muminus >= mu+mustep/2.0) {
                       fs[lnew][unew] += ftemp;
if (lnew>=12 && lnew<=14) printf("UR1 fs[%ld][%d] = %lf\n",lnew,unew,fs[lnew][unew]);
                       break;
                     } else if (!flagp && flagm) {
                       fs[lnew][unew] += ftemp * (muminus-(mu-mustep/2.0))
                                              / mustep;
if (lnew>=12 && lnew<=14) printf("UR2 fs[%ld][%d] = %lf\n",lnew,unew,fs[lnew][unew]);
                       break;
                    } else if (flagm && flagp) {
                       fs[lnew][unew] += ftemp * (muminus-muplus)
                                              / mustep;
if (lnew>=12 && lnew<=14) printf("UR3 fs[%ld][%d] = %lf\n",lnew,unew,fs[lnew][unew]);
                    } else if (flagp && !flagm) {
                       fs[lnew][unew] += ftemp * (mu+mustep/2.0-muplus)
                                              / mustep;
if (lnew>=12 && lnew<=14) printf("UR4 fs[%ld][%d] = %lf",lnew,unew,fs[lnew][unew]);
                    }
                 }
              }
```

```
/* begin */

      /* If particles cross from downstream to upstream,
         take special account of changes in mu and p.

         All particles are transmitted if they cross
         from downstream to upstream.

         If mustard is the value of mu in the downstream
         solar wind frame for which mu is 0 in the de Hoffmann-
         Teller (shock) frame, then mustard < 0.

         Is there a chance that flux for mu < mustard
         will be transmitted? NO, because grid points
         with mu <= 0 are never transmitted (convection
         moves them farther from the shock).
      */

   } else if (l <= nz[w]/2 && lnew > nz[w]/2) {

      mustaru = mustar1(m,p[w],&mustard,&mustarr); /* return mu_u */

       pu = calc_pu(m,p[w],mu,&dpudpd);
      /* pu = p[w];
       dpudpd = 1.0; */
printf("pu/p[%d]=%lf, dpudpd=%lf\n",w,pu/p[w],dpudpd);
```

```
/* printf("\n pu = %lf p[w] = %lf",pu,p[w]); */


   /* Find fpd, the z-interpolated value of f at (t,z,pd) */

if (pu < p[w] - TINY) nrerror("stream: anti-acceleration");
if (w > 1) {

   l1 = z/zstep[w-1] + nz[w-1]/2.0 + 0.5;
   frac = z/zstep[w-1] + nz[w-1]/2.0 + 0.5 - (double)l1;

   if (l1 < 1 || l1+1 > nz[w+1]) {   /* cannot interpolate */
      lower = 0.0;
   } else {                          /* can interpolate */
      if (f[w-1][l1][u] == 0.0 || f[w-1][l1+1][u] == 0.0) {
         lower = 0.0;
      } else {
         lower = (1.0-frac)*f[w-1][l1][u]
                 + frac*f[w-1][l1+1][u];
      }
   }

      /* fpd is F(p[w],mu,zold) * pow(2.0-pu/p[w],expon),
         where pu is the momentum upstream
         corresponding to p[w] downstream.
         ftemp is fpd*d(p_d)/d(p_u).  This will be split into
         different mu-cells according to the fraction
```

```
            of the interval [mu_d - mustep/2,mu_d + mustep/2]
            which maps into different cells.
          */

        if (lower <= 0.0) {
          if (printextra && l%lprint==1 && w==np)
            printf("\nNote: f[%d][%d][%d] = %le, lower = 0",
              w,l,u,f[w][l][u]);
          fpd = f[w][l][u];
        } else {
          expon  = (log(f[w][l][u])-log(lower)) / (lnp1-lnp0);
          fpd    = f[w][l][u] * pow(2.0-pu/p[w],expon);
        }

          /* Otherwise, w=1 -> extrapolate from lower p assuming
            f \propto p^-GAMMAL.
          */

      } else {

        /* fpd is F(p[w],mu,zold) * pow(2.0-pu/p[w],expon),
          where pu is the momentum upstream
          corresponding to p[w] downstream.
          ftemp is fpd*d(p_d)/d(p_u).  This will be split into
          different mu-cells according to the fraction
          of the interval [mu_d - mustep/2,mu_d + mustep/2]
          which maps into different cells.
```

