1. Derive the 1st order form of SP_5 with isotropic source and vacuum boundary conditions.

We're going to employ the heuristic derivation of the SP_5 equations. Let's consider the P_N equations for $l' = 0, 1, \ldots$

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

with $\phi_{-1} = 0$ and $\phi_{N+1} = 0$ or $\frac{d}{dx}\phi_{N+1} = 0$.

To generate the SP_N equations from this, we make two transformations to generalize the equation into 3D space. The first is on $\phi_{l'}$. For odd values of l', $\phi_{l'} \rightarrow \vec{\phi}_{l'} = (\phi^x_{l'}, \phi^y_{l'}, \phi^z_{l'})^t$.

The second transformation is to change how we treat the derivative on x. In the even l' equations, the derivative on x is replaced by a divergence operator: $\frac{d}{dx} \rightarrow \nabla\cdot$; in the odd l' equations, that derivative is changed to a gradient: $\frac{d}{dx} \rightarrow \nabla$.

Thus, the first-order form of the SP_N equations is:

$$\begin{gathered}
\nabla\cdot\vec{\phi}_1 + \Sigma_a\phi_0 = s_0 \\
\left(\frac{l'+1}{2l'+1}\right)\nabla\phi_{l'+1} + \left(\frac{l'}{2l'+1}\right)\nabla\phi_{l'-1} + \Sigma_t\vec{\phi}_{l'} = \Sigma_{sl'}\vec{\phi}_{l'} + s_{l'} \quad \text{for odd } l', \\
\left(\frac{l'+1}{2l'+1}\right)\nabla\cdot\vec{\phi}_{l'+1} + \left(\frac{l'}{2l'+1}\right)\nabla\cdot\vec{\phi}_{l'-1} + \Sigma_t\phi_{l'} = \Sigma_{sl'}\phi_{l'} + s_{l'} \quad \text{for even } l' > 0.
\end{gathered}$$

Assuming an isotropic source (for $l' > 0, s_{l'} = 0$), we can write the SP_5 equations in their first order form as follows:

$$\begin{gathered}
\nabla\cdot\vec{\phi}_1 + \Sigma_a\phi_0 = s_0 \\
\frac{2}{3}\nabla\phi_2 + \frac{1}{3}\nabla\phi_0 + [\Sigma_t - \Sigma_{s1}]\,\vec{\phi}_1 = 0 \\
\frac{3}{5}\nabla\cdot\vec{\phi}_3 + \frac{2}{5}\nabla\cdot\vec{\phi}_1 + [\Sigma_t - \Sigma_{s2}]\,\phi_2 = 0 \\
\frac{4}{7}\nabla\phi_4 + \frac{3}{7}\nabla\phi_2 + [\Sigma_t - \Sigma_{s3}]\,\vec{\phi}_3 = 0 \\
\frac{5}{9}\nabla\cdot\vec{\phi}_5 + \frac{4}{9}\nabla\cdot\vec{\phi}_3 + [\Sigma_t - \Sigma_{s4}]\,\phi_4 = 0 \\
\frac{5}{11}\nabla\phi_4 + [\Sigma_t - \Sigma_{s5}]\,\vec{\phi}_5 = 0
\end{gathered}$$

Note that $\phi_{-1} = \phi_{N+1=6} = 0$.

We want to include vacuum boundary conditions; for this, we consider the Marshak boundary conditions for the P_N equations. The half-range moments of ψ, the angular flux, are equated with the incoming angular flux at the boundary, Ψ^-. For an Nth order expansion where N is odd, there are $(N+1)/2$ boundary conditions:

$$2\pi \int_0^{\pm 1} P_{2m-1}(\mu)\psi^d \mu = \sum_{n=0}^{N} \frac{2n+1}{2}\phi_n(x) \int_0^{\pm 1} P_{2m-1}(\mu)P_n(x)d\mu$$
$$= 2\pi \int_0^{\pm 1} P_{2m-1}(\mu)\Psi^-(x,\mu)d\mu$$

for x = 0, X (the boundaries) and $m = 1, 2, \ldots, (N+1)/2$.

To adjust these to the SP_N equations, we make some simple replacements. We replace the ϕ_n with the SP_N unknowns and μ with $\hat{n}\cdot\hat{\Omega}$ ($\hat{n}$ is the unit inward normal to the boundary).

$$\sum_{n \text{ even}}^{N} \frac{2n+1}{4\pi}\phi_n(\vec{r}) \int_{\hat{n}\cdot\hat{\Omega}>0} P_{2m-1}(\hat{n}\cdot\hat{\Omega})P_n(\hat{n}\cdot\hat{\Omega})d^2\hat{\Omega} +$$
$$\sum_{n \text{ odd}}^{N} \frac{2n+1}{4\pi}\hat{n}\cdot\vec{\phi}_n(\vec{r}) \int_{\hat{n}\cdot\hat{\Omega}>0} P_{2m-1}(\hat{n}\cdot\hat{\Omega})P_n(\hat{n}\cdot\hat{\Omega})d^2\hat{\Omega} =$$
$$\int_{\hat{n}\cdot\hat{\Omega}>0} P_{2m-1}(\hat{n}\cdot\hat{\Omega})\Psi^-(\hat{n}\cdot\hat{\Omega})d^2\hat{\Omega}$$

for $\vec{r} \in d\Gamma$ and $m = 1, 2, \ldots, (N+1)/2$.

These boundary conditions are a collection of 1-D Marshak BCs where the SP_N unknowns can be interpreted as components of a Legendre polynomial expansion.

For more information about the Marshak BCs, I referred to the following:
http://www.tandfonline.com/doi/pdf/10.1080/00411450.2010.535088?needAccess=true (http://www.tandfonline.com/doi/pdf/10.1080/00411450.2010.535088?needAccess=true)
http://www.casl.gov/docs/CASL-U-2014-0352-000.pdf (http://www.casl.gov/docs/CASL-U-2014-0352-000.pdf)

2. Consider the integral

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

The LQ_N quadrature set is given in Figure 1. 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

Level	n	μ_n	w_n^b
S_4	1	0.3500212	0.3333333
	2	0.8688903	
S_6	1	0.2666355	0.1761263
	2	0.6815076	0.1572071
	3	0.9261808	

In this problem, the quadrature set is taking the place of integration. The function $\hat{\Omega}$ is evaluated at the quadrature points and multiplied by a weight. This is repeated for each octant.

