Computational Astrophysics
Hydrostatic Equilibrium and Angular Momentum
Monsoon 2022

1 Problem Summary

Solving for hydrostatic equilibrium of an astrophysical body with some angular momentum. Following system of coupled partial (nondimensionalized) differential equations needs to be solved

$$\text{Poisson: } \nabla^2\Phi = \rho \tag{1}$$

$$\text{Force: } \frac{\nabla P}{\rho} = -\alpha\nabla\Phi + \beta\nabla(r\sin\theta)^2 \tag{2}$$

$$\text{EOS: } P = \rho^2 \tag{3}$$

where $\beta << \alpha$ (3 orders of smaller magnitude in our case).

2 Code Listing[1]

Poisson equation is solved via a spectral scheme which utilizes Fast Fourier Transform. Jacobi method is used to solve the boundary value problem in P. Some basic machinery has been developed for prolongation and restriction of the grid to implement a multigrid optimization [2].

a. myParam.py (stores the parameters for global access)

```
# Boundary Conditions
R, rhoB = 10, 1          # Radius, Surface Density

# Physical Parameters
A, B = 1, 1e-3           # Gravitational Contribution, Centrifugal Contribution
```

b. poissonFFT.py (solved Poisson equation for gravitational potential)

```
from numpy import fft, zeros, cos, sin, pi, array #
from myParam import *

'''
Converts polar coordinates to cartesian coordinates (in 2D)
Input: Polar coordinate components (as a numpy array )
Output: Cartesian coordinate components (as a numpy array)
'''
def pol2cart(k):
    x = k[0] * cos(k[1])
    y = k[0] * sin(k[1])

    return array([x, y])

'''
Solves Poisson Equation on a 2 dimensional J x L grid using Fast Fourier Transform
Input  : Rho (complex array), x-span (int), y-span (int)
Output : Phi array J x L (dtype = complex)
'''

def PoissonSolveFFT2D(rho, J, L):

    delta = (2.4 * R)/len(rho)
    rhoHat = fft.fft2(rho)
    phiHat = zeros((J, L), dtype=object)

    for m in range(J):
        for n in range(L):
            denom = 2 * (cos(2 * pi * m/J) + cos(2 * pi * n/L) - 2)
            if denom != 0:
                phiHat[m,n] = (rhoHat[m,n] * delta**2) / denom

    phi = fft.ifft2(phiHat)

    return phi

```

[1] Please note that this mini project has not been stress-tested and might not converge. Do not use for an scientific grade computations
[2] A complete implementation of V-cycle or W-cycle with exact matrix inversion at coarsest level is still in development

