Appendix A

Miscellaneous void model and spot model calculations

A.1 Exact solution for a particle PDF in the void model

In chapter 2 it was shown that the PDF $\rho(x,z)$ of a particle diffusing in the void model follows the equation

$$\rho_z = 2b\frac{\partial}{\partial x}\left(\frac{\rho\eta_x}{\eta}\right) - b\rho_{xx}.$$

where $\eta(x,z)$ is the void probability density. In this section, we find the exact solution for a single particle in the presence of voids diffusing for a point orifice at the origin, corresponding to

$$\eta(x,z) = \frac{1}{\sqrt{4\pi bz}}e^{-x^2/4bz}. \tag{A.1}$$

The initial condition $\rho(x,z_0) = \delta(x-x_0)$ is used, corresponding to a particle initially located at (x,z). Two solutions are provided, the first using the method of characteristics, and the second using substitution. While the second method is more direct, the first method may have broader applicability. Although not addressed here, it appears that the first approach could handle situations where the two diffusion parameters

occurring in the PDE are different, as would occur in a spot model formulation.

A.1.1 Solution using the method of characteristics

From equation A.1, we see that

$$\frac{\eta_x}{\eta} = -\frac{x}{2by}$$

and therefore the equation for ρ becomes

$$z\rho_z = -bz\rho_{xx} - \frac{\partial}{\partial x}\left(\rho x\right).$$

To solve this equation, we first take the Fourier transform with respect to x, which yields

$$z\tilde{\rho}_z = -bz(ik)^2\tilde{\rho} - (ik)\left(i\frac{\partial}{\partial k}\right)\tilde{\rho},$$

$$z\tilde{\rho}_z - k\tilde{\rho}_k = bzk^2\tilde{\rho}.$$

This is a first order partial differential equation, so we can apply the method of characteristics. The characteristics are given by

$$\frac{dz}{z} = -\frac{dk}{k} = \frac{d\tilde{\rho}}{\tilde{\rho}bzk^2}$$

from which we obtain

$$kz = C \tag{A.2}$$

and

$$\frac{\partial\tilde{\rho}}{\partial k} = -b\tilde{\rho}kz = -b\tilde{\rho}C \qquad \Longrightarrow \qquad \tilde{\rho} = De^{-k^2zb}$$

for some constants C and D. Thus the general solution is

$$\tilde{\rho}(k,z) = F(kz)e^{-k^2zb}.$$

Taking the Fourier transform of the initial condition gives $\tilde{\rho}(k, y_0) = e^{-ikx_0}$ and hence

$$\begin{aligned} F(kz_0)e^{-k^2 z_0 b} &= e^{-ix_0 k} \\ F(kz_0) &= e^{-ix_0 k + k^2 z_0 b} \\ F(\lambda) &= \exp\left(\frac{\lambda^2 b - ix_0\lambda}{z_0}\right). \end{aligned}$$

Thus

$$\tilde{\rho}(k, z) = \exp\left(-bk^2 z\left(1 - \frac{z}{z_0}\right) - i\frac{x_0 k z}{z_0}\right)$$

and taking the inverse Fourier transform gives

$$\begin{aligned} \rho(x, z) &= \frac{1}{2\pi}\int_{-\infty}^{\infty} dk\, \exp\left(-bk^2\left(1 - \frac{z}{z_0}\right) + ik\left(x - \frac{x_0 z}{z_0}\right)\right) \\ &= \frac{1}{\sqrt{4\pi b z\left(1 - \frac{z}{z_0}\right)}} \exp \frac{-\left(x - \frac{x_0 z}{z_0}\right)^2}{4bz\left(1 - \frac{z}{z_0}\right)}. \end{aligned}$$

Thus we see that the probability density function of the particle position is a gaussian for all values of y, with mean and variance

$$\mu = \frac{x_0 z}{z_0}, \qquad \sigma^2 = 2bz\left(1 - \frac{z}{z_0}\right).$$

Although not considered here, this method of solution would potentially work in some situations where the two diffusion constants occurring in equation A.1 were not equal. In that case, the k occurring in equation A.2 would be taken to some power rather than occurring linearly.

A.1.2 Solution via substitution

The derivation of equation A.1 made use of an auxiliary quantity, $\sigma = \rho/\eta$, which satisfied

$$\sigma_z = -b\sigma_{xx},$$

representing a quantity diffusing in the opposite direction to the voids. It is therefore possible to find ρ by first solving the simpler equation for σ and then substituting back. The initial condition for ρ implies that

$$\sigma(x, z_0) = \sqrt{4\pi b z_0} e^{x_0^2/4bz_0} \delta(x - x_0)$$

which gives

$$\begin{aligned}
\sigma(x,z) &= \sqrt{\frac{z_0}{z_0 - z}} \exp\left(\frac{x_0^2}{4bz_0} - \frac{(x-x_0)^2}{4b(z_0-z)}\right) \\
&= \sqrt{\frac{z_0}{z_0 - z}} \exp\left(\frac{x_0^2 z_0 - x_0^2 z - x^2 z_0 + 2x x_0 z_0 - x_0^2 z_0}{4bz_0(z_0-z)}\right) \\
&= \sqrt{\frac{z_0}{z_0 - z}} \exp\left(\frac{-x_0^2 z - x^2 z_0 + 2x x_0 z_0}{4bz_0(z_0-z)}\right).
\end{aligned}$$

Hence

$$\begin{aligned}
\rho(x,z) &= \eta(x,z)\sigma(x,z) \\
&= \frac{1}{\sqrt{4\pi bz\left(1 - \frac{z}{z_0}\right)}} \exp\left(\frac{-x_0^2 z - x^2 z_0 + 2x x_0 z_0}{4bz_0(z_0-z)} - \frac{x^2}{4bz}\right) \\
&= \frac{1}{\sqrt{4\pi bz\left(1 - \frac{z}{z_0}\right)}} \exp\left(\frac{-x_0^2 z - x^2 z_0^2 + 2x x_0 z z_0}{4bz_0 z(z_0-z)}\right) \\
&= \frac{1}{\sqrt{4\pi bz\left(1 - \frac{z}{z_0}\right)}} \exp \frac{-\left(x - \frac{x_0 z}{z_0}\right)^2}{4bz\left(1 - \frac{z}{z_0}\right)}
\end{aligned}$$

which matches the solution from the previous method. This method of solution can be extended to handle arbitrary void and particle densities, since it only ever requires the solution of a simple diffusion equation.

A.2 Two different interpretations of the spot density drop

Suppose we consider a monodisperse granular materials composed of spherical particles with volume V_p, at a packing fraction ϕ. We consider spots, which when displaced by an amount $\mathbf{\Delta r_s}$ cause motions of particles in the opposite direction of size $\mathbf{\Delta r_p} = -w\mathbf{\Delta r_s}$. As described in chapter 2, if a spot influences N particles, then it makes sense to attribute a volume of

$$V_s = NwV_p$$

to it. In his original description of the spot model [17], Bazant proposed that it was possible to get an estimate of the size of w by looking at the drop in packing fraction $\Delta\phi$ that occurs when a granular material goes from a static state to flowing state. Bazant proposed the formula

$$w = \frac{\Delta\phi}{\phi^2}. \tag{A.3}$$

and by attributing $\Delta\phi/\phi = 1\%$ to the presence of a single spot, arrived at an estimate of $w \approx 0.017$. Attributing the density drop to the presence of several spots which are overlapped would lower the estimate of w, and thus a range of $w = 10^{-2}$ to $w = 10^{-3}$ seemed appropriate.

The justification for equation A.3 comes from considering a region of particles of volume V_0, which would be precisely filled by a single spot, and would contain N particles. Note that $V_0 \neq V_s$, since the former is the volume occupied by a spot, whereas the latter is a measure of the physical influence carried by the spot, and typically $V_s \ll V_0$. Initially, we have

$$\begin{aligned} \phi &= \frac{\text{Volume occupied by particles}}{\text{Total volume}} \\ &= \frac{NV_p}{V_0} \end{aligned} \tag{A.4}$$

Bazant proposed that when a spot enters the region it expands by the spot's vol-

ume. The total volume increases by V_s, while the amount of volume occupied by the particles remains the same. Thus

$$\begin{aligned}
\phi - \Delta\phi &= \frac{\text{Volume occupied by particles}}{\text{Total volume}} \\
&= \frac{NV_p}{V_0 + V_s} \\
&= \frac{NV_p}{V_0}\left(1 + \frac{V_s}{V_0}\right)^{-1} \\
&\approx \frac{NV_p}{V_0}\left(1 - \frac{V_s}{V_0}\right) \\
&= \frac{NV_p}{V_0} - \frac{NV_pV_s}{V_0^2} \\
&= \phi - \frac{\phi(NwV_p)}{V_0} \\
&= \phi - w\phi^2
\end{aligned}$$

from which equation A.3 directly follows.