```
        */

    fpd     = f[w][l][u] * pow(2.0-pu/p[w],-GAMMAL);
}
ftemp = fpd / dpudpd;

muplus = 1;
flagp = muplus > mu-mustep/2.0
        && muplus <= mu+mustep/2.0;
for (unew=nmu;unew>=1 &&
 (munew=(unew-0.5)*mustep-1.0)+mustep/2.0 >= mustarr;
  unew--,muplus=muminus,flagp=flagm) {

  if (munew-mustep/2.0 <= mustarr) {
     muminus = -1;
     flagm = 0;
  } else {
     flagm = (muminus = mu_d(m,pu,munew-mustep/2.0))
           > mu-mustep/2.0 && muminus <= mu+mustep/2.0;
  }
  printf("unew=%d, muminus=%lf\n",unew,muminus);
  if (muminus <= mu-mustep/2.0
    && muplus > mu+mustep/2.0) {
     fs[lnew][unew] += ftemp;
if (lnew>=12 && lnew<=14) printf("DU1 fs[%ld][%d] = %lf\n",lnew,unew,fs[lnew][unew]);
     break;
  } else if (!flagm && flagp) {
```

```
                        fs[lnew][unew] += ftemp * (muplus-(mu-mustep/2.0))
                                                / mustep;
if (lnew>=12 && lnew<=14) {
   printf("DU2 fs[%ld][%d] = %lf\n",lnew,unew,fs[lnew][unew]);
   printf("\t muminus=%lf, muplus=%lf\n",muminus,muplus);
   printf("\t ftemp=%lf, fpd=%lf, dpudpd=%lf\n",ftemp,fpd,dpudpd);
}
                        break;
                     } else if (flagm && flagp) {
                        fs[lnew][unew] += ftemp * (muplus-muminus)
                                                / mustep;
if (lnew>=12 && lnew<=14) printf("DU3 fs[%ld][%d] = %lf\n",lnew,unew,fs[lnew][unew]);
                     } else if (flagm && !flagp) {
                        fs[lnew][unew] += ftemp * (mu+mustep/2.0-muminus)
                                                / mustep;
if (lnew>=12 && lnew<=14) {
   printf("DU4 fs[%ld][%d] = %lf\n",lnew,unew,fs[lnew][unew]);
   printf("\t muminus=%lf, muplus=%lf\n",muminus,muplus);
}
                     }
                  }
               /* end */

                  /* Finally, if particles stay on the same side of the shock,
                     and stay within the simulation region, move to new location.
                  */
```

```
                } else if (lnew <= nz[w]) {
                    fs[lnew][u] += f[w][l][u];
                }
            }
        }
    }

    for (u=1;u<=nmu;u++) {
  printf("\t\tu=%d\n",u);
   for (l=1;l<=nz[w];l++) {
            f[w][l][u] = fs[l][u];
 if (l >= 12 && l <= 14) printf("\t\tf[%d][%ld][%d]=%lf\n",w,l,u,f[w][l][u]);
        }
    }
  }
}


/*
   compute pd and dpd/dpu  when given

    mu_u : cos(psi) in upstream local frame and
     pu : momentum in upstream local frame
*/

double calc_pd(m,pu,mu_u,dpddpu)
```

```
double m, pu, mu_u, *dpddpu;
{
    double Eu, difEu, puper, pupar, pusper, puspar, difpusper, difpuspar;
    double gswu, pus, difpus, mu_us, difmu_us, pds, difpds, mu_ds, difmu_ds;
    double Eds, difEds, pd, difpd, pdpar, pdper, difpdpar, difpdper,gswd;

    /* now we are in upstream local frame
     */

       Eu = sqrt( pu*pu+m*m );
       difEu = pu/Eu;
       puper = pu*sqrt( 1.0-mu_u*mu_u );
       pupar = pu*mu_u;
       gswu = 1.0/sqrt( 1.0-BETASWU*BETASWU );

   /* Transforming from upstream local frame
      to upstream dHT frame ( observer is in
      upstream dHT frame.)
    */

      pusper = puper;
      difpusper = sqrt( 1.0-mu_u*mu_u );
      puspar = gswu*( pupar-BETASWU*Eu );
      difpuspar = gswu*( mu_u-BETASWU*difEu );
      pus = sqrt( puspar*puspar+pusper*pusper );
      difpus = (pusper*difpusper+puspar*difpuspar)/pus;
      mu_us = puspar/pus;
```

```
        difmu_us = (pus*difpuspar-puspar*difpus)/(pus*pus);