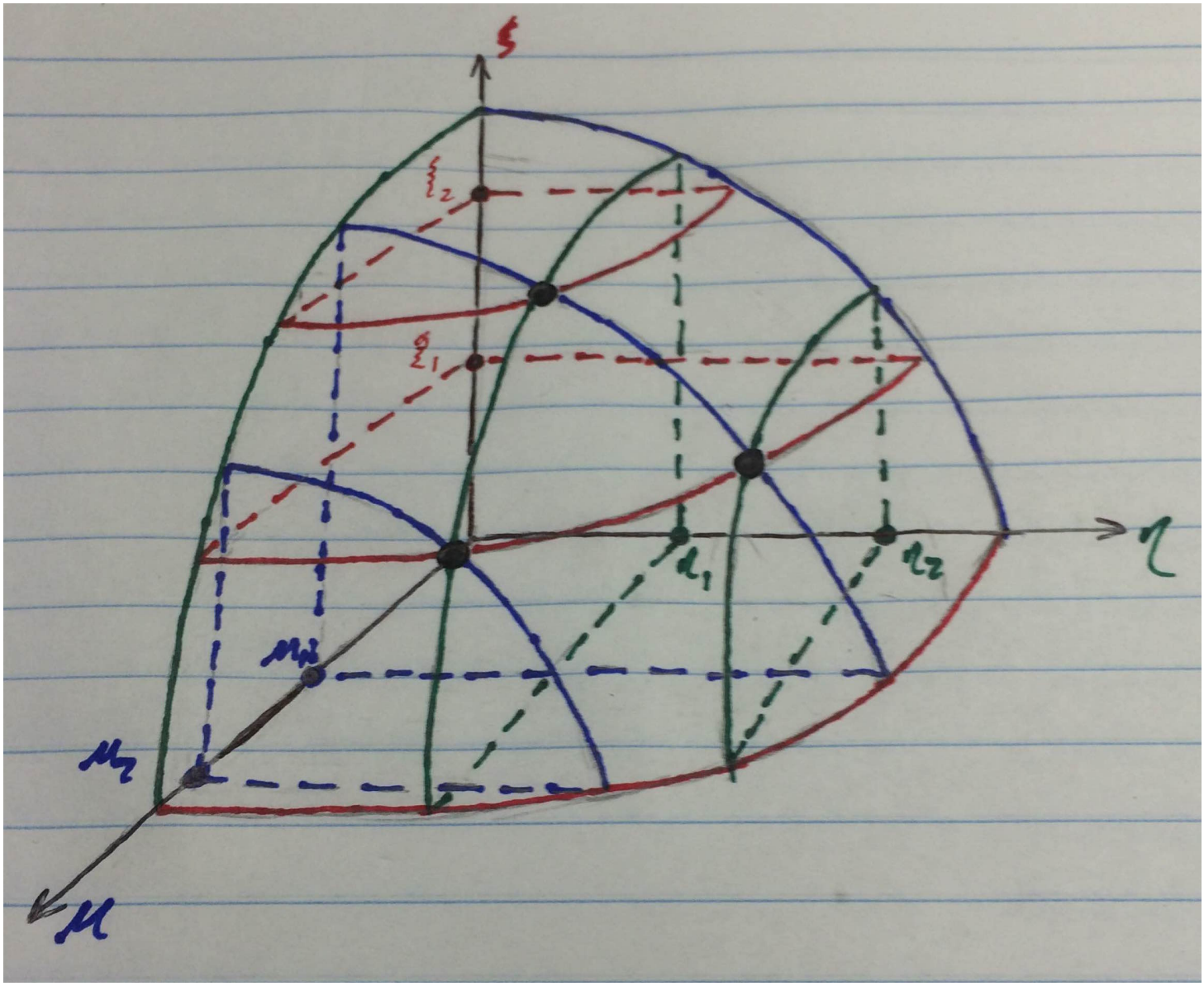
$$\int_{4\pi} d\hat{\Omega}\, \hat{\Omega} = \sum_i \omega_i \Omega_i$$

In this case,

$$\hat{\Omega} = \sqrt{\mu^2 + \eta^2 + \xi^2}$$

Here, because we are considering the level symmetric case for S_4 , we know that $\mu_1 = \eta_1 = \xi_1$ and $\mu_2 = \eta_2 = \xi_2$. However, what we don't know is what combination of points (μ, η, ξ) we should put together to get the proper result. In addition, while we know that the magnitude of each is the same, we also must consider the sign of each depending on what octant we're considering.

In S_4 , each quadrature point gets the same weight. In the octant where all μ, η, ξ are positive, the points that should be chosen are: (μ_1, η_1, ξ_2), (μ_2, η_1, ξ_1), and (μ_1, η_2, ξ_1). This was determined by creating a sketch similar to that in Lewis and Miller, pg. 159. This sketch is shown below:



Each of these points gets a weight of $\frac{1}{3}$. For the purposes of brevity, because all angle cosines of equal level are the same in the level symmetric quadrature, we will refer to the points as μ_i.

For this octant, the evaluation is as follows:

$$\frac{1}{3}\sqrt{\mu_1^2+\eta_1^2+\xi_2^2}+\frac{1}{3}\sqrt{\mu_2^2+\eta_1^2+\xi_1^2}+\frac{1}{3}\sqrt{\mu_1^2+\eta_2^2+\xi_1^2}$$

This should be repeated in the 7 remaining octants. In each, the sign of one of the angles will change, which will change the result. Octants will be by the label (+/- +/- +/-), corresponding to which sign of the direction cosines in that octant. The octant above was (+++).

The math is carried out explicitly in the code below.

In [180]:
```
import math
mu1 = 0.3500212
mu2 = 0.8688903
eta1 =0.3500212
eta2 = 0.8688903
xi1 = 0.3500212
xi2 = 0.8688903
w = 0.333333333
lqn4 = {}
# The octant we solved above
lqn4['+++'] = w*math.sqrt(mu1**2+eta1**2+xi2**2)+\
    w*math.sqrt(mu2**2+eta1**2+xi1**2)+\
    w*math.sqrt(mu1**2+eta2**2+xi1**2)
# In this next octant, mu is negative while eta and xi are still posit
ive
lqn4['-++'] = w*math.sqrt((-mu1)**2+eta1**2+xi2**2)+\
    w*math.sqrt((-mu2)**2+eta1**2+xi1**2)+\
    w*math.sqrt((-mu1)**2+eta2**2+xi1**2)
# In this octant, eta is negative while mu and xi are still positive
lqn4['+-+'] = w*math.sqrt(mu1**2+(-eta1)**2+xi2**2)+\
    w*math.sqrt(mu2**2+(-eta1)**2+xi1**2)+\
    w*math.sqrt(mu1**2+(-eta2)**2+xi1**2)
# In this octant, xi is negative while mu and xi are still positive
lqn4['++-'] = w*math.sqrt(mu1**2+eta1**2+(-xi2)**2)+\
    w*math.sqrt(mu2**2+eta1**2+(-xi1)**2)+\
    w*math.sqrt(mu1**2+eta2**2+(-xi1)**2)
# Now, mu and eta are negative while xi is positive.
lqn4['--+'] = w*math.sqrt(+(-mu1)**2+(-eta1)**2+xi2**2)+\
    w*math.sqrt((-mu2)**2+(-eta1)**2+xi1**2)+\
    w*math.sqrt((-mu1)**2+(-eta2)**2+xi1**2)
# Here, mu and xi are negative while eta is positive.
lqn4['-+-'] = w*math.sqrt(+(-mu1)**2+eta1**2+(-xi2)**2)+\
    w*math.sqrt((-mu2)**2+eta1**2+(-xi1)**2)+\
    w*math.sqrt((-mu1)**2+eta2**2+(-xi1)**2)
# mu is positive while eta and xi are negative
lqn4['+--'] = w*math.sqrt(mu1**2+(-eta1)**2+(-xi2)**2)+\
    w*math.sqrt(mu2**2+(-eta1)**2+(-xi1)**2)+\
    w*math.sqrt(mu1**2+(-eta2)**2+(-xi1)**2)
# # Finally, all are negative
lqn4['---'] = w*math.sqrt((-mu1)**2+(-eta1)**2+(-xi2)**2)+\
    w*math.sqrt((-mu2)**2+(-eta1)**2+(-xi1)**2)+\
    w*math.sqrt((-mu1)**2+(-eta2)**2+(-xi1)**2)
```

