Ay 190 Assignment 9

Poisson Equation

March 7, 2013

1 Validation of Solvers

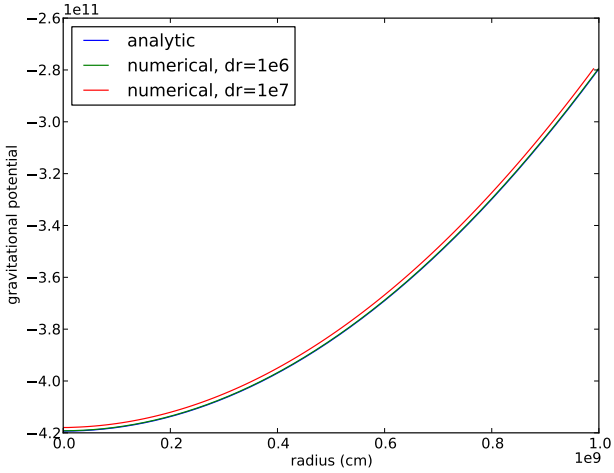

Figure 1: Density profile for a homogeneous sphere of $\rho = 1$ calculated using the direct ODE method using two different radial grids. The solution converges as $\mathcal{O}(h)$, as it should (the ODEs were integrated using the forward Euler method).

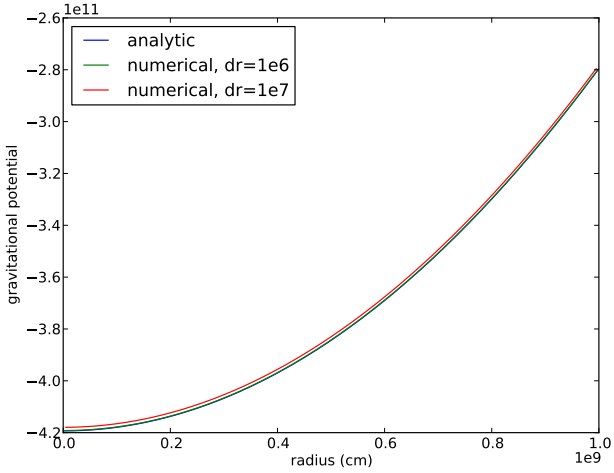

Figure 2: Density profile for a homogeneous sphere of $\rho = 1$ calculated using the matrix method.

2 Gravitational potential of the presupernova model

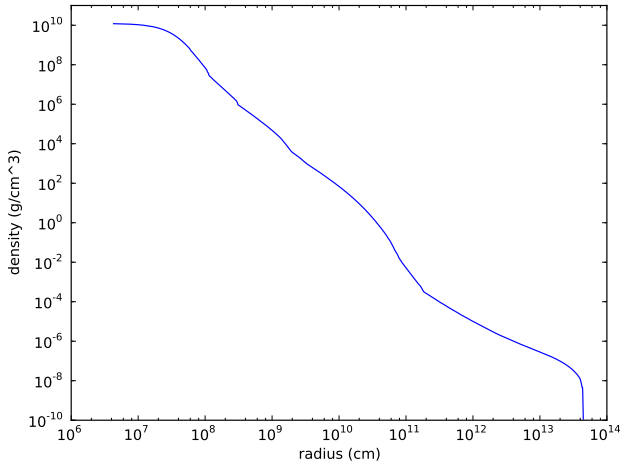

Figure 3: Density profile of the presupernova model. The density profile was interpolated onto the radial grids on which Poisson's equation was solved using the piecewise quadratic method.