Bazant's approach has the advantage that one can think directly about how a cluster of N particles will expand when a spot is introduced. However the current author believes that one step of the above argument is inappropriate. When the spot is introduced, the total volume is increased by V_s. The current author feels that it is more appropriate to say that when a spot is introduced, the particle volume is decreased by V_s, since when the spot volume was defined, it was directly in terms of how it would displace the particle volume in the opposite direction, and not in terms of the interstitial space.

Using this argument, equation A.4 still holds, but we have

$$\begin{aligned}
\phi - \Delta\phi &= \frac{\text{Volume occupied by particles}}{\text{Total volume}} \\
&= \frac{NV_p - V_s}{V_0} \\
&= \frac{NV_p}{V_0} - \frac{V_s}{V_0} \\
&= \phi - \frac{NwV_p}{V_0} \\
&= \phi - w\phi
\end{aligned}$$

and hence

$$w = \frac{\Delta\phi}{\phi} \tag{A.5}$$

which differs from Bazant's result by a factor of ϕ.

It is reasonable to suggest that the details of the process by which a spot enters a packing will have an effect on the amount of dilation, and thus it could be argued that both approaches could be valid under different physical assumptions. However, it is possible to construct an argument where purely by considering a spot moving in the bulk of a granular material, one can obtain a result which is consistent with A.5.

To do this, it is helpful to consider spots having the shape of a cube with side length L, as there is nothing in the above volume arguments which requires that spots be spherical. Consider a box with dimensions $2L \times L \times L$ which is initially filled with particles at a uniform packing fraction ϕ. The box is divided into a left half and a right half. Imagine that spot is initially located in the right side, and that it is displaced a distance L into the left of the box. The particles will be displaced to the right by a distance wL, and thus a box of particles of dimensions $wL \times L \times L$ is displaced from the left half of the box to the right half. Before the spot moves, the packing fraction in the left half of the box is given by

$$\begin{aligned} \phi &= \frac{\text{Volume occupied by particles}}{\text{Total volume}} \\ &= \frac{NV_p}{L^3} \end{aligned}$$

and after the spot moves, the packing fraction in the left half of the box is given by

$$\begin{aligned} \phi - \Delta\phi &= \frac{\text{Volume occupied by particles}}{\text{Total volume}} \\ &= \frac{NV_p - \phi L^3 w}{L^3} \\ &= \phi - w\phi. \end{aligned}$$

This argument was created purely by considering a spot motion in the bulk, with no ambiguities about how a spot would enter at a free surface. As the spot moves to

the left of the box, we see that is displaces a volume of V_s particle volume out, rather than introducing V_s interstitial space. We thus conclude that equation A.5 is more appropriate.

For the current purposes, where we wish to gain an order of magnitude estimate for w, the difference between equations A.3 and A.5 is not that significant, since the additional ϕ term is an order 1 quantity. However, the result does suggest that it is more appropriate to think of spots as carrying negative particle volume, as opposed to extra interstitial space, which is an important physical distinction.

Appendix B

Numerical solution of the Kinematic Model in the cylindrical reactor geometry

In the Kinematic Model for drainage the vertical downward velocity u in the container is assumed to follow a diffusion equation of the form

$$\frac{\partial v}{\partial z} = b\nabla_{\perp}^{2} v$$

where $\nabla_{\perp}^{2}$ is the horizontal Laplacian. By exploiting the axial symmetry, v can be treated as a function of z and r only. In cylindrical coordinates the Laplacian is

$$\begin{aligned} \frac{\partial v}{\partial z} &= b\frac{1}{r}\frac{\partial}{\partial r}\left(r\frac{\partial v}{\partial r}\right) \\ &= b\frac{\partial^2 v}{\partial r^2} + b\frac{1}{r}\frac{\partial v}{\partial r}. \end{aligned}$$

The radial velocity component is given by

$$u = b\frac{\partial v}{\partial r}$$

and by enforcing that the velocity field at the wall must be tangential to the wall, we can obtain boundary conditions for solving v.

To solve the above equation in a cylinder is straightforward, since we can make use of a rectangular grid. The boundary condition reduces to $v_r = 0$ at the wall. However, to solve this equation in the reactor geometry, we must also consider the complication of the radius of the wall, R, being a function of z. To ensure accurate resolution in the numerical solution of v at the wall, we introduce a new coordinate $\lambda = r/R(z), \eta = z$, which then allows us to solve for u over the range $0 < \lambda < 1$. Under this change of variables, the partial derivatives transform according to

$$\begin{aligned} \frac{\partial}{\partial r} &= \frac{1}{R(\eta)}\frac{\partial}{\partial \lambda} \\ \frac{\partial}{\partial z} &= \frac{\partial}{\partial y} - \frac{\lambda R'(\eta)}{R(\eta)}\frac{\partial}{\partial \lambda}. \end{aligned}$$

In the transformed coordinates

$$R^2 v_\eta = \frac{b}{\lambda} v_\lambda + b v_{\lambda\lambda} + \lambda R R' v_\lambda.$$

To ensure differentiability at $r = 0$, we use the boundary condition

$$\left.\frac{\partial v}{\partial \lambda}\right|_{\lambda=0} = 0, \tag{B.1}$$

and by ensuring zero normal velocity at the wall we find that

$$\left.\frac{\partial v}{\partial \lambda}\right|_{\lambda=1} = -\frac{v R' R}{b}. \tag{B.2}$$

To numerically solve this partial differential equation, we make use of the implicit Crank-Nicholson integration scheme. We write $v_j^n = v(j\Delta\lambda, n\Delta\eta)$, and solve in the range $j = 0, 1, \ldots, N$ where $N = \Delta\lambda^{-1}$. Away from the end points, the Crank-

Nicholson scheme tells us that

$$\begin{aligned}\frac{v_j^{n+1}-v_j^n}{\Delta\eta} &= \frac{b}{2\Delta\lambda^2R^2}\big(v_{j+1}^{n+1}-2v_j^{n+1}+v_{j-1}^{n+1}+v_{j+1}^n\\ &\quad -2v_j^n+v_{j-1}^n\big)+\left(\frac{b}{4j\Delta\lambda^2R^2}+\frac{jR'}{4R}\right)\\ &\quad \times\big(v_{j+1}^{n+1}-v_{j-1}^{n+1}+v_{j+1}^n-v_{j-1}^n\big),\end{aligned}$$

where all references to R and R' are evaluated at $\eta=\Delta\eta(j+\frac{1}{2})$. If $j=0$, then by reference to equation B.1, we find that

$$\frac{v_0^{n+1}-v_0^n}{\Delta\eta}=\frac{b}{\Delta\lambda^2R^2}\left(v_1^{n+1}-v_0^{n+1}+v_1^n-v_0^n\right).$$

Similarly, for $j=N$, by reference to equation B.2, we see that effectively

$$\frac{v_{N+1}^n-v_{N-1}^n}{2\Delta\lambda}=-\frac{v_N^nR'R}{b}$$

and hence

$$\begin{aligned}\frac{v_N^{n+1}-v_N^n}{\Delta\eta} &= \frac{b}{\Delta\lambda^2R^2}\left(v_{N-1}^{n+1}-v_N^{n+1}+v_{N-1}^n-v_N^n\right)\\ &\quad -\left(\frac{(2N+1)R'}{2R}+\frac{R'^2}{2b}\right)\left(v_N^{n+1}+v_N^n\right).\end{aligned}$$

If we write $\mathbf{v}^n=(v_0^n,v_1^n,\ldots,v_N^n)^T$, then the above numerical scheme can be written in the form $S\mathbf{v}^{n+1}=T\mathbf{v}^n$ where S and T are tridiagonal matrices; this system can be efficiently solved by recursion in $O(N)$ time. The above scheme was implemented in C++, and gives extremely satisfactory results, even with a relatively small number of gridpoints.

Appendix C

The spot model simulation code

C.1 Overview

This appendix documents the computer codes that were developed to carry out the spot model simulations in chapter 3. Although the simulations presented in this thesis concentrated wholly on the case of granular drainage, the spot model microscopic mechanism introduced in figure 3-2 could potentially be applied to many other situations, and it was therefore chosen to develop the code in a method that could be easily adapted to other purposes.

The code is written as a library of routines in object-oriented C++, that aims to provide a flexible environment for the general task of managing particles in a container. Routines exist for creating particles in the container, outputting a snapshot of particles to a file, and carrying out diagnostic tests (such as the calculation of the radial distribution function). Most importantly, the library provides routines to carry out the spot motion and elastic relaxation.

Since simulations of granular materials involve a large number of particles, efficiency was a key consideration in the development of the library. The simplest method of storing a total of N particle positions would be as a single, unsorted list. However, finding all the particles influenced by a single spot would require looking over the entire list, which would be an $O(N)$ calculation. To avoid this, the library divides the simulation up into a rectangular grid of regions, and each region keeps a list of

all particles within it. When applying a spot motion, only the particles in regions overlapping the spot need to be tested, significantly reducing the computational complexity. This arrangement of data does bring with it some computational challenges. For example, when a spot is moved, the code must first find which regions to test, and it must also deal with the possible migration of particles within regions. However, these computational difficulties can be packaged away as low-level routines in the library. Note that the neighbor list technique [6], popular with in other molecular simulations, is not employed.