      /* Transforming from upstream dHT frame to downstream
         dHT frame
       */

      pds = pus;
      difpds = difpus;
      mu_ds = -sqrt( 1.0 - ( 1.0-mu_us*mu_us )*BDOVERBU );
      difmu_ds = ( mu_us/mu_ds )*BDOVERBU*difmu_us;
      Eds = sqrt( pds*pds + m*m );
      difEds = (pds/Eds)*difpds;

      /* Transforming from downstream dHT frame to downstream
         local frame
      */

      gswd = 1.0/sqrt( 1.0-BETASWD*BETASWD );
      pdpar = gswd*( mu_ds*pds+BETASWD*Eds );
      difpdpar = gswd*( mu_ds*difpds + pds*difmu_ds + BETASWD*difEds );
      pdper = pds*sqrt( 1.0-mu_ds*mu_ds );
      difpdper = (-mu_ds*pds*pds*difmu_ds + (1.0-mu_ds*mu_ds)*pds*difpds)/pdper;
      pd = sqrt( pdpar*pdpar + pdper*pdper);
      difpd = (pdpar*difpdpar + pdper*difpdper)/pd;
     *dpddpu = difpd;
   return pd;
}
```

```
/*
   compute pu: momentum in upstream local frame when given
           pd: momentum in downstream local frame
         mu_d: mu in downstream local frame
 */

double calc_pu( m, pd, mu_d, dpddpu )
double m, pd, mu_d, *dpddpu;
{
   double Ed, difEd, pdpar, difpdpar, pdper, difpdper, pds, difpds, pdspar;
   double difpdspar, pdsper, difpdsper, gswd, mu_ds, difmu_ds, mu_us, difmu_us;
   double pus, difpus, pusper, difpusper, Eus, difEus, gswu, pu, difpu;
   double pupar, difpupar, puper, difpuper;

 printf("\n BETASWD should be less than %lf",mu_d*pd/sqrt(pd*pd+m*m));

/*******************/

/* in downstream local frame
 */

  Ed = sqrt( pd*pd + m*m );
```

```
   difEd = pd/Ed;

   pdpar = mu_d*pd;

   difpdpar = mu_d;

   pdper = pd*sqrt( 1.0-mu_d*mu_d );

   difpdper = sqrt( 1.0-mu_d*mu_d );


/* Transforming from downstream local frame to
   downstream dHT frame
 */


   gswd = 1.0/sqrt( 1.0-BETASWD*BETASWD );

   pdspar = gswd*( pdpar-BETASWD*Ed );

   difpdspar = gswd*( difpdpar-BETASWD*difEd );

   pdsper = pdper;

   difpdsper = difpdper;

   pds = sqrt( pdspar*pdspar + pdsper*pdsper );

   difpds = (pdsper*difpdsper + pdspar*difpdspar)/pds;

   mu_ds = pdspar/pds;

   difmu_ds = (pds*difpdspar-pdspar*difpds)/(pds*pds);


/* It is possible for a particle to recede
   from the shock.
 */

   if(mu_ds <= 0.0)

      printf("\n mu_d = %lf a particle recede from the shock\n",mu_d);


/* Transforming from downstream dHT frame to
```

```
    upstream dHT frame
  */

  mu_us = sqrt( 1.0 - (1.0-mu_ds*mu_ds)/BDOVERBU );
  difmu_us = (mu_ds/mu_us)*(1.0/BDOVERBU)*difmu_ds;
  pus = pds;
  difpus = difpds;
  Eus = sqrt( pus*pus + m*m );
  difEus = (pus/Eus)*difpus;
  pusper = pus*sqrt( 1.0 - mu_us*mu_us );
  difpusper = (-mu_us*pus*pus*difmu_us+(1.0-mu_us*mu_us)*pus*difpus)/pusper;

 /* Transforming from upstream dHT frame to upstream
    local frame
  */

  gswu = 1.0/sqrt( 1.0-BETASWU*BETASWU );
  pupar = gswu*( mu_us*pus + BETASWU*Eus );
  difpupar = gswu*( mu_us*difpus + pus*difmu_us + BETASWU*difEus );
  puper = pusper;
  difpuper = difpusper;
  pu = sqrt( pupar*pupar + puper*puper );
  difpu = (pupar*difpupar + puper*difpuper)/pu;
  *dpddpu = 1.0/difpu;
  return pu;
}
```