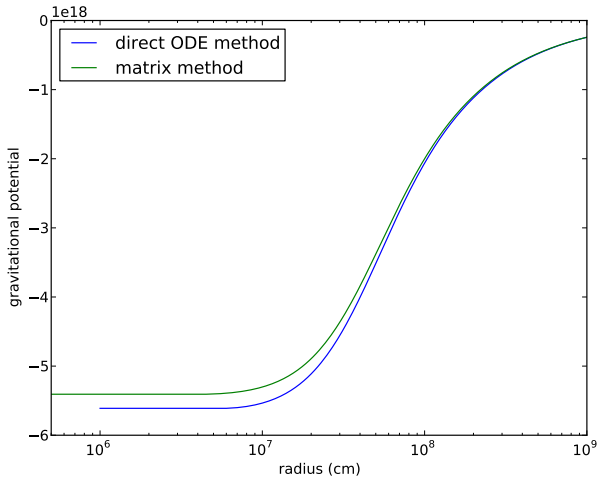

Figure 4: Density profile for the given presupernova model calculated using the matrix method. The solution calculated via the direct ODE and matrix methods. The solution was computed on a grid with a resolution of $\Delta r = 1 \times 10^6$ cm for both methods. The matrix method produces more accurate results (with errors lower by a factor of about 1-2). The matrix method is faster when $\Delta r \geq 10^7$, but the direct ODE method is faster when $\Delta r = 10^6$.

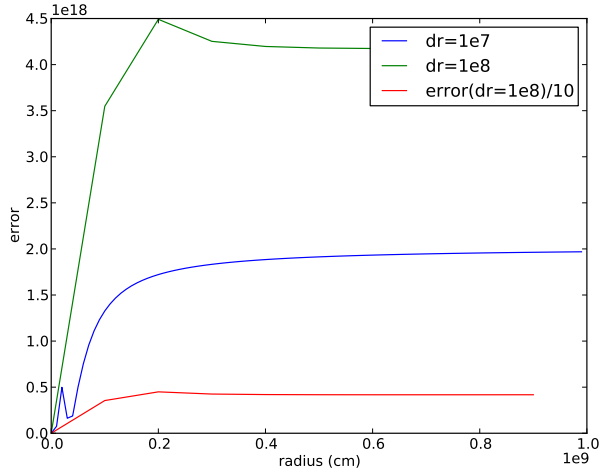

Figure 5: Error under the direct ODE method computed using grid resolutions of $\Delta r = 10^7$ and 10^8, compared against the solution computed with a resolution of $\Delta r = 10^6$. The offset added to the solution in order to match the outer boundary condition was subtracted before computing the error. The error does not decrease by a factor of 10 (shown by the red line).

