Ay 190 Assignment 10

N-body Oct Trees

February 25, 2013

1 Oct-tree Implementation

Oct-trees were implemented in the module `octtree.py`, which contains the `OctTree` class and related functions. A root `OctTree` is initialized by calling `create_root`, which creates a 3D cube centered on the origin large enough to enclose the particle furthest from the origin. A `OctTree` can be recursively subdivided into eight child oct-trees by calling `subdivide`. The total mass and center of mass is calculated for each `OctTree` upon initialization.

The `octtree` module requires the `particles` module (see Appendix A), which defines the `Particles` class (a `Particle` object stores a particle's mass, position, and velocity). The entirety of the `octtree` module is reproduced below.

```
# octtree.py
# created 2/20/13 by stacykim (modified 2D quadtree.py)
#
# Implements a 3D oct-tree containing particles.


import sys
import math
from numpy import *
from particles import *


class OctTree(object):
    """
    A rectangular oct-tree subdivided into cells each with 0 or 1 particles.
    To set up a subdivided oct-tree, call
        root.create_root(particles)
        root.subdivide()
    where particles is an array of Particles. The OctTree and its particles can be
    printed (to separate files) by calling
        root.print_tree().
    """

    def __init__(self, xrange, yrange, zrange, particles):
        """
        Initialize a cubical cell that covers the given range and contains the
        given particles.
```

```
    """
    self.xrange=array(xrange)
    self.yrange=array(yrange)
    self.zrange=array(zrange)
    self.size=xrange[1]-xrange[0]        # length of each side of cell
    self.m=sum([p.m for p in particles]) # total mass contained in the cell
    # calculate center of mass
    self.com=reduce(lambda com, p: com+p.m*p.r, particles,array([0,0,0]))/self.m if self.m!=0 else a
    self.children=particles  # either a Particle or an OctTree


def subdivide(self):
    """
    Divides the given oct-tree into 8 cells.  Assumes that oct.children
    is an array of Particles, which is reassigned to an array of OctTrees.
    """
    if len(self.children)<2: return

    # Calculate ranges for each cell
    x0,x2=self.xrange
    x1=(x2+x0)/2.

    y0,y2=self.yrange
    y1=(y2+y0)/2.

    z0,z2=self.zrange
    z1=(z2+z0)/2.

    # Assign particles to appropriate cell
    particles_sub=[[],[],[],[],[],[],[],[]]

    for p in self.children:
        # cells are numbered in the same order as in the 2D Cartesian system,
        # first for [z1,z2] then [z0,z1]
        if   (x1 <= p.r[0] <= x2) and (y1 <= p.r[1] <= y2) and (z1 <= p.r[2] <= z2): particles_sub[0
        elif (x0 <= p.r[0] <= x1) and (y1 <= p.r[1] <= y2) and (z1 <= p.r[2] <= z2): particles_sub[1
        elif (x0 <= p.r[0] <= x1) and (y0 <= p.r[1] <= y1) and (z1 <= p.r[2] <= z2): particles_sub[2
        elif (x1 <= p.r[0] <= x2) and (y0 <= p.r[1] <= y1) and (z1 <= p.r[2] <= z2): particles_sub[3
        elif (x1 <= p.r[0] <= x2) and (y1 <= p.r[1] <= y2) and (z0 <= p.r[2] <= z1): particles_sub[4
        elif (x0 <= p.r[0] <= x1) and (y1 <= p.r[1] <= y2) and (z0 <= p.r[2] <= z1): particles_sub[5
        elif (x0 <= p.r[0] <= x1) and (y0 <= p.r[1] <= y1) and (z0 <= p.r[2] <= z1): particles_sub[6
        elif (x1 <= p.r[0] <= x2) and (y0 <= p.r[1] <= y1) and (z0 <= p.r[2] <= z1): particles_sub[7
        else:
            print 'Found particle outside cell:\n p.r={0}\n xrange={1}\n yrange={2}\n zrange={3}\n'
                  ''.format(p.r,[x0,x1,x2],[y0,y1,y2],[z0,z1,z2])
            print len(self.children)
            for i in range(20): print self.children[i].r
            sys.exit()

    # Create 8 sub-OctTrees
    self.children=[OctTree([x1,x2],[y1,y2],[z1,z2],particles_sub[0]),
                   OctTree([x0,x1],[y1,y2],[z1,z2],particles_sub[1]),
                   OctTree([x0,x1],[y0,y1],[z1,z2],particles_sub[2]),
                   OctTree([x1,x2],[y0,y1],[z1,z2],particles_sub[3]),
```

```
                        OctTree([x1,x2],[y1,y2],[z0,z1],particles_sub[4]),
                        OctTree([x0,x1],[y1,y2],[z0,z1],particles_sub[5]),
                        OctTree([x0,x1],[y0,y1],[z0,z1],particles_sub[6]),
                        OctTree([x1,x2],[y0,y1],[z0,z1],particles_sub[7])]

        # Recursively divide sub-OctTrees
        for child in self.children: child.subdivide()


    def print_tree(self, fqn, fpn):
        # Open oct-tree output file
        fq=open(fqn, 'w')
        fq.write(fqn+'\n')

        # Open particles output file
        fp=open(fpn, 'w')
        fp.write(fpn+'\n')

        self.print_oct(fq, fp)

        fq.close()
        fp.close()


    def print_oct(self, fq, fp):
        """
        Prints the edges of the given OctTree and its sub-OctTrees to the given
        output file and the particles in the OctTrees to the given particle output
        file.
        """
        s=''
        for x in self.xrange:
            for y in self.yrange:
                for z in self.yrange:
                    s+='{0} {1} {2} '.format(x,y,z)
        fq.write(s)

        for oct in self.children:
            if type(oct)==OctTree:   oct.print_oct(fq,fp)
            elif type(oct)==Particle: fp.write('{0} {1} {2}\n'.format(oct.r[0],oct.r[1],oct.r[2]))


def create_root(particles):
    """
    Creates a root OctTree with spatial ranges large enough to contain the
    furthest particle.
    """
    # Find spatial range of particles
    range=0
    for p in particles:
        m=max(abs(p.r))
        if m > range: range=m

    range += 1e-4  # so no particles exactly on border
```

```
return OctTree([-range, range], [-range, range], [-range,range], particles)
```