```
/*
   check whether a particle in downstream recedes from
   the shock?
 */

int recede( m, pd, mu_d )
double m, pd, mu_d;
{
    double Ed, pdper, pdpar, pdspar, pdsper, pds, mu_ds, gswd;

     /* in downstream local frame
      */

     Ed = sqrt( pd*pd + m*m );
     pdpar = mu_d*pd;
     pdper = pd*sqrt( 1.0 - mu_d*mu_d );

     /* Transforming from downstream local frame to
        downstream dHT frame
      */

     gswd = 1.0/sqrt( 1.0-BETASWD*BETASWD );
     pdspar = gswd*( pdpar - BETASWD*Ed );
     pdsper = pdper;
     pds = sqrt( pdspar*pdspar + pdsper*pdsper );
     mu_ds = pdspar/pds;
```

```
    if ( mu_ds <= 0.0 ) return 1; /* it recedes from the shock */
    if ( mu_ds > 0.0 ) return 0;
}


/*  determine mu_d from pu, mu_u and when

    (a) |mu_u| > 1+TINY report error
    (b) 1-TINY < mu_u < 1+TINY return +1
    (c) -1-TINY < mu_u < -1+TINY return -1.
 */

double  mu_d( m, pu, mu_u )
double  m, pu, mu_u;
{
      double m2, pd, mu_d;
      double Eu, p_us_par, p_us_per, p_us, mu_us, gamma_us;
      double mu_ds, p_ds, p_ds_par, p_ds_per, Eds, gamma_ds;

          m2 = m*m;

       if( fabs(mu_u) > 1.0+TINY )
         nrerror("error in routine mu_d(): mu_u out of bounds");

       else if( 1.0-TINY < mu_u && mu_u < 1.0+TINY )
         return 1.0;
```

```
else if( -1.0-TINY < mu_u && mu_u < -1.0+TINY )
  return -1.0;

else {

/* energy in upstream local frame
 */
      printf("#19 = %lf\n",pu*pu+m2);
      Eu =  sqrt( pu*pu + m2 );

/*  transforming from upstream local frame to
    upstream dHT frame
 */

      printf("#20 = %lf\n",1.0-BETASWU*BETASWU);
      gamma_us = 1.0 / sqrt( 1.0 - BETASWU*BETASWU );
      printf("#21 = %lf\n",pu*pu*(1.0-mu_u*mu_u));
      p_us_per = sqrt( pu*pu*( 1.0-mu_u*mu_u ));
      p_us_par = gamma_us*( pu*mu_u-BETASWU*Eu );
      printf("#22 = %lf\n",p_us_per*p_us_per+p_us_par*p_us_par);
      p_us     = sqrt( p_us_per*p_us_per + p_us_par*p_us_par  );
      mu_us    = p_us_par / p_us;

/* transforming from upstream dHT frame
   to downstream dHT frame
```

```
        */

        printf("#23 = %lf\n",1.0-BETASWD*BETASWD);
        gamma_ds = 1.0/sqrt(1.0-BETASWD*BETASWD);
        printf("#24 = %lf\n",1.0-(1.0-mu_us*mu_us)*BDOVERBU);
        printf("mu_us = %lf\n",mu_us);
        mu_ds    = sqrt( 1.0 - ( 1.0 - mu_us*mu_us ) * BDOVERBU );
        p_ds     = p_us;
        printf("#25 = %lf\n",p_ds*p_ds+m2);
        Eds      = sqrt( p_ds*p_ds+m2 );
        printf("#26 = %lf\n",p_ds*p_ds*(1.0-mu_ds*mu_ds));
        p_ds_per = sqrt( p_ds*p_ds*(1.0-mu_ds*mu_ds));
        p_ds_par = gamma_ds*( p_ds*mu_ds+BETASWD*Eds );

      /* transforming from downstream dHT frame to
         downstream local frame
       */
          printf("#27 = %lf\n",p_ds_per*p_ds_per+p_ds_par*p_ds_par);
          pd   = sqrt( p_ds_per*p_ds_per + p_ds_par*p_ds_par );
          mu_d = p_ds_par / pd;

        return mu_d;
      }
}

/*  determine mu_u from pd, mu_d and when
```

```
    (a) |mu_d| > 1+TINY report error
    (b) 1-TINY < mu_d < 1+TINY return +1
    (c) -1-TINY < mu_d < -1+TINY return -1.
 */

double  mu_u( m, pd, mu_d )
double  m, pd, mu_d;
{
      double m2, pu, mu_u;
      double Ed, p_ds_par, p_ds_per, p_ds, mu_ds, gamma_ds;
      double mu_us, p_us, p_us_par, p_us_per, Eus, gamma_us;

       m2 = m*m;

       if( fabs(mu_d) > 1.0+TINY )
         nrerror("error in routine mu_u():mu_d out of bounds");

       else if( 1.0-TINY < mu_d && mu_d < 1.0+TINY )
         return 1.0;

       else if( -1.0-TINY < mu_d && mu_d < -1.0+TINY )
         return -1.0;

     else {
```

```
/* energy in downstream local frame
 */
      printf("#28 = %lf\n",pd*pd+m2);
      Ed =  sqrt( pd*pd + m2 );

/*  transforming from downstream local frame to
    downstream dHT frame
 */

      printf("#29 = %lf\n",1.0-BETASWD*BETASWD);
      gamma_ds = 1.0 / sqrt( 1.0 - BETASWD*BETASWD );
      printf("#30 = %lf\n",pd*pd*(1.0-mu_d*mu_d));
      p_ds_per = sqrt( pd*pd*( 1.0-mu_d*mu_d ));
      p_ds_par = gamma_ds*( pd*mu_d-BETASWD*Ed );
      printf("#31 = %lf\n",p_ds_per*p_ds_per+p_ds_par*p_ds_par);
      p_ds     = sqrt( p_ds_per*p_ds_per + p_ds_par*p_ds_par );
      mu_ds    = p_ds_par / p_ds;


/* transforming from downstream dHT frame
   to upstream dHT frame
 */
printf("#32 = %lf\n",1.0-BETASWU*BETASWU);
gamma_us = 1.0 / sqrt(1.0-BETASWU*BETASWU);
printf("#33 = %lf\n",1.0-(1.0-mu_ds*mu_ds)/BDOVERBU);
mu_us    = -sqrt( 1.0 - ( 1.0 - mu_ds*mu_ds ) / BDOVERBU );
p_us     = p_ds;
```

```
        printf("#34 = %lf\n",p_us*p_us+m2);
        Eus      = sqrt( p_us*p_us+m2 );
        printf("#36 = %lf\n",p_us*p_us*(1.0-mu_us*mu_us));
        p_us_per = sqrt( p_us*p_us*(1.0-mu_us*mu_us));
        p_us_par = gamma_us*( p_us*mu_us+BETASWU*Eus );

       /* transforming from upstream dHT frame to
          upstream local frame
        */
           printf("#36 = %lf\n",p_us_per*p_us_per+p_us_par*p_us_par);
           pu    = sqrt( p_us_per*p_us_per + p_us_par*p_us_par );
           mu_u = p_us_par / pu;

        return mu_u;
    }
 }


double   mustar1(m,pu,mustard,mustarr)
double   m, pu, *mustard, *mustarr;
{
   double newsqrt();

   double mu_us,mu_u;
   double Eu, gswu,gswd,pus,pdpar,pupar;
   double a,b,c;
```

```
    /* This routine considers mu_ds = 0. */

    gswu = 1.0 / newsqrt( 1.0 - BETASWU*BETASWU );
    Eu = newsqrt( pu*pu + m*m );
    mu_us = -newsqrt(1.0-1.0/BDOVERBU);

    a= ( 1.0-mu_us*mu_us ) * gswu*gswu * pu*pu + mu_us*mu_us*pu*pu;
    b= -2.0 * pu * BETASWU * Eu * gswu*gswu * ( 1.0-mu_us*mu_us );
    c= BETASWU*BETASWU * Eu*Eu * gswu*gswu * ( 1.0-mu_us*mu_us )
       - mu_us*mu_us * pu*pu;
    mu_u = ( -b - newsqrt( b*b - 4.0*a*c )) / ( 2.0*a );
    *mustarr = ( -b + newsqrt( b*b - 4.0*a*c)) / ( 2.0*a );

    /* compute mustard
     */

    gswd = 1.0 / newsqrt( 1.0 - BETASWD*BETASWD );
    pus = newsqrt(gswu*( mu_u*pu-BETASWU*Eu )*gswu*( mu_u*pu-BETASWU*Eu )
              + pu*pu*(1.0-mu_u*mu_u));
    pdpar = gswd*BETASWD*newsqrt( m*m + pus*pus );
    *mustard = pdpar / newsqrt( pus*pus+pdpar*pdpar );
 return mu_u;
}

