Solutions to Problem Set 3

Chris H. Rycroft

October 30, 2006

1 Inelastic diffusion

1.1 The PDF of $P_N(x)$

The PDF of the random variable $a^n \Delta X_n$ is $p(xa^{-n})a^{-n}$. If ΔX_n has characteristic function $\hat{p}(k)$ then $a^n \Delta X_n$ has characteristic function

$$\begin{aligned} \int_{-\infty}^{\infty} e^{-ikx} \frac{p(xa^{-n})}{a^n} dx &= \int_{-\infty}^{\infty} e^{-ikya^n} p(y)\, dy \\ &= \hat{p}(ka^n). \end{aligned}$$

By the convolution theorem, we know that the characteristic function of X_N can be written as

$$\hat{P}_N(k) = \prod_{n=1}^{N} \hat{p}(ka^n)$$

and hence the PDF is given by

$$\begin{aligned} P_N(k) &= \int_{-\infty}^{\infty} \frac{e^{ikx}\, dx}{2\pi} \prod_{n=1}^{N} \hat{p}(ka^n) \\ &= \int_{-\infty}^{\infty} \frac{e^{ikx}\, dx}{2\pi} \prod_{n=1}^{N} \int_{-\infty}^{\infty} e^{-ikya^n} p(y)\, dy. \end{aligned}$$

1.2 The cumulants of X_N

The cumulant generating function of the nth step is given by

$$\psi_n(k) = \log \hat{p}(ka^n)$$

and therefore the lth cumulant is given by

$$i^{-l} \left. \frac{d^l \psi_n}{dk^l} \right|_{k=0} = a^{ln} i^{-l} \left. \frac{d^l}{dk^l} \psi_n \right|_{k=0} = a^{ln} c_l.$$

If random variables are added, then their cumulants add. We therefore know that the cumulants of X_N are given by

$$C_{N,l} = \sum_{n=1}^{N} a^{ln} c_l = \frac{a^l(1-a^{lN})}{1-a^l} c_l.$$

1.3 Limits as $N \to \infty$

By taking $N \to \infty$ in the above expression, we see that the cumulants of X_∞ are given by

$$C_l = \frac{a^l}{1-a^l} c_l.$$

If $a = 1 - \epsilon$ where $\epsilon > 0$ is small, then we see that

$$\begin{aligned}
\frac{C_{2m}}{C_m^2} &= \frac{a^{2m}}{1-a^{2m}} \left(\frac{1-a^2}{a^2} \right)^m \frac{c_{2m}}{c_m^2} \\
&= \frac{(2\epsilon - \epsilon^2)^m}{1-(1-\epsilon)^{2m}} \frac{c_{2m}}{c_m^2} \\
&= \frac{2^m \epsilon^m \left(1 - \frac{\epsilon}{2}\right)^m}{2m\epsilon - \frac{2m(2m-1)}{2}\epsilon^2 + \dots} \frac{c_{2m}}{c_m^2} \\
&= 2^m \epsilon^{m-1} \frac{1 - \epsilon m + \dots}{2m - \frac{2m(2m-1)}{2}\epsilon + \dots} \frac{c_{2m}}{c_m^2} \\
&= O(\epsilon^{m-1}).
\end{aligned}$$

1.4 A "Central Limit Theorem"

The PDF of X_∞ can be written as

$$\begin{aligned}
P_\infty(x) &= \int_{-\infty}^{\infty} \frac{dk}{2\pi} e^{ikx + \psi_\infty(k)} \\
&= \int_{-\infty}^{\infty} \frac{dk}{2\pi} \exp\left(ikx + 0 - \frac{C_2 k^2}{2} - \frac{iC_3 k^3}{6} + \frac{C_4 k^4}{24} \right).
\end{aligned}$$

If $x = \zeta C_2^{1/2}$ we see that

$$\phi(\zeta, \epsilon) = C_2^{1/2} P_\infty(\zeta C_2^{1/2}) = \int_{-\infty}^{\infty} \frac{dl}{2\pi} \exp\left(il\zeta - \frac{l^2}{2} - i\frac{C_3 l^3}{6C_2^{1/2}} + \frac{C_4 l^4}{24 C_2^2} \right).$$

From the previous result, we know that as $\epsilon \to 0$, the terms involving the higher cumulants tend to zero. Therefore we have

$$\begin{aligned}
\phi(\zeta, \epsilon) \to \phi(\zeta) &= \int_{-\infty}^{\infty} \frac{dl}{2\pi} e^{il\zeta - l^2/2} \\
&= \frac{e^{-\zeta^2/2}}{\sqrt{2\pi}}.
\end{aligned}$$

2 Breakdown of the CLT for decaying walks

2.1 The PDF of X_∞ for $a = 0.99$

Appendix A shows a simple C++ code to simulate 10^5 walkers from the decaying PDF. Figure 1 shows the PDF of x component of this distribution, and figure 2 shows a comparison to the

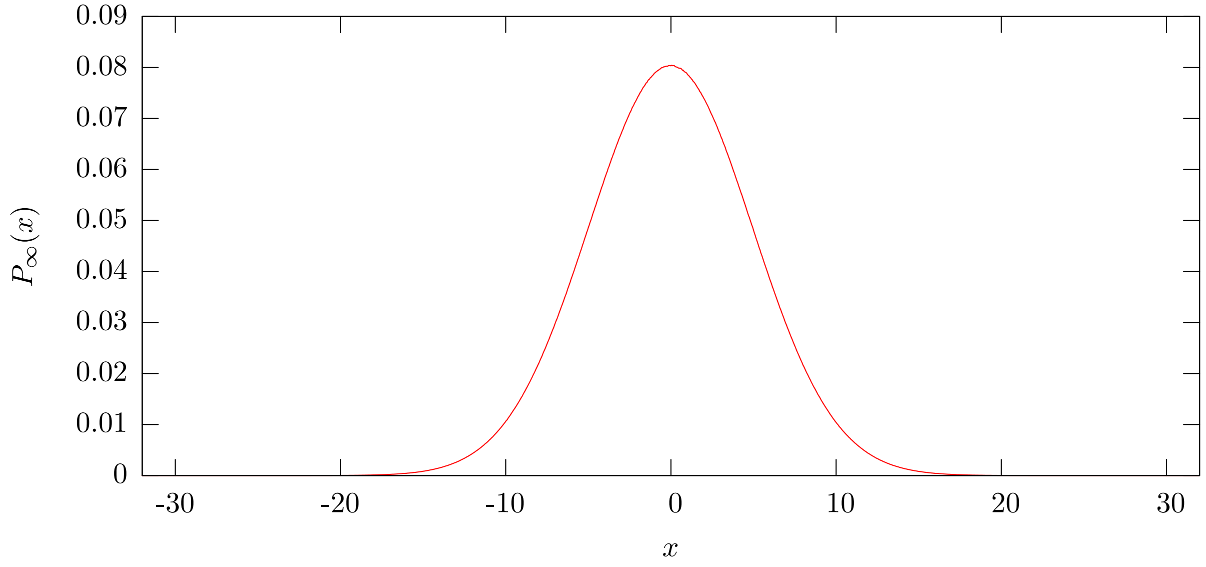

Figure 1: Simulated PDF of X_∞ for $a = 0.99$ based on 10^9 trials.