2 Force computation

The Barnes-Hut algorithm was implemented in the Python module `nbodyBH` (see Appendix B). This module contains routines to initialize a particle distribution, calculate the Barnes-Hut approximation of the gravitational force on each particle, and evolve an array of particles forward in time.

The gravitational force on the particle computed by direct calculation of all 2-body interactions and by the Barnes-Hut algorithm was compared for the first 100 particles in the above particle distribution. An opening angle of at least about 0.1 was required for the runtimes of the Barnes-Hut algorithm to compete with direct computation. A loss of about 1% accuracy was required for a factor of 10 improvement in runtime.

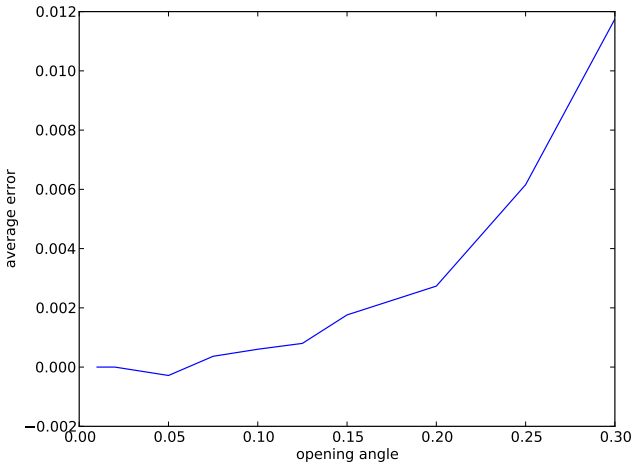

Figure 1: Average error of the Barnes-Hut algorithm for various opening angles in computing the gravitational force on 100 particles. The norm of the forces computed using the Barnes-Hut algorithm and direct computation was compared.

