University of Mississippi

eGrove

Electronic Theses and Dissertations | Graduate School

2012

Simulation and Phases of Macroscopic Particles in Vortex Flow

Heath Eric Rice

Follow this and additional works at: https://egrove.olemiss.edu/etd

Part of the Physics Commons

Recommended Citation

Rice, Heath Eric, "Simulation and Phases of Macroscopic Particles in Vortex Flow" (2012). *Electronic Theses and Dissertations*. 243.
https://egrove.olemiss.edu/etd/243

This Dissertation is brought to you for free and open access by the Graduate School at eGrove. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of eGrove. For more information, please contact egrove@olemiss.edu.

SIMULATION AND PHASES OF MACROSCOPIC PARTICLES IN VORTEX FLOW

A Thesis
presented in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Physics
The University of Mississippi

by

HEATH ERIC RICE

May 2012

Copyright © 2012 by Heath Eric Rice
All rights reserved.

0.1 Abstract

Granular materials are an interesting class of media in that they exhibit many disparate characteristics depending on conditions. The same set of particles may behave like a solid, liquid, gas, something in-between, or something completely unique depending on the conditions. Practically speaking, granular materials are used in many aspects of manufacturing, therefore any new information gleaned about them may help refine these techniques. For example, learning of a possible instability may help avoid it in practical application, saving machinery, money, and even personnel.

To that end, we intend to simulate a granular medium under tornado-like vortex airflow by varying particle parameters and observing the behaviors that arise. The simulation itself was written in Python from the ground up, starting from the basic simulation equations in Pöschel [1]. From there, particle spin, viscous friction, and vertical and tangential airflow were added. The simulations were then run in batches on a local cluster computer, varying the parameters of radius, flow force, density, and friction. Phase plots were created after observing the behaviors of the simulations and the regions and borders were analyzed.

Most of the results were as expected: smaller particles behaved more like a gas, larger particles behaved more like a solid, and most intermediate simulations behaved like a liquid. A small subset formed an interesting crossover region in the center, and under moderate forces began to throw a few particles at a time upward from the center in a fountain-like effect. Most borders between regions appeared to agree with analysis, following a parabolic critical rotational velocity at which the parabolic surface of the material dips to the bottom of the mass of particles. The fountain effects seemed to occur at speeds along and slightly faster than this division.

Contents

0.2 List of Figures

1 INTRODUCTION

The goal of this master's thesis project is to create a particle-dynamics simulation designed to study interesting behavior in a granular medium subject to tangential (vortex) and vertical flows. The simulation and analysis scripts are written in Python completely from scratch and make extensive use of the Numpy package for array operations [2,3]. Simulation visualization is done using VMD (Visual Molecular Dynamics), and phase plots and other final analysis are done using Mathematica.

Outputs such as xyz coordinates and angular velocities are stored in large matrices (50,000 $\times$ 256 $\times$ 3 in the case of position), and the Numpy package makes operating on these matrices possible. Without it, instead of each single matrix operation we would have to loop through each dimension and address each particle individually. This would make the whole process slow to the point of impracticality, with each simulation taking exponentially more time.

The simulation itself always consists of 256 macroscopic hard-sphere interacting particles in a cylindrical tank subject to various interparticle and flow forces. Parameters open to manipulation include particle size, particle density, particle hardness, vortex velocity, collision damping factors, coefficients of friction, and an α scaling fraction used in the vertical flow force. With enough time, it would be interesting to run simulations varying all parameters, but because of the time requirements of running a useful array of simulations we decided to vary the primary factors of particle size and vortex velocity hoping they would have the most visible impact on the system. These seemed to be the types of parameters that would have the most influence on the system. Other factors were set to realistic values where possible, for example density and hardness were set to approximately that of steel, and damping and friction factors were set high enough to avoid total chaos while still allowing for interesting behavior.

The motivation for this project is twofold: theoretical and practical. Theoretically, granular materials are interesting in that they exhibit characteristics of solids, liquids, and gases (sometimes simultaneously), as well as their own unique behaviors. Sometimes they do exactly what classical mechanics expects them to do, and sometimes they do something completely different. In this simulation, most configurations behave somewhat like liquids, but each end of the size scale shows elements of a gas or solid. Also, unique fountain-like effects were discovered for specific combinations of parameters.

Practically, granular materials are everywhere from sand to cereal. More specifically, granular materials are exposed to all kinds of forces and fields in industrial applications. Cast and forged metal parts are polished in vibrating tanks filled with abrasive ceramic stones, and construction crews excavate and haul various soils around the globe nonstop.

Edible examples include grain particles draining from silos, and candies like jelly beans spinning in their coating drums. This simulation does not pretend to have anywhere near the nuance and complexity of the physical world, but even limited as it may be, any bit of understanding that can be added to such a ubiquitous class of materials helps form a base of knowledge.

2 SIMULATION

2.1 Basic Setup

We wanted to model macroscopic particles interacting in an enclosed tank while in the spinning wind of a vortex. A cylindrical tank was created and collisions with it are governed by the same interaction routines as between the particles themselves. 256 particles were added to the tank. This number was chosen because it was enough to still give a usefully-sized total volume of particles at small particle radii without overfilling the tank for larger particle sizes. In future versions of the simulation it may be useful to vary the number of particles with size to maintain a constant total particle volume inside the tank.

We added a vertical wind force to lift particles upward, and a tangential vortex to rotate them around the tank much like a small tornado. The upward component of the flow is scaled in such a way that particles rest on a cushion of air rather than the tank bottom, and the tank has no top. In the rendered output video and stills, the bottom half of the tank can be seen as a green grid, though the grid itself is not in the simulation trajectory files.

Mathematically the flow field inside the tank at position $\vec{x} = (x_r, x_\theta, x_z)$ is

$$\vec{v}_f = \vec{v}_{f,\perp} + \vec{v}_{f,z} = \vec{x}_r \times \vec{\Omega} + |\vec{v}_{f,z}|e^{-\beta x_z} \tag{1}$$

Here, $\vec{\Omega}$ is the angular velocity of the flow field, and one of the two primary parameters we will be varying in our simulation runs. The base magnitude of the vertical wind speed, $|\vec{v}_{f,z}|$, does not vary with the radial coordinate of the tank, and is constant across all simulations. The exponential factor assures that there is a vertical equilibrium position. Physically this would be justified by saying that the momentum loss in the fluid occurs due to viscosity effects, and where the wind field is concerned, the tank is modeled as somewhat wider than it is tall. This and the fact that the tank is open at the top results in a lowering the wind velocity with z. All of this is further explained in later sections.

The bulk of the simulations analyzed here have parameters chosen to give maximum stability in the simulations, rather than being scaled to realistic material values. Eventually these were modified to match known materials, but this had its own problems as we will see in a later section. In making the phase plots, particle radii ranged from 1/30th the tank size to 1/10th. Anything smaller has so little mass that the simulations take far too long to reach a steady state, and anything larger has too little resolution (is too chunky) to read any useful surface contours.

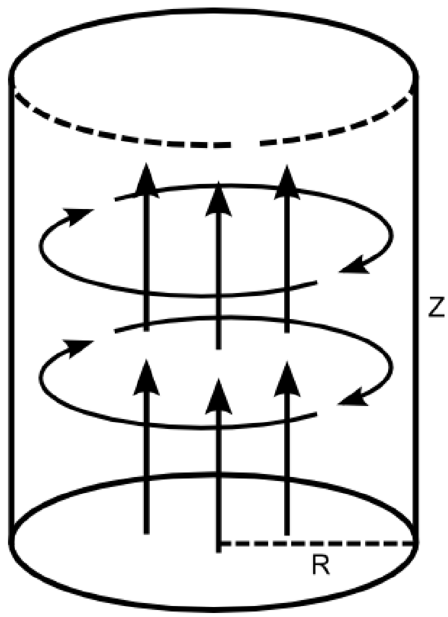


Figure 1: Flow field inside the simulation tank.

To help visualize the behavior of our particle system, we tried to figure out what our chosen material parameters correspond to in the real world. Setting the material viscosity to that of air and then backing-out the parameters that correspond to the forces experienced in simulation, it turns out that our default setup is an approximate match for very hard, very low density balls of about 3 cm in size. This means that we can think of our first couple of simulations as something like ping-pong balls spinning about and colliding in air.

2.2 Interaction Forces

2.2.1 Central and Normal Forces

Unlike aerosols [4], our macroscopic particles do not neglect particle-particle interactions. Rather than integrate equations of motion, this simulation elects to use a common granular computational shortcut to calculate hard-sphere interaction forces between particles, as outlined in *Computational Granular Dynamics* by Pöschel and Schwager [1]. The normal conservative elastic and dissipative forces on particle i normal to the collision point with particle j (or a wall) are given by

$$f_{n,ij}^{(\mathrm{el})} = \frac{2Y\sqrt{r_{\mathrm{eff},ij}}}{3(1-\nu^2)}\xi_{ij}^{3/2} \tag{2}$$

$$f_{n,ij}^{(\mathrm{dis})} = \frac{2Y\sqrt{r_{\mathrm{eff},ij}}}{3(1-\nu^2)} A\sqrt{\xi_{ij}}\frac{d\xi_{ij}}{dt} \tag{3}$$

where Y and ν are properties of the material (Young's modulus and Poisson's ratio, respectively), r_{eff} is an effective radius between the two colliding particles or a particle and a wall, A is a damping coefficient related to the deformation of the particles, and ξ is an overlap factor created by subtracting the distance between two particles from their summed radii:

$$\xi_{ij} = r_i + r_j - |\vec{x}_i - \vec{x}_j| \tag{4}$$

$$A = \frac{1}{3}\frac{(3\eta_2 - \eta_1)^2}{(3\eta_2 + 2\eta_1)}\left[\frac{(1-\nu^2)(1-2\nu)}{Y\nu^2}\right] \tag{5}$$

The parameters $\eta_{1/2}$ are viscous constants of the material and have units of $\mathrm{kg}\cdot\mathrm{m}^{-1}$. Just to confirm the math we will examine the units involved in these force equations.

$$A \rightarrow \left[\frac{(\mathrm{kgm}^{-1} - \mathrm{kgm}^{-1})^2}{(\mathrm{kgm}^{-1} + \mathrm{kgm}^{-1})}\left(\frac{(1-\nu^2)(1-2\nu)}{\mathrm{kgm}^{-1}\mathrm{s}^{-2}}\right)\right] = [\mathrm{s}] \tag{6}$$

$$f_{n,ij}^{(\mathrm{el})} \rightarrow \left[\frac{\mathrm{kg}}{\mathrm{ms}^2}\sqrt{\mathrm{m}}\mathrm{m}^{3/2}\right] = \left[\frac{\mathrm{kgm}}{\mathrm{s}^2}\right] = [\mathrm{N}] \tag{7}$$

$$f_{n,ij}^{(\mathrm{dis})} \rightarrow \left[\frac{\mathrm{kg}}{\mathrm{ms}^2}\sqrt{\mathrm{m}}\mathrm{s}\sqrt{\mathrm{m}}\frac{\mathrm{m}}{\mathrm{s}}\right] = \left[\frac{\mathrm{kgm}}{\mathrm{s}^2}\right] = [\mathrm{N}] \tag{8}$$

For a full derivation of these relations see Brilliantov [5]. Now that we believe that the two components of the normal force are indeed forces, we can write out the complete force equation for normal interactions,

$$f_{n,ij} = \frac{2Y\sqrt{r_{\mathrm{eff},ij}}}{3(1-\nu^2)}\left(\xi_{ij}^{3/2} + A\sqrt{\xi_{ij}}\frac{d\xi_{ij}}{dt}\right) \tag{9}$$

$$\begin{cases} \text{for} & |\vec{x}_i - \vec{x}_j| < r_i + r_j \\ 0 & \text{Otherwise} \end{cases}$$

Right now all of these parameters are the same for all interactions (meaning the walls are made of the same material as the particles). Because we scale the force using this overlap factor, we don't actually need to track velocities at this point. Later in the simulation we will use the various forces to calculate velocities and, in turn, new displacements, but for the immediate purpose of calculating the normal force during a collision we do not need to know the particle's velocity before the collision. If a particle is moving faster and therefore generating more reaction force on impact, it will merely overlap more when we increment the timestep. Even later when we do have velocities calculated, because they are not needed here we do not have to save them for use in the next step. This saves processing time and memory, since there is no need to create and store another $n \times 3$ array. This only works well if the timesteps are short enough to give good resolution during collisions. Tests on a sparsely populated energetic system show collision durations ranging from 13 to 400 timesteps, with most in the twenties and thirties.

The implementation of this interaction is the heart of the simulation. At its core is the main distance matrix. Overlap ξ factors are calculated from an $n \times n$ master distance matrix containing the distances from each particle to each other particle in the case of interactions, and directly calculated from known boundary positions in the case of wall and floor collisions.

```
    # Distance matrix
    dist = numpy.sqrt(pow(xx1 - xx2, 2) + pow(xy1 - xy2, 2) + pow(xz1 - xz2, 2))
    dist = numpy.where(dist == 0, 1e-6, dist)
```

Here a divide-by-zero error is avoided by setting the self-distance terms to 10^{-6} instead of zero,

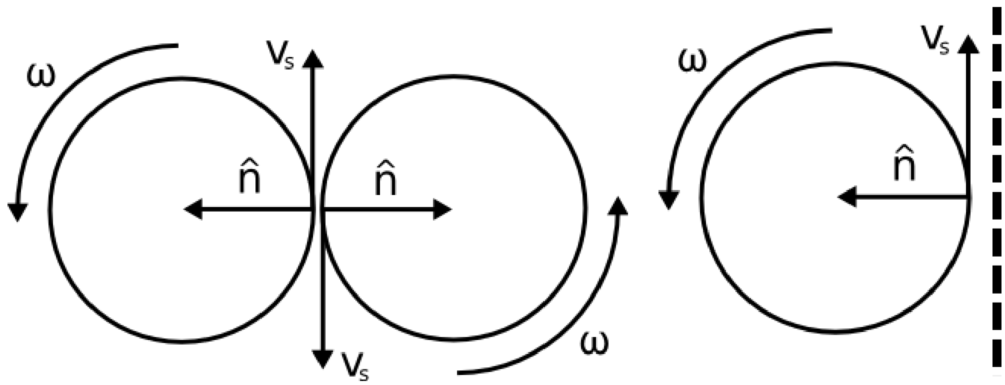
```
    # Overlap factor (>0 means collision)
    xi = xr1 + xr2 - dist
    xi = numpy.where(xi > 0, xi, 0)
    diag_zero = numpy.ones((n,n)) - numpy.diag((1,)*n)
    xi = xi * diag_zero
```

and the diagonal elements of ξ are removed as well to avoid self-collisions.

The main calculation above for $f_{n,ij}$ in Eq. 9 is only the magnitude of the normal force; a normal vector must also be calculated independently. This is similar to the distance calculation in structure, only it keeps components separate and has an output that is either $n \times n \times 3$ containing unit vectors from the center of every particle to the center of every other particle, or $n \times n$ containing unit vectors from every particle to the nearest wall in r and z separately. Now we know the direction in which to aim the normal force.

```
    # Normal vector matrix
    nhat = numpy.array([(xx1 - xx2),(xy1 - xy2),(xz1 - xz2)] / dist)
    nhat_sign = numpy.where(nhat == 0, 0, numpy.sign(nhat))
```

```
    # Normal vector matrix
    normfactor = numpy.sqrt(numpy.sum(pow(x[i], 2), axis=1))
    normfactor = numpy.where(normfactor == 0, 1e-6, normfactor)
    nhat = (-x[i,:, 0], -x[i,:, 1], [0.0] * n) / normfactor
```



(a) Particle-Particle (b) Particle-Wall

Figure 2: Quantities involved in calculating rotational collision forces.

```
    # Normal vectors
    nhat = numpy.zeros ((3, n))
    nhat[2, :] = numpy.ones(n)
```

All that is left now is to calculate the magnitude of the interaction force, which is straightforward.

```
    # Damping factor
    d_xi = (xi - xi_old) / dt
    damp = (pow(xi, 1.5) + A * numpy.sqrt(xi) * d_xi)
    damp = numpy.where(damp < 0.0, 0.0, damp)

    # Effective radius
    reff = 1.0 / (1.0 / xr1 + 1.0 / xr2)

    # Scalar normal force matrix
    fn = (2.0 * Y * numpy.sqrt(2.0 * reff)) / (3.0 * (1.0 -
        pow(nu, 2))) * damp
```

2.2.2 Tangential Forces and Torques

Because the particles are macroscopic, we take spin into account as well. This includes the effect of trajectory at impact on spin, and conversely, the effect of spin at impact on trajectory. Here we work in terms of surface and rotational velocities. A diagram of two types of particle collision and the various quantities involved is given in Fig. 2. The velocity on the surface of a spinning sphere at the point of impact is given by

$$\vec{v}_{s,i} = \vec{\omega}_i \times \vec{r}_i \tag{10}$$

where $\vec{\omega}_i$ is the angular velocity of the particle and $\vec{r}_i$ is a vector from the center of the particle to the point on its surface where contact occurs.

