NE 255 - Homework 3

University of California, Berkeley
Department of Nuclear Engineering

Daniel Hellfeld
dhellfeld@berkeley.edu

Problem 1
Derive the 1$^{\text{st}}$ order form of SP_5 with isotropic source and vacuum boundary conditions.

$\Rightarrow$ We begin by considering the slab geometry P_N equations derived in class

$$\left(\frac{l'+1}{2l'+1}\right)\frac{d}{dx}\phi_{l'+1}(x)+\left(\frac{l'}{2l'+1}\right)\frac{d}{dx}\phi_{l'-1}(x)+\Sigma_t(x)\phi_{l'}=\Sigma_{sl'}(x)\phi_{l'}(x)+s_{l'}(x)\,,$$

for $l'=0,1,...,N$. We also set $\phi_{-1}=0$ and $\phi_{N+1}=0$ or $(\frac{d}{dx}\phi_{N+1}=0)$. Now to *heuristically* derive the SP_N equations, we use Gelbard's idea of making simple substitutions to put us into a 3D system (we could use an asymptotic analysis to derive SP_N, but the heuristic approach is much simpler). To do so, we first replace odd l' values of $\phi_{l'}$ with a vector:

$$\phi_{l'}\rightarrow\vec{\phi}_{l'}=(\phi^x_{l'},\phi^y_{l'},\phi^z_{l'})^t\,.$$

Then, for even l' equations, the space derivative becomes a divergence:

$$\frac{d}{dx}\rightarrow\nabla\cdot\,,$$

and for the odd l' equations the space derivative becomes a gradient:

$$\frac{d}{dx}\rightarrow\nabla\,.$$

Making these substitutions into the P_N equations above and separating into $l'=0$, l' even, and l' odd equations, we arrive at the first-order form of the SP_N equations

$$\nabla\cdot\vec{\phi}_1+\Sigma_a\phi_0=s_0, \qquad (l'=0)$$

$$\Big(\frac{l'+1}{2l'+1}\Big)\nabla\phi_{l'+1}+\Big(\frac{l'}{2l'+1}\Big)\nabla\phi_{l'-1}+\Sigma_t\vec{\phi}_{l'}=\Sigma_{sl'}\vec{\phi}_{l'}+s_{l'}, \qquad (l'\text{ odd})$$

$$\Big(\frac{l'+1}{2l'+1}\Big)\nabla\cdot\vec{\phi}_{l'+1}+\Big(\frac{l'}{2l'+1}\Big)\nabla\cdot\vec{\phi}_{l'-1}+\Sigma_t\phi_{l'}=\Sigma_{sl'}\phi_{l'}+s_{l'}, \qquad (l'\text{ even},\ l'>0)$$

where $l'=0,1,2,...,N$, ϕ is the three-dimensional scalar flux, $\phi_{l'}$ is the l'^{th} moment of the scalar flux, $\vec{\phi}_{l'}=(\phi^x_{l'},\phi^y_{l'},\phi^z_{l'})^T$ (for odd l'), Σ_t is the total interaction cross-section, $\Sigma_{sl'}$ is the l'^{th} moment scattering cross-section, and $s_{l'}$ is the l'^{th} moment of an arbitrary neutron source (which could include fission). I did not include space dependences in the equations as to avoid clutter, but the scalar flux, cross-sections, and source will have spatial (x,y,z) dependencies.

Plugging in for $N=5$, we arrive at the following six equations

$$\begin{aligned}
l'=0 \quad &\Rightarrow \quad \nabla\cdot\vec{\phi}_1+\Sigma_a\phi_0=s_0\\
l'=1 \quad &\Rightarrow \quad \Big(\frac{2}{3}\Big)\nabla\phi_2+\Big(\frac{1}{3}\Big)\nabla\phi_0+\Sigma_t\vec{\phi}_1=\Sigma_{s1}\vec{\phi}_1+s_1\\
l'=2 \quad &\Rightarrow \quad \Big(\frac{3}{5}\Big)\nabla\cdot\vec{\phi}_3+\Big(\frac{2}{5}\Big)\nabla\cdot\vec{\phi}_1+\Sigma_t\phi_2=\Sigma_{s2}\phi_2+s_2\\
l'=3 \quad &\Rightarrow \quad \Big(\frac{4}{7}\Big)\nabla\phi_4+\Big(\frac{3}{7}\Big)\nabla\phi_2+\Sigma_t\vec{\phi}_3=\Sigma_{s3}\vec{\phi}_3+s_3\\
l'=4 \quad &\Rightarrow \quad \Big(\frac{5}{9}\Big)\nabla\cdot\vec{\phi}_5+\Big(\frac{4}{9}\Big)\nabla\cdot\vec{\phi}_3+\Sigma_t\phi_4=\Sigma_{s4}\phi_4+s_4\\
l'=5 \quad &\Rightarrow \quad \Big(\frac{6}{11}\Big)\nabla\phi_6+\Big(\frac{5}{11}\Big)\nabla\phi_4+\Sigma_t\vec{\phi}_5=\Sigma_{s5}\vec{\phi}_5+s_5
\end{aligned}$$

In the SP_N approximation, we set $\phi_{N+1} = 0$ or equivalently $\nabla\phi_{N+1} = 0$, therefore $\nabla\phi_6 = 0$. With an isotropic source, we can set all $l' \geq 1$ moments of the source to 0 ($s_1 = s_2 = s_3 = s_4 = s_5 = 0$):