To get the final result, we sum all the values from each octant

In [181]:
```
print(sum(lqn4.values()))
```

```
8.00000012933
```

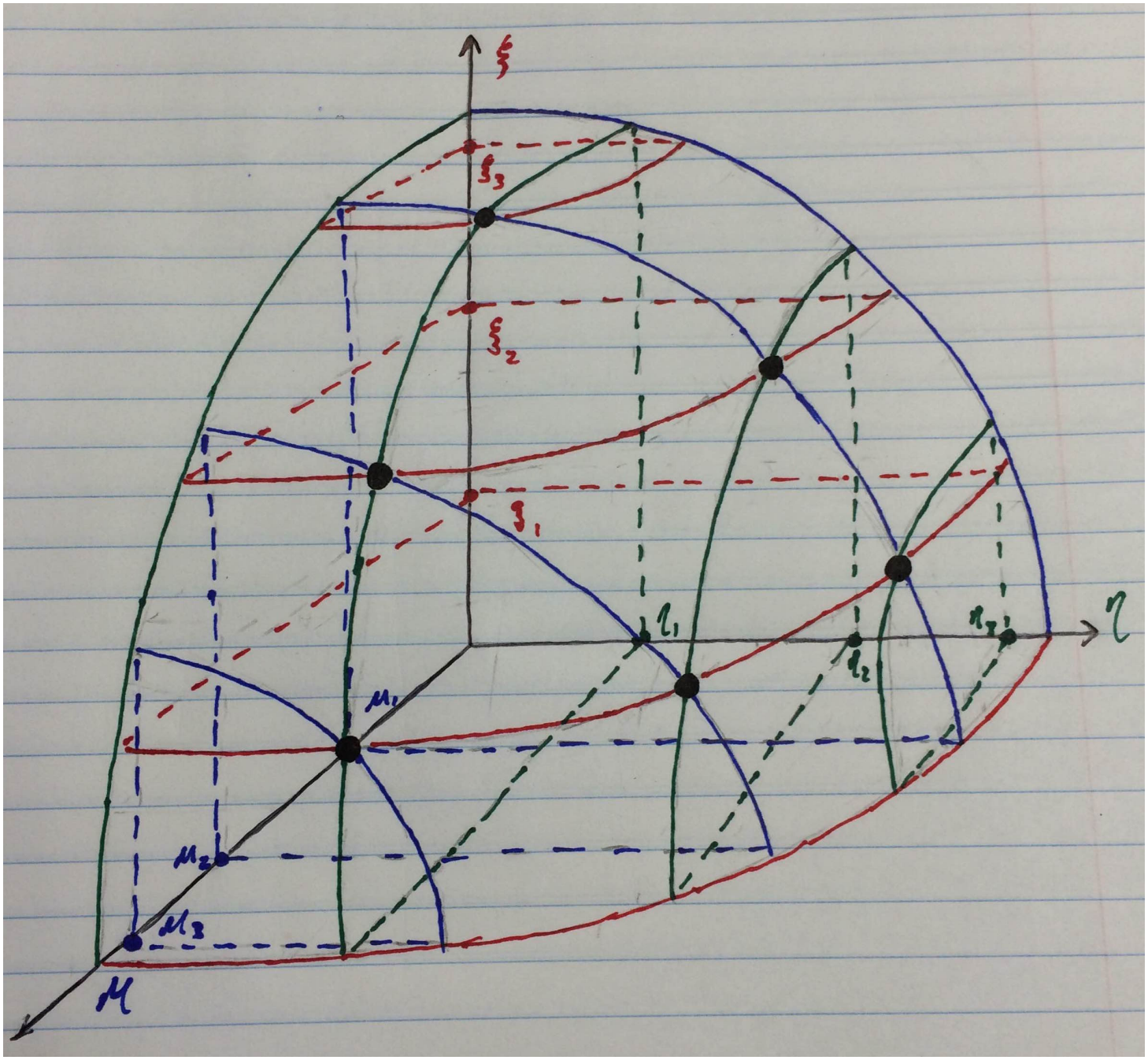
Clearly, however, this should be 4π. Thus, we multiply the sum by a normalization constant, $c = \frac{4\pi}{8}$.

In [182]:
```
c = 4*math.pi/8
print(c*sum(lqn4.values()))
```

```
12.5663708175
```

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

The combination of points can be identified by drawing another picture similar to that in Lewis and Miller.



As shown above, the point combinations and their associated weights are:

- $(\mu_1, \eta_1, \xi_3); w = w_1$
- $(\mu_1, \eta_2, \xi_2); w = w_2$
- $(\mu_1, \eta_3, \xi_1); w = w_1$
- $(\mu_2, \eta_1, \xi_2); w = w_2$
- $(\mu_2, \eta_2, \xi_1); w = w_2$
- $(\mu_3, \eta_1, \xi_1); w = w_1$

We can utilize the same explicit calculation method as before. However, more points will have to be added to the summation within each octant. In addition, this time we have to consider different weights.

In [186]:
```
# Define everything as lists; slightly prettier that way
# (still ugly tho)
# 0th index left blank because I want things to correspond
# to the 1, 2, and 3 indices used above.
mu = ['lol', 0.2666355, 0.6815076, 0.9261808]
eta = ['rofl', 0.2666355, 0.6815076, 0.9261808]
xi = ['lmao', 0.2666355, 0.6815076, 0.9261808]
w = ['herpderp', 0.1761263, 0.1572071]
lqn6 = {}
```