c. smoothening.py (for a minimalist use-case, this is the only script which has to be called by the user)

```
import numpy as np
from poissonFFT import *
from myParam import *
import timeit

'''
Smoothening function
Input: Grid Dimension, No of iterations desired,
       optional - Plot Initial State (off by default), Plot final state (off by default), guess value for density (constant by default)
Output: Pressure (complex array), Density (complex array), Potential  (complex array), List of max residue at each iteration
'''
def smooth(N, iterations, initGraph=False,  finGraph=False, guessGiven = 0):

    #
    # Initializing Arrays
    #

    x = np.linspace(-(R*1.2), (R*1.2), N)
    y = np.linspace(-(R*1.2), (R*1.2), N)
    delta = abs(x[-1]-x[0])/N                  # Spatial and Time Step
    rhoGuess = np.zeros((N, N), dtype=complex) # Empty array for storing guess values of density

    #
    # Defining Functions
    #

    '''
    Populates the initial guess matrix with density values
    Input: Radial Distance from the center (float)
    Output: Density (Scalar)
    '''
    def rhoG(r):
        # Defining Boundary
        if abs(r - R) <= delta:
            return rhoB
        elif r < R:
            return rhoB

        # Defining Initial Guess
        else:
            return 0

    '''
    Computes the residue for one Jacobi Relaxation step
    Input: x-index, y-index, Guess Solution for Pressure (complex matrix of scalars), Guess Solution for potential (complex matrix of scalars)
    Output: Derivative term for Jacobi relaxation (tuple)
    '''
    def residue(i, j, pGuess, phiSol):

        # Coordinate Converstion of term 2
        r = np.sqrt(x[i]**2 + y[j]**2)
        theta = np.arctan2(y[j], x[i])

        # Gradient vector of Pressure
        term_1 = np.array([pGuess[i+1, j] - pGuess[i, j],
                           pGuess[i, j+1] - pGuess[i, j]])/delta

        # Gradient vector of gravitational potential
        term_2 = np.array([(phiSol[i+1, j] - phiSol[i, j]),
                           (phiSol[i, j+1] - phiSol[i, j])])/delta

        # Gradient vector of centrifugal force
        term_3 = np.array([2 * r * np.sin(theta)**2,
                           r**2 * np.sin(2*theta)])

        # Component transformation matrix (r, theta) -> (x, y)
        term_3 = pol2cart(term_3)

        return term_1 + (A*term_2 - B*term_3) * rhoGuess[i, j]


    '''
    Computes one Relaxation cycle using pressure residue on the grid
    Input: Pressure Guess (complex array), potential guess (complex array)
    Output: Improved Pressure Approximation (complex array), Residue on the grid (complex array)
    '''
    def JacobiRelaxation(pGuess, phiSol):

        # Initializing Arrays for Pressure
        pStep = np.ones((N, N), dtype=tuple) * pGuess  # Populating first step pressure with guess values
        pSol  = np.zeros((N, N), dtype=tuple)
        res   = np.zeros((N, N), dtype=tuple)

        for i in range(N-1):
            for j in range(N-1):
                r = np.sqrt(x[i]**2 + y[j]**2)
                if abs(r - R) <= delta:
                    pSol[i, j] = pStep[i, j]
```

```
                if r < R:
                    res[i,j] = residue(i, j, pStep, phiSol)
                    pSol[i, j] = pStep[i, j] + res[i,j]*(delta**2)/4

        pStep = pSol

        for m in range(N):
            for n in range(N):
                pGuess[m, n] = np.mean(pSol[m,n])
                res[m, n] = np.mean(res[m,n])

        return pGuess, res

    '''
    Solve the PDE System iteratively using FFT for Poisson and Jacobi for Pressure
    Input: Relaxation Steps (integer), Guess value for Pressure (complex array), Guess Value for Potential (