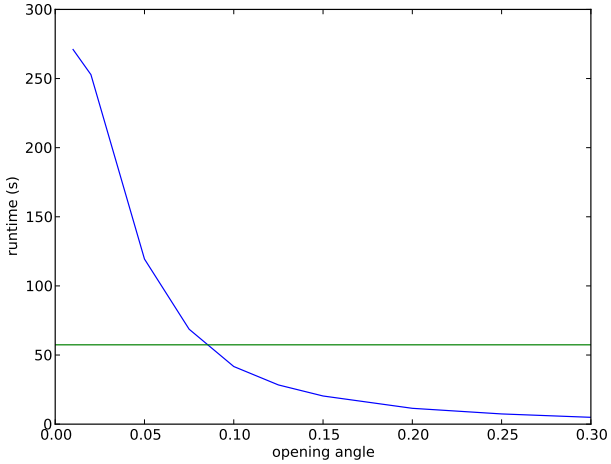

Figure 2: The runtime of the Barnes-Hut algorithm for various opening angles (blue). In comparison, the time it takes to compute forces directly is shown in green. An opening angle of at least about 0.1 is required for the Barnes-Hut algorithm to produce better runtimes.

Appendices

The Python modules and scripts used in this assignment include

A The Particle class

```
# particles.py
# created 2/27/12 by stacykim (cut class from nbody.py)
#
# Implements a particle with mass, position, and velocity.
#
#
# modified 3/4/12: cast x, v as Vectors upon initialization
# modified 2/10/13:
#     switched support of vector arith. from homegrown Vector class to numpy


from numpy import *


class Particle(object):
    """A particle with mass m, position r and velocity v."""

    def __init__(self, m, r, v):
        #r=[x,y,z], v=[vx,vy,vz]
        self.m=m
        self.r=array(r)
        self.v=array(v)
```