formulae calculated in question 1. We see that the two curves match to a very high degree of accuracy, particularly in the central region. We know however that this match will not continue forever: since $\vec{X}_\infty$ is bounded, its PDF will be uniformly zero outside of some region. However a Gaussian features non-zero probabilities everywhere.

2.2 The exact PDF of X_∞ for $a = 1/2$

To find the PDF of X_∞ for $a = 1/2$ we first consider the change of variables $\vec{X} = (x, y) \to \vec{R} = (r, s)$ given by

$$\begin{aligned} r &= \frac{1+x+y}{2} \\ s &= \frac{1+x-y}{2}. \end{aligned}$$

In the transform variables, the walk starts from $\vec{R}_0 = (1/2, 1/2)$ and the PDF of the nth step is given by

$$p_n(r,s) = \frac{1}{4}(\delta_{r,-1/2^{n+1}} + \delta_{r,1/2^{n+1}})(\delta_{s,-1/2^{n+1}} + \delta_{s,1/2^{n+1}}).$$

From this from, it is clear that the variables r and s are independent. We see that the Nth step of the r component can be written as

$$R_N = \frac{1}{2} + \sum_{n=1}^{N} \frac{2I_n - 1}{2^{n+1}}$$

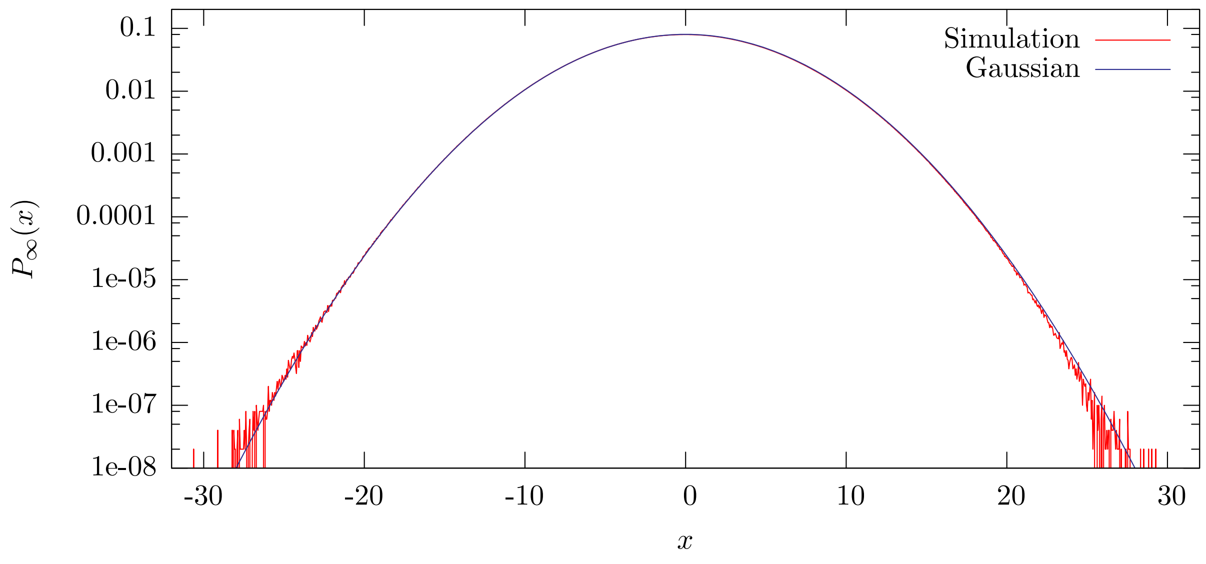

Figure 2: Simulated PDF of X_∞ for $a = 0.99$ based on 10^9 trials on a log scale, compared with a Gaussian curve with a variance calculated using the formula derived in question 1.

where the I_n's are independent random variables which take values 0 and 1 with equal probability. This can be rewritten as

$$\begin{aligned} R_N &= \frac{1}{2} - \sum_{n=1}^{N} \frac{1}{2^{n+1}} + \sum_{n=1}^{N} \frac{I_n}{2^n} \\ &= \frac{1}{2} - \frac{1}{2^2}\left(\frac{1-\frac{1}{2^N}}{1-\frac{1}{2}}\right) + \sum_{n=1}^{N} \frac{I_n}{2^n} \\ &= \frac{1}{2^{N+1}} + \sum_{n=1}^{N} \frac{I_n}{2^n}. \end{aligned}$$

In the limit, as $N \to \infty$, we obtain

$$R_\infty = \sum_{n=1}^{N} \frac{I_n}{2^n}.$$

Thus, the binary expansion of R_∞ is

$$R_\infty = 0.I_1 I_2 I_3 I_4 I_5 I_6 \ldots$$

From this form, it is clear that every possible binary expansion between 0 and 1 can be achieved, each with equal probability. Thus R_∞ is uniformly distributed on the interval $[0, 1]$. Since the same is true for S_∞, we know that the joint PDF is given by

$$P_\infty(r, s) = \begin{cases} 1 & \text{for } 0 < r < 1,\ 0 < s < 1 \\ 0 & \text{otherwise.} \end{cases}$$

Thus the PDF of X_∞ is given by

$$P_\infty(r, s) = \begin{cases} \frac{1}{2} & \text{for } |x - y| < 1,\ |x + y| < 1 \\ 0 & \text{otherwise.} \end{cases}$$

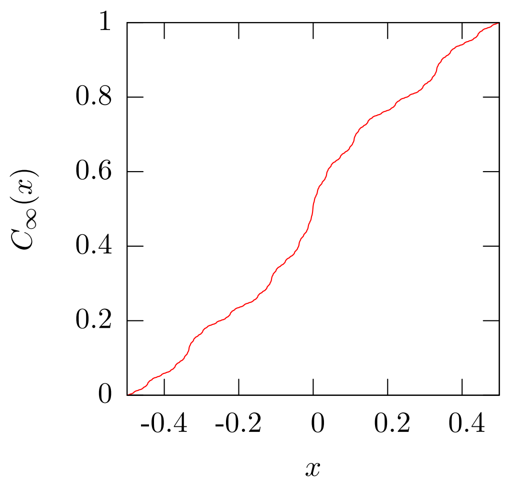

Figure 3: The cumulative distribution function of the x component of $\vec{X}_\infty$ for $a = 1/3$.