Appendices

The Python modules and scripts used in this assignment include

A The Poisson Equation Solvers

```
# set9.py
# created 2/20/13 by stacy kim

import sys
from math import *
from numpy import *
from matplotlib.pyplot import *
from interpolate import *
from time import clock


data = array([[float(el) for el in line.split(' ')[1:] if el!='']
              for line in open('presupernova.dat','r').read().split('\n')[1:-1]])
r=data[:,1]
mass=data[:,0]
rho=data[:,3]

ROUT=1.0e9
G=6.67398e-8

"""
plot(r,rho)
xlabel('radius (cm)')
ylabel('density (g/cm^3)')
xscale('log')
yscale('log')
savefig('rho.pdf')
show()

# plot all over r
for i in range(len(data[0])-1):
    subplot(2,3,i+1)
    plot(data[:,1],data[:,i])
    xscale('log')
    if i != 4 and i!= 5: yscale('log')
savefig('all_columns_vs_r.pdf')
show()

# check if guesses to which columns are mass, radius, and rho are correct
dm=[4*pi/3*r[0]**3*rho[0]]+[4*pi/3*(r[i]**3-r[i-1]**3)*rho[i] for i in range(1,len(r))]
for mm in dm:
    if mm<0: print mm
mass=[sum(dm[:i]) for i in range(len(r))]
```

```
for i in range(30): print r[i],mass[i]

plot(r,data[:,0],'r')
plot(r,mass)
xscale('log')
yscale('log')
savefig('mass_check.pdf')
show()
"""

# THE 1D POISSION EQUATION SOLVERS ----------------------------------------------

def rhs(r,x,rho):
    """Implements the 1D Poisson equation."""
    # x = [phi,z]
    dphi=x[1]
    dz=4*pi*G*rho - 2*x[1]/r

    return array([dphi,dz])


def directODE(r,rho,dr,rout):
    """
    Solves Poisson's equation via the direct ODE method for the given density
    profile r, rho on a grid that extends from 0 to rout.  If a constant density
    profile is to be assumed, a constant rho can be given (r will be ignored).

    The solution and the grid it was computed on (r_grid,phi) is returned.
    """
    # construct the grid and calculate Mtot
    r_grid=arange(0.,rout,dr)
    nr=len(r_grid)
    try:
        len(rho)
        rho_grid=pw_quadratic(r,rho,r_grid)

        for ir in range(len(rho)):
            if r[ir] > ROUT: break
        M=4*pi/3*(rho[0]*r[0]**3 +
                  sum([rho[i]*(r[i]**3-r[i-1]**3) for i in range(1,ir)]))
    except TypeError:
        rho_grid=rho*ones(nr)
        M=4*pi/3*rout**3

    # solve Poisson's eqn
    t0=clock()
    x=[array([0,4*pi*G*rho_grid[0]])] # (phi,z) boundary conditions
    for i in range(1,nr):
        x.append(x[i-1]+dr*rhs(r_grid[i],x[i-1],rho_grid[i-1]))

    phi_offset=-G*M/rout-x[-1][0]
    phi=array([p+phi_offset for p,z in x])
    print 'dODE: took',clock()-t0,'sec'
```

```
    return r_grid,phi,phi_offset


def matrix_method(r,rho,dr,rout):
    """
    Solves Poisson's equation via the matrix method for the given density profile
    r, rho on a grid that extends from dr/2 to rout+dr/2 (shifted to avoid the
    singularity at r=0). If a constant density profile is to be assumed, a constant
    rho can be given (r will be ignored).

    The solution and the grid it was computed on (r_grid,phi) is returned.
    """
    # construct the grid and calculate Mtot
    r_grid=arange(0.,rout,dr)+0.5*dr
    nr=len(r_grid)
    try:
        len(rho)
        rho_grid=pw_quadratic(r,rho,r_grid)
        for ir in range(len(rho)):
            if r[ir] > ROUT: break
        M=4*pi/3 * (r[0]**3*rho[0] +
                    sum([(r[i]**3-r[i-1]**3)*rho[i] for i in range(1,ir)]))
    except TypeError:
        rho_grid=rho*ones(nr)
        M=4*pi/3*ROUT**3

    # construct the system of equations to be solved
    t0=clock()
    J=zeros([nr,nr])
    J[0,0]=-1./dr**2-1/(r_grid[0]*dr)
    J[0,1]=1./dr**2+1./(r_grid[0]*dr)
    for j in range(1,nr):