     complex array)
    Output: Relaxed Pressure (complex array), Density (complex array), Potential (complex array), list of
     maximum residue at each iteration
    '''
    def varUpdate(steps, pGuess, phiSol):

        maxErList = []
        k = 1 # loop counter
        while k <= steps:

            pGuess, error = JacobiRelaxation(pGuess, phiSol)
            rhoGuess = np.sqrt(pGuess)
            phiSol = PoissonSolveFFT2D(rhoGuess, N, N)

            maxErList.append(np.max(error).real)
            k += 1 # increment

        return pGuess, rhoGuess, phiSol, maxErList

    #
    # End of function definitions
    #

    # Checking if a density distribution has already been provided
    if np.array_equal(guessGiven, 0):
        # Populating Solution Array with Initial Guess
        for i in range(len(x)):
            for j in range(len(y)):
                rhoGuess[i, j] = rhoG(np.sqrt(x[i]**2 + y[j]**2))
    else:
        rhoGuess = guessGiven

    # Converting density to pressure using Equation of state
    pGuess = rhoGuess**2

    # Guess Solution for Phi
    phiSol = PoissonSolveFFT2D(rhoGuess, N, N)

    # If user commands, the following snippet plots the initial guess values of the variables
    if initGraph == True:

        import matplotlib.pyplot as plt

        fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 7))

        ax2.imshow(pGuess.real, cmap='gnuplot',
                   extent =[-1.2*R, 1.2*R,-1.2*R, 1.2*R])
        ax2.set_title("Pressure")
        ax1.imshow(rhoGuess.real, cmap='gnuplot',
                   extent =[-1.2*R, 1.2*R,-1.2*R, 1.2*R])
        ax1.set_title("Density")
        ax3.imshow(phiSol.real, cmap='gnuplot',
                   extent =[-1.2*R, 1.2*R,-1.2*R, 1.2*R])
        ax3.set_title("Potential")
        fig.suptitle('Initial Guess Distribution \n Grid Size = %i' %N, size=20)
        plt.savefig('fig/preCon.png')
        plt.show()

    start_time = timeit.default_timer()
    # This line computes the solution after relaxation
    p, rho, phi, er = varUpdate(iterations, pGuess, phiSol)
    elapsed = timeit.default_timer() - start_time  #Computing the runtime of relaxation

    # If user commands, the following snippet plots the arrays computed after relaxation
    if initGraph == True:

        fig, (ax1, ax2, ax3) = plt.subplots(1, 3, figsize=(20, 7))

        ax2.imshow(p.real, cmap='inferno',
                   extent =[-1.2*R, 1.2*R,-1.2*R, 1.2*R])
        ax2.set_title("Pressure")
        ax1.imshow(rho.real, cmap='inferno',
                   extent =[-1.2*R, 1.2*R,-1.2*R, 1.2*R])
        ax1.set_title("Density")
        ax3.imshow(phi.real, cmap='inferno',
```

```
                    extent =[-1.2*R, 1.2*R,-1.2*R, 1.2*R])
        ax3.set_title("Potential")
        fig.suptitle('Output after %i steps' %iterations, size=20)
        plt.savefig('fig/postCon.png')
        plt.show()

    print(iterations, 'relaxation steps on ', N,' X ', N, ' grid took ', np.round(elapsed, 2), ' sec')
    print('Maximum residue on the grid is ', er[-1])

    return p, rho, phi, er

'''
Plots the a heatmap of error on the grid after smoothening has been performed
Input: Error matrix as generated by smoothening function
Output: None - just displays a graph
(This redundant function is defined as a quick fix for some datatype compatibilities)
'''
def plotEr(res):
    from matplotlib.pyplot import colorbar, imshow, title
    N = (len(res))
    resGrid = np.zeros((N, N), dtype=float)
    for i in range(N):
        for j in range(N):
            resGrid[i, j] = abs(res[i, j].real)
    a = imshow(resGrid)
    colorbar(a)
    title("Heat Map of Residues on Grid")