$$\begin{aligned}
l' = 0 \quad &\Rightarrow \quad \nabla \cdot \vec{\phi}_1 + \Sigma_a \phi_0 = s_0 \\
l' = 1 \quad &\Rightarrow \quad \Big(\frac{2}{3}\Big)\nabla\phi_2 + \Big(\frac{1}{3}\Big)\nabla\phi_0 + (\Sigma_t - \Sigma_{s1})\vec{\phi}_1 = 0 \\
l' = 2 \quad &\Rightarrow \quad \Big(\frac{3}{5}\Big)\nabla \cdot \vec{\phi}_3 + \Big(\frac{2}{5}\Big)\nabla \cdot \vec{\phi}_1 + (\Sigma_t - \Sigma_{s2})\phi_2 = 0 \\
l' = 3 \quad &\Rightarrow \quad \Big(\frac{4}{7}\Big)\nabla\phi_4 + \Big(\frac{3}{7}\Big)\nabla\phi_2 + (\Sigma_t - \Sigma_{s3})\vec{\phi}_3 = 0 \\
l' = 4 \quad &\Rightarrow \quad \Big(\frac{5}{9}\Big)\nabla \cdot \vec{\phi}_5 + \Big(\frac{4}{9}\Big)\nabla \cdot \vec{\phi}_3 + (\Sigma_t - \Sigma_{s4})\phi_4 = 0 \\
l' = 5 \quad &\Rightarrow \quad \Big(\frac{5}{11}\Big)\nabla\phi_4 + (\Sigma_t - \Sigma_{s5})\vec{\phi}_5 = 0
\end{aligned}$$

We could relate some of the ϕ's to each other and reduce these equations to get to the *second order form*, but we will stop here.

The vacuum boundary conditions in the SP_N equations are obtained for all $\phi_{l'}$ by replacing μ with $\vec{n} \cdot \hat{\Omega}$ in the P_N Marshak vacuum boundary conditions, where $\vec{n}$ is the inward-normal and $\hat{\Omega}$ describes the direction of the flux. The P_N Marshak vacuum boundary conditions are given by

$$\int_{\mu_{in}} d\mu P_l(\mu)\psi(\vec{r}_0, \mu) = 0, \quad \text{for odd } l \ (= 1, 3, 5, ..., N)$$

where $P_l(\mu)$ are the Legendre polynomials and $\vec{r}_0$ is the position of the boundary. Expanding the angular flux in Legendre polynomials, we arrive at

$$\int_{\mu_{in}} d\mu P_l(\mu) \sum_{n=0}^{\infty} \Big(\frac{2n+1}{4\pi}\Big)\phi_n(\vec{r}_0)P_n(\mu) = 0, \quad \text{for odd } l \ (= 1, 3, 5, ..., N)$$

and pulling out some terms from the integral

$$\Rightarrow \sum_{n=0}^{\infty} \Big(\frac{2n+1}{4\pi}\Big)\phi_n(\vec{r}_0) \int_{\mu_{in}} d\mu P_l(\mu)P_n(\mu) = 0, \quad \text{for odd } l \ (= 1, 3, 5, ..., N)$$

Now, we replace the μ's with $\vec{n} \cdot \hat{\Omega}$, the integral over the inward coming μ's becomes an integral over $\hat{\Omega}$ where $\vec{n} \cdot \hat{\Omega} > 0$ (which is to say the inward coming flux), and finally we replace the ϕ_n's with our even and odd ϕ's from the SP_N equations (I will now use n instead of l' as to not get confused with l). When we separate out the even and odd ϕ's, we will have two terms and we will have to dot the odd ϕ's (vector) with the inward normal vector ($\vec{n}$). Doing all this, we arrive at

$$\begin{aligned}
&\sum_{n \text{ even}} \Big(\frac{2n+1}{4\pi}\Big)\phi_n(\vec{r}_0) \int_{\vec{n}\cdot\hat{\Omega}>0} d\hat{\Omega} P_l(\vec{n} \cdot \hat{\Omega})P_n(\vec{n} \cdot \hat{\Omega}) \ + \\
&\qquad \sum_{n \text{ odd}} \Big(\frac{2n+1}{4\pi}\Big)\vec{n} \cdot \vec{\phi}_n(\vec{r}_0) \int_{\vec{n}\cdot\hat{\Omega}>0} d\hat{\Omega} P_l(\vec{n} \cdot \hat{\Omega})P_n(\vec{n} \cdot \hat{\Omega}) = 0, \qquad \text{for odd } l \ (= 1, 3, 5, ..., N)
\end{aligned}$$

Now if we were integrating over the whole range, we would be able to use the orthogonality of Legendre polynomials to cancel out the even terms (because l is only defined to be odd), but this is not the case.

Problem 2
Consider the integral

$$\int_{4\pi} d\hat{\Omega}\,\hat{\Omega}$$

The LQ_N quadrature set is given in Table 4-1 in Lewis and Miller (*E.E. Lewis and W.F. Miller, Computational Methods of Neutron Transport, John Wiley & Sons, Inc., 1984.*). Recall that $\mu_i = \eta_i = \xi_i$ for a given level, i.

(a) Use the S_4 LQ_N quadrature set to execute this integral.

$\Rightarrow$ A quadrature set allows us to evaluate the integral of a function over some domain using a discrete set of points and cooresponding weights:

$$\int dx f(x) = \sum_{n\in\text{set}} w_n f(n)$$

Now, we can decompose $\hat{\Omega}$ into its directional cosines (μ, η, and ξ) like $\hat{\Omega} = \sqrt{\mu^2+\eta^2+\xi^2}$ $(= 1)$ and use the LQ_N quadrature set to determine the weights at each μ_i, η_i, and ξ_i, where at each level $\mu_i = \eta_i = \xi_i$. In each S_N set, we have $N(N+2)/8$ quadrature points in each octant, therefore for S_4 we have 3 points which, using Fig. 1a below, correspond to $\{\mu_1, \eta_1, \xi_2\}$, $\{\mu_1, \eta_2, \xi_1\}$, and $\{\mu_2, \eta_1, \xi_1\}$. Using Table 4-1, we have that $\mu_1 = \eta_1 = \xi_1 = 0.3500212$ with a weight of 0.3333333 and $\mu_2 = \eta_2 = \xi_2 = 0.8688903$ also with a weight of 0.3333333. Plugging all this in, we get

