```
#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. ")
```