Ph11: Hurdle 1

Ben Bartlett
bartlett@caltech.edu

Abstract

Given a set of glass spheres of various sizes resting atop a bed of sand which is disturbed at regular intervals by an insect searching for food, we utilize a computational model to simulate the movements of the insect over the sand and the subsequent effects on the sand levels and sphere heights. We model the insect's attraction to the food sources as analogous to a potential well created by a group of electric point charges and model the various sand heights in a large square matrix. The distance the spheres sink after each iteration of the program was found by treating the sand underneath moving objects as behaving in a slightly vibrofluidized state. The food is spread evenly over the bed of sand, but the presence of the spheres causes an asymmetry in the food distribution. This, combined with the random initial placement of the beetle, creates a highly stochastic system, suggesting the use of numerical methods to solve the problem. Using Monte Carlo methods, we determine a distribution of final sphere heights with respect to sphere radii and time elapsed.

1 The Problem

"Solid glass spheres of various sizes sit on a bed of sand in a large glass box. On this mini-desert, beetles crawl vigorously in search of food. The masses of the glass spheres range from about that of individual sand grains to much larger than the beetles. Every morning, beetle food and a small amount of sand are shaken uniformly onto the surface. This continues until the added sand has a height many times that of the largest glass spheres. Determine the height distribution above the original sand surface for the various size spheres as a function of time."

1.1 Assumptions

For the purposes of simplifying the problem, we make several assumptions in the interpretation of the problem:

- The beetle is unable to fly or crawl over the glass spheres and can push them around.
- The beetle follows specific search patterns for food as outlined in 2.1.
- Coefficient of kinetic friction of sand against glass is the same as that of glass against glass ($\mu_k = 0.4$) [1] due to similar molecular structure.
- For physical purposes, the size of a grain of sand is assumed to be sufficiently small enough to exhibit fluid-like behaviors when excited through vibration
- For logging purposes, the size of a grain of sand is $1mm$ diameter (chosen to make the size of the sandLevels array manageable - see section 3.4)
- The bed of sand is a $1m \times 1m$ box
- The beetle is approximated as spheres with radius $10mm$
- Mass of beetle is $30g$
- Glass spheres range from size of a grain of sand ($.5mm$ radius) to 20 times the beetle's radius ($200mm$)
- Density of sand $\rho_{glass} = 1281 \frac{kg}{m^3} = .001281 \frac{g}{mm^3}$ [2]
- Density of glass $\rho_{glass} = 2.6 \times 10^3 \frac{kg}{m^3} = .0026 \frac{g}{mm^3}$ [3]
- The beetle must be within 1mm of its food to consume it.
- Some beetles (Hercules beetles) can lift upwards of 850 times their body mass [4], so we assume the beetle can push spheres of arbitrarily large masses over the sand without necessarily lifting them.
- The beetle will eat all food particles before a new layer of sand and food is scattered in the box, even if large spheres must be pushed to access the remaining food.
- All sand and food particles falling on glass spheres will slide off of the sphere and land on the sand below.
- Spheres will immediately begin to sink at their terminal velocity in sand that is in a vibrofluidized state.

2 Analysis

The problem was originally interpreted to be a granular convection problem - the argument could be made that the beetle's movements serve purely to provide a continuous impulse into the system, vibrofluidizing the sand particles and causing the larger glass balls to rise to the top, while the smaller ones sink to the bottom due to respective sphere packing densities. Thus, the specific movements of the beetle could be ignored in favor of a simpler approach. However, supplying enough energy to keep a large bed of sand and glass beads in a completely vibrofluidized state (as opposed to a locally perturbed state) would require an unrealistically large energy output from the beetle. Furthermore, the presence of large amounts of

very small particles in the box would ensure that almost all of the glass spheres would stay at the top of the sand in this model, due to their comparatively massive size. Finally, the granular convection explanation is optimal for materials of the same density, but the density of glass is far greater than sand, so sphere packing is not important enough of a variable to play a major role in height distribution. Thus, the problem was approached in less simplistic manner that concentrated more on the beetle's movements and their effects. To simulate these movements and the subsequent effects on the system, a Python script was created to simulate the conditions in the problem and run the "experiment" many times. There were two major components to this script: approximating the beetle's movements in its search for food and calculating the effects said movement would have on the sand height and the sphere heights. Both of these problems are discussed below.

2.1 Approximating the beetle's motion

Several methods of approximating the beetle's movements were considered, ranging from simply pointing the beetle in the direction of the nearest piece of food to using Lagrangian mechanics and the principle of least action to determine an action-minimizing path for the beetle. However, the former method was decidedly too simple to mimic the movements of a beetle, while the latter was too complex, making the beetle's path too efficient to seem realistic. However, in the latter option, the food particles would be modeled like point charges in an electric field; this idea suggested a simpler and more elegant solution: the beetle would simply follow the gradient of the potential field created by the food particles, like a hiker following the path of steepest ascent.

Thus, if we model the attractive potential the food creates as analogous to that of a set of point charges, then the potential well for an individual food particle is:

$$V = \frac{Q}{4\pi\epsilon_0 r} \propto \frac{1}{r}$$

where Q is charge and r is separation. Since all food particles are considered to be identical in this model, we can discard Q from our calculations and discard $\frac{1}{4\pi\epsilon_0}$ as an arbitrary constant. We can reparametrize this potential with respect to x and y in the form:

$$V = \frac{1}{r} = \frac{1}{\sqrt{\Delta x^2 + \Delta y^2}} = \frac{1}{\sqrt{(x-x_0)^2 + (y-y_0)^2}}.$$

This will give us the potential, as experienced from (x, y), for a single food particle at a coordinates (x_0, y_0). Since the food and sand were stated to be distributed uniformly, they were modeled as falling down onto the box below on a regularly spaced grid pattern; if they initially fell on a glass sphere, they were assumed to have slid off before the beetle was released. (Though a quick calculation, given a coefficient of kinetic friction of sand on glass of $\mu_k = 0.4$, will show that the sand would stay on top of the spheres for any angle $\phi = 62°$ above the horizontal, we assume the sand and food was falling from a significant enough height that the disturbance resulting from the collision with the sphere was sufficient to knock it to the edges.) We sum the potentials of the n food particles in the bed of sand and obtain the formula:

$$V(x,y) = \sum_{i=1}^{n} \frac{1}{\sqrt{(x-x_i)^2 + (y-y_i)^2}},$$

where x and y are the coordinates of the beetle and x_i and y_i are the coordinates of the i^{th} food particle. Since the beetle will want to maximize its height on the potential function at any given moment (by approaching food) while expending the least amount of energy (by moving the shortest distance), it will want to follow the path of steepest ascent on this potential function, given by the gradient of the potential function, such that $\vec{v} \propto \vec{\nabla} V$. Thus,

$$\vec{v}(x,y) \propto \vec{\nabla}\left(\sum_{i=1}^{n} \frac{1}{\sqrt{(x-x_i)^2 + (y-y_i)^2}}\right).$$

Since the gradient returns a vector of partial derivatives, and $\frac{\partial}{\partial x}(f_1 + f_2) = \frac{\partial}{\partial x} f_1 + \frac{\partial}{\partial x} f_2$ for arbitrary f_1 and f_2, then the gradient of a sum of potential functions is equivalent to the sum of the gradients of the potential functions. Thus,

$$\vec{v}(x,y) \propto \vec{\nabla}\left(\sum_{i=1}^{n} V_i\right) = \sum_{i=1}^{n} \vec{\nabla} V_i = \vec{\nabla} V_{total}.$$

Computing this manually, we have:

$$\vec{\nabla} V_i = \left\langle \frac{x_i - x}{\sqrt{(x-x_i)^2 + (y-y_i)^2}^3}, \frac{y_i - y}{\sqrt{(x-x_i)^2 + (y-y_i)^2}^3} \right\rangle.$$

Thus, if we assume that the beetle moves at a constant velocity, then we simply solve for some scalar multiple (the beetle's speed, denoted as s_b) of the unit vector of $\vec{v}$ to find the true velocity:

$$\begin{aligned} \vec{v} &= s_b \widehat{\vec{\nabla} V_{total}} \\ &= s_b \sum_{i=1}^{n} \left[\frac{\left\langle \frac{x_i - x}{\sqrt{(x-x_i)^2 + (y-y_i)^2}^3}, \frac{y_i - y}{\sqrt{(x-x_i)^2 + (y-y_i)^2}^3} \right\rangle}{\sqrt{\frac{(x_i-x)^2 + (y_i-y)^2}{((x-x_i)^2 + (y-y_i)^2)^3}}} \right] \end{aligned} \quad (1)$$