In the following section, the class structure and code organization are discussed. In section C.3 the individual routines are described in detail. Section C.4 provides two simple examples of using the routines in the code, and section C.6 contains the code listings.

C.2 Code structure

The code presented in this chapter is a trimmed-down version of the original library, which concentrates on the core routines only. For reasons of clarity and economy, routines that were either experimental, or created for a specific purpose, have been omitted. The version of the library presented here consists of four files: "vec.hh", "vec.cc", "container.hh", and "container.cc" which are listed in subsections C.6.1 to C.6.4.

C.2.1 Vector manipulation

Since the spot model simulation requires frequent manipulation of vectors, a simple vector structure named `vec` was created, which is listed in "vec.hh" and "vec.cc". This structure consists of three floating point numbers `x`, `y`, and `z` to represent three coordinates, which can be accessed and set independently. A simple example code would be:

```
#include <cstdio>
#include <iostream>
```

```
using namespace std;            // Standard C++ header

#include "vec.cc"               // Load the structure

int main() {
  vec a;                        // Create two vectors
  a.x=3;a.y=4;a.z=5;            // Set coordinates
  a.z-=4;                       // Subtract 4 from z co-ord

  cout c.x << " " << c.y;
  cout << " " << c.z << endl;   // Output to screen
}
```

A powerful feature of the C++ language is the ability to "overload" the conventional arithmetic operators (`+`, `-`, `*`, `/`) when they are applied to new structures. Routines were therefore constructed so that these operators took on their usual meaning when applied to `vec` objects. In addition simple routines were created to carry out typical vector operations, such as creating a scalar product, or finding the magnitude of a vector. The code listing below provides some examples:

```
#include <cstdio>
#include <iostream>
using namespace std;            // Standard C++ header

#include "vec.cc"               // Load the structure

int main() {
  vec a(1,2,3),b(2,3,1),c;      // Create two vectors
  float d,e;                    // Initialize two numbers
  c=a+b;                        // Add a and b together
  c/=3;                         // Divide the result by 3
  d=sp(a,b);                    // Scalar product of a and b
  e=radsq(a);                   // Magnitude squared of a

  cout c.x << " " << c.y << " " << c.z << endl;
  cout d << " " << e << endl;   // Output to screen
}
```

The spot model code makes extensive use of the `vec` structure. It considerably simplifies the source code, and lowers the chance of introducing the common coding error of accidentally transposing `x`, `y`, and `z` in vector manipulations. It also provides a convenient method of passing vectors (or references to them) between functions.

All low level vector operations are specified using the `inline` command to improve performance.

C.2.2 The container class hierarchy

The files "container.cc" and "container.hh" contain the bulk of the spot model library. There are three main classes that form a hierarchy. At the lowest level is the `particle` structure, containing all the data associated with a single particle. In the code presented here, this contains a position vector `v` and a numerical label `n`, but in a general case it could contain other information, such as the particle diameter, mass, or rotation characteristics.

At the middle layer is the `region` class, which represents a particular box-shaped region of particles in the container. This contains an fixed-size array of particle elements, and several low-level routines for accessing and handling the particles.

The `container` class is at the top layer, and it is the most important for running a simulation. It represents a box-shaped container of particles, and it has a constructor of the form:

```
container::container(float minx, float maxx, float miny,
                     float maxy, float minz, float maxz,
                     int xn, int yn, int zn);
```

The first six arguments set the bounds of the container in each coordinate. The following three arguments set how many regions in each direction the container is to be subdivided into. When a container object is created, it initializes an array of `xn*yn*zn` regions. The container object has a large number of functions associated with it, which are described in detail in the following section.

C.3 Spot model library commands

C.3.1 Particle input

```
void put(int n,vec &p);
void put(int n,float x,float y,float z);
```

This function adds a particle to a container. Two versions of the routine are provided, with one accepting a vector and the other accepting three individual coordinates. The code finds the region which contains this position and gives the particle to that region.

```
void import();
```

This function reads in particle positions from the standard input. Each particle is represented by one line containing four numbers separated by spaces. The first number is the particle's numerical label, and the following three following three are the particle coordinates. The code sorts the particles into the correct spatial regions.

```
void importrestart(fstream &rf);
```

This function reads in a snapshot of particle positions from an open file stream `rf` that are stored in the GranFlow snapshot format. The code sorts the particles into the correct spatial regions.

```
void clear();
```

This function clears the container of particles, by zeroing the particle counts in all the regions.

C.3.2 Particle output

```
void dump(char *filename);
void dumpraster(char *filename);
void dumppov(char *filename);
```

These functions dump a snapshot of all the particles in the container to a file. The `dump` routine outputs a text file with one line of information about each particle. The `dumpraster` and `dumppov` create input files for the Raster3D and POV-Ray raytracers respectively.

```
void dumprestart(int t, fstream &of);
```

This function writes a snapshot of all particle positions in the GranFlow snapshot format to an open file stream `of`. The timestep in the GranFlow header is given by the integer `t`.

```
void regionshot(float minx,float maxx,float miny,
                float maxy,float minz,float maxz);
```

This function outputs all the particle positions within a specific subregion of the container to the standard output.

C.3.3 Spot motion and elastic relaxation

```
int count(vec &p,float r);
```

This function returns the number of particles that are within a distance `r` of the vector `v`. It is useful for telling whether a spot is still within the particle packing. To carry this out, the code first finds which regions the test area overlaps with, and then tests all the particles within those regions.

```
void spot(vec &p,vec &v,float r);
```

This function carries out a spot displacement `v` on all particles which are within a distance `r` of position `p`. The code first calculates which regions the test area overlaps with, and then tests all the particles within those regions, moving the particles which are within range. If a particle is within range, it is moved. If a particle's new position is no longer within its original region, then it is migrated to the region it now occupies. The affected regions are tested in such an order to ensure that any particles which migrate always do so to a region which was already tested. In a one-dimensional situation, this would mean that if the displacement vector pointed to the left, then the regions would be tested from left to right. This avoids the possibility of a migrated particle being displaced more than once.

```
void spotg(vec &p,vec &v,float r,float l);
```

This function carries out a Gaussian-smoothed spot displacement centered on a position `p`. Particles which are a distance `s` away from `p` experience a displacement of `v*exp(-s*s/(2*l*l))`. Particles outside a cutoff radius `r` experience no displacement.

```
void relaxslow(vec &p,float r,float s
               float force,float damp,int steps);
```

This function applies an elastic relaxation on a sphere of radius `s`, centered at `p`. Particles with radii between `s` and `r` fixed to avoid long range disruption. The magnitude of the correcting force is given by `force`. After each step, a velocity damping of magnitude `damp` is applied. Vertical walls also contribute a force, but the container bottom does not, to prevent jamming at the orifice. Using pointers, particle positions are updated in situ and never copied out. A final step is applied to move any particles that moved into a different region – this is done in reverse to prevent memory conflicts. Memory for the buffers is declared as static to prevent unnecessary memory reallocation each time the routine is called.

```
void relax(vec &p,float r,float s,
           float force,float damp,int steps);
```

This function carries out exactly the same task as `relaxslow`, except that during the relaxation process, the particles are spatially sorted into an $n \times n \times n$ grid. This significantly speeds up the calculation, since to find the neighbors of a particle requires a search over only a small subregion of the grid, and not a loop over all the affected particles. The value of n can be set in the code by declaring the quantity `rgrid`. The default value is `rgrid=6`, which was found to give the best results in the hopper drainage simulations considered here. The quantity `rbuf` sets the memory allocation for each grid element, and the default value is 80. The results of this function should agree with the results of `relaxslow` up to rounding error.

C.3.4 Diagnostic routines

```
void regioncount();
```

This function prints a list of the regions and the number of particles that each contains.

```
void gofr(int *b);
```

This command calculates the frequency distribution of interparticle separations in the entire container. The results for separations from radii 0 to $8d$ and stores the results in a 4,000 element matrix `b`. Calculating $g(r)$ requires normalizing the resulting distribution by an appropriate factor. In an infinite container, the normalizing factor is proportional to r^2, although for constrained geometries it is more complicated.

```
void bondangle(int *b,float r);
```

This function calculates the bond angle distribution in the entire container, based on defining two particles as neighboring if their centers are a distance `r` apart. All triplets of particles are considered, and the results are stored in a 4,000 element matrix `b`.

```
void spaces(float xs,float ys,float zs,
            float mnz,float mxz,int *b);
```

This function computes the free space size distribution in the strip of the container in the range `mnz<z<mxz`. The region is covered by an `xs` by `ys` by `zs` grid, and at each point in the grid which is not inside a particle, the maximum radius of void that will fit at that location is computed. The frequency distribution of radii over the range 0 to $1.5d$ is stored in a 300 element array `b`.