        J[j,j-1] = 1./dr**2-1./(r_grid[j]*dr)
        J[j,j]   = -2./dr**2
        if j!=(nr-1): J[j,j+1] = 1./dr**2+1./(r_grid[j]*dr)

    b=4*pi*G*array(rho_grid)

    # solve Poisson's eqn
    ymatrix=linalg.solve(J,b)

    phi_offset=-G*M/rout-ymatrix[-1]
    phi=array([p+phi_offset for p in ymatrix])
    print 'mm: took',clock()-t0,'sec'

    return r_grid,phi,phi_offset


# VALIDATION ------------------------------------------------------------------
# Solve Poisson's eqn for homogeneous sphere
rho=1
drr=[1e6,1e7]

analytic = lambda r,rho: 2./3*pi*G*rho*(r**2 - 3*ROUT**2)
```

```
r_ana=arange(0,ROUT,1e6)
phi_ana=array([analytic(r,rho) for r in r_ana])


# via the direct ODE method
plot(r_ana,phi_ana)

err=[]
for i in range(len(drr)):
    r_grid,phi = directODE(0,rho,drr[i],ROUT)
    plot(r_grid,phi)
    err.append([phi[i]-analytic(r_grid[i],rho) for i in range(len(phi))])

xlabel('radius (cm)')
ylabel('gravitational potential')
legend(['analytic','numerical, dr=1e6','numerical, dr=1e7','numerical, dr=1e8'],loc='best')
savefig('validation_directODE.pdf')
show()

plot(arange(0,ROUT,drr[0]),err[0],
     arange(0,ROUT,drr[1]),err[1],
     arange(0,ROUT,drr[1]),array(err[1])/10.)
xlabel('radius (cm)')
ylabel('error')
legend(['dr=1e6','dr=1e7','error(dr=1e7)/10'],loc='best')
show()


# via the matrix method
plot(r_ana,phi_ana)

err=[]
for i in range(len(drr)):
    r_grid,phi = matrix_method(0,rho,drr[i],ROUT)
    plot(r_grid,phi)
    err.append([phi[i]-analytic(r_grid[i],rho) for i in range(len(phi))])

xlabel('radius (cm)')
ylabel('gravitational potential')
legend(['analytic','numerical, dr=1e6','numerical, dr=1e7','numerical, dr=1e8'],loc='best')
savefig('validation_matrix.pdf')
show()

plot(arange(0,ROUT,drr[0])+drr[0]/2,err[0],
     arange(0,ROUT,drr[1])+drr[1]/2,err[1],
     arange(0,ROUT,drr[1])+drr[0]/2,array(err[1])/10.)
xlabel('radius (cm)')
ylabel('error')
legend(['dr=1e6','dr=1e7','error(dr=1e7)/10'],loc='best')
show()


# THE PRESUPERNOVA MODEL ----------------------------------------------------
# Solve Poisson's eqn for given density profile
```

```
r=data[:,1]
rho=data[:,3]
dr=1e6

r_directODE,phi_directODE,phi_off_dODE = directODE(r,rho,dr,ROUT) #direct ODE method
r_matrix,phi_matrix,phi_off_matrix = matrix_method(r,rho,dr,ROUT) # matrix method

plot(r_directODE,phi_directODE)
plot(r_matrix,phi_matrix)
xlabel('radius (cm)')
ylabel('gravitational potential')
xscale('log')
xlim([5e5,ROUT])
legend(['direct ODE method','matrix method'],loc='best')
savefig('comparison.pdf')
show()


# convergence tests
r_dODE=[r_directODE]
r_matrix=[r_matrix]
phi_dODE=[phi_directODE]
phi_matrix=[phi_matrix]
phi_off=[phi_off_dODE]

drr=[1e6,1e7,1e8]

plot(r_directODE,phi_directODE,'r')
plot(r_matrix,phi_matrix,'b')

for dr in drr[1:]:
    r_sol,phi,offset=directODE(r,rho,dr,ROUT)
    r_dODE.append(r_sol)
    phi_dODE.append(phi)
    phi_off.append(offset)
    plot(r_sol,phi,'r')

    r_sol,phi,offset=matrix_method(r,rho,dr,ROUT)
    r_matrix.append(r_sol)
    phi_matrix.append(phi)
    plot(r_sol,phi,'b')

show()

print phi_off

err_dODE=[[phi_dODE[i][j] - phi_dODE[0][where(r_dODE[0]==r_dODE[i][j])[0][0]]
           for j in range(len(r_dODE[i]))] for i in range(1,len(drr))]

err_dODE[0] = abs(array(err_dODE[0]) - phi_off[1] + phi_off[0])
err_dODE[1] = abs(array(err_dODE[1]) - phi_off[2] + phi_off[0])

plot(r_dODE[1],err_dODE[0],
     r_dODE[2],err_dODE[1],
```

```
     r_dODE[2],array(err_dODE[1])/10.)
xlabel('radius (cm)')
ylabel('error')
legend(['dr=1e7','dr=1e8','error(dr=1e8)/10'],loc='best')
savefig('dODE_preSN_convergence.pdf')
show()
```