2.2 Effects of object motion on sand elevation levels

Due to time constraints on both computational complexity and coding time, a simplified model of interactions between the beetle, the spheres, and the sand elevation levels had to be used. We approximate the beetle's geometry as that of a sphere. If we have an arbitrary sphere of mass m, radius R, and density ρ sitting in a bed of sand at some depth such that it has an effective base of area A, then we have a downward force of $F_g = mg$ and an upward force of F_{sand}. If we vibrofluidize the sand with a disturbance, such as the beetle or a sphere moving in the sand, we can loosely approximate the equations of force in the sand as that of a viscous fluid, in the form $F_{sand} = F_b + F_v$ where F_b represents the effective buoyancy force and F_v is the resistive force due to viscosity. Since the spheres would be sinking at a very low velocity through what is effectively a very viscous medium, we use equations describing Stoke's drag for low velocity objects in high-viscosity non-Newtonian fluids to model the sinking objects. We solve for the buoyant force

$$F_b = mg\frac{\rho_s}{\rho_g},$$

where m is object mass, ρ_s is the density of sand, and ρ_g is the density of glass, and use the equation for resistive force in Stoke's drag for F_v of the form:

$$F_v = 6\pi\eta R v$$

for viscosity η, sphere radius R, and sinking velocity v. If we assume that the sphere quickly accelerates to its terminal velocity and falls at constant velocity v_t, we have:

$$\begin{aligned} mg\frac{\rho_s}{\rho_g} + 6\pi\eta R v_t &= mg \\ 6\pi\eta R v_t &= mg(1 - \frac{\rho_s}{\rho_g}) \\ v_t &= \frac{mg(1 - \frac{\rho_s}{\rho_g})}{6\pi\eta R} \end{aligned}$$

Expressing the mass of a glass sphere as $m = V\rho_{glass}$ for volume V, we have:

$$\begin{aligned} v_t &= \frac{V\rho_{glass} \cdot g(1 - \frac{\rho_s}{\rho_g})}{6\pi\eta R} \\ &= \frac{\frac{4}{3}\pi R^3 \rho_{glass} \cdot g(1 - \frac{\rho_s}{\rho_g})}{6\pi\eta R} \\ &= \frac{2R^2 g(\rho_g - \rho_s)}{9\eta} \end{aligned} \quad (2)$$

In a paper by Knappman, [5] it was found that in a bed of sand that was vibrofluidized by passing air through it, viscosity with respect to flow rate w in L·sec^{-1} is of the form $\eta(w) = 2.3 \cdot \frac{78}{w-78}$, where 78L·sec^{-1} was the critical flow rate where sand started behaving like a liquid. Since the sand in the box would only be barely fluidized by the comparably small vibrations of the beetle and the objects, we choose $w = 79$L·sec^{-1} as our choice for η, though other choices could be made based on empirical data from experiments involving beetle movements across sand. Note that even if this choice of equivalent flow rate is incorrect, the ratio of the terminal velocities $\frac{v_{t1}}{v_{t2}} = \frac{R_1^2}{R_2^2}$ for spheres with respect to masses remains independent of η.

For the purposes of this program, we assume that when the spheres and/or beetles are moving, the sand is locally vibrofluidized and the moving objects will quickly accelerate to their terminal velocity v_t and sink at that velocity, stopping immediately once they stop moving across the sand. As the objects sink, the sand levels are lowered under the object. However, since sand must be conserved, an equal amount of sand is positively displaced in a ring with radius $\sqrt{2}$ times the radius of the object depressing the sand.[1] This is consistent with qualitative observations about sand interactions: dragging a rock across the sand with a constant downward force will create a trench with a depressed center and elevated sides, but the rock will remain at roughly the same level in the sand, since the material in front of it that it elevated while moving was later depressed; however, grinding the rock with the same downward force in a tight circle in the same place in the sand will push it further into the sand.

Note that the degree to which the spheres are depressed in the sand is not only proportional to their terminal velocity in fluidized sand, but also to the amount they are moved in a localized region. Thus, it is expected that the displacement of the spheres from sand level, given no outside interference from other spheres, should be proportional to the cube of the radius, since the terminal velocity is proportional to the square of the radius, and the number of times the beetle must disturb the sphere to access food around its perimeter is proportional to the circumference of the sphere, which is proportional the the radius.

[1] The $\sqrt{2}$ ratio of outer to inner radii was chosen so that the altitudes of displacement would have the same magnitude - a sphere with radius $10mm$ that sank $1mm$ in the sand would create a ring of outer radius $10\sqrt{2}mm$ and inner radius $10mm$ also with height $1mm$, but where the sand is elevated, not depressed.

2.3 Calculating the distance a sphere moves on collision

When the beetle collides with a sphere, we approximate the collision as an impulse into the sphere applied once every time the beetle moves every cycle in the program - thus, the beetle repeatedly "kicks" the spheres as it moves for a distance which is dependent on the coefficient of friction in the sand and the change in height of the sand. We approximate the energy loss due to displacing the sand beneath the sphere while rolling as equal to that of simply sliding the sphere over an unyielding body of sand. Thus, the problem arises to calculate the distance the sphere travels over each "kick" the beetle imparts into it. This is a fairly simple calculation. For an initial velocity of the beetle of v_{i_1} and a stationary sphere, we calculate the final sphere velocity after the collision with the beetle using elastic collisions:

$$v_{f_2} = \frac{v_{i_2}(m_s - m_b) + 2m_b v_{i_b}}{m_b + m_s},$$

where m_b and m_s are the beetle and sphere masses, v_{i_s} and v_{f_s} are the initial and final sphere velocities, and v_{i_b} and v_{f_b} are the initial and final beetles, respectively. Since the sphere is initially stationary, we have:

$$v_{f_2} = \frac{2m_b v_{i_b} \sin\theta}{m_b + m_s},$$

where θ is the angle between the beetle's velocity vector and the edge of the sphere. The reason for the presence of $\sin\theta$ in this term is due to the fact that the vector orthogonal to the edge of the sphere will push the sphere in that direction, while the one parallel to the sphere will only torque it about its axis in that direction. Now, we solve for the distance the sphere will travel over the sands of changing heights with this initial velocity. Given conservation of energy and redefining the sphere velocity immediately after the collision, v_{f_2} as the initial velocity after the collision, v_i, we have:

$$\begin{aligned} E_i &= E_f \\ \frac{1}{2}mv_i^2 &= F_\mu \cdot \Delta x + \Delta V \\ \frac{1}{2}mv_i^2 &= \mu_k mg\Delta x + mg\Delta h \\ v_i^2 &= 2g(\mu_k \Delta x + \Delta h) \\ v_i &= \sqrt{2g(\mu_k \Delta x + \Delta h)} \end{aligned}$$

Thus, the initial velocity required to slide over sand with coefficient of kinetic friction μ_k and change in elevation Δh is $\sqrt{2g(\mu_k \Delta x + \Delta h)}$. However, since h is not necessarily a linear function, we compute the deceleration over time and apply it every iteration in the program:

$$\begin{aligned} v_f^2 &= v_i^2 + 2a\Delta x \\ 0 &= \sqrt{2g(\mu_k \Delta x + \Delta h)}^2 + 2a\Delta x \\ -2a\Delta x &= 2g(\mu_k \Delta x + \Delta h) \\ a &= -g(\mu_k + \frac{\Delta h}{\Delta x}) \end{aligned} \quad (3)$$

Thus, we set the initial velocity of the sphere immediately after the collision to be $v_{f_2} = \frac{2m_b v_{i_b} cos\theta}{m_b + m_s}$ and decelerate this velocity by the deceleration $a = -g(\mu_k + \frac{\Delta h}{\Delta x})$ every iteration of the program until the sphere's velocity is less than or equal to zero, allowing us to compute the distance traveled while taking into account the effects of changing elevation levels without the use of integration.

3 Implementation[2]

3.1 Program setup and initial conditions

A Python script was developed to simulate the conditions in the problem and yield results that would be insertable into Mathematica for further analysis. After setting up a number of key variables, the script began by placing many spheres, which were indexed in a large list, of random sizes ranging from the size of a grain of sand to 20 times that of the beetle at random locations in a 1×1 meter box.[3] After sphere placement was finished, the program simulated scattering a 5mm layer of sand and food uniformly over the bed of sand and glass. An overlapTest() function was developed to check whether sand/food would overlap with the spheres when falling down. If it did, it returned the index of the sphere the overlap occurred with and scaled the displacement vector of the particle from the center of the sphere such that the magnitude was equal to the radius of the sphere, simulating the particle sliding off the edge of the sphere. This caused a large portion of the food to collect around the edges of the spheres, with larger spheres having a higher food concentration on their edges than the smaller spheres, since the amount of food falling on the sphere was proportional to the area, while the amount of space it could collect in was proportional only to the circumference.