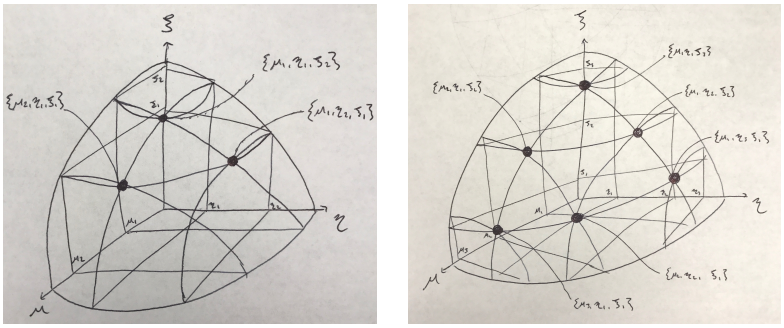
$$\int_{4\pi} d\hat{\Omega}\,\hat{\Omega} \approx \sum_{8\text{ octants}} w\sqrt{\mu_1^2+\eta_1^2+\xi_2^2} + w\sqrt{\mu_1^2+\eta_2^2+\xi_1^2} + w\sqrt{\mu_2^2+\eta_1^2+\xi_1^2}$$

Since all the directional cosines are squared in the equation for $\hat{\Omega}$, we do not need to care much about the fact that they can be negative in certain octants (however in general we would have to be careful about signs). Therefore the sum over the 8 octants can be reduced to just mutplying by 8.

$$\begin{aligned}\int_{4\pi} d\hat{\Omega}\,\hat{\Omega} &\approx 8\Big(w\sqrt{\mu_1^2+\eta_2^2+\xi_2^2} + w\sqrt{\mu_2^2+\eta_2^2+\xi_1^2} + w\sqrt{\mu_2^2+\eta_1^2+\xi_2^2}\Big)\\ &\approx 8\Big(0.3333333\sqrt{1} + 0.3333333\sqrt{1} + 0.3333333\sqrt{1}\Big)\\ &\approx 8(0.9999999)\end{aligned}$$

Now, the weights are normalized to 1 in each quadrant and thus 8 over the entire sphere. However, we know that the integral over the sphere should be equal to 4π. Therefore we have to multiply by a factor of $4\pi/8$ to get the correct answer. Doing so, we get

$$(4\pi/8)(8)(0.9999999) = (4\pi)(0.9999999)$$



(a) (b)

Fig. 1. S4 (a) and S6 (b) quaderature point sketches. Please excuse my terrible drawing abilities.

(b) Repeat it with S_6. What do you observe?

$\Rightarrow$ Using S_6, we will have the following 6 points (see Fig. 1b): $\{\mu_1, \eta_1, \xi_3\}$, $\{\mu_3, \eta_1, \xi_1\}$, and $\{\mu_1, \eta_3, \xi_1\}$ with weight 0.1761263, and $\{\mu_2, \eta_1, \xi_2\}$, $\{\mu_1, \eta_2, \xi_2\}$, and $\{\mu_2, \eta_2, \xi_1\}$ with weight 0.1572071. Plugging in everything, we get

$$\int_{4\pi} d\hat{\Omega}\,\hat{\Omega} \approx (4\pi/8) \sum_{8 \text{ octants}} \Big(w_1\sqrt{\mu_1^2+\eta_1^2+\xi_3^2} + w_1\sqrt{\mu_3^2+\eta_1^2+\xi_1^2} + w_1\sqrt{\mu_1^2+\eta_3^2+\xi_1^2} + w_2\sqrt{\mu_2^2+\eta_1^2+\xi_2^2} + w_2\sqrt{\mu_1^2+\eta_2^2+\xi_2^2} + w_2\sqrt{\mu_2^2+\eta_2^2+\xi_1^2} \Big)$$

$$\Rightarrow (4\pi/8)(8)\Big((3)(0.1761263) + (3)(0.1572071)\Big)$$
$$= (4\pi)(1.0000002)$$

We see that this slightly overshoots the true value, by an amount slightly above the value we undershot using S_4. But in essence we calculate the same value, as expected.

(c) Write a short code to execute this integration (and higher orders if you'd like). Try a few different functions. Turn in the code and the evaluation of these functions. Include comments on what you observe.

$\Rightarrow$ See the Python code below for the quadrature integration of three functions ($f(\hat{\Omega}) = \hat{\Omega}$, $f(\hat{\Omega}) = \mu + \eta + \xi$, and $f(\hat{\Omega}) = \mu^2 = \cos^2(\theta)$) with the ability to use N = 2, 4, 6, 8, 12, and 16.

```
# NE 255 - Homework 3, Problem 2c
# Daniel Hellfeld
# 10/20/16

# Imports
import numpy as np
import sys

# Function that retrieves the quadrature points, weights, and point-weight triangles
def PointsAndWeights(N):

    if N == 2:
        mu  = eta = xi = np.array([0.5773503])
        w = np.array([1.0000000])
        cells = [np.array(a) for a in [[1]]]
    elif N == 4:
        mu  = eta = xi = np.array([0.3500212, 0.8688903])
        w = np.array([0.3333333])
        cells = [np.array(a) for a in [[1], [1,1]]]
    elif N == 6:
        mu  = eta = xi = np.array([0.2666355, 0.6815076, 0.9261808])
        w = np.array([0.1761263, 0.1572071])
        cells = [np.array(a) for a in [[1], [2,2], [1,2,1]]]
    elif N == 8:
        mu  = eta = xi = np.array([0.2182179, 0.5773503, 0.7867958, 0.9511897])
        w = np.array([0.1209877, 0.0907407, 0.0925926])
        cells = [np.array(a) for a in [[1], [2,2], [2,3,2], [1,2,2,1]]]
    elif N == 12:
        mu  = eta = xi = np.array([0.1672126, 0.4595476, 0.6280191, 0.7600210, 0.8722706, 0.9716377])
        w = np.array([0.0707626, 0.0558811, 0.0373377, 0.0502819, 0.0258513])
        cells = [np.array(a) for a in [[1], [2,2], [3,4,3], [3,5,5,3], [2,4,5,4,2], [1,2,3,3,2,1]]]
    elif N == 16:
        mu = eta = xi = np.array([0.1389568, 0.3922893, 0.5370966, 0.6504264, 0.7467506, 0.8319966, 0.9092855, 0.9805009])
        w = np.array([0.0489872, 0.0413296, 0.0212326, 0.0256207, 0.0360486, 0.0144589, 0.0344958, 0.0085179])
        cells = [np.array(a) for a in [[1], [2,2], [3,5,3], [4,6,6,4], [4,7,8,7,4], [3,6,8,8,6,3], [2,5,6,7,6,5,2],
             [1,2,3,4,4,3,2,1]]]
    else:
        sys.exit("Error: I can only do N = 2, 4, 6, 8, 12, and 16. You entered N = %i." % N)

    return mu,eta,xi,w,cells,N

# Get the signs of [mu,eta,xi] in each octant
def GetSigns(octant):