The relative velocity between the two surfaces at that point is

$$\Delta\vec{v}_{s,ij} = \vec{v}_{s,i} - \vec{v}_{s,j} \tag{11}$$

and using a coefficient of friction, μ, the tangential force on the particle is therefore simply

$$\vec{f}_{t,ij} = \frac{\mu m_i \Delta\vec{v}_{s,ij}}{\Delta t} \tag{12}$$

The factors μ and m_i are a coefficient of friction and the mass of particle i respectively.

For the torque we need the difference in rotational velocities, but only in the plane perpendicular to the collision.

$$\Delta\vec{\omega}_{\perp,ij} = \Delta\vec{\omega}_{ij} - \Delta\vec{\omega}_{ij} \cdot \hat{n}_{ij} \tag{13}$$

We then set a simple vector torque based on this maximum change in rotational velocity and scaled by a friction coefficient,

$$\vec{\tau}_i = \frac{\mu I_i \Delta\vec{\omega}_{\perp,ij}}{\Delta t} \tag{14}$$

with I_i being the moment of inertia of particle i.

Computationally, we use another set of grand interaction matrices, this time in $\vec{\omega}$ and $\vec{v}$. These contain the relative rotational and translational velocities between all particles, or between particles and boundaries.

```
    # Delta-w and delta-v matrces
    dw_m = numpy.array([(wx1+wx2),(wy1+wy2),(wz1+wz2)])
    dw_m = (dw_m*collide)
    dv_m = numpy.array([(vx1-vx2),(vy1-vy2),(vz1-vz2)])
    dv_m = (dv_m*collide).T
```

Surface speeds at points of impact are calculated using the previous normal vector matrices and the two matrices in the previous step. The difference in surface speeds is taken for colliding particles only, crossed with the normal vector to get the change in rotational velocity, and then the torque to be applied is calculated using Eq. 14. Boundary collisions are handled the same way, being a simplification where the second particle is considered flat and stationary.

```
    # Surface speed
    dv_s = numpy.cross(dw_m.T, nhat.T) * r_m
    dv_s = (dv_s.T*collide).T
```

The tangential particle forces, Eq. 12 are then only a one-line calculation,

```
    # Tangential force due to spin
    ft1 = numpy.sum(mu * (-dv_s * m_m) / dt, axis=1)
```

All that is left to do now is compute the perpendicular change in $\vec{\omega}$ and compute the torque per Eq. 14,

```
    dw_m2 = dw_m - abs(nhat)*dw_m

```

```
    # Torque from spin-contact
    tau1 = numpy.sum(mu * (-dw_m2.T * I) / dt, axis=1)
```

2.2.3 Total Interaction Forces

Impacting at an angle causes rotation, and rotation can store energy from a collision. Our final force equation takes this into account through Eq. 12.

$$\vec{f}_{ij} = f_{n,ij}\hat{n} + \vec{f}_{t,ij} \tag{15}$$

expands out to

$$\vec{f}_{ij} = \frac{2Y\sqrt{r_{\text{eff},ij}}}{3(1-\nu^2)}\left(\xi_{ij}^{3/2} + A\sqrt{\xi_{ij}}\frac{d\xi_{ij}}{dt}\right)\hat{n} + \frac{\mu m_i \Delta\vec{v}_{s,ij}}{\Delta t} \tag{16}$$

forming the final, complete force equation for our simulations.

The first term in (16) is the force of the collision antiparallel to the collision itself, while the second term is the influence of rotation on the trajectory. Each step that calculates forces and torques due to an interaction adds them to overall force and torque arrays, f and τ, which are applied at the end and reset at the beginning of each timestep.

2.3 Vortex Forces

An object moving through air experiences a drag force proportional to its velocity (relative to the air) that opposes its motion. In our case the drag forces are driving the motion of the system, but that is because the air itself is moving. Typically this drag force is approximated as having linear and quadratic components (Fowles and Landau [6, 7]):

$$f(v) = c_1 v + c_2 v^2 \tag{17}$$

In these simulations, we separate the linear and quadratic portions of the drag force and run them separately, then compare the differences this creates in the phase plots as well as attempt to determine which of the two is of greater importance. Classically for lower velocities the linear term takes precedence, and at higher speeds the quadratic takes over. We also separated horizontal and vertical drag components such that only horizontal velocities contribute to horizontal drag likewise for the vertical. This leaves off the cross term that is technically necessary for complete physical accuracy.

One thing that came to our attention after these simulations were run is that mathematically, a quadratic-only drag force will never reach equilibrium. Solving the differential equation

$$m\frac{\mathrm{d}v}{\mathrm{d}t} = -c_2 v^2 \tag{18}$$

yields position and velocity functions of the form

$$v(t) = \frac{v_0}{1+\frac{t}{\tau}} \tag{19}$$

$$x(t) = v_0 \tau \ln\left(1+\frac{t}{\tau}\right) \tag{20}$$

where $\tau = m/cv_0$. The velocity approaches zero asymptotically, never actually coming to a complete stop. Physically, the linear portion of the drag force is needed to take over in the small v domain and bring the system to equilibrium. This reason, among others, is why we have both terms in the third iteration of the simulation method.

2.3.1 Linear Drag

The primary feature of the simulation is that the particles are in a spinning airflow field. To do this, let us first consider only the flow forces tangential to the tank. An angular flow velocity is set and a force tangential to the tank is calculated based on the difference between the particle velocity and the flow velocity. The flow velocity increases with the radial component of $\vec{x}_i$ to maintain a constant angular velocity, $\vec{\Omega}$. The force of the vortex on a particle is then simply proportional to this velocity difference and the surface area of the particle, as well as the density of the fluid, ρ_f, another coefficient of friction,

$$\vec{v}_{f\perp,i} = \vec{x}_{r,i} \times \vec{\Omega} \tag{21}$$

$$\vec{f}_{f\perp,i} = \mu\pi r_i^2 \rho_f(\vec{v}_{f\perp,i} - \vec{v}_{\perp,i}). \tag{22}$$

Setting an effective wind speed in this way eliminates the need for a separate viscous friction term, as the particles will eventually reach the flow velocity and stop accelerating. At first this drag force scaled linearly with the velocity difference.

2.3.2 Quadratic Drag

Upon further analysis we decided that a dependence on the square of the velocity would be more physically accurate. When researching aerodynamic drag, we found that the quadratic relation was the standard approach, with this form of the drag equation having been attributed to Lord Rayleigh and Sir Isaac Newton himself. As stated before, our drag force is simplified somewhat, but is still based on the classical drag equation. Only for extremely slow speeds and no turbulence is drag linearly proportional to velocity (Stokes' drag). Certain that our velocities were above this threshold, we changed the code and ran another batch of simulations. The actual relation between linear and quadratic is explored further in section 2.5. The new drag force looks like this:

$$\vec{f}_{f\perp,i} = \mu\pi r_i^2 \rho_f(\vec{v}_{f\perp,i} - \vec{v}_{\perp,i})|\vec{v}_{f\perp,i} - \vec{v}_{\perp,i}| \tag{23}$$

It will be shown later in the analysis section that this change in flow force had a quite easily observable impact on the phase boundaries of the system. Both drag models use the same coefficient of friction, so for a given velocity difference the quadratic version will be larger. This should not be a concern because we care about what behaviors the systems exhibit at a steady state. Both of the drag forces should go to zero in the time frame of the simulations we will be considering. The difference should only be visible in how the systems to reach equilibrium and in how well the particles are held to the flow velocity.

2.4 Rectangular Counting Force

Now we will consider the vertical component of the flow field. The goal is to simulate flow through particulate matter, so the particles need to be able to interact with the airflow and shield each other. To accomplish this, we devised a "counting" force,

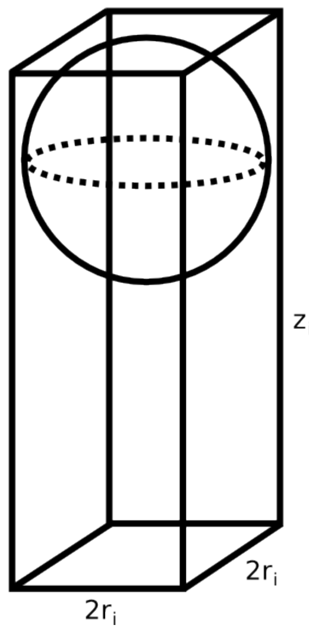


Figure 3: Rectangular counting force diagram.

$$f_{z,i}^{(\text{stack})} = f_{z,i}^{(\text{drag})} \alpha^n e^{-\beta x_{z,i}}. \tag{24}$$

We start with a form of the drag equation that includes both linear and quadratic terms (more on this in section 2.5), then we scale it in what we believe to be a novel manner. Here α is some scaling fraction, and n is the number of particles below particle i in a rectangular prism of shape $2r_i \times 2r_i \times z_i$ as shown in Fig. 3. Particles are counted if any part of their volume is inside the prism, not just the center. This way the vertical force on each particle is reduced by a factor of alpha for every particle below it inside the prism. For example, if α is $\frac{1}{2}$ as it was in the final simulations, a stack of particles will experience forces scaled by a factor of 1, 1/2, 1/4, and 1/8 respectively from bottom to top. This allows a solid block of particles to form with the weight of the upper particles resting on the lower ones, which are held up by air pressure. If a particle doesn't have enough other particles above it to weigh it down or below it to shield it from the vertical wind, it moves upward.

The vertical flow speed is the same for every simulation, and we let the force trail off in z as an exponential, providing the particles an equilibrium point in which to sit. Leaving this exponential off results in either particles sitting on the floor of the cylinder (insufficient vertical force), or particles exiting the top of the cylinder (excessive vertical force). It is worth mentioning that this exponential is only present in the vertical flow forces and the vortex flow does not decrease with height.

```
    # Box footprint of particle
    dx = numpy.abs(xx1 - xx2)
    dy = numpy.abs(xy1 - xy2)
    dz = (xz1 - xz2)

    rect_zx = numpy.where(dx < 2.0*r, 1, 0)
    rect_zy = numpy.where(dy < 2.0*r, 1, 0)

    fzx = numpy.where(dz < 0, rect_zx, 0)
    fzy = numpy.where(dz < 0, rect_zy, 0)

    # Scales by alpha for every particle in dx x dy x dz
    v_flow = 8.0
    fz = (-c1*(v[:,2] - v_flow).T - c2*((v[:,2] - v_flow)*abs(v[:,2] - v_flow)).T)
    fz *= pow(alpha, numpy.sum((fzx * fzy), axis = 0))
```

```
            # Flow forces
            beta2 = 25.0    # 50.0
            ffz = numpy.exp(-beta2 * (x[dt1-1,:,2] + (tank[2]/1.5)))
            f_stack = 100.0 * ffz * flow_rect(dt1-1,x,v,r,alpha,c1,c2,tank)
```

The code for this section just scans the coordinate matrix for particles that meet the location conditions and multiplies by the α factor.

2.5 Realistic Parameter Conversion

The simulation thus far has one limitation: particle parameters were chosen and scaled based on what created nicely-behaved simulations that reached steady-state quickly; these parameters were not chosen to match any realistic values. Late in the cycle of this project the simulation script was completely overhauled to use parameters that matched known materials. Particles were rescaled to the hardness and density of steel, the vortex flow viscosity was lowered to that of actual air, and the tank radius was set at 6 cm.

Mathematically nearly everything was the same, but, for example, instead of a radius of 0.6 and density of 5, particles now had radii of 0.006 meters and densities of 10,000 kg/m^3. This drastically rescaled everything and it took several days to recalibrate into a working system. Even so, the phase plot for our realistic system is far messier than any of the idealized simulations.

The one major difference in the realistic version is the new drag force. Based on the velocity-dependent fluid force from Fowles [6], it combines both the linear and the quadratic terms, and has drag coefficients based on real-world physics,

$$\vec{f}_{f\perp,i} = -c_1(\vec{v}_{\perp,i} - \vec{v}_{f\perp,i}) - c_2(\vec{v}_{\perp,i} - \vec{v}_{f\perp,i})|\vec{v}_{\perp,i} - \vec{v}_{f\perp,i}|. \tag{25}$$

Here,

$$c_1 = 3\pi\eta D \approx 1.55 \times 10^{-4} \frac{\text{kg}}{\text{m} \cdot \text{s}} D \tag{26}$$

$$c_2 = \frac{1}{2} c_d \rho_f A \approx 0.22 \frac{\text{kg}}{\text{m}} D^2 \tag{27}$$

with

η	:	fluid viscosity
D	:	particle diameter
c_d	:	drag coefficient (0.47 for a sphere)
ρ_f	:	fluid density
A	:	cross-sectional area

All approximations are for spheres in air.

Again, in the code only a few lines change.

```
    # Tangential airflow
    v_flow = numpy.zeros(numpy.shape(v))
    v_flow[:,0] = W * x[:, 1]
    v_flow[:,1] = -W * x[:, 0]

    f_flow = (-c1*(v-v_flow).T - c2*((v-v_flow)*abs(v-v_flow)).T)
```

In order to justify which type of drag force is most appropriate for this simulation, we must first examine the velocities involved. For low velocities typically the first order drag force would dominate, and for higher velocities the second order term takes over. We can use a plot of flow and particle velocities for various simulations to figure exactly the ratio of the quadratic to linear drag terms.

Fig. 4 shows the averaged velocity magnitudes for both the particles themselves as well as the spinning wind force at the particle locations. These values mostly represent movement in the horizontal plane, as the simulations settle vertically and nearly all relative motion is in the rotation about the tank. Note that because the flow force is calculated for each particle at each particle location and does not exist independently, the net flow force on the particles is dependent on time and increases as the system is pushed outward. Immediately apparent is that the particles never reach the actual flow velocity, and that some sets of particles continue to increase in velocity through the end of the simulation. Simulations that do not appear to be gaining speed are likely losing vast amounts of energy due to internal friction, which can be much stronger than the flow force at low velocities. Fig. 15(d) is an example of such a case, since it has relatively large particles and a slow rotational velocity, and does not appear to be gaining energy. Given enough time, it likely that these simulations will eventually reach some equilibrium state that is closer to the flow velocity, but the amount of time required is simply impractical.

Using the average particle velocities from these plots we calculated the importance of the quadratic drag term relative to the linear term,

$$\frac{0.22v^2D^2}{1.55\times10^{-4}vD} = 1.4\times10^3 vD. \tag{28}$$

For a range of flow velocities across both the smallest and largest particle sizes we get these values:

r	Ω	Δv	Ratio
2 mm	30 s^{-1}	1.18797 ms^{-1}	0.59398
2 mm	90 s^{-1}	4.10134 ms^{-1}	2.05067
2 mm	150 s^{-1}	5.90051 ms^{-1}	2.95026
6 mm	30 s^{-1}	1.14316 ms^{-1}	1.71474
6 mm	90 s^{-1}	3.31572 ms^{-1}	4.97358
6 mm	150 s^{-1}	5.88083 ms^{-1}	8.82125

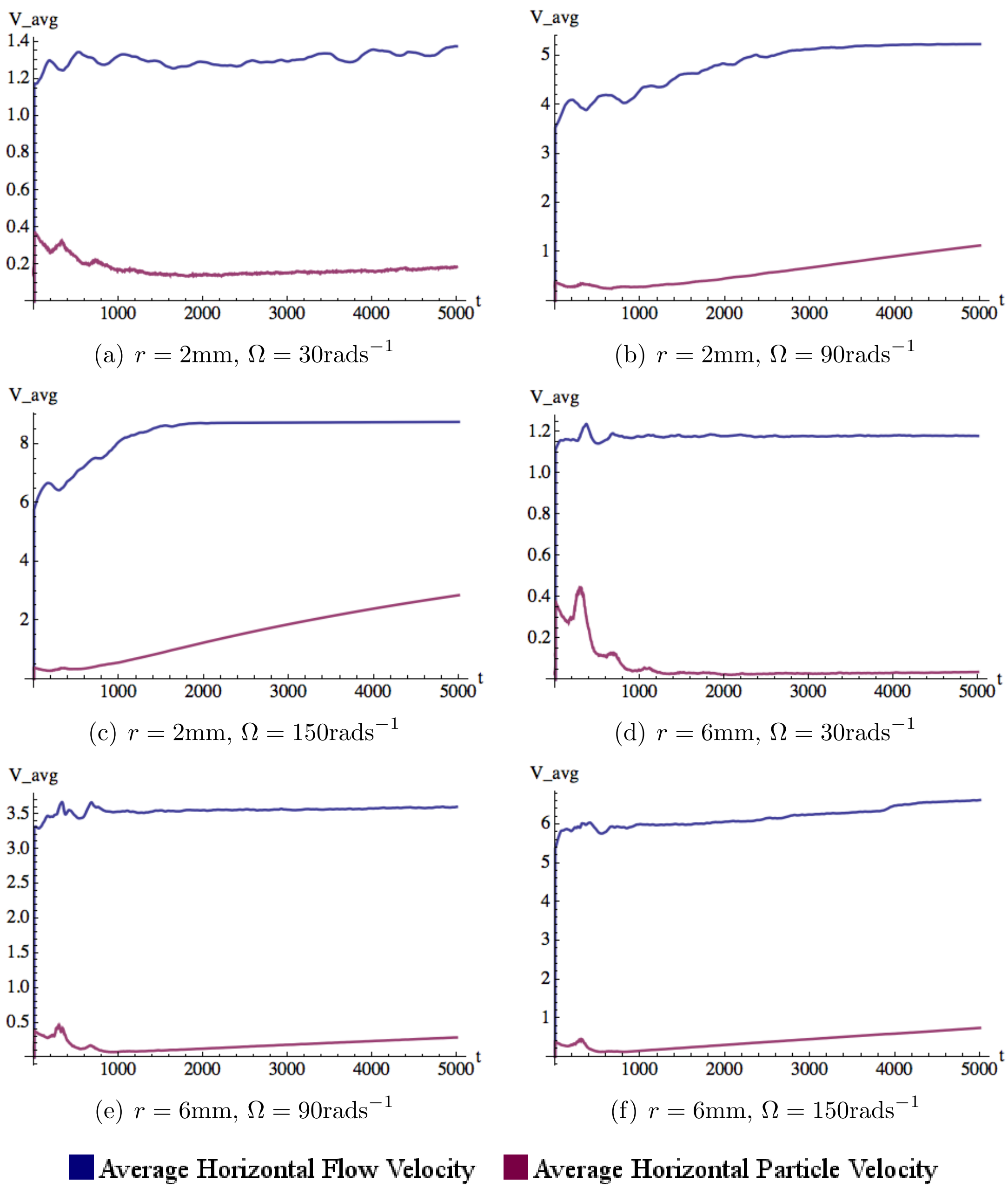
Judging from these values, the quadratic version of the flow force would be the larger of the two for all but the slowest small-particle simulations, but it almost never quite dominates in a way that would let us completely disregard the linear term. This would seem to make the quadratic phase plot the more useful of the two, but in the end what we need is a flow force that has both first and second order dependencies on flow and particle velocities.