B The Barnes-Hut Algorithm

```
# nbodyBH.py
# created 2/27/12 by stacykim (imported body of nobody.py)
#
# Computes a gravitational N-body simulation using the Barnes-Hut algorithm
# with force softening.  The position of each particle is calculated using
# the sympletic Euler method.
#
#
# modified 2/27/12: exported Particle class to particles.py
# modified 3/4/12:  implemented Barnes-Hut algorithm
# modified 3/6/12
#    added output of total energy at every timestep
#    added function that calculates norm of a given vector, norm()
```

```
# modified 2/20/13: modified to only support BH algorithm (i.e. removed script)


import sys
import math
from numpy import *
from random import random
from time import time
from particles import *
from octtree import *

G=1               # gravitational constant

# BARNES-HUT ALGORITHM ------------------------------------------------------

depth=0
def gravi_nbodyBH(r, p, oct, opening_angle,a):
    """
    Equations of motion for the Particle p at updated position r under
    the gravitational forces from the particles in the OctTree oct.
    Recursively calculates only the velocity component.
    """
    #r=array([x,y,z])
    global depth

    depth+=1
    # Check if particle inside given cell
    inside=((oct.xrange[0] <= p.r[0] <= oct.xrange[1]) and
            (oct.yrange[0] <= p.r[1] <= oct.yrange[1]) and
            (oct.zrange[0] <= p.r[2] <= oct.zrange[1]))

    # If no particle inside, then no force from this cell
    if len(oct.children) == 0:
        return array([0,0,0])

    # If one particle, then calculate exact 2-particle force
    elif len(oct.children) == 1:
        if not inside: return two_body(r,oct.children[0],a)
        else:          return array([0,0,0])

    # If more than one particle, det if approx using COM or not
    elif len(oct.children) > 1:
        # Calculate proper com (i.e. if particle in cell, recalculate com without it)
        if not inside: com=oct.com
        else:          com=oct.m*(oct.com-p.m*p.r)/(oct.m-p.m)#oct.com-p.m*p.r/oct.m

        d=math.pow((r[0]-com[0])**2+(r[1]-com[1])**2+(r[2]-com[2])**2,0.5)
        if oct.size/d < opening_angle:
            return two_body(r,Particle(oct.m,com,[0,0,0]),a)
        else:
            accel = array([0,0,0])
            for cell in oct.children:
                accel = accel + gravi_nbodyBH(r,p,cell,opening_angle,a)
            return accel
```

```
def two_body(r,p,a):
    """
    Returns the force-softened gravitational acceleration imparted by
    Particle p on another particle at position x.
    """
    dist=(r[0]-p.r[0])**2+(r[1]-p.r[1])**2+(r[2]-p.r[2])**2+a**2
    ax=-G*p.m*(r[0]-p.r[0])/dist**1.5
    ay=-G*p.m*(r[1]-p.r[1])/dist**1.5
    az=-G*p.m*(r[2]-p.r[2])/dist**1.5

    return array([ax,ay,az])


def norm(x):
    """Returns the norm of the given 3-vector."""
    return math.sqrt(x[0]*x[0]+x[1]*x[1]+x[2]*x[2])


def init_particles_from_file(fn,N):
    """
    Initializes and returns an array of the first N particles from the file with
    name fn.  The file can either specify mass, position, and velocity of each
    particle (one per line) or just the positions (in which case m=1 and v=[0,0,0]).
    """
    particles=[[]]*N
    # Read inputs from input file if given
    f=open(fn,'rU')
    f.readline()  # Skip first 3 lines of input file
    f.readline()

    # Read in initial conditions for first N particles given in file
    for i in range(N):
        line=f.readline()[:-1].rsplit(' ')
        for j in range(line.count('')): line.remove('')

        if len(line)==7:
            params = array([float(line[j]) for j in range(7)])
        elif len(line)==3: # assume only positions, set m=1,v=[0,0,0]
            params = array([1.]+[float(line[j]) for j in range(3)]+[0.,0.,0.])
        else:
            print '{0}: Unrecognized line in input file:{1}'.format(i,line)
            sys.exit()

        particles[i]=Particle(params[0],params[1:4],params[4:])

    f.close()
    return particles


def init_particles_random(r_range,v_range,dim,N):
    """
    Initializes and returns an array of N particles each with a random position in
```

```
    a circle (dim==2) or sphere (dim==3) of radius r_range centered on the origin
    and with random velocities in the range [-v_range, v_range].  The array of
    particles is outputted in the file partiles<N>.dat.
    """
    fn='particles{0}.dat'.format(N)
    f=open(fn,'w')
    f.write('{1}\n{0:>19}'.format('m',fn))
    for i in range(3): f.write('r{0}'.format(i).rjust(19))
    for i in range(3): f.write('v{0}'.format(i).rjust(19))
    f.write('\n')

    particles=[[]]*N
    for i in range(N):
        mass = m if m != 0 else random()

        r=rrange*random()
        if dim==2:
            th=2*math.pi*random()
            x=[r*math.cos(th),r*math.sin(th),0]
        if dim==3:
            th=math.pi*random()
            ph=2*math.pi*random()
            x=[r*math.sin(th)*math.cos(ph),r*math.sin(th)*math.sin(ph),r*math.cos(th)]

        v=[2*vrange*random()-vrange for j in range(3)]
        if dim==2: v[2]=0

        particles[i]=Particle(mass,array(x),array(v))

        f.write('{0!s:>19}'.format(m))
        for j in range(3): f.write('{0!s:>19}'.format(x[j]))
        for j in range(3): f.write('{0!s:>19}'.format(v[j]))
        f.write('\n')

    f.close()
    return particles


def update_particle(p,root,h):
    """
    Calculate gravitational force on particle p from the other particles in the
    OctTree root using the symplectic Euler method and a time step of h. Returns
    the updated position and velocity of the particle.
    """
    r=p.r+h*p.v
    v=p.v+h*gravi_nbodyBH(r,p,root)
    return r,v


def update(N,particles,h):
    """
    Evolves the N particles forward by a time step of h using the symplectic Euler
    method to integrate forward in time and the Barnes-Hut algorithm.
    """
```

```
    particles0=particles[:] # make a copy of the unevolved system
    root=create_root(particles0)
    root.subdivide()

    for i in range(N):
        p=particles0[i]
        particles[i].r,particles[i].v=update_particle(p,root,h)
```