C.4 Example: A test of the relaxation scheme

The code listed in subsection C.6.5 was written as a simple test of the spot model library. It creates a container object using dimensions $-25d < x < 25d$, $-4d < y <$

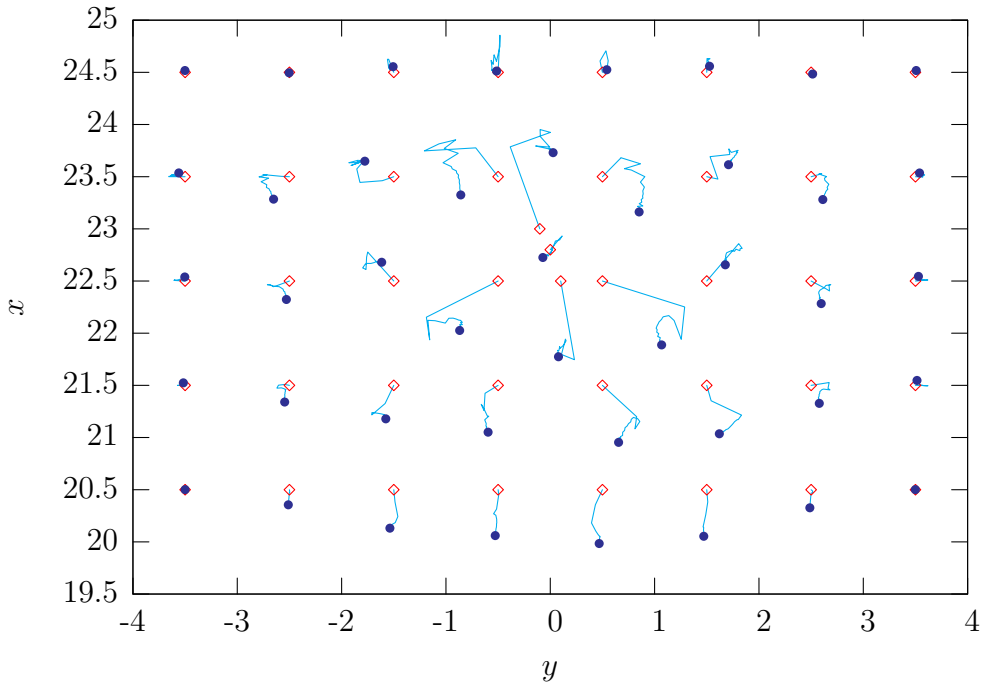

Figure C-1: Results of the "relaxtest.cc" example program. A rectangular grid of particles is initialized, along with three extra at $(23, -0.1, 10)$, $(22.8, 0, 10)$, and $(22.5, 0.1, 10)$. Their initial positions are shown by the red diamonds. A twenty step relaxation process is carried out, shown by the light blue lines, until the particles reach their final positions, shown by the dark blue circles.

$4d$, and $0 < z < 150d$, and then introduces a two dimensional rectangular grid of particles in the (x, y) plane at $z = 10d$. Three extra particles are then positioned at $(23, -0.1, 10)$, $(22.8, 0, 10)$, and $(22.5, 0.1, 10)$. A twenty step relaxation process is then carried out, using `force=0.7` and `damp=0.4`. The results of the relaxation process after n steps are outputted to text files named `relax`n, where $n = 0, 1, \ldots, 20$.

Figure C-1 shows the initial particle positions, final particle positions, and the traces of the particle trajectories. The particles in the square lattice are pushed apart to accommodate for the other particles. After twenty steps, the maximum overlap of particles in the grid is $0.06d$.

C.5 Example: A spot model simulation code

The spot simulation code that was used to generate the results of chapter 3 is listed in subsection C.6.6. First, standard header files and the spot model library are loaded. In the next section, all the free parameters are defined. Four of these (the spot insertion rate, the spot move rate, the displacement scale factor, and the spot radius) directly relate to the calibrated parameters. In addition to these, the parameters `xst`, `yst`, and `zst` set the size of the random walk step, to ensure that the walk has the correct diffusion parameter. The parameter `clu` sets the number of spots that are introduced at each insertion event; in the current simulation, only one spot was introduced at each time, but this allows for the possibility of introducing spots in groups. The parameter `max` sets the size of the array for the spot positions, and is chosen to be large enough to handle when the simulation reaches steady state, and the number of spots being inserted is balanced by the number of spots exiting at the free surface. The parameters `xbu` and `ybu` affect how close the spot centers are allowed to come to the side walls, and can be used to adjust the boundary layer behavior. After these declarations, the code defines a simple function `rnd()` which returns a floating point number in the range 0 to 1.

The main program then starts, and after declaring some variables, the code initializes the container object, with size $-25d < x < 25d, -4d < y < 4d, 0 < z < 150d$, divided into a $10 \times 3 \times 30$ grid of regions. Particle positions from a DEM simulation are then imported from the standard input. An output file stream is then open.

The main loop of the simulation then starts. The simulation runs from `t=0` to `t=500`, and outputs a snapshot every two time units. The simulation is event driven, with insertion events happening at a rate `ipr`, and movement of existing spots happening at a rate of `mpr`. Based on there being `n` spots in the container, the code first calculates the time to the next event, by sampling from an exponential distribution with rate `ipr+n*mpr`. If a snapshot needed to be recorded between this event and the previous one, then it is done so here.

Based on the number of spots in the container, the code then randomly decides

whether the current event is an insertion or a move. To begin with, when there are no spots in the container, the event will always be an insertion. The code then initializes `clu` new spots at the orifice, storing their positions in the vector array `s`. Their z coordinate is set to `-rad/3`, and their x and y coordinates are chosen uniformly in the range from $-1.5d$ to $1.5d$.

If a movement event occurs, then the code first chooses an existing spot at random to be moved. If its z coordinate is less than `rad/3`, it is treated as being in the orifice it moves upwards only by `zst`, with no horizontal motion. It is first moved half of the way, and then a spot motion and relaxation are applied, centered on this intermediate position. After this motion, the spot is moved the rest of the way.

If the z coordinate of the moving spot is bigger than `rad/3` then it is treated as being in the bulk of the container, undergoing the random walk. The displacement of the spot is stored in `v` and is chosen randomly from has four possible directions: it can either take a positive or negative step in the x direction, or a positive and negative step in the y direction. If the motion would take the spot outside its range, then the displacement vector is truncated as necessary. The spot is then moved half-way, its influence is applied, and it is moved the rest of the way.

When the spot's vertical coordinate reaches $130d$, the spot is deleted. This is done by the command `s[b]=s[--n]`, which lowers the number of spots by one, and copies the last element of the array over the spot being deleted, to ensure that the remaining spots form a contiguous block of memory.

C.6 Spot model code listings

C.6.1 vec.hh

```
// Fast spot model code — vector object header file
//
// Author  : Chris H. Rycroft
// Version : 2.0
// Date    : July 26th 2004

#ifndef VEC_HH
#define VEC_HH
```

```
class vec {
  public:
    float x,y,z;
    vec () {};
    inline vec (float a, float b, float c);
    inline vec operator+ (vec p);
    inline vec operator- (vec p);
    inline vec operator* (float a);
    inline vec operator/ (float a);
    inline void operator+= (vec p);
    inline void operator-= (vec p);
    inline void operator*= (float a);
    inline void operator/= (float a);
    inline vec operator* (vec a);
};

struct vptr {
  int n;
  vec *q;
};

struct intrec {
  int lx,ly,lz,ux,uy,uz;
};

inline float radsq(vec &r,vec &s);
inline float radsq(vec &r);
inline float sp(vec &r,vec &s);
inline float stp(vec &r,vec &s,vec &t);

#endif
```