[2] The entirety of the source code for this project is attached at the end of this paper, should you need to reference it. Also, a digital version can be downloaded at https://db.tt/Qn4vtMFD. Note that this requires a graphics package if you want to run the animation engine. The package can be downloaded at http://mcsp.wartburg.edu/zelle/python/graphics.py

[3] The script was designed such that all variables could be easily changed by editing their initial values, so all values listed in this section were simply the default values for the initial runs. It was later found that large numbers of spheres in a box this small would produce tremendous amounts of interference with the other spheres' final heights per iteration, with smaller spheres getting caught in larger spheres' potential wells, so a set of runs with only one sphere per box was run for a large number of instances to simulate a box with spheres far enough from each other such that interaction between spheres could be negligible. See section 4 for more detail on this.

A rudimentary animation engine was also developed using the Python graphics package to qualitatively check the operations being developed in the program, as shown in Figure 1.

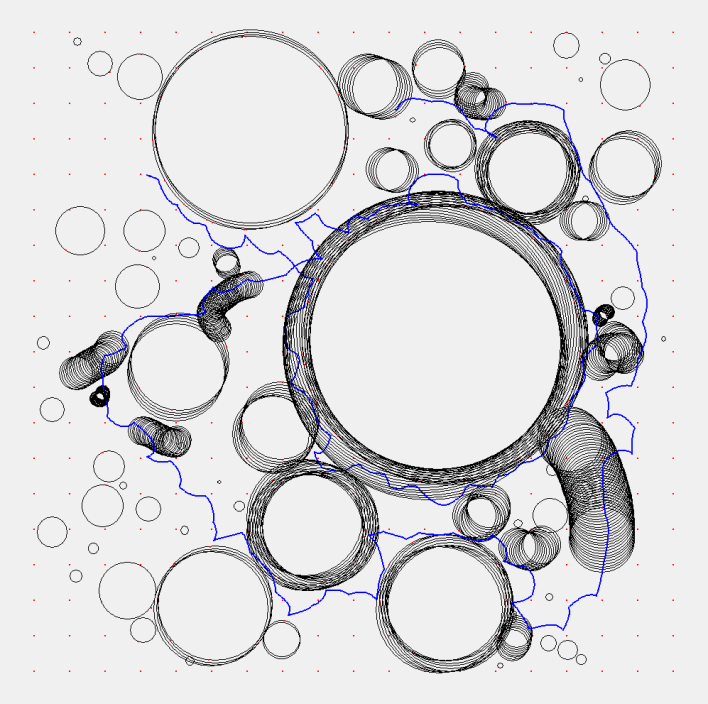

Figure 1: An output of the animation algorithm from a top-down view of the box. The blue line represents the path traveled by the beetle, the red dots are food particles, and the black circles are positions of the spheres over time as they are moved by the beetle. Note that the beetle width is not displayed here, but is larger than the width of the blue line.

Following the calculation and placement of food positions, a while loop was created that evaluates the quantity checkSum, which represents the total number of food particles remaining, and loops until checkSum equals zero. This represents the body of the simulation, where the beetle moves itself and the spheres, affecting sand elevations. Four main functions are called during this loop: moveBeetle(), depressSand(), moveSphere(), and eat(), each of which utilize many smaller subfunctions. (See the source code at the end of this paper for details.) These will be discussed below.

3.2 The moveBeetle() function

The moveBeetle() function calls a function to determine the path of steepest ascent into the potential field created by the food particles by calculating the gradient of the function, as described in 2.1. It then normalizes the vector to have a magnitude corresponding to a predefined velocity, and increments the beetle's position by the x and y components of the vector. This ensures that, when food is plenty, the beetle will generally head to the nearest source of food, and when food is almost completely consumed, it will head towards other available regions of the box. Though more efficient search patterns exist, we assume the beetle lacks the capacity to

plan ahead, so at any given time, it is focused only on immediately maximizing its food potential function. The beetle is allowed to move to any location within the box, while the spheres are limited to the size of the box minus a buffer zone to keep them from being caught in the corners of the box.

3.3 The eat() function

If the beetle is within a distance from a food particle that is equal to $eatDistance = beetle\,radius + 1$, then it will "eat" the food, turning off the corresponding element in the foodPresent array, which is a large square array initially composed entirely of 1's with edge length equal to the square root of the total number of food particles (which will always be an integer, since food is dispersed on a square matrix). The food particles have separate indices and x/y positions, so the foodPresent array turns from a 1 to a 0 the element at the vertical and horizontal index of the array corresponding to the closest particle whose distance is less than eatDistance, effectively excluding that particle from future potential function calculations. One of the biggest challenges in the course of writing this program arose in this function, where a misuse of the min() function in Python on a two-dimensional array was causing the foodPresent array to fail to update for particles that were outside their natural alignment due to having slid down the sides of spheres. This caused the beetle to oscillate back and forth endlessly. The min() function was replaced with a simple brute-force algorithm to determine the distances from the beetle to all of the spheres and take the smallest of them as the minimum of the distances. Fortunately, this solved the problem without wasting too much computational power compared the computation power required for the rest of the program.

3.4 The depressSand() function

As the beetle moves around the sand, it activates the depressSand() function for both itself and the spheres it moves. As discussed in 2.2, the velocity at which the beetle or sphere sinks while moving in the sand is equal to its terminal velocity while falling through the vibrofluidized sand. The total displacement value is this velocity divided by a constant representing the number of iterations in one second, which is arbitrarily set as 10, but cancels out through various calculations throughout the program. This displacement value is subtracted from a circular subset of the square sandLevels array, with a radius and center corresponding to the radius and position of the sphere on the sand. The sandLevels array has dimensions $\frac{boxSize}{sandSize}$, where boxSize is the edge length of the box in mm, and sandSize is the size of a grain of sand in mm. To prevent the calculations from taking infeasibly large amounts of memory, sandSize is set to 1mm, since the size of the sand grains is not included in the fluidity calculations (we assume they are sufficiently small as to display the type of behavior shown in the study by Knappman), and as such has no bearing on the results of the program other than the resolution of the final sandLevels array. However, the beetle displaces the same amount of sand upwards in a ring outside its depression zone as it displaces downward, ensuring that sand is conserved and that the beetle will stay at a constant elevation if it keeps moving across the sand to regions where it has not already been. An example of the sand elevation output after one iteration in the program from the depressSand() function when applied to the beetle and the spheres can be seen in Figure 2.

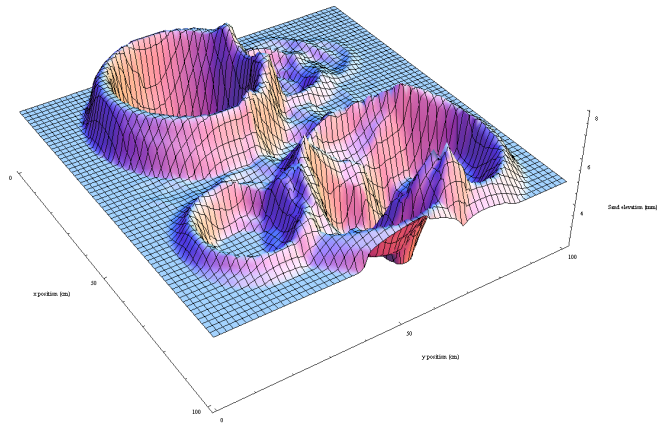

Figure 2: Sand elevation levels after one iteration in the program, plotted in Mathematica, with the height exaggerated to show detail. Note that the x and y axes span 100 cm, while the height variation for z is about 4 mm.

3.5 The moveSphere() function

When the beetle comes in contact with a sphere in the sand, it activates the moveSphere() function, which calculates the angle at which the beetle is colliding with the sphere and moves the sphere in the direction orthogonal to the line that is tangent to the edge of the sphere at the point of contact. (Since the vector component parallel to the tangent line would simply torque the sphere into rotating about its axis in that direction, only the orthogonal vector is used.) It then starts the sphere moving with initial velocity and deceleration as calculated in 2.3. For simplification purposes, and in order to discretize the continual movement of the sphere into the finite number of iterations in the program, we assume the beetle does not resume moving until the sphere has reached its final destination. The program then updates the arrays representing the x and y positions of the spheres to their new values, and, if the graphical interface is enabled (it is disabled for long runs to save computation power), it draws the new position of the spheres. (It does not erase the old sphere position, which is the reason for the many concentric circles in Figure 1.)

3.6 Data exportation

At the end of the while loop, the program compiles all of the final data describing sand elevation, sphere height, and sphere height trends over time into a .dat file formatted in a manner that, with a small modification, can easily be imported into Mathematica for further analysis. These files are set up as a set of x, y, z coordinates for the sand heights and sphere heights over time, and as a set of x, y coordinates for final sphere heights at the end of the simulation. When exporting the sand elevation data, the program

reduces the size of the matrix to include every element at coordinates $[n, n]$, where n is a predefined value stored in the variable mathematicaExportResolution. This allows the data to be easily plottable in Mathematica, as data sets too large will crash the program. Optionally, it will also export all data per iteration, though this option was disabled during the computation with the Amazon Web Services cluster (see section 4) to halve the amount of data that must be downloaded following the simulation.

4 Results

4.1 Initial results, multiple spheres per simulation

The program was initially run with a large number (about 50) of spheres packed into the box in order to gather the maximum amount of data per run as was possible. However, upon analyzing this initial data, it was found that no clear correlation between sphere size and elevation above the original sand level was present, as shown in Figure 3.

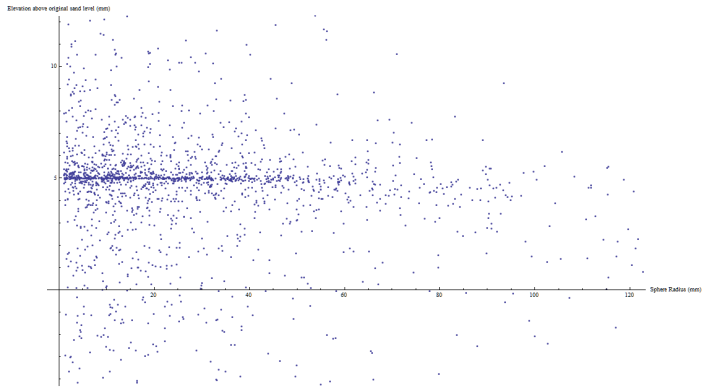

Figure 3: Sphere elevation as compared to sphere radius. Original data obtained from using a large number of spheres per simulation.

4.2 Single-sphere simulations

After further analysis, this problem was found to be due to the fact that many of the smaller spheres were getting caught in the potential wells created by the larger spheres after tens of iterations had passed, resulting in a set of data that was, for all intents and purposes, random noise. To remedy this problem, simulations with only one sphere per box were conducted. This would approximate having a large box with spheres placed sufficiently far away from each other so as to avoid any problematic interactions between them that would introduce extraneous noise into the data. The initial results of the first few runs of this simulation are shown below.

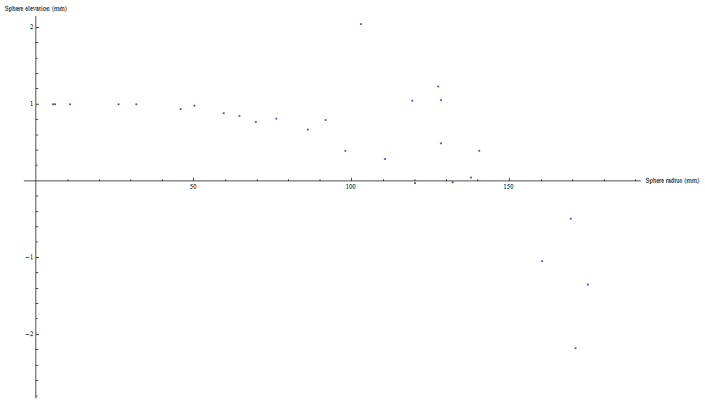

Figure 4: Initial data from single-sphere simulation after one iteration of the program. Notice the much clearer correlation between mass and elevation levels, similar to the ones described in 2.2.

4.3 Large-scale computation of single-sphere simulations

As observable in Figure 4, this shows a far clearer correlation in sphere heights compared to sphere radii, and even seems to follow the cubic function discussed at the end of 2.2. Though the single-sphere simulation produced data with lower noise levels, far less data could be gathered per simulation. This necessitated the use of large numbers of simulations, which, due to the computational complexity and inefficiency of the program, would be problematic even for powerful desktop processors. To solve this problem, a large-scale version the simulation was run on a computing-optimized cc2.8xlarge cluster rented from Amazon Web Services[4]. The program was modified to accept a programID argument in the command line of the SSH terminal, such that it could use this to output data to non-conflicting files, and the each of the 32 cores was given a bundle of 25 instances of the simulation to run, each with 20 iterations, except the last core, which was kept idle to ensure the system remained responsive to user input. This resulted in a total of 775 individual tests each with 20 iterations. The cores ran near 100% capacity for approximately 7.5 hours before they finished their calculations and wrote the final data to the .dat files, which were then combined and inserted into Mathematica for further analysis.

The results from this simulation yielded a data set that was much more in line with what the calculations in the first part of this paper predicted, albeit with a rather large error bound appearing towards large numbers of iterations. Shown in Figure 5 is the final sphere height distribution compared to sphere radii after all 20 iterations of the simulation had been completed.

Data had also been gathered for each of the iterations in the 20 iteration run, allowing the trend in sphere heights to be evaluated over time. Figure 6 shows a 3D plot of the sphere elevation data with respect to time (number of iterations, measured in days) and

[4]Stats on the computing cluster: 64-bit processor, 32 cores, 60.5GB RAM, 3.36TB storage, 10Gigabit network connection, 88EC2 compute units. (One EC2 compute unit provides the equivalent CPU capacity of a 1GHz 2007 Xeon processor.) [6]

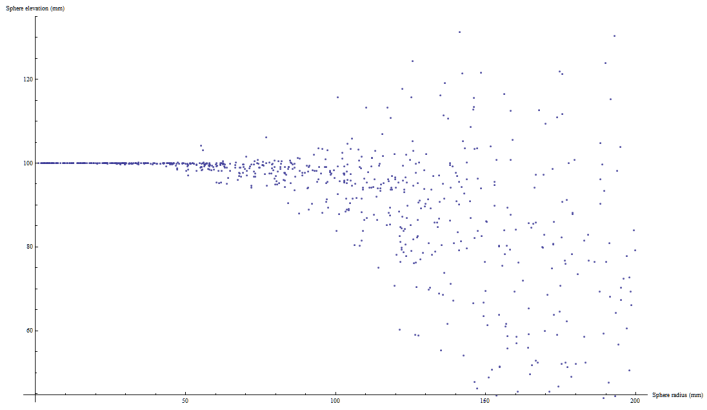

Figure 5: Final results of sphere height over sphere radii from the large-scale simulation after all 20 iterations had completed.

sphere radius.

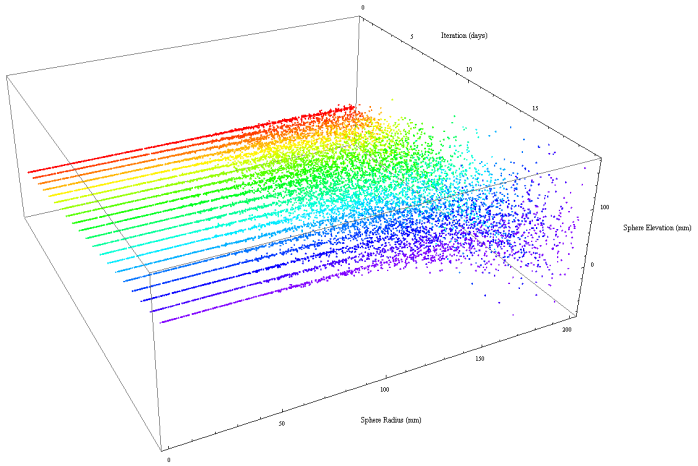

Figure 6: Results of sphere heights tracked from back to front over time (in 20 discrete, color-coded iterations) and from left to right over sphere radius.

4.4 Cubic regression on data sets

Though the more practical method for determining height distribution above the surface would be to simply subdivide the data found in Figure 6 into subsets and make a data table for height compared to radius and time, the problem specifically asked to give the height distribution as a function of time. This suggested we perform some sort of non-linear regression on the data to yield a polynomial function of height with respect to sphere radius and time. Using the previously derived cubic relation between sphere height and sphere elevation found in 2.2, Mathematica's $NonlinearModelFit[]$ function was utilized to create a third-order regression to approximate the sphere elevation values over their radii as a plottable function.