B The Interpolation Routines

```
# interp.py
# created 1/14/13 by stacy kim
#
# Contains a suite of interpolation routines.
#
# NOTE: In all of the folowing routines, x is an array of values to interpolate the
#  discretized function given by arrays xdat, ydat.
#
# modified 3/7/13: pw_quadratic and pw_linear returned interpolated values in a
#     2D array of single-element arrays; should now return 1D array

import sys, math
import numpy as np
from operator import mul


def lagrange(xdat, ydat, x):
    """Lagrange interpolation."""
    xdat=np.array(xdat,dtype=np.float)
    ydat=np.array(ydat,dtype=np.float)

    return [sum([ydat[j]*reduce(mul,[(xx-xk)/(xdat[j]-xk) for xk in np.concatenate((xdat[:j],xdat[j+1:])
                for j in range(len(xdat))]) for xx in x]


def pw_linear(xdat, ydat, x):
    """Piecewise linear interpolation (for non-piecewise, supply 2 data pts)."""
    # sort data (x,y) pairs by xdat values
    a=np.array([xdat,ydat],dtype=np.float)
    dat=a.T[a.T[:,0].argsort()]
    xsrt,ysrt=dat[:,0],dat[:,1]

    # interpolate at given locations x
    y=[] # interpolated values
    for xx in x:
        for i in range(len(xsrt)):
            if xx < xsrt[i]: break
        i-=1
        y.append(lagrange(xsrt[i:i+2],ysrt[i:i+2],[xx])[0])
        #y.append(ysrt[i]+(ysrt[i+1]-ysrt[i])/(xsrt[i+1]-xsrt[i])*(xx-xsrt[i]))

    return y
```

```
def pw_quadratic(xdat, ydat, x):
    """Piecewise quadratic interpolation (for non-piecewise, supply 3 data pts)."""
    # sort data (x,y) pairs by xdat values
    a=np.array([xdat,ydat],dtype=np.float)
    dat=a.T[a.T[:,0].argsort()]
    xsrt,ysrt=dat[:,0],dat[:,1]

    # interpolate at given locations x
    y=[] # interpolated values
    for xx in x:
        for i in range(len(xsrt)-1):
            if xx< xsrt[i]: break
        i-= 1
        y.append(lagrange(xsrt[i:i+3],ysrt[i:i+3],[xx])[0])

    return y


def cubic_hermite(xdat, ydat, ypdat, x):
    """Piecewise cubic Hermite interpolation."""
    psi0 = lambda z: 2*z**3-3*z**2+1
    psi1 = lambda z: z**3-2*z**2+z

    # assumes xdat,ydat,ypdat are already sorted by xdat values
    xsrt,ysrt,ypsrt=xdat,ydat,ypdat

    # sort data (x,y,yp) triples by xdat values
    a=np.array([xdat,ydat,ypdat])
    dat=a.T[a.T[:,0].argsort()]
    xsrt,ysrt,ypsrt=dat[:,0],dat[:,1],dat[:,2]

    # interpolate at given locations x
    y=[] # interpolated values
    for xx in x:
        for i in range(len(xsrt)-1):
            if xx < xsrt[i]: break
        i-=1
        z=(xx-xsrt[i])/(xsrt[i+1]-xsrt[i])
        y.append(ysrt[i]*psi0(z)+ysrt[i+1]*psi0(1-z)
                 +ypsrt[i]*(xsrt[i+1]-xsrt[i])*psi1(z)
                 -ypsrt[i+1]*(xsrt[i+1]-xsrt[i])*psi1(1-z))

    return y
```