```

d. multiGrid.py (machinery for multigrid optimizations)

```
from myParam import R, rhoB
from smoothening import smooth

'''
Restriction Operator (uses 9 target points)
Input: Numpy Array of size 2N x 2N
Output: Numpy Array scaled down to N x N, new dimension of the matrix
'''
def restriction(test_c):

    N = len(test_c)
    newN = int(N/2)
    out = ones((newN, newN), dtype=complex)

    for i in range(1, newN-1):
        for j in range(1, newN-1):
            out[i , j] = (1/16) * (4 * test_c[2*i, 2*j] +
                                   2 * (test_c[2*i, 2*j-1] + test_c[2*i, 2*j+1] + test_c[2*i+1, 2*j] + test_c
     [2*i-1, 2*j])
                                   + (test_c[2*i-1, 2*j-1] + test_c[2*i+1, 2*j-1] + test_c[2*i-1, 2*j+1] +
     test_c[2*i+1, 2*j+1]))

    return out*10, newN

'''
Prolongation Operator (uses 9 target points)
Input: Numpy Array of size N x N
Output: Numpy Array scaled down to 2N x 2N, new dimension of the matrix
'''
def prolongation(test_f):

    N = len(test_f)
    newN = int(2*N)
    out = ones((newN, newN), dtype=complex)

    for i in range(1, N-1):
        for j in range(1, N-1):
            out[2*i, 2*j]  = test_f[i, j]
            out[2*i+1, 2*j] = 0.5*(test_f[i+1, j] + test_f[i, j])
            out[2*i, 2*j+1] = 0.5*(test_f[i, j+1] + test_f[i, j])
            out[2*i+1, 2*j+1] = 0.25*(test_f[i, j] + test_f[i+1, j] + test_f[i, j+1] + test_f[i+1, j+1])

    return out/10, newN

'''
Prolongation Operator (uses 9 target points) but artificially imposes boundary conditions again after rescaling
Input: Numpy Array of size N x N
Output: Numpy Array scaled down to 2N x 2N, new dimension of the matrix
'''
def prolongImposeBC(coarse):
    fine, newN = prolongation(coarse)
    x = linspace(-(R*1.2), (R*1.2), newN)
    y = linspace(-(R*1.2), (R*1.2), newN)
    delta = abs(x[-1]-x[0])/newN

    for i in range(newN):
        for j in range(newN):
            r = sqrt((x[i]**2 + y[j]**2))
            if abs(r - R) <= delta:
                fine[i, j] = rhoB
```

```
            elif r > R:
                fine[i, j] = 0

    return fine, newN

'''
Restriction Operator (uses 9 target points) but artificially imposes boundary conditions again after rescaling
Input: Numpy Array of size 2N x 2N
Output: Numpy Array scaled down to N x N, new dimension of the matrix
'''
def restrictImposeBC(fine):
    coarse, newN = restriction(fine)
    x = linspace(-(R*1.2), (R*1.2), newN)
    y = linspace(-(R*1.2), (R*1.2), newN)
    delta = abs(x[-1]-x[0])/newN

    for i in range(newN):
        for j in range(newN):
            r = sqrt((x[i]**2 + y[j]**2))
            if abs(r - R) <= delta:
                coarse[i, j] = rhoB
            elif r > R:
                coarse[i, j] = 0

    return fine, newN

'''
Solves using FFT+Jacobi but with an initial guess arrived at by solving on a coarse grid and prolongating the
     solution to a finer grid
Input: Desired grid size N, number of relaxation steps,
       optional - Boolean, Save Graphs (False by default)
'''
def multiGridRelax(N, steps, graphSave = False):

    N_guess = int(N/2)
    print("First solving on grid of size N = ", N_guess)
    pSolG, rhoSolG, phiSolG, er = smooth(N_guess, int(0.5*steps), False, False)

    print("Now prolongating to desired resolution")
    fine, fineN = prolongImposeBC(rhoSolG)
    pSol, rhoSol, phiSol, er = smooth(fineN, steps, False, False, fine)

    if graphSave == True:

        import matplotlib.pyplot as plt

        fig1, (ax1, ax2) = plt.subplots(1, 2, figsize=(18, 7))

        ax1.imshow(rhoSolG.real, cmap='gnuplot',
                   extent =[-1.2*R, 1.2*R,-1.2*R, 1.2*R])
        ax1.set_title("Solution on Coarse Grid N = %i" %N_guess)

        ax2.imshow(fine.real, cmap='gnuplot',
                   extent =[-1.2*R, 1.2*R,-1.2*R, 1.2*R])
        ax2.set_title("Grid Prolongation to N = %i" %fineN)

        fig1.suptitle('Utilizing Multigrid Machinery', size=20)
        plt.savefig('fig/preCon_multiGrid.png')
        plt.show()

        fig2, (ax4, ax5, ax6) = plt.subplots(1, 3, figsize=(20, 7))

        ax4.imshow(rhoSol.real, cmap='gnuplot',
                   extent =[-1.2*R, 1.2*R,-1.2*R, 1.2*R])
        ax4.set_title("Density")
        ax5.imshow(pSol.real, cmap='gnuplot',
                   extent =[-1.2*R, 1.2*R,-1.2*R, 1.2*R])
        ax5.set_title("Pressure")
        ax6.imshow(phiSol.real, cmap='gnuplot',
                   extent =[-1.2*R, 1.2*R,-1.2*R, 1.2*R])
        ax6.set_title("Potential")
        fig2.suptitle('Final solution on finer grid \n Grid Size = %i' %fineN, size=20)
        plt.savefig('fig/postCon_multiGrid.png')
        plt.show()

    return pSol, rhoSol, phiSol, er
```

3 Figures

First, notice in Fig. 1 how relaxing for some steps on a coarse grid and then prolongation it to desired resolution as an initial guess gives the algorithm a noticeable boost after 250 iterations.