```

```
44      if   octant == 0: return [+1,+1,+1]
45      elif octant == 1: return [-1,+1,+1]
46      elif octant == 2: return [-1,-1,+1]
47      elif octant == 3: return [+1,-1,+1]
48      elif octant == 4: return [+1,+1,-1]
49      elif octant == 5: return [-1,+1,-1]
50      elif octant == 6: return [-1,-1,-1]
51      elif octant == 7: return [+1,-1,-1]
52
53  # Compute the quadrature integral
54  def QuadratureIntegral(f,mu,eta,xi,w,cells,N):
55
56      sum_ = 0
57      for octant in range(8):
58          signs = GetSigns(octant)
59
60          for i in range(N/2):
61              for j in range(N/2):
62                  for k in range(N/2):
63
64                      if np.isclose(mu[i]**2 + eta[j]**2 + xi[k]**2, 1):
65
66                          val    = f(mu[i]*signs[0], eta[j]*signs[1], xi[k]*signs[2])
67                          weight = w[ cells[((N/2)-1) - k][j] - 1 ]
68
69                          sum_ += val*weight
70
71      norm = 4.*np.pi/8.
72      return norm * sum_
73
74
75  # --------
76
77  def Omega(mu,eta,xi):
78      return np.sqrt(mu**2 + eta**2 + xi**2)
79
80  def Function1(mu,eta,xi):
81      return mu + eta + xi
82
83  def Function2(mu,eta,xi):
84      return mu**2
85
86
87  # ----------
88
89  # Choose a quadrature set (N = 2,4,6,8,12,16)
90  N = 8
91
92  # Calculate the integral over Omega for the three functions above
93  print QuadratureIntegral(Omega,*PointsAndWeights(N)) # expect to be 4pi
94  print QuadratureIntegral(Function1,*PointsAndWeights(N)) # expect to be 0
95  print QuadratureIntegral(Function2,*PointsAndWeights(N)) # expect to be 2pi*(2/3)
```

Now, we expect the following results for the functions in the script (I do not feel the need to explicitly show the steps)

$$\int_{4\pi} \hat{\Omega}\, d\hat{\Omega} = 4\pi; \qquad \int_{4\pi} (\mu + \eta + \xi)\, d\hat{\Omega} = 0; \qquad \int_{4\pi} \mu^2\, d\hat{\Omega} = \int_{4\pi} \cos^2\theta\, d\hat{\Omega} = 2\pi(2/3)$$

And if we run the script above for multiple values of N, we get the following results

	$f(\hat{\Omega}) = \hat{\Omega}$	$f(\hat{\Omega}) = \mu + \eta + \xi$	$f(\hat{\Omega}) = \mu^2 = \cos^2(\theta)$
True Value	12.5663706	0.0	4.1887902
N = 2	12.5663713	0.0	4.1887907
N = 4	12.5663696	-8.7196712e-17	4.1887899
N = 6	12.5663718	-6.1037699e-16	4.1887901
N = 8	12.5663695	2.8338932e-16	4.1887899
N = 12	12.5663730	-6.6487493e-16	4.1887910
N = 16	12.5663619	1.2316536e-15	4.1887874

All the quadrature sets seem to approximate the integrals fairly well, up to what seems to be most likely just rounding errors. Also notice that is seems to alternate in overshooting and undershooting as N is increased.

Problem 3

(a) Briefly compare the diffusion equation, deterministic methods, and monte carlo methods in terms of complexity, accuracy, run time, and range of applicability.

⇒ The diffusion equation is an appriximation to the transport equation - specifically under the assumptions of azimuthally symmetric scattering, scattering is at most linear anisotropic, and that the angular flux is also at most linearly anisotropic. Solutions are not valid when solved for near voids, boundarys, sources, or in strongly absorbing media. This severly limits the applicability of the diffusion equation, but for most macroscopic quantities we are concerned about in nuclear reactor physics (such as nuclear interaction rates, i.e. fission) this can be acceptable. The diffusion equation is quite simple and can in some cases be solved analytically by hand or using simple diffusion solvers.

Deterministic methods are methods by which we try to solve the transport or diffusion equation. This is done by discretizing all the variables in the equation (including time, space, energy, and direction), solving equations in each discretized space, and correctly communicating information between the discretized spaces. The complexity, accuracy, and run time are all dependent on the extent of discretization. If finer discretization is used, the solutions are more accurate, but the equations become more complex and the computational expenses significantly increase. While the equations in the solvers can be complex, the inputs given to the solvers tend to be quite simple (I want this solution in this system using these discretizations, etc.). Deterministic solutions are also global in the sense that the solutions we produce are valid in the entire space (constrained by the meshing, of course).

Monte Carlo methods are akin to deterministic methods as they are both methods to solve the diffusion or transport equation. In contrast to deterministic methods, however, in Monte Carlo we use continuous variables and sample every single interaction every particle has until we have sampled enough times that the solution we generate is statistically significant. We therefore do not necessary solve any equations (apart from the interaction probability sampling equations), but instead follow multiple particles through multiple stochastic histories to find an average behavior of some unknown (such as the neutron flux). Monte Carlo is regarded, in most cases, as much less complex than deterministic methods (though correctly sampling interactions can be difficult as well as building the geometry and including all the necessary physics), but it comes at a price of run time. Because so many particle histories need to be run in order to gain sufficient statistics, Monte Carlo methods are typically slower than deterministic. However, Monte Carlo is also much easier to parallelize (becuase each particle history is independent of the others and thus they can be simulated at the same time) which can lead to significant speed ups. With increased run time, we will get a more precise answer (a smaller uncertainty), however it does not always promise a more accurate answer (that relies on the correct modeling of both the system and the physics). The solutions we generate with Monte Carlo are also considered local solutions (as opposed to deterministic methods) becuase of the use of local tallies, such as the neutron flux at a specific point in space (as opposed to the flux everywhere). We do this becuase to generate enough statistics to the get the global solution would take a enormous amout of time!