Also worth pointing out is that few of the example simulations used in these calculations are at any kind of equilibrium velocity. The particles will continue to increase in speed, but since that would only serve to increase the dominance of the quadratic term in our flow force relation, the use of Fig. 4 as a source of velocity data is still justified. These velocities are only relevant to the previous table and Fig. 14, and have nothing to do with the better-behaved systems in Fig. 13.

Another thing to note is that with steel particles, the vertical flow velocity required to lift the particles up has to be so large that it becomes turbulent, thus making this aspect of the simulation inaccurate.

2.6 Simulation Procedure

The simulation itself can be broken down into three main phases: initialization, timesteps, and finalization.



(a) $r = 2\text{mm}$, $\Omega = 30\text{rads}^{-1}$

(b) $r = 2\text{mm}$, $\Omega = 90\text{rads}^{-1}$

(c) $r = 2\text{mm}$, $\Omega = 150\text{rads}^{-1}$

(d) $r = 6\text{mm}$, $\Omega = 30\text{rads}^{-1}$

(e) $r = 6\text{mm}$, $\Omega = 90\text{rads}^{-1}$

(f) $r = 6\text{mm}$, $\Omega = 150\text{rads}^{-1}$

Figure 4: Sample realistic parameter simulation velocity plots for various values of r and Ω. Velocity magnitudes are averaged and plotted for the flow at particle locations as well as the particles themselves.

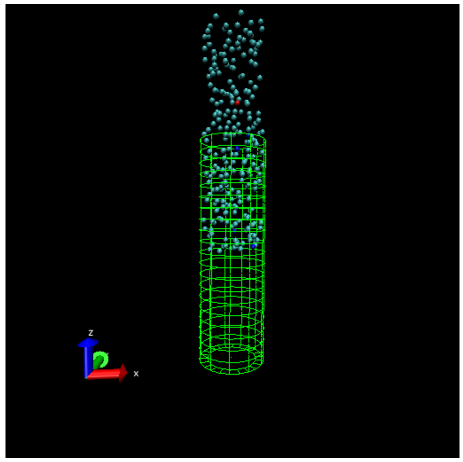


Figure 5: Initial random particle configuration.

2.6.1 Initialization

At the beginning of a simulation, the code first sets all the main physical parameters, then generates a random set of particle positions. The placement algorithm itself makes sure no particles overlap, and is capable of restarting itself should it be unable to find space for all of the required particles. Each particle starts with a small random linear and angular velocity, and the function also generates all the various arrays related to the particles themselves such as moments of inertia and masses,

```
        # Generate a random position
        position = numpy.random.rand(3) * 2.0 * scale - [tank[0], tank[0], tank[2]/2]
        count+=1
        xradial = numpy.sqrt(pow(position[0],2)+pow(position[1],2))

        # Test for tank fit and particle overlap
        if xradial < (tank[0] - r1):
            distance = numpy.sqrt(pow(position[0] - x[0,:,0], 2) +
                pow(position[1] - x[0,:,1], 2) +
                pow(position[2] - x[0,:,2], 2))
            if numpy.all(distance > 2.0 * r1):
                x[0,j,:] = position
```

```
    # Generate other attributes
    for i in xrange (0, n):
        v[i] = numpy.random.rand(3) - 0.5
        w[i] = numpy.random.rand(3) - 0.5
        m[i] = 4/3 * rho * math.pi * pow(r1,3)
```

```
        r[i] = r1
        I_sph = (0.4) * m[i] * pow(r[i], 2)
        I[i] = ([I_sph, I_sph, I_sph])
```

Fig. 5 shows an example of a random starting set of particle coordinates. Originally particles started out in a grid, but this caused various wave and compression effects as the square simulation settled into a round tank that took far too many timesteps to dampen out. Checking for initial particle overlap is crucial. Without it, larger particles would start the simulation partially overlapped, generating huge forces and causing minor explosions that ruin the result.

After the particles are placed and a few other arrays are initialized, the main simulation timesteps begin.

2.6.2 Timestep

The main timestep is where everything happens. At the beginning of each step, the force and torque arrays are zeroed, and then the three main particle interaction functions are run. These calculate the forces and torques generated by the interactions of particle and particle, wall, and floor. These are then added to the main force and torque arrays for the step.

```
    # Normal force due to collision
    fn1 = fn * nhat

    # Tangential force due to spin
    ft1 = mu * dv_s * m_m / dt
```

Now the environmental flow forces are calculated using Eqs. 24 and 22, and are added to the force array along with gravity.

```
            # Flow forces
            beta2 = 25.0    # 50.0
            ffz = numpy.exp(-beta2 * (x[dt1-1,:,2] + (tank[2]/1.5)))
            f_stack = 100.0 * ffz * flow_rect(dt1-1,x,v,r,alpha,c1,c2,tank)
            f_xi = 10.0 * ffz * flow_xi(xi_p)

            f[2, :] += (-ag*m) + f_stack + f_xi
            f += flow_tan_phys(x[dt1-1],v,m,W,c1,c2,tank,dt)
```

Using the now-complete force array, new velocities for each particle are calculated using a basic Euler step,

```
            # Calculate new velocities
            v[:,0] += f.T[:,0] * dt / m
            v[:,1] += f.T[:,1] * dt / m
            v[:,2] += f.T[:,2] * dt / m
```

as are new positions from the velocities. Positions for the current step are stored in the proper i^{th} slice of a giant $i \times n \times 3$ array. Likewise we get a change in rotational velocity for each particle based on the torque array (Fetter [8]).

```
            # Apply movements
            x[dt1] = x[dt1-1] + (v * dt)
            w += tau.T / I * dt
```

Whereas translational motion has the viscous flow function to dampen it, we have to add something to dampen spin here. According to Schiffrik [9], rotational velocity of a spinning sphere in air dampens linearly with $\vec{\omega}$. In the code, we set a small torque proportional to $\vec{\omega}$ that pushes counter to the direction of rotation.

```
            # Viscous rotational friction
            w += -(6e-8 * w.T).T / I * dt
```

The same effect could be accomplished by simply scaling $\vec{\omega}$ back by some small percentage.

For use later in the analysis script, we also store translational, rotational, and potential energies separately at this point (Fetter [8]). They will be later output along with position into the coordinate file.

```
            # Energies
            E_tns = numpy.sum(0.5 * numpy.array((m,m,m)).T * pow(v,2),axis=1)
            E_rot = numpy.sum(0.5 * I * pow(w,2),axis=1)
            E_pot = ag*m * x[dt1,:,2] - group_dot(f.T,x[dt1]-x[dt1-1])

            E[dt1,:,0] = E_tns
            E[dt1,:,1] = E_rot
            E[dt1,:,2] = E_pot
```

Finally, a few safety checks are run. The simulation will continue until it reaches its predetermined timestep limit regardless of malfunction, so it is worthwhile to check a few things at the end of each timestep to keep from wasting time on a ruined simulation.

First all arrays are checked for "NaN" values that are usually the result of attempting to divide by zero, and then the coordinate files are checked to make sure no particles have strayed too far outside the bounds of the tank. This occasionally happens in the event of extreme interparticle pressures or overlap glitches.

```
            # Safety checks
            b_flag = test_nan(f)
            if b_flag == 1:
                break
```

```
            test_x = abs(x[dt1,:,0]) > 2*tank[0]
            test_y = abs(x[dt1,:,1]) > 2*tank[0]
            test_z = abs(x[dt1,:,2]) > 10*tank[2]
            if (test_x.any() or test_y.any() or test_z.any()):
                print('Atom out of bounds at step ' + str(dt1))
                break
```

If all is well, the script returns to the beginning of the timestep section and calculates the next set of data, continuing to loop until the simulation ends.

2.6.3 Finalization

This step is simply the output. Position and energy arrays are combined and then written in binary to a file. This is the end of the simulation.

```
        # Stepped output array
        x_out = numpy.zeros((i_max/step, n, 4))
        x_out[:,:,0:3] = x[::step]
        x_out[:,:,3] = numpy.sum(E[::step],2)

        # Write dump file
        numpy.save(dump_name[0:-4],x_out)
```

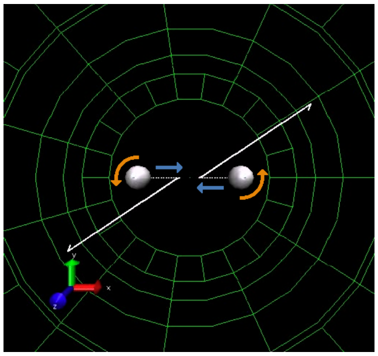
2.7 Test Simulations

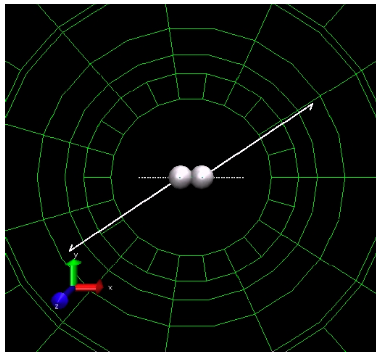
Before major sets of simulations were started on grids of points, a few tests were conducted to make sure various interactions were working properly. The following tests are only of collision dynamics and do not involve any of the flow forces of the final simulation. In fact, half of them don't even involve gravity.

2.7.1 Spin and Particle Collisions

Fig. 6 is a test to see how two spinning particles react to a collision. The two particles are given parallel spins (both pointing upward) and are pushed toward each other through empty space. There is no gravity and the particles are not in contact with any of the surfaces of the tank. As the trajectory tracks clearly show, the two particles collide and spin against each other, pushing themselves off at angles. Another test collision (not shown) with counter-aligned spins has the particles bouncing away along their original path as expected.

Next (Fig. 7), a spinning particle (horizontal axis) is dropped onto a stationary particle resting on the floor of the tank. Without spin the two would simply bounce upward, but here the top particle rolls off in one direction while pushing the stationary particle in the opposite direction.


(a) Initial


(b) Collision

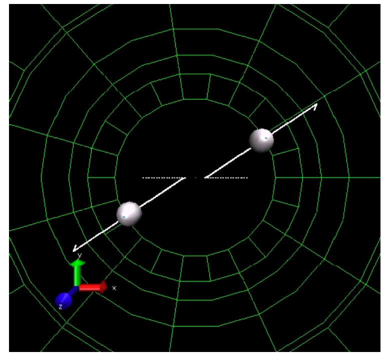
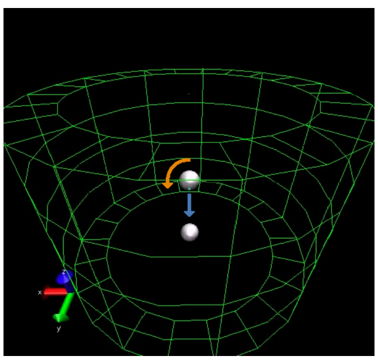
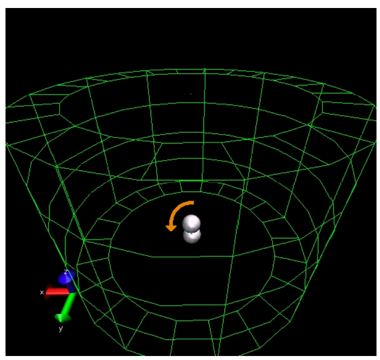
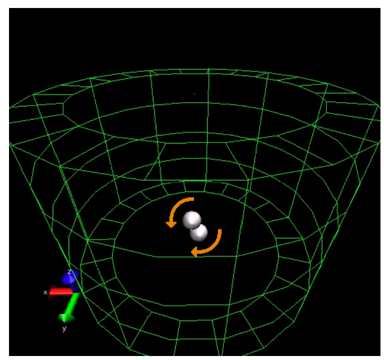
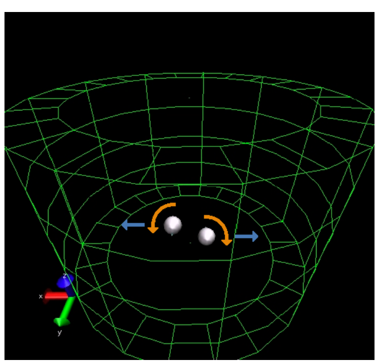

(c) Final

Figure 6: Particles with parallel spins collide and deflect.


(a) Initial fall


(b) Collision


(c) Rolling off


(d) Falling

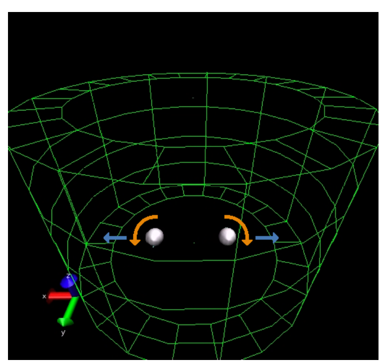
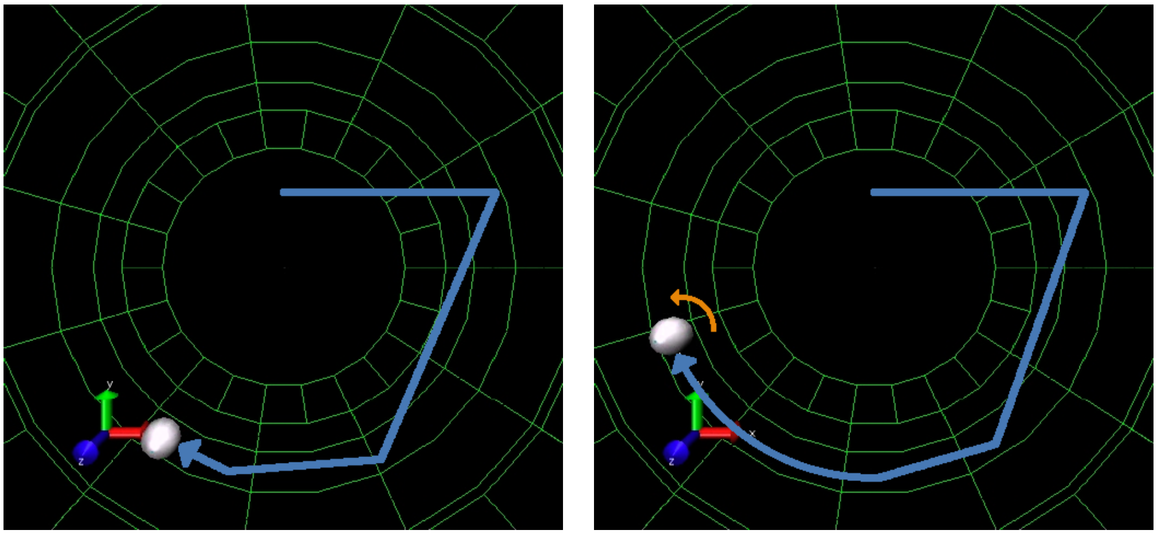

(e) Rolling away

Figure 7: A spinning particle falls onto a stationary one and the two roll apart.