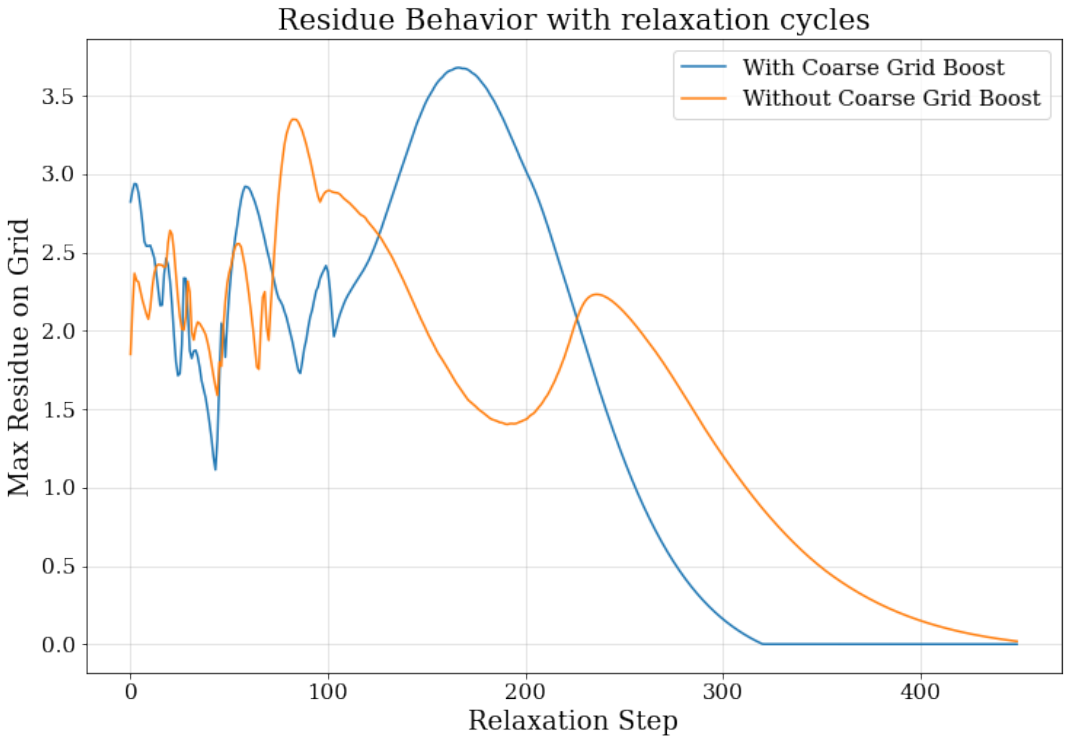
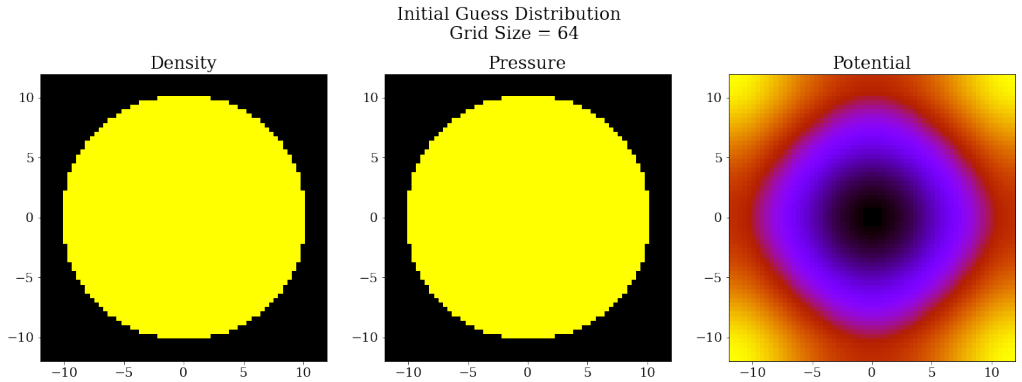


Figure 1: Residue behavior with relaxation iterations

Without the coarse grid approximation (i.e. using constant density all over as our initial guess), the output generated has been displayed in Fig. 2. Performing 450 relaxation steps on a 64 x 64 grid took 64 seconds on my laptop.



The following figures have been generated by calling multiGrid.py script. Fig. 3 shows the solution generated on a coarse grid and its prolongation to our desired resolution. Fig. 4 shows the final output.

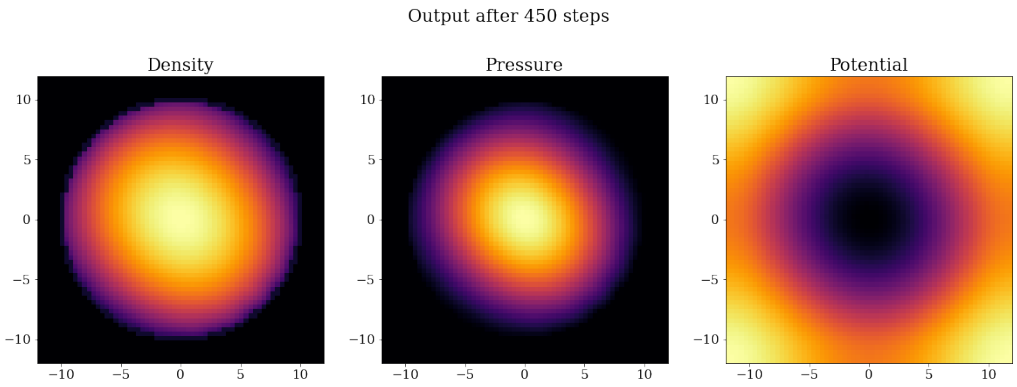


Figure 2: Figures generated without using approximation from a coarse grid

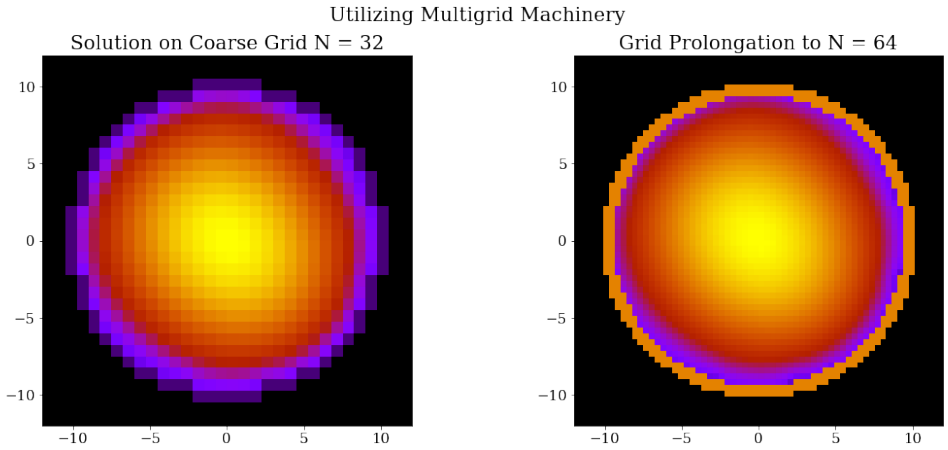


Figure 3: multiGrid.prolongImposeBC acting on a solution over coarse grid

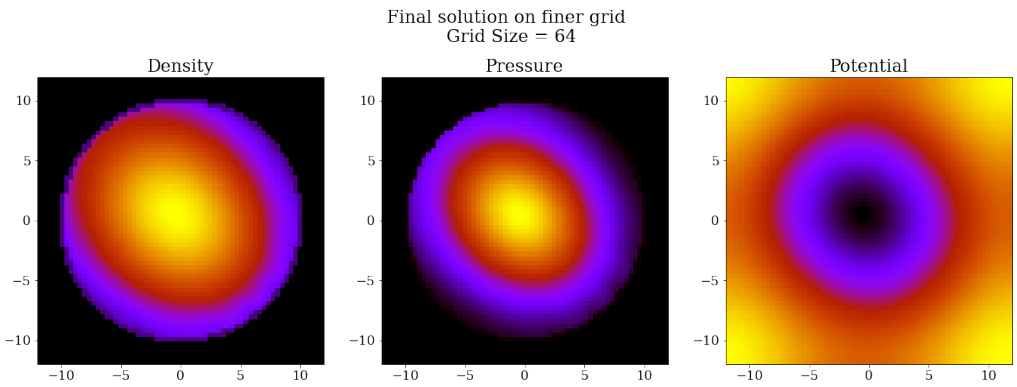


Figure 4: Figures generated using approximation on coarse grid as initial guess