In [196]:
```
lqn6['+++'] = w[1]*math.sqrt(mu[1]**2+eta[1]**2+xi[3]**2) + w[2]*math.
sqrt(mu[1]**2+eta[2]**2+xi[2]**2)+\
    w[1]*math.sqrt(mu[1]**2+eta[3]**2+xi[1]**2) + w[2]*math.sqrt(mu[2]
**2+eta[1]**2+xi[2]**2)+\
    w[2]*math.sqrt(mu[2]**2+eta[2]**2+xi[1]**2) + w[1]*math.sqrt(mu[3]
**2+eta[1]**2+xi[1]**2)
# In this next octant, mu is negative while eta and xi are still posit
ive
lqn6['-++'] = w[1]*math.sqrt((-mu[1])**2+eta[1]**2+xi[3]**2) + w[2]*ma
th.sqrt((-mu[1])**2+eta[2]**2+xi[2]**2)+\
    w[1]*math.sqrt((-mu[1])**2+eta[3]**2+xi[1]**2) + w[2]*math.sqrt((-
mu[2])**2+eta[1]**2+xi[2]**2)+\
    w[2]*math.sqrt((-mu[2])**2+eta[2]**2+xi[1]**2) + w[1]*math.sqrt((-
mu[3])**2+eta[1]**2+xi[1]**2)
# In this octant, eta is negative while mu and xi are still positive
lqn6['+-+'] = w[1]*math.sqrt(mu[1]**2+(-eta[1])**2+xi[3]**2) + w[2]*ma
th.sqrt(mu[1]**2+(-eta[2])**2+xi[2]**2)+\
    w[1]*math.sqrt(mu[1]**2+(-eta[3])**2+xi[1]**2) + w[2]*math.sqrt(mu
[2]**2+(-eta[1])**2+xi[2]**2)+\
    w[2]*math.sqrt(mu[2]**2+(-eta[2])**2+xi[1]**2) + w[1]*math.sqrt(mu
[3]**2+(-eta[1])**2+xi[1]**2)
# In this octant, xi is negative while mu and xi are still positive
lqn6['++-'] = w[1]*math.sqrt(mu[1]**2+eta[1]**2+(-xi[3])**2) + w[2]*ma
th.sqrt(mu[1]**2+eta[2]**2+(-xi[2])**2)+\
    w[1]*math.sqrt(mu[1]**2+eta[3]**2+(-xi[1])**2) + w[2]*math.sqrt(mu
[2]**2+eta[1]**2+(-xi[2])**2)+\
    w[2]*math.sqrt(mu[2]**2+eta[2]**2+(-xi[1])**2) + w[1]*math.sqrt(mu
[3]**2+eta[1]**2+(-xi[1])**2)
# Now, mu and eta are negative while xi is positive.
lqn6['--+'] = w[1]*math.sqrt((-mu[1])**2+(-eta[1])**2+xi[3]**2) + w[2]
*math.sqrt((-mu[1])**2+(-eta[2])**2+xi[2]**2)+\
    w[1]*math.sqrt((-mu[1])**2+(-eta[3])**2+xi[1]**2) + w[2]*math.sqrt
((-mu[2])**2+(-eta[1])**2+xi[2]**2)+\
    w[2]*math.sqrt((-mu[2])**2+(-eta[2])**2+xi[1]**2) + w[1]*math.sqrt
((-mu[3])**2+(-eta[1])**2+xi[1]**2)
    # Here, mu and xi are negative while eta is positive.
lqn6['-+-'] = w[1]*math.sqrt((-mu[1])**2+eta[1]**2+(-xi[3])**2) + w[2]
*math.sqrt((-mu[1])**2+eta[2]**2+(-xi[2])**2)+\
    w[1]*math.sqrt((-mu[1])**2+eta[3]**2+(-xi[1])**2) + w[2]*math.sqrt
((-mu[2])**2+eta[1]**2+(-xi[2])**2)+\
    w[2]*math.sqrt((-mu[2])**2+eta[2]**2+(-xi[1])**2) + w[1]*math.sqrt
((-mu[3])**2+eta[1]**2+(-xi[1])**2)
```

```
# mu is positive while eta and xi are negative
lqn6['+--'] = w[1]*math.sqrt(mu[1]**2+(-eta[1])**2+(-xi[3])**2) + w[2]
*math.sqrt(mu[1]**2+(-eta[2])**2+(-xi[2])**2)+\
    w[1]*math.sqrt(mu[1]**2+(-eta[3])**2+(-xi[1])**2) + w[2]*math.sqrt
(mu[2]**2+(-eta[1])**2+(-xi[2])**2)+\
    w[2]*math.sqrt(mu[2]**2+(-eta[2])**2+(-xi[1])**2) + w[1]*math.sqrt
(mu[3]**2+(-eta[1])**2+(-xi[1])**2)
# Finally, all are negative
lqn6['---'] = w[1]*math.sqrt((-mu[1])**2+(-eta[1])**2+(-xi[3])**2) + w
[2]*math.sqrt((-mu[1])**2+(-eta[2])**2+(-xi[2])**2)+\
    w[1]*math.sqrt((-mu[1])**2+(-eta[3])**2+(-xi[1])**2) + w[2]*math.s
qrt((-mu[2])**2+(-eta[1])**2+(-xi[2])**2)+\
    w[2]*math.sqrt((-mu[2])**2+(-eta[2])**2+(-xi[1])**2) + w[1]*math.s
qrt((-mu[3])**2+(-eta[1])**2+(-xi[1])**2)

# Again, we must normalize by the same constant
print(c*sum(lqn6.values()))
```

```
12.5663717764
```

Performing this integral with S_4 and S_6 quadrature, we see that the result is the same, as it should be equal to 4π.

(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.

In [238]:
```
def index_sum(order):
    sum = (order/2)+2
    return(sum)

def num_points(N):
    return(int(N*(N+2)/8))

import itertools
import numpy as np
def det_mu_index(N):
    # determine which combinations of mu, eta, and xi are needed
    # for the quadrature
    x = list(itertools.combinations_with_replacement(list(range(1,(N/2
)+1)),3))
    x = list(set([i for i in x if sum(i) == index_sum(N)]))
    poop=[]
    for i in x:
        a = list(set(itertools.permutations(i)))
        for j in a:
            poop.append(list(j))
    # returning the sorted indices gives the points in a specific
    # order that allows one to determine the associated weights.
    return(sorted(poop))
```

In [270]:
```
# This function initializes the data for the quadrature integration
# here is all the data from LM and the tables, etc.
import math
def initialize_data(N):
    # We can predefine the mu values and weights according to the tabl
e.
    # Depending on the requested order, the appropriate array is chose
n.
    mu_n = {}
    mu_n[2] = np.array([1/math.sqrt(3.0)])
    mu_n[4] = np.array([0.3500212, 0.8688903])
    mu_n[6] = np.array([0.2666355, 0.6815076, 0.9261808])
    mu_n[8] = np.array([0.2182179, 0.5773503, 0.7867958, 0.9511897])
    mu_n[12] = np.array([0.1672126, 0.4595476, 0.6280191, 0.7600210, \
                         0.8722706, 0.9716377])
    mu_n[16] = np.array([0.1389568, 0.3922893, 0.5370966, 0.6504264, \
                         0.7467506, 0.8319966, 0.9092855, 0.9805009])

    #pre specify weights
    w = {}
    w[2] = np.array([1.0])
    w[4] = np.array([0.3333333])
    w[6] = np.array([0.1761263, 0.1572071])
    w[8] = np.array([0.1209877, 0.0907407, 0.0925926])
    w[12] = np.array([0.0707626, 0.0558811, 0.0373377, 0.0502819, 0.02
58513])
    w[16] = np.array([0.0489872, 0.0413296, 0.0212326, 0.0256207, 0.03
60486, \
```

```
                    0.0144589, 0.0344958, 0.0085179])
    # I couldn't (read: didn't have the patience) to figure out an
    # algorithm to do determine which weights go with which points,
    # so I hard coded it.
    weightIndex = {}
    weightIndex[2] = np.array([1])
    weightIndex[4] = np.array([1, 1, 1])
    weightIndex[6] = np.array([1, 2, 1, 2, 2, 1])
    weightIndex[8] = np.array([1, 2, 2, 1, 2, 3, 2, 2, 2, 1])
    weightIndex[12] = np.array([1, 2, 3, 3, 2, 1, 2, 4, 5, 4, 2, 3, 5,
5, 3, \
                                3, 4, 3, 2, 2, 1])
    weightIndex[16] = np.array([1, 2, 3, 4, 4, 3, 2, 1, 2, 5, 6, 7, 6,
5, 2, \
                                3, 6, 8, 8, 6, 3, 4, 7, 8, 7, 4, 4, 6,
6, 4, \
                                3, 5, 3, 2, 2, 1])

    # this will control the sign of mu in each octant
    octants = {}
    octants[1] = np.array([1, 1, 1])
    octants[2] = np.array([-1, 1, 1])
    octants[3] = np.array([1, -1, 1])
    octants[4] = np.array([1, 1, -1])
    octants[5] = np.array([1, -1, -1])
    octants[6] = np.array([-1, 1, -1])
    octants[7] = np.array([-1, -1, 1])
    octants[8] = np.array([-1, -1, -1])

    return(mu_n[N], w[N], weightIndex[N], octants)
```

In [271]:
```
def integrate_lqn(N, fxn, prnt='off', cnorm=4*math.pi/8):
    # IMPLEMENT INTEGRATION BOUNDS????

    # N is the quadrature order
    # The function is the one that should be integrated;
    # for example, in (a) and (b) the function was f = sqrt(mu^2+eta^2
+xi^2)
    # The function is evaluated at the quadrature points and weighted.
    # *** The function MUST take in three values! ***
    # The normalization constant can be input but is by default 4pi/8
    # Call in appropriate data
    muVals, weights, wtIndx, octants = initialize_data(N)
    muIdx = det_mu_index(N)

    # number of octants (if i can figure out how to loop this guy)
    n_octants = 8

    #running total
    total = 0.0

    # Need to cover all octants: +++, +-+, ++-, +--, -+-, --+, ---
    for o in range(1,n_octants+1):
        partialSum = [0]*len(muIdx)
        for i in range(0, len(muIdx)):
            mu = [octants[o][0]*muVals[muIdx[i][0]-1],\
                  octants[o][1]*muVals[muIdx[i][1]-1],\
                  octants[o][2]*muVals[muIdx[i][2]-1]]
            partialSum[i] = weights[wtIndx[i]-1]*fxn(mu[0], mu[1], mu[
2])
        if(prnt == 'on'):
            print('octant = '+str(octants[o])+\
                  ';  partial sum = '+str(sum(partialSum)))
        total+=sum(partialSum)

    return(cnorm*total)
```

The above is the function that performs LQ_N quadrature integration over an input function. The function itself is fairly short but relies on a lot of secondary code. I wasn't sure how to implement the algorithms for some parts - for instance, the weights that should go with each μ as given in L&M - so I just hard coded that as 'input data'.