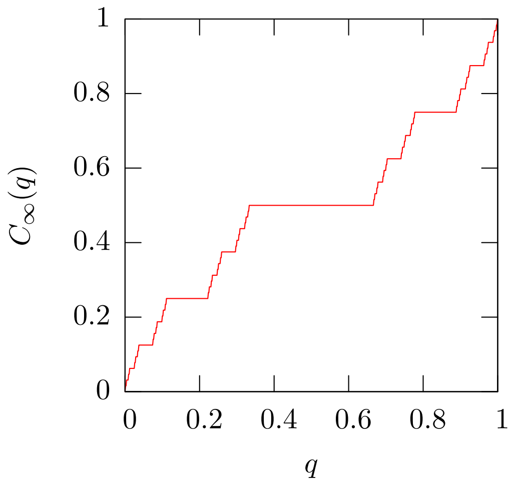

Figure 4: The cumulative distribution function of the rotated variable $q = 1/2 + x - y$ for $a = 1/3$.

2.3 The CDF of $\vec{X}_\infty$ for $a = 1/3$

To compute the CDF of x component of $\vec{X}_\infty$ for $a = 1/3$, we first consider the possible values that it can take. The maximum that x can increase by at the nth step is $1/3^n$ and thus the maximum value that x_∞ can take is

$$\sum_{n=1}^{\infty} \frac{1}{3^n} = \frac{1}{3(1-\frac{1}{3})} = \frac{1}{2}.$$

By symmetry, we therefore know that X_∞ can take values in the range $[-1/2, 1/2]$. We also note that the sum of steps starting at $n = 2$ follows the same distribution as X_∞, but scaled by $1/3$. If the first step is ΔX_1, then X_∞ lies in the interval $[\Delta X_1 - 1/6, \Delta X_1 + 1/6]$. For the three possible choices $\Delta X_1 = -1/3, 0, 1/3$ these intervals are mutually exclusive, and this allows us to write the CDF $C_\infty(x)$ recursively as

$$C_\infty(x) = \begin{cases} C_\infty(3x+1)/4 & \text{for } x < -\frac{1}{6} \\ 1/4 + C_\infty(3x)/2 & \text{for } -\frac{1}{6} < x < \frac{1}{6} \\ 3/4 + C_\infty(3x-1)/4 & \text{for } \frac{1}{6} < x \end{cases}$$

This can be easily calculated using a recursive function, and a graph is shown in figure 3. It is also interesting to consider the CDF of $\vec{X}_\infty$ in a coordinate rotated by 45°. Suppose we consider the coordinate transformation

$$\begin{aligned} p &= \frac{1}{2} + x + y \\ q &= \frac{1}{2} + x - y \end{aligned}$$

Then we find that the walk from starts from $(p, q) = (1/2, 1/2)$ and the PDF of the nth step is given by

$$p_n(r, s) = \frac{1}{4}(\delta_{r,-1/3^n} + \delta_{r,1/3^n})(\delta_{s,-1/3^n} + \delta_{s,1/3^n}).$$

From this from, it is clear that the variables p and q are independent. We see that the Nth step of the q component can be written as

$$\begin{aligned} Q_N &= \frac{1}{2} + \sum_{n=1}^{N} \frac{2I_n - 1}{3^n} \\ &= \frac{1}{2} - \sum_{n=1}^{N} \frac{1}{3^n} + \sum_{n=1}^{N} \frac{2I_n}{3^n} \\ &= \frac{1}{2} - \frac{1}{3}\left(\frac{1-\frac{1}{3^N}}{1-\frac{1}{3}}\right) + \sum_{n=1}^{N} \frac{2I_n}{3^n} \\ &= \frac{1}{3^{N+1}} + \sum_{n=1}^{N} \frac{2I_n}{3^n} \end{aligned}$$

where the I_N are independent random variables that take values of 0 and 1 with equal probability, as in the previous section. In the limit as $N \to \infty$, we obtain

$$Q_\infty = \sum_{n=1}^{N} \frac{I_n}{2^n}.$$

Thus, the expansion in base-3 of R_∞ is

$$Q_\infty = 0.(2I_1)(2I_2)(2I_3)(2I_4)(2I_5)(2I_6)\ldots$$

and hence every possible number whose base-3 expansion contains only 0's and 2's with no 1's occurs with equal probability. The numbers which satisfy this property form the Cantor set, a well known fractal distribution that can be obtained by taking the unit interval, removing the middle third, and then recursively removing the third of each new interval. The cumulative distribution can be written as

$$C_\infty(q) = \begin{cases} C_\infty(3q)/2 & \text{for } q < \frac{1}{3} \\ 1/2 & \text{for } \frac{1}{3} < q < \frac{2}{3} \\ C_\infty(3q-2)/2 + 1/2 & \text{for } \frac{2}{3} < q. \end{cases}$$

This is referred to as the Cantor function and is shown in figure 4. It is a type of function referred to as "Devil's staircase". A function $f(x)$ is a Devil's staircase on the interval $[a,b]$ if it satisfies the following properties:

- $f(x)$ is continuous on $[a,b]$.
- There exists a set N of measure 0 such that for all x outside of N, $f'(x)$ exists and is zero.
- $f(x)$ in nondecreasing on $[a,b]$.
- $f(a) < f(b)$.

2.4 Other values of a

Figure 5 shows the PDF of the decaying walk for twelve different values of a. As a increases, we see a progression from a sparse PDF (similar to a Cantor set), through a uniform PDF for $a = 0.5$, to a PDF approaching a Gaussian as a approaches 1. For the case when $a = 2^{-1/2}$, we can write the random variable as

$$\vec{X}_\infty = \sum_{n=1}^{\infty} \frac{\vec{I}_n}{2^{n/2}}$$

Where the $\vec{I}_n$ take values $(1, 0)$, $(-1, 0)$, $(0, 1)$, and $(0, -1)$ with a quarter probability each. This can be rewritten as

$$\begin{aligned} \vec{X}_\infty &= \sqrt{2}\sum_{m=1}^{\infty} \frac{\vec{I}_{2m-1}}{2^m} + \sum_{m=1}^{\infty} \frac{\vec{I}_{2m}}{2^m} \\ &= \sqrt{2}\vec{Y}_1 + \vec{Y}_0 \end{aligned}$$

where $\vec{Y}_1$ and $\vec{Y}_0$ are decaying walks with parameter $a = 1/2$. From part (b), we know that these are uniformly distributed on the region $|x| + |y| < 1$, which allows us to easily construct an analytic solution to the resultant PDF as a convolution two simple PDFs. Similarly, we have

$$\begin{aligned} \vec{X}_\infty &= \sum_{i=0}^{3} 2^{i/4}\vec{Y}_i \qquad \text{for } a = 2^{-1/4} \\ \vec{X}_\infty &= \sum_{i=0}^{7} 2^{i/8}\vec{Y}_i \qquad \text{for } a = 2^{-1/8} \end{aligned}$$

where the $\vec{Y}_i$ are all decaying walks with parameter $a = 1/2$; again, this easily allows us to construct the resultant PDF as a convolution of a finite number of simple PDFs.

Figure 5 also shows the PDF for the case for when a is equal to the golden mean, $(\sqrt{5} - 1)/2$. For this value, $1 - a = a^2$, and this special relationship results in sections of the PDF appearing self-similar.

3 Shock structure

We are interested in finding a traveling wave solution $c(x, t) = f(x - vt)$ of Burgers' equation

$$c_t + cc_x = Dc_{xx}$$

which satisfies the boundary conditions $c(-\infty, t) = c_-$ and $c(\infty, t) = c_+$. Writing $z = x - vt$, we find that $f(z)$ satisfies

$$(-v + f)f' = Df''$$

which can be integrated to obtain

$$-vf + \frac{f^2}{2} = Df' + A$$

for some constant A. Applying the boundary conditions, and assuming $f'(z) \to 0$ as $z \to \pm\infty$, we find that

$$\begin{aligned} -vc_- + \frac{c_-^2}{2} &= A \\ -vc_+ + \frac{c_+^2}{2} &= A \end{aligned}$$

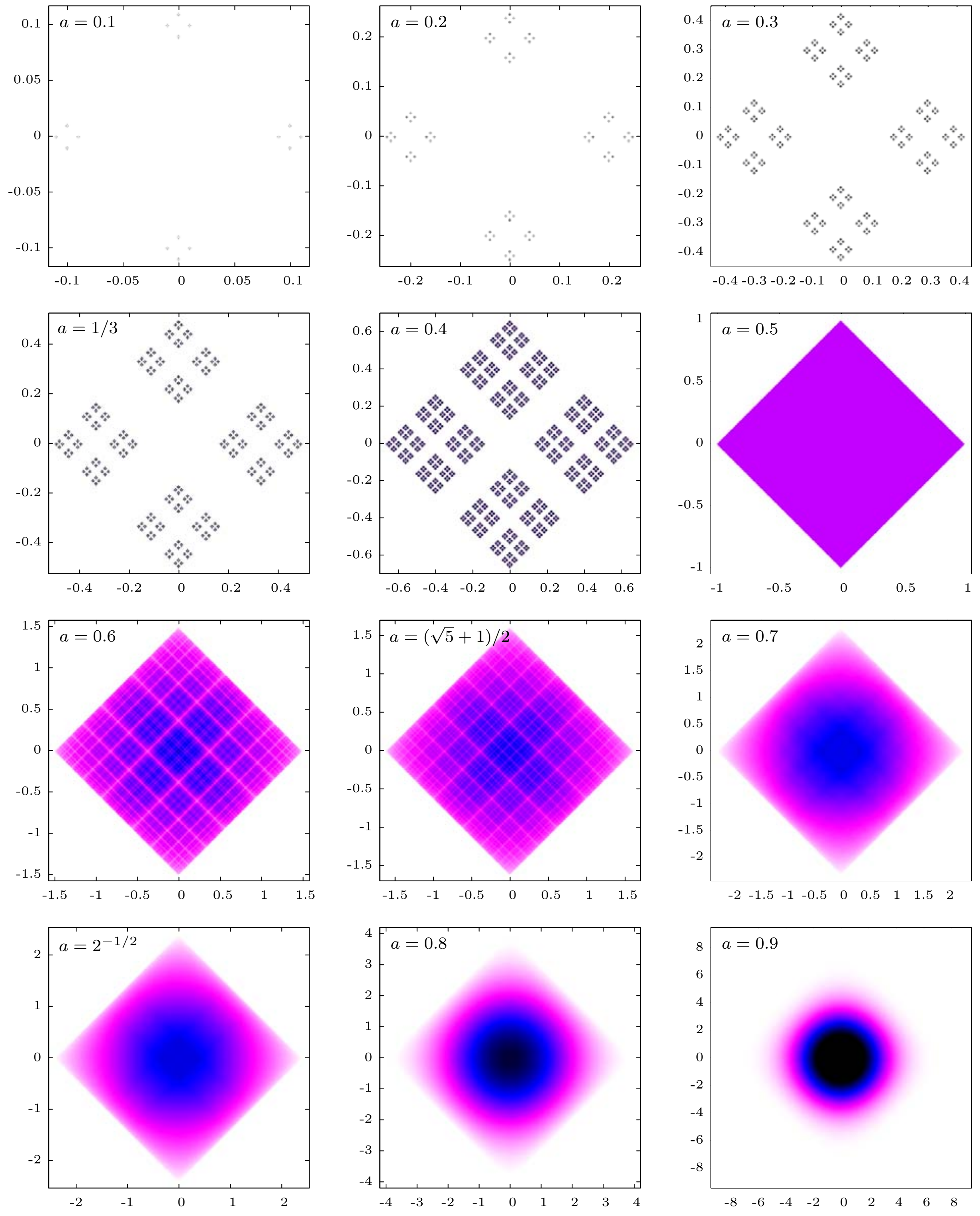

Figure 5: Plots of the PDF of the decaying random walk for twelve different values of a. For each graph, 5×10^8 trials were performed. The color scheme goes from white (zero probability density), through purple and blue, to black (high probability density).

from which we find that the velocity must satisfy

$$\begin{aligned} 2v(c_+ - c_-) &= (c_+^2 - c_-^2) \\ v &= \frac{c_+ + c_-}{2}. \end{aligned}$$

From this, we find that

$$A = -\frac{c_+ c_-}{2}$$

and hence f must satisfy

$$(c_+ c_-)f - f^2 + 2Df' = c_+ c_-.$$

We can now apply separation of variables to obtain

$$\begin{aligned} \int \frac{2D\,df}{f^2 - (c_+ + c_-) + c_+ c_-} &= \int dz \\ \int \frac{2D\,df}{(f - c_+)(f - c_-)} &= \int dz \\ \int \frac{2D\,df}{c_- - c_+}\left(-\frac{df}{f - c_+} - \frac{df}{c_- - f}\right) &= z + k \\ \frac{2D}{c_- - c_+}\left(-\log(f - c_+) + \log(c_- - f)\right) &= z + k. \end{aligned}$$

The variable k corresponds to a translation and does not affect the form of the function; it is convenient to set $k = 0$. Then we find

$$\begin{aligned} \frac{c_- - f}{f - c_+} &= e^{\frac{c_- - c_+}{2D}z} \\ c_- - f &= f e^{\frac{c_- - c_+}{2D}z} - c_+ e^{\frac{c_- - c_+}{2D}z} \end{aligned}$$

and hence

$$\begin{aligned} f(z) &= \frac{c_- + c_+ e^{\frac{c_- - c_+}{2D}z}}{1 + e^{\frac{c_- - c_+}{2D}z}} \\ &= \frac{c_- + c_+}{2} + \frac{c_- - c_+}{2}\left(\frac{1 - e^{\frac{c_- - c_+}{2D}z}}{1 + e^{\frac{c_- - c_+}{2D}z}}\right) \\ &= \frac{c_- + c_+}{2} + \frac{c_- - c_+}{2}\tanh\left(\frac{c_+ - c_-}{4D}z\right). \end{aligned}$$

4 Discrete versus continuous models with nonlinear drift

Appendices B and C provide C++ codes to simulate the nonlinear drift problem for models A and B respectively. These codes follow an approach very similar to that stated in the problem. The largest difference is in the application of the boundary conditions. We assume that the density of particles to the left of the simulation has a constant value of $\rho_{\max}/4$. If we take an interval of finite length λ then the number of particles in this region should follow a Poisson distribution with mean $\rho_{\max}\lambda/4$. Similarly, to the right of the simulation, the density is assumed to have constant value of

$3\rho_{\max}/4$, and the number of particles in an interval of length λ follows a Poisson distribution with mean $3\rho_{\max}\lambda/4$.

For model A, our steps take the form $u(\rho)\tau \pm \sqrt{2D\tau}$, and for the parameters in this question, the diffusive term is significantly larger than the drift. A hypothetical particle to the left of the simulation region could enter the simulation region if it was in the interval $-L - \sqrt{2D\tau} - u(\rho)\tau < x < -L$, and if it took a step to the right. Since the chance of stepping right is exactly 1/2, it was chosen to introduce particles according to a Poisson process with mean

$$\frac{\rho_{\max}}{4} \times \frac{1}{2} \times \left(u(\rho_{\max}/4)\tau + \sqrt{2D\tau} \right)$$

into the interval $-L < x < -L + \sqrt{2D\tau} + u(\rho_{\max}/4)$. A similar analysis shows that at the right edge, particles need to be introduced at a rate of

$$\frac{3\rho_{\max}}{4} \times \frac{1}{2} \times \left(\sqrt{2D\tau} - u(3\rho_{\max}/4)\tau \right)$$

into the interval $L - \sqrt{2D\tau} + u(3\rho_{\max}/4) < x < L$. For model B, assuming that the ρ is roughly constant near the boundary, and thus ρ_x is roughly zero, particles take steps of size $u(\rho)\tau$. Thus we need to introduce no particles at the right boundary, and particles at a rate of $u(\rho_{\max}/4)\tau\rho_{\max}/4$ at the left boundary.

When calculating the estimates of ρ and ρ_x at a point p in the interval, we make use of the prescription given in the question. ρ is calculated by looking at dividing the number of particles in the range $p - l < x < p + l$ by $2l$. ρ_x is calculated by finding the difference between the number of particles in the regions $p - l < x < p$ and $p < x < p + l$ and dividing by l^2. If any of these counting regions overlap with the boundary, say by an amount δ, then we make an additional contribution to the density of $\rho\delta$.

Figures 6 and 7 show simulation output for the two models. In both cases, we see a good agreement with the predicted result, although there are some discrepancies. After many timesteps, particularly for model A, there is a tendency for the standing wave to drift to the right. A more careful treatment of the boundary conditions may solve this.

Model B works surprisingly well. The presence of the ρ_x/ρ term causes the particles to distribute themselves very evenly over the simulation region, meaning that smooth density plots are possible even over a short number of iterations. Unfortunately, the simple boundary condition for introduction of particles at the left side appears not to be sufficient. If the density close to the left simulation edge grows above $\rho_{\max}/4$, then the particles will not move away from the edge fast enough to balance the rate of particle introduction, causing some degree of positive feedback. This could be fixed by making the particle introduction rate dependent on the density of particles close to the left edge.

To investigate these models further, a more advanced C++ code was written which can efficiently handle much larger numbers of particles, which is given in appendix D. Rather than having a single array for the walker positions, walkers are binned depending on their x position. In order to calculate the estimates for ρ over an interval, it is only necessary to scan the particles in the bins which overlap the interval. Furthermore, if a bin lies wholly within the interval, one just needs to add the total number of particles in that bin to the calculation of ρ. These speedups effectively reduce the computations of ρ at each step from an $O(N^2)$ calculation to an $O(N)$ calculation, allowing for many thousands of particles to be simulated.

The simulation results show some interesting behaviour. While the predicted curves frequently follow the hyperbolic tangent form in accordance with the answer, we also see transients in the

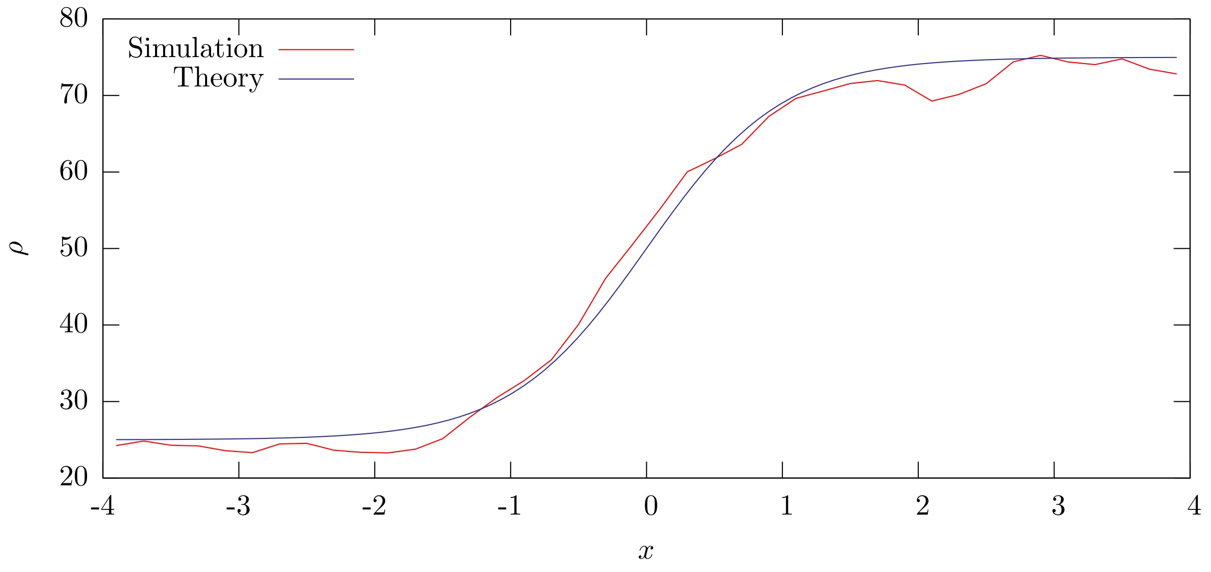

Figure 6: Simulation output for model A of the nonlinear drift problem, compared to the theoretical curve. The simulation results are time-averaged from the 100th iteration to the 5100th iteration.

density profile forming at the top of the standing wave. These tend to grow and then dissipate; an example is shown in figure 8.

A C++ code for calculating the decaying random walk

This code simulates 10^5 walkers from the decaying PDF. The code receives the value of a as the first input from the command line, and calculates the number of steps that must be computed in order for the final step to be of size 10^{-5}. The code prints a list of $\vec{X}_\infty$ points to the standard output.

```
#include <cstdio>
#include <iostream>
#include <cmath>
using namespace std;

const int trials=100000;
const float tol=0.00001;

int main (int argc, char* argv[]) {
        float a=atof(argv[1]),x,y,s;int i,j,k,r;
        k=int (log(tol)/log(a)+1);
        for(i=0;i<trials;i++) {
                x=y=0;s=a;
                for (j=0;j<k;j++) {
                        r=rand()%4;
                        if (r>1) x+=r==2?-s:s;
                        else y+=r==0?-s:s;
```

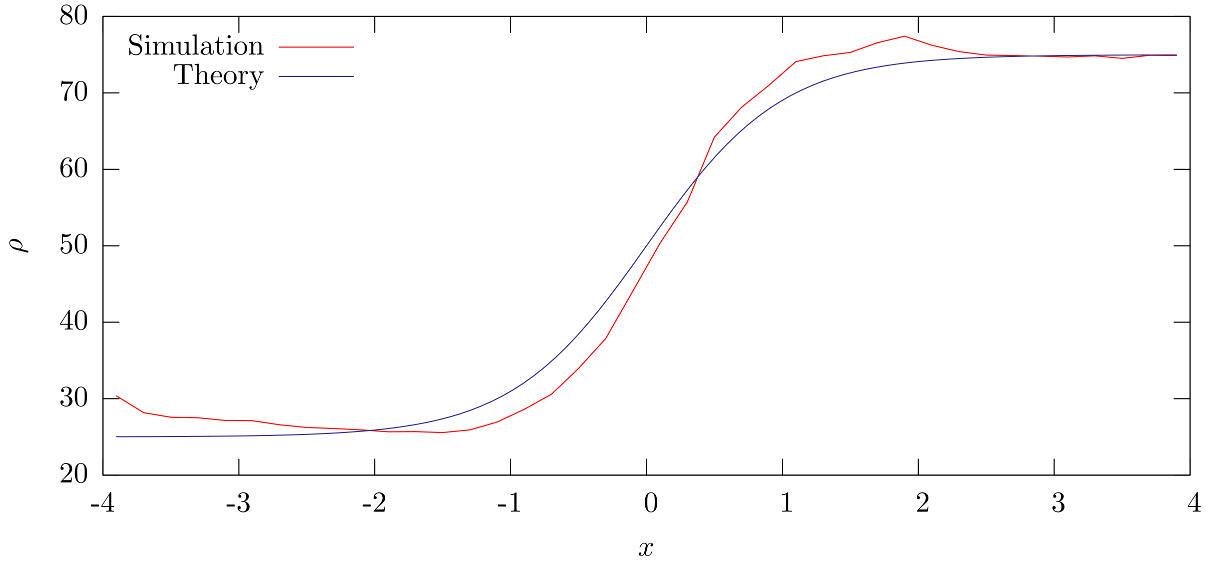

Figure 7: Simulation output for model B of the nonlinear drift problem, compared to the theoretical curve. The simulation results are time-averaged from the 100th iteration to the 400th iteration.

```
                s*=a;
            }
            cout << i << " " << x << " " << y << endl;
        }
}
```

B Simple C++ code for simulating model A

```
#include <string>
#include <iostream>
#include <cstdio>
#include <cmath>
using namespace std;

const int iter=5100;    //Total number of iterations
const float dx=0.2;     //Width for calculating local density
const float rhomax=100; //Value of rho_max
const float lrho=25;    //Value of rho for x<-4
const float rrho=75;    //Value of rho for x>4
const float mx=4;       //Half-width of simulation region
const float udt=0.01;   //u_max*dt;
const float ddt=sqrt(0.01*2/4); //sqrt(2*D*dt);
const int blocks=40;    //Number of blocks to save out

const float pi=3.1415926535897932384626433832795;
```

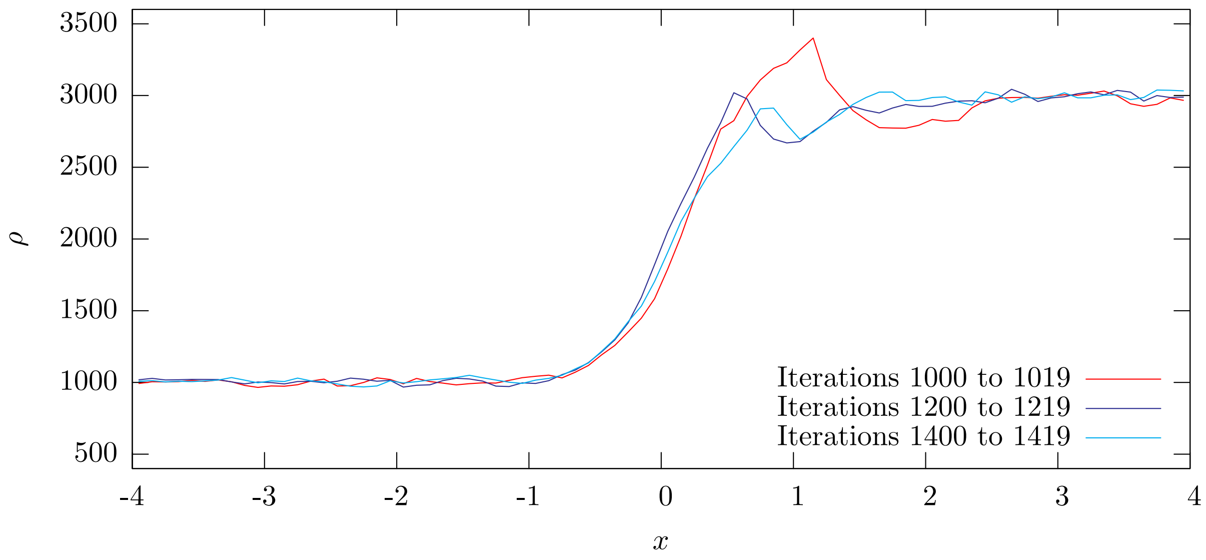

Figure 8: A transient in the density profile, occurring in a much larger version of the simulation. At approximately 1000 iterations a large peak is seen at the top of the standing wave but this dissipates as the simulation progresses.

```
const int lwalk=int(mx*lrho);           //Total initial walkers on LHS
const int twalk=int(mx*(lrho+rrho));    //Total initial walkers
const float lspeed=udt*(1-lrho/rhomax)+ddt;
const float rspeed=ddt-udt*(1-rrho/rhomax);
const float lrate=0.5*lspeed*lrho;      //L. walker intro rate
const float rrate=0.5*rspeed*rrho;      //R. walker intro rate
const int mem=twalk*2;                  //Total memory
const float isp=float(blocks)/(2*mx);   //Inverse block spacing

inline float rnd() {return (float(rand())+0.5)/RAND_MAX;}
inline int poisson(float l) {
        if (l>20) {
                int r=int(sqrt(-2*log(rnd())*l)*cos(2*pi*rnd())+0.5+l);
                return r>0?r:0;
        } else {
                double a=rnd()*exp(l)-1,b=1;int r=0;
                while(a>0) {a-=b*=l/++r;}
                return r;
        }
}

int main() {
        int a=0,i,j=0,t;float w[mem],v[mem],as[blocks],r,x;
        while(j++<lwalk) w[a++]=-mx*rnd();
        while(j++<twalk) w[a++]=mx*rnd();
        for(i=0;i<blocks;i++) as[i]=0;
        for(t=0;t<iter;t++) {
                for(i=0;i<a;i++) {
                        j=int((w[i]+mx)*isp);
                        if (j>=0&&j<blocks&&t>100) as[j]++;
                        r=w[i]<dx-mx?lrho*(dx-mx-w[i]):0;
                        r+=w[i]>mx-dx?rrho*(w[i]-mx+dx):0;
                        for(j=0;j<a;j++) {
                                if (abs(w[i]-w[j])<dx) r++;
                        }
                        r/=2*dx;
                        v[i]=udt*(1-r/rhomax)+(rand()%2==1?ddt:-ddt);
                }
                i=0;
                while(i<a) {
                        if (abs(w[i]+=v[i])>mx) {
                                v[i]=v[a];
                                w[i]=w[--a];
                        }
                        else i++;
                }
                j=poisson(lrate);
                for(i=0;i<j;i++) w[a++]=lspeed*rnd()-mx;
                j=poisson(rrate);
                for(i=0;i<j;i++) w[a++]=mx-rspeed*rnd();
        }
        for(i=0;i<blocks;i++) {
                r=(float(i)+0.5)/isp-mx;
                cout << r << " "
                     << float(as[i])*blocks/2/mx/(iter-100) << endl;
        }
}
```

C Simple C++ code for simulating model B

```
#include <string>
#include <iostream>
#include <cstdio>
#include <cmath>
using namespace std;

const int iter=400;      //Total number of iterations
const float dx=0.2;      //Width for calculating local density
const float rhomax=100; //Value of rho_max
const float lrho=25;     //Value of rho for x<-4
const float rrho=75;     //Value of rho for x>4
const float mx=4;        //Half-width of simulation region
const float udt=0.01;   //u_max*dt;
const float ddt=0.0025; //D*dt;
const int blocks=40;    //Number of blocks to save out

const float pi=3.1415926535897932384626433832795;
const int lwalk=int(mx*lrho);          //Total initial walkers on LHS
const int twalk=int(mx*(lrho+rrho));   //Total initial walkers
const float lspeed=udt*(1-lrho/rhomax);
const float lrate=lspeed*lrho;         //L. walker intro rate
const int mem=twalk*2;                 //Total memory
const float isp=float(blocks)/mx*0.5;  //Inverse block spacing

inline float rnd() {return (float(rand())+0.5)/RAND_MAX;}
inline int poisson(float l) {
        if (l>20) {
                int r=int(sqrt(-2*log(rnd())*l)*cos(2*pi*rnd())+0.5+l);
                return r>0?r:0;
        } else {
                double a=rnd()*exp(l)-1,b=1;int r=0;
                while(a>0) {a-=b*=l/++r;}
                return r;
        }
}

int main() {
        int a=0,i,j=0,t;float w[mem],v[mem],as[blocks],r,s,x;
        while(j++<lwalk) w[a++]=-mx*rnd();
        while(j++<twalk) w[a++]=mx*rnd();
        for(i=0;i<blocks;i++) as[i]=0;
        for(t=0;t<iter;t++) {
                for(i=0;i<a;i++) {
                        j=int((w[i]+mx)*isp);
                        if (j>=0&&j<blocks&&t>100) as[j]++;
                        s=r=w[i]<dx-mx?lrho*(dx-mx-w[i]):0;
                        r+=w[i]>mx-dx?rrho*(w[i]-mx+dx):0;
                        for(j=0;j<a;j++) {
                                x=w[j]-w[i];
                                if (x>-dx) {
                                        if (x<dx) {
                                                r++;if (x<0) s++;
                                        }
                                }
                        }
                        s=(r-2*s)/(dx*dx);
                        r/=2*dx;
                        v[i]=udt*(1-r/rhomax)-ddt*s/r;
                }
                i=0;
                while(i<a) {
                        if (abs(w[i]+=v[i])>mx) {
                                v[i]=v[a];
                                w[i]=w[--a];
                        }
                        else i++;
                }
                j=poisson(lrate);
                for(i=0;i<j;i++) w[a++]=lspeed*rnd()-mx;
        }
        for(i=0;i<blocks;i++) {
                r=(float(i)+0.5)/isp-mx;
                cout << r << "␣"
                     << float(as[i])*blocks/2/mx/(iter-100) << endl;
        }
}
```

D Advanced C++ code for simulating the nonlinear drift model

```
#include <string>
#include <iostream>
#include <cstdio>
#include <cmath>
using namespace std;

const int iter=20000;           //Total number of iterations
const int bl=3200;              //Number of memory blocks
const float dx=0.1;             //Width for calculating local density
const float rhomax=40000;       //Value of rho_max
const float lrho=10000;         //Value of rho for x<-4
const float rrho=30000;         //Value of rho for x>4
const float mx=4;               //Half-width of simulation region
const float udt=0.01;           //u_max*dt;
const float ddt=0.05;           //D*dt;
const int smooth=40;            //Smoothing factor

const float pi=3.1415926535897932384626433832795;
const int lwalk=int(mx*lrho);           //Total initial walkers on LHS
const int twalk=int(mx*(lrho+rrho));    //Total initial walkers
const float ibs=bl/mx/2;                //Inverse block spacing
const float lspeed=udt*(1-lrho/rhomax)+ddt;
const float rspeed=ddt-udt*(1-rrho/rhomax);
const float lrate=0.5*lspeed*lrho;      //L. walker intro rate
const float rrate=0.5*rspeed*rrho;      //R. walker intro rate
const int res=bl/smooth;                //Number of data points to save
const int mem=int(rhomax*2*2*mx/float(bl));     //Memory estimate
float w[bl][mem];                               //Walker positions
float v[bl][mem];                               //Walker velocities
int a[bl];                                      //Tally for each memory block

inline int b(float x) {return int((mx+x)*ibs+10)-10;}
inline void put(float x) {int i=b(x);
        if(i<0||i>=bl) {cout << "***" << i << "␣" << x << endl;return;}
        w[i][a[i]++]=x;
}

inline float rnd() {return (float(rand())+0.5)/RAND_MAX;}
void blocks() {
                for(int i=0;i<bl;i++) cout << a[i] << "␣";
                cout << endl;
}
inline int poisson(float l) {
        if (l>25) {
                int r=int(sqrt(-2*log(rnd())*l)*cos(2*pi*rnd())+0.5+l);
                return r>0?r:0;
        } else {
                double a=rnd()*exp(l)-1,b=1;int r=0;
                while(a>0) {a-=b*=l/++r;}
                return r;
        }
}

inline float rho(float x) {
        float lx=x-dx,rx=x+dx,s;
        int l=b(lx),r=b(rx),m;
        if (l==r) {
                s=0;for(m=0;m<a[l];m++) {
                        if (w[l][m]>lx&&w[l][m]<rx) s+=1;
                        return s/(2*dx);
                }
        }
        if (l<0) {l=0;s=lrho*(-mx-lx);}
        else {
                s=0;for(m=0;m<a[l];m++) if(w[l][m]>lx) s+=1;l++;
        }
        if (r>=bl) {r=bl-1;s+=rrho*(rx-mx);}
        else {
                for(m=0;m<a[r];m++) if(w[r][m]<rx) s+=1;r--;
        }
        while (l<r) s+=float(a[l++]);
        return s/(2*dx);
}


int main() {
        int *as[iter];
        int i,j=0,k,t;float r,s,x;for (i=0;i<bl;i++) a[i]=0;
        while(j++<lwalk) put(-mx*rnd());
        while(j++<twalk) put(mx*rnd());
        for(t=0;t<iter;t++) {
                as[t]=new int[res];
                for(i=0;i<res;i++) as[t][i]=0;
                for(i=0;i<bl;i++) {
                        as[t][i/smooth]+=a[i];
                        for(j=0;j<a[i];j++) v[i][j]=udt*(1-rho(w[i][j])
                                /rhomax)+(rand()%2==1?ddt:-ddt);
                }
                for(i=0;i<bl;i++) {
                        j=0;
                        while(j<a[i]) {
                                x=w[i][j]+v[i][j];
                                if ((k=b(x))==i) w[i][j++]=x;
                                else {
                                        w[i][j]=w[i][--a[i]];
                                        v[i][j]=v[i][a[i]];
                                        if(k>=0&&k<bl) {
                                                w[k][a[k]]=x;
                                                v[k][a[k]++]=0;
                                        }
                                }
                        }
                }
                j=poisson(lrate);k=poisson(rrate);
                for(i=0;i<j;i++) put(lspeed*rnd()-mx);
                for(i=0;i<k;i++) put(mx-rspeed*rnd());
        }
        for(i=0;i<res;i++) {
                cout << mx*(2*(float(i)+0.5)/res-1);
                for(t=0;t<iter;t++) cout << "␣" << as[t][i];
                cout << endl;
        }
}
```