Given conservation of sand, we know that the average height of the sand at n iterations past starting is $5 \cdot n$ mm, since 5mm of sand is added per iteration, as discussed in 3.1. As the size of the glass spheres approaches that of the grains of sand, the beetle would collide into the spheres less and the depression value of the collision in the sand would approach zero. Therefore, we can assume that the average height for glass spheres with size similar to that of sand is equal to the average height of the sand at some number of iterations. Thus, the function must take the form ax^3+100, where a, we presume by looking at the data, is negative. Performing this regression on the data for the final sphere positions yields the function describing sphere elevation of

$$h(r) = -5.62 \times 10^{-6} r^3 + 100,$$

where r is the sphere radius. Shown in Figure 6 is the overlay of this function onto the data.

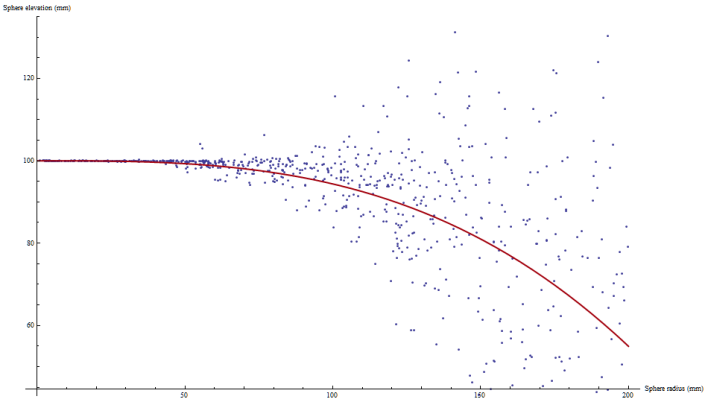

Figure 7: The regression mentioned above overlaid over Figure 5.

Similarly, we can use a three-dimensional analog of this regression method to analyze the data over the course of several iterations. Applying this three-dimensional regression would yield some function in the form $z = axy^3 + 5x$, where x represents time in iterations, y represents sphere radii, and z represents sphere elevation. The rationale for the use of this form of function is as follows: for sufficiently small sphere radii (geometrically, near the line $y = 0$), z depends linearly on x, since 5mm of sand are added every iteration. Along each iteration (where there is zero change in x), z varies with y^3. However, at the beginning of the simulation (seen in Figure 6 as the red data set in the back), there has been almost no disturbance in the system, so z is dependent only on the initial sand levels, which increase with x (increase over time). (Hypothetically, should a 0^{th} iteration have been performed, all spheres would be on exactly the same elevation of $z = 0$.) Thus, the degree to which y^3 is present in z is also linearly dependent on x. Therefore, along a given iteration, z varies with xy^3. Combining these two axes, we have our originally described function $z = axy^3 + 5x$.

Inserting this function into the $NonlinearModelFit[]$ function in Mathematica yields the regressed function

$$h(x, y) = -2.62 \times 10^{-7} xy^3 + 5x$$

. Qualitatively, this function fits very well with the expected contour of the function when plotted against the data, as shown in Figure 8.

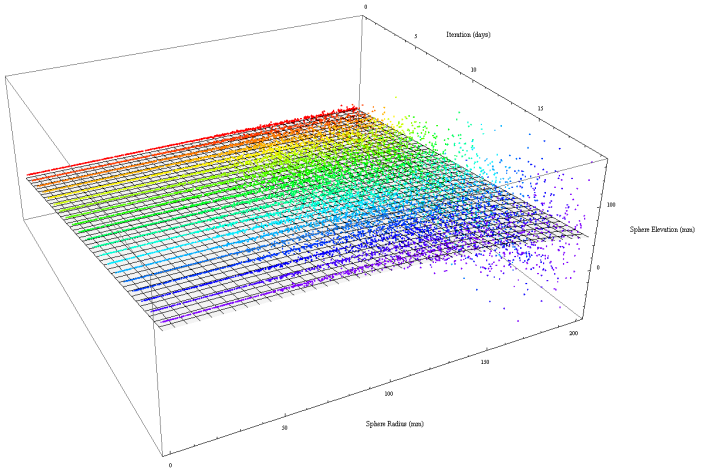

Figure 8: Figure 6 overlaid against the regression function.

4.5 Error bound analysis

As is visually obvious in Figure 7, the variation from the expected values of the sphere heights increases as radius increases. This is due to the fact that, due to higher food concentrations around the edges, larger spheres have more interactions with the beetle, causing the system to become more stochastic. This holds true over time as well, as shown in Figure 8, with the variation from the expected elevation values for small sphere radii remaining almost zero even after 20 iterations of the program. Furthermore, the variance from the expected values of height in Figure 8 at larger radii is almost zero for low iteration numbers, and is very large for high iteration numbers. Thus, we can qualitatively say that the variance of the system is dependent on the sphere radius and time.

This could suggest that we break up the data set into small pieces along the xy plane and determine the average variance along each small portion of the data set. We could then use a similar regression method as the one used in 4.4 to determine a function describing the magnitude of the variance with respect to x and y. However, not enough research was put into the analysis of the variance in this highly stochastic system to justify its regression into a particular type of function, as we do not know to what degree the variance is dependent on x and y. Thus, for the purposes of this paper, we express the magnitude of the variance as some general unknown error function varying along some relation to x and y, denoted as $\delta(x, y)$. Alternately, δ could simply represent n number of standard deviations for the sphere heights among some subsection $(\Delta x, \Delta y)$ of the overall data set, allowing us to make the prediction that some large percentage of spheres with a given radius and at a given time will be within a certain height interval.

5 Conclusion

Assembling all of the information gathered in the above sections into a single formula, the expected value for the sphere heights as a function with respect to time and sphere radii is:

$$h(t, r) = -2.62 \times 10^{-7} t r^3 + 5t \tag{4}$$

where t is time, measured in days (iterations), r is the sphere radius in mm, 5 represents the 5 mm of sand per day that is added along with the new food prior to the beetle being released, and -2.62 is a constant determined through experimental analysis, measured in arbitrary units of $mm^{-2} \cdot s^{-1}$to ensure unit consistency. Adding in our variance function, the range of possible sphere heights over time and radii would then be:

$$h(t, r) = -2.62 \times 10^{-7} t r^3 + 5t \pm \delta(t, r)$$

where $\delta(t, r)$ is the variance function representing the uncertainty of the height as the system becomes increasingly stochastic.

6 References

[1]:http://www.engineeringtoolbox.com/friction-coefficients-d_778.html

[2]:http://www.engineeringtoolbox.com/density-materials-d_1652.html

[3]:http://www.engineeringtoolbox.com/density-solids-d_1265.html

[4]:http://en.wikipedia.org/wiki/Hercules_beetle

[5]:http://www.physics.ohio-state.edu/~reu/99reu/final_reports/paper_knappman.pdf

[6]:http://aws.amazon.com/ec2/instance-types/

```
#Partial Solution to Ph11 Hurdle 1
#Ben Bartlett
##############################################################################################################
#############################################################

#Importations
from math import *
import random
from graphics import *
import copy
import operator
import time
import sys

#Start timer.  Used to estimate how long the group of simulations will take.
startTime = time.ctime()
drawMode = True #determines if the program uses the animations
if drawMode:
  print "drawMode = True"
  from graphics import *
else:
  print "drawMode = False"

#Functions
definitions##############################################################################################################
################################################

def placeSphere(): #places sphere at x,y with radius r
  """placeSphere(): used for sphere and beetle placement.  optionally returns x and y
  coordinates in a vector"""
  global r
  global x
  global y
  r = random.uniform(minSphereSize, maxSphereSize)
  x = random.uniform(r+bufferSize, boxSize-r-bufferSize)
  y = random.uniform(r+bufferSize, boxSize-r-bufferSize)
  return [x,y]

def overlapTest(xpsn,ypsn,r): #tests for overlaps
  """overlapTest(x position,y position,radius of testing particle):
    Tests to see whether a sphere or food placement overlaps with an existing sphere.
    Input r=0 for point particles.
    Returns -1 if no overlap is detected, else returns the index of the sphere it overlaps
    with"""
  for i in xrange(len(spheresX)):
    if sqrt((xpsn-spheresX[i])**2+(ypsn-spheresY[i])**2) < spheresR[i]+r: #if the sphere radii
    overlap
      return i
  return -1

def drawMap(): #draws a top-down map.  mainly used for testing purposes
  #print "drawing..."
  winString = 'Hurdle 1 - Iteration ',iteration," of ", numIterations-1
  win = GraphWin(winString, boxSize, boxSize) # give title and dimensions
```

```
  for i in xrange(len(spheresX)): #indexes each sphere and draws it
    sphere = Circle(Point(spheresX[i],spheresY[i]), spheresR[i])
    sphere.draw(win)
  for j in xrange(int(boxSize*foodDensity)): #gets respective x and y locations for all food
  points and draws them
    for i in xrange(int(boxSize*foodDensity)):
      food = Point(foodX[i][j], foodY[i][j])
      food.setFill("red")
      food.draw(win)
  return win

def sphereFall(xOriginal,yOriginal,n):
  """sphereFall(x position, y position, index of sphere food collided with):
    Moves food such that it is no longer on the sphere, simulating it sliding off."""
  #calculates the angle from the center of the sphere the food is located at and changes the
  locations accordingly
  theta = atan2((yOriginal-spheresY[n]),(xOriginal-spheresX[n]))
  Ret = [0,0]
  xRet = spheresX[n]+copysign(spheresR[n]*cos(theta), (xOriginal-spheresX[n]))
  yRet = spheresY[n]+copysign(spheresR[n]*sin(theta), (yOriginal-spheresY[n]))
  Ret[0] = xRet
  Ret[1] = yRet
  return Ret

def gradField(x,y):
  """gradField(x position, y position):
    Returns a unit vector in the direction of the gradient.
    Calculates the gradient with respect to position of the nearest 2*proximityValue food
    sources, modeled as if they are electric point charges.
    Since we can assume food that is sufficiently far away will not influence the beetle's
    behavior greatly, we use proximityValue to save computational time.
    The potential function takes the form of a summation of 1/sqrt((x-xi)^2+(y-yi)^2), where x
    and y are current position, and xi and yi are positions of particles being indexed.
    Since gradient of the sum of a function equals the sum of the gradient of a function, we
    can simply solve for a general form of the gradient as:
    grad(1/sqrt((x-xi)^2+(y-yi)^2)) = <(xi-x)/((xi-x)^2+(yi-y)^2)^(3/2);
    (yi-y)/((xi-x)^2+(yi-y)^2)^(3/2)>"""
  #convert x and y to nearest i and j indices
  xindex = int(round(x*foodDensity))
  yindex = int(round(y*foodDensity))
  global foodPresent
  grad = [0,0]
  for j in xrange(int(max(yindex-proximityValue, 0)), int(min(yindex+proximityValue, boxSize*
  foodDensity))):
    #sets up i to include the food pieces of xindex(+/-)proximityValue.
    #the min/max statements ensure it doesn't try and reference food outside of the box
    for i in xrange(int(max(xindex-proximityValue, 0)), int(min(xindex+proximityValue, boxSize*
    foodDensity))):
      if foodPresent[i][j] == 1: #only includes food which has not yet been eaten in the
      potential function
        xi = foodX[i][j]
        yi = foodY[i][j]
        grad[0] += (xi-x)/(((x-xi)**2+(y-yi)**2)**1.5)
        grad[1] += (yi-y)/(((x-xi)**2+(y-yi)**2)**1.5)
```

```
  gradMagnitude = sqrt(grad[0]**2+grad[1]**2) #normalize the vector to maintain constant movespeed
  grad[0] *= beetleVelocity/gradMagnitude
  grad[1] *= beetleVelocity/gradMagnitude
  return grad

def moveBeetle(x,y):
  """moveBeetle(beetle's x position, beetle's y position):
    Returns nothing.
    Moves the beetle along the grad() function"""
  global beetleX
  global beetleY
  grad = gradField(x,y)
  beetleX += grad[0]
  beetleY += grad[1]
  return grad

def depressionSize(m,R):
  """depressionSize(mass, area in contact with sand, [0 for beetle, 1 for sphere]):
    Returns a value proportional to the amount sand should be depressed for a given object and
    surface area."""
  return 2*(R**2)*glassDensity*g*(1-(sandDensity/glassDensity))/(9*sandViscosity*
  iterationsPerSecond)
  #return m/(2*pi*R**2)

def depressSand(xPos,yPos,size,mass): #modify this to be circular and use for spheres.
  """depressSand(x position, y position, radius of depressor object, mass of depressor):
    Modifies position of nearby sand particles to simulate the beetle or spheres leaving tracks.
    Currently this track is approximated as a depressed circle of radius size and an
    elevated circle of radius size*sqrt(2)"""
  global sandLevel
  if drawMode:
    global win
  sandIndexX = xPos/sandSize
  sandIndexY = yPos/sandSize
  outerArea = pi/4*(int(min(sandIndexX + size*outerInnerRatio, boxSize/sandSize))-int(max(
  sandIndexX - size*outerInnerRatio,0)))**2 #figure out areas of circles so we can ensure sand
  is conserved
  innerArea = pi/4*(int(min(sandIndexX + size, boxSize/sandSize))-int(max(sandIndexX - size,0)))**2
  for j in xrange(int(max(sandIndexY - size/sandSize,0)), int(min(sandIndexY + size/sandSize,
  boxSize/sandSize))): #sets up loop to depress central circle
    for i in xrange(int(max(sandIndexX - size/sandSize,0)), int(min(sandIndexX + size/sandSize,
    boxSize/sandSize))):
      separation = sqrt(((i-sandIndexX)*sandSize)**2+((j-sandIndexY)*sandSize)**2) #calculate x
      and y distance from center of beetle
      if separation <= size: #select a circular region of the square superset
        if (outerArea-innerArea) != 0:
          sandLevel[i][j] -= depressionSize(mass, size)
          #print "Object at ", beetleX,", ", beetleY, " is depressing sand point: ", i, ", ",
          j, " to level ", sandLevel[i][j], " by a change of ", depressionSize(mass, size)
          #print "Plotting depression at screen point ", int(round(i*sandSize)),",
          ",int(round(j*sandSize))
          #depressionDot = Point(int(round(i*sandSize)), int(round(j*sandSize)))
          #depressionDot.setFill("yellow")
          #depressionDot.draw(win)
```

```
  for j in xrange(int(max(sandIndexY - size*outerInnerRatio/sandSize,0)), int(min(sandIndexY + size*
  outerInnerRatio/sandSize, boxSize/sandSize))): #sets up loop to elevate outer circle
    for i in xrange(int(max(sandIndexX - size*outerInnerRatio/sandSize,0)), int(min(sandIndexX +
    size*outerInnerRatio/sandSize, boxSize/sandSize))):
      separation = sqrt(((i-sandIndexX)*sandSize)**2+((j-sandIndexY)*sandSize)**2)
      if separation <= size*outerInnerRatio and separation >= size: #makes sure the point is not in
      the inner circle
        if (outerArea-innerArea) != 0: #for some very small spheres, the program cannot
        discretize their values into the matrix, so we ignore them to prevent division by zero
        errors.
          sandLevel[i][j] += depressionSize(mass, size)/(outerArea-innerArea)*innerArea #this
          step ensures that the total amount of sand is conserved - for every indentation,
          there must be a rise in sand further from the epicenter
          #print "Object at ", beetleX,", ", beetleY, " is elevating sand point: ", i, ", ", j,
          " to level ", sandLevel[i][j], " by a change of ", depressionSize(mass,
          size)/(outerArea-innerArea)*innerArea
          #print "Plotting elevation at screen point ", int(round(i*sandSize)),",
          ",int(round(j*sandSize))
          #elevationDot = Point(int(round(i*sandSize)), int(round(j*sandSize)))
          #elevationDot.setFill("green")
          #elevationDot.draw(win)

def sphereMass(n):
  return 4/3*pi*spheresR[n]**3*glassDensity #calculates volume and multiplies by density

def moveSphere(xpos,ypos,mass,grad):
  """moveSphere(x position, y position, mass of collider particle):
    This is a two-part function.  First, it checks to see if the beetle is overlapping the
    spheres.
    Then, if it is, it calculates an elastic collision to determine the rebound from the sphere
    and moves the sphere accordingly.
    The sphere motion is, for the moment, instantaneous, though dynamically moving spheres may
    be implemented given enough time.
    This function also works for when the spheres roll over the food, so the beetle must push
    them out of the way."""
  if drawMode:
    global win
  c = overlapTest(xpos, ypos, beetleSize)
  if c != -1: #if the beetle has collided with a sphere
    #set up a vector representing displacement from center of the colliding sphere
    r1 = [(xpos-spheresX[c]),(ypos-spheresY[c])] #sets up displace
    r1[0] /= sqrt((ypos-spheresY[c])**2+(xpos-spheresX[c])**2) #normalize the vector
    r1[1] /= sqrt((ypos-spheresY[c])**2+(xpos-spheresX[c])**2)
    #this returns by using dot product properties cos(theta) for theta the angle between the
    edge of the sphere and the beetle's velocity vector or sin(phi) as the angle between the
    center of the sphere and the velocity vector.
    #note that phi=theta+pi/2, so we use sin
    sinPhi = fabs(r1[0]*grad[0]+r1[1]*grad[1])/(sqrt(r1[0]**2+r1[1]**2)*sqrt(grad[0]**2+grad[1]**
    2))
    r1[0] *= sinPhi
    r1[1] *= sinPhi
    vmag = (0*(sphereMass(c)-mass)+2*mass*beetleVelocity)/(sphereMass(c)+mass) #calculate sphere
    velocity from the elastic collision
    xi = spheresX[c]/sandSize
```

```
    yi = spheresY[c]/sandSize
    xiOld = spheresX[c]/sandSize #set up initial position to calculate delta(h)/delta(x)
    yiOld = spheresY[c]/sandSize
    #MOVE SPHERES
    #approximate spheres rolling over semi-compressible solid as a sphere sliding over
    incompressible surface.
    #print "Sphere ", c, " moved from ", xi, ", ", yi
    while vmag > 0:
      xi += vmag*r1[0]
      yi += vmag*r1[1]
      xi = max(xi,0+bufferSize) #constrain x and y to within the box
      xi = min(xi,boxSize/sandSize-bufferSize)
      yi = max(yi,0+bufferSize)
      yi = min(yi,boxSize/sandSize-bufferSize)
      a = 1*g*(mu+(sandLevel[int(xi)][int(yi)]-sandLevel[int(xiOld)][int(yiOld)])) #calculated
      deceleration value.  see paper for details.
      vmag -= a #decelerate vmag
      xiOld += vmag*r1[0]
      yiOld += vmag*r1[1]
      xiOld = max(xiOld,0+bufferSize) #constrain xOld and yOld to within the box
      xiOld = min(xiOld,boxSize/sandSize-bufferSize)
      yiOld = max(yiOld,0+bufferSize)
      yiOld = min(yiOld,boxSize/sandSize-bufferSize)
      depressSand(xiOld,yiOld,spheresR[c],sphereMass(c)) #depresses the sand from the glass
      balls rolling around.
    spheresX[c] = xiOld
    spheresY[c] = yiOld
    #print "To ",xiOld,", ",yiOld
    if drawMode:
      sphere = Circle(Point(spheresX[c],spheresY[c]), spheresR[c])
      sphere.draw(win)

def closestFood(x,y):
  """closestFood(xposition, yposition): returns a vector [i,j] for an index of the closest food
  particle to the beetle's current position"""
  smallest = 999
  smallestIndex = [0,0]
  for j in xrange(int(boxSize*foodDensity)):
    for i in xrange(int(boxSize*foodDensity)):
      if foodPresent[i][j] == 1:
        if sqrt((x-foodX[i][j])**2+(y-foodY[i][j])**2) < smallest:
          smallest = sqrt((x-foodX[i][j])**2+(y-foodY[i][j])**2)
          smallestIndex = [i,j]
  return smallestIndex

def eat(x,y):
  """eat(beetle's x position, beetle's y position):
    Returns a sum representing the sum of all of the elements in foodPresent.  If the sum is 0,
    all food has been eaten.
    Also, checks to see if the beetle is very close to the nearest food element.  If it within
    eatDistance, it "eats" it."""
  global foodPresent
  global checkSum
  closestIndex = closestFood(x,y)
```

```
  xindex = closestIndex[0]
  yindex = closestIndex[1]
  #use different method to find xindex, yindex
  xi = foodX[xindex][yindex]
  yi = foodY[xindex][yindex]
  #print "Distance from food ",[xindex],",",[yindex],": ", sqrt((x-xi)**2+(y-yi)**2)
  if sqrt((x-xi)**2+(y-yi)**2) < eatDistance and foodPresent[xindex][yindex] == 1:
    foodPresent[xindex][yindex] = 0
    checkSum = sum(foodPresent[i][j] for i in xrange(int(boxSize*foodDensity)) for j in xrange(int(
    boxSize*foodDensity)))

def addSand():
  """Adds a layer of sand and food at the end of each iteration."""
  global sandLevel
  for j in xrange(int(ceil(boxSize/sandSize+1))):
    for i in xrange(int(ceil(boxSize/sandSize+1))):
      sandLevel[i][j] += sandAdded
  for j in xrange(int(ceil(boxSize/sandSize+1))):
    y = j #increments y by 1/foodDensity to evenly distribute space
    for i in xrange(int(ceil(boxSize/sandSize+1))):
      x = i #increments x
      overlapIndex = overlapTest(x,y,0)
      if overlapIndex != -1:
        coords = sphereFall(x,y,overlapIndex)
        x = max(min(coords[0],int(ceil(boxSize/sandSize+1)-1)),0)
        y = max(min(coords[1],int(ceil(boxSize/sandSize+1)-1)),0)
      sandLevel[int(x)][int(y)] += sandAdded
      y = j
  print "Sand layering complete."

def addSand2():
  """Adds a layer of sand without running a collision test with spheres"""
  global sandLevel
  for j in xrange(int(ceil(boxSize/sandSize+1))):
    for i in xrange(int(ceil(boxSize/sandSize+1))):
      sandLevel[i][j] += sandAdded

def writeSpheresDepth():
  global spheresDepth
  global spheresDepthPerIteration
  for i in xrange(numSpheres):
    xpos = int(round(spheresX[i]))
    ypos = int(round(spheresY[i]))
    spheresDepth[i] = sandLevel[xpos][ypos]
  spheresDepthPerIteration.append(copy.deepcopy(spheresDepth))

def writeSandData():
  """writes sand elevation data to a file"""
  print "Compressing sandLevel() array..."
  condensedSandLevel=[[0 for i in xrange(int(ceil(boxSize/sandSize+1))/mathematicaExportResolution
  )] for j in xrange(int(ceil(boxSize/sandSize+1))/mathematicaExportResolution)]
  for j in xrange(int(ceil(boxSize/sandSize+1))/mathematicaExportResolution):
    for i in xrange(int(ceil(boxSize/sandSize+1))/mathematicaExportResolution):
      condensedSandLevel[i][j] = sandLevel[i*mathematicaExportResolution][j*
```

```
      mathematicaExportResolution]
  #write sand elevation data
  filename = "Sand_Elevation_"+str(instanceNumber)+"_"+str(iteration)+".dat"
  filehandle = open(filename, "w")
  print "Writing elevation data to file..."
  filehandle.write(str(condensedSandLevel).replace("[","{").replace("]","}"))
  filehandle.close()
  #write sphere height data
  filename = "Final_Sphere_Heights_"+str(instanceNumber)+".dat"
  filehandle = open(filename, "w")
  print "Writing height data to file..."
  filehandle.write(str(spheresDepth).replace("[","{").replace("]","}"))
  filehandle.write("\n")
  filehandle.write("\n")
  filehandle.write(str(spheresR).replace("[","{").replace("]","}"))
  filehandle.close()
  #write all sphere height data for each iteration
  filename = "All_Sphere_Heights_"+str(instanceNumber)+".dat"
  filehandle = open(filename, "w")
  print "Writing height data over time to file..."
  filehandle.write(str(spheresDepthPerIteration).replace("[","{").replace("]","}"))
  filehandle.write("\n")
  filehandle.write("\n")
  filehandle.write(str(spheresR).replace("[","{").replace("]","}"))
  filehandle.close()

def returnFinalData():
  returnString = ""
  for i in xrange(numSpheres-1):
    returnString += "{"+str(spheresR[i])+","+str(spheresDepth[i])+"}, "
  returnString += "{"+str(spheresR[numSpheres-1])+","+str(spheresDepth[numSpheres-1])+"}"
  return returnString

def returnFinalDataOverTime():
  returnString = ""
  for i in xrange(numSpheres):
    for j in xrange(numIterations):
      #if j != numIterations - 1 and i != numSpheres - 1:
      returnString += "{"+str(j)+","+str(spheresR[i])+","+str(spheresDepthPerIteration[j][i])+
      "}, "
      #else:
        #returnString +=
        "{"+str(numIterations-1)+","+str(spheresR[numSpheres-1])+","+str(spheresDepthPerIteration
        [numIterations-1][numSpheres-1])+"}"
  return returnString

##########################################################################################################
##########################################################################################

#Module to run the program multiple times.
numSimulations = 1
#Disable this command to run directly from computer
programID = 0
#programID = int(sys.argv[1]) #For running multiple programs at the same time, this ensures the
```

```