C.6.2 vec.cc

```
// Fast spot model code - vector object routines
//
// Author  : Chris H. Rycroft
// Version : 2.0
// Date    : July 26th 2004

#ifndef VEC_CC
#define VEC_CC

#include "vec.hh"

inline vec::vec (float a,float b,float c) {
   x=a;y=b;z=c;
};

inline vec vec::operator+ (vec p) {
  vec t;
  t.x=x+p.x;t.y=y+p.y;t.z=z+p.z;
  return t;
};

inline vec vec::operator- (vec p) {
  vec t;
  t.x=x-p.x;t.y=y-p.y;t.z=z-p.z;
  return t;
};

inline vec vec::operator* (float a) {
  vec t;
  t.x=x*a;t.y=y*a;t.z=z*a;
  return t;
```

```
};

inline vec vec::operator/ (float a) {
  vec t;
  t.x=x/a;t.y=y/a;t.z=z/a;
  return t;
};

inline void vec::operator+= (vec p) {
  x+=p.x;y+=p.y;z+=p.z;
};

inline void vec::operator-= (vec p) {
  x-=p.x;y-=p.y;z-=p.z;
};

inline void vec::operator*= (float a) {
  x*=a;y*=a;z*=a;
};

inline void vec::operator/= (float a) {
  x/=a;y/=a;z/=a;
};

inline float radsq(vec &r,vec &s) {
  return (r.x-s.x)*(r.x-s.x)+(r.y-s.y)*(r.y-s.y)+(r.z-s.z)*(r.z-s.z);
};

inline float radsq(vec &r) {
  return r.x*r.x+r.y*r.y+r.z*r.z;
};

inline float sp(vec &r,vec &s) {
  return r.x*s.x+r.y*s.y+r.z*s.z;
};

inline vec vec::operator* (vec a) {
  vec t;
  t.x=y*a.z-z*a.y;t.y=z*a.x-x*a.z;t.z=x*a.y-y*a.x;
  return t;
};

inline float stp(vec &r,vec &s,vec &t) {
  return r.x*s.y*t.z+r.y*s.z*t.x+r.z*s.x*t.y-r.z*s.y*t.x-r.y*s.x*t.z-r.x*s.z*t.y;
};

#endif
```

C.6.3 container.hh

```
// Fast spot model code - container object header file
//
// Author  : Chris H. Rycroft
// Version : 2.0
// Date    : July 26th 2004

#ifndef CONTAINER_HH
#define CONTAINER_HH

#ifndef maxpoints
#define maxpoints 100
#endif

struct particle {
  vec p;
```

```
  int n;
};

struct particlev {
  particle data;
  vec v;
};

struct particlevdiag : particlev {
  vec op;
};

class region {
  public:
    region();
    void dump(fstream &of);
    void dumpraster(fstream &of);
    void dumppov(fstream &of);
    void dumprestart(fstream &of);
    void put(int n,vec &p);
    particle data[buf];
    int num;
};

class container {
  public:
    container(float minx,float maxx,float miny,float maxy,
              float minz,float maxz,int xn,int yn,int zn);
    void dump(char *filename);
    void dumpraster(char *filename);
    void dumppov(char *filename);
    void dumprestart(int t, fstream &of);
    void import();
    void importrestart(fstream &rf);
    void regioncount();
    void regionshot(float minx,float maxx,float miny,float maxy,float minz,float maxz);
    void put(int n,vec &p);
    void put(int n,float x,float y,float z);
    void spot(vec &p,vec &v,float r);
    void spotg(vec &p,vec &v,float r,float l);
    int count(vec &p,float r);
    void relax(vec &p,float r,float s,float force,float damp,int steps);
    void relaxslow(vec &p,float r,float s,float force,float damp,int steps);
    void gofr(int *b);
    void bondangle(int *b,float r);
    void spaces(float xs,float ys,float zs,float mnz,float mxz,int *b);
    void clear();
    region *reg;
  private:
    float xmin,xmax,ymin,ymax,zmin,zmax;
    float xsp,ysp,zsp;
    int nx,ny,nz;
    int blocks;
    inline int myint(float co);
    inline intrec boundbox(vec &p,float r);
};

#endif
```

C.6.4 container.cc

```
// Fast spot model code — container object routines
//
// Author  : Chris H. Rycroft
// Version : 2.0
```

```
// Date     : July 26th 2004

#ifndef CONTAINER_CC
#define CONTAINER_CC

#include "container.hh"

#ifndef rgrid
#define rgrid 6
#endif

#ifndef rbuf
#define rbuf 80
#endif

// Region constructor - sets initial number of particles to zero.
region::region() {
  num=0;
};

// Dumps all particles in a region to an open file stream.
void region::dump(fstream &of) {
  int j=0;
  while(j<num) {
    of << data[j].n << " " << data[j].p.x << " ";
    of << data[j].p.y << " " << data[j++].p.z << endl;
  }
};

// Dumps all particles in a region to an open file stream.
void region::dumprestart(fstream &of) {
  int j=0;
  while(j<num) {
    of << data[j].n << " 1 1 " << data[j].p.x << " ";
    of << data[j].p.y << " " << data[j++].p.z << endl;
  }
};

// Dumps all particles in a region to an open file stream.
void region::dumpraster(fstream &of) {
  int j=0;
  while(j<num) {
    of << "2" << endl;
    of << data[j].p.x << " " << data[j].p.y << " " << data[j].p.z;
    if (data[j++].n==0) of << " 0.5 0.25 0.25 0.40" << endl;
    else of << " 0.5 1.00 1.00 0.80" << endl;
  }
};

// Dumps all particles in a region to an open file stream.
void region::dumppov(fstream &of) {
  int j=0;
  while(j<num) {
    of << "sphere{<" << data[j].p.x << "," << data[j].p.y;
    of << "," << data[j].p.z << ">,0.5}" << endl;
    j++;
  }
};

// Adds a particle to a region.
void region::put(int n,vec &p) {
  if (num<=buf) {
    data[num].n=n;
    data[num++].p=p;
  }
};

// Initializes a particle container object.
```

```
container::container(float minx,float maxx,float miny,float maxy,
                     float minz,float maxz,int xn,int yn,int zn) {
  xmin=minx;xmax=maxx;
  ymin=miny;ymax=maxy;
  zmin=minz;zmax=maxz;
  nx=xn;ny=yn;nz=zn;
  xsp=float(nx)/(xmax-xmin);
  ysp=float(ny)/(ymax-ymin);
  zsp=float(nz)/(zmax-zmin);
  blocks=nx*ny*nz;
  reg=new region[nx*ny*nz];
};

// Dumps particle positions and characteristic to a specified file.
void container::dump(char * filename) {
  int i;
  fstream file;
  file.open(filename,fstream::out|fstream::trunc);
  for(i=0;i<blocks;i++) {
    reg[i].dump(file);
  }
  file.close();
};

// Dumps a particle snapshot to an open restart file stream.
void container::dumprestart(int t,fstream &of) {
  int i,j=0;
  for(i=0;i<blocks;i++) {
    j+=reg[i].num;
  }
  of << "␣ITEM:␣TIMESTEP" << endl << t << endl;
  of << "␣ITEM:␣NUMBER␣OF␣ATOMS" << endl << j << endl;
  of << "␣ITEM:␣BOX␣BOUNDS" << endl;
  of << xmin << "␣" << xmax << endl;
  of << ymin << "␣" << ymax << endl;
  of << zmin << "␣" << zmax << endl;
  of << "␣ITEM:␣ATOMS" << endl;
  for(i=0;i<blocks;i++) {
    reg[i].dumprestart(of);
  }
};

// Dumps particle positions to a file readable by the raster3D raytracer.
void container::dumpraster(char * filename) {
  int i;
  fstream file;
  file.open(filename,fstream::out|fstream::trunc);
  file << "@header.r3d" << endl;
  for(i=0;i<blocks;i++) {
    reg[i].dumpraster(file);
  }
  file.close();
};

// Dumps particle positions to a file readable by the POVRay raytracer.
void container::dumppov(char * filename) {
  int i;
  fstream file;
  file.open(filename,fstream::out|fstream::trunc);
  file << "#declare␣container=union{" << endl;
  for(i=0;i<blocks;i++) {
    reg[i].dumppov(file);
  }
  file << "}" << endl;
  file.close();
};

// Adds a particle to the container, placing it in the correct region.
```

```
// (Vector input)
void container::put(int n,vec &p) {
  int mx,my,mz;
  mx=myint((p.x-xmin)*xsp);
  my=myint((p.y-ymin)*ysp);
  mz=myint((p.z-zmin)*zsp);
  if (mx>=0&&mx<nx&&my>=0&&my<ny&&mz>=0&&mz<nz) {
    reg[mx+nx*(my+ny*mz)].put(n,p);
  }
};

// Adds a particle to the container, placing it in the correct region.
// (Three scalar input)
void container::put(int n,float x,float y,float z) {
  int mx,my,mz;
  vec p;
  mx=myint((x-xmin)*xsp);
  my=myint((y-ymin)*ysp);
  mz=myint((z-zmin)*zsp);
  if (mx>=0&&mx<nx&&my>=0&&my<ny&&mz>=0&&mz<nz) {
    p.x=x;p.y=y;p.z=z;
    reg[mx+nx*(my+ny*mz)].put(n,p);
  }

};

//Reads a particle position file from stdin and puts them into the container.
void container::import() {
  int n;vec p;
  cin >> n >> p.x >> p.y >> p.z;
  while (!cin.eof()) {
    put(n,p);
    cin >> n >> p.x >> p.y >> p.z;
  }
};

//Reads in a snapshot from an open restart file stream.
void container::importrestart(fstream &rf) {
  int i,d,e,n;float f;vec p;char q[256];
  for(i=0;i<blocks;i++) reg[i].num=0;
  for(i=0;i<3;i++) rf.getline(q,256);
  rf >> n;
  for(i=0;i<6;i++) rf.getline(q,256);
  for(i=0;i<n;i++) {
    rf >> d >> e >> f >> p.x >> p.y >> p.z;
    put(d,p);
  }
  rf.getline(q,256);
};

// Move a spot in the container, position p, velocity v, radius r.
// Each particle is moved individually and shifted into a new region if
// necessary. Only the regions which can be affected are tested. Regions are
// tested in the correct order to avoid moving a particle twice.
void container::spot(vec &p,vec &v,float r) {
  int ax,ay,az,i,j,k,fx,fy,fz;
  vec t;intrec a;
  bool rx,ry,rz;
  a=boundbox(p,r);r*=r;
  rx=(v.x<0);ry=(v.y<0);rz=(v.z<0);
  for (az=rz?a.lz:a.uz;rz?(az<=a.uz):(az>=a.lz);az+=rz?1:-1) {
    for (ay=ry?a.ly:a.uy;ry?(ay<=a.uy):(ay>=a.ly);ay+=ry?1:-1) {
      for (ax=rx?a.lx:a.ux;rx?(ax<=a.ux):(ax>=a.lx);ax+=rx?1:-1) {
        i=ax+nx*(ay+ny*az);
        j=0;
        while(j<reg[i].num) {
          if (radsq(reg[i].data[j].p,p)<r) {
            t=reg[i].data[j].p+v;
```

```
            fx=myint((t.x-xmin)*xsp);
            fy=myint((t.y-ymin)*ysp);
            fz=myint((t.z-zmin)*zsp);
            if (fx==ax&&fy==ay&&fz==az) reg[i].data[j++].p=t;
            else {
              if (fx>=0&&fx<nx&&fy>=0&&fy<ny&&fz>=0&&fz<nz) {
                k=fx+nx*(fy+ny*fz);
                reg[k].data[reg[k].num].n=reg[i].data[j].n;
                reg[k].data[reg[k].num++].p=t;
              }
              reg[i].data[j]=reg[i].data[--reg[i].num];
            }
          }
          else j++;
        }
      }
    }
  }
};

// Move a gaussian spot in the container, position p, velocity v, radius r.
// Each particle is moved individually and shifted into a new region if
// necessary. Only the regions which can be affected are tested. Regions are
// tested in the correct order to avoid moving a particle twice.
void container::spotg(vec &p,vec &v,float r,float l) {
  int ax,ay,az,i,j,k,fx,fy,fz;r*=r;l=-1/(l*l*2);float s;
  vec t;intrec a;
  bool rx,ry,rz;
  a=boundbox(p,r);
  rx=(v.x<0);ry=(v.y<0);rz=(v.z<0);
  for (az=rz?a.lz:a.uz;rz?(az<=a.uz):(az>=a.lz);az+=rz?1:-1) {
    for (ay=ry?a.ly:a.uy;ry?(ay<=a.uy):(ay>=a.ly);ay+=ry?1:-1) {
      for (ax=rx?a.lx:a.ux;rx?(ax<=a.ux):(ax>=a.lx);ax+=rx?1:-1) {
        i=ax+nx*(ay+ny*az);
        j=0;
        while(j<reg[i].num) {
          s=radsq(reg[i].data[j].p,p);
          if (s<r) {
            t=reg[i].data[j].p+v*exp(s*l);
            fx=myint((t.x-xmin)*xsp);
            fy=myint((t.y-ymin)*ysp);
            fz=myint((t.z-zmin)*zsp);
            if (fx==ax&&fy==ay&&fz==az) reg[i].data[j++].p=t;
            else {
              if (fx>=0&&fx<nx&&fy>=0&&fy<ny&&fz>=0&&fz<nz) {
                k=fx+nx*(fy+ny*fz);
                reg[k].data[reg[k].num].n=reg[i].data[j].n;
                reg[k].data[reg[k].num++].p=t;
              }
              reg[i].data[j]=reg[i].data[--reg[i].num];
            }
          }
          else j++;
        }
      }
    }
  }
};

// Applies elastic relaxation on a sphere of radius s, centered at p. Particles
// with radii between s and r fixed to avoid long range distruption. The
// magnitude of the correcting force is given by "force". After each step, a
// velocity damping of magnitude "damp" is applied. Vertical walls also
// contribute a force, but the container bottom does not, to prevent jamming at
// the orifice. Using pointers, particle positions are updated in situ and
// never copied out. A final step is applied to move any particles that moved
// into a different region - this is done in reverse to prevent memory
// conflicts. Memory for the buffers is static to prevent lots of memory
```

```
// reallocation each time the routine is called.
void container::relaxslow(vec &p,float r,float s,
                          float force,float damp,int steps) {
  static particle * e[1024];
  static particle * c[1024];
  static vec v[1024];
  static int b[1024];
  vec d;intrec a;float t;
  int ax,ay,az,ct=0,cs=0,i,j;
  a=boundbox(p,s);
  for (az=a.lz;az<=a.uz;az++) {
    for (ay=a.ly;ay<=a.uy;ay++) {
      for (ax=a.lx;ax<=a.ux;ax++) {
        i=ax+nx*(ay+ny*az);
        for(j=0;j<reg[i].num;j++) {
          t=radsq(reg[i].data[j].p,p);
          if (t<s*s) {
            if (t<r*r) {
              e[ct]=&(reg[i].data[j]);
              v[ct].x=0;v[ct].y=0;v[ct].z=0;
              b[ct++]=i;
            }
            else c[cs++]=&(reg[i].data[j]);
          }
        }
      }
    }
  }
  for(ax=0;ax<steps;ax++) {
    for(i=0;i<ct;i++) {
      for(j=i+1;j<ct;j++) {
        d=e[i]->p-e[j]->p;
        t=radsq(d);
        if (t<1) {t=sqrt(t);d*=force*(1-t)/t;v[i]+=d;v[j]-=d;}
      }
      for(j=0;j<cs;j++) {
        d=e[i]->p-c[j]->p;
        t=radsq(d);
        if (t<1) {t=sqrt(t);d*=force*(1-t)/t;v[i]+=d;}
      }
      if (e[i]->p.x<xmin+0.5) v[i].x-=force*(e[i]->p.x-xmin-0.5);
      if (e[i]->p.x>xmax-0.5) v[i].x-=force*(e[i]->p.x-xmax+0.5);
      if (e[i]->p.y<ymin+0.5) v[i].y-=force*(e[i]->p.y-ymin-0.5);
      if (e[i]->p.y>ymax-0.5) v[i].y-=force*(e[i]->p.y-ymax+0.5);
//    if (e[i]->p.z<zmin+0.5) v[i].z-=force*(e[i]->p.z-zmin-0.5);
//    if (e[i]->p.z>zmax-0.5) v[i].z-=force*(e[i]->p.z-zmax+0.5);
    }
    for(i=0;i<ct;i++) {e[i]->p+=v[i];v[i]*=damp;}
  }
  for (i=ct-1;i>=0;i--) {
    ax=myint((e[i]->p.x-xmin)*xsp);
    ay=myint((e[i]->p.y-ymin)*ysp);
    az=myint((e[i]->p.z-zmin)*zsp);
    j=ax+nx*(ay+ny*az);
    if (j!=b[i]) {
      if (ax>=0&&ax<nx&&ay>=0&&ay<ny&&az>=0&&az<nz)
        reg[j].data[reg[j].num++]=*e[i];
      *e[i]=reg[b[i]].data[--reg[b[i]].num];
    }
  }
}

// More efficient relaxion scheme
#define rgridc rgrid*rgrid*rgrid
void container::relax(vec &p,float r,float s,float force,float damp,int steps) {
  static particlev b[rgridc][rbuf];
  static int c[rgridc],e[rgridc],cc[rgridc];
  int ax,ay,az,bx,by,bz,i,j,k,l,m;vec d;intrec a;float t;
```

```
for (i=0;i<rgridc;i++) {c[i]=0;e[i]=rbuf;}
a=boundbox(p,s);
for (az=a.lz;az<=a.uz;az++) {
  for (ay=a.ly;ay<=a.uy;ay++) {
    for (ax=a.lx;ax<=a.ux;ax++) {
      i=ax+nx*(ay+ny*az);j=0;
      while(j<reg[i].num) {
        d=reg[i].data[j].p-p;
        t=radsq(d);
        if (t<s*s) {
          bx=int((d.x+s)/(2*s)*rgrid);
          by=int((d.y+s)/(2*s)*rgrid);
          bz=int((d.z+s)/(2*s)*rgrid);
          k=bx+rgrid*(by+rgrid*bz);
          if (t<r*r) {
            b[k][c[k]].data=reg[i].data[j];
            b[k][c[k]].v.x=0;b[k][c[k]].v.y=0;b[k][c[k]++].v.z=0;
            reg[i].data[j]=reg[i].data[--reg[i].num];
          }
          else b[k][--e[k]].data.p=reg[i].data[j++].p;
        }
        else j++;
      }
    }
  }
}
i=0;
while(i<steps) {
  for(j=0;j<rgridc;j++) {
    for(k=0;k<c[j];k++) {
      d=b[j][k].data.p-p;
      a.lx=int((d.x-1+s)/(2*s)*rgrid);if (a.lx<0) a.lx=0;if (a.lx>=rgrid) a.lx=rgrid-1;
      a.ly=int((d.y-1+s)/(2*s)*rgrid);if (a.ly<0) a.ly=0;if (a.ly>=rgrid) a.ly=rgrid-1;
      a.lz=int((d.z-1+s)/(2*s)*rgrid);if (a.lz<0) a.lz=0;if (a.lz>=rgrid) a.lz=rgrid-1;
      a.ux=int((d.x+1+s)/(2*s)*rgrid);if (a.ux<0) a.ux=0;if (a.ux>=rgrid) a.ux=rgrid-1;
      a.uy=int((d.y+1+s)/(2*s)*rgrid);if (a.uy<0) a.uy=0;if (a.uy>=rgrid) a.uy=rgrid-1;
      a.uz=int((d.z+1+s)/(2*s)*rgrid);if (a.uz<0) a.uz=0;if (a.uz>=rgrid) a.uz=rgrid-1;
      for(az=a.lz;az<=a.uz;az++) {
        for(ay=a.ly;ay<=a.uy;ay++) {
          for(ax=a.lx;ax<=a.ux;ax++) {
            l=ax+rgrid*(ay+rgrid*az);
            for(m=rbuf-1;m>=e[l];m--) {
              d=b[j][k].data.p-b[l][m].data.p;
              t=radsq(d);
              if (t<1) {t=sqrt(t);d*=force*(1-t)/t;b[j][k].v+=d;}
            }
            if(j<l) {
              for(m=0;m<c[l];m++) {
                d=b[j][k].data.p-b[l][m].data.p;
                t=radsq(d);
                if (t<1) {t=sqrt(t);d*=force*(1-t)/t;b[j][k].v+=d;b[l][m].v-=d;}
              }
            }
            else if (j==l) {
              for(m=0;m<k;m++) {
                d=b[j][k].data.p-b[l][m].data.p;
                t=radsq(d);
                if (t<1) {t=sqrt(t);d*=force*(1-t)/t;b[j][k].v+=d;b[l][m].v-=d;}
              }
            }
          }
        }
      }
      if (b[j][k].data.p.x<xmin+0.5) b[j][k].v.x-=force*(b[j][k].data.p.x-xmin-0.5);
      if (b[j][k].data.p.x>xmax-0.5) b[j][k].v.x-=force*(b[j][k].data.p.x-xmax+0.5);
      if (b[j][k].data.p.y<ymin+0.5) b[j][k].v.y-=force*(b[j][k].data.p.y-ymin-0.5);
      if (b[j][k].data.p.y>ymax-0.5) b[j][k].v.y-=force*(b[j][k].data.p.y-ymax+0.5);
    }
```

```
      cc[j]=c[j];
    }
    if(++i<steps) {
      for(j=0;j<rgridc;j++) {
        k=0;
        while(k<cc[j]) {
          b[j][k].data.p+=b[j][k].v;
          d=b[j][k].data.p-p;
          b[j][k].v*=damp;
          bx=int((d.x+s)/(2*s)*rgrid);
          by=int((d.y+s)/(2*s)*rgrid);
          bz=int((d.z+s)/(2*s)*rgrid);
          l=bx+rgrid*(by+rgrid*bz);
          if(l!=j) {
            b[l][c[l]++]=b[j][k];
            b[j][k]=b[j][--c[j]];
            if(c[j]>=cc[j]) k++;else cc[j]--;
          }
          else k++;
        }
      }
    }
  }
  for(j=0;j<rgridc;j++) {
    for(k=0;k<c[j];k++) {
      b[j][k].data.p+=b[j][k].v;
      ax=myint((b[j][k].data.p.x-xmin)*xsp);
      ay=myint((b[j][k].data.p.y-ymin)*ysp);
      az=myint((b[j][k].data.p.z-zmin)*zsp);
      i=ax+nx*(ay+ny*az);
      if (ax>=0&&ax<nx&&ay>=0&&ay<ny&&az>=0&&az<nz)
        reg[i].data[reg[i].num++]=b[j][k].data;
    }
  }
};

// Special int function required to pick out the correct regions.
// Returns int(i) if i>0 and int(i)-1 if i<0 to give consistent stepping from
// positive to negative. Thus myint(-1.5,-0.5,0.5,1.5)=-2,-1,0,1.
inline int container::myint(float co) {
  return int((co<0)?int(co)-1:int(co));
}

// Counts the particles in a spherical region.
int container::count(vec &p,float r) {
  intrec a;
  int ax,ay,az,i,j,ct=0;
  a=boundbox(p,r);
  for (az=a.lz;az<=a.uz;az++) {
    for (ay=a.ly;ay<=a.uy;ay++) {
      for (ax=a.lx;ax<=a.ux;ax++) {
        i=ax+nx*(ay+ny*az);
        for(j=0;j<reg[i].num;j++) {
          if (radsq(reg[i].data[j].p,p)<r*r) ct++;
        }
      }
    }
  }
  return ct;
};

// Finds g(r) for the entire container, for small separations.
void container::gofr(int *b) {
  int i,j,k,l,m,ax,ay,az;vec t;
  float r;intrec a;
  for(i=0;i<blocks;i++) {
    for(j=0;j<reg[i].num;j++) {
      a=boundbox(reg[i].data[j].p,3);
```

```
      for (az=a.lz;az<=a.uz;az++) {
        for (ay=a.ly;ay<=a.uy;ay++) {
          for (ax=a.lx;ax<=a.ux;ax++) {
            k=ax+nx*(ay+ny*az);
            for(l=0;l<reg[k].num;l++) {
              r=radsq(reg[k].data[l].p,reg[i].data[j].p);
              if (r<9/*&&k!=i&&l!=j*/) {m=int(sqrt(r)*500);b[m]++;}
            }
          }
        }
      }
    }
  }
};

// Finds the bond angle distribution for the entire container.
void container::bondangle(int *b,float r) {
  int i,j,k,l,m,n,w,ax,ay,az,bx,by,bz;
  float s,t,rr=r*r;intrec a;vec u,v;
  for(i=0;i<blocks;i++) {
    for(j=0;j<reg[i].num;j++) {
      a=boundbox(reg[i].data[j].p,r);
      for (az=a.lz;az<=a.uz;az++) {
        for (ay=a.ly;ay<=a.uy;ay++) {
          for (ax=a.lx;ax<=a.ux;ax++) {
            k=ax+nx*(ay+ny*az);
            for(l=0;l<reg[k].num;l++) {
              u=reg[k].data[l].p-reg[i].data[j].p;s=radsq(u);
              if (s<rr&&(k!=i||l!=j)) {
                for (bz=a.lz;bz<=a.uz;bz++) {
                  for (by=a.ly;by<=a.uy;by++) {
                    for (bx=a.lx;bx<=a.ux;bx++) {
                      m=bx+nx*(by+ny*bz);
                      for(n=0;n<reg[m].num;n++) {
                        v=reg[m].data[n].p-reg[i].data[j].p;t=radsq(v);
                        if (t<rr&&(m!=k||n!=l)&&(m!=i||n!=j)) {
                          w=int(acos(sp(u,v)/sqrt(s*t))*477.4648292756860073066512 9);
                          b[w]++;
                        }
                      }
                    }
                  }
                }
              }
            }
          }
        }
      }
    }
  }
};


// Computes the distribution of hole sizes for the container.
void container::spaces(float xs,float ys,float zs,
                       float mnz,float mxz,int *b) {
  vec p;float xp,yp,zp,r,s,t;intrec a;int ax,ay,az,j,k;float lar;
  xp=(xmax-xmin)/xs;
  yp=(ymax-ymin)/ys;
  zp=(mxz-mnz)/zs;
  for(p.z=mnz+zp/2;p.z<mxz;p.z+=zp) {
    for(p.y=ymin+yp/2;p.y<ymax;p.y+=yp) {
      for(p.x=xmin+xp/2;p.x<xmax;p.x+=xp) {
        r=(p.x-xmin>1.4975)?1.495:p.x-xmin;if (r>xmax-p.x) r=xmax-p.x;
        if (r>p.y-ymin) r=p.y-ymin;if (r>ymax-p.y) r=ymax-p.y;
        if (r>p.z-zmin) r=p.z-zmin;if (r>zmax-p.z) r=zmax-p.z;
        a=boundbox(p,r);t=4;
```

```
        for (az=a.lz;az<=a.uz;az++) {
          for (ay=a.ly;ay<=a.uy;ay++) {
            for (ax=a.lx;ax<=a.ux;ax++) {
              k=ax+nx*(ay+ny*az);
              for(j=0;j<reg[k].num;j++) {
                s=radsq(reg[k].data[j].p,p);
                if (t>s) t=s;
              }
            }
          }
        }
        t=sqrt(t)-0.5;
        if (t>0) {j=int(((t<r)?t:r)*200);b[j]++;}
        lar=(t<r)?t:r;if (lar>0.40) cout << lar << endl;
// Use this line if you want to isolate boundary effects
//        if (t>0&&t<r) {j=int(t*200);b[j]++;}
      }
    }
  }
};

// Finds all the regions that lie within radius r of p.
inline intrec container::boundbox(vec &p,float r) {
  intrec a;
  a.lx=int((p.x-r-xmin)*xsp);if (a.lx<0) a.lx=0;if (a.lx>=nx) a.lx=nx-1;
  a.ly=int((p.y-r-ymin)*ysp);if (a.ly<0) a.ly=0;if (a.ly>=ny) a.ly=ny-1;
  a.lz=int((p.z-r-zmin)*zsp);if (a.lz<0) a.lz=0;if (a.lz>=nz) a.lz=nz-1;
  a.ux=int((p.x+r-xmin)*xsp);if (a.ux<0) a.ux=0;if (a.ux>=nx) a.ux=nx-1;
  a.uy=int((p.y+r-ymin)*ysp);if (a.uy<0) a.uy=0;if (a.uy>=ny) a.uy=ny-1;
  a.uz=int((p.z+r-zmin)*zsp);if (a.uz<0) a.uz=0;if (a.uz>=nz) a.uz=nz-1;
  return a;
};

// Diagnostic routine to output the number of particles in each region.
void container::regioncount() {
  int ax,ay,az,i,ct=0;
  for (az=0;az<nz;az++) {
    for (ay=0;ay<ny;ay++) {
      for (ax=0;ax<nx;ax++) {
        i=ax+nx*(ay+ny*az);
        cout << "(x,y,z)=(" << ax << "," << ay << "," << az
             << ") : " << reg[i].num << " particles" << endl;
        ct+=reg[i].num;
      }
    }
  }
  cout << ct << " particles in total" << endl;
};

//Diagnostic routine to output the all the particles in a specific
//subregion to standard output.
void container::regionshot(float minx,float maxx,float miny,
                           float maxy,float minz,float maxz) {
  intrec a;int ax,ay,az,i,j;
  a.lx=int((minx-xmin)*xsp);if (a.lx<0) a.lx=0;if (a.lx>=nx) a.lx=nx-1;
  a.ly=int((miny-ymin)*ysp);if (a.ly<0) a.ly=0;if (a.ly>=ny) a.ly=ny-1;
  a.lz=int((minz-zmin)*zsp);if (a.lz<0) a.lz=0;if (a.lz>=nz) a.lz=nz-1;
  a.ux=int((maxx-xmin)*xsp);if (a.ux<0) a.ux=0;if (a.ux>=nx) a.ux=nx-1;
  a.uy=int((maxy-ymin)*ysp);if (a.uy<0) a.uy=0;if (a.uy>=ny) a.uy=ny-1;
  a.uz=int((maxz-zmin)*zsp);if (a.uz<0) a.uz=0;if (a.uz>=nz) a.uz=nz-1;
  for (az=a.lz;az<=a.uz;az++) {
    for (ay=a.ly;ay<=a.uy;ay++) {
      for (ax=a.lx;ax<=a.ux;ax++) {
        i=ax+nx*(ay+ny*az);
        for(j=0;j<reg[i].num;j++) {
          if (reg[i].data[j].p.x>minx&&reg[i].data[j].p.x<maxx&&
              reg[i].data[j].p.y>miny&&reg[i].data[j].p.y<maxy&&
              reg[i].data[j].p.z>minz&&reg[i].data[j].p.z<maxz) {
```

```
            cout << reg[i].data[j].n << " " << reg[i].data[j].p.x << " ";
            cout << reg[i].data[j].p.y << " " << reg[i].data[j].p.z << endl;
          }
        }
      }
    }
  }
};

//Clears the container of all particles.
void container::clear() {
 for(int i=0;i<blocks;i++) reg[i].num=0;
}

#endif
```

C.6.5 relaxtest.cc

```
// Spot elastic relaxation test
//
// Author  : Chris H. Rycroft
// Version : 1.0
// Date    : April 6th 2004
//
// This code uses the spot model routines to initialize a small group of
// particles that is too densely packed, and then apply several elastic
// relaxation steps.

// Standard library includes
#include <cstdio>
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;

// Spot model object routines
#define buf 2048
#include "vec.cc"
#include "container.cc"

int main () {
  float x,y;int i,j;
  container con(-25,25,-4,4,0,150,10,3,30);     // Initialize a container
  char filename[8];vec p(23,0,10);
  for(i=0;i<=20;i++) {
    j=0;
    for(x=24.5;x>20;x-=1) {
      for(y=-3.5;y<4;y+=1) {                    // Make a small regular
        con.put(j++,x,y,10);                    // grid of particles
      }
    }
    con.put(j++,23,-0.1,10);                    // Add three extra particles
    con.put(j++,22.8,0,10);
    con.put(j,22.5,0.1,10);
    con.relax(p,10,10,0.7,0.4,i);               // Apply a relaxation
    sprintf(filename,"relax%d",i);
    con.dump(filename);                         // Dump out positions
    con.clear();
  }
};
```

C.6.6 spot15.cc

```
// Fast spot model code — main routine
//
// Author  : Chris H. Rycroft
// Version : 1.0
// Date    : April 6th 2004
//
// This program uses the spot container routines to make a simple event driven
// spot simulation. Spots are introduced in clusters at the orifice and
// propagated upwards according to a simple continuous time random walk.
// Elastic relaxation is applied after every spot movement.

// Standard library includes
#include <cstdio>
#include <iostream>
#include <fstream>
#include <cmath>
using namespace std;

// Spot model object routines
#define buf 2048
#include "vec.cc"
#include "container.cc"

// Spot model parameters
#define max 16000       //Maximum number of spots
#define clu 1           //Number of spots in a cluster
#define ipr 375.774     //Spot insert rate
#define mpr 27.9540     //Spot move rate
#define rad 2.60266     //Spot radius
#define xst 0.675910    //x step
#define yst 0.675910    //y step
#define zst 0.1         //z step
#define scf 399.341     //Displacement scale factor
#define xbu 1           //Buffer amount of spot center from x walls
#define ybu 1           //Buffer amount of spot center from y walls

// Convenient random number routines
inline double rnd() {
  int r=rand();
  while(r==0) r=rand();
  return (double) rand()/RAND_MAX;
};

int main () {
  double a,t=0;
  vec u(0,0,0),v,s[max],w;
  float xa,xs,ya,ys,sc;
  int i=-1,j,n=0,b,xp,xm,yp,ym,rn;
  char buffer[20];
  fstream file;
  container con(-25,25,-4,4,0,150,10,3,30);     // Initialize container object
  con.import();                                 // Import particles from stdin
  file.open("spot15snap",fstream::out|fstream::trunc);
  while(t<1000) {
    t+=-log(rnd())/(ipr+mpr*n);                 // Work out time to next event
    j=int(t/2);
    if (i<j) con.dumprestart(j*20000,file);
    i=j;
    a=rnd()*(ipr+mpr*n);                        // Decide whether to introduce
    if (a<mpr*n) {                              // or move a spot
      b=int(a/mpr);
      if (s[b].z<rad/3) {
        s[b].z+=zst/2;u.z=-zst/scf;con.spot(s[b],u,rad);
        con.relax(s[b],rad+1,rad+2,0.8,0,1);
        s[b].z+=zst/2;
      } else {
        if (s[b].z<=130) {
          if (s[b].x+xst>25-xbu) xa=(25-xbu-s[b].x)/2;else xa=xst/2;
```

```
        if (s[b].x-xst<xbu-25) xs=(25-xbu+s[b].x)/2;else xs=xst/2;
        if (s[b].y+yst>4-ybu) ya=(4-ybu-s[b].y)/2;else ya=yst/2;
        if (s[b].y-yst<ybu-4) ys=(4-ybu+s[b].y)/2;else ys=yst/2;
        rn=rand()%4;
        if (rn<2) {v.x=(rn<1)?xa:-xs;v.y=0;}
        else {v.x=0;v.y=(rn<3)?ya:-ys;}
        v.z=zst/2;
        w=v/(-scf/2);
        s[b]+=v;                                // Move the spot halfway
        con.spot(s[b],w,rad);                   // Apply its influence
        con.relax(s[b],rad+1,rad+2,0.8,0,1);    // Apply elastic relaxation
        s[b]+=v;                                // Move it the rest of the way
      } else s[b]=s[--n];
    }
  } else {
    b=clu+n;
    if (b<=max) while(n<b)
    {
      s[n].x=rnd()*3-1.5;                       // Introduce up to clu new
      s[n].y=rnd()*3-1.5;                       // spots at the orifice
      s[n++].z=-rad/3;
    }
  }
 }
 file.close();
};
```