We can test the integration for various functions. For example, in the case of (a) and (b), where we want to solve $\int_{4\pi} d\hat{\Omega}\,\hat{\Omega}$:

In [272]:
```
# First
def Omegahat(mu, eta, xi):
    return(math.sqrt(mu**2+eta**2+xi**2))

print(integrate_lqn(2, Omegahat))
print(integrate_lqn(4, Omegahat))
print(integrate_lqn(6, Omegahat))
print(integrate_lqn(8, Omegahat))
print(integrate_lqn(12, Omegahat))
print(integrate_lqn(16, Omegahat))
```

```
12.5663706144
12.5663695734
12.5663717764
12.5663694807
12.5663729919
12.5663619448
```

Not bad! We can also try some other functions too:

In [273]:
```
def sum_coords(mu, eta, xi):
    # just return the sum of mu, eta, and xi
    return(mu+eta+xi)

# Should return 0 over the unit sphere, because opposite quadrants
# will produce opposite results that cancel out.
print(integrate_lqn(2, sum_coords))
print(integrate_lqn(4, sum_coords))
print(integrate_lqn(6, sum_coords))
print(integrate_lqn(8, sum_coords))
print(integrate_lqn(12, sum_coords))
print(integrate_lqn(16, sum_coords))
```

```
0.0
0.0
0.0
0.0
0.0
3.48786849801e-16
```

In [274]:
```
def exp_coords(mu, eta, xi):
    return(math.exp(mu+eta+xi))

print(integrate_lqn(2, exp_coords, prnt='on'))
print(integrate_lqn(4, exp_coords, prnt='on'))
print(integrate_lqn(6, exp_coords, prnt='on'))
print(integrate_lqn(8, exp_coords, prnt='on'))
print(integrate_lqn(12, exp_coords, prnt='on'))
print(integrate_lqn(16, exp_coords, prnt='on'))
```

```
octant = [1 1 1];  partial sum = 5.65223367403
```

```
octant = [-1  1  1];  partial sum = 1.78131217411
octant = [ 1 -1  1];  partial sum = 1.78131217411
octant = [ 1  1 -1];  partial sum = 1.78131217411
octant = [ 1 -1 -1];  partial sum = 0.561383913799
octant = [-1  1 -1];  partial sum = 0.561383913799
octant = [-1 -1  1];  partial sum = 0.561383913799
octant = [-1 -1 -1];  partial sum = 0.176921206318
20.1961103034
octant = [1 1 1];  partial sum = 4.80152031473
octant = [-1  1  1];  partial sum = 1.87105464566
octant = [ 1 -1  1];  partial sum = 1.87105464566
octant = [ 1  1 -1];  partial sum = 1.87105464566
octant = [ 1 -1 -1];  partial sum = 0.674257760445
octant = [-1  1 -1];  partial sum = 0.674257760445
octant = [-1 -1  1];  partial sum = 0.674257760445
octant = [-1 -1 -1];  partial sum = 0.208267326691
19.8638581593
octant = [1 1 1];  partial sum = 4.68018662615
octant = [-1  1  1];  partial sum = 1.88934280685
octant = [ 1 -1  1];  partial sum = 1.88934280685
octant = [ 1  1 -1];  partial sum = 1.88934280685
octant = [ 1 -1 -1];  partial sum = 0.693753335189
octant = [-1  1 -1];  partial sum = 0.693753335189
octant = [-1 -1  1];  partial sum = 0.693753335189
octant = [-1 -1 -1];  partial sum = 0.215212877768
19.862229354
octant = [1 1 1];  partial sum = 4.6266499375
octant = [-1  1  1];  partial sum = 1.89590450382
octant = [ 1 -1  1];  partial sum = 1.89590450382
octant = [ 1  1 -1];  partial sum = 1.89590450382
octant = [ 1 -1 -1];  partial sum = 0.703816335469
octant = [-1  1 -1];  partial sum = 0.703816335469
octant = [-1 -1  1];  partial sum = 0.703816335469
octant = [-1 -1 -1];  partial sum = 0.218878816862
19.8622346038
octant = [1 1 1];  partial sum = 4.58578051913
octant = [-1  1  1];  partial sum = 1.90096227742
octant = [ 1 -1  1];  partial sum = 1.90096227742
octant = [ 1  1 -1];  partial sum = 1.90096227742
octant = [ 1 -1 -1];  partial sum = 0.711416566584
octant = [-1  1 -1];  partial sum = 0.711416566584
octant = [-1 -1  1];  partial sum = 0.711416566584
octant = [-1 -1 -1];  partial sum = 0.221777848321
19.8622403015
octant = [1 1 1];  partial sum = 4.56859751347
octant = [-1  1  1];  partial sum = 1.90298858559
octant = [ 1 -1  1];  partial sum = 1.90298858559
octant = [ 1  1 -1];  partial sum = 1.90298858559
octant = [ 1 -1 -1];  partial sum = 0.714689524669
octant = [-1  1 -1];  partial sum = 0.714689524669
octant = [-1 -1  1];  partial sum = 0.714689524669
octant = [-1 -1 -1];  partial sum = 0.223051815501
19.8622226462
```

Interesting result! Some functions don't depend on the quadrature at all. For example, for some functions integration over the unit sphere is guaranteed to give 0 or 1 no matter what points you use because the results in different octants will cancel out or equal one by design.

However, it seems as though some functions (like the exp_coords function I made up) actually do depend on quadrature. It seems like as N increases, the result trends toward some value, which I might assume to be the true value if the integral were done analytically.

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 a simplified form of the transport equation. Angular dependence is removed and so only the scalar flux is given as a solution. In this way, it is not very complex nor accurate, but it is much more straightforward to solve than the full TE (so run time is short). The assumptions used to derive the diffusion equation from the TE are applicable when the solution is not near:

- a void
- boundaries
- sources
- strong absorbers

The diffusion equation can take into account discretization in energy (few-group) to increase accuracy while still providing a simple and fast solution.

Deterministic solutions to the TE involve discretizing all independent variables - space, direction (discrete ordinates or spherical harmonics) and energy (multi-group). The size of the problem and the quality and behavior of the solutions are governed by the discretization. As such, the solutions can range from simple (coarse meshes in all variables, simplifications such as symmetry, etc) to very difficult (thousands of spatial mesh points and complex expansions of energy and angle). Due to these complexities, the run time is longer than anything involving the diffusion equation, but in return one gets a much more accurate result.