composite files are not overwriting each other.  argv[] allows for it to easily be edited via
command line
finalData = ""#"{"
finalDataTime = ""#"{"
for instanceNumber in xrange(numSimulations):
  print "##############  INSTANCE ", instanceNumber, " of ",numSimulations-1," STARTED.
  ##########################"
  #Variable delcarations
  g = 9.81 #gravity in box
  mu = .4 #coefficient of kinetic friction for glass on glass, used to approximate glass on sand
  due to similar molecular structure.
  flowRate = 79 #used to easily adjust sandViscosity
  sandViscosity = 2.3*78/(flowRate-78)
  bufferSize = 50 #distance from edges the spheres can get.  prevents them from getting stuck in
  the corners.
  sphereBufferSize = 0 #closest distance one sphere can be from another.  prevents interactions
  between multiple spheres
  numSpheres = 40 #Number of spheres to generate
  iterationsPerSecond = 10 #number of iterations that equate to 1 second of time passing
  numIterations = 1 #Number of iterations to run the "level" through before recording final data.
  sandAdded = 5 #layers of sand that are added to the grid over each iteration.
  sandSize = 1 #size of sand grain in mm.  (!) Very small sandSize can overload memory.
  mathematicaExportResolution = 5 #takes every nth element for rows and columns of sandLevel()
  to keep the output manageable
  boxSize = 1000 #Edge size of square box in mm
  foodDensity = .02 #Food is scattered evenly at every lattice point on the grid in mm.
  foodDensity = pieces of food/linear mm
  beetleSize = 10 #beetle radius in mm (beetle is approximated as a circle)
  beetleMass = 30 #beetle mass in grams
  glassDensity = .0026 #glass density in grams/mm^3
  sandDensity = .001261 #sand density in grams/mm^3
  minSphereSize = sandSize #minimum radius of sphere in mm
  maxSphereSize = 200 #maximum radius of sphere in mm
  proximityValue = 99999999999 #when calculating the gradient field, uses the nearest x number
  of food points in all directions to avoid computational overload for very large amounts of food
  eatDistance = 1.0 + beetleSize #number of mm the beetle must be from food to eat it
  beetleVelocity = 2.0 #beetle's velocity in mm/iteration.  scales the unit vector that
  gradField() returns.
  outerInnerRatio = sqrt(2) #in the primitive model of footprints, describes the ratio of the
  outer box edge length to the inner box edge length.  conventionally sqrt(2) to make sand
  troughs and peaks even in magnitude of deviation.
  #Array declarations
  spheresR = [] #sphere radius array
  spheresX = [] #x positions
  spheresY = [] #y positions
  spheresDepth = [] #depth beneath the original sand level.  spheres can sink by repeatedly
  being pushed into the ground by rolling around in place, and can rise by being pusnhed up a
  "hill" of sand.
  spheresDepthPerIteration = [] #array to keep a permanent log of the spheresDepth over time
  sandLevel = [[0 for i in xrange(int(ceil(boxSize/sandSize+1)))] for j in xrange(int(ceil(boxSize/
  sandSize+1)))] #initialize an array keeping track of relative heights of columns of sand.
  ceil is used to prevent indexing errors.
  #the next two lines set up a 2d array of indices to keep track of the food particles and
  their x and y values
```

```
  #generally, the x and y positions will equal the respective i and j indices
  (x=i/foodDensity), but if it falls on the spheres, it will slide off

  #set up r, x, and y to be global so you can run overlapTest on them.
  #these will first be used for sphere placement and later for food placement
  #print "Variable declaration completed."

  #Places spheres of random sizes in the box
  placeSphere() #initial sphere placement
  #print "PlaceSphere()"
  for i in xrange(numSpheres):
    #Keeps placing the sphere randomly until it finds an unoccupied place to put it.
    #Note that this can cause an infinite loop if for very large numSpheres or very small boxSize
    while overlapTest(x,y,r+sphereBufferSize) != -1:
      placeSphere()
      #print "placeSphere() ",i
    #write to arrays
    spheresR.append(r)
    spheresX.append(x)
    spheresY.append(y)
    spheresDepth.append(0)
  #print "Sphere placement completed"

  #Main
  module##################################################################################################
  ########################################################
  for iteration in xrange(numIterations):
    print "Starting iteration ",iteration," of ", numIterations - 1,"."
    foodX = [[0 for i in xrange(int(boxSize*foodDensity))] for j in xrange(int(boxSize*foodDensity))]
     #foodX[xindex][yindex]
    foodY = [[0 for i in xrange(int(boxSize*foodDensity))] for j in xrange(int(boxSize*foodDensity))]
     #foodY[xindex][yindex]
    foodPresent = [[1 for i in xrange(int(boxSize*foodDensity))] for j in xrange(int(boxSize*
    foodDensity))] #determines whether the beetle has eaten the food yet. 1 is present, 0 is
    eaten.
    checkSum = sum(foodPresent[i][j] for i in xrange(int(boxSize*foodDensity)) for j in xrange(int(
    boxSize*foodDensity))) #sums up all elements of foodPresent
    initialCheckSum = checkSum
    checkSumCounter = 1
    #Add sand
    print "Adding sand... May take a while."
    addSand2()
    #Evenly place food
    print "Beginning simulation..."
    for j in xrange(int((boxSize*foodDensity))):
      y = j/foodDensity #increments y by 1/foodDensity to evenly distribute space
      for i in xrange(int((boxSize*foodDensity))):
        x = i/foodDensity #increments x
        overlapIndex = overlapTest(x,y,0)
        if overlapIndex != -1:
          coords = sphereFall(x,y,overlapIndex)
          x = coords[0]
          y = coords[1]
        foodX[i][j] = x
```

```
        foodY[i][j] = y
        y = j/foodDensity
    #print "Food placement completed"

    #Draw the map to visually check things are working.  Disabled during long computational runs.
    if drawMode:
      win = drawMap()

    # Initialize beetleX and beetleY
    beetleX = random.uniform(beetleSize, boxSize-beetleSize)
    beetleY = random.uniform(beetleSize, boxSize-beetleSize)

    #Place beetle randomly on any position that is not on a sphere
    while overlapTest(beetleX,beetleY,beetleSize) != -1:
      location = placeSphere()
      beetleX = location[0]
      beetleY = location[1]

    #Animates the beetle crawling around in the sand.
    while checkSum != 0:
      #draws the beetle on the map.  Disabled during computational runs.
      #print beetleX,", ",beetleY
      global checkSumCounter
      if checkSum == int(initialCheckSum-initialCheckSum/10*checkSumCounter):
        print "Iteration ",10*checkSumCounter," percent complete."
        checkSumCounter += 1
      grad = moveBeetle(beetleX, beetleY)
      depressSand(beetleX, beetleY, beetleSize, beetleMass)
      moveSphere(beetleX, beetleY, beetleMass, grad)
      eat(beetleX, beetleY)
      if drawMode:
        beetleDot = Point(beetleX, beetleY)
        beetleDot.setFill("blue")
        beetleDot.draw(win)

    if drawMode:
      win.close()

    writeSpheresDepth()
    writeSandData()
    print "Iteration ",iteration," of ", numIterations - 1, " completed."

#endTime = time.ctime()
#print "Simulation finished.  Started at ", startTime, " and finished at ", endTime

sandCheckSum = 0
for i in xrange(int(ceil(boxSize/sandSize+1))):
  sandCheckSum += sum(sandLevel[i])
print "sandCheckSum: ",sandCheckSum

print "Returning final data..."
instanceFinalData = str(returnFinalData())
print "Retunring final data over time..."
instanceFinalDataOverTime = str(returnFinalDataOverTime())
```

```
  print "##############  INSTANCE ", instanceNumber, " COMPLETED.  ##########################"
#################################################################################################################
#####
  #append the strings to the Mathematica output
  finalData += instanceFinalData
  #if instanceNumber != numSimulations - 1:
  finalData += ", "
  #else:
  #    finalData += "}"
  finalDataTime += instanceFinalDataOverTime
  #if instanceNumber != numSimulations - 1:
  finalDataTime += ", "
  #else:
  #    finalDataTime += "}"

filename = "Final_Sphere_Heights_Composite_"+str(programID)+".dat"
filehandle = open(filename, "w")
print "Writing composite height data to file..."
filehandle.write(finalData)
filehandle.close()

filename = "All_Sphere_Heights_Composite_"+str(programID)+".dat"
filehandle = open(filename, "w")
print "Writing composite height data over time to file..."
filehandle.write(finalDataTime)
filehandle.close()

endTime = time.ctime()
print "\n"
print "(!) Simulation complete. ",numSimulations," simulations and ", numSimulations*
numIterations," iterations performed."
print "Start time: ",startTime
print "End time: ",endTime
raw_input("Press enter to close this terminal window. ")
```