(b) Given what you've learned about deterministic methods so far, discuss strengths and weaknesses.

⇒ The strengths of deterministic methods are that they are fast, they produce global solutions that have the same quality everywhere in the solution space, and the inputs to the solvers can be simple. The weaknesses are that the quality of the solutions are dependent on the level of discretization (which can become very computationally intensive, and it remains difficult to parallelize deterministic codes on CPUs) and solutions will always include some truncation error. Also, the solutions we produce are only valid in the meshing we input and we are subject to Ray Effects - if we only solve along certain directions, then in certain cases such as low scattering media, then our solutions are constrained only to the directions we input and the solution space we solve in can become quite sparse.

Problem 4
Write a function that generates the associated Legendre Polynomials

$$P_l^m(x) = \frac{(-1)^m}{2^l l!}(1-x^2)^{m/2}\frac{d^{l+m}}{dx^{l+m}}(x^2-1)^l$$

Use this function in a function that generates spherical harmonics

$$Y_{lm}(\theta,\varphi) = (-1)^m\sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}}P_{lm}(\cos\theta)e^{-im\varphi}$$

(a) Generate and plot the following $l = 0, 1, 2$ for $l \leq m \leq l$ (recall we can relate the negative m to positive m values). You will need to discretize θ and μ fairly finely (I suggest 30 increments in each to start so you get a real sense of the shape of the harmonics).

$\Rightarrow$ See the attached Python script below for the generation of the associated Legenedre Polynomials and Spherical Harmonics. The script produces 3D plots using the mayavi package, and 2D screeshots are shown in Fig. 1 for both the real and imaginary parts. In the plots, I use two methods to visualize the spherical harmonics - a heat map on a unit sphere where the color corresponds to the value of the spherical harmonic at a particular θ and ϕ, and a distorted sphere where the distance from the origin represents the absolute value of the spherical harmoinc at a particular θ and ϕ (the color represents the true value of the spherical harmonic).

```
# NE 255 - Homework 3, Problem 4a
# Daniel Hellfeld
# 10/20/16

# Imports
import numpy as np
import matplotlib.pyplot as plt
from mayavi import mlab

# Define fatcorial to return a float instead of int
def factorial_(a):
    return float(np.math.factorial(a))

# Define absolute value to return a float instead of int
def abs_(a):
    return np.fabs(a)

def Derivative(f):
    # Discrete approximation to a derivative (central difference)
    # Choose h small, but not too small to cause precision issues
    def df(x, h=1.e-2):
        return ( f(x + h) - f(x - h) ) / (2.*h)
    return df

# Function to generate the Associated Legendre Polynomials
def AssocLegendrePoly(x,l,m):
    # Terms
    a = ( ((-1.)**abs_(m)) / ( (2.**l) * factorial_(l) ) )
    b = (1. - x**2.)**(abs_(m)/2.)
    def q(y): return (y**2. - 1.)**l
    c = q

    # Take derivatives
    if (l+m > 0.):
        for i in range(int(l + abs_(m))):
            c = Derivative(c)

    # Multiply terms together
    d = a * b * c(x)

    # Check if m is negative, adjust accordingly
    if m < 0:
        p = (-1.)**abs_(m) * (factorial_(l - abs_(m)) / factorial_(l + abs_(m)))
        return p * d
    else:
        return d

def SphericalHarmonics(theta,phi,l,m):
    # Terms
```

```
50      a = (-1.)**abs_(m) * np.sqrt( ((2.*l + 1.)/(4.*np.pi)) * ((factorial_(l - abs_(m)))/(factorial_(l + abs_(m)))) )
51      b = AssocLegendrePoly(np.cos(theta),l,abs_(m))
52      c = np.exp(1.j * abs_(m) * phi)
53
54      # Multiply terms together
55      d = a*b*c
56
57      # Check if m is negative, adjust accordingly
58      if m < 0:
59          return (-1)**abs_(m) * np.conjugate(d)
60      else:
61          return d
62
63  def SH_plot(l, m, space='real'):
64
65      # Define our theta and phi spacing
66      theta = np.linspace(0, np.pi, 100.)
67      phi = np.linspace(0, 2.*np.pi, 100.)
68
69      # Make into a mesh
70      [THETA,PHI] = np.meshgrid(theta,phi)
71
72      # Define x,y,z from our theta and phi
73      r = 1.
74      x = r*np.sin(THETA)*np.cos(PHI)
75      y = r*np.sin(THETA)*np.sin(PHI)
76      z = r*np.cos(THETA)
77
78      # Determine whether we want real or imaginary part (real is default)
79      if space == "real":
80          ww = np.real(SphericalHarmonics(THETA,PHI,l,m))
81      elif space == 'imag':
82          if m == 0: return # imaginary part is zero everywhere when m = 0
83          else: ww = np.imag(SphericalHarmonics(THETA,PHI,l,m))
84      else:
85          print "I do not recognize your input space command (either 'real' or 'imag')"
86
87
88      # Get the absolute value of the SH functions
89      # This is used to generate the distored sphere images
90      qq = abs(ww)
91
92      # Plot distored sphere with intensity
93      mlab.figure(bgcolor=(1,1,1),fgcolor=(0,0,0))
94      p = mlab.mesh(x*qq, y*qq, z*qq, scalars=ww, colormap='jet')
95      mlab.colorbar(p, orientation="vertical", title="%s(Y$%i%i$)" %(space,l,m))
96      mlab.axes(extent=[-0.6, 0.6, -0.6, 0.6, -0.6, 0.6], nb_labels=3)
97
98      # Plot a scaled down unit sphere with intensity
99      scale = 0.3
100     p1 = mlab.mesh(x*scale, y*scale, z*scale + 1.5, scalars=ww, colormap='jet')
101
102     mlab.view(45, 70, 7)
103
104
105
106 # ------
107
108
109 l_array = np.array((0,1,2))
110
111 # Plot the real and imaginary values
112 for l in l_array:
113     if l == 0 :
114         m = 0xc
115         SH_plot(l,m,'real')
116         SH_plot(l,m,'imag')
117     else:
118         for m in range(-l, l+1):
119             SH_plot(l,m,'real')
120             SH_plot(l,m,'imag')
121
122 mlab.show()
```

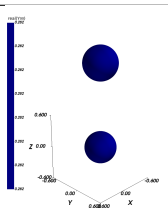
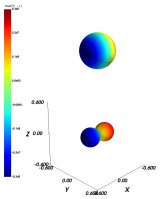
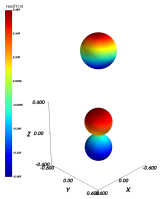
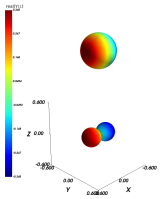
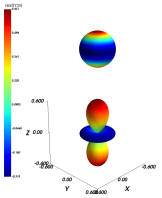
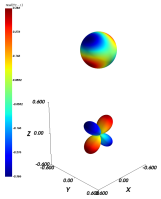
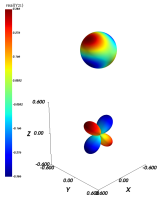
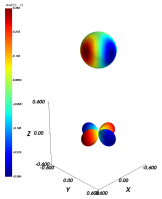
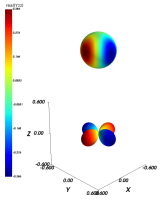
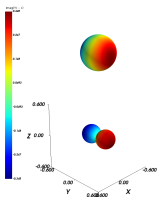
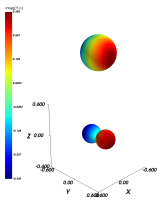
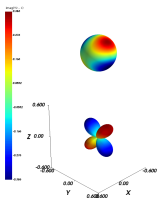
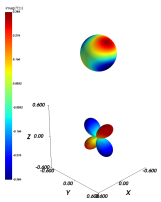
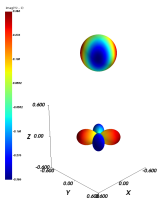
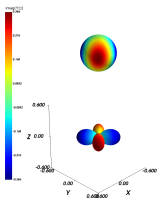
Real	m = 0	m = -1	m = 1	m = -2	m = 2
l = 0					
l = 1					
l = 2					
Imaginary	m = 0	m = -1	m = 1	m = -2	m = 2
l = 0					
l = 1					
l = 2					

Fig. 2. Real and imaginary parts of the Spherical Harmonics for $l = 0, 1, 2$ and $-l \leq m \leq l$. Notice that the imaginary parts are zero when $m = 0$.

(b) Now, we will approximate the external source. Using the S_4 quadrature to do the integrations and $q_e = 1$ for all angles: use the equations for external source we developed in class, calculate the external source for $N = 0, 1, 2$.

$\Rightarrow$ The expansion of external source can be written as follows

$$q_e^g(\vec{r}, \hat{\Omega}) = \sum_{l=0}^{N} \left[Y_{l0}^e(\hat{\Omega}) q_{l0}^g(\vec{r}) + \sum_{m=1}^{l} \left(Y_{lm}^e(\hat{\Omega}) q_{lm}^g(\vec{r}) + Y_{lm}^o(\hat{\Omega}) s_{lm}^g(\vec{r}) \right) \right],$$

where Y_{lm} are the spherical harmonics of order l and degree m, e and o represent the even and odd moments, respectively, and the even and odd source moments are given by

$$q_{lm}^g = \int_{4\pi} Y_{lm}^e(\hat{\Omega}) q_e^g(\hat{\Omega})\, d\hat{\Omega} = \int_{4\pi} Y_{lm}^e(\hat{\Omega})\, d\hat{\Omega}\,, \quad m \geq 0$$
$$s_{lm}^g = \int_{4\pi} Y_{lm}^o(\hat{\Omega}) q_e^g(\hat{\Omega})\, d\hat{\Omega} = \int_{4\pi} Y_{lm}^o(\hat{\Omega})\, d\hat{\Omega}\,, \quad m > 0$$

where

$$Y_{lm}^e(\hat{\Omega}) = D_{lm} P_{lm}(\cos\theta) \cos(m\phi)$$
$$Y_{lm}^o(\hat{\Omega}) = D_{lm} P_{lm}(\cos\theta) \sin(m\phi)$$
$$D_{lm} = (-1)^m \sqrt{(2 - \delta_{m0}) \frac{2l+1}{4\pi} \frac{(l-m)!}{(l+m)!}}$$

Now we will need to evaluate these integrals for $l = 0, 1, 2$ and $m = 0 \to l$. And we will us the S_4 quadrature to evaluate the integrals. The spherical harmonics are defined using θ and ϕ and the quadrature uses μ, η, and ξ. Therefore we first need to convert from one to the other. In this treatment, μ is like z, η is like x, and ξ is like y. Therefore we have

$$\theta = \cos^{-1}(\mu)$$
$$\phi = \tan^{-1}\left(\frac{\xi}{\eta}\right)$$

Next, we can use the quadrature code and the spherical harmonic code to evaluate all the integrals using S_4. In doing so (see the Python code below), we get the following results for the even components (q_{lm}^g) (accounting for the signs of μ, η, and ξ in the 8 octants):

$$q_{00}^g = \int_{4\pi} Y_{00}^e(\hat{\Omega})\, d\hat{\Omega} = \frac{4\pi}{24} \sum_{8 \text{ octants}} Y_{00}^e(\mu_1, \eta_1, \xi_2) + Y_{00}^e(\mu_1, \eta_2, \xi_1) + Y_{00}^e(\mu_2, \eta_1, \xi_1) = 3.5449$$
$$q_{10}^g = \int_{4\pi} Y_{10}^e(\hat{\Omega})\, d\hat{\Omega} = \frac{4\pi}{24} \sum_{8 \text{ octants}} Y_{10}^e(\mu_1, \eta_1, \xi_2) + Y_{10}^e(\mu_1, \eta_2, \xi_1) + Y_{10}^e(\mu_2, \eta_1, \xi_1) \approx 0$$
$$q_{11}^g = \int_{4\pi} Y_{11}^e(\hat{\Omega})\, d\hat{\Omega} = \frac{4\pi}{24} \sum_{8 \text{ octants}} Y_{11}^e(\mu_1, \eta_1, \xi_2) + Y_{11}^e(\mu_1, \eta_2, \xi_1) + Y_{11}^e(\mu_2, \eta_1, \xi_1) \approx 0$$
$$q_{20}^g = \int_{4\pi} Y_{20}^e(\hat{\Omega})\, d\hat{\Omega} = \frac{4\pi}{24} \sum_{8 \text{ octants}} Y_{20}^e(\mu_1, \eta_1, \xi_2) + Y_{20}^e(\mu_1, \eta_2, \xi_1) + Y_{20}^e(\mu_2, \eta_1, \xi_1) \approx 0$$
$$q_{21}^g = \int_{4\pi} Y_{21}^e(\hat{\Omega})\, d\hat{\Omega} = \frac{4\pi}{24} \sum_{8 \text{ octants}} Y_{21}^e(\mu_1, \eta_1, \xi_2) + Y_{21}^e(\mu_1, \eta_2, \xi_1) + Y_{21}^e(\mu_2, \eta_1, \xi_1) \approx 0$$
$$q_{22}^g = \int_{4\pi} Y_{22}^e(\hat{\Omega})\, d\hat{\Omega} = \frac{4\pi}{24} \sum_{8 \text{ octants}} Y_{22}^e(\mu_1, \eta_1, \xi_2) + Y_{22}^e(\mu_1, \eta_2, \xi_1) + Y_{22}^e(\mu_2, \eta_1, \xi_1) \approx 0$$

Notice the factor of (1/24), this is becuase of the $(4\pi/8)$ out in front being multiplied by the (1/3) weight for each quaderature point. I used the $\approx$ symbol, becuase the calculation gave very small answers in most cases

($\sim 10^{-16}$). Also note that there is no spatial dependence in the final results. For the odd components (s^g_{lm}), we get

$$s^g_{00} = \int_{4\pi} Y^o_{00}(\hat{\Omega})\, d\hat{\Omega} = \frac{4\pi}{24} \sum_{8 \text{ octants}} Y^o_{00}(\mu_1,\eta_1,\xi_2) + Y^o_{00}(\mu_1,\eta_2,\xi_1) + Y^o_{00}(\mu_2,\eta_1,\xi_1) \approx 0$$

$$s^g_{10} = \int_{4\pi} Y^o_{10}(\hat{\Omega})\, d\hat{\Omega} = \frac{4\pi}{24} \sum_{8 \text{ octants}} Y^o_{10}(\mu_1,\eta_1,\xi_2) + Y^o_{10}(\mu_1,\eta_2,\xi_1) + Y^o_{10}(\mu_2,\eta_1,\xi_1) \approx 0$$

$$s^g_{11} = \int_{4\pi} Y^o_{11}(\hat{\Omega})\, d\hat{\Omega} = \frac{4\pi}{24} \sum_{8 \text{ octants}} Y^o_{11}(\mu_1,\eta_1,\xi_2) + Y^o_{11}(\mu_1,\eta_2,\xi_1) + Y^o_{11}(\mu_2,\eta_1,\xi_1) \approx 0$$

$$s^g_{20} = \int_{4\pi} Y^o_{20}(\hat{\Omega})\, d\hat{\Omega} = \frac{4\pi}{24} \sum_{8 \text{ octants}} Y^o_{20}(\mu_1,\eta_1,\xi_2) + Y^o_{20}(\mu_1,\eta_2,\xi_1) + Y^o_{20}(\mu_2,\eta_1,\xi_1) \approx 0$$

$$s^g_{21} = \int_{4\pi} Y^o_{21}(\hat{\Omega})\, d\hat{\Omega} = \frac{4\pi}{24} \sum_{8 \text{ octants}} Y^o_{21}(\mu_1,\eta_1,\xi_2) + Y^o_{21}(\mu_1,\eta_2,\xi_1) + Y^o_{21}(\mu_2,\eta_1,\xi_1) \approx 0$$

$$s^g_{22} = \int_{4\pi} Y^o_{22}(\hat{\Omega})\, d\hat{\Omega} = \frac{4\pi}{24} \sum_{8 \text{ octants}} Y^o_{22}(\mu_1,\eta_1,\xi_2) + Y^o_{22}(\mu_1,\eta_2,\xi_1) + Y^o_{22}(\mu_2,\eta_1,\xi_1) \approx 0$$

Again, all the calculations results in either zero or very small numbers. These results make sense we when analytically calculate the integrals, which is easy enough to check with Mathematica. And this makes our life easier because all of the terms are zero except for q^g_{00}. So, plugging these results back into our external source expansion, we get the following for $N = 0, 1, 2$

$$N = 0 \quad \Rightarrow \quad q^g_e(\vec{r},\hat{\Omega}) = Y^e_{00}(\hat{\Omega}) q^g_{00} = \frac{3.5449}{2}\sqrt{\frac{1}{\pi}}$$

$$N = 1 \quad \Rightarrow \quad q^g_e(\vec{r},\hat{\Omega}) = Y^e_{00}(\hat{\Omega}) q^g_{00} = \frac{3.5449}{2}\sqrt{\frac{1}{\pi}}$$

$$N = 2 \quad \Rightarrow \quad q^g_e(\vec{r},\hat{\Omega}) = Y^e_{00}(\hat{\Omega}) q^g_{00} = \frac{3.5449}{2}\sqrt{\frac{1}{\pi}}$$