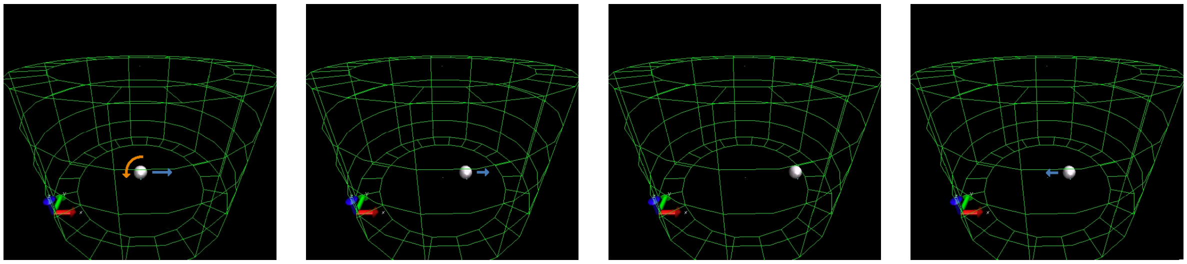
(a) Without spin (b) With spin

Figure 8: Particles collide with a wall at an angle. Only the right particle is allowed to spin.

2.7.2 Spin and Wall Collisions

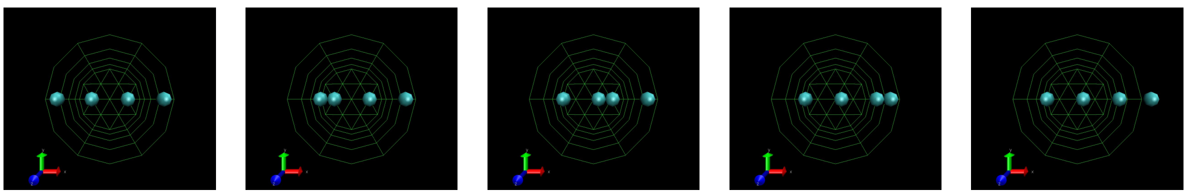
In Fig. 8 we test the influence of spin on the trajectory of a particle as it bounces off the walls of the circular tank. The particles start at the beginning of the track, just upward of the center of the tank traveling rightward, with no initial spin, again through empty space with no gravity or tank contact. The left image is not allowed to spin; it simply bounces off the wall at each collision, forming a series of short, straight segments. The right image is allowed to spin, and with each collision some of the translational energy is transferred into rotational energy. This causes each rebound to be slightly shallower, and eventually leaves the particle rolling along the wall of the tank in a circular path at a constant translational and rotational velocity. This helps to demonstrate the importance of spin on particle trajectory.

To test rolling conditions and friction, in Fig. 9 a particle with horizontal spin is placed on the floor of the tank and given an initial velocity against the direction the particle would roll due to its spin. As it should, the particle moves in the direction of its initial velocity, slowing as it goes until eventually the rolling friction gets the better of it, at which point it begins to roll back toward its initial position.



(a) Initial (b) Begins to slow (c) Velocity reverses (d) Rolls backward

Figure 9: A particle with spin against the direction of motion will slow and roll backwards.



(a) Initial (b) Bounce 1 (c) Bounce 2 (d) Bounce 3 (e) Final

Figure 10: Collision test in the form of a Newton's cradle.

2.7.3 Newton's Cradle

Just for fun, we set up a classic one-dimensional Newton's cradle test in Fig. 10. Four particles are lined up without gravity or tank contact, and the leftmost particle is given an initial rightward velocity. The transfer of momentum appears to be perfect, with no perceptible drift after several cycles.

3 CLASSIFICATION

Simulations were run varying radius and flow to create a grid of simulations for a set range of variables. Upon completion of a grid, a separate analysis script is run which loads the dump files and then makes a judgment as to the final state of the system. Upon observation, the simulations demonstrated several dynamic phases. Larger particles tended to dampen out and, under higher forces, formed the standard paraboloidal surface features of a spinning fluid. This is labeled the fluid phase, Fig. 11(a). Smaller particles under extreme forces were pushed outward into the tank walls, forming a large void in the center of the simulation. This is labeled the pinned phase, Fig. 11(b). Most interestingly, if the particle size and wind velocities were just right for the parabola to be extremely thin in the center, some particles were accelerated upward from the middle of this region, well above the heights of the rest of the particle mass. This is labeled the spout state, Fig. 11(c).

The behavioral analysis script is like a series of sieves. It loads a dump file, and then passes it through a number of classification functions sequentially, each designed to give a yes-or-no output concerning whether or not the simulation matches the criteria sought by the function. A simulation must meet several conditions (each function requiring a different number) for a certain percentage of a certain number of timesteps before giving a positive result. For example, to qualify as a pinned state, all particles in a simulation must be within two radii of the wall for half of 2000 timesteps. This helps avoid false positives should a simulation happen to meet the requirements for a brief period, and gives some leeway for noise.

3.1 Steady State Determination

Every simulation starts with a random arrangement of particles and must settle into some sort of steady state before it can be analyzed. A number of different methods were tried to determine exactly when this state is attained, but ultimately most were flawed. The final solution was perhaps the most obvious one: look for the point where the system reaches a mostly-constant energy.

The simulation script outputs translational, rotational, and potential energies in the dump file with positions (Fetter [8]).

$$E_i^{(\mathrm{tns})} = \frac{1}{2} m_i v_i^2 \tag{29}$$

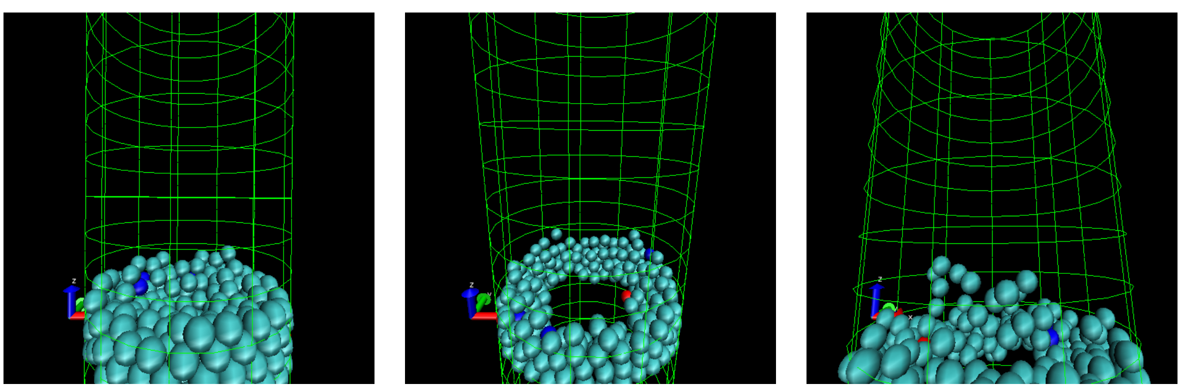
$$E_i^{(\mathrm{rot})} = \frac{1}{2} I_i \omega_i^2 \tag{30}$$

$$E_i^{(\mathrm{pot})} = m_i g x_{z,i} \tag{31}$$

```
# Energies
E_tns = numpy.sum(0.5 * numpy.array((m,m,m)).T * pow(v,2),axis=1)
E_rot = numpy.sum(0.5 * I * pow(w,2),axis=1)
E_pot = ag*m * x[dt1,:,2] - group_dot(f.T,x[dt1]-x[dt1-1])
```

```
E1[0:-1] = E
E2[1:] = E
DE = abs((E1-E2) / (1+E2))

maxes = numpy.where(x[:,:,2]>=0.20,1,0)
maxes = numpy.sum(maxes,axis=1)

for j in xrange (0,i_max-avg_i):
    if numpy.sum(maxes[j:j+avg_i]) < 1:

        # Look for small energy change
        if numpy.average(DE[j:j+avg_i]) <= tolerance:
            settle_yn = 1
```

Fig. 12 shows sample averaged energies for each of the various behaviors we discovered, which will be further explained in the following sections. Since the particles start out very high in the tank, the potential energy ends up being negative, so we rescaled this so the zero of the gravitational potential energy occurred at the average height at which



(a) Fluid (b) Pinned (c) Spout

Figure 11: Sample simulation behaviors.

the particles settled. Since the steady state we're looking for occurs here, it seemed like a logical choice. Ultimately we care more about how the energy stabilizes than where it actually is in the absolute sense.

Here we simply look for a window where the change in energy is below a certain threshold for 500 timesteps. Just to be a little more on the safe side, the analysis script uses the midpoint in the 500-frame window for extra stability. Every subsequent classification function uses this point as the starting timestep for its analysis.

3.2 Fluid State

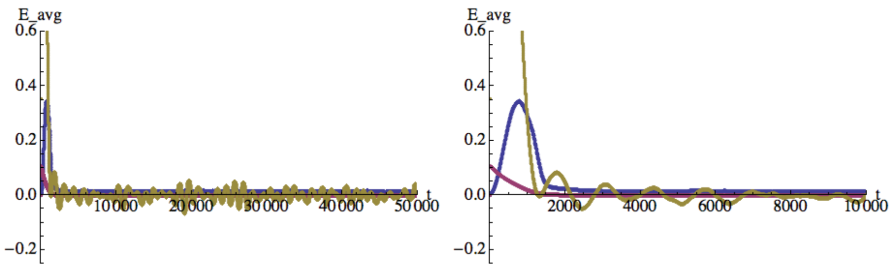
When a liquid spins slowly, the surface forms a paraboloidal shape due to the apparent outward centripetal pseudo-force and downward attraction of gravity. Under the right circumstances the vortex simulation does the same. This can be thought of as the root state on which all of the other states are variations, as will be explained in the next few sections.

Computationally, the fluid state classification is the most intensive of the analysis functions, even after simplification and revision. The difficulty here lies in the fact that we need to plot out the surface of a collection of particles. The current method involves dividing the tank into bins radially, finding the topmost particle in that bin, and then simply checking to see that most of the bins follow a trend of increasing height as the bins approach the outer wall of the tank. (Due to the complex nature of this task, there are quite a few bookkeeping lines in the code that do not really contribute to the understanding of the method. For the full function, see section 7.3 on page 57, lines 93-142.)

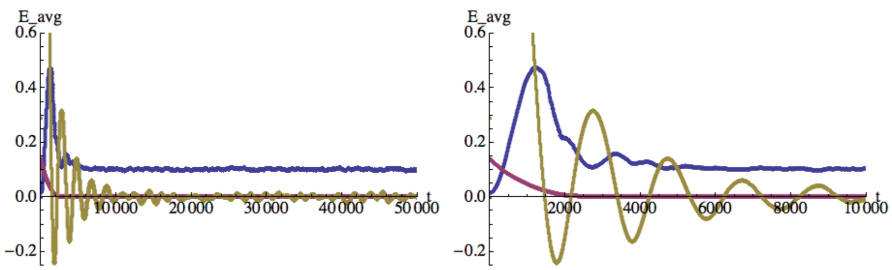
```
104         # Divide simulation into bins
105         bin_tot = int(round(R / (2.0 * r)))
106         bin_width = R / bin_tot
107
```

```
114         for j in xrange (0,bin_tot):
115             bins[j,:] = numpy.where((xr > bin_width*j) & (xr <= bin_width*(j+1)),
116                         x[:,:,2]+100, 0)
117
118         # Find tops of bins
119         z_max = numpy.amax(bins,axis=2)
```

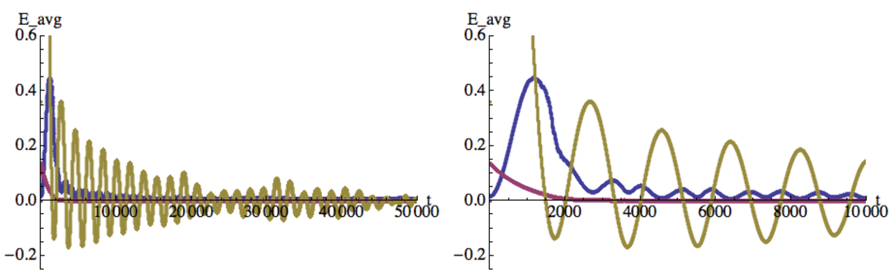
```
125                 # Look for rising bin height from center to wall
126                 if z_max[j,i] - z_max[j-1,i] >= 0.1*r:
127                     check_n[j-1] = 1
```



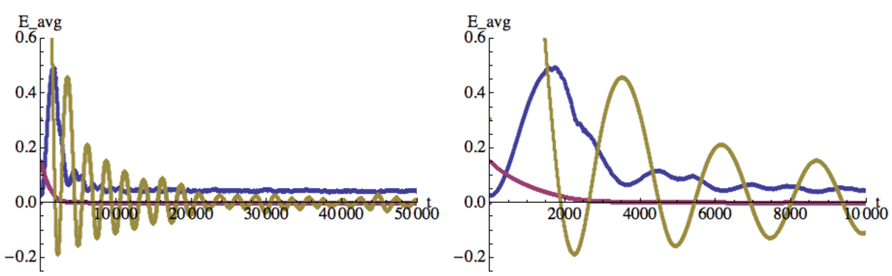
(a) Fluid



(b) Pinned



(c) Flat



(d) Spout

Translational Energy Rotational Energy Potential Energy

Figure 12: Sample simulation energy plots for various behaviors defined in sections 3.2, 3.3, and 3.4. The left column runs over the entire simulation, while the right column is zoomed to the first ten thousand timesteps.

3.3 Pinned State

If the vortex is too strong, the smoothly curving surface gets pushed outward to the point that all of the particles in the system are stacked together in a few layers against the outer wall of the tank. Technically speaking, this is a degenerate form of a parabola where there is just not enough of the substance to cover the bottom of the tank. While this is exactly the sort of behavior we would expect from this kind of system, it is not particularly difficult or interesting.

```
    for j in xrange (0,i_max):
        r_min = numpy.amin(xr[j,:])

        # Look for hole in center of tank
        if R - r_min < R-2.0*r:
            check[j%avg_i] = 1
            if numpy.average(check) >= 0.5:
                pinned_yn = 1
                pinned_i = j + (i_full-i_max) - avg_i
                break
```

3.4 Spout State

And now we come to the most interesting state. Apparently, if the system's rotation is just strong enough, the paraboloid can dip down far enough that only a thin layer of particles exists in the center of the tank. Since our upward counting force (Eq. 24) is a balancing act between the air pressure below and the weight of particles from above, this thin layer is suddenly no longer being held in place. The resulting imbalance actually propels the particles from the center of the tank upward in a sort of particle fountain. They then fall outward into the stacks of particles and a new set slides in to take their place, continuing the convective-like cycle.

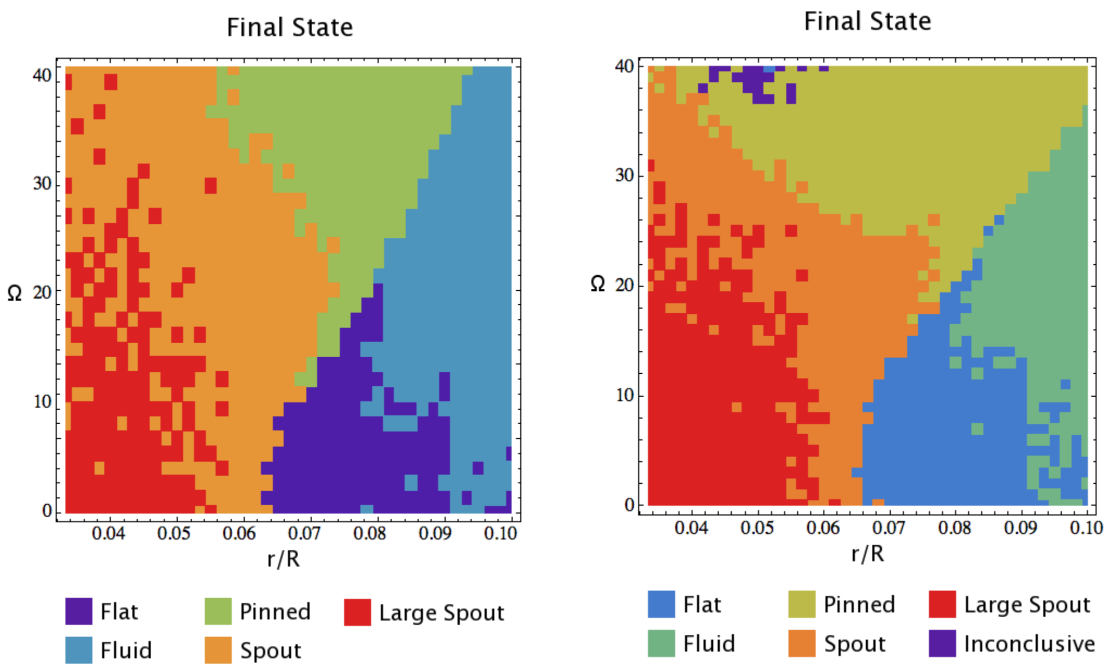
This effect can range from a small fountain to a large percentage of the particles at smaller sizes. Because of this, the phase plots are divided into two spout states, the smaller case and the larger one. (Again, some of the code is left out for brevity, for the full text see section 7.3 on page 57, lines 168-209.)

This state is particularly difficult to automatically classify due to its chaotic nature (it has the most movement of any state), but this approach seems to work well enough. First we create matrices of all particles in the center 3/4 of the tank and of all the particles with upward velocity greater than 100th of a radius per timestep. The scaling of the velocity criteria by radius is necessary to accommodate the difference in speeds particles of different sizes have when in the spout. The analysis script then counts particles belonging to both of these categories and decides between small spout, large spout, and no spout based on that number. If more than five and less than one eighth of the particles qualify, the state is a small spout. Between one eighth and one quarter is a large spout, and anything more is a chaotic state ignored by this portion of the analysis code.

```
        xr = numpy.sqrt(pow(x[:,:,0],2) + pow(x[:,:,1],2))
        vz = x[1::,:,2] - x[0:-1,:,2]
        z_avg = numpy.sum(x[:,:,2], axis=1)/n
        vz_avg = z_avg[1::] - z_avg[0:-1]

        xr_mat = numpy.where(xr <= 3.0*R/4.0, 1, 0)
        vz_mat = numpy.where(vz.T - vz_avg >= r/100.0, 1, 0)

        sum_over_n = numpy.sum(xr_mat[:-1,:] * vz_mat.T, axis=1)

        # Look for 1/8 of particles in the air (small spout)
        binary_over_n1 = numpy.where(((sum_over_n >= avg_n)&(sum_over_n < n/8)), 1, 0)

        # Look for 1.4 of particles in air (large spout)
        binary_over_n2 = numpy.where(((sum_over_n >= avg_n)&(sum_over_n < n/4)), 1, 0)
```

4 ANALYSIS

The final goal of these simulations is to produce phase plots mapping different behaviors to the size and speed variables. The final phase plots presented Fig. 13 and the following sections are created by having the analysis routines assign each type of behavior a value, mapping those values to colors, and plotting the resulting colored regions in the corresponding space of force versus particle size relative to tank size.



(a) Linearly-dependent. (b) Quadratically-dependent.

Figure 13: Primary theoretical phase plots.

Upon visual inspection, the phase borders in Fig. 13 seem to follow fairly well-behaved curves, and in section 4.1 we present a physical rationale that we believe explains one of these borders. Further exploration of these phase borders had to be left for future research again due to time constraints.

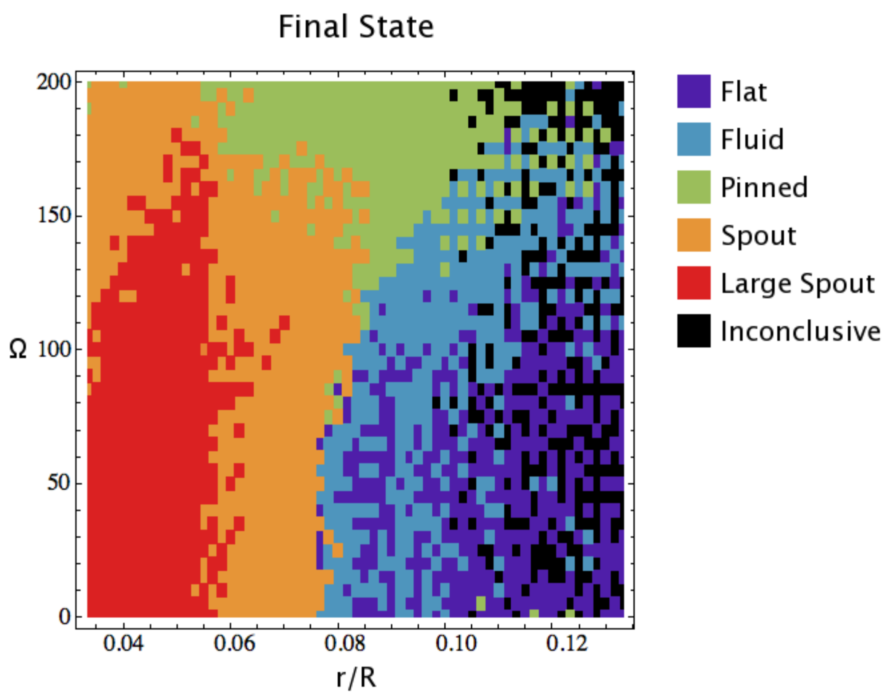


Figure 14: Incomplete realistic parameter drag force phase plot.

4.0.1 Realistic Parameter Conversion

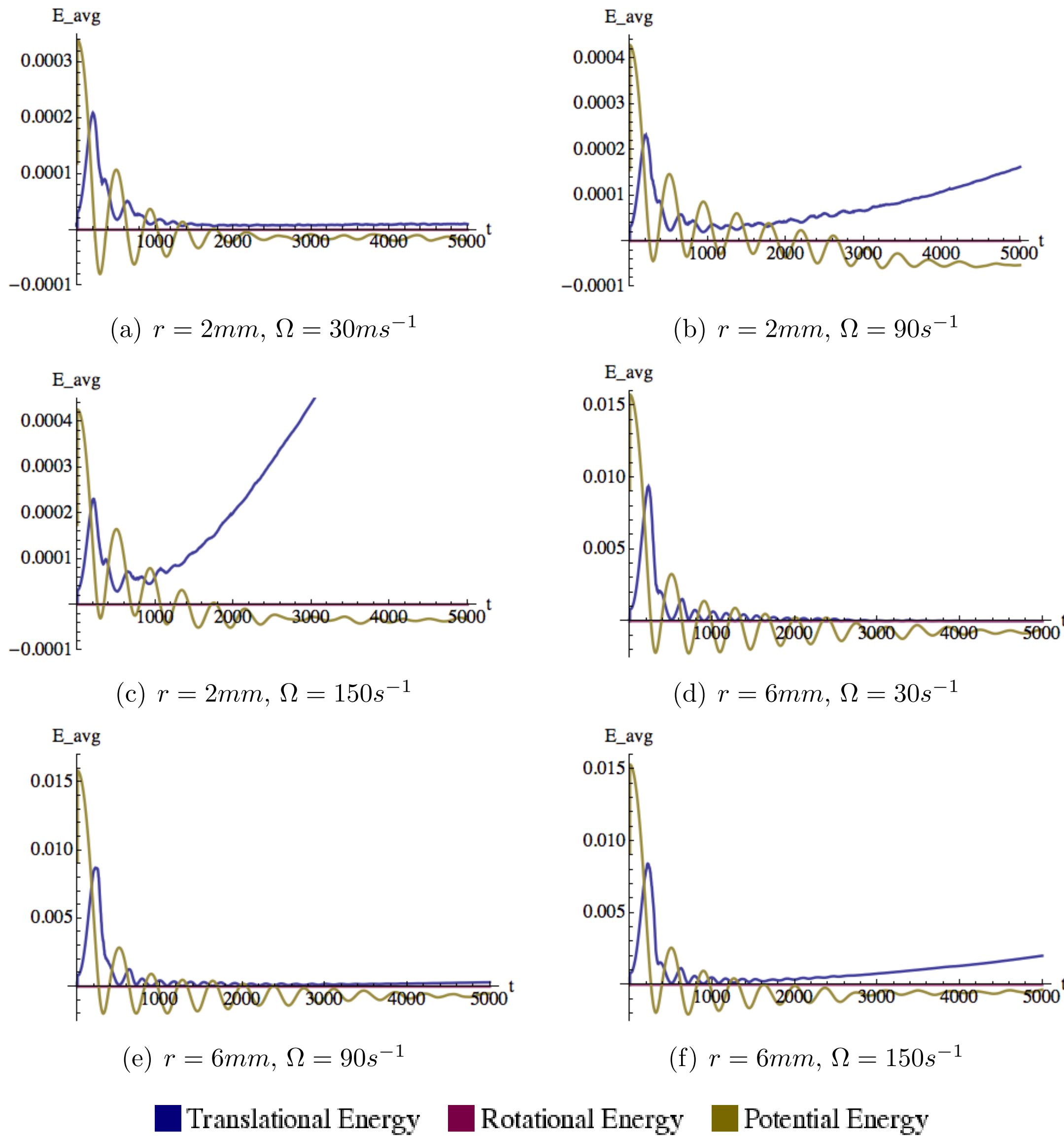
While the third iteration of the simulation is important because of both its relation to properly-scaled parameters and the fact that it was the only version that included both linear and quadratic terms in the drag force, it becomes apparent when comparing the phase plots (Fig. 14) that the realistic system is far less cleanly resolved. Indeed, Fig. 14 can hardly be called a phase plot at all, since far too few of the simulations reached a classifiable state. The shapes are roughly the same, but the borders are fuzzy and most simulations with values of r/R greater than 0.10 are showing a garbled mess of various states and unresolved simulations. Since the basic structure of the simulations remained unchanged, we have to conclude that this is due to the changes in the flow force, and specifically the change to a realistic air medium. Previously the density of the fluid was set to what was apparently a relatively high value. The realistic version now has much smaller cross-sections for air resistance (though relatively the same, of course) and the air itself has a much smaller effect on particle behavior. Where originally particles were able to very quickly reach a terminal velocity that was near that of the fluid, they now accelerate to terminal velocity much more slowly.

For comparison, Fig. 15 shows simulation energies for six example simulations. Only half of these seem to reach a steady state; the other three show exponential gain in kinetic energy to varying degrees. This means that more than likely most of the points in the phase diagram that show as a classifiable state are in fact not at any kind of equilibrium. What is probably happening is that the energy gain is low enough that it falls within the tolerances for what the classification scripts look for as "constant energy," and they are being classified based on their behavior at that time. Given more time, many of the simulation points in the phase plot will probably change type as their energy increases and their behavior shifts as a result.

Another place this difference is visible is in how long the simulations take to settle into a steady state, as shown in Fig. 16. In the linear and quadratic flow models, large particles settle quickly and smaller ones tend to bounce around much longer before reaching equilibrium. In the new simulations due to the slow acceleration, larger particles with their greater mass take longer to settle, continuing to gain speed long after the smaller ones have calmed.

4.1 Fluid Dynamics Approach

For now, let us constrain our analysis to the first two simulation methods because of their much more stable behavior. We can derive the surface geometry of a rotating fluid fairly simply starting from the equation of motion for a small volume of fluid within a larger fluid mass, as given by Marshall and Plumb [10].



(a) $r = 2mm$, $\Omega = 30ms^{-1}$

(b) $r = 2mm$, $\Omega = 90s^{-1}$

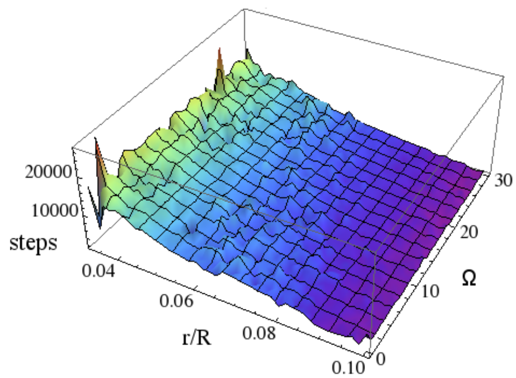
(c) $r = 2mm$, $\Omega = 150s^{-1}$

(d) $r = 6mm$, $\Omega = 30s^{-1}$

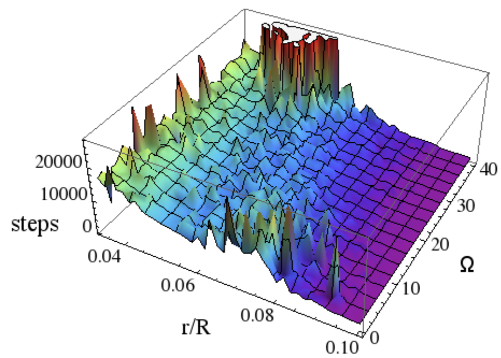
(e) $r = 6mm$, $\Omega = 90s^{-1}$

(f) $r = 6mm$, $\Omega = 150s^{-1}$

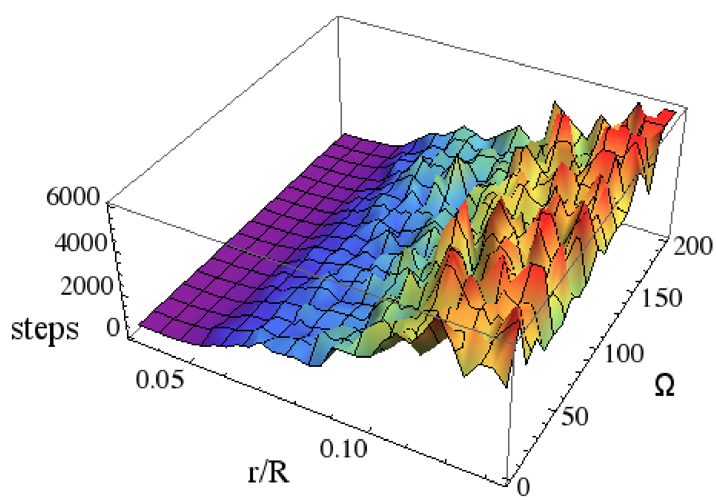
Figure 15: Sample realistic parameter drag force simulation energy plots for various values of r and Ω.



(a) Linearly-dependent settle time



(b) Quadratically-dependent settle time



(c) Realistic parameter settle time

Figure 16: Settle times for three flow models.

$$\frac{D\vec{v}}{Dt} + \frac{1}{\rho}\nabla p + \nabla\phi = F_{\text{ext}} \tag{32}$$

$$\phi_{\text{inertial}} = gx_z \tag{33}$$

Here, $\vec{v}$ is the vector velocity of the fluid volume, ρ is its density, p is the pressure acting on it, ϕ is the potential (such as gravity), and F_{ext} is any external force acting on the fluid. Of course, our fluid simulation is rotating. The centripetal acceleration in a reference frame rotating with the fluid at rotational velocity $\vec{\Omega}$ is

$$\vec{a}_{\text{centripetal}} = -\vec{\Omega} \times \vec{\Omega} \times \vec{x}_r = \nabla\frac{\Omega^2 x_r^2}{2} \tag{34}$$

If the fluid is in solid-body rotation (it appears to be still when viewed from its rotating reference frame) then $\vec{v}$ and F_{ext} are both zero. In the rotating reference frame, the potential is no longer only that due to gravity, but now includes the centripetal force from before. This can be thought of as a sort of effective gravity in the rotating frame.

$$\phi_{\text{rotating}} = gx_z - \frac{\Omega^2 x_r^2}{2} \tag{35}$$

In this case, $\vec{\Omega}$ is the rotational velocity of the mass of particles inside the tank. If the simulation is at equilibrium and the particles have reached terminal velocity, then this $\vec{\Omega}$ is also the rotational speed of the vortex flow itself. We are assuming that this is indeed the case and that the two angular velocities are one and the same.
This leads us to the equation of motion for our rotating fluid in the rotating reference frame

$$\frac{D\vec{v}}{Dt} + \frac{1}{\rho}\nabla p + \nabla\left(gx_z - \frac{\Omega^2 x_r^2}{2}\right) = F_{\text{ext}} \tag{36}$$

$$\frac{1}{\rho}\nabla p + \nabla\left(gx_z - \frac{\Omega^2 x_r^2}{2}\right) = 0 \tag{37}$$

Now consider the surface of our rotating fluid. For Eq. 37 to hold,

$$\frac{p}{\rho} + gx_z - \frac{\Omega^2 x_r^2}{2} = \text{constant} \tag{38}$$

and at the surface there is no pressure, which only leaves

$$gx_z - \frac{\Omega^2 x_r^2}{2} = \text{constant} \tag{39}$$

Now simply solve for x_z to get the shape of the surface.

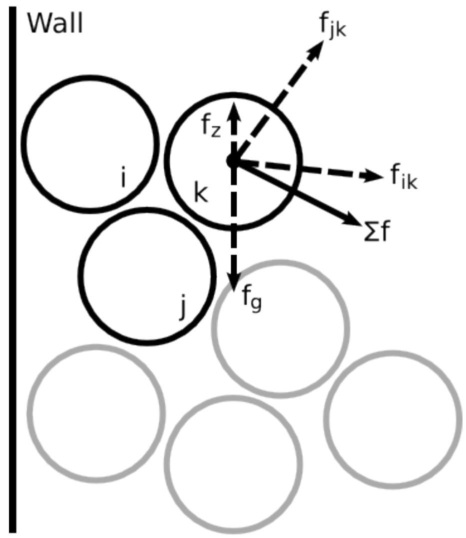


Figure 17: Downward force due to gravity and particle interactions cause the paraboloidal arrangement to relax into a lower energy state.

$$x_z(x_r) = x_z(0) + \frac{\Omega^2 x_r^2}{2g} \tag{40}$$

In this case, $x_z(0)$ is the height of the fluid surface at $x_r = 0$. This is quite clearly a parabola.

The paraboloid structure of a spinning particle simulation tends to relax into a lower energy state due to a combination of the gravitational force and particle interactions. Gravity pulls particles downward, and due to the arrangement, the net force on a particle in contact with other particles in a sloped arrangement tends to be "downhill" toward the center of the simulation tank, as shown in Fig. 17. In this way the downward force due to gravity nets an inward restorative force causing the simulation to flatten out.

4.1.1 Critical Rotation

The most interesting behavior in our simulations tends to occur when the parabolic cross-section of the simulation dips down to the bottom of the collection of particles. We can solve for this condition as a function of rotational velocity and particle size by integrating Eq. 40 to find the total volume in the parabola. We know the simulation has dipped sufficiently in the center when this volume is roughly equal to the entire volume of particles in the simulation.

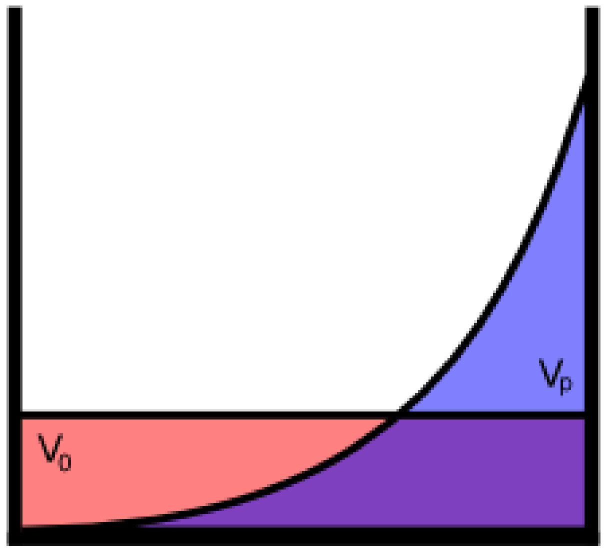


Figure 18: Parabolic cross-section of a simulation.

$$V_p = \int_0^{2\pi} \int_0^R \frac{\Omega^2 x_r^2}{2g} x_r \, \mathrm{d}x_r \mathrm{d}\theta = \frac{\pi \Omega^2 R^4}{4g} \tag{41}$$

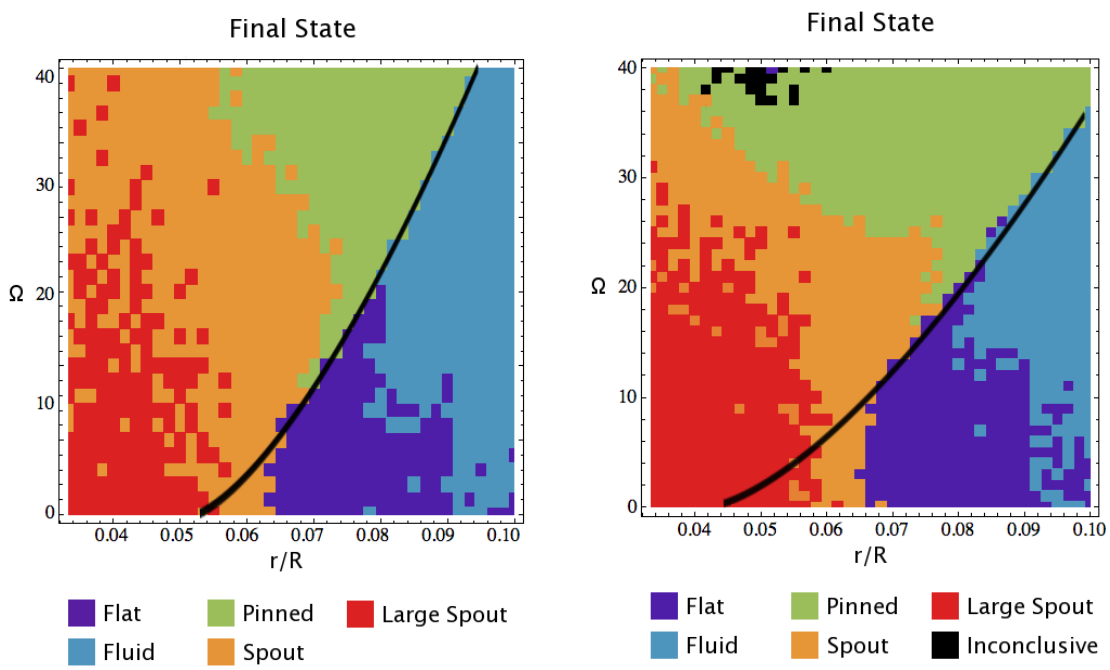
$$V_0 \approx \sum_{i=1}^{n} V_i \frac{4}{3}\pi = \sum_{i=1}^{n} \frac{4}{3}\pi r_i^3 \frac{3\sqrt{2}}{\pi} = n4r_i^3\sqrt{2} \tag{42}$$

Here x_r is the radial coordinate of the simulation, R is the radius of the tank, and V_i and r_i are the volume and radius of particle i. We are also estimating V_0 as the sum of all the individual particle volumes multiplied by a packing fraction for close-packed spheres.

If we set Eqs. 41 and 42 equal to each other and solve for Ω we find the critical rotation speed at which the parabola touches the bottom of the particle collection,

$$\Omega_c = \frac{2}{R^2}\sqrt{\frac{V_0 g}{\pi}} = \frac{4}{R^2}\sqrt{\frac{\sqrt{2}}{\pi} n r_i^3 g} \tag{43}$$

Since this condition is what separates the fluid state from the pinned state, and we believe the spout state to occur below this division, this contour, (Fig. 19), should be the primary feature of our phase plots. Comparing to Fig. 13 we see that this does indeed match except for simulations with low rotational velocities. This can be explained by remembering that we are modeling these simulations as fluids when they are in fact not.



(a) Linearly-dependent. (b) Quadratically-dependent.

Figure 19: Critical rotational velocities as a function of particle radius.

For low rotational velocities there simply is not enough energy in the rotation to break the settled particle arrangement apart. An energy barrier exists due to the "lumpiness" of the medium; static friction holds the particles together and does not allow any parabolic structure to emerge. As we increase the speed the fluid approximation becomes more valid, and the quadratically-dependent version's weaker dependence on radius can be seen in the way its phase plot requires more energy to overcome that initial static clump.

4.1.2 Upward Force

We've shown that there is a critical rotation speed at which the simulation parabola contacts the bottom of the particle mass. We must now find an explanation for where in the phase plots we find the spout state.

The spout state is a vertical phenomenon, and is therefore simply a result of the sum of vertical forces on a particle for a given set of parameters. These vertical forces are as follows.

$$F_z^{(\text{stack})} \propto A_i \Omega^2 \alpha^n e^{-\beta x_{z,i}} = \pi r_i^2 \Omega^2 \alpha^n e^{-\beta x_{z,i}} \tag{44}$$

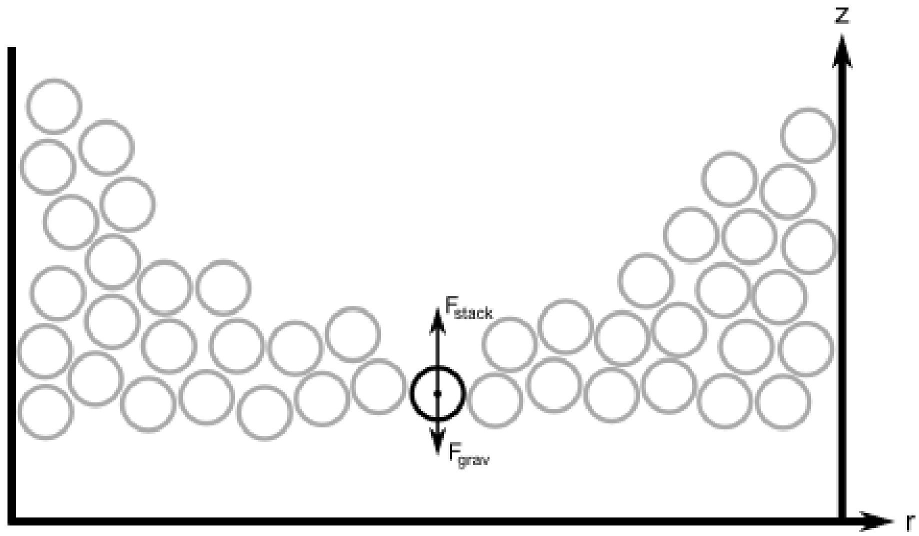


Figure 20: Diagram of the vertical forces on a particle that has just entered the single-layer thin zone in the center of the tank.

$$F_z^{(\mathrm{grav})} \propto m_i g = V_i \rho_i g = \frac{4}{3}\pi r_i^3 \rho_i g \tag{45}$$

If we only consider the net force on the bottom layer of particles (as would be appropriate in the region below the critical rotation speed discussed in the previous section) we see that the total vertical force on a particle that has just entered the single-layer thin zone in the center of the tank as shown in Fig. 20 is

$$F_z^{(\mathrm{net})} \propto \pi r_i^2 \Omega^2 e^{-\beta x_{z,i}} - \frac{4}{3}\pi r_i^3 \rho_i g \tag{46}$$

As shown in Fig. 21, for a given value of Ω the net vertical force rises at first as the term dependent on cross-sectional area (r^2) is greater, then falls as the volume term (r^3) catches up. This means that at a particular rotational velocity, beyond a certain size particles become simply too heavy to be lifted by the stacking force at all. We believe that this and the fact that at larger particle sizes the tank becomes too full to reach Ω_c effectively explain why larger particles exhibit much less vertical motion. This leaves the spout state exclusively in the lower regions of the r/R scale.

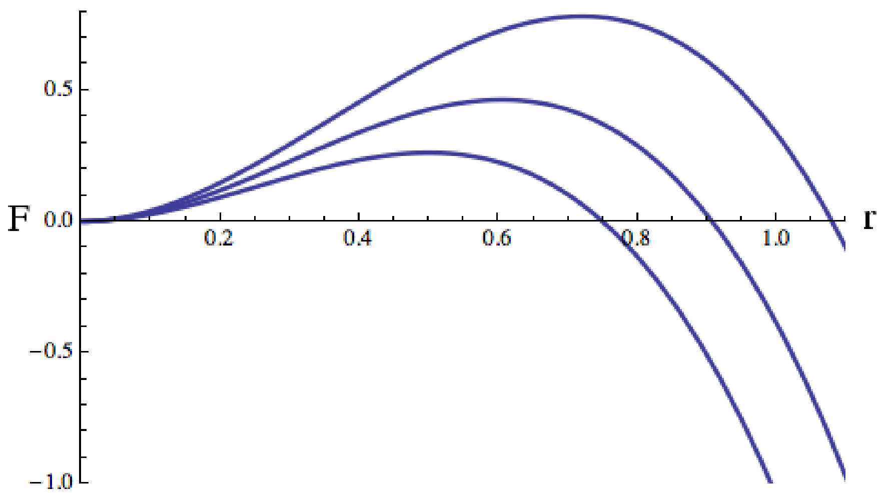


Figure 21: Net vertical force on a particle that has just entered the single-layer thin zone in the center of the tank as a function of radius for three rotational velocities.

5 CONCLUSION

As initially planned, we've built a macroscopic particle simulation with interparticle collisions, particle spin, viscous friction, and vortex airflow and classified its behavior as the parameters of particle radius and flow force were varied. The resulting phase plot shows boundaries that correlate with what our mathematical models predict, and interesting behaviors we did not expect. For low particle size and moderate force, the system enters a spout-like state, propelling a few particles at a time upward from the center. For large particle size the system has very little interparticle motion, with the spheres mostly locking together forming what we called the fluid state. For higher speeds, the particles are flung outward and held against the walls in the pinned state. Lastly in the region where the borders between fluid and spout overlap we have a convective flat state where the spout pushes particles upward just enough to fill in the parabola created by the fluid state, and the particles circulate throughout the tank.

Inspection and comparison to the well-known dynamics of a rotating fluid show state boundaries closely following the critical angular velocity Ω_c at which the fluid parabola dips to contact the bottom of the particle mass. In both cases, the difference between border and curve at low rotational velocities is explained by the frictional energy barrier caused by particle "lumpiness" that is inherent to granular media and not present in actual fluids.

Though the motivation behind this work is purely academic, the crafting of a complex system and study of its interesting behaviors, in terms of useful applications we believe that these simulations shed light on both behaviors that may require avoidance as well as desirable behaviors in the many and varied industries where granular materials are present. For example, if what was explored here holds true, convection corresponding to our flat region can be achieved at low rotational speeds and low vertical forces for particles larger than 0.04 mm or 6% the size of the tank. The high-speed pinning and low-speed parabolas are no surprise, but the fountaining behavior we discovered outside the flat region and below the pinned could be used to enhance convection, or avoided lest it cause problems with machinery. On the other hand, this behavior may find its way into more novel applications, such as next-generation lottery machines, or be used as a curiosity or for purely aesthetic displays, with particle fountains becoming sandy counterparts to today's water fountains.

LIST OF REFERENCES

6 LIST OF REFERENCES

[1] Pöschel, Thorsten, and Thomas Schwager. *Computational Granular Dynamics: Models and Algorithms.*
Berlin: Springer, 2005. Print.

[2] Langtangen, Hans Petter, *Python Scripting for Computational Science.*
Berlin: Springer, 2004. Print.

[3] Kiusalaas, Jaan, *Numerical Methods in Engineering with Python.*
Cambridge: Cambridge University Press, 2005. Print.

[4] Fuchs, N.A., *The Mechanics of Aerosols.*
Oxford: Pergamon Press, 1989. Print.

[5] Brilliantov, Nikolai V., et al., *A model for collisions in granular gasses.*
Phys. Rev. E, Vol. 53, 5382, 1996.

[6] Fowles, Grant R., *Analytical mechanics.*
Saunders College Publishing, 1986. Print.

[7] Landau, L.D., and E.M. Lifshitz, *Fluid Mechanics.*
Oxford: Butterworth-Heinemann, 2007. Print.

[8] Fetter, Alexander L., and John Dirk Walecka, *Theoretical Mechanics of Particles and Continua.*
Mineola: Dover Publications, 2003. Print.

[9] Schiffrik, Nathan D., *Examination of Viscous Torque Relationships.*
Physics Department, The College of Wooster, Wooster, Ohio, 1998.

[10] Marshall, John, and R. Alan Plumb., *Atmosphere, Ocean, And Climate Dynamics, An Introductory Text.*
Academic Press, 2008. Print.

APPENDIX

7 APPENDIX

7.1 Control

This is the control script which supplies parameters to the main simulation.

```
#!/usr/bin/env python

import numpy
import Vortex_0_48_11 as simfile

# Settings
#i = 50000
i = 50000
n = 256

# Density
p = numpy.array((10000.0))

# Radii
#r = numpy.array((0.20, 0.21, 0.22, 0.23, 0.24, 0.25, 0.26, 0.27, 0.28, 0.29,
#                 0.30, 0.31, 0.32, 0.33, 0.34, 0.35, 0.36, 0.37, 0.38, 0.39,
#                 0.40, 0.41, 0.42, 0.43, 0.44, 0.45, 0.46, 0.47, 0.48, 0.49,
#                 0.50, 0.51, 0.52, 0.53, 0.54, 0.55, 0.56, 0.57, 0.58, 0.59,
#                 0.60))

r = numpy.array((0.0020, 0.0030, 0.0040, 0.0050, 0.006,
                 0.0025, 0.0035, 0.0045, 0.0055))

# Flow velocities
#f = numpy.array((00.0, 01.0, 02.0, 03.0, 04.0, 05.0, 06.0, 07.0, 08.0, 09.0,
#                 10.0, 11.0, 12.0, 13.0, 14.0, 15.0, 16.0, 17.0, 18.0, 19.0,
#                 20.0, 21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0, 28.0, 29.0,
#                 30.0, 31.0, 32.0, 33.0, 34.0, 35.0, 36.0, 37.0, 38.0, 39.0,
#                 40.0))

W = numpy.array((000.0, 050.0, 100.0, 150.0, 200.0,
                 025.0, 075.0, 125.0, 175.0))

# Run simulation
simfile.pprun(i,n,r,p,W)
```

7.2 Vortex 1.48.12

This is the main simulation script.

```
#!/usr/bin/env python

## Python Particle Vortex Simulation
import numpy, numpy.random, math, os, sys, time, pp

def cyl_to_cart (array):
    r = array[0]
    theta = array[1]
    z = array[2]
    x = r * math.cos(theta)
    y = r * math.sin(theta)
    z = z
    return numpy.array([x, y, z])

def group_dot (a,b):
    # Take the dot product of 2n vectors in two n x 3 arrays.
    c = numpy.sum(a * b, axis=1)
    return c

def test_nan (f):
    # Tests array for NaN values and reports error
    b_flag = 0
    test = numpy.isnan(f)
    if test.any() == 1:
        print('NaN error')
        b_flag = 1
    return b_flag

def particles (i,n,r1,rho,tank,dt):
    # Generate n random particles
    x = numpy.ones ((i,n,3))
    v = numpy.zeros ((n,3))
    w = numpy.zeros ((n,3))
    I = numpy.zeros ((n,3))
    r = numpy.zeros ((n))
    m = numpy.zeros ((n))
    E = numpy.zeros ((i,n,3))
    V = numpy.zeros ((i,n,3))

    # Scale (0,1) random value generators to tank size
    scale = tank[0], tank[0], tank[2]

    j = 0
    count = 0
```

```
    retry = 0
    while j < n:

        # Generate a random position
        position = numpy.random.rand(3) * 2.0 * scale - [tank[0], tank[0], tank[2]/2]
        count+=1
        xradial = numpy.sqrt(pow(position[0],2)+pow(position[1],2))

        # Test for tank fit and particle overlap
        if xradial < (tank[0] - r1):
            distance = numpy.sqrt(pow(position[0] - x[0,:,0], 2) +
                pow(position[1] - x[0,:,1], 2) +
                pow(position[2] - x[0,:,2], 2))
            if numpy.all(distance > 2.0 * r1):
                x[0,j,:] = position
                j+=1
                count = 0

        # If all particles cannot be placed, start over and try again
        if count > 10000:
            if retry < 5:
                x = numpy.ones ((i,n,3))
                j = 0
                count = 0
                retry +=1
            else:
                print 'Retry limit'
                sys.exit(1)

    # Generate other attributes
    for i in xrange (0, n):
        v[i] = numpy.random.rand(3) - 0.5
        w[i] = numpy.random.rand(3) - 0.5
        m[i] = 4/3 * rho * math.pi * pow(r1,3)
        r[i] = r1
        I_sph = (0.4) * m[i] * pow(r[i], 2)
        I[i] = ([I_sph, I_sph, I_sph])

    return x, v, w, m, r, I, E, V

def manual (i,n,r1,rho):
    # Manual particle placement for testing
    x = numpy.ones ((i,n,3))
    v = numpy.zeros ((n,3))
    w = numpy.zeros ((n,3))
    I = numpy.zeros ((n,3))
    r = numpy.zeros ((n))
    m = numpy.zeros ((n))
```

```
    E = numpy.zeros ((i,n,3))
    V = numpy.zeros ((i,n,3))

    # Test Configurations
    # Direct impact, no spin
    x[0,0,:] = ([-0.02, 0.0, -0.1])
    v[0] = ([1.0, 0.0, 0.0])
    w[0] = ([0.0, 0.0, 0.0])
    r[0] = r1
    m[0] = 4/3 * rho * math.pi * pow(r1,3)
    I_sph = (0.4) * m[0] * pow(r[0], 2)
    I[0] = ([I_sph, I_sph, I_sph])

    x[0,1,:] = ([0.02, 0.0, -0.1])
    v[1] = ([-1.0, 0.0, 0.0])
    w[1] = ([0.0, 0.0, 0.0])
    r[1] = r1
    m[1] = 4/3 * rho * math.pi * pow(r1,3)
    I_sph = (0.4) * m[1] * pow(r[1], 2)
    I[1] = ([I_sph, I_sph, I_sph])

    # Direct impact, counter spin
    x[0,2,:] = ([0.0, -0.02, -0.08])
    v[2] = ([0.0, 1.0, 0.0])
    w[2] = ([0.0, 0.0, -1000.0])
    r[2] = r1
    m[2] = 4/3 * rho * math.pi * pow(r1,3)
    I_sph = (0.4) * m[2] * pow(r[2], 2)
    I[2] = ([I_sph, I_sph, I_sph])

    x[0,3,:] = ([0.0, 0.02, -0.08])
    v[3] = ([0.0, -1.0, 0.0])
    w[3] = ([0.0, 0.0, 1000.0])
    r[3] = r1
    m[3] = 4/3 * rho * math.pi * pow(r1,3)
    I_sph = (0.4) * m[3] * pow(r[3], 2)
    I[3] = ([I_sph, I_sph, I_sph])

    # Direct impact, aligned spin
    x[0,4,:] = ([-0.02, -0.02, -0.06])
    v[4] = ([1.0, 1.0, 0.0])
    w[4] = ([0.0, 0.0, 1000.0])
    r[4] = r1
    m[4] = 4/3 * rho * math.pi * pow(r1,3)
    I_sph = (0.4) * m[4] * pow(r[4], 2)
    I[4] = ([I_sph, I_sph, I_sph])

    x[0,5,:] = ([0.02, 0.02, -0.06])
```

```
    v[5] = ([-1.0, -1.0, 0.0])
    w[5] = ([0.0, 0.0, 1000.0])
    r[5] = r1
    m[5] = 4/3 * rho * math.pi * pow(r1,3)
    I_sph = (0.4) * m[5] * pow(r[5], 2)
    I[5] = ([I_sph, I_sph, I_sph])

    # Glancing blow, no spin
    x[0,6,:] = ([-0.02, -0.005, -0.04])
    v[6] = ([1.0, 0.0, 0.0])
    w[6] = ([0.0, 0.0, 0.0])
    r[6] = r1
    m[6] = 4/3 * rho * math.pi * pow(r1,3)
    I_sph = (0.4) * m[6] * pow(r[6], 2)
    I[6] = ([I_sph, I_sph, I_sph])

    x[0,7,:] = ([0.02, 0.005, -0.04])
    v[7] = ([-1.0, 0.0, 0.0])
    w[7] = ([0.0, 0.0, 0.0])
    r[7] = r1
    m[7] = 4/3 * rho * math.pi * pow(r1,3)
    I_sph = (0.4) * m[7] * pow(r[7], 2)
    I[7] = ([I_sph, I_sph, I_sph])

    # Wall impact, no spin
    x[0,8,:] = ([-0.04, 0.0, -0.02])
    v[8] = ([-1.0, 0.0, 0.0])
    w[8] = ([0.0, 0.0, 0.0])
    r[8] = r1
    m[8] = 4/3 * rho * math.pi * pow(r1,3)
    I_sph = (0.4) * m[8] * pow(r[8], 2)
    I[8] = ([I_sph, I_sph, I_sph])

    x[0,9,:] = ([0.04, 0.0, -0.02])
    v[9] = ([1.0, 0.0, 0.0])
    w[9] = ([0.0, 0.0, 0.0])
    r[9] = r1
    m[9] = 4/3 * rho * math.pi * pow(r1,3)
    I_sph = (0.4) * m[9] * pow(r[9], 2)
    I[9] = ([I_sph, I_sph, I_sph])

    # Wall impact, spin
    x[0,10,:] = ([0.0, -0.04, 0.0])
    v[10] = ([0.0, -1.0, 0.0])
    w[10] = ([0.0, 0.0, 1000.0])
    r[10] = r1
    m[10] = 4/3 * rho * math.pi * pow(r1,3)
    I_sph = (0.4) * m[10] * pow(r[10], 2)
```

```
    I[10] = ([I_sph, I_sph, I_sph])

    x[0,11,:] = ([0.0, 0.04, 0.0])
    v[11] = ([0.0, 1.0, 0.0])
    w[11] = ([0.0, 0.0, 1000.0])
    r[11] = r1
    m[11] = 4/3 * rho * math.pi * pow(r1,3)
    I_sph = (0.4) * m[11] * pow(r[11], 2)
    I[11] = ([I_sph, I_sph, I_sph])

    # Wall impact, offset
    x[0,12,:] = ([0.02, 0.04, 0.02])
    v[12] = ([1.0, 0.0, 0.0])
    w[12] = ([0.0, 0.0, 0.0])
    r[12] = r1
    m[12] = 4/3 * rho * math.pi * pow(r1,3)
    I_sph = (0.4) * m[12] * pow(r[12], 2)
    I[12] = ([I_sph, I_sph, I_sph])


    return x, v, w, m, r, I, E, V

def interact_pp (i,n,x,w,v,r,r_m,m_m,I,Y,nu,A,mu,dt,xi_old):
    # Particle-particle collisions

    # Permutation matrices for add/sub
    xx1, xx2 = numpy.ix_(x[i,:,0], x[i,:,0])
    xy1, xy2 = numpy.ix_(x[i,:,1], x[i,:,1])
    xz1, xz2 = numpy.ix_(x[i,:,2], x[i,:,2])
    xr1, xr2 = numpy.ix_(r,r)

    # Distance matrix
    dist = numpy.sqrt(pow(xx1 - xx2, 2) + pow(xy1 - xy2, 2) + pow(xz1 - xz2, 2))
    dist = numpy.where(dist == 0, 1e-6, dist)

    # Overlap factor (>0 means collision)
    xi = xr1 + xr2 - dist
    xi = numpy.where(xi > 0, xi, 0)
    diag_zero = numpy.ones((n,n)) - numpy.diag((1,)*n)
    xi = xi * diag_zero

    # Normal vector matrix
    nhat = numpy.array([(xx1 - xx2),(xy1 - xy2),(xz1 - xz2)] / dist)
    nhat_sign = numpy.where(nhat == 0, 0, numpy.sign(nhat))

    # Damping factor
    d_xi = (xi - xi_old) / dt
    damp = (pow(xi, 1.5) + A * numpy.sqrt(xi) * d_xi)
```

```
    damp = numpy.where(damp < 0.0, 0.0, damp)

    # Effective radius
    reff = 1.0 / (1.0 / xr1 + 1.0 / xr2)

    # Scalar normal force matrix
    fn = (2.0 * Y * numpy.sqrt(2.0 * reff)) / (3.0 * (1.0 -
        pow(nu, 2))) * damp

    # Matrix to remove values for non-interacting particles
    collide = numpy.where(fn == 0, 0, 1)

    # Rotational Component
    wx1, wx2 = numpy.ix_(w[:,0], w[:,0])
    wy1, wy2 = numpy.ix_(w[:,1], w[:,1])
    wz1, wz2 = numpy.ix_(w[:,2], w[:,2])

    vx1, vx2 = numpy.ix_(v[:,0], v[:,0])
    vy1, vy2 = numpy.ix_(v[:,1], v[:,1])
    vz1, vz2 = numpy.ix_(v[:,2], v[:,2])

    # Delta-w and delta-v matrces
    dw_m = numpy.array([(wx1+wx2),(wy1+wy2),(wz1+wz2)])
    dw_m = (dw_m*collide)
    dv_m = numpy.array([(vx1-vx2),(vy1-vy2),(vz1-vz2)])
    dv_m = (dv_m*collide).T
    dw_m2 = dw_m - abs(nhat)*dw_m

    # Surface speed
    dv_s = numpy.cross(dw_m.T, nhat.T) * r_m
    dv_s = (dv_s.T*collide).T

    # Torque from spin-contact
    tau1 = numpy.sum(mu * (-dw_m2.T * I) / dt, axis=1)

    # Normal force due to collision
    fn1 = -numpy.sum(fn * nhat, axis=1)

    # Tangential force due to spin
    ft1 = numpy.sum(mu * (-dv_s * m_m) / dt, axis=1)

    xi_old = xi

    return fn1 + ft1.T, tau1.T, xi_old

def interact_r (i,n,x,w,v,r,r_m,m_m,I,Y,nu,A,mu,dt,xi_old,tank):
    # Particle-wall collisions in r

```

```
    # Distance matrix
    dist = numpy.sqrt(numpy.sum(pow(x[i,:, 0:2], 2), axis=1))

    # Overlap factor (>0 means collision)
    xi = r - (tank[0] - dist)
    xi = numpy.where(xi > 0, xi, 0)

    # Normal vector matrix
    normfactor = numpy.sqrt(numpy.sum(pow(x[i], 2), axis=1))
    normfactor = numpy.where(normfactor == 0, 1e-6, normfactor)
    nhat = (-x[i,:, 0], -x[i,:, 1], [0.0] * n) / normfactor

    # Damping factor
    d_xi = (xi - xi_old) / dt
    damp = (pow(xi, 1.5) + A * numpy.sqrt(xi) * d_xi)
    damp = numpy.where(damp < 0.0, 0.0, damp)

    # Scalar normal force array
    fn = (2.0 * Y * numpy.sqrt(2.0 * r)) / (3.0 * (1.0 -
        pow(nu, 2))) * damp

    # Matrix to remove values for non-interacting particles
    collide = numpy.where(fn == 0, 0, 1)

    # Surface speeds
    dv_s = numpy.cross(w, nhat.T) * r_m
    dv_s = (dv_s.T*collide).T
    dw = (w - abs(nhat.T) * w).T * collide

    # Torque from spin-contact
    tau1 = mu * -dw.T * I / dt

    # Normal force due to collision
    fn1 = fn * nhat

    # Tangential force due to spin
    ft1 = mu * dv_s * m_m / dt

    xi_old = xi

    return fn1 + ft1.T, tau1.T, xi_old

def interact_z (i,n,x,w,v,r,r_m,m_m,I,Y,nu,A,mu,dt,xi_old,tank):
    # Particle-wall collisions in z

    # Overlap factor (>0 means collision)
    xi = r + (-x[i,:,2] - tank[2])
    xi = numpy.where(xi > 0, xi, 0)
```

```
    # Normal vectors
    nhat = numpy.zeros ((3, n))
    nhat[2, :] = numpy.ones(n)

    # Damping factor
    d_xi = (xi - xi_old) / dt
    damp = (pow(xi, 1.5) + A * numpy.sqrt(xi) * d_xi)
    damp = numpy.where(damp < 0.0, 0.0, damp)

    # Scalar normal force array
    fn = (2.0 * Y * numpy.sqrt(2.0 * r)) / (3.0 * (1.0 -
        pow(nu, 2))) * damp

    # Matrix to remove values for non-interacting particles
    collide = numpy.where(fn == 0, 0, 1)

    # Surface speeds
    dv_s = numpy.cross(w, nhat.T) * r_m
    dv_s = (dv_s.T*collide).T
    dw = (w - abs(nhat.T) * w).T * collide

    # Torque from spin-contact
    tau1 = mu * -dw.T * I / dt

    # Normal force due to collision
    fn1 = fn * nhat

    # Tangential force due to spin
    ft1 = mu * dv_s * m_m / dt

    xi_old = xi

    return fn1 + ft1.T, tau1.T, xi_old

def flow_xi (xi):
    # Contact-based flow force
    fz = numpy.sum(xi,axis=0)

    return fz

def flow_rect (i,x,v,r,alpha,c1,c2,tank):
    # Rectangular counting flow force
    xx1, xx2 = numpy.ix_(x[i,:,0], x[i,:,0])
    xy1, xy2 = numpy.ix_(x[i,:,1], x[i,:,1])
    xz1, xz2 = numpy.ix_(x[i,:,2], x[i,:,2])

    # Box footprint of particle
```

```
    dx = numpy.abs(xx1 - xx2)
    dy = numpy.abs(xy1 - xy2)
    dz = (xz1 - xz2)

    rect_zx = numpy.where(dx < 2.0*r, 1, 0)
    rect_zy = numpy.where(dy < 2.0*r, 1, 0)

    fzx = numpy.where(dz < 0, rect_zx, 0)
    fzy = numpy.where(dz < 0, rect_zy, 0)

    # Scales by alpha for every particle in dx x dy x dz
    v_flow = 8.0
    fz = (-c1*(v[:,2] - v_flow).T - c2*((v[:,2] - v_flow)*abs(v[:,2] - v_flow)).T)
    fz *= pow(alpha, numpy.sum((fzx * fzy), axis = 0))

    return fz

def flow_tan_phys (x,v,m,W,c1,c2,tank,dt):
    # Tangential airflow
    v_flow = numpy.zeros(numpy.shape(v))
    v_flow[:,0] = W * x[:, 1]
    v_flow[:,1] = -W * x[:, 0]

    f_flow = (-c1*(v-v_flow).T - c2*((v-v_flow)*abs(v-v_flow)).T)

    return f_flow

def dirscan ():
    # Scans directory for .npy files
    directory = os.getcwd()
    files = os.listdir(directory)
    npys = []

    for i in xrange (0,len(files)):
        if files[i][len(files[i])-4:len(files[i])] == '.npy':
            npys.append(files[i])

    return npys

def runsim (i_max,n,r1,rho,W,dump_name):
    # Main simulation
    npys = dirscan()

    # Avoid overwriting completed simulations
    if dump_name not in npys:
    #if 1:

        # Manual Settings
```

```
        dt = 0.0001                  # Timestep
        Y = 2e5                      # Young's modulus
        nu = 0.3                     # Poisson ratio
        A = 0.001                    # Normal dissipative constant
        mu = 0.01                    # Spin friction coefficient
        ag = 9.8                     # Acceleration due to gravity
        alpha = 0.5                  # Multiplier for flow force
        c1 = 1.55e-4 * 2*r1          # Linear drag constant
        c2 = 0.22 * pow(2*r1,2)      # Quadratic drag constant
        step = 10                    # Output step

        # Cylinder dimensions with (0,0,0) at center
        tank = numpy.array([0.06, 2.0*math.pi, 0.20])

        # Particle Placement
        x, v, w, m, r, I, E, V = particles (i_max,n,r1,rho,tank,dt)
        #x, v, w, m, r, I, E, V = manual (i_max,n,r1,rho)

        # Array Initialization
        f = numpy.zeros ((3,n))
        xi_p = numpy.zeros ((n,n))
        xi_r = numpy.zeros ((n))
        xi_z = numpy.zeros ((n))
        m_m = numpy.array([m,m,m]).T
        r_m = numpy.array([r,r,r]).T

        # Main Loop
        for dt1 in xrange (1, i_max):

            # Zero arrays
            tau = numpy.zeros ((3, n))
            f = numpy.zeros ((3, n))

            # Interactions
            fp, taup, xi_p = interact_pp (dt1-1,n,x,w,v,r,r_m,m_m,I,Y,nu,A,mu,dt,xi_p)
            fr, taur, xi_r = interact_r  (dt1-1,n,x,w,v,r,r_m,m_m,I,Y,nu,A,mu,dt,xi_r,tank)
            fz, tauz, xi_z = interact_z  (dt1-1,n,x,w,v,r,r_m,m_m,I,Y,nu,A,mu,dt,xi_z,tank)

            # Set for this step
            f += fp + fr + fz
            tau += taup + taur + tauz

            # Flow forces
            beta2 = 25.0    # 50.0
            ffz = numpy.exp(-beta2 * (x[dt1-1,:,2] + (tank[2]/1.5)))
            f_stack = 100.0 * ffz * flow_rect(dt1-1,x,v,r,alpha,c1,c2,tank)
            f_xi = 10.0 * ffz * flow_xi(xi_p)

```

```
            f[2, :] += (-ag*m) + f_stack + f_xi
            f += flow_tan_phys(x[dt1-1],v,m,W,c1,c2,tank,dt)

            # Calculate new velocities
            v[:,0] += f.T[:,0] * dt / m
            v[:,1] += f.T[:,1] * dt / m
            v[:,2] += f.T[:,2] * dt / m

            # Apply movements
            x[dt1] = x[dt1-1] + (v * dt)
            w += tau.T / I * dt

            # Viscous rotational friction
            w += -(6e-8 * w.T).T / I * dt

            # Energies
            E_tns = numpy.sum(0.5 * numpy.array((m,m,m)).T * pow(v,2),axis=1)
            E_rot = numpy.sum(0.5 * I * pow(w,2),axis=1)
            E_pot = ag*m * x[dt1,:,2] - group_dot(f.T,x[dt1]-x[dt1-1])

            E[dt1,:,0] = E_tns
            E[dt1,:,1] = E_rot
            E[dt1,:,2] = E_pot

            # Safety checks
            b_flag = test_nan(f)
            if b_flag == 1:
                break

            test_x = abs(x[dt1,:,0]) > 2*tank[0]
            test_y = abs(x[dt1,:,1]) > 2*tank[0]
            test_z = abs(x[dt1,:,2]) > 10*tank[2]
            if (test_x.any() or test_y.any() or test_z.any()):
                print('Atom out of bounds at step ' + str(dt1))
                break

        # Stepped output array
        x_out = numpy.zeros((i_max/step, n, 4))
        x_out[:,:,0:3] = x[::step]
        x_out[:,:,3] = numpy.sum(E[::step],2)

        # Write dump file
        numpy.save(dump_name[0:-4],x_out)

    print dump_name
    return dump_name

def pprun (i,n,r,p,W):
```

```
    # Create job server
    ppservers = ()
    ncpus = 20
    job_server = pp.Server(ncpus, ppservers=ppservers)

    print 'Starting pp with', ncpus, 'workers'
    print '---'
    print 'i   ', i
    print 'n   ', n

    nr = numpy.size(r)
    if nr == 1:
        r = numpy.array((r,0.0))
    np = numpy.size(p)
    if np == 1:
        p = numpy.array((p,0.0))
    nW = numpy.size(W)
    if nW == 1:
        W = numpy.array((W,0.0))

    print 'Detecting', (nr*np*nW), 'jobs'
    print 'Working...'

    # Main parallel job command
    jobs = [(job_server.submit(runsim,
        (i, n, r[j], p[k], W[l], ('dump_r' + str(r[j]).zfill(6) + '_p' +
        str(p[k]).zfill(7) + '_f' + str(W[l]).zfill(5) + '.npy'),),
        (cyl_to_cart, group_dot, test_nan, particles, manual, interact_pp,
        interact_r, interact_z, flow_xi, flow_rect, flow_tan_phys, dirscan),
        ("numpy", "math", "os", "sys", "time",)))
        for j in xrange (0, nr) for k in xrange (0, np) for l in xrange (0, nW)]

    job_server.wait()
    job_server.print_stats()

    # Serial Command Single
    # runsim(i, n, r[0], p[0], W[0], ('dump_r' + str(r[0]).zfill(6) +
    # '_p' + str(p[0]).zfill(7) + '_f' + str(W[0]).zfill(5) + '.npy'))
```

7.3 Analysis 1.09.01

This is the analysis program which loads dump files and classifies behaviors.

```
#!/usr/bin/env python

import numpy, os, math, sys, pp, time

```

```
def dirscan ():
    # Scan directory for .npy files
    directory = os.getcwd()
    files = os.listdir(directory)
    dumps = []

    for i in xrange (0,len(files)):
        if files[i][len(files[i])-4:len(files[i])] == '.npy':
            dumps.append(files[i])

    if dumps == []:
        print 'No dump files found'
        sys.exit(1)
    else:
        print 'Detecting',len(dumps),'dump files'
        dumps = numpy.sort(dumps)

    return dumps

def write_pplot (array, filename):
    # Write the main plot file
    file = open(filename,'w')
    for i in xrange (0, numpy.shape(array)[0]):
        for j in xrange (0, numpy.shape(array)[1]):
            file.write(str(array[i,j]) + ', ')
        file.write('\n')
    file.close()
    return

def get_values (dump_name):
    # Get simuation values from filename
    values = numpy.zeros(3)

    values[0] = dump_name[6:11]
    values[1] = dump_name[13:20]
    values[2] = dump_name[22:27]

    return values

def settled (x,E,r,R):
    # Determine settling point from which to start analysis
    tolerance = 3e-5

    # Hold criteria for avg_i timesteps
    avg_i = 500
    i_max = numpy.shape(x)[0]
    n = numpy.shape(x)[1]
    settle_i = i_max
```

```
    settle_yn = 0
    check = numpy.zeros(avg_i)

    # Check for out of bounds error (all coordinates go to [1,1,1])
    aob = numpy.where(x[-1,:,:] == [1,1,1])
    if numpy.sum(aob) < 2:

        E1 = numpy.zeros((i_max+1))
        E2 = numpy.zeros((i_max+1))

        # Differences in energy across steps
        E1[0:-1] = E
        E2[1:] = E
        DE = abs((E1-E2) / (1+E2))

        maxes = numpy.where(x[:,:,2]>=0.20,1,0)
        maxes = numpy.sum(maxes,axis=1)

        for j in xrange (0,i_max-avg_i):
            if numpy.sum(maxes[j:j+avg_i]) < 1:

                # Look for small energy change
                if numpy.average(DE[j:j+avg_i]) <= tolerance:
                    settle_yn = 1

                    # Start analyzing midway through avg_i timesteps
                    settle_i = j + (avg_i)/2
                    break

    else:
        settle_yn = -1
        settle_i = -i_max
        print 'Out of bounds'

    return settle_yn, settle_i

def fluid (x,r,R,i_full):
    # Look for fluid state
    fluid_yn = 0
    fluid_i = i_full

    i_max = numpy.shape(x)[0]
    n = numpy.shape(x)[1]

    if i_max != 0:

        # Hold criteria for avg_i timesteps
        avg_i = 1500
```

```
        check_i = numpy.zeros(avg_i)
        check = numpy.zeros(avg_i)

        # Divide simulation into bins
        bin_tot = int(round(R / (2.0 * r)))
        bin_width = R / bin_tot

        bins = numpy.zeros((bin_tot, i_max,n))
        bin_n = numpy.zeros((bin_tot, i_max,n))

        xr = numpy.sqrt(pow(x[:,:,0],2)+pow(x[:,:,1],2))

        # Sort particles into bins
        for j in xrange (0,bin_tot):
            bins[j,:] = numpy.where((xr > bin_width*j) & (xr <= bin_width*(j+1)),
                        x[:,:,2]+100, 0)

        # Find tops of bins
        z_max = numpy.amax(bins,axis=2)

        for i in xrange (0,i_max):
            check_n = numpy.zeros(bin_tot-1)
            for j in xrange (1,bin_tot):

                # Look for rising bin height from center to wall
                if z_max[j,i] - z_max[j-1,i] >= 0.1*r:
                    check_n[j-1] = 1

            # Allow one bin to be out of order
            if numpy.average(check_n) >= (bin_tot-2.0)/(bin_tot-1.0):
                check_i[i%avg_i] = 1
            else:
                check_i[i%avg_i] = 0
            if numpy.average(check_i) >= 0.9:
                fluid_i = i + (i_full-i_max) - (avg_i/2.0)
                fluid_yn = 1
                break

    return fluid_yn, fluid_i

def pinned (x,r,R,i_full):
    # Look for pinned state
    # Hold criteria for avg_i timesteps
    avg_i = 2000
    i_max = numpy.shape(x)[0]
    n = numpy.shape(x)[1]
    pinned_i = i_full
    pinned_yn = 0
```

```
    check = numpy.zeros(avg_i)

    xr = numpy.sqrt(pow(x[:,:,0],2)+pow(x[:,:,1],2))

    for j in xrange (0,i_max):
        r_min = numpy.amin(xr[j,:])

        # Look for hole in center of tank
        if R - r_min < R-2.0*r:
            check[j%avg_i] = 1
            if numpy.average(check) >= 0.5:
                pinned_yn = 1
                pinned_i = j + (i_full-i_max) - avg_i
                break

    return pinned_yn, pinned_i

def spout (x,r,R,i_full):
    # Look for spout state
    spout_yn = 0
    spout_i = i_full

    i_max = numpy.shape(x)[0]
    n = numpy.shape(x)[1]

    if i_max != 0:

        # Find avg_n particles in the air at one time
        avg_n = 5
        # Hold criteria for avg_i timesteps
        avg_i = 2000

        xr = numpy.sqrt(pow(x[:,:,0],2) + pow(x[:,:,1],2))
        vz = x[1::,:,2] - x[0:-1,:,2]
        z_avg = numpy.sum(x[:,:,2], axis=1)/n
        vz_avg = z_avg[1::] - z_avg[0:-1]

        xr_mat = numpy.where(xr <= 3.0*R/4.0, 1, 0)
        vz_mat = numpy.where(vz.T - vz_avg >= r/100.0, 1, 0)

        sum_over_n = numpy.sum(xr_mat[:-1,:] * vz_mat.T, axis=1)

        # Look for 1/8 of particles in the air (small spout)
        binary_over_n1 = numpy.where(((sum_over_n >= avg_n)&(sum_over_n < n/8)), 1, 0)

        # Look for 1.4 of particles in air (large spout)
        binary_over_n2 = numpy.where(((sum_over_n >= avg_n)&(sum_over_n < n/4)), 1, 0)

```

```
        for i in xrange (0,i_max-avg_i):
            if numpy.average(binary_over_n1[i:i+avg_i]) >= 1.0:
                spout_i = i + (i_full-i_max)
                spout_yn = 1
                break
            if numpy.average(binary_over_n2[i:i+avg_i]) >= 1.0:
                spout_i = i + (i_full-i_max)
                spout_yn = 2
                break

    return spout_yn, spout_i

def bounce (x):
    # Find periods of bouncing simulations
    z_avg = numpy.average(x[:,:,2],axis=1)

    # Only look at the last 1000 steps
    if len(z_avg) > 1000:
        z_avg = z_avg - numpy.average(z_avg[-1000:-1])

    # Look for sign change in height about average
    sign_change = []
    for i in xrange (1,len(z_avg)):
        if numpy.sign(z_avg[i-1]) != numpy.sign(z_avg[i]):
            sign_change = numpy.append(sign_change,i)

    changes = len(sign_change)
    if changes >= 4:
        if changes%2 != 0:
            sign_change = sign_change[0:-1]

        sign_change.shape = (changes/2,2)
        periods = numpy.zeros((changes/2-1,2))

        for i in xrange (0,changes/2-1):
            periods[i] = sign_change[i+1] - sign_change[i]

        avg_period = numpy.average(periods)

    else:
        avg_period = -1

    return avg_period

def analyze (dump):
    # Main Analysis
    tank = numpy.array([0.06, 2.0*math.pi, 0.2])
    values = get_values(dump)
```

```
    x = numpy.load(dump)
    i_full,n,d = numpy.shape(x)

    # Separate out energy and position values
    E = x[:,:,3]
    x = x[:,:,0:3]

    E_avg = numpy.average(E, axis=1)

    # Apply filters
    settle_yn, settle_i = settled(x,E_avg,values[0],tank[0])
    if settle_yn > 0:
        fluid_yn, fluid_i = fluid(x[settle_i:,:,:],values[0],tank[0],i_full)
        pinned_yn, pinned_i = pinned(x[settle_i:,:,:],values[0],tank[0],i_full)
        spout_yn, spout_i = spout(x[settle_i:,:,:],values[0],tank[0],i_full)
        #period = bounce(x[settle_i:,:,:])

    else:
        fluid_yn = -1
        fluid_i = -i_full
        pinned_yn = -1
        pinned_i = -i_full
        spout_yn = -1
        spout_i = -i_full
        #period = -1

    # Set final exclusive state
    if spout_yn == 2:
        final = 5
    elif spout_yn == 1:
        final = 4
    elif pinned_yn == 1:
        final = 3
    elif fluid_yn == 1:
        final = 2
    elif settle_yn == 1:
        final = 1
    else:
        final = 0

    # Final values
    out = (round(values[0]/0.006,4), round(values[2],4), settle_yn, settle_i,
          fluid_yn, fluid_i, pinned_yn, pinned_i, spout_yn, spout_i, final)
    print out
    return out

# Settings
print 'Analyzing grid'
```

```
dumps = dirscan()

pplot = numpy.zeros((len(dumps),11))

# Start job server
ppservers = ()
ncpus = 3
job_server = pp.Server(ncpus, ppservers=ppservers)
print 'Starting pp with', ncpus, 'workers'

# Main parallel job command
jobs = [(job_server.submit(analyze,(dump,),
            (settled,fluid,pinned,spout,bounce,get_values),
            ("numpy","os","sys","math",))) for dump in dumps]

for i in xrange (0, len(jobs)):
    pplot[i,:] = jobs[i]()

job_server.wait()

# Write output file
write_pplot(pplot, 'stats.csv')

job_server.print_stats()

# Serial job command
#for dump in dumps:
#    analyze(dump)
```

8 VITA

Education

B.S. in Physics, May 2007
The University of Mississippi, University, MS

Employment

Design Engineer
Radiance Technologies, Ruston, LA, February 2012 - Present

Graduate Assistant
National Center for Physical Acoustics, Oxford, MS, May 2007 January 2012

Teaching Assistant
UM Department of Physics and Astronomy, Oxford, MS, August 2007 - May 2009

Publications

JASA - Journal of the Acoustical Society of America
Infrasound measurements during hurricane Katrina
Ronald Wagstaff, Eric Goggans, Heath Rice, and Carrick Talmadge
NCPA, Univ. of Mississippi, 1 Coliseum Dr., University, MS 38677

Signal processor for detection of signals in cluttered environments
Ronald Wagstaff and Heath Rice
NCPA, Univ. of Mississippi, 1 Coliseum Dr., University, MS 38677

SPIE - The International Society for Optics and Photonics
Improving temporal coherence to enhance gain and improve detection performance
Ronald A. Wagstaff, Heath E. Rice
NCPA, Univ. of Mississippi, 1 Coliseum Dr., University, MS 38677