In Monte Carlo methods, the physics of every interaction of every particle are sampled in order to determine statistical quantities of interest. Because MC relies on sampling, the independent variables (space, energy, angle) can be treated as continuous, and, for well-posed problems, can result in a very accurate solution. The MC methodology is fairly simple and the process is easy to parallelize, but the inputs can be complex. The run time can be very long because enough samples need to be taken so the results can be statistically significant. If an MC problem is ill-posed or too few particles are simulated, the results can be very inaccurate.

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

Deterministic methods offer (relatively) fast and global solutions to the TE. In addition, the solution quality is the same over the entire simulated phase space (space, energy, and angle). Inputs can be simple.

However, the discretization governs the solution quality, and fine meshes can be complicated. Depending on solution strategy, numerical anamolies can arise. For example, ray effects occur when using a discrete ordinates methods because the flux is only solved along specific angles and can result in weird flux shapes. In addition, the solution contains truncation error. Contrary to MC methods, deterministic methods can be difficult to parallelize.

4. Write a function that generates the associated Legendre Polynomials:

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

In [299]:
```
# function to perform combinatorics; i.e. "n choose r"
# http://stackoverflow.com/questions/4941753/is-there-a-math-ncr-funct
ion-in-python
import operator as op
def ncr(n, r):
    r = min(r, n-r)
    if(r == 0):
        return 1
    numer = reduce(op.mul, xrange(n, n-r, -1))
    denom = reduce(op.mul, xrange(1, r+1))
    return(numer//denom)

def det_bin_coeffs(l):
    # This function determines the coefficients of the polynomial
    # (x^2-1)^l according to the binomial coefficient
    # https://en.wikipedia.org/wiki/Binomial_coefficient
    nterms = int(2*abs(l)+1)
    coeffs = [0]*nterms
    for i in range(0, nterms):
        if(i%2==1):
            # no odd exponents in polynomial
            continue
        else:
            # if exponent is even, assign it a coefficient according t
o
            # the binomial series
            if((i/2)%2==1):
                coeffs[i]=-ncr(int(l), i/2)
            else:
                coeffs[i]=ncr(int(l), i/2)
    # the above generated the coeffs from highest to lowest. For the
    # numpy differentiation, we need them from lowest to highest; thus
    # we return the reverse.
    return(coeffs[::-1])

import math
```

```
import numpy as np
from numpy import polyval
from numpy.polynomial import polynomial as P

def Plm(l, m):
    l=float(l)
    m=float(m)
    def assoc_leg_poly(x):
        # P = value of fxn @ x
        # make l and m floats
        a = ((-1)**m)/(2**l*math.factorial(l))
        b = (1-x**2)**(m/2)
        c = np.polyval(P.polyder(det_bin_coeffs(l),l+m)[::-1], x)
        # print(str(a)+'   '+str(b)+'   '+str(c))
        return(a*b*c)
    # return the function that takes x input and yields Plm
    return(assoc_leg_poly)

# NOTE: Python defines 0**0 = 1.0
```

Use this function in a function that generates spherical harmonics:

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

The spherical harmonics will be split into $m = 0$, $m =$ even, and $m =$ odd components. This moves the imaginary parts under the square root so we can put them in code. Because they will be later dropped (since our solution is only real), we can incorporate logic to neglect them.

In [410]:
```
def kron_delt(a,b):
    if(a == b):
        return(1.0)
    else:
        return(0.0)

import cmath
def sph_harmonic(l, m):
    l = float(l)
    m = float(m)
    # Test that l and m are appropriately specified?
    def Ylm(theta, phi):
        a = (-1)**m
        b = math.sqrt((2*l+1)*\
            math.factorial(l-m)\
            /(4*math.pi)/math.factorial(l+m))
        try:
            c = Plm(l,m)(math.cos(theta))*cmath.exp(cmath.sqrt(-1)*m*p
hi)
        except:
            ZeroDivisionError
            c = 0.0
        return(a*b*c)
    return(Ylm)
```

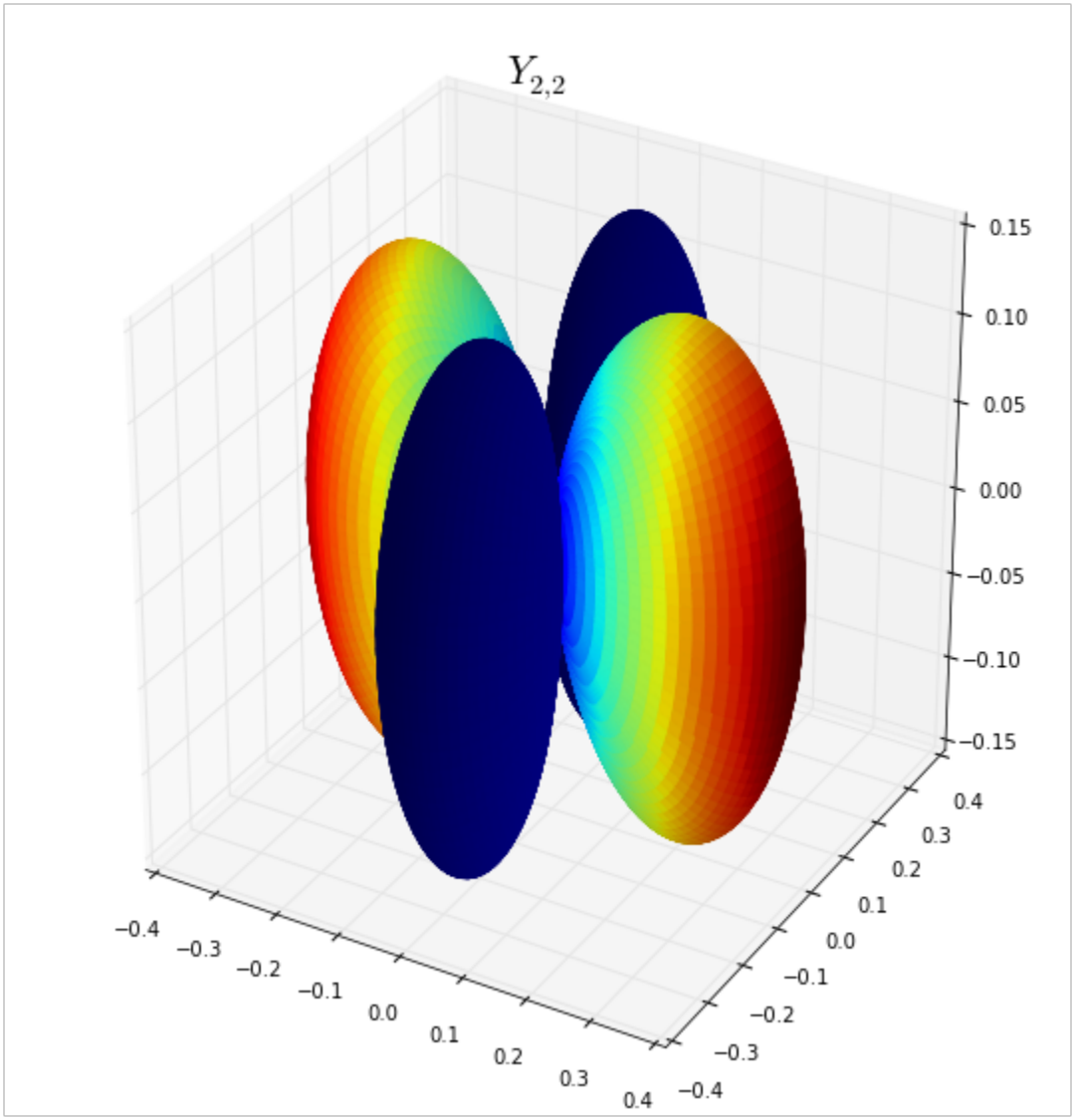
(a) Generate and plot the following $l = 0, 1, 2$ for $-l \leq m \leq l$ (recall we can relate the negative m to positive m values). You will need to discretize θ and φ fairly finely (30 increments in each to start to get a sense of the shape of the harmonics).