C Script for set 10

```
# set10.py
# created 2/20/13 by stacy kim

from numpy import *
from nbodyBH import *
from matplotlib.pyplot import *
import sys

iter=0

def gravi_nbody(r,p1,Npm1):
    """
    Equations of motion for particle p1 under the gravitational forces from
    the other Npm1 particles.  Calculates only the velocity component.
    """
    #x=array([x,y,z])

    xsum,ysum,zsum=0,0,0
    for p2 in Npm1:
        dist=(r[0]-p2.r[0])**2+(r[1]-p2.r[1])**2+(r[2]-p2.r[2])**2+a**2
        xsum+=p2.m*(r[0]-p2.r[0])/dist**1.5
        ysum+=p2.m*(r[1]-p2.r[1])/dist**1.5
        zsum+=p2.m*(r[2]-p2.r[2])/dist**1.5

    return -G*array([xsum,ysum,zsum])


# MAIN -------------------------------------------------------------------------

a=0         # force-softening constant
ntest=100   # number of particles to compare direct and BH methods

fn='HW5_Data.txt'
N=len(open(fn,'rU').read().split('\n')[1:])-2


# Initialize particles and oct-tree
particles=init_particles_from_file(fn,N)
root=create_root(particles)
root.subdivide()
print 'done subdividing'
```

```
# directly calculate force on particles
fg_direct=[]
start_time=time()
for i in range(ntest):
    p=particles[i]
    if i<5: print p.r
    fg_direct.append(gravi_nbody(p.r,p,particles[:i]+particles[i+1:]))
time_direct=time()-start_time
print 'took',time_direct,'s to compute forces directly'

np_fg_direct=array(fg_direct)
for i in range(5): print np_fg_direct[i],norm(np_fg_direct[i])


# calculate force via BH algorithm using various opening angles
opening_angles=[0.01,0.02,0.05,0.075,0.1,0.125,0.15,0.2,0.25,0.3]
noa=len(opening_angles)
times=zeros(noa)
err=zeros(noa)
for j in range(noa):
    opening_angle=opening_angles[j]
    start_time=time()
    fg_BH=[]

    for i in range(ntest):
        p=particles[i]
        fg_BH.append(gravi_nbodyBH(p.r,p,root,opening_angle,a))

    times[j]=time()-start_time
    np_fg_BH=array(fg_BH)
    for i in range(5): print np_fg_BH[i],norm(np_fg_BH[i])
    err[j]=sum([norm(np_fg_BH[i])/norm(np_fg_direct[i]) for i in range(ntest)])/ntest

    print 'oa={0}: {1} sec with average error of {2}'.format(opening_angle,times[j],err[j])


# algorithm performance comparisons
plot(opening_angles,times)
plot([0,0.3],[time_direct,time_direct])
xlabel('opening angle')
ylabel('runtime (s)')
savefig('runtimes.pdf')
show()

plot(opening_angles,err)
xlabel('opening angle')
ylabel('average accuracy')
savefig('accuracy.pdf')
show()

plot(opening_angles,[1-ee for ee in err])
xlabel('opening angle')
ylabel('average error')
```

```
savefig('errors.pdf')
show()
```