We see that this is just a constant value over all space and angle. This makes sense becuase we used $q_e = 1$, which is to say the source strength is constant and uniform over all space and angle, so it is reasonable that the spherical harmonic expansion of the source will also be constant in space and angle.

```
# NE 255 - Homework 3, Problem 4b
# Daniel Hellfeld
# 10/20/2016

# Imports
import numpy as np
import matplotlib.pyplot as plt
from mayavi import mlab

# Define fatcorial to return a float instead of int
def factorial_(a):
    return float(np.math.factorial(a))

# Define absolute value to return a float instead of int
def abs_(a):
    return np.fabs(a)

def Derivative(f):
    # Discrete approximation to a derivative (central difference)
    # Choose h small, but not too small to cause precision issues
    def df(x, h=1.e-4):
        return ( f(x + h) - f(x - h) ) / (2.*h)
    return df

# Function to generate the Associated Legendre Polynomials
def AssocLegendrePoly(x,l,m):
    # Terms
    a = ( ((-1.)**abs_(m)) / ( (2.**l) * factorial_(l) ) )
```

```
    b = (1. - x**2.)**(abs_(m)/2.)
    def q(y): return (y**2. - 1.)**l
    c = q

    # Take derivatives
    if (l+m > 0.):
        for i in range(int(l + abs_(m))):
            c = Derivative(c)

    # Multiply terms together
    d = a * b * c(x)

    # Check if m is negative, adjust accordingly
    if m < 0:
        p = (-1.)**abs_(m) * (factorial_(l - abs_(m)) / factorial_(l + abs_(m)))
        return p * d
    else:
        return d

def SphericalHarmonics(theta,phi,l,m):
    # Terms
    a = (-1.)**abs_(m) * np.sqrt( ((2.*l + 1.)/(4.*np.pi)) * ((factorial_(l - abs_(m)))/(factorial_(l + abs_(m)))) )
    b = AssocLegendrePoly(np.cos(theta),l,abs_(m))
    c = np.exp(1.j * abs_(m) * phi)

    # Multiply terms together
    d = a*b*c

    # Check if m is negative, adjust accordingly
    if m < 0:
        return (-1)**abs_(m) * np.conjugate(d)
    else:
        return d

# Get the signs of [mu,eta,xi] in each octant
def GetSigns(octant):

    if   octant == 0: return [+1,+1,+1]
    elif octant == 1: return [-1,+1,+1]
    elif octant == 2: return [-1,-1,+1]
    elif octant == 3: return [+1,-1,+1]
    elif octant == 4: return [+1,+1,-1]
    elif octant == 5: return [-1,+1,-1]
    elif octant == 6: return [-1,-1,-1]
    elif octant == 7: return [+1,-1,-1]


def MuEtaXiConvert(mu,eta,xi):

    theta = np.arccos(mu)
    phi = np.arctan2(xi,eta)
    return theta, phi


def D(l,m):
    if m == 0:
        deltam = 1
    else:
        deltam = 0

    return ((-1.)**m) * np.sqrt( (2.-deltam) * ((2.*l + 1.)/(4*np.pi)) * (factorial_(l-m)/factorial_(l+m)))


def Yeven(theta,phi,l,m):
    return D(l,m) * AssocLegendrePoly(np.cos(theta),l,m) * np.cos(m*phi)


def Yodd(theta,phi,l,m):
    return D(l,m) * AssocLegendrePoly(np.cos(theta),l,m) * np.sin(m*phi)


def S4_Theta_Phi():

    N = 4

    MU  = np.zeros(8*N*(N+2)/8)
    ETA = np.zeros(8*N*(N+2)/8)
    XI  = np.zeros(8*N*(N+2)/8)

    # S4 mu,eta,xi
    mu = eta = xi = np.array([0.3500212, 0.8688903])

```

```
111     qq = 0
112     for octant in range(8):
113         signs = GetSigns(octant)
114         for i in range(N/2):
115             for j in range(N/2):
116                 for k in range(N/2):
117                     if np.isclose(mu[i]**2 + eta[j]**2 + xi[k]**2, 1):
118                         MU[qq] = mu[i] * signs[0]
119                         ETA[qq] = eta[i]* signs[1]
120                         XI[qq] = xi[i] * signs[2]
121                         qq += 1
122 
123     return MuEtaXiConvert(MU,ETA,XI)
124 
125 
126 # ----
127 
128 
129 # Get the theta and phi for all 8 octants
130 [theta,phi] = S4_Theta_Phi()
131 
132 for l in range(3):
133     for m in range(l+1):
134         print "l = ", l, ", m = ", m, " qlm = ", (4.*np.pi / 24.) * np.sum(Yeven(theta,phi,l,m))
135         print "l = ", l, ", m = ", m, " slm = ", (4.*np.pi / 24.) * np.sum(Yodd(theta,phi,l,m))
```

```
OUTPUT:

l =  0, m = 0, qlm = 3.54490770181
l =  0, m = 0, slm = 0.0
l =  1, m = 0, qlm = 2.83912495738e-13
l =  1, m = 0, slm = 0.0
l =  1, m = 1, qlm = 2.90655708167e-16
l =  1, m = 1, slm = 5.81311416335e-17
l =  2, m = 0, qlm = 2.18032173099e-07
l =  2, m = 0, slm = 0.0
l =  2, m = 1, qlm = -1.16262283267e-16
l =  2, m = 1, slm = -2.32524566534e-16
l =  2, m = 2, qlm = -2.77665249661e-16
l =  2, m = 2, slm = 0.0
```

Problem 5
What are the major nuclear data libraries and which countries manage them?

⇒ The three major nuclear data libraries are the Evalucated Nuclear Data File (ENDF), Joint Evaluated Fission and Fusion File (JEFF), and Japanese Evaluated Nuclear Data Library (JENDL). ENDF is managed by a Canadian and USA cross section working group consisting of national laboratories, universities, and industry. JEFF is managed by the Nuclear Energy Agency under the Organization for Economic Cooperation and Development (OECD) - which previoulsy consisted of only European member states until 1961. JENDL is managed by the Nuclear Data Center within the Japan Atomic Energy Agency (JAEA) in Japan.