In [355]:
```
# NOTE: Plotting these in python is hard. I mimicked the code on this
# site to do this; some of it, like the actual plotting code, I just
# copied and pasted and then coded my stuff around it.
# http://balbuceosastropy.blogspot.com/2015/06/spherical-harmonics-in-
python.html

# FOR TESTING
# -------------------------------------
import scipy
from scipy.special import sph_harm
# -------------------------------------

# Set up the space in which the spherical harmonics are plotted
# This is the mesh grid that is used to plot the points
phi, theta = np.mgrid[0:2*np.pi:200j, 0:np.pi:100j]
# The theta and phi vectors is used in the loop to calculate the value
s
# of the spherical harmonic; I know it's confusing to have both, but
# the plotting requires the mesh grid and my spherical harmonic functi
on
```

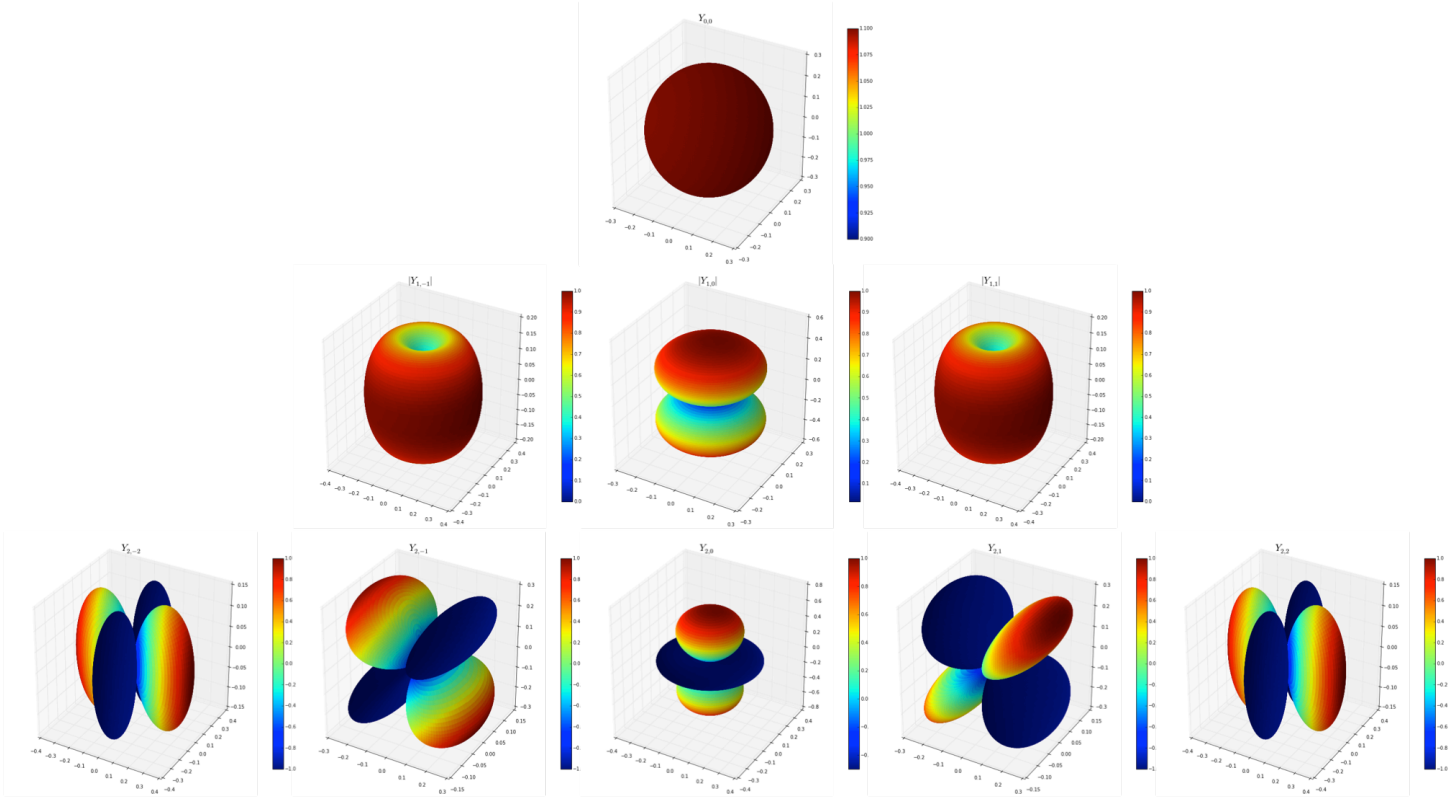
```
# doesn't take arrays as inputs.
thetaVEC = np.linspace(0, np.pi, 100)
phiVEC = np.linspace(0, 2*np.pi, 200)
# Y is defined as a dictionary. The keys of the dictionary are to
# be 'lm' corresponding to the degree and order of the spherical
# harmonic. The value in the dictionary is an array that holds the
# points.

# We do this four times to compare the results of slightly different
# methodologies
Y = {}
Y['real'] = {}
Y['abs'] = {}
Y['builtin abs'] = {}
Y['builtin real'] = {}
# Before we get into the loop, we have to make three
# more dictionaries; these will correspond to the
# cartesian points for each spherical harmonic lm.
xpts = {}
xpts['real'] = {}
xpts['abs'] = {}
xpts['builtin abs'] = {}
xpts['builtin real'] = {}

ypts = {}
ypts['real'] = {}
ypts['abs'] = {}
ypts['builtin abs'] = {}
ypts['builtin real'] = {}

zpts = {}
zpts['real'] = {}
zpts['abs'] = {}
zpts['builtin abs'] = {}
zpts['builtin real'] = {}
```

In [363]:
```
# we're interesed in l between 0 and 2
l = [0, 1, 2]
for degree in l:
    # m is given as -l <= m <= l
    for order in range(-degree, degree+1):
        # specify the dictionary key
        key = str(degree)+','+str(order)
        # preallocate the array
        for ftype in Y:
            Y[ftype][key] = np.zeros((len(thetaVEC), len(phiVEC)))
        # now, for that array, we iterate over all values of theta and
phi
        # to calculate the spherical harmonic entries of the array.
        #print key
        for i in range(0, len(thetaVEC)):
            for j in range(0, len(phiVEC)):
                # Take the absolute value of the spherical harmonic; e
lminates the imaginary part
                try:
                    Y['real'][key][i,j] = sph_harmonic(degree,order)(t
hetaVEC[i],phiVEC[j]).real
                    Y['abs'][key][i,j] = abs(sph_harmonic(degree,order
)(thetaVEC[i],phiVEC[j]))
                    Y['builtin abs'][key][i,j] = abs(sph_harm(order, d
egree, phiVEC[j], thetaVEC[i])) # ***
                    Y['builtin real'][key][i,j] = sph_harm(order, degr
ee, phiVEC[j], thetaVEC[i]).real
                except:
                    print(str(i)+'   '+str(j))
        for ftype in Y:
            #print ftype
            # for the purposes of plotting, we transpose the array
            Y[ftype][key] = np.transpose(Y[ftype][key])
            # now, we transfer the spherical harmonic vals into cartes
ian coords.
            xpts[ftype][key] = Y[ftype][key]*np.sin(theta)*np.cos(phi)
            ypts[ftype][key] = Y[ftype][key]*np.sin(theta)*np.sin(phi)
            zpts[ftype][key] = Y[ftype][key]*np.cos(theta)
            # we normalize the array it to its max value.
            Y[ftype][key] = Y[ftype][key]/Y[ftype][key].max()
            # and voila! We're ready to plot!
```