/*
```

```
    compute pr: momentum refecting at the shoch front back to upstream
                local frame when given
             pu: momentum incident at the shock front in upstream local
                frame
          mu_u: cos(phi) in upstream local frame
*/

double refl_pr( m, pu, mu_u, dpudpr )
double m, pu, mu_u, *dpudpr;
{
   double Eu, difEu, pus, difpus, puspar, difpuspar, pusper, difpusper;
   double gswu, prs, difprs, prspar, difprspar, prsper, difprsper;
   double mu_rs, difmu_rs, Ers, difErs, pr, difpr, prpar, difprpar;
   double prper, difprper, mu_us;

 /* in upstream local frame
  */
     Eu = sqrt( pu*pu+m*m );
     difEu = pu/Eu;

 /* Transforming from upstream local frame to
    upstream dHT frame.
  */

     gswu = 1.0/sqrt( 1.0 - BETASWU*BETASWU );
     puspar = gswu*( mu_u*pu - BETASWU*Eu );
     difpuspar = gswu*( mu_u - BETASWU*difEu );
```

```
   pusper = pu*sqrt( 1.0 - mu_u*mu_u );
   difpusper = sqrt( 1.0 - mu_u*mu_u );
   mu_us = puspar/sqrt( puspar*puspar + pusper*pusper );
    if( mu_us >= 0.0 )
      printf("\n in dHT frame mu_us = %lf",mu_us);

/* Reflecting at the shock front in dHT frame back to
   upstream
 */

   prspar = -puspar;
   difprspar = -difpuspar;
   prsper = pusper;
   difprsper = difpusper;
   prs = sqrt( prspar*prspar + prsper*prsper );
   difprs = ( prspar*difprspar + prsper*difprsper ) / prs;
   mu_rs = prspar/prs;
   difmu_rs = ( prs*difprspar - prspar*difprs ) / ( prs*prs );
   Ers = sqrt( prs*prs + m*m );
   difErs = ( prs/Ers )*difprs;

/* Transforming back from dHT frame to upstream local frame
 */

   prpar = gswu*( prspar + BETASWU*Ers );
   difprpar = gswu*( mu_rs*difprs + prs*difmu_rs + BETASWU*difErs );
   prper = prsper;
```

```
     difprper = difprsper;
     pr = sqrt( prpar*prpar + prper*prper );
     difpr = ( prpar*difprpar + prper*difprper )/pr;
    *dpudpr = 1.0/difpr;
     return pr;
}


/* compute mu after reflecting at the shock front from
   upstream local frame back to upstream local frame

    NOTE: momentum magnitude conservs in dHT frame
    pu -> momentum in upstream local frame
    mu -> mu in upstream local frame
 */

double  mu_refl( m,pu,mu )
double m,pu,mu;
{
   double pus,pus_par,pus_per,Eu,Eus,gswu,pr;
   double pus_refl_par,pus_refl_per,pu_refl_par,pu_refl_per;

   /* in upstream local frame we compute energy
      and gamma of particle
    */
```

```
Eu = sqrt( m*m + pu*pu );
gswu = 1.0 / sqrt( 1.0-BETASWU*BETASWU );

/* transforming from upstream local frame to
   dHT frame
 */
pus_par = gswu*( mu*pu - BETASWU*Eu );
pus_per = pu*sqrt( 1.0-mu*mu );


/*  reflection in dHT frame conservs energy, and
    momentum change its direction but its magnitude
    does not.
 */

pus_refl_par = -pus_par;
pus_refl_per = pus_per;
Eus = sqrt( pus_refl_par*pus_refl_par+pus_refl_per*pus_refl_per+m*m );

/* transforming back from dHT frame to upstream
   local frame
 */
pu_refl_par = gswu*( pus_refl_par + BETASWU*Eus );
pu_refl_per = pus_refl_per;
pr = sqrt( pu_refl_par*pu_refl_par + pu_refl_per*pu_refl_per );

return pu_refl_par/pr;
```

```
}



/* compute square root of x
   if |x| < TINY
   it return 0
 */

double newsqrt( double x )
{
   if( x < -TINY ) nrerror("newsqrt error");
   if( fabs(x) < TINY ) return 0.0;
   else return sqrt(x);
}
```

Curriculum Vitae

Mr. Songklod Riyavong was born on January 2, 1971 in Udonthani. He received his B.Sc. degree in physics from Khon Kaen University in 1992. He received support from the Institute for the Promotion of Teaching Science and Technology for both his undergraduate and graduate studies.