In [378]:
```
% matplotlib inline
ftype = 'real'
l, m = 2, 2
key = str(l)+','+str(m)
import matplotlib
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm, colors
fig, ax = plt.subplots(subplot_kw=dict(projection='3d'), figsize=(12,1
0))
im = ax.plot_surface(xpts[ftype][key], ypts[ftype][key], zpts[ftype][k
ey], rstride=1, cstride=1, facecolors=cm.jet(Y[ftype][key]))
ax.set_title(r'$Y_{'+str(key)+'}$', fontsize=20)
m = cm.ScalarMappable(cmap=cm.jet)
m.set_array(Y[ftype][key])  # Assign the unnormalized data array to th
e mappable
                            # so that the scale corresponds to the val
ues of R
fig.colorbar(m, shrink=0.8);
plt.show()

# Note: my plots seem to differ from what I see on wikipedia for anyth
ing other than m = 0...
# https://en.wikipedia.org/wiki/Spherical_harmonics#/media/File:Spheri
cal_Harmonics.png

# Note for me: If I want to make subplots:
# http://matplotlib.org/users/pyplot_tutorial.html
```


$Y_{2,2}$
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.4
−0.3
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0

A couple of things: first, I apologize, but I did not put axes titles on these. I think it's pretty clear that they are x, y, and z.



Second, some of these look pretty weird, especially those where l = 1. Notice that these are the absolute value of the spherical harmonic; I tried a bunch of methods to try and plot these, but this was the best I could do. I realize that they look different from what's on wikipedia but if you utilize the scipy built-in spherical harmonic function in place of mine, you get the same result. By running the code above, you automatically get four different versions of the result - two with the real component, two having taken the absolute value, each done with my function and with the built in function from scipy.

(b) Now, we will approximate the external source. Using the S4 quadrature to do the integrations and $q_e = 1$ for all angles: use the equations for external source we developed in class (eqns. 19-21), calculate the external source for $l = 0, 1, 2$.

The spherical harmonic functions take in the components of $\hat{\Omega}$ - θ and φ - as inputs. However, for the quadrature integration we require three independent variables μ, η, ξ. We have to take those three variables and transform them into what is input into the spherical harmonic functions.

Lewis and Miller show on page 176 how we can convert back and forth between the two sets of variables. $\hat{\Omega}$ is expressed in terms of either μ and η or ξ and ω in a $p-z$ geometry, where $\theta = \cos^{-1}(\xi)$ and the relationships for ω are given as the following:

$$\mu = \sqrt{(1-\xi^2)}\,\cos(\omega) \qquad \eta = \sqrt{(1-\xi^2)}\,\sin(\omega)$$

It must be noted that using level-symmetric quadrature here presents a bit of a problem. If μ_i and η_i are equal as prescribed in the LQ_N methodology, there are cases where the values of ω that result from inverting the equations above are not equal! This may occur when the some of the values of μ, η, and ξ are negative. When η is negative, the result for ω found using that equation is the negative of that of found by using μ. However, when μ is negative, the results for ω are not just of opposite sign. Based on a few guess-and-check test cases, when these are plugged into the spherical harmonic, it seems like the result is either unchanged or of the opposite sign.

I'm going to assume that we can just pick one of these equations to identify ω; specifically, I'm going to use the one in terms of η. Then, we can express $Y_{lm}(\hat{\Omega}) = Y_{lm}(\theta, \varphi) = Y_{lm}(\theta, \omega)$.

In [380]:
```
def coords_to_angles(mu, eta, xi):
    l = 1
    m = -1
    # Lewis and Miller pg. 176: Omega is expressed in terms of either
    # xi and mu or xi and eta (in a p-z geometry)
    # for now...choosing eta.
    theta = math.acos(xi)
    omega1 = math.asin(eta/math.sqrt(1-xi**2))
    omega2 = math.acos(mu/math.sqrt(1-xi**2))
    return(theta, omega1, omega2)

l=1
m=1
a = coords_to_angles(-0.8688903, 0.3500212, 0.3500212)
print(a)
print(sph_harmonic(l, m)(a[0], a[1]))
print(sph_harmonic(l, m)(a[0], a[2]))
```

```
(1.2132025916079618, 0.382949717410125, 2.7586429926242313)
(0.300196508357+0.120930276791j)
(-0.300196515182+0.120930259846j)
```

I've written a function that will yield the external source as a function of input angle, as noted in equation 19 of the notes. Sometimes, the function outputs a negative value, which is not physically significant. In this case, I automatically set the result to 0.

In [406]:
```
# Now, we will approximate the external source.
# Using the S4 quadrature to do the integrations and  qe=1
# for all angles: use the equations for external source we
# developed in class (eqns. 19-21), calculate the external
# source for l=0,1,2



# This function takes in Omega (in terms of theta and phi)
# and returns the value of the external source at that angle.
# N is the degree of the spherical harmonics (l)
def ext_source(theta, phi, N):
    source = 0.0
    qe = 1
    for l in range(0, N+1):
        for m in range(-l, l+1):
            # wrapper function for spherical harmonic
            def sph_wrapper(mu, eta, xi):
                a = coords_to_angles(mu, eta, xi)
                qe = 1
                # We only keep the real part of the spherical harmonic
.
                # qe equals one for all angles
                return(qe*sph_harmonic(l,m)(a[0], a[1]).real)
            #print('l = '+str(l)+';   m = '+str(m))
            qlm = qe*integrate_lqn(4, sph_wrapper)
            ylm = sph_harmonic(l,m)(theta, phi).real
            source += qlm*ylm
            #print('qlm = '+str(qlm))
            #print('ylm = '+str(ylm))
            #print('source = '+str(source))
    if source<=0.0:
        return(0.0)
    else:
        return(source)

# print(ext_source(2,0,0))
# print(ext_source(2,0,1))
# print(ext_source(2,0,2))

s = np.zeros((5,5))
for i in range(0,5):
    for j in range(0,5):
        s[i,j]=ext_source(i,j,2)

print s
```

```
[[ 0.99999999  0.99999999  0.99999999  0.99999999  0.99999999]
 [ 2.32021092  1.71331304  0.45059828  0.          0.13705234]
 [ 2.42662599  1.77080936  0.40631396  0.          0.06749477]
 [ 1.22140774  1.11962711  0.90786185  0.78080796  0.85527822]
 [ 2.18737181  1.64153976  0.50587887  0.          0.22388181]]
```

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

- ENDF (Evaluated Nuclear Data File) - USA
- JENDL - Japan
- JEFF (Joint Evaluated Fission and Fusion File) - Collaboration among Nuclear Energy Agency (NEA) Data bank Member Countries (Austria, Belgium, Czech Republic, Denmark, Finland, France, Germany, Greece, Hungary, Italy, Japan, Korea, Mexico, Netherlands, Norway, Poland, Portugal, Russia, Slovak Republic, Slovenia, Spain, Sweden, Switzerland, Turkey, United Kingdom) [1]

[1] https://www.oecd-nea.org/general/about/mcdb.html (https://www.oecd-nea.org/general/about/mcdb.html)

In [ ]: