

Personal Notes:
**Derivation of the neutron transport equation, and the Discrete
Ordinates Method**

August, 2018

Jan Vermaak
Rev 1.00

Contents

	Page
1 Basics - The Boltzman Neutron Transport Equation	4
2 Spherical harmonics and Associated Legendre Polynomials	7
3 The scattering Kernel and Legendre expansion	8
4 Expansion of angular variables in Spherical Harmonics	9
5 Multi-group approximation of the energy	10
6 The Discrete Ordinates Method (S_N Method)	11
6.1 Recasting the moment indices	12
6.2 Operator form	14
6.3 No scattering - Computing the fixed source	15
6.4 With scattering - Computing the scattering source (Richardson Iteration)	16
7 Piece-wise Linear Shape functions in the Finite Element application	17
7.1 Piecewise linear shape functions on a 2D triangle	18
7.2 Piecewise linear shape functions on a 2D polygon	22
7.3 Piecewise linear shape functions on a 3D tetrahedron	25
7.4 Piecewise linear shape functions on a 3D Polyhedron	28
8 Code design	31
8.1 Overall strategy	31
8.2 Handling the mesh	31
8.2.1 Pre-generated meshes	31
8.2.2 Internally generated meshes	34
8.2.3 Partially internally generated meshes	34
8.2.4 Meshing utilities	34
8.2.5 Mesh partitioning	35
8.3 Groups and Angles	36
8.4 Sweep Plane Data Structure (SPDS)	36
8.5 Flux Data Structure (FLUDS)	38
8.5.1 FLUDS α -pass Local Portion	39
Appendix A Quadrature rules for integration over angle-space	42
A.1 Gauss-Legendre quadrature rule	42
A.2 Gauss-Chebyshev quadrature rule	44

A.3	Application to Discrete Ordinates	46
A.4	Gauss-Legendre-Legendre product quadrature	46
A.5	Gauss-Legendre-Chebyshev product quadrature	48
A.6	Evaluation of product quadratures	50
Appendix B More on Spherical Harmonics		52
B.1	Expansion of a function of two angles $f(\varphi, \theta)$	52
B.2	Prototype code for spherical harmonics	53
B.2.1	Approximating an isotropic flux	56
B.2.2	Approximating an anisotropic but smooth flux	56
B.2.3	Approximating a directional flux (i.e. anisotropic + not-smooth)	58
B.3	Prototype code for real form of the spherical harmonics	59
Appendix C Creating simple materials for testing		61
C.1	Simple neutron-nuclide scattering processes	61
C.2	Combining probabilities	65
C.3	Legendre expansion of the scattering term	68
Appendix D Quadrature rule for integration of triangle space		70
Appendix E Quadrature rule for integration of tetrahedron space		71
List of Figures		
Figure 1.1	Orientation of direction in cartesian space.	4
Figure 1.2	Control volume for neutron balance.	5
Figure 6.1	Indices of Y_j as a subset of $Y_{\ell m}$	13
Figure 7.1	Mapping of a 2D triangle to a reference triangle in natural coordinates.	18
Figure 7.2	Basis function on a 2D polygon.	22
Figure 7.3	Basis functions on a triangle and on a quadrilateral.	23
Figure 7.4	Mapping of a 3D tetrahedron to a reference tetrahedron in natural coordinates.	25
Figure 7.5	Connection of the vertex of interest to the tetrahedrons that comprise the cell.	28
Figure 7.6	Influence of different vertices on the shape functions of within a tetrahedral portion of the cell.	30
Figure 8.1	Cell mapping logic.	35
Figure 8.2	Sweep ordering structure.	37
Figure 8.3	Graphical depiction of the “lockbox” concept.	38
Figure A1.1	Quadrature points and weights (colors) for the Gauss-Legendre quadrature set for both the polar and azimuthal angles with $N_a = 8$ and $N_p = 8$	47

Figure A1.2	Quadrature points and weights (colors) for the Gauss-Legendre quadrature for the polar integration and the Gauss-Chebyshev quadrature for the azimuthal angles with $N_a = 8$ and $N_p = 8$	49
Figure B2.1	Approximation of a pure isotropic function with spherical harmonics. The plot is shown for the azimuthal angle φ only.	56
Figure B2.2	Approximation of an anisotropic smooth function with spherical harmonics. The plot is shown for the azimuthal angle φ only. The radial dimension represents the flux magnitude.	57
Figure B2.3	Approximation of an anisotropic smooth function with spherical harmonics. The plot is shown for the azimuthal angle φ only. The radial dimension represents the flux magnitude.	58
Figure B2.4	Approximation of an anisotropic smooth function with spherical harmonics. The plot is shown for the azimuthal angle φ only.	59
Figure C.1	Collision kinematics of a stationary nuclide in both the laboratory reference frame and the center-of-mass reference frame.	61
Figure C.2	Cumulative probability distribution for a neutron scattering off a stationary nucleus of mass A	64
Figure C.3	Probability distribution for a neutron scattering off a stationary nucleus of mass A	65
Figure C.4	Kernel function for a neutron scattering off of a stationary carbon nuclear ($A = 12$) and scattering from group 0 to 1.	69

List of Tables

Table 1	Quadrature points and weights used for tetrahedron elements.	71
---------	--	----

1 Basics - The Boltzman Neutron Transport Equation

Let us denote the position of a neutron in space by $\mathbf{r} = [x \ y \ z]$ and the direction along which it is traveling by the normal vector $\hat{\Omega}$ such that

$$\hat{\Omega} = [\sin \theta \cdot \cos \varphi \quad \sin \theta \cdot \sin \varphi \quad \cos \theta]$$

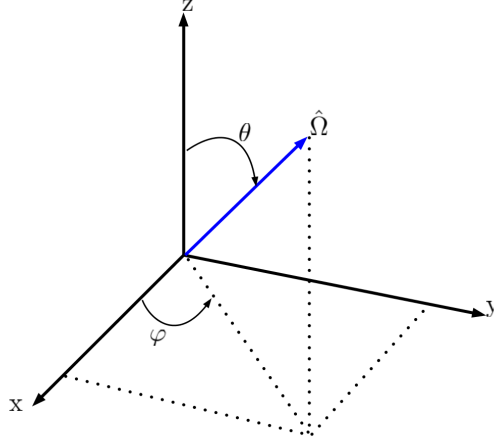


Figure 1.1: Orientation of direction in cartesian space.

Now we want to observe a control volume and the balance of neutrons within it. But first we have to define a few terms. First is the neutron density n , with units $[\frac{\text{neutrons}}{\text{cm}^3}]$, dependent on position, direction and energy, written as

$$n(r, \hat{\Omega}, E, t).$$

For this scope we will drop the notion of time dependence. Neutron density can be multiplied by the velocity associated with its energy to determine the angular flux Ψ , written as

$$\Psi(r, E, \hat{\Omega}) = v(E)n(r, E, \hat{\Omega}).$$

This has units of $[\frac{\text{neutrons}}{\text{cm}^2 \text{s}}]$ and is essential to couple reactions to known cross-sections in the form of

$$\text{Reaction rate} = \Sigma_t \Psi$$

Let us now turn our attention to the balance of neutrons in a control volume.

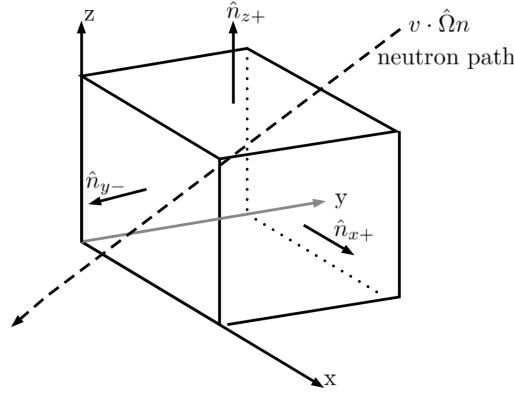


Figure 1.2: Control volume for neutron balance.

Consider the control volume as shown in figure 1.2. The time rate of change of neutrons over the volume $\frac{dn}{dt}$ is given by

$$\begin{aligned}
 \int_{dV} \frac{dn}{dt} . dr = 0 = & - \int_S \left[\hat{n} \cdot \hat{\Omega} \Psi(r, E, \hat{\Omega}) \right] . dA - \int_{dV} \Sigma_a(r, E) \Psi(r, E, \hat{\Omega}) . dr \\
 & - \int_{dV} \int_E \int_{4\pi} \Sigma_s(r, E \rightarrow E', \hat{\Omega} \rightarrow \hat{\Omega}') \Psi(r, E, \hat{\Omega}) . d\hat{\Omega}' . dE' . dr \\
 & + \int_{dV} \int_E \int_{4\pi} \Sigma_s(r, E' \rightarrow E, \hat{\Omega}' \rightarrow \hat{\Omega}) \Psi(r, E', \hat{\Omega}') . d\hat{\Omega}' . dE' . dr \\
 & + \frac{\chi(E)}{4\pi} \int_{dV} \int_E \int_{4\pi} \Sigma_f(r, E', \hat{\Omega}') \Psi(r, E', \hat{\Omega}') . d\hat{\Omega}' . dE' . dr \\
 & + \int_{dV} S(r, E, \hat{\Omega}) . dr
 \end{aligned} \tag{1.1}$$

Gauss's divergence theorem on the streaming term:

$$\int_S \left[\hat{n} \cdot \hat{\Omega} \Psi(r, E, \hat{\Omega}) \right] . dA = \int_{dV} \left[\hat{\Omega} \nabla \Psi(r, E, \hat{\Omega}) \right] . dr$$

Now:

$$\begin{aligned}
 \int_{dV} \frac{dN}{dt} . dr = 0 = & - \int_{dV} \left[\hat{\Omega} \nabla \Psi(r, E, \hat{\Omega}) \right] . dr - \int_{dV} \Sigma_a(r, E) \Psi(r, E, \hat{\Omega}) . dr \\
 & - \int_{dV} \int_E \int_{4\pi} \Sigma_s(r, E \rightarrow E', \hat{\Omega} \rightarrow \hat{\Omega}') \Psi(r, E, \hat{\Omega}) . d\hat{\Omega}' . dE' . dr \\
 & + \int_{dV} \int_E \int_{4\pi} \Sigma_s(r, E' \rightarrow E, \hat{\Omega}' \rightarrow \hat{\Omega}) \Psi(r, E', \hat{\Omega}') . d\hat{\Omega}' . dE' . dr \\
 & + \frac{\chi(E)}{4\pi} \int_{dV} \int_E \int_{4\pi} \Sigma_f(r, E', \hat{\Omega}') \Psi(r, E', \hat{\Omega}') . d\hat{\Omega}' . dE' . dr \\
 & + \int_{dV} S(r, E, \hat{\Omega}) . dr
 \end{aligned} \tag{1.2}$$

Dropping all the $\int_{dV} .dr$ gives:

$$\begin{aligned}
 0 = & -\hat{\Omega}\nabla\Psi(r, E, \hat{\Omega}) - \Sigma_a(r, E)\Psi(r, E, \hat{\Omega}) \\
 & - \int_E \int_{4\pi} \Sigma_s(r, E \rightarrow E', \hat{\Omega} \rightarrow \hat{\Omega}')\Psi(r, E, \hat{\Omega}).d\hat{\Omega}'.dE' \\
 & + \int_E \int_{4\pi} \Sigma_s(r, E' \rightarrow E, \hat{\Omega}' \rightarrow \hat{\Omega})\Psi(r, E', \hat{\Omega}').d\hat{\Omega}'.dE' \\
 & + \frac{\chi(E)}{4\pi} \int_E \int_{4\pi} \Sigma_f(r, E', \hat{\Omega}')\Psi(r, E', \hat{\Omega}').d\hat{\Omega}'.dE' \\
 & + S(r, E, \hat{\Omega})
 \end{aligned} \tag{1.3}$$

If we leave the NTE in this form we will do the scattering integral over all energy groups where neutrons are scattering to this group, and another integral over all energy groups where neutrons are scattering from this group to another. To reduce the complexity/difficulty of this we combine the scattering from group E to E' into the total removal cross-section:

$$\begin{aligned}
 \Sigma_t(r, E, \hat{\Omega})\Psi(r, E, \hat{\Omega}) = & \Sigma_a(r, E)\Psi(r, E, \hat{\Omega}) \\
 & + \int_E \int_{4\pi} \Sigma_s(r, E \rightarrow E', \hat{\Omega} \rightarrow \hat{\Omega}')\Psi(r, E, \hat{\Omega}).d\hat{\Omega}'.dE'
 \end{aligned}$$

Which gives us the base NTE:

$$\begin{aligned}
 \hat{\Omega}\nabla\Psi(r, E, \hat{\Omega}) + \Sigma_t(r, E)\Psi(r, E, \hat{\Omega}) = & \int_E \int_{4\pi} \Sigma_s(r, E' \rightarrow E, \hat{\Omega}' \rightarrow \hat{\Omega})\Psi(r, E', \hat{\Omega}').d\hat{\Omega}'.dE' \\
 & + \frac{\chi(E)}{4\pi} \int_E \int_{4\pi} \Sigma_f(r, E', \hat{\Omega}')\Psi(r, E', \hat{\Omega}').d\hat{\Omega}'.dE' \\
 & + S(r, E, \hat{\Omega})
 \end{aligned} \tag{1.4}$$

If we want to remove more complexity we can combine the fission source term into the source to get the Neutron Transport Equation in its most basic form:

$$\begin{aligned}
 \left(\hat{\Omega}\nabla + \Sigma_t(r, E) \right) \Psi(r, E, \hat{\Omega}) = & \int_E \int_{4\pi} \Sigma_s(r, E' \rightarrow E, \hat{\Omega}' \rightarrow \hat{\Omega})\Psi(r, E', \hat{\Omega}').d\hat{\Omega}'.dE' \\
 & + S(r, E, \hat{\Omega})
 \end{aligned} \tag{1.5}$$

2 Spherical harmonics and Associated Legendre Polynomials

The entry point for a person researching spherical harmonics is inevitably the wikipedia and wolfram-alpha websites describing the complex valued spherical harmonics. However, from various sources it is evident that we can compute the same expansion by using the real forms of the spherical harmonics, called the **tesseral spherical harmonics**. This expansion is done in the form

$$f(\varphi, \theta) = \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} \frac{2\ell+1}{4\pi} f_{\ell m} Y_{\ell m}(\varphi, \theta) \quad (2.1)$$

where

$$f_{\ell m} = \int_0^{2\pi} \int_0^{\pi} f(\varphi, \theta) Y_{\ell m}(\varphi, \theta) \sin \theta \, d\theta \, d\varphi$$

and

$$Y_{\ell m}(\theta, \varphi) = \begin{cases} \sqrt{2} \sqrt{\frac{(\ell-|m|)!}{(\ell+|m|)!}} P_{\ell}^{|m|}(\cos \theta) \sin |m| \varphi & \text{if } m < 0 \\ P_{\ell}^m(\cos \theta) & \text{if } m = 0 \\ \sqrt{2} \sqrt{\frac{(\ell-m)!}{(\ell+m)!}} P_{\ell}^m(\cos \theta) \cos m \varphi & \text{if } m \geq 0 \end{cases} \quad (2.2)$$

And finally the polynomials called the **Associated Legendre Polynomials** are determined from

$$\begin{aligned} P_0^0 &= 1, & P_1^0 &= x, \\ P_{\ell}^{\ell} &= -(2\ell-1)\sqrt{1-x^2} P_{\ell-1}^{\ell-1}(x) \quad \text{and} \\ (\ell-m)P_{\ell}^m &= (2\ell-1)x P_{\ell-1}^m(x) - (\ell+m-1)P_{\ell-2}^m(x). \end{aligned} \quad (2.3)$$

It is important to note that from the basic definition of the associated Legendre Polynomials

$$P_{\ell}^m(x) = (-1)^m (1-x^2)^{m/2} \frac{d^m}{dx^m} (P_{\ell}(x))$$

we can see that the associated Legendre Polynomials equates to the ordinary Legendre Polynomials when $m = 0$.

3 The scattering Kernel and Legendre expansion

The scattering cross-section of materials are normally available as a function of energy only. It is up to the user of the data to add the appropriate scattering kernels for which the scattering angles are a function of the masses of the neutron and its colliding nucleus as well as the energy. But first let us express the scattering terms in a Kernel fashion. We start with the scattering source term as is appears in equation 1.5:

$$\int_E \int_{4\pi} \Sigma_s(r, E' \rightarrow E, \hat{\Omega}' \rightarrow \hat{\Omega}) \Psi(r, E', \hat{\Omega}') . d\hat{\Omega}' . dE'.$$

We then split the macroscopic scattering cross-section into an energy dependent but not angularly dependent term, $\Sigma_s(r, E')$, and a separate kernel $K(E' \rightarrow E, \hat{\Omega}' \rightarrow \hat{\Omega})$ as

$$\int_E \int_{4\pi} \Sigma_s(r, E') . K(E' \rightarrow E, \hat{\Omega}' \rightarrow \hat{\Omega}) . \Psi(r, E', \hat{\Omega}') . d\hat{\Omega}' . dE'.$$

Naturally, we require

$$\int_E \int_{4\pi} K(E' \rightarrow E, \hat{\Omega}' \rightarrow \hat{\Omega}) . d\hat{\Omega}' . dE' = 1. \quad (3.1)$$

Now we expand this Kernel using Legendre polynomials but before we do this we have to handle the notion of $\hat{\Omega}' \rightarrow \hat{\Omega}$. We redefine the kernel as being dependent on the scattering angle θ_s where

$$\cos\theta_s = \hat{\Omega}' \bullet \hat{\Omega} = \mu$$

$$\begin{aligned} K(\cos\theta_s, E' \rightarrow E) &= \sum_{\ell=0}^{\infty} \frac{2\ell+1}{2} P_{\ell}(\cos\theta_s) K_{\ell}(E' \rightarrow E) \text{ or} \\ K(\mu, E' \rightarrow E) &= \sum_{\ell=0}^{\infty} \frac{2\ell+1}{2} P_{\ell}(\mu) K_{\ell}(E' \rightarrow E) \end{aligned}$$

This expansion is over the entire radial cone 2π and therefore we include a normalization factor of $\frac{1}{2\pi}$. The scattering term therefore becomes

$$\Sigma_s(r, E') K(E' \rightarrow E, \hat{\Omega}' \rightarrow \hat{\Omega}) = \Sigma_s(r, E') \sum_{\ell=0}^{\infty} \frac{2\ell+1}{4\pi} P_{\ell}(\cos\theta_s) K_{\ell}(E' \rightarrow E). \quad (3.2)$$

4 Expansion of angular variables in Spherical Harmonics

We can expand the angular flux that appears in the scattering source into spherical harmonics as

$$\Psi(r, E', \hat{\Omega}') = \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} \frac{2\ell+1}{4\pi} \phi_{\ell m}(r, E') Y_{\ell m}(\hat{\Omega}'), \quad (4.1)$$

where

$$\phi_{\ell m}(r, E') = \int_{\hat{\Omega}} \Psi(r, E', \hat{\Omega}) Y_{\ell m}(\hat{\Omega}) d\hat{\Omega},$$

however, we never calculate $\phi_{\ell m}$ with this integral since Ψ is still an unknown. Instead we calculate it from a set of linear equations as we will observe later. Now, if we plug equation 3.2 and 4.1 into equation 1.5 we get

$$\begin{aligned} & \left(\hat{\Omega} \nabla + \Sigma_t(r, E) \right) \Psi(r, E, \hat{\Omega}) \\ &= \int_E \int_{4\pi} \left[\Sigma_s(r, E') \sum_{\ell=0}^{\infty} \frac{2\ell+1}{4\pi} P_{\ell}(\cos\theta_s) K_{\ell}(E' \rightarrow E) \sum_{\ell'=0}^{\infty} \sum_{m'=-\ell'}^{\ell'} \frac{2\ell'+1}{4\pi} \phi_{\ell' m'}(r, E') Y_{\ell' m'}(\hat{\Omega}') \right] d\hat{\Omega}' dE' \\ &+ S(r, E, \hat{\Omega}). \end{aligned} \quad (4.2)$$

Now, the addition theorem of spherical harmonics states

$$P_{\ell}(\cos\theta_s) = P_{\ell}(\hat{\Omega}' \cdot \hat{\Omega}) = \sum_{m=-\ell}^{\ell} Y_{\ell m}(\hat{\Omega}) Y_{\ell m}(\hat{\Omega}'), \quad (4.3)$$

which if added into equation 4.2 gives

$$\begin{aligned} & \left(\hat{\Omega} \nabla + \Sigma_t(r, E) \right) \Psi(r, E, \hat{\Omega}) \\ &= \int_E \int_{4\pi} \left[\Sigma_s(r, E') \sum_{\ell=0}^{\infty} \frac{2\ell+1}{4\pi} \sum_{m=-\ell}^{\ell} Y_{\ell m}(\hat{\Omega}) Y_{\ell m}(\hat{\Omega}') K_{\ell}(E' \rightarrow E) \sum_{\ell'=0}^{\infty} \sum_{m'=-\ell'}^{\ell'} \frac{2\ell'+1}{4\pi} \phi_{\ell' m'}(r, E') Y_{\ell' m'}(\hat{\Omega}') \right] d\hat{\Omega}' dE' \\ &+ S(r, E, \hat{\Omega}). \end{aligned} \quad (4.4)$$

Now using the orthogonality of spherical harmonics

$$\int_{4\pi} Y_{\ell' m'}(\hat{\Omega}') Y_{\ell m}(\hat{\Omega}') d\hat{\Omega}' = \frac{4\pi}{2\ell'+1} \delta_{\ell\ell'} \delta_{mm'}, \quad (4.5)$$

equation 4.4 becomes

$$\begin{aligned}
& \left(\hat{\Omega} \nabla + \Sigma_t(r, E) \right) \Psi(r, E, \hat{\Omega}) \\
&= \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} \frac{2\ell+1}{4\pi} Y_{\ell m}(\hat{\Omega}) \left[\int_E \Sigma_s(r, E') K_{\ell}(E' \rightarrow E) \phi_{\ell m}(r, E') . dE' \right] \\
&+ S(r, E, \hat{\Omega}).
\end{aligned} \tag{4.6}$$

In practise the separation of the scattering cross-section from the scattering kernel is unnecessary and thus we can simplify the above equation by defining

$$\Sigma_{s\ell}(E') = \Sigma_s(r, E') K_{\ell}(E' \rightarrow E)$$

This leaves us with the convenient form

$$\begin{aligned}
& \left(\hat{\Omega} \nabla + \Sigma_t(r, E) \right) \Psi(r, E, \hat{\Omega}) \\
&= \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} \frac{2\ell+1}{4\pi} Y_{\ell m}(\hat{\Omega}) \left[\int_E \Sigma_{s\ell}(E') \phi_{\ell m}(r, E') . dE' \right] \\
&+ S(r, E, \hat{\Omega}).
\end{aligned} \tag{4.7}$$

5 Multi-group approximation of the energy

Pretty easy to see:

$$\begin{aligned}
& \left(\hat{\Omega} \nabla + \Sigma_{tg}(r) \right) \Psi_g(r, \hat{\Omega}) \\
&= \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} \frac{2\ell+1}{4\pi} Y_{\ell m}(\hat{\Omega}) \left[\sum_{g'=0}^G \Sigma_{s\ell, g' \rightarrow g}(r) \phi_{g'\ell m}(r) \right] \\
&+ S_g(r, \hat{\Omega}).
\end{aligned} \tag{5.1}$$

6 The Discrete Ordinates Method (S_N Method)

For a particular direction $\hat{\Omega}_n$ we write

$$\Psi_g(r, \hat{\Omega}_n) = \Psi_{gn}(r)$$

The multi-group neutron transport equation (i.e. equation 5.1) now becomes the angular neutron transport equation

$$\begin{aligned} & \left(\hat{\Omega}_n \nabla + \Sigma_{tg}(r) \right) \Psi_{gn}(r) \cdot dr \\ &= \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} \frac{2\ell+1}{4\pi} Y_{\ell m}(\hat{\Omega}_n) \left[\sum_{g'=0}^G \Sigma_{s\ell, g' \rightarrow g}(r) \phi_{g'\ell m}(r) \right] \cdot dr \\ &+ S_{gn}(r) \cdot dr. \end{aligned} \tag{6.1}$$

Recall that an angular function expanded using real spherical harmonics is denoted as

$$f(\theta, \varphi) = \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} \frac{2\ell+1}{4\pi} f_{\ell m} Y_{\ell m}(\theta, \varphi),$$

where

$$f_{\ell m} = \int_0^{\pi} \int_0^{2\pi} f(\theta, \varphi) Y_{\ell m}(\theta, \varphi) \cdot d\varphi \cdot d\theta$$

Similarly,

$$\phi_{g\ell m}(r) = \int_{4\pi} \Psi_g(r, \hat{\Omega}) Y_{\ell m}(\hat{\Omega}) \cdot d\hat{\Omega}.$$

Now the fundamental trick is to approximate this integral with a **quadrature rule** or a set of quadrature rules such that

$$\begin{aligned} & \int_{4\pi} \Psi_g(r, \hat{\Omega}) Y_{\ell m}(\hat{\Omega}) \cdot d\hat{\Omega} \approx \sum_{n=0}^N w_n \cdot \Psi_g(r, \hat{\Omega}_n) Y_{\ell m}(\hat{\Omega}_n) \\ & \therefore \int_{4\pi} \Psi_g(r, \hat{\Omega}) Y_{\ell m}(\hat{\Omega}) \cdot d\hat{\Omega} \approx \sum_{n=0}^N w_n \cdot \Psi_{gn}(r) Y_{\ell m}(\hat{\Omega}_n), \end{aligned} \tag{6.2}$$

Where the weights w_n and associated directions $\hat{\Omega}_n$ are particular to the quadrature rule (or rule set) used. Plugging this into equation 6.1 we get the Discrete Ordinates (S_n) equations

$$\begin{aligned}
 & \left(\hat{\Omega}_n \nabla + \Sigma_{tg}(r) \right) \Psi_{gn}(r) \\
 &= \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} \frac{2\ell+1}{4\pi} Y_{\ell m}(\hat{\Omega}_n) \left[\sum_{g'=0}^G \Sigma_{s\ell, g' \rightarrow g}(r) \left(\sum_{n'=0}^N w_{n'} Y_{\ell m}(\hat{\Omega}_{n'}) \Psi_{g'n'}(r) \right) \right] \\
 &+ S_{gn}(r).
 \end{aligned} \tag{6.3}$$

6.1 Recasting the moment indices

At this moment it is necessary to think about how to solve this problem. The indices of the spherical harmonics ($Y_{\ell m}$) is bothersome since it is not linear and therefore we can recast them. Let us suppose we have the regular spherical harmonics $Y_{lm_{true}}$, we propose Y_m with only one index such that:

$$\begin{aligned}
 & \text{when } \ell = 0, \quad Y_0 = Y_{0,0} \\
 & \text{when } \ell = 1, \quad Y_1 = Y_{1,-1} \quad Y_2 = Y_{1,0} \quad Y_3 = Y_{1,1} \\
 & \text{when } \ell = 2, \quad Y_4 = Y_{2,-2} \quad Y_5 = Y_{2,-1} \quad Y_6 = Y_{2,0} \quad Y_7 = Y_{2,1} \quad Y_8 = Y_{2,2} \\
 & \quad \quad \quad \vdots \\
 & \quad \quad \quad \text{and so forth.}
 \end{aligned}$$

The maximum index per ℓ follows the sequence

$$\begin{array}{ccccccc}
 \ell = 0 & \ell = 1 & \ell = 2 & \dots & \ell = L \\
 0 & 0+(2+1) & 0+(2+1)+(4+1) & \dots & 0+(2+1)+(4+1)+\dots+(2L+1)
 \end{array} \tag{6.4}$$

which can be expressed as a series that has the form

$$m_{max} = \sum_{\ell=1}^L (2\ell+1), \tag{6.5}$$

but when expanded in a forward and backward sense such that

$$\begin{aligned}
 & \sum_{\ell=1}^L (2\ell+1) = (2+1)+(4+1)+(6+1)+\dots+(2(L-1)+1)+(2L+1) \\
 & \text{and } \sum_{\ell=1}^L (2\ell+1) = (2L+1)+(2(L-1)+1)+\dots+(6+1)+(4+1)+(2+1),
 \end{aligned}$$

we can observe a pattern when adding these sequences together:

$$\begin{aligned}
 2 \sum_{\ell=1}^L (2\ell+1) &= (2L+4)+(2L+4)+\dots+(2L+4)+(2L+4)+(2L+4) \\
 &= L(2L+4)
 \end{aligned}$$

Therefore this series reduces to

$$m_{max} = L(L+2) \quad (6.6)$$

Therefore the maximum m for a given ℓ is $\ell(\ell+2)$ and corresponds to $m_{true} = +\ell$. This leads us to the index of any $m = \ell(\ell+2) + m_{true} - \ell$ with the test shown in figure 6.1 below.

$\ell \backslash m$	-5	-4	-3	-2	-1	0	1	2	3	4	5
0						0					
1					1	2	3				
2				4	5	6	7	8			
3			9	10	11	12	13	14	15		
4		16	17	18	19	20	21	22	23	24	
5	25	26	27	28	29	30	31	32	33	34	35

Figure 6.1: Indices of Y_j as a subset of $Y_{\ell m}$.

With this convention we can recast equation 6.3 using

$$\phi_{mg'} = \sum_{n'=0}^N w_{n'} Y_m(\hat{\Omega}'_{n'}) \cdot \Psi_{g'n'}(r),$$

and the notion that $\phi_{mg'} = \phi_{\ell m_{true} g'}$ for $m = \ell(\ell+2) + m_{true} - \ell$, as well as $\Sigma_{sm, g' \rightarrow g} = \Sigma_{sm_{true}, g' \rightarrow g}$, to

$$\begin{aligned} & \left(\hat{\Omega}_n \nabla + \Sigma_{tg}(r) \right) \Psi_{gn}(r) \\ &= \sum_{m=0}^{m_{max}} M_m(\hat{\Omega}_n) \sum_{g'=0}^G \left[\Sigma_{sm, g' \rightarrow g}(r) \cdot \left(\sum_{n'=0}^N w_{n'} \cdot Y_m(\hat{\Omega}'_{n'}) \cdot \Psi_{g'n'}(r) \right) \right] \\ &+ S_{gn}(r). \end{aligned} \quad (6.7)$$

Where M_m is developed from each ℓ, m_{true} pair to get

$$\begin{aligned} m &= \ell(\ell+1) + m_{true} \\ M_m(\hat{\Omega}_n) &= \frac{2\ell+1}{4\pi} Y_{\ell m} \end{aligned}$$

We now define the notation $M_m(\hat{\Omega}_n) = M_{mn}$ and $Y_m(\hat{\Omega}_n) = Y_{mn}$, drop the notation (r) for simplicity and rearrange the terms in equation 6.7 to get

$$\begin{aligned} & \left(\hat{\Omega}_n \nabla + \Sigma_{tg} \right) \Psi_{gn} \\ &= \sum_{g'=0}^G \left[\sum_{m=0}^{m_{max}} M_{mn} \cdot \Sigma_{sm, g' \rightarrow g} \cdot \left(\sum_{n'=0}^N w_{n'} \cdot Y_{mn'} \cdot \Psi_{g'n'} \right) \right] \\ &+ S_{gn}. \end{aligned} \quad (6.8)$$

6.2 Operator form

The streaming and removal terms can be denoted as L such that

$$\left(\hat{\Omega}_n \nabla + \Sigma_{tg} \right) \Psi_{gn} = \mathbf{L}_{gn} \Psi_{gn}$$

For the scattering source term we see that we have a Discrete-to-moment operator \mathbf{D}_{mn} , resulting from quadrature rule of integration over angle, such that

$$\mathbf{D}_m = \sum_{n'=0}^N w_{n'} \cdot Y_{mn'} \cdot \Psi_{g'n'} = \begin{bmatrix} w_0 Y_{m0} & \dots & w_N Y_{mN} \end{bmatrix} \begin{bmatrix} \Psi_{g'0} \\ \vdots \\ \Psi_{g'N} \end{bmatrix} = \mathbf{D}_m \Psi_{g'}(r) = \phi_{mg'}.$$

$$\therefore \mathbf{D}_{mn} \begin{bmatrix} \Psi_{g'0} \\ \vdots \\ \Psi_{g'N} \end{bmatrix} = \phi_{g'}$$

We also have the scattering operator $\mathbf{S}_{m,g' \rightarrow g}$ such that

$$\begin{aligned} \mathbf{S}_{m,g' \rightarrow g} \mathbf{D}_m \Psi_{g'} &= \Sigma_{sm,g' \rightarrow g}(r) \cdot \left(\sum_{n'=0}^N w_{n'} \cdot Y_{mn'} \cdot \Psi_{g'n'}(r) \right) \\ &= \begin{bmatrix} \Sigma_{s,m,g' \rightarrow g} \end{bmatrix} \begin{bmatrix} w_0 Y_{m0} & \dots & w_N Y_{mN} \end{bmatrix} \begin{bmatrix} \Psi_{g'0} \\ \vdots \\ \Psi_{g'N} \end{bmatrix} \\ &= \mathbf{S}_{m,g' \rightarrow g} \phi_{mg'} \end{aligned}$$

and the Moment-to-Discrete operator \mathbf{M}_{mn} such that

$$\left(\mathbf{M}_{mn} \mathbf{S}_{m,g' \rightarrow g} \mathbf{D}_m \Psi_{g'} \right) = \begin{bmatrix} M_{mn} \end{bmatrix} \begin{bmatrix} \Sigma_{s,m,g' \rightarrow g} \end{bmatrix} \begin{bmatrix} w_0 Y_{m0} & \dots & w_N Y_{mN} \end{bmatrix} \begin{bmatrix} \Psi_{g'0} \\ \vdots \\ \Psi_{g'N} \end{bmatrix}.$$

The summation over moments now becomes:

$$\begin{aligned}
 & \sum_{m=0}^{m_{max}} Y_{mn} \cdot \Sigma_{sm, g' \rightarrow g}(r) \cdot \left(\sum_{n'=0}^N w_{n'} \cdot Y_{m^* n'} \cdot \Psi_{g' n'}(r) \right) \\
 &= \sum_{m=0}^{m_{max}} \left(\mathbf{M}_{mn} \mathbf{S}_{m, g' \rightarrow g} \mathbf{D}_m \Psi_{g'} \right) \\
 &= \begin{bmatrix} \mathbf{M}_{0n} \mathbf{S}_{0, g' \rightarrow g} \mathbf{D}_0 & \dots & \mathbf{M}_{m_{max}n} \mathbf{S}_{m_{max}, g' \rightarrow g} \mathbf{D}_{m_{max}} \end{bmatrix} \begin{bmatrix} \Psi_{g'0} \\ \vdots \\ \Psi_{g'N} \end{bmatrix} \\
 &= \begin{bmatrix} \mathbf{M}_{0n} \mathbf{S}_{0, g' \rightarrow g} & \dots & \mathbf{M}_{m_{max}n} \mathbf{S}_{m_{max}, g' \rightarrow g} \end{bmatrix} \begin{bmatrix} \mathbf{D}_0 \\ \vdots \\ \mathbf{D}_{m_{max}} \end{bmatrix} \begin{bmatrix} \Psi_{g'0} \\ \vdots \\ \Psi_{g'N} \end{bmatrix} \\
 &= \begin{bmatrix} \mathbf{M}_{0n} & \dots & \mathbf{M}_{m_{max}n} \end{bmatrix} \begin{bmatrix} \mathbf{S}_{0, g' \rightarrow g} \\ \vdots \\ \mathbf{S}_{m_{max}, g' \rightarrow g} \end{bmatrix} \begin{bmatrix} \mathbf{D}_0 \\ \vdots \\ \mathbf{D}_{m_{max}} \end{bmatrix} \begin{bmatrix} \Psi_{g'0} \\ \vdots \\ \Psi_{g'N} \end{bmatrix} \\
 &= \mathbf{M}_n \mathbf{S}_{g' \rightarrow g} \mathbf{D} \Psi_{g'}
 \end{aligned}$$

Reducing our transport equation to

$$\mathbf{L}_{gn} \Psi_{gn} = \sum_{g'=0}^G \left[\mathbf{M}_n \mathbf{S}_{g' \rightarrow g} \mathbf{D} \Psi_{g'} \right] + s_{gn} \quad (6.9)$$

And then to

$$\mathbf{L}_{gn} \Psi_{gn} = \mathbf{M}_n \mathbf{S} \mathbf{D} \Psi + s_{gn} \quad (6.10)$$

If we then sum over all angles we get:

$$\mathbf{L} \Psi = \mathbf{M} \mathbf{S} \mathbf{D} \Psi + s \quad (6.11)$$

6.3 No scattering - Computing the fixed source

With no scattering the discrete problem to solve per cell is

$$\begin{aligned}
 \mathbf{L}_{gn} \Psi_{gn} &= s_{gn} \\
 \Psi_{gn} &= \mathbf{L}_{gn}^{-1} s_{gn}
 \end{aligned}$$

and can be solved by sweeping along the direction of each angle. Suppose we store the angular flux then per cell and sweep through all angles. We can then obtain the scalar flux from

$$\phi = \mathbf{D}\Psi = \mathbf{D}\mathbf{L}^{-1}s.$$

However, the storage of the angular flux is prohibitively expensive, and therefore we have to look at the moment-by-moment accumulation of the scalar flux moments. First we can look again at

$$\sum_{n=0}^N w_n \cdot Y_{mn} \cdot \Psi_{gn} = \begin{bmatrix} w_0 Y_{m0} & \dots & w_N Y_{mN} \end{bmatrix} \begin{bmatrix} \Psi_{g0} \\ \vdots \\ \Psi_{gN} \end{bmatrix} = \mathbf{D}_m \Psi_g(r) = \phi_{mg}.$$

This equation hints at the possibility of *accumulating* the scalar flux by starting with the first angle

$$\phi_{mg}^{new} = 0 + w_0 Y_{m0} \Psi_{g0}$$

and thereafter

$$\phi_{mg}^{new} = \phi_{mg}^{old} + w_n Y_{mn} \Psi_{gn}.$$

In this fashion we only ever have to store the scalar flux moments in each cell. For fixed source problems, it can be advantageous to call this step the *fixed source caculation* since it does not change when adding scattering or reaction based neutron source.

6.4 With scattering - Computing the scattering source (Richardson Iteration)

The addition of a scattering source, or for that matter a reaction based neutron source like fission, transforms the problem into an implicit system of equations to solve. The reason for this is that the scattering source requires a flux driver ... one we don't have at the start of the problem. To this end we manipulate our transport equation as follows:

$$\begin{aligned} \mathbf{L}\Psi &= \mathbf{MSD}\Psi + s \\ \Psi &= \mathbf{L}^{-1}\mathbf{MSD}\Psi + \mathbf{L}^{-1}s \\ \mathbf{D}\Psi &= \mathbf{DL}^{-1}\mathbf{MSD}\Psi + \mathbf{DL}^{-1}s \\ \therefore \phi &= \mathbf{DL}^{-1}\mathbf{MS}\phi + \mathbf{DL}^{-1}s \end{aligned}$$

One solution to this form of the problem is: when we are computing ϕ at iteration ℓ (i.e. $\phi^{(\ell)}$) we can approximate the scattering source from the previous iteration's ϕ (i.e. $\phi^{(\ell-1)}$) which leads to

$$\phi^{(\ell)} = \mathbf{DL}^{-1}\mathbf{MS}\phi^{(\ell-1)} + \mathbf{DL}^{-1}s.$$

This form is known as **Richardson Iteration**.

7 Piece-wise Linear Shape functions in the Finite Element application

The lower triangular operator \mathbf{L} in equation 6.11 is the operator to a hyperbolic partial differential equation for which the method of choice to discretize space is the Discontinuous Galerkin method (DG). There are two well known methods for constructing the basis functions for the DG-method, one is the use of shape functions defined on a node and connecting to all nodes of subscribing cells and when defined as linear functions these are called Piece-Wise Linear basis functions. The other method is the use of Locally Discontinuous basis functions which will not be discussed here. We shall refer to the Discontinuous Galerkin method with the use of Piece-wise Linear basis functions as PWLD for further discussions.

To see where this fits into the finite element method we consider a simple first order partial differential equation

$$\Omega \cdot \nabla \Psi + \sigma \Psi = q. \quad (7.1)$$

For each node we multiply by the (not yet defined) trial space τ_i and require that

$$\begin{aligned} \int_V \left[\tau_i \Omega \cdot \nabla \Psi + \sigma \tau_i \Psi - \tau_i q \right] dV &= 0. \\ \therefore \int_V \tau_i \Omega \cdot \nabla \Psi dV + \int_V \sigma \tau_i \Psi dV - \int_V \tau_i q dV &= 0. \end{aligned} \quad (7.2)$$

Applying integration by parts for the first term results in

$$\int_V \Omega \cdot \nabla (\tau_i \Psi) dV - \int_V \Psi \Omega \cdot \nabla \tau_i dV + \int_V \sigma \tau_i \Psi dV - \int_V \tau_i q dV = 0 \quad (7.3)$$

where we can apply Gauss's divergence theorem to the first term to obtain

$$\int_{\partial V} (\Omega \cdot \hat{n}) \tau_i \Psi dS - \int_V \Psi \Omega \cdot \nabla \tau_i dV + \int_V \sigma \tau_i \Psi dV - \int_V \tau_i q dV = 0. \quad (7.4)$$

We now apply an upwinding scheme to the boundary integral

$$\int_{\partial V} (\Omega \cdot \hat{n}) \tau_i \tilde{\Psi} dS - \int_V \Psi \Omega \cdot \nabla \tau_i dV + \int_V \sigma \tau_i \Psi dV - \int_V \tau_i q dV = 0.$$

where

$$\tilde{\Psi} = \begin{cases} \Psi_{within cell} & \text{if } \hat{n} \cdot \Omega > 0 \\ \Psi_{upwind cell} & \text{if } \hat{n} \cdot \Omega < 0 \end{cases} \quad (7.5)$$

We now apply integration again to the second term

$$\int_{\partial V} (\Omega \cdot \hat{n}) \tau_i \tilde{\Psi} dS - \int_V \Omega \cdot \nabla (\tau_i \Psi) dV + \int_V \tau_i \Omega \cdot \nabla \Psi dV + \int_V \sigma \tau_i \Psi dV - \int_V \tau_i q dV = 0.$$

and then Gauss's divergence theorem again on the second term of this

$$\int_{\partial V} (\Omega \cdot \hat{n}) \tau_i (\tilde{\Psi} - \Psi) dS + \int_V \tau_i \Omega \cdot \nabla \Psi dV + \int_V \sigma \tau_i \Psi dV - \int_V \tau_i q dV = 0.$$

Now, when we expand Ψ into basis functions, with the basis functions b_j essentially being the same as the trial functions τ_i (i.e. when $i = j$), as

$$\Psi = \sum_{j=0}^{N_{dof}-1} b_j \Psi_j$$

we arrive at the “per DOF” version of the finite element equation

$$\sum_{j=0}^{N_{dof}-1} \left[\int_{\partial V} (\Omega \cdot \hat{n}) \tau_i (\tilde{\Psi} - \Psi_j \cdot b_j) \cdot dS + \int_V \Psi_j \tau_i \Omega \cdot \nabla b_j \cdot dV + \int_V \Psi \sigma \tau_i b_j \cdot dV \right] = \int_V \tau_i q \cdot dV. \quad (7.6)$$

From this equation we see that we need expressions for the shape function τ_i and b_j as well as the derivative of the trial function $\nabla \tau_i$. We also need a means to efficiently integrate these functions over surface and volume.

7.1 Piecewise linear shape functions on a 2D triangle

For a two dimensional simulation using triangular elements we seek to map a triangle in cartesian space to a reference triangle in natural coordinates. We do this because we can develop a method to perform integration or differentiation for the reference triangle that can be mapped to a triangle of any shape and location. An example of the two triangles in different coordinate space is shown in Figure 7.1.

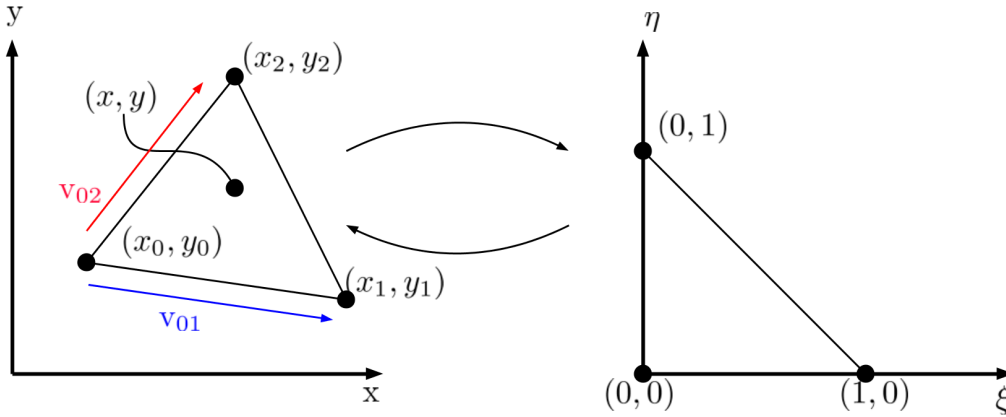


Figure 7.1: Mapping of a 2D triangle to a reference triangle in natural coordinates.

The linear basis functions for the reference triangle are

$$N_0(\xi, \eta) = 1 - \xi - \eta$$

$$N_1(\xi, \eta) = \xi$$

$$N_2(\xi, \eta) = \eta.$$

From these functions we can interpolate the point (x, y) with the following

$$x = N_0x_0 + N_1x_1 + N_2x_2$$

$$y = N_0y_0 + N_1y_1 + N_2y_2$$

We can now express x and y as functions of ξ and η by substituting the expressions for N_0 , N_1 and N_2 into the expressions for x and y

$$x = (1-\xi-\eta)x_0 + (\xi)x_1 + (\eta)x_2$$

$$= x_0 - \xi x_0 - \eta x_0 + \xi x_1 + \eta x_2$$

$$= x_0 + (x_1 - x_0)\xi + (x_2 - x_0)\eta$$

and

$$y = (1-\xi-\eta)y_0 + (\xi)y_1 + (\eta)y_2$$

$$= y_0 - \xi y_0 - \eta y_0 + \xi y_1 + \eta y_2$$

$$= y_0 + (y_1 - y_0)\xi + (y_2 - y_0)\eta$$

In terms of the vectors from vertex 0 to the other two vertices (refer to Figure 7.1) we can write this as

$$x = x_0 + v_{01x}\xi + v_{02x}\eta \quad (7.7)$$

$$y = y_0 + v_{01y}\xi + v_{02y}\eta \quad (7.8)$$

which is in the form of a linear transformation and from which we can determine the very important Jacobian matrix

$$\mathbf{J} = \begin{bmatrix} \frac{dx}{d\xi} & \frac{dx}{d\eta} \\ \frac{dy}{d\xi} & \frac{dy}{d\eta} \end{bmatrix} = \begin{bmatrix} v_{01x} & v_{02x} \\ v_{01y} & v_{02y} \end{bmatrix} = \begin{bmatrix} (x_1 - x_0) & (x_2 - x_0) \\ (y_1 - y_0) & (y_2 - y_0) \end{bmatrix}. \quad (7.9)$$

The first application of the Jacobian will be for the integration of the trial or basis function in the finite element method. For simplicity let us consider the integration of a function over x and y which can be transformed to an integration of the linear transformation of function f , i.e. function g , over ξ and η using fundamental linear algebra. This integration is then

$$\int \int f(x, y).dx.dy = \int \int g(\xi, \eta).|J|.d\xi.d\eta$$

where $|J|$ is the determinant of the Jacobian. This integration can then easily be done either analytically or by using a quadrature rule. For the reference triangle this can easily be done using the method of undetermined coefficients as detailed in appendix D.

We need to define one more item that is related to the finite element method and that is the derivative of the basis functions, $\nabla N_i(\xi, \eta)$, which can be developed by noting that

$$\begin{aligned}\frac{\partial N_i}{\partial \xi} &= \frac{\partial N_i}{\partial x} \cdot \frac{\partial x}{\partial \xi} + \frac{\partial N_i}{\partial y} \cdot \frac{\partial y}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} &= \frac{\partial N_i}{\partial x} \cdot \frac{\partial x}{\partial \eta} + \frac{\partial N_i}{\partial y} \cdot \frac{\partial y}{\partial \eta}\end{aligned}$$

which can be written as

$$\begin{aligned}\begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \end{bmatrix} &= \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix} \begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \end{bmatrix} \\ \therefore \begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \end{bmatrix} &= \mathbf{J}^T \begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \end{bmatrix}.\end{aligned}$$

Now we can invert \mathbf{J}^T to get

$$\begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \end{bmatrix} = (\mathbf{J}^T)^{-1} \begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \end{bmatrix} \quad (7.10)$$

and since the inverse of a 2×2 matrix is given by

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

we have

$$(\mathbf{J}^T)^{-1} = \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix}^{-1} = \frac{1}{|J|} \begin{bmatrix} \frac{\partial y}{\partial \eta} & -\frac{\partial y}{\partial \xi} \\ -\frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \xi} \end{bmatrix} \quad (7.11)$$

In summary:

In two dimensions using triangular cells, we wish to represent equation 7.6 (repeated here)

$$\sum_{j=0}^{N_{dof}-1} \left[\int_{\partial V} (\Omega \cdot \hat{n}) \tau_i (\tilde{\Psi} - \Psi_j \cdot b_j) \cdot dS + \int_V \Psi_j \tau_i \Omega \cdot \nabla b_j \cdot dV + \int_V \Psi \sigma \tau_i b_j \cdot dV \right] = \int_V \tau_i q \cdot dV.$$

where $\tau_i(\xi, \eta) = b_i(\xi, \eta) = N_i(\xi, \eta)$, the latter being the shape functions for each of the vertices of a triangular cell. We can evaluate these shape functions at any value of ξ and η within the reference triangle shown in Figure 7.1 as

$$\begin{aligned} N_0(\xi, \eta) &= 1 - \xi - \eta \\ N_1(\xi, \eta) &= \xi \\ N_2(\xi, \eta) &= \eta. \end{aligned}$$

The derivatives of the shape functions $\nabla \tau_i(\xi, \eta) = \nabla b_i(\xi, \eta) = \nabla N_i(\xi, \eta)$ can be computed with a given triangle using equations 7.10, 7.11 and 7.9, repeated respectively here

$$\begin{aligned} \begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \end{bmatrix} &= (\mathbf{J}^T)^{-1} \begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \end{bmatrix} \\ (\mathbf{J}^T)^{-1} &= \begin{bmatrix} \frac{\partial x}{\partial \xi} & \frac{\partial y}{\partial \xi} \\ \frac{\partial x}{\partial \eta} & \frac{\partial y}{\partial \eta} \end{bmatrix}^{-1} = \frac{1}{|J|} \begin{bmatrix} \frac{\partial y}{\partial \eta} & -\frac{\partial y}{\partial \xi} \\ -\frac{\partial x}{\partial \eta} & \frac{\partial x}{\partial \xi} \end{bmatrix} \\ \mathbf{J} &= \begin{bmatrix} \frac{dx}{d\xi} & \frac{dx}{d\eta} \\ \frac{dy}{d\xi} & \frac{dy}{d\eta} \end{bmatrix} = \begin{bmatrix} v_{01x} & v_{02x} \\ v_{01y} & v_{02y} \end{bmatrix} = \begin{bmatrix} (x_1 - x_0) & (x_2 - x_0) \\ (y_1 - y_0) & (y_2 - y_0) \end{bmatrix}. \end{aligned}$$

Integrals of the form

$$\int_V \tau_i b_j \cdot dV \quad \text{and} \quad \int_V \tau_i \Omega \cdot \nabla b_j \cdot dV \quad \text{and} \quad \int_V \tau_i \cdot dV$$

can now be evaluated using a quadrature rule of the form

$$\int_V f(x, y) \cdot dx \cdot dy = \int_{-1}^1 \int_0^{1-\eta} g(\xi, \eta) \cdot |J| \cdot d\xi \cdot d\eta = |J| \sum_{q=0}^{N_q-1} w_q g(\xi_q, \eta_q)$$

where the quadrature weights and points are given in appendix D and $g(\xi, \eta)$ are combinations of the basis functions N_i . Note the dot-product where the derivative of the base function is used. Also note that the derivative of the basis function results in a vector.

7.2 Piecewise linear shape functions on a 2D polygon

The development of the methodology as applied to a 2D triangle has direct application when applied to a polygon since a polygon most likely presents more complexity than classical reference elements like quadrilaterals and therefore using triangles as *grpsubsetsofpolygonsovercomesthiscomplexity.Theuseofsubsettrianglesfort* linear basis functions [5]. *From the latter paper it is this author's judgement that bi-linear basis functions offer little bene*

$$P_i(x, y) = N_i(x, y) + \beta_i N_c(x, y) \quad (7.12)$$

where the functions N_i and N_c are the standard linear functions defined on triangles. The subscripts i and c refer to the vertices i and center of the polygon, respectively. The β_i value is a weighting constant defined such that

$$\begin{bmatrix} x \\ y \end{bmatrix}_c = \sum_{s=0}^{N_s} \beta_s \begin{bmatrix} x \\ y \end{bmatrix}_{s,avg} \quad (7.13)$$

Naturally it follows that $\beta_s = \frac{1}{N_s}$ where N_s is the amount of sides. $[x \ y]_{s,avg}$ is the average coordinate of the two vertices of a side. An example basis function is shown in Figure 7.2. It should be noted that a single basis function now requires integration on each of the sub-triangles of the polygon instead of just a single one.

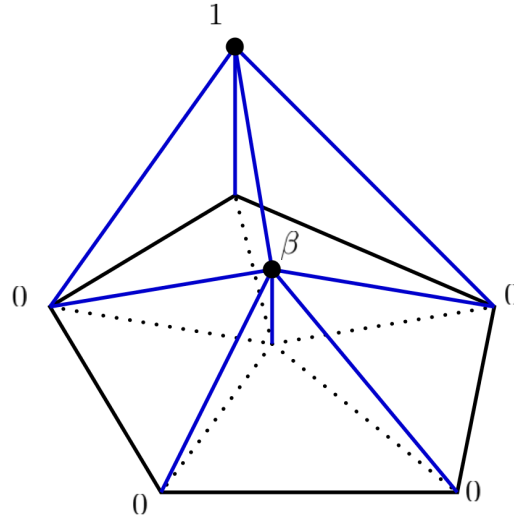


Figure 7.2: Basis function on a 2D polygon.

It is hard to visualize that this approach leads to an equivalent representation but with a few tests one can establish that this is equivalent. Some aspects of the paper presenting this method [4] that are not intuitive is the explanation that integration is now over all sides. Also, the paper does not explicitly state

that the cell centroid never features as an unknown in the solution. The customary assembly of the matrix in triangle or quadrilateral based meshes is to assemble cell-by-cell with an inner loop over the DOF of the cell. This approach is essentially the same but the paper states that integration is per side without saying that each DOF (except the cell center) of each side is also an inner loop of this integration. The subtle differences in the two algorithms can be seen in algorithm 7.1 and 7.2.

The method is versatile enough to applied to triangles and quadrilaterals where examples of the shape functions are shown in Figure

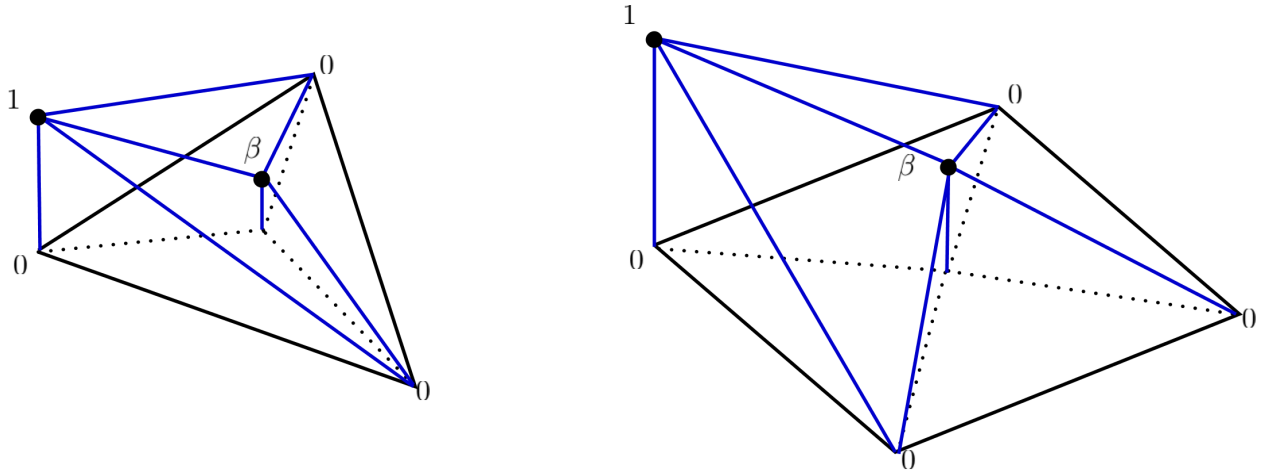


Figure 7.3: Basis functions on a triangle and on a quadrilateral.

Algorithm 7.1: Triangle Based algorithm

```

1 foreach cell do
2    $A^{N_v \times N_v} \leftarrow$  Initialize matrix ;
3    $b^{N_v \times 1} \leftarrow$  Initialize right hand side ;
4   foreach vertex i do
5     foreach vertex j do
6       foreach quadrature point qp do
7          $a_{ij} = a_{ij} + \leftarrow$  Eq. with  $\tau_i$  and  $b_j$  integrated over cell;
8          $b_i = b_i + \leftarrow$  Eq. with  $\tau_i$  and  $b_j$  integrated over cell;
9       end
10    end
11  end
12   $\phi \leftarrow$  Solve system;
13 end

```

Algorithm 7.2: Polygon Based algorithm

```

1 foreach cell do
2    $A^{N_v \times N_v} \leftarrow$  Initialize matrix ;
3    $b^{N_v \times 1} \leftarrow$  Initialize right hand side ;
4   foreach vertex i do
5     foreach vertex j do
6       foreach sub-triangle s do
7         foreach quadrature point qp do
8            $a_{ij} = a_{ij} + \leftarrow$  Eq. with  $\tau_i$  and  $b_j$  integrated over sub-triangle;
9            $b_i = b_i + \leftarrow$  Eq. with  $\tau_i$  and  $b_j$  integrated over sub-triangle;
10        end
11      end
12    end
13  end
14   $\phi \leftarrow$  Solve system;
15 end

```

7.3 Piecewise linear shape functions on a 3D tetrahedron

For a three dimensional simulation using tetrahedral elements we seek to map a tetrahedron in cartesian space to a reference tetrahedron in natural coordinates. We do this because we can develop a method to perform integration or differentiation for the reference tetrahedron that can be mapped to a tetrahedron of any shape and location. An example of the two tetrahedron in different coordinate space is shown in Figure 7.4.

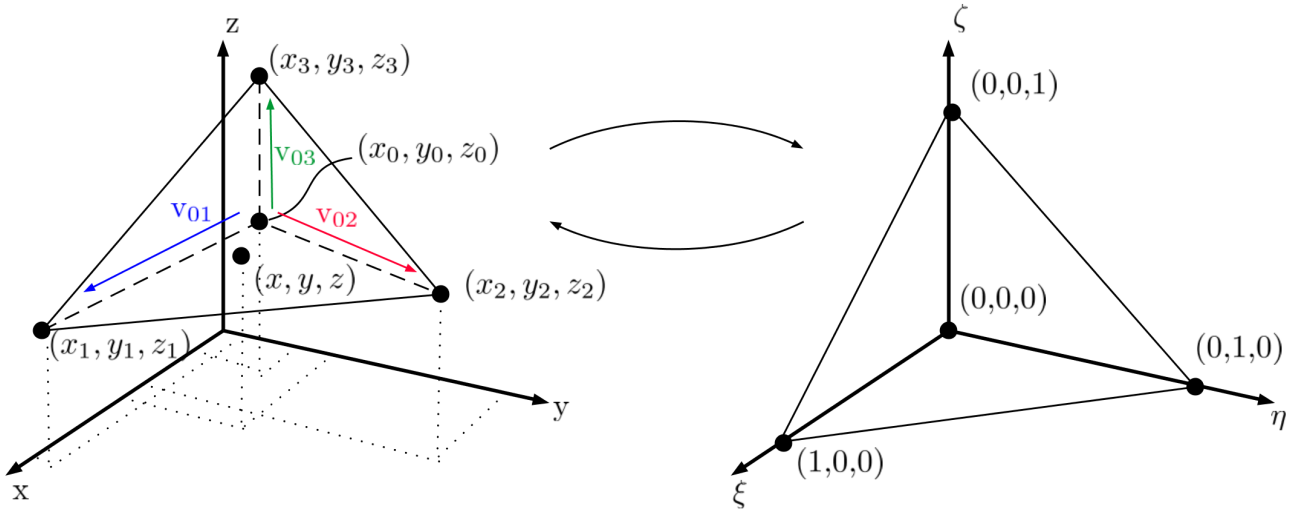


Figure 7.4: Mapping of a 3D tetrahedron to a reference tetrahedron in natural coordinates.

The linear basis functions for the reference tetrahedron are

$$N_0(\xi, \eta, \zeta) = 1 - \xi - \eta - \zeta$$

$$N_1(\xi, \eta, \zeta) = \xi$$

$$N_2(\xi, \eta, \zeta) = \eta$$

$$N_3(\xi, \eta, \zeta) = \zeta$$

From these functions we can interpolate the point (x, y, z) with the following

$$x = N_0x_0 + N_1x_1 + N_2x_2 + N_3x_3$$

$$y = N_0y_0 + N_1y_1 + N_2y_2 + N_3y_3$$

$$z = N_0z_0 + N_1z_1 + N_2z_2 + N_3z_3$$

We can express x , y and z as functions of ξ , η and ζ by substituting the basis functions into these expression to obtain

$$\begin{aligned}x &= x_0 + (x_1 - x_0)\xi + (x_2 - x_0)\eta + (x_3 - x_0)\zeta \\y &= y_0 + (y_1 - y_0)\xi + (y_2 - y_0)\eta + (y_3 - y_0)\zeta \\z &= z_0 + (z_1 - z_0)\xi + (z_2 - z_0)\eta + (z_3 - z_0)\zeta\end{aligned}$$

In terms of the vectors from vertex 0 to the other three vertices (refer to Figure 7.4) we can write this as

$$x = x_0 + v_{01x}\xi + v_{02x}\eta + v_{03x}\zeta \quad (7.14)$$

$$y = y_0 + v_{01y}\xi + v_{02y}\eta + v_{03y}\zeta \quad (7.15)$$

$$z = z_0 + v_{01z}\xi + v_{02z}\eta + v_{03z}\zeta \quad (7.16)$$

which is in the form of linear transformation and from which we can determine the very important Jacobian matrix

$$\mathbf{J} = \begin{bmatrix} \frac{dx}{d\xi} & \frac{dx}{d\eta} & \frac{dx}{d\zeta} \\ \frac{dy}{d\xi} & \frac{dy}{d\eta} & \frac{dy}{d\zeta} \\ \frac{dz}{d\xi} & \frac{dz}{d\eta} & \frac{dz}{d\zeta} \end{bmatrix} = \begin{bmatrix} v_{01x} & v_{02x} & v_{03x} \\ v_{01y} & v_{02y} & v_{03y} \\ v_{01z} & v_{02z} & v_{03z} \end{bmatrix} = \begin{bmatrix} (x_1 - x_0) & (x_2 - x_0) & (x_3 - x_0) \\ (y_1 - y_0) & (y_2 - y_0) & (y_3 - y_0) \\ (z_1 - z_0) & (z_2 - z_0) & (z_3 - z_0) \end{bmatrix}. \quad (7.17)$$

As was the case with the triangle we now seek

$$\begin{bmatrix} \frac{\partial N_i}{\partial x} \\ \frac{\partial N_i}{\partial y} \\ \frac{\partial N_i}{\partial z} \end{bmatrix} = (\mathbf{J}^T)^{-1} \begin{bmatrix} \frac{\partial N_i}{\partial \xi} \\ \frac{\partial N_i}{\partial \eta} \\ \frac{\partial N_i}{\partial \zeta} \end{bmatrix} \quad (7.18)$$

where we need to find the inverse of transpose of the Jacobian, with the transpose given by

$$\mathbf{J}^T = \begin{bmatrix} v_{01x} & v_{01y} & v_{01z} \\ v_{02x} & v_{02y} & v_{02z} \\ v_{03x} & v_{03y} & v_{03z} \end{bmatrix} \quad (7.19)$$

we can compute the determinant $|J^T| = |J|$ as

$$\begin{aligned}|J| &= |J^T| = v_{01x}(v_{02y}v_{03z} - v_{03y}v_{02z}) \\ &\quad - v_{01y}(v_{02x}v_{03z} - v_{03x}v_{02z}) \\ &\quad + v_{01z}(v_{02x}v_{03y} - v_{03x}v_{02y})\end{aligned} \quad (7.20)$$

we now compute the matrix of minors and subsequently the matrix of cofactors

$$\begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

where

$$m_{11} = v_{02y}v_{03z} - v_{03y}v_{02z}$$

$$m_{12} = v_{03x}v_{02z} - v_{02x}v_{03z}$$

$$m_{13} = v_{02x}v_{03y} - v_{03x}v_{02y}$$

$$m_{21} = v_{03y}v_{01z} - v_{01y}v_{03z}$$

$$m_{22} = v_{01x}v_{03z} - v_{03x}v_{01z}$$

$$m_{23} = v_{03x}v_{01y} - v_{01x}v_{03y}$$

$$m_{31} = v_{01y}v_{02z} - v_{02y}v_{01z}$$

$$m_{32} = v_{02x}v_{01z} - v_{01x}v_{02z}$$

$$m_{33} = v_{01x}v_{02y} - v_{02x}v_{01y}$$

To see how this was done a good explanation is available in [6]. We finally transpose this matrix and find the inverse of the transpose of the Jacobian as

$$(\mathbf{J}^T)^{-1} = \begin{bmatrix} \frac{dx}{d\xi} & \frac{dy}{d\xi} & \frac{dz}{d\xi} \\ \frac{dx}{d\eta} & \frac{dy}{d\eta} & \frac{dz}{d\eta} \\ \frac{dx}{d\zeta} & \frac{dy}{d\zeta} & \frac{dz}{d\zeta} \end{bmatrix} = \frac{1}{|J|} \begin{bmatrix} m_{11} & m_{21} & m_{31} \\ m_{12} & m_{22} & m_{32} \\ m_{13} & m_{23} & m_{33} \end{bmatrix} \quad (7.21)$$

Applying this process to the finite element problem is similar to procedure for the 2D triangle quadrature points provided in appendix E.

7.4 Piecewise linear shape functions on a 3D Polyhedron

From the same authors that presented the shape functions for polygons (in [4]) another paper was presented for polyhedrons in [8] where the shape function for each vertex i of the polyhedron is given by

$$P_i(x, y, z) = N_i(x, y, z) + \sum_{\text{faces at } i} \beta_f N_f(x, y, z) + \alpha_c N_c(x, y, z) \quad (7.22)$$

where the functions $N(x, y, z)$ are the standard linear shape functions defined on a tetrahedron. β_s and α_c is the weight that gives the face midpoint, \bar{r}_{fc} , and cell midpoint, \bar{r}_{cc} , respectively from the sum of the vertices that constitute them. i.e.

$$\bar{r}_{fc} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{fc} = \sum_{v=0}^{N_{vf}} \beta_f \begin{bmatrix} x \\ y \\ z \end{bmatrix}_v \quad (7.23)$$

$$\bar{r}_{cc} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}_{cc} = \sum_{v=0}^{N_{vc}} \alpha_c \begin{bmatrix} x \\ y \\ z \end{bmatrix}_v \quad (7.24)$$

where N_{vf} is the number of vertices for the given face and N_{vc} is the number of vertices for the entire cell. Naturally it follows that $\beta_f = \frac{1}{N_{vf}}$ and $\alpha_c = \frac{1}{N_{vc}}$. The format of equation 7.22 is not intuitive at first sight ... it is hard to comprehend the summation over faces “at j”, but let us try to clarify this with a diagram (see Figure 7.5).

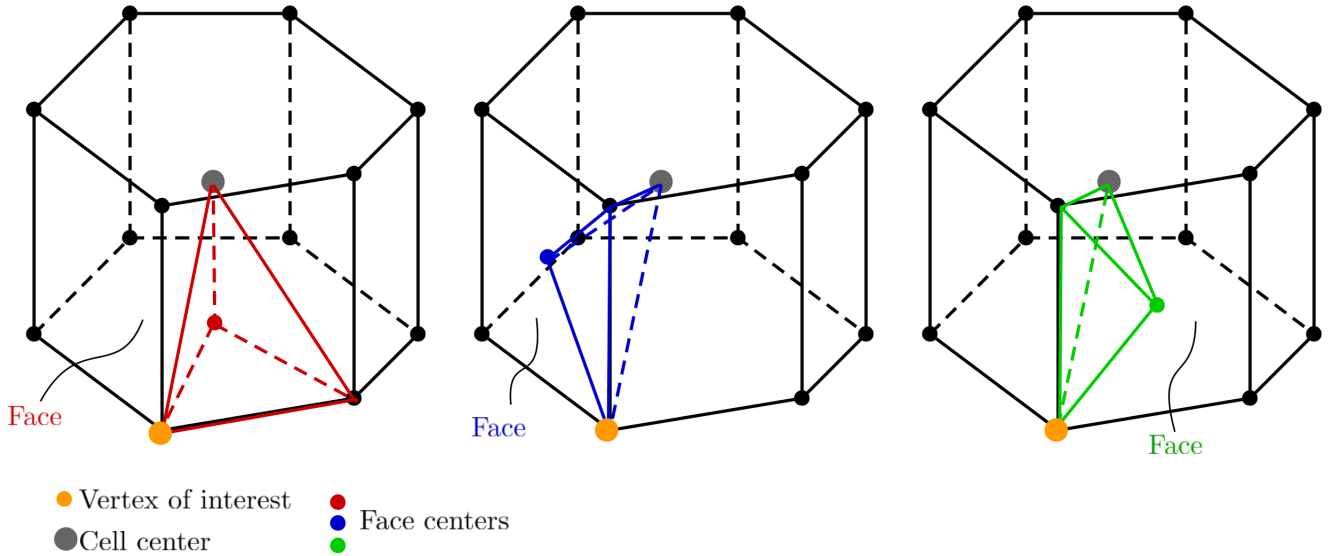


Figure 7.5: Connection of the vertex of interest to the tetrahedrons that comprise the cell.

Firstly we split the polyhedron into faces, where each face can be a polygonal face. Each face is then split into a number of sides. A side is a tetrahedron, corresponding to a face, which is formed from each edge of the polygonal face where the vertex collection are the two vertices of the edge, the face center and the cell center. In other words the face center and the two vertices of the edge forms a triangle, and the cell center makes it a tetrahedron.

As was the case with the polygon, all the other vertices j (not i) are connected to the vertex of interest i through the cell center. Again, we don't include the cell center as a point in the simulation so we have to spread its effect through to each of the vertices using the α_c factor. We also have face centered shape functions associated with the division of each polygonal face into sides. On each face, to which vertex i belongs, the shape functions defined on the face centers will protrude into the tetrahedron under consideration (i.e. the tetrahedron associated with vertex i associated with face f , side s). Therefore more clearly we can express the shape functions on a tetrahedron-by-tetrahedron basis

$$P_i^{tet}(x, y, z) = \begin{cases} \alpha_c N_c(x, y, z) & \text{no matter which tetrahedron} \\ +\beta_f N_f(x, y, z) & \text{if vertex } i \text{ is part of the face} \\ +N_i(x, y, z) & \text{if vertex } i \text{ is part of the face-side pair} \end{cases} \quad (7.25)$$

Figure 7.6 shows the influence of a shape function (centered on a specific point as denoted by the start of an arrow) from a specific vertex (color). The orange colored vertex's influence is shown on the left most figure where the shape function is then the full equation because all of the conditions are met; i.e. $\alpha_c N_c$ is always present, vertex i is indeed part of the face where this tetrahedron is defined and therefore $\beta_f N_f$ is present, finally it is also part of the side of the tetrahedron and therefore its basic shape function N_i is present.

The middle figure shows the red vertex as a vertex that only has the contribution of $\alpha_c N_c$ and $\beta_f N_f$ because the red vertex is not on the side composing the tetrahedron of interest. The rightmost figure shows only the contribution of $\alpha_c N_c$ because the blue vertex is not part of any adjacent face.

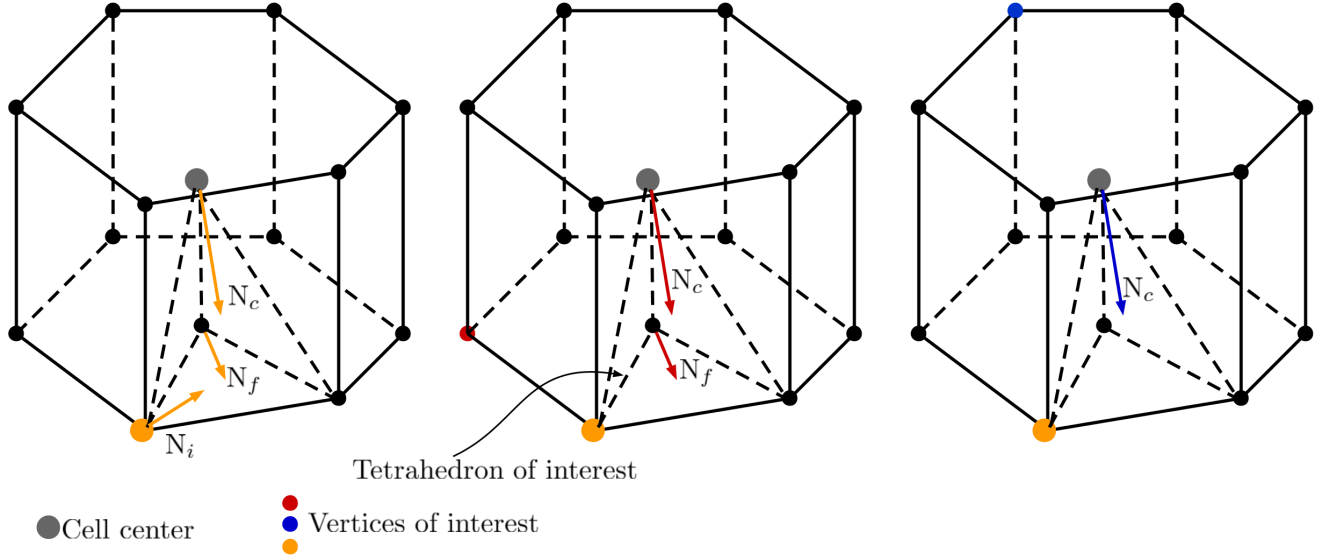


Figure 7.6: Influence of different vertices on the shape functions of within a tetrahedral portion of the cell.

Algorithm 7.3: Polyhedron Based algorithm

```

1 foreach cell do
2    $A^{N_v \times N_v} \leftarrow$  Initialize matrix ;
3    $b^{N_v \times 1} \leftarrow$  Initialize right hand side ;
4   foreach vertex  $i$  do
5     foreach vertex  $j$  do
6       foreach face  $f$  do
7         foreach sub-tetrahedral  $s$  do
8           foreach quadrature point  $qp$  do
9              $a_{ij} = a_{ij} + \leftarrow$  Eq. with  $\tau_i$  and  $b_j$  integrated over sub-triangle;
10             $b_i = b_i + \leftarrow$  Eq. with  $\tau_i$  and  $b_j$  integrated over sub-triangle;
11          end
12        end
13      end
14    end
15  end
16   $\phi \leftarrow$  Solve system;
17 end
    
```

8 Code design

8.1 Overall strategy

The neutron transport equation, with the scattering term and angular flux expanded, results in a large amount of variables to be stored and computed. Before introducing any form of spatial discretization the code has to handle the scalar flux moments $\phi_0 \dots \phi_L$ for which we have a set for each group. When a spatial discretization scheme, along with a collection of cells, is introduced the problem size is scaled by the product of the number of cells and each cells respective degrees of freedom (DOF). Therefore the primary strategy of the code design is to support domain decomposition.

8.2 Handling the mesh

All mesh related items will be contained within the `namespace` called `chi_mesh`. The primary item will be a storage space that will allow us to store all mesh entities into a global container we call the `chi_mesh::MeshHandler` class. All mesh creation and manipulations happen with a mesh handler, set to “current”, and therefore in theory the code supports the use of multiple meshes, but that is a design feature to explore later.

Mesh construction, in general, can be split into 3 different methods:

- Use of pre-generated meshes (PGM). Such meshes have been built outside of the code and is in a form conducive to the direct construction of internal mesh data structures. This is a rather complex method since it requires knowledge of the source of such methods.
- Generation of meshes within the code, i.e. internally generated meshes (IGM). Such a method will require the complete specification of the geometry within the code input structure as well as the algorithms required to create fully functional meshes with boundary identification.
- Partial internally generated meshing (PIGM). This method mostly implies the extrusion of a 2D surfaces to a 3D mesh but could extend to a 3D-printed type mesh..

8.2.1 Pre-generated meshes

Pre-generated meshes (PGMs) is a large topic for computational physics and not just a topic in this work. Many well developed packages exist that can produce high quality 3D meshes very efficiently and in most cases in parallel. Historically a software package generates a mesh file that is to be read by the code that uses it and for most applications a single file with all the mesh information suffices. However, in consideration of scaling simulations (i.t.o. number of spatial unknowns) one has to consider the “friendliness” of such meshes towards scaling.

In the simplest case consider a mesh comprising hexahedrals, each cell will at minimum require:

- **4byte** The cell memory pointer.
- **byte** Number of vertices.
- **4byte** Pointer to vertex array.
- **8×4bytes** Vertex index array.
- **byte** Number of faces.
- **4byte** Pointer to face information array.
- **4×4 bytes** for each face to store vertex indices, **96 bytes** total over the 6 faces.
- **4bytes** for each face storing the neighboring cell or boundary, **16bytes** total over the 6 faces.
- **4bytes** The material index.

This is a total of 162 bytes for the storage of each cell and therefore we can fit approximately 6.6 million cells into 1 GB of memory. In practice though cell information includes a number of other variables that ease the lookup of information and the calculation of simulation inputs and in this regard it is more realistic to assume less than this number. However, if each process in a parallel simulation had to read and process the pregenerated mesh file then the total size of the mesh will essentially be capped by the average memory per core which is currently only as high as 3.5 GB per core (i.e. the Lawrence Livermore National Laboratory's Quartz cluster).

Scalable PGMs therefore require support for domain decomposition and methods to do so are being developed. The concept of generating cell counts in excess of 1 billion with a mesh generator is also a cause for concern since such a meshing tool will also require a means to do so efficiently and mostly in parallel as well. The end-specification however remains constant: PGMs need to exist in multiple files that are conducive to each process reading only the mesh that is allocated to it. Such efforts will require advanced domain decomposition and load balancing.

Support for PGMs has been an engineered features of ChiTech from the start because of the light-weight structure employed for containing mesh information. Interfaces can easily be developed to move meshes of any format into the database. To this end mesh information is stored within ChiTech as either **CellPolygon** or **CellPolyhedron** . The data structures for each of these is simple:

```
class chi_mesh::Cell
{
public:
    int cell_global_id;
    int cell_local_id;
    std::pair<int,int> xy_partition_indices;
    std::tuple<int,int,int> xyz_partition_indices;
    int partition_id;
    Vertex centroid;
    int material_id;
}
```

```
class chi_mesh::CellPolygon : public chi_mesh::Cell
{
public:
    std::vector<int> v_indices;
    std::vector<int*> edges; ///< Stores arrays of edge indices
    std::vector<chi_mesh::Vector> edgenormals;
};
```

```
class chi_mesh::CellPolyhedron : public chi_mesh::Cell
{
public:
    std::vector<int> v_indices;
    std::vector<PolyFace*> faces;
};

struct chi_mesh::PolyFace
{
    std::vector<int> v_indices;
    std::vector<int*> edges;
    int face_indices[3];

    chi_mesh::Normal geometric_normal;
    chi_mesh::Vertex face_centroid;
};
```

In these data structures the normals and centroids can be computed as they are loaded and knowledge of neighbors is incorporated via [edges](#) or [face_indices](#). The base-class [Cell](#) also doubles as a very lightweight “ghost” cell which can conveniently be communicated between process to ease algorithm detail downstream of mesh loading. This brings one to the topic of global cell indices which ChiTech uses. This eases the burden on engineering programmers where the concepts of graphs are not always solid. It also assists in restarting simulations with different partitioning strategies, sometimes even after a field function has already been computed. This strategy has proved very successful.

Reading decomposed 2D meshes will be incorporated in the [VolumeMesherPredefined2D](#) volume mesher, and reading decomposed 3D meshes will be incorporated through the [VolumeMesherPredefined3D](#) volume meshers.

8.2.2 Internally generated meshes

In the ideal case the code would have had the capability of generating, in parallel, a fully functional 3D mesh similar to what can be read with the predefined 3D mesh reader, however, a simple method of obtaining 3D meshes is to extrude 2D meshes and if the 2D meshes themselves are generated with the code then technically such a method is considered completely internally generated. The planning therefore is to use a Delaunay triangulation mesher to generate a 2D mesh and to extrude this mesh. The triangulation mesher will be named `SurfaceMesherDelaunay` and the corresponding extruder will be named `VolumeMesherExtruder`.

8.2.3 Partially internally generated meshes

As a subset of a 3D mesh generator using extrusion is the case where the 2D surface mesh is pre-defined and for this case we will name this process the `SurfaceMesherPredefined` mesher which will feed into the `VolumeMesherExtruder`.

8.2.4 Meshing utilities

In support of meshing operations a number of data structures and classes are defined to make the namespace all-inclusive. These are:

- `Vector`. Also doubling for `Vertex`, `Normal` and `Node`. A general data structure that stores a vector of 3 components (x, y, z) and supports vector arithmetic, dot-products, cross-products and the L2-norm.
- `Edge` and `EdgeLoop` for the storage of edges.
- `Face` and `Polyface` for the storage of triangular and polygonal faces, respectively.
- `LineMesh` for storing a collection of lines used for the Delaunay mesher and to define boundaries.
- `SurfaceMesh` for storing either surface meshes read from file or store surface meshes generated within the code.
- `Boundary`, `Region` and `MeshContinuum` with each nesting from left to right is used to define continuum on which the mesh is defined.
- `CellTriangle`, `CellPolygon` and `CellPolyhedron` as a basic data structure to be handled by any process downstream. The end-goal of meshing is to have a collection of these.
- `Cellset` a collection of cell id's belonging to the local process.

8.2.5 Mesh partitioning

The current design for mesh partitioning requires unique global cell indices that map down to local cell indices. Cells are stored in a data structure named `MeshContinuum` which gets pushed onto a `Region` datastructure during the execution of the volume-mesher. This allows us to facilitate different regions of meshing. Within the `MeshContinuum` the cells and their associated nodes are stored along with an important mapping vector `local_cell_glob_indices` which contains the global index mapping of local cells. Programmers can simply supply the local cell index to obtain the global cell index (the inverse mapping is also computed by the meshing routines).

Only local cells are fully defined, as either `CellPolygon` or `CellPolyhedron`, whereas non-local cells are stored as a near empty reference to the parent cell type `Cell` which only stores its centroid and its partition id. We can call the parent cell type a “ghost”-cell because it gets used as a placeholder when a cell is not local. Using these ghost cells is very advantageous for communicating data and is very easy to implement with an extruder mesher. An example of this is shown in Figure 8.1 below.

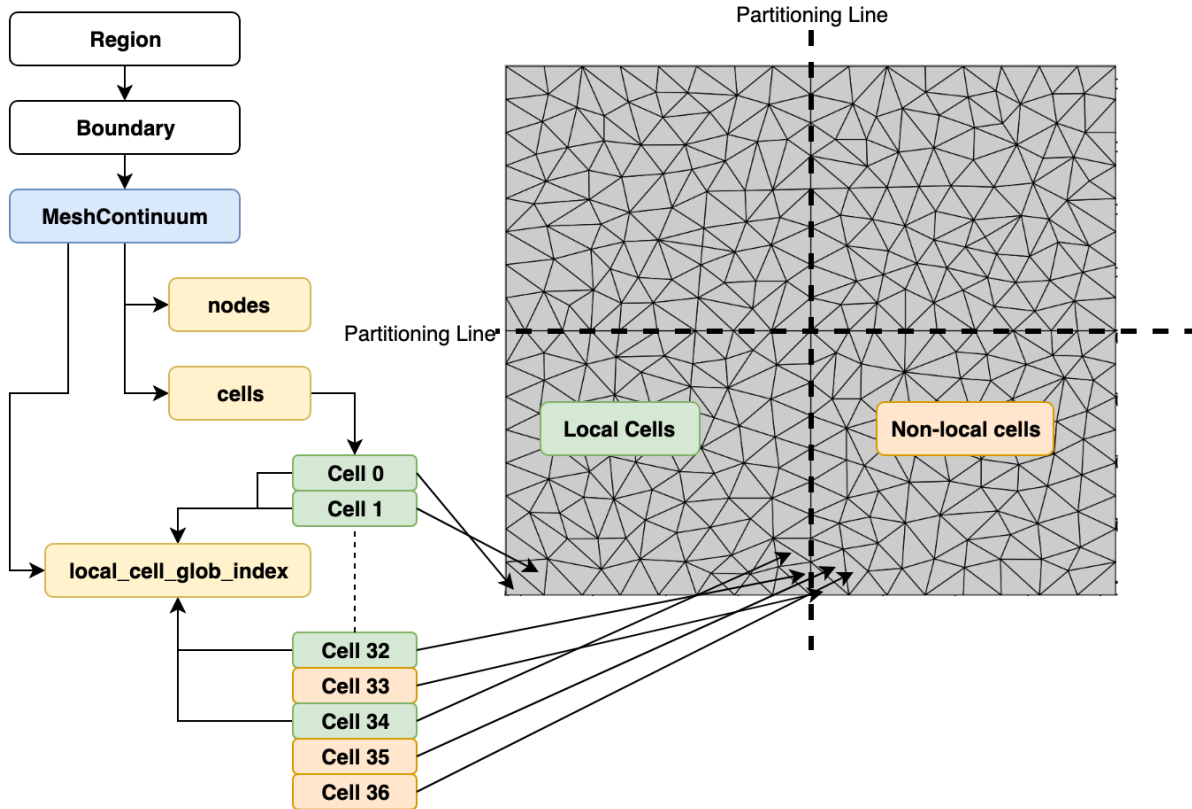


Figure 8.1: Cell mapping logic.

8.3 Groups and Angles

As is often the case with neutral particle transport, the notion of a `Group` arises mostly because the material properties are dependent on particle energy and hence require discretization in the energy domain. Particles can be transferred (i.e. scattered) back and forth between energy groups and hence we will require a material property called the `ScatteringMatrix` which holds the likelihood of a particle scattering from group g' to group g . Hence, for G groups, the scattering matrix can be a $G \times G$ dense matrix, however in reality we can introduce some sparsity to these matrices since particles can scatter in a band of groups.

Because the problem size essentially scales with the number of groups, a very useful memory-saving technique is to split the collection of groups into a number of distinct collections, each called a `Groupset` with unique properties. One important property that a groupset can have is its `ProductQuadrature` rule for integration over angle space. Particular quadrature rules are discussed in the appendices. Discrete ordinate angles directly related to the abscissae used in the quadrature rule and therefore the `Angles` are determined by these rules. As will be seen later, angles, like groups, can have similar properties which we can group into `AngleSets` in order to speed up parallel computations.

An additional feature incorporated into groupsets is `Groupset::Subsets` which allows further separation of anglesets. This has proven advantageous for the development of “feedstock” to the sweeping algorithm where the parallel efficiency is dominated by the ratio of idle tasks to total number of tasks.

8.4 Sweep Plane Data Structure (SPDS)

When sweeping along a certain direction, cell dependencies are defined by the notion of upstream dependencies. That is, for current cell i ... if the dot product of the sweep direction vector with the surface normal is negative (i.e. $\Omega \cdot \hat{n} < 0$), and the face associated with that surface is connected to another cell j and not a boundary, then cell i is dependent on cell j . Connecting all the cells in this fashion for a particular direction results in the computer science concept called a Directed-Graph, and a useful algorithm developed in this field is the **topological sort algorithm**. A topologically sorted Directed-Graph provides a sorted list of indices which ensures that a sequential reference to the cells pointed by these indices will reference cells with their dependencies already met during each subsequent downstream reference.

The `boost::graph` library has this particular functionality with the restriction that the graph must be a Directed-Acyclic-Graph (DAG) which essentially means that cells cannot have cyclic dependencies. Fortunately by enforcing the use of strictly convex cells, acyclic dependencies are ensured. This also means that partitioning cuts need to cut through a cell instead of separating along faces. We must also note that for a parallel implementation we will have a “fine-view” of the sweep ordering as well as a “course-view” which is the processor dependency for the given sweep ordering.

Sweeping on meshes is handled by the `SweepManagement` namespace. Given a grid and a direction (Ω), the function call `SweepManagement::CreateSweepOrdering` will firstly assemble a DAG of the local cells and create a single Sweep Plane Local Subgrid (SPLS) that contains the indices of all the local cells, sorted topologically against the direction. This provides the “fine-view” of sweep ordering. This is stored in the vector `SPLS.item_id` on each sweep ordering. Note that at the time of writing only a single sweep plane is assembled for the local sweep, this is because the local sweeping is inherently serial and there is no parallel benefit to using multiple sweep planes, however, future support for memory optimizing schemes are supported by allowing multiple sweep planes. During the process of generating a local sweep plane, a set (`std::set`) of location dependencies is also constructed which feeds into the subsequent step which is building the “course view” of the SPDS. The dependencies are split into `location_predecessors` and `location_successors` for later use. The latter step will generate one or more Global Sweep Plane Orderings (GSPOs) from which the parallel nature arises, with each global sweep plane denoting the locations that are “in-play”. The purpose of the “course view” sweep planes is hard to explain without further context, however, such a discussion will have to be deferred to later but for now it will be said that these planes are used in the Depth-Of-Graph sweep scheduling algorithm. The final piece of each SPDS is the Flux Data Structure (FLUDS) which is the topic of the next subsection but required to be mentioned here to give context to Figure 8.2 below, which depicts the hierarchy of datastructures.

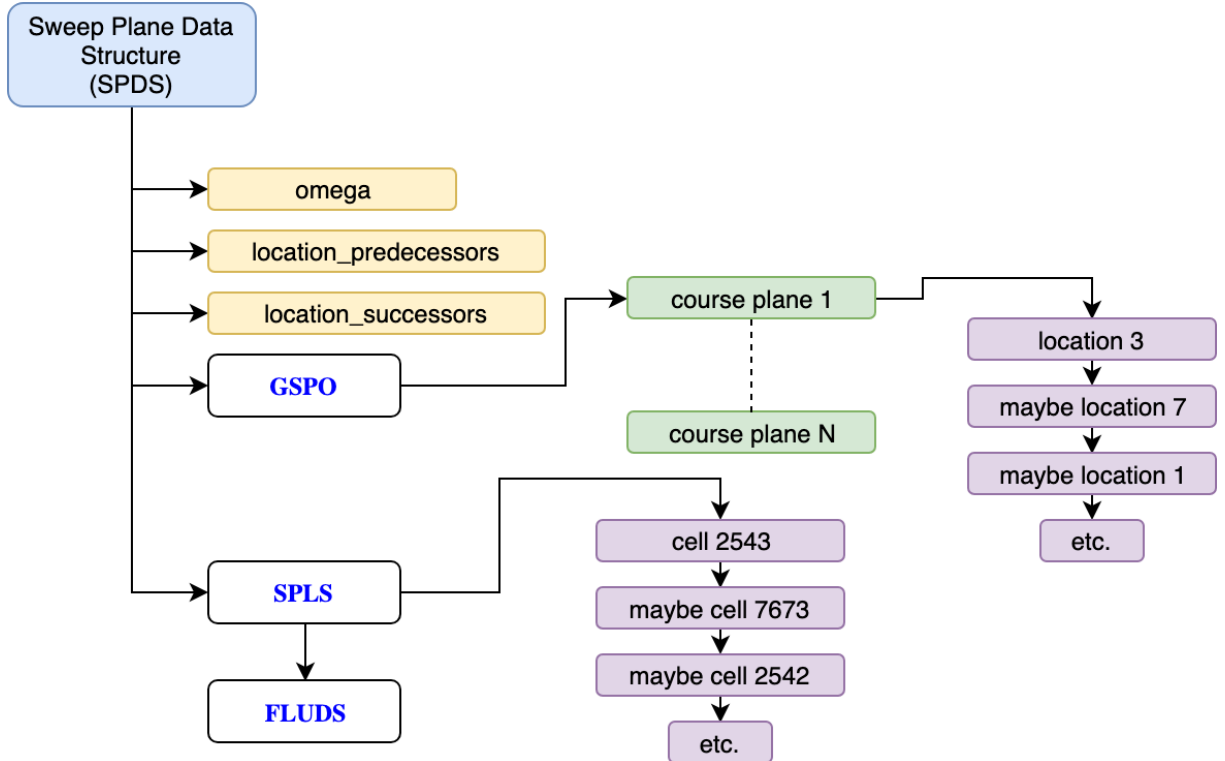


Figure 8.2: Sweep ordering structure.

8.5 Flux Data Structure (FLUDS)

A flux datastructure needs to cater for a host of requirements including the prevention of having to filter for stored information. PDT's STAPL library operates on the principle of filtering and plans are in progress [10] to develop a better implementation. Based on this work ChiTech has been equipped with a similar FLUDS.

Let us start with the local subgrid traversal (SPLS). A topological sorting of the Directed Graph was constructed during the development of the [SweepOrdering](#) datastructure and contains the order in which local cells are to be traversed in a sweep for a given angleset. One of the first memory saving principles is that a downstream cell can reuse the memory space where its incoming information was stored. We shall denote this concept as a “lockbox” (`vector<pair<int,short>> lockbox`) and to this end one can imagine an operator that places information in the next available slot and opens up a used slot whenever information is withdrawn. With this paradigm the maximum lockbox size is ultimately determined only by the maximum cells in play. It operates as follows (see Figure 8.3):

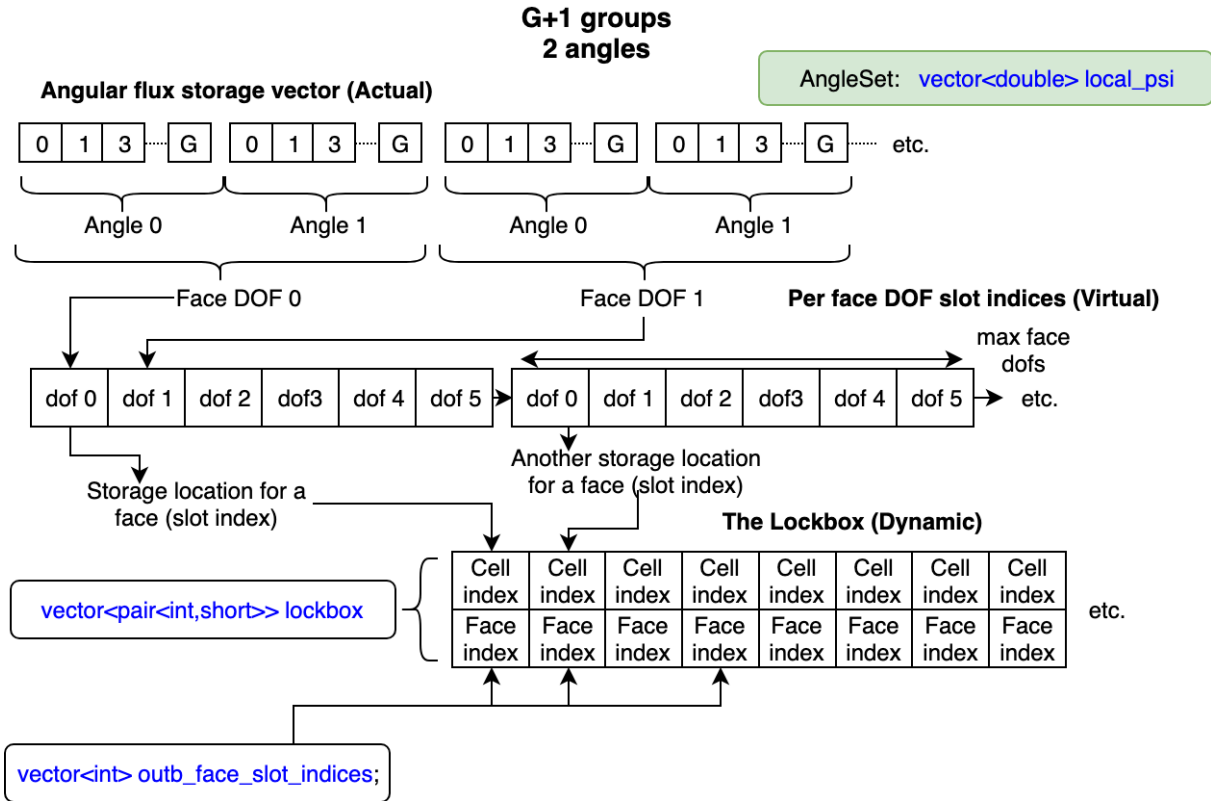


Figure 8.3: Graphical depiction of the “lockbox” concept.

Note in the figure above we only need to know the slot index of where a cell-face pair will store its data. The actual locations in the `local_psi` vector can be computed when the angle and group is provided.

Some discussion is also warranted on an idea developed in [10]. The idea is that, according to the above paradigm, the local Ψ vector will be sized according to the number of DOFs on the face with the maximum number of DOFs. This has the potential of allocating a large amount of unused memory if, for arguments sake, 95% of the cells have the same amount of face dofs (say 4) and only 5% have lets say 12 DOFs. Effectively 95% of the faces will then allocate space for 12 DOFs per face where it only requires 4. There is a strategy to overcome this but for now this strategy hasn't been employed since we found it hard to test.

8.5.1 FLUDS α -pass Local Portion

Filling the lockbox phase:

In this phase we do a simplistic loop over faces. Note that during this phase we also determine the maximum degrees of freedom per face which will define our local storage requirement.

- Initially the lockbox (vector of pairs) is empty, the first cell is encountered. We also instantiate an outbound face counter (`outgoing_face_count`) and a vector of integers denoting slot indices. (`vector<int> outb_face_slot_indices`)
- For each face index of the cell (which will never be a large number hence the use of `short`) it is determined if the face is outgoing (dot product with the sweep ordering's reference direction vector). If it is, we push the cell's global index and the associated face index **as a pair** to the lockbox. We also increment the outbound face counter and assign a new slot index.
- For each subsequent cell in the topological sorting we do the following:
 - We first loop over all the incoming faces. From our meshing datastructure we will know the upstream neighbor of that face and because the neighbor is local we can determine (from the vertices) which face of the neighbor is associated with the current face. This provides us with a look up pair (global cell index and face index)
 - We then filter the lockbox with this pair as lookup and once we find the lockbox slot we empty it by assigning a negative number to the cell index.
 - We now loop over the outgoing faces but instead of simply pushing a new pair into the lockbox, we first filter through the lockbox looking for an empty slot. If we found an empty slot we place the current cell's global index and face index pair into this slot (as well as assign a new slot index). If we did not find an open slot we push into a new slot (and assign the slot index).

Mapping local incident faces phase:

With the lockbox slotting strategy developed we now start a another phase. During the first phase we determined where outgoing faces will store their information. The storage information for each of those faces is determined by the first DOF of every outgoing face since the data will be contiguous for face DOF $i = 0 \dots N$. The contiguous nature does not apply to the downstream cell's incoming face since the face

vertices are in reverse order (right hand rule) and their respective DOF_0 is not necessarily matched. We therefore need to fill the following data structure for each cell:

```
vector<pair<int,vector<int>>> inco_face_dof_mapping
```

The first level of this structure is a vector incremented by a incident face counter and then holding a pair comprising a lockbox slot and a vector. It is trivial to envision that we can determine an incident face's lockbox slot from the previously developed information, i.e. we know the neighboring cell's index and we can determine the associated face, therefore we can determine where that face will start storing its data (which will be a small contiguous block). We then just have to map each of the current cell's face DOFs to a location in this small contiguous block.

- For each cell in the topological sort we loop over only the incident faces. `incoming_face_count` is incremented. We determine the face's upstream cell as well as its associated face. This determines the slot index.
- We then do a mapping of the face DOFs. A fairly trivial operation by which we loop over the current cell face DOFs and internally loop over the upstream cell's face DOFs. If a match is found the mapping is pushed to the vector portion of the mapping pair.

At the end of this pass we will have two primary FLUDS data structures:

```
vector<vector<int>> so_cell_outb_face_slot_indices
```

Given a cell's sweep ordering index, then outbound face counter, this structure holds the location index of the start of the face's outbound storage information.

```
vector<vector<pair>> so_cell_inco_face_dof_indices
```

Given a cell's sweep ordering index, then inbound face counter, this structure holds a pair. The first item of the pair is where the upstream information block starts and the second is a mapping of the face's DOFs inside this block.

These two data structures completely define where a cell can store its outgoing angular flux (per face) and also where it can retrieve its incoming angular flux (per face). Accessing these location within a sweep chunk is achieved with the two functions calls:

See next page

```
double* chi_mesh::SweepManagement::FLUDS::
OutgoingPsi(int cell_so_index, int outb_face_counter,
            int face_dof, int n)
{
    int index =
        local_psi_Gn_block_stride*G*n +
        so_cell_outb_face_slot_indices[cell_so_index][outb_face_counter]*
        local_psi_stride*G +
        face_dof*G;

    return &ref_local_psi->operator [] (index);
}
```

```
double* chi_mesh::SweepManagement::FLUDS::
UpwindPsi(int cell_so_index, int inc_face_counter,
          int face_dof, int g, int n)
{
    int index =
        local_psi_Gn_block_stride*G*n +
        so_cell_inco_face_dof_indices[cell_so_index][inc_face_counter].first*
        local_psi_stride*G +
        so_cell_inco_face_dof_indices[cell_so_index][inc_face_counter].
            second[face_dof]*G + g;

    return &ref_local_psi->operator [] (index);
}
```

Appendix A Quadrature rules for integration over angle-space

Suppose we have a function of azimuthal angle φ and polar angle θ namely, $f(\varphi, \theta)$, and we integrate this function over the entire angular domain. We seek a quadrature rule (or a combination of rules) that will allow the simplified integration in the form

$$\int_0^{2\pi} \int_0^\pi f(\varphi, \theta) \sin\theta \, d\theta \, d\varphi \approx \sum_n^N w_n f((\varphi, \theta)_n).$$

Here the values w_n are weights and $(\varphi, \theta)_n$ are the abscissae of the t.b.d. quadrature rule. However, the form of this integral is not yet in a form conducive to the application of quadrature rules. To this end we observe that

$$\mu = \cos\theta$$

and

$$\frac{d\mu}{d\theta} = -\sin\theta$$

We can now express f as a function of μ instead of θ for which we have

$$\begin{aligned} & \int_0^{2\pi} \int_1^{-1} -f(\varphi, \mu) \, d\mu \, d\varphi \\ &= \int_0^{2\pi} \int_{-1}^1 f(\varphi, \mu) \, d\mu \, d\varphi \end{aligned}$$

This equation is convenient since we can apply quadrature rules to each angular case. To see this let us assign

$$g(\varphi) = \int_{-1}^1 f(\varphi, \mu) \, d\mu \tag{A.1}$$

for which we now have

$$\int_0^{2\pi} g(\varphi) \, d\varphi. \tag{A.2}$$

A.1 Gauss-Legendre quadrature rule

The *Gauss-Legendre* quadrature rule is used when the weighting function is unity, i.e. $w(x)=1$, and therefore we can integrate the function in the form

$$\int_{-1}^1 f(x) \, dx \approx \sum_{n=1}^N w_n f(x_n).$$

Here the abscissae x_n are the roots of the Legendre polynomial $P_n(x)$, and the weights are

$$w_n = \frac{2(1-x_n^2)}{(n+1)^2(P_{n+1}(x_n))^2} \tag{A.3}$$

From this equation we can see we need to evaluate the values of the Legendre polynomials. For this purpose we use the recursion of Legendre polynomials stating that

$$\begin{aligned} P_0(x) &= 1 \\ P_1(x) &= x \\ P_{n+1}(x) &= \left(\frac{2n+1}{n+1}\right)x.P_n(x) - \left(\frac{n}{n+1}\right)P_{n-1}(x) \end{aligned} \tag{A.4}$$

Using the code below these functions can be evaluated.

```
def Legendre(N,x):
    Pnm1 = 1;
    Pn = x;
    Pnp1 = 0;

    if (N==0):
        return 1;

    if (N==1):
        return x;

    for n in range(2,N+1):
        ns=n-1
        Pnp1 = ((2*ns+1)/(ns+1))*x*Pn - (ns/(ns+1))*Pnm1;
        Pnm1 = Pn;
        Pn = Pnp1;

    return Pnp1
```

The other unknown in equation A.3 is the abscissae x_n which are the roots of the Legendre polynomial equations. An algorithm for finding these roots is given by Barrerra-Figueroa [1]. This algorithm utilizes Newton's method for finding a root and therefore we need to also have a function for finding the derivate of the Legendre polynomials

$$P'_n(x) = \frac{nx}{x^2-1}.P_n(x) - \frac{n}{x^2-1}P_{n-1}(x) \tag{A.5}$$

The code below obtains the derivative $P'_n(x)$:

```
def dLegendredx(N,x):
    if (N==0):
        return 0;

    if (N==1):
        return 1;

    return (N*x/(math.pow(x,2)-1))*Legendre(N,x)- \
        (N/(math.pow(x,2)-1))*Legendre(N-1,x);
```

Finally applying the root finding equation in [1]

$$x_k^{(\ell+1)} = x_k^{(\ell)} - \frac{f(x_k^{(\ell)})}{f'(x_k^{(\ell)}) - f(x_k^{(\ell)}) \sum_{j=1}^{k-1} \frac{1}{x_k^{(\ell)} - x_j}} \tag{A.6}$$

We get the code

```
def LegendreRoots(N,maxiters=1000,tol=1.0e-10):
    xn = np.linspace(-0.999,0.999,N); #Initial guessed values

    print("Initial guess:")
    print(xn)

    wn = np.zeros((N));

    for k in range(0,N):
        print("Finding root %d of %d" % (k+1,N), end='')
        i=0;
        while (i<maxiters):
            xold = xn[k]
            a = Legendre(N,xold)
            b = dLegendredx(N,xold)
            c = 0;
            for j in range(0,k):
                c=c+(1/(xold-xn[j]))

            xnew = xold - (a/(b-a*c))

            res=abs(xnew-xold)
            xn[k]=xnew

            if (res<tol):
                print('tr',end='')
                break
            i=i+1

        wn[k] = 2*(1-xn[k]*xn[k])/(N+1)/(N+1)/ \
            Legendre(N+1,xn[k])/Legendre(N+1,xn[k])
        print(" root %f, weight=%f, test=%f" %(xn[k],wn[k],Legendre(N,xn[k])))

    return xn,wn;
```

With the result for $P_5(x)$:

```
Finding root 1 of 5 root1 -0.906180, weight=0.236927
Finding root 2 of 5 root1 -0.538469, weight=0.478629
Finding root 3 of 5 root1 0.000000, weight=0.568889
Finding root 4 of 5 root1 0.538469, weight=0.478629
Finding root 5 of 5 root1 0.906180, weight=0.236927
```

A.2 Gauss-Chebyshev quadrature rule

The *Gauss-Chebyshev* quadrature rule is used when the weighting is $w(x) = \frac{1}{\sqrt{1-x^2}}$ and therefore the integral is of the form

$$\int_{-1}^1 \frac{f(x)}{\sqrt{1-x^2}}.dx \approx \sum_{n=1}^N w_n f(x_n).$$

Here the abscissae x_n are the roots of the Chebyshev polynomials of the second kind which have an explicit solution:

$$x_n = \cos\left(\frac{(2n-1)\pi}{2N}\right). \quad (\text{A.7})$$

The quadrature weights are given by the simple relation

$$w_n = \frac{\pi}{N}. \quad (\text{A.8})$$

The code below is a simple implementation of these formulas

```
def ChebyshevRoots(N):
    xn = np.linspace(-1,1,N)
    wn = np.linspace(-1,1,N)

    for n in range(0,N):
        ns=n+1
        xn[n]=math.cos((2*ns-1)*math.pi/2/N)
        wn[n]=math.pi/N

        print("Finding root %d of %d, root=%f, weight=%f" %(n+1,N,xn[n],wn[n]))

    return xn,wn
```

With the result for $U_5(x)$:

```
Finding root 1 of 5, root=0.951057, weight=0.628319
Finding root 2 of 5, root=0.587785, weight=0.628319
Finding root 3 of 5, root=0.000000, weight=0.628319
Finding root 4 of 5, root=-0.587785, weight=0.628319
Finding root 5 of 5, root=-0.951057, weight=0.628319
```

A.3 Application to Discrete Ordinates

We can find the polar angles associated with the polar quadrature directly from our earlier definition

$$g(\varphi) = \int_{-1}^1 f(\varphi, \mu) \cdot d\mu \approx \sum_{i=0}^{2N_p-1} w_i f(\varphi, \mu_i)$$

where N_p is the number of polar quadrature points/angles in the first octant. The abscissae μ_i are obtained from the roots of the Legendre polynomial P_{2N_p} and the weights are

$$w_i = \frac{2(1-\mu_i^2)}{(i+2)^2(P_{i+2}(\mu_i))^2}. \quad (\text{A.9})$$

The angles associated with the abscissae are then

$$\theta_i = \cos^{-1} \mu_i \quad (\text{A.10})$$

A.4 Gauss-Legendre-Legendre product quadrature

The integration over all azimuthal angles requires some thought. In its defined form

$$\int_0^{2\pi} g(\varphi) \cdot d\varphi.$$

it can utilize the Gauss-Legendre quadrature after a change of intervals from $[0, 2\pi]$ to $[-1, 1]$ as

$$\begin{aligned} \int_0^{2\pi} g(\varphi) \cdot d\varphi &= \frac{2\pi-0}{2} \int_{-1}^1 g\left(\frac{2\pi-0}{2}y + \frac{2\pi+0}{2}\right) \cdot dy \\ &= \pi \int_{-1}^1 g(\pi y + \pi) \cdot dy \\ &\approx \sum_{j=0}^{4N_a-1} w_j g(\pi y_j + \pi) \end{aligned}$$

where again N_a is the amount of azimuthal angles per octant and the quadrature points are the abscissae of the Legendre polynomial P_{4N_a} and the weights are

$$w_j = \frac{2\pi(1-y_j^2)}{(j+2)^2(P_{j+2}(y_j))^2}. \quad (\text{A.11})$$

The angles associated with the abscissae are then

$$\varphi_j = \pi y_j + \pi \quad (\text{A.12})$$

and we now have a product quadrature of the form

$$\int_0^{2\pi} \int_0^\pi f(\varphi, \theta) \sin\theta d\theta d\varphi \approx \sum_{j=0}^{4N_a-1} w_j \left[\sum_{i=0}^{2N_p-1} w_i f(\varphi_j, \theta_i) \right]. \quad (\text{A.13})$$

An example of this quadrature is shown for the first octant in Figure A1.1 where the colors and dot size indicate the quadrature weight and the dot's position indicates the quadrature point on the unit sphere.

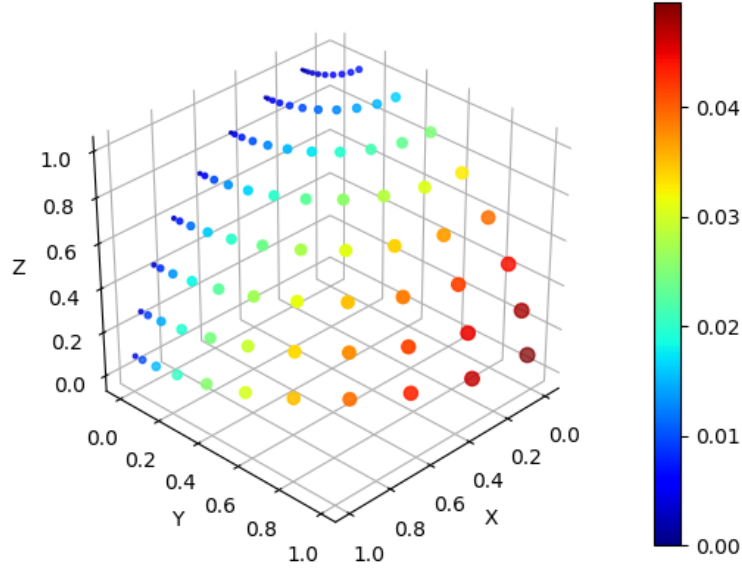


Figure A1.1: Quadrature points and weights (colors) for the Gauss-Legendre quadrature set for both the polar and azimuthal angles with $N_a = 8$ and $N_p = 8$.

A.5 Gauss-Legendre-Chebyshev product quadrature

Instead of changing the intervals of the integration over the azimuthal angles φ we can instead look towards utilizing these intervals by defining

$$y = \cos\left(\frac{\varphi}{2}\right)$$

and

$$\frac{dy}{d\varphi} = -\frac{1}{2}\sin\left(\frac{\varphi}{2}\right).$$

Therefore

$$\begin{aligned} d\varphi &= -\frac{2}{\sin\left(\frac{\varphi}{2}\right)}.dy \\ &= -\frac{2}{\sqrt{1-\cos^2\left(\frac{\varphi}{2}\right)}}.dy \\ \therefore d\varphi &= -\frac{2}{\sqrt{1-y^2}}.dy \end{aligned}$$

which can be used in equation A.2 as

$$\begin{aligned} \int_0^{2\pi} g(\varphi).d\varphi &= \int_1^{-1} -2\frac{g(2\cos^{-1}y)}{\sqrt{1-y^2}}.dy \\ &= 2 \int_{-1}^1 \frac{g(2\cos^{-1}y)}{\sqrt{1-y^2}}.dy \end{aligned}$$

and when we define, for simplicity, $h(y) = g(2\cos^{-1}y)$ we get a familiar form

$$\int_0^{2\pi} g(\varphi).d\varphi = 2 \int_{-1}^1 \frac{h(y)}{\sqrt{1-y^2}}.dy. \quad (\text{A.14})$$

This equation can be approximated with a Gauss-Chebyshev quadrature with abscissae

$$y_j = \cos\left(\frac{(2j+1)\pi}{8N_a}\right) \quad (\text{A.15})$$

and equal weights

$$w_j = \frac{2\pi}{4N_a} \quad (\text{A.16})$$

where N_a is the number of quadrature points per octant, to get the quadrature formula

$$\int_{-1}^1 \frac{h(y)}{\sqrt{1-y^2}}.dy \approx \sum_{j=0}^{4N_a-1} w_j h(y_j). \quad (\text{A.17})$$

The angles associated with the abscissae are then

$$\varphi_j = \frac{(2j+1)\pi}{8N_a} \quad (\text{A.18})$$

and we now have a product quadrature of the form

$$\int_0^{2\pi} \int_0^\pi f(\varphi, \theta) \sin\theta d\theta d\varphi \approx \sum_{j=0}^{4N_a-1} w_j \left[\sum_{i=0}^{2N_p-1} w_i f(\varphi_j, \theta_i) \right]. \quad (\text{A.19})$$

An example of this quadrature is shown for the first octant in Figure A1.2 where the colors and dot size indicate the quadrature weight and the dot's position indicates the quadrature point on the unit sphere.

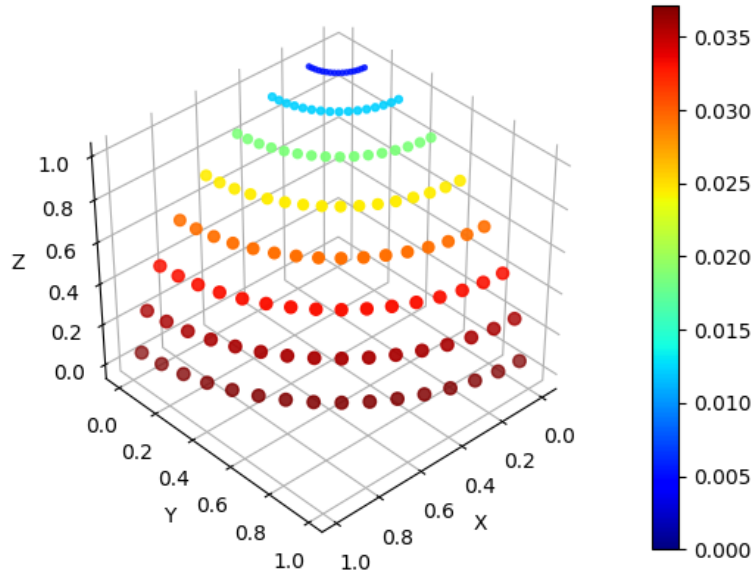


Figure A1.2: Quadrature points and weights (colors) for the Gauss-Legendre quadrature for the polar integration and the Gauss-Chebyshev quadrature for the azimuthal angles with $N_a = 8$ and $N_p = 8$.

A.6 Evaluation of product quadratures

With the quadrature candidates established we can now look at their performance. The application of the product quadratures is done to calculate the integral

$$\int_{4\pi} \Psi_g(r, \hat{\Omega}) Y_{\ell m^*}(\hat{\Omega}) d\hat{\Omega}$$

where the angular flux $\Psi(r, \hat{\Omega})$ is known at the point where the integration is to be performed. Hence we are essentially looking for a function to integrate the spherical harmonic $Y_{\ell m^*}(\hat{\Omega})$ where, repeated here

$$Y_{\ell m^*}(\theta, \varphi) = (-1)^m Y_{\ell(-m)}(\theta, \varphi)$$

$$Y_{\ell m}(\theta, \varphi) = \begin{cases} (-1)^m \sqrt{(2)} \sqrt{\frac{(2\ell+1)}{4\pi} \frac{(\ell-|m|)!}{(\ell+|m|)!}} P_{\ell}^{|m|}(\cos\theta) \sin |m|\varphi & \text{if } m < 0 \\ (-1)^m \sqrt{(2)} \sqrt{\frac{(2\ell+1)}{4\pi} \frac{(\ell-m)!}{(\ell+m)!}} P_{\ell}^m(\cos\theta) \cos m\varphi & \text{if } m \geq 0 \end{cases} \quad (\text{A.20})$$

where the associated Legendre polynomials, P_{ℓ}^m can be determined from

$$\begin{aligned} P_0^0 &= 1, & P_1^0 &= x, \\ P_{\ell}^{\ell} &= -(2\ell-1)\sqrt{1-x^2} P_{\ell-1}^{\ell-1}(x) \quad \text{and} \\ (\ell-m)P_{\ell}^m &= (2\ell-1)x P_{\ell-1}^m(x) - (\ell+m-1)P_{\ell-2}^m(x). \end{aligned} \quad (\text{A.21})$$

An algorithm implementation to evaluate these associated Legendre polynomials is shown below.

```
def AssociatedLegendre(ell, m, x):
    if (abs(m) > ell):
        return 0;

    #====m=0, l=1
    Pn = x;

    #====m=1, l=1
    Pnpos = -math.sqrt(1-math.pow(x, 2));
    #====m=-1, l=1
    Pnneg = -0.5*Pnpos;

    #====m=0, l=0
    if (ell==0):
        return 1;

    if (ell==1):
        if (m==-1):
            return Pnneg;
        if (m==0):
            return Pn;
        if (m==1):
            return Pnpos;

    Pmlp1 = 0
    if (ell==m):
        Pmlp1 = -(2*ell-1)*math.sqrt(1-math.pow(x, 2.0))* \
            AssociatedLegendre(ell-1, ell-1, x)
```

```

else:
    Pmlp1 = (2*ell-1)*x*AssociatedLegendre(ell-1,m,x);
    Pmlp1 = Pmlp1 - (ell+m-1)*AssociatedLegendre(ell-2,m,x)
    Pmlp1 = Pmlp1 / (ell-m)

return Pmlp1

```

An algorithm implementation of the tesseral spherical harmonics is shown below.

```

=====
def fac(x):
    if (x==0):
        return 1
    if (x==1):
        return 1

    return fac(x-1)*x;

=====
def Min1powerm(m):
    if (m==0):
        return 1;
    if ((m%2)==0):
        return 1
    else:
        return -1

=====
def Ylm(ell,m,theta,varphi):
    if (m<0):
        return Min1powerm(m)*math.sqrt( \
            ( (2*ell+1)/(2*math.pi) ) * \
            fac(ell-abs(m))/fac(ell+abs(m)) ) * \
            AssociatedLegendre(ell,abs(m),math.cos(theta))* \
            math.sin(abs(m)*varphi)

    else:
        return Min1powerm(m)*math.sqrt( \
            ( (2*ell+1)/(2*math.pi) ) * \
            fac(ell-m)/fac(ell+m) ) * \
            AssociatedLegendre(ell,m,math.cos(theta))* \
            math.cos(m*varphi)

```

Appendix B More on Spherical Harmonics

B.1 Expansion of a function of two angles $f(\varphi, \theta)$

We seek to expand an angularly dependent function $f(\varphi, \theta)$. Why? Don't know yet, but one such expansion from fundamental theory is spherical harmonics:

$$f(\varphi, \theta) = \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} f_{\ell}^m Y_{\ell}^m(\varphi, \theta). \quad (\text{B.1})$$

This expansion will be very unfamiliar for engineers since most of the scientific computing in common disciplines deal with cartesian coordinates. Unfortunately it will be nearly impossible to explain the development of an expansion of an angular function in spherical harmonics without first exploring the means to calculate its unknowns. To this end let us begin with stating that **there are two flavors of spherical harmonics**. The regular form Y_{ℓ}^m , which contains complex numbers (challenging to handle), and a real form $Y_{\ell m}$. The unknowns in equation B.1 has a trail of components the first of which is

$$\begin{aligned} f_{\ell}^m &= \int_{\hat{\Omega}} f(\varphi, \theta) Y_{\ell}^{m*}(\varphi, \theta) d\hat{\Omega} \\ &= \int_0^{2\pi} \int_0^{\pi} f(\varphi, \theta) Y_{\ell}^{m*}(\varphi, \theta) \sin \theta d\theta d\varphi. \end{aligned}$$

The reader should try to comprehend that the f_{ℓ}^m components are almost never computed in this form since doing so means one already has a analytical representation of $f(\varphi, \theta)$. Additionally we have

$$Y_{\ell}^m(\varphi, \theta) = (-1)^m \sqrt{\frac{(2\ell+1)}{4\pi} \frac{(\ell-m)!}{(\ell+m)!}} P_{\ell}^m(\cos \theta) e^{(m\varphi)i} \quad (\text{B.2})$$

and its complex conjugate

$$Y_{\ell}^{m*}(\varphi, \theta) = (-1)^m Y_{\ell}^{(-m)}(\varphi, \theta).$$

This form of the spherical harmonics functions can be very unruly and therefore its more common place to calculate them from the real forms as

$$Y_{\ell}^m(\varphi, \theta) = \begin{cases} \frac{1}{\sqrt{2}}(Y_{\ell|m|} - iY_{\ell, -|m|}) & \text{if } m < 0 \\ Y_{\ell 0} & \text{if } m = 0 \\ \frac{(-1)^m}{\sqrt{2}}(Y_{\ell|m|} + iY_{\ell, -|m|}) & \text{if } m > 0 \end{cases} \quad (\text{B.3})$$

Here the real forms are represented by:

$$Y_{\ell m}(\theta, \varphi) = \begin{cases} (-1)^m \sqrt{(2)} \sqrt{\frac{(2\ell+1)}{4\pi} \frac{(\ell-|m|)!}{(\ell+|m|)!}} P_{\ell}^{|m|}(\cos \theta) \sin |m|\varphi & \text{if } m < 0 \\ \sqrt{\frac{(2\ell+1)}{4\pi}} P_{\ell}^0(\cos \theta) & \text{if } m = 0 \\ (-1)^m \sqrt{(2)} \sqrt{\frac{(2\ell+1)}{4\pi} \frac{(\ell-m)!}{(\ell+m)!}} P_{\ell}^m(\cos \theta) \cos m\varphi & \text{if } m \geq 0 \end{cases} \quad (\text{B.4})$$

Finally the associated Legendre polynomials, P_ℓ^m can be determined from

$$\begin{aligned} P_0^0 &= 1, & P_1^0 &= x, \\ P_\ell^\ell &= -(2\ell-1)\sqrt{1-x^2} P_{\ell-1}^{\ell-1}(x) \quad \text{and} \\ (\ell-m)P_\ell^m &= (2\ell-1)x P_{\ell-1}^m(x) - (\ell+m-1)P_{\ell-2}^m(x). \end{aligned} \tag{B.5}$$

With all these unknowns we see that before we can calculate the expansion we need to choose the maximum order $L = \ell_{max}$ after which we need to compute each P_ℓ^m , each $Y_{\ell m}$, each Y_ℓ^m and each Y_ℓ^{m*} . Only then can we compute f_ℓ^m . If we do all the work this way we can approximate some functions.

B.2 Prototype code for spherical harmonics

We begin with code to compute the associated Legendre polynomials

```
def AssociatedLegendre(ell,m,x):
    if (abs(m)>ell):
        return 0;

    #====m=0,l=1
    Pn = x;

    #====m=1,l=1
    Pnpos= -math.sqrt(1-math.pow(x,2));
    #====m=-1,l=1
    Pnneg= -0.5*Pnpos;

    #====m=0,l=0
    if (ell==0):
        return 1;

    if (ell==1):
        if (m==-1):
            return Pnneg;
        if (m==0):
            return Pn;
        if (m==1):
            return Pnpos;

    Pmlp1 = 0
    if (ell==m):
        Pmlp1 = -(2*ell-1)*math.sqrt(1-math.pow(x,2.0))* \
            AssociatedLegendre(ell-1,ell-1,x)
    else:
        Pmlp1 = (2*ell-1)*x*AssociatedLegendre(ell-1,m,x);
        Pmlp1 = Pmlp1 - (ell+m-1)*AssociatedLegendre(ell-2,m,x)
        Pmlp1 = Pmlp1 / (ell-m)

    return Pmlp1
```

We then depict code to calculate the real forms of the spherical harmonics

```

=====
def fac(x):
    if (x==0):
        return 1
    if (x==1):
        return 1

    return fac(x-1)*x;

=====
def Min1powerm(m):
    if (m==0):
        return 1;
    if ((m%2)==0):
        return 1
    else:
        return -1

=====
def Ylm(ell,m,varphi,theta):
    if (m<0):
        return Min1powerm(m)*math.sqrt( \
            ( (2*ell+1)/(2*math.pi) ) * \
            fac(ell-abs(m))/fac(ell+abs(m)) ) * \
            AssociatedLegendre(ell,abs(m),math.cos(theta))* \
            math.sin(abs(m)*varphi)

    elif (m==0):
        return math.sqrt( \
            ( (2*ell+1)/(4*math.pi) ) * \
            fac(ell-m)/fac(ell+m) ) * \
            AssociatedLegendre(ell,m,math.cos(theta))* \
            math.cos(m*varphi)

    else:
        return Min1powerm(m)*math.sqrt( \
            ( (2*ell+1)/(2*math.pi) ) * \
            fac(ell-m)/fac(ell+m) ) * \
            AssociatedLegendre(ell,m,math.cos(theta))* \
            math.cos(m*varphi)

```

And then the complex form of the spherical harmonics

```

=====
def Yl_m(ell,m,varphi,theta):
    v = 0+0j;
    if (m<0):
        v = (1/math.sqrt(2))*complex(Ylm(ell,abs(m),varphi,theta), - \
            Ylm(ell,-abs(m),varphi,theta))

    elif (m==0):
        v = complex(Ylm(ell,0,varphi,theta),0)
    else:
        v = Min1powerm(m)* \
            (1/math.sqrt(2))*complex(Ylm(ell,abs(m),varphi,theta), \
            Ylm(ell,-abs(m),varphi,theta))

    return v

```

From here we need to have a means to compute the integral

$$\begin{aligned} f_\ell^m &= \int_{\hat{\Omega}} f(\varphi, \theta) Y_\ell^{m*}(\varphi, \theta) d\hat{\Omega} \\ &= \int_0^{2\pi} \int_0^\pi f(\varphi, \theta) Y_\ell^{m*}(\varphi, \theta) \sin \theta d\theta d\varphi. \end{aligned}$$

Suppose we have any function of angle F0, F1, F2 and so forth we can define a function that will add the complex conjugate of the spherical harmonics with the simple code

```
#=====
def Flm(ell,m,varphi,theta):
    return F2(varphi,theta)*Minipowerm(m)* \
        (Legendre.Yl_m(ell,-m,varphi,theta))
```

We can then precompute a vector containing all the $f_{\ell m}$ coefficients using either a Riemann integral or a quadrature rule

```
GLC = Legendre.Quadrature()
GLC.InitializeWithGLC(8,8)

#===== Build flm
L = 7
k=-1
flm = np.zeros((L*(L+2)+1),dtype=np.complex_)
for ell in range(0,L+1):
    for m in range(-ell,ell+1):
        k=k+1
        #flm[k] = Legendre.RiemannAngLM(Flm,ell,m)

        #print("l=%f, m=%f, flm=" %(ell,m), end='')
        #print(flm[k])
        flm[k] = Legendre.QuadratureIntegrateLM(Flm,GLC,ell,m)
        print("l=%f, m=%f, flm=" %(ell,m), end='')
        print(flm[k])
```

Finally we can compute a representation of the expansion over the span of φ using the code

```
#===== Build yil
Ni1=200
varphi1=np.linspace(0,2*math.pi,Ni1)
yi1=np.zeros((Ni1))

for i in range(0,Ni1):
    yi1[i]= 0;
    v = 0+0j
    k=-1;
    for ell in range(0,L+1):
        for m in range(-ell,ell+1):
            k=k+1
            v=v+(flm[k])* \
                (Legendre.Yl_m(ell,m,varphi1[i],math.pi/2))
            yi1[i] = v.real
```


B.2.1 Approximating an isotropic flux

The simplest function to represent is an isotropic flux (i.e. $f(\varphi, \theta) = 1$). Such a function is perfectly captured with $L = 0$, i.e. a single expansion, as shown in Figure B2.1. This is not surprising since the combination of spherical harmonics with order and moment zero results in $\sqrt{\frac{1}{4\pi}} \times \sqrt{\frac{1}{4\pi}}$ which integrates to unity and hence the original function is recovered.

```
def F0(varphi, theta):
    return 1
```

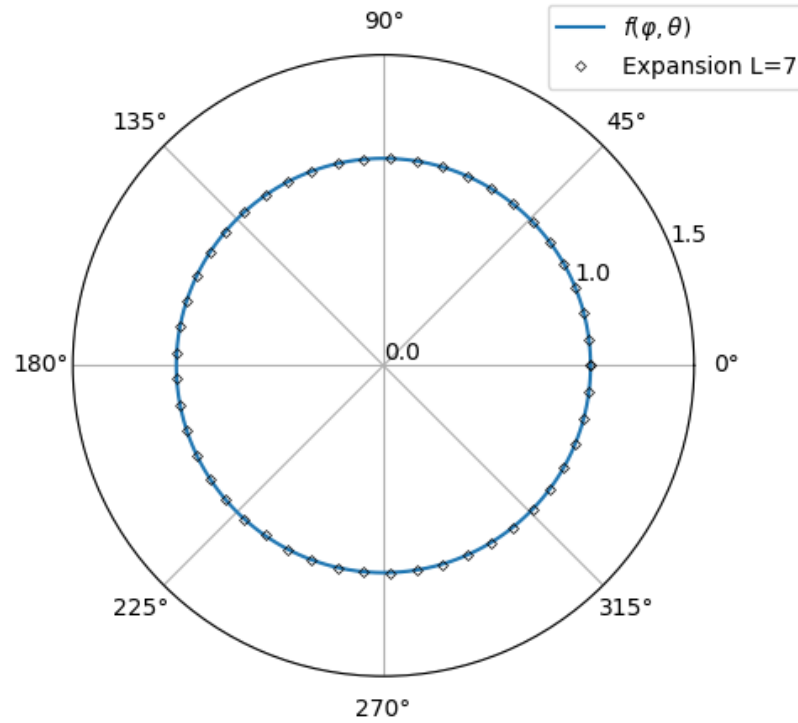


Figure B2.1: Approximation of a pure isotropic function with spherical harmonics. The plot is shown for the azimuthal angle φ only.

B.2.2 Approximating an anisotropic but smooth flux

We can construct an anisotropic function of azimuthal angle as

$$f(\varphi, \theta) = 1 + \cos(4\varphi)$$

```
def F1(varphi, theta):
    return 1.0 + 0.1 * math.cos(varphi * 4)
```

Such a function requires a few more orders of spherical harmonics in order to capture the shape and, as shown in Figure B2.2, $L = 7$ closely resembles the shape. A function of this shape could appear in a fuel assembly lattice where the effective scattering and absorption is a strong function of azimuthal angle.

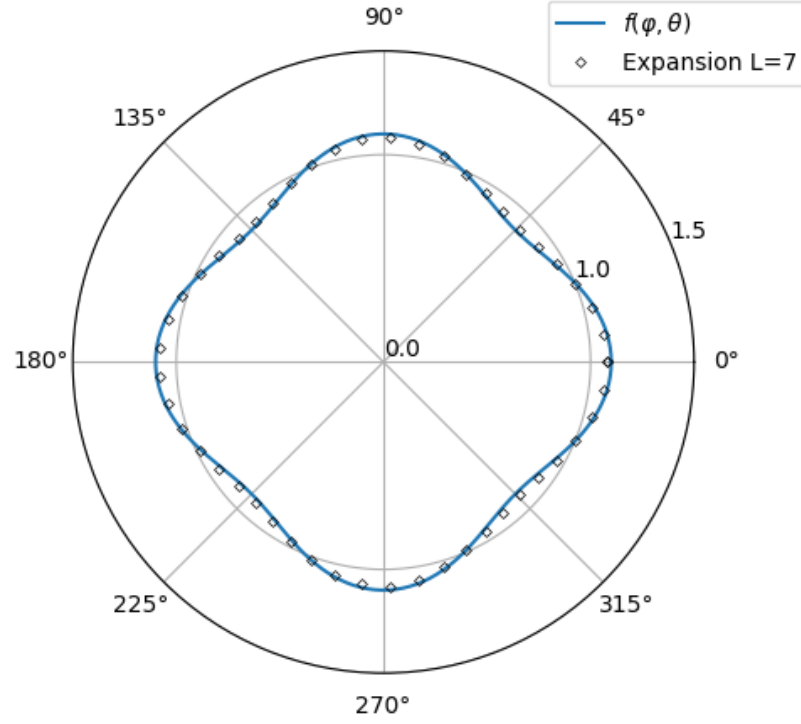


Figure B2.2: Approximation of an anisotropic smooth function with spherical harmonics. The plot is shown for the azimuthal angle φ only. The radial dimension represents the flux magnitude.

B.2.3 Approximating a directional flux (i.e. anisotropic + not-smooth)

As a final consideration we try to construct a function that is very angular, like a beam. Such a function of angle could be

$$f(\varphi, \theta) = \begin{cases} \frac{2}{10} & \text{if } \varphi < \frac{7}{8}\pi \\ \frac{2}{10} + \frac{6}{5} \cos(4\varphi) & \text{if } \frac{7}{8}\pi \leq \varphi \leq \frac{9}{8}\pi \\ \frac{2}{10} & \text{if } \varphi > \frac{9}{8}\pi \end{cases}$$

```
def F2(varphi, theta):
    if (varphi < (7*math.pi/8)):
        return 0.2;
    if (varphi > (9*math.pi/8)):
        return 0.2;

    return 1.2*math.cos(varphi*4)+0.2
```

As expected a total number of 12 spherical harmonic orders are required to accurately represent such a directional flux (see Figure B2.3). An additional 2D plot is shown in Figure B2.4 which more clearly shows the oscillations of the expansion at the directions not aligned with the directional nature of the function.

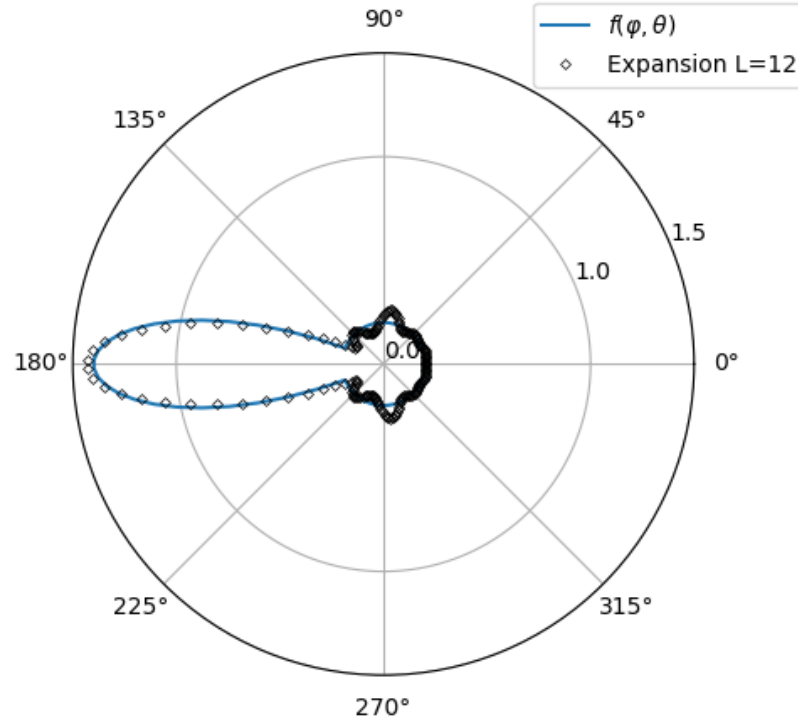


Figure B2.3: Approximation of an anisotropic smooth function with spherical harmonics. The plot is shown for the azimuthal angle φ only. The radial dimension represents the flux magnitude.

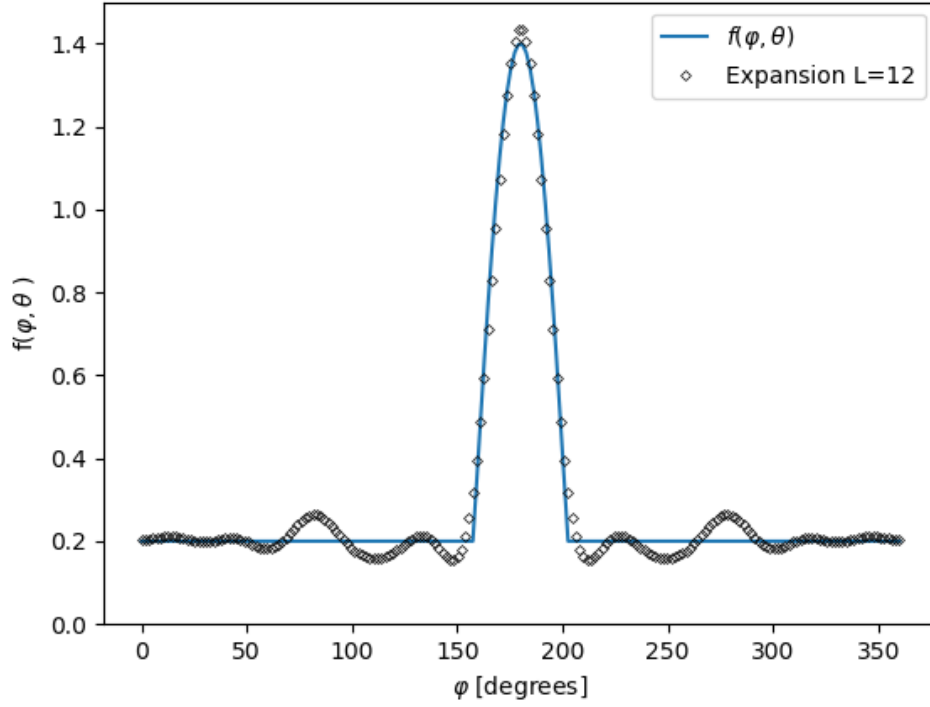


Figure B2.4: Approximation of an anisotropic smooth function with spherical harmonics. The plot is shown for the azimuthal angle φ only.

B.3 Prototype code for real form of the spherical harmonics

For the real form of the spherical harmonics we have a slightly modified real form

$$Y_{\ell m}(\theta, \varphi) = \begin{cases} \sqrt{(2)}\sqrt{\frac{(2\ell+1)}{4\pi}}\frac{(\ell-|m|)!}{(\ell+|m|)!}P_{\ell}^{|m|}(\cos\theta)\sin|m|\varphi & \text{if } m < 0 \\ \sqrt{\frac{(2\ell+1)}{4\pi}}P_{\ell}^m(\cos\theta) & \text{if } m = 0 \\ \sqrt{(2)}\sqrt{\frac{(2\ell+1)}{4\pi}}\frac{(\ell-m)!}{(\ell+m)!}P_{\ell}^m(\cos\theta)\cos m\varphi & \text{if } m \geq 0 \end{cases} \quad (\text{B.6})$$

and the expansion coefficients are also different

$$f(\varphi, \theta) = \sum_{\ell=0}^{\infty} \sum_{m=-\ell}^{\ell} f_{\ell m} Y_{\ell m}(\varphi, \theta) \quad (\text{B.7})$$

where

$$f_{\ell m} = \int_0^{2\pi} \int_0^{\pi} f(\varphi, \theta) Y_{\ell m}(\varphi, \theta) \sin\theta \, d\theta \, d\varphi$$

for which the code is

```

=====
def Ylmcoeff(ell,m,varphi,theta):
    if (m<0):
        return math.sqrt( \
            ( (2*ell+1)/(2*math.pi) ) * \
            fac(ell-abs(m))/fac(ell+abs(m)) ) * \
            AssociatedLegendre(ell,abs(m),math.cos(theta))* \
            math.sin(abs(m)*varphi)
    elif (m==0):
        return math.sqrt( \
            ( (2*ell+1)/(4*math.pi) ) * \
            fac(ell-m)/fac(ell+m) ) * \
            AssociatedLegendre(ell,m,math.cos(theta))* \
            math.cos(m*varphi)
    else:
        return math.sqrt( \
            ( (2*ell+1)/(2*math.pi) ) * \
            fac(ell-m)/fac(ell+m) ) * \
            AssociatedLegendre(ell,m,math.cos(theta))* \
            math.cos(m*varphi)

```

And

```

def Flm(ell,m,varphi,theta):
    return F3(varphi,theta)* \
        (Legendre.Ylmcoeff(ell,m,varphi,theta))

===== Build flm
L =12
k=-1
flm = np.zeros((L*(L+2)+1),dtype=np.complex_)
for ell in range(0,L+1):
    for m in range(-ell,ell+1):
        k=k+1
        #flm[k] = Legendre.RiemannAngLM(Flm,ell,m)

        #print("l=%f, m=%f, flm=" %(ell,m), end='')
        #print(flm[k])
        flm[k] = Legendre.QuadratureIntegrateLM(Flm,GLC,ell,m)
        print("l=%f, m=%f, flm=" %(ell,m), end='')
        print(flm[k])

===== Build yil
Ni1=400
varphi1=np.linspace(0,2*math.pi,Ni1)
yi1=np.zeros((Ni1))

for i in range(0,Ni1):
    yi1[i]= 0;
    v = 0
    k=-1;
    for ell in range(0,L+1):
        for m in range(-ell,ell+1):
            k=k+1
            v=v+(flm[k])* \
                (Legendre.Ylmcoeff(ell,m,varphi1[i],math.pi/2))
            yi1[i] = v

```

Appendix C Creating simple materials for testing

Neutral particle transport involves three basic processes:

- Absorption. The elimination of the particle from a current group
- Scattering. Change in angle and group essentially removing the particle from the group-angle pair.
- Source. Both in the form of a fixed source and as reactions to absorption processes (i.e. (n,n), (n,2n), (n,fission), etc.)

We also have a fundamental definition that the total removal process is the sum of the absorption process and the scattering process. In terms of nuclide cross-sections we can write this as

$$\sigma_t = \sigma_a + \sigma_s$$

Where σ_t , σ_a and σ_s represent the total-, absorption- and scattering cross-sections respectively.

C.1 Simple neutron-nuclide scattering processes

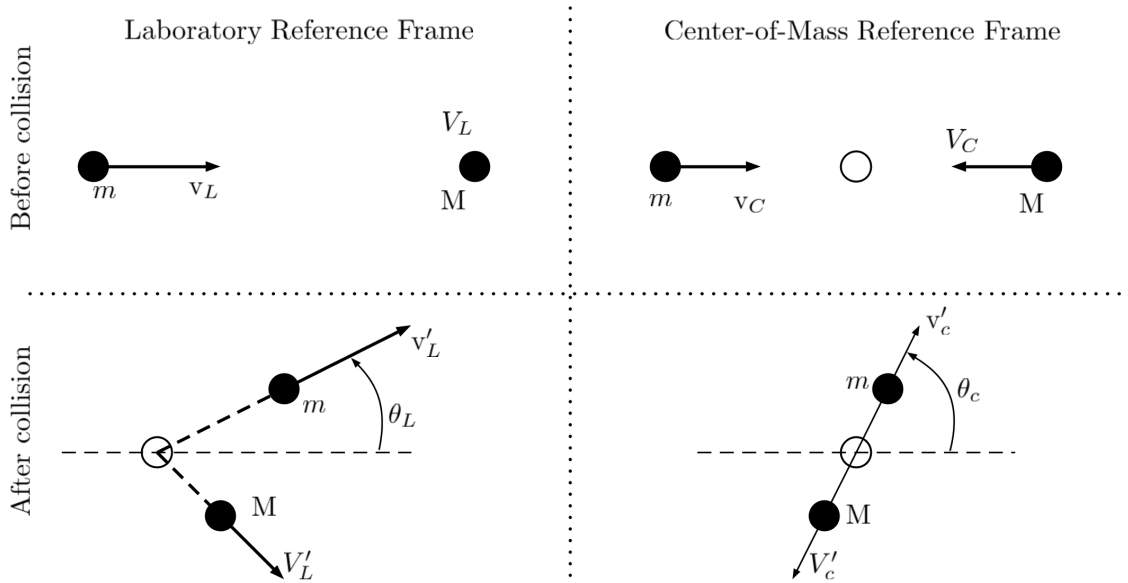


Figure C.1: Collision kinematics of a stationary nuclide in both the laboratory reference frame and the center-of-mass reference frame.

Let us consider a stationary target nucleus X_Z^A suspended in space and a neutron moving towards this nucleus (from left to right) at velocity v_L , where L denotes the laboratory reference frame (i.e. the one we

are living in) as denoted in Figure C.1. Assuming the target nucleus is at rest (an invalid assumption that will be treated later) with velocity V_L we know the energies associated with these particles to be

$$E_L = \frac{1}{2}mv_L^2$$

$$E_{L_A} = \frac{1}{2}MV_L^2$$

where E_L is the energy of the neutron and E_{L_A} is the energy of the target nucleus, both in the laboratory reference frame, and $m = 1$ is the mass of the neutron and $M = A$ is the mass of the target nucleus. Fortunately the derivation of the mass-momentum equations relating the center-of-mass energies and angles to the laboratory reference frame quantities are comprehensively depicted in the textbook by Duderstadt and Hamilton [3]. In this book the scattering angle in the laboratory reference frame, θ_L , is related to the scattering angle in the center-of-mass reference frame, θ_c , as

$$\tan \theta_L = \frac{\sin \theta_c}{\frac{1}{A} + \cos \theta_c}. \quad (C.1)$$

Associated with this, [3] also derives the neutron energy change $E_L \rightarrow E'_L$ as

$$E'_L = \left[\frac{(1+\alpha) + (1-\alpha) \cos \theta_c}{2} \right] E_L \quad (C.2)$$

where $\alpha = (\frac{A-1}{A+1})^2$. For light nuclei, where one can assume the scattering angle in the center-of-mass frame is isotropic [3] we can determine the probability distribution function for scattering through an angle θ_L . From equation C.1 we can get θ_L as

$$\theta_L = \begin{cases} \pi + \cos^{-1} \left(\frac{\sin \theta_c}{\frac{1}{A} + \cos \theta_c} \right) & \text{if } (\frac{1}{A} + \cos \theta_c) < 0 \\ \frac{\pi}{2} & \text{if } (\frac{1}{A} + \cos \theta_c) = 0 \\ \cos^{-1} \left(\frac{\sin \theta_c}{\frac{1}{A} + \cos \theta_c} \right) & \text{if } (\frac{1}{A} + \cos \theta_c) > 0 \end{cases} \quad (C.3)$$

Since $\cos \theta_c$, $\theta_c \in [0, \pi]$, is essentially our cumulative probability when linearly mapped to $[0, 1]$ we require the inverse of equation C.1. Therefore we begin by setting $x = \cos \theta_c$ and inserting it into equation C.3

$$\tan \theta_L = \frac{\sqrt{1-x^2}}{\frac{1}{A} + x}$$

and for simplicity we set the unknowns to constants

$$C = \frac{\sqrt{1-x^2}}{B+x}$$

$$C(B+x) = \sqrt{1-x^2}$$

$$C^2(x^2+2Bx+B^2) = 1-x^2$$

$$C^2x^2+2BC^2x+B^2C^2 = 1-x^2$$

$$(C^2+1)x^2+2BC^2x+B^2C^2-1 = 0$$

which is now in the familiar form $ax^2+bx+c=0$ for which we complete the square to find

$$x = \frac{-2BC^2 \pm \sqrt{4B^2C^4 - 4(C^2+1)(B^2C^2-1)}}{2(C^2+1)}$$

$$\therefore x = \frac{-2BC^2 \pm 2C\sqrt{1-B^2+\frac{1}{C^2}}}{2(C^2+1)}.$$

The two possible x values obtained this way was incurred because we applied a square to remove the square-root term and therefore will give us the angle corresponding to $\tan \theta_L$ as well as $-\tan \theta_L$. Since we know that we started with a positive $\tan \theta_L$ we are only interested in the solution

$$\cos \theta_c = x = f(\theta_L) = g(\mu) = \frac{-2BC^2 + 2C\sqrt{1-B^2+\frac{1}{C^2}}}{2(C^2+1)}. \quad (C.4)$$

And therefore our mapping to a cumulative probability distribution becomes

$$\int_{-1}^1 P(\mu) d\mu = \frac{g(\mu)-1}{2} \quad (C.5)$$

where $B=1/A$ and $C=\tan(\cos^{-1}\mu)$. The cumulative probability function for different masses of the target nucleus is shown in Figure C.2. Obtaining the probability density function, $P(\mu)$ is then a simple differentiation that can be done numerically

$$P(\mu) = \frac{1}{2} \frac{dg}{d\mu} \quad (C.6)$$

for which the results are shown in Figure C.3. The algorithm applied to find $P(\mu)$ is shown below

```

=====
def ThetaC(thetaL,A):
    B = 1/A
    if (thetaL == (math.pi/2)):
        C=1
    else:
        C = min(math.tan(thetaL),1.0e6)

    root = (1-B**2+1/(C**2))

    x1 = (-2*B*C**2 + 2*C*math.sqrt(root))/2/(C**2+1)
    #x2 = (-2*B*C**2 - 2*C*math.sqrt(root))/2/(C**2+1)

    #Safety catches
    if (x1>1.0):
        return math.acos(1)
    if (x1<-1.0):
        return math.acos(-1)

    return math.acos(x1)
    
```



```

===== Probability scattering mu
def Pmu(mu,A):
    thetaLA = math.acos(mu-0.0000001)
    thetaLB = math.acos(mu+0.0000001)
    thetacA = ThetaC(thetaLA,A)
    thetacB = ThetaC(thetaLB,A)
    CPLA    = 0.5-0.5*math.cos(thetacA)
    CPLB    = 0.5-0.5*math.cos(thetacB)
    return -(CPLB - CPLA)/0.0000002
    
```

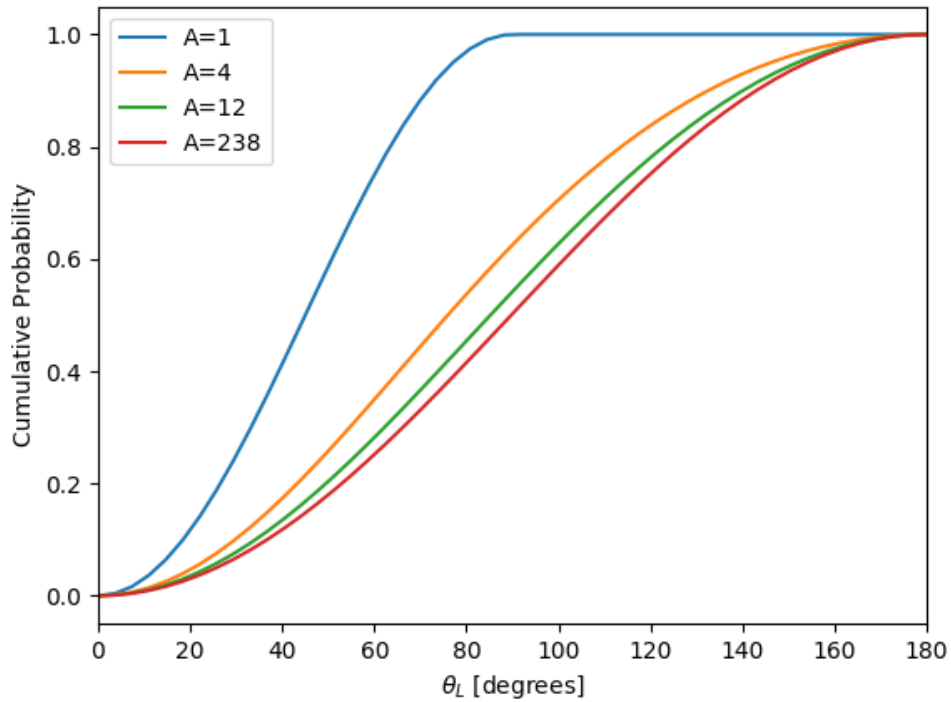


Figure C.2: Cumulative probability distribution for a neutron scattering off a stationary nucleus of mass A .

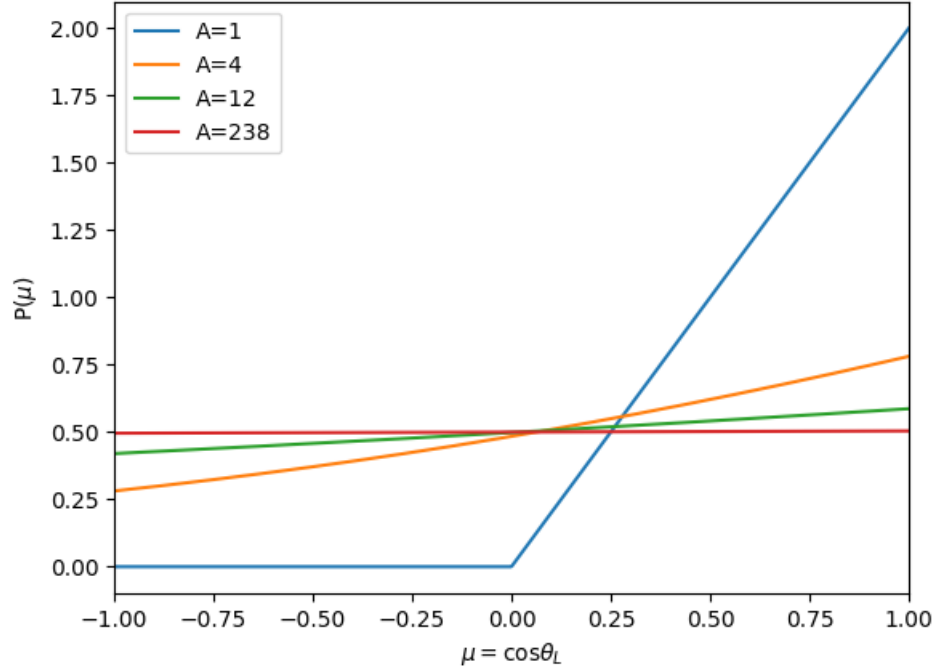


Figure C.3: Probability distribution for a neutron scattering off a stationary nucleus of mass A.

C.2 Combining probabilities

Now, since μ corresponds to a discrete θ_c which also corresponds to a discrete $\frac{E'_L}{E_L}$, the kernel value $K(\mu, E' \rightarrow E)$ also will have only discrete points where it is non-zero, i.e.

$$K(\mu, E' \rightarrow E) = \begin{cases} P(\mu)P(E' \rightarrow E) & , \text{if } E' - \left[\frac{(1+\alpha) + (1-\alpha)g(\mu)}{2} \right] E = 0 \\ 0 & , \text{otherwise} \end{cases}$$

This discrete behavior requires significant numerical effort to resolve, however, multi-group integrations of the source- and destination energy groups alleviates this somewhat since

$$\begin{aligned} K(\mu, E_{g'} \rightarrow E_g) &= \int_{E'_{g+1}}^{E'_g} \int_{E_{g+1}}^{E_g} K(\mu, E' \rightarrow E) . dE . dE' \\ &= P(\mu)P(E_{g'} \rightarrow E_g) \end{aligned}$$

An algorithm to implement this kernel is shown below

```
def Kernel(mu,gprime,g,Eg,A,Ng=1000):
    #== Bin boundaries
    Eiupp = Eg[gprime]
    Eilow = Eg[gprime+1]
    Efupp = Eg[g]
    Eflow = Eg[g+1]

    dEi = (Eiupp - Eilow)/Ng
    binWidth = (Efupp-Eflow)

    sumprobs=0
    thetaL = math.acos(mu)
    thetac = ThetaC(thetaL,A)
    muc = math.cos(thetac)
    for iE in range(0,Ng):
        Ein = Eilow + dEi/2 + dEi*iE
        Eout=Ef(muc,A,Ein)

        if ((Eout<=Efupp) and (Eout>=Eflow)):
            sumprobs = sumprobs + 1/Ng

    return sumprobs*Pmu(mu,A)
```

In order to test the multi-group implementation of this kernel we can build a simple 10 group energy discretization $E \in [0, 1]$ MeV with linearly spaced bins

```
G = 10
Eg = np.linspace(1,0,G+1)
```

The requirement here is that the continuous form obeys

$$\int_{-1}^1 \int_0^\infty K(\mu, E' \rightarrow E).dE.d\mu = 1$$

and therefore the multi-group form must obey

$$\int_{-1}^1 \left(\sum_{g=0}^G K(\mu, E_{g'} \rightarrow E_g) \right).d\mu = 1.$$

The code to implement this integration is

```
Np=100
mu=np.linspace(-0.9999,0.9999,Np)
ydis=np.zeros((Np))
sumofdis=0
sumovergroupsdis=0
for i in range(0,Np):
    for gdes in range(0,G):
        sumovergroupsdis=sumovergroupsdis+ \
            Kernel(mu[i],gprime,gdes,Eg,A)*(2/Np)
```

and proves that the integral is unity as intended. Another test is to integrate over all angle with

$$\begin{aligned}
 & \int_{4\pi} \left(\sum_{g=0}^G K(\Omega' \cdot \Omega, E_{g'} \rightarrow E_g) \right) d\Omega' \\
 &= \int_0^{2\pi} \int_0^\pi \left(\sum_{g=0}^G K(\Omega' \cdot \Omega, E_{g'} \rightarrow E_g) \right) \sin \theta' d\theta' d\varphi' \\
 &= 2\pi
 \end{aligned}$$

where $\Omega' = [\sin \theta' \cos \varphi', \sin \theta' \sin \varphi', \cos \theta']$ and Ω is chosen arbitrarily (i.e. $\Omega = [1, 0, 0]$). The code to compute this integral, using the previous denoted 10-group energy structure, as well as scattering from group 0, is shown below

```

Np = 100
Na = 200
polar = np.linspace(0.0001, math.pi*0.9999, Np)
azimu = np.linspace(0.0001, 2*math.pi*0.99999, Na)
dtheta = (math.pi)/Np
dvarphi = (math.pi*2)/Na

nref = np.array([1.0, 0.0, 0.0])
ndir = np.array([0.0, 0.0, 0.0])

sumprob=0.0
gprime=0
for i in range(0, Na):
    print(i)
    for j in range(0, Np):
        varphi = azimu[i]
        theta = polar[j]

        ndir[0] = math.sin(theta)*math.cos(varphi)
        ndir[1] = math.sin(theta)*math.sin(varphi)
        ndir[2] = math.cos(theta)

        mu = np.dot(ndir, nref)

        for gdes in range(0, G):
            sumprob=sumprob+ \
                Kernel(mu, gprime, gdes, Eg, A)*math.sin(theta)*dtheta*dvarphi
    
```

Indeed this does then integrate to 2π .

C.3 Legendre expansion of the scattering term

The discrete ordinates method involves the expansion of the scattering term using Legendre polynomials as basis functions. This expansion is of the form

$$K(\mu, E_{g'} \rightarrow E_g) = \sum_{\ell=0}^{\infty} \frac{2\ell+1}{2} P_{\ell}(\mu) K_{\ell}(E_{g'} \rightarrow E_g)$$

where the expansion coefficients are given by

$$K_{\ell}(E_{g'} \rightarrow E_g) = \int_{-1}^1 K(\mu, E_{g'} \rightarrow E_g) \cdot P_{\ell}(\mu) \cdot d\mu$$

The code to compute the expansion coefficients requires just a small modification of the multi-group kernel in the sense that the Kernel is multiplied by the Legendre polynomial. The code is shown below

```
def KernelL(ell, gprime, g, Eg, A):
    groupprob=0
    Np=800
    mu=np.linspace(-0.9999,0.9999,Np)
    dmu = (0.9999*2)/Np
    print("Integrating group %d to %d moment %d" %(gprime, g, ell))
    for i in range(0, Np):
        groupprob = groupprob + Kernel(mu[i], gprime, g, Eg, A) * \
            Legendre.Legendre(ell, mu[i]) * dmu

    return groupprob
```

We can now decide to truncate our expansion at the L -th moment and precompute the expansion coefficient $K_{\ell}(E_{g'} \rightarrow E_g)$ with an example scattering from group $g' = 0$ to group $g = 1$ the code is

```
L=10
KL=np.zeros((L+1))
gprime = 0
g=1
sumgroups=0
for ell in range(0, L+1):
    KL[ell] = KernelL(ell, gprime, g, Eg, A)
```

We can now compute the discrete and expanded form as a function of μ for which the code is shown below. A graphical plot of the expanded and discrete form is shown, for different orders of expansion, in Figure C.4.

```

===== Discrete form vs Expansion for Group 0 to 1
Np=100
mu=np.linspace(-0.9999,0.9999,Np)
yexp=np.zeros((Np))
ydis=np.zeros((Np))
sumofdis=0
sumofexp=0
sumovergroupsdis=0
for i in range(0,Np):
    yexp[i] = expansion(L,mu[i],KL)
    ydis[i] = Kernel(mu[i],gprime,g,Eg,A)
    sumofexp=sumofexp+yexp[i]*(2/Np)
    sumofdis=sumofdis+ydis[i]*(2/Np)

    for gdes in range(0,G):
        sumovergroupsdis=sumovergroupsdis+ \
            Kernel(mu[i],gprime,gdes,Eg,A)*(2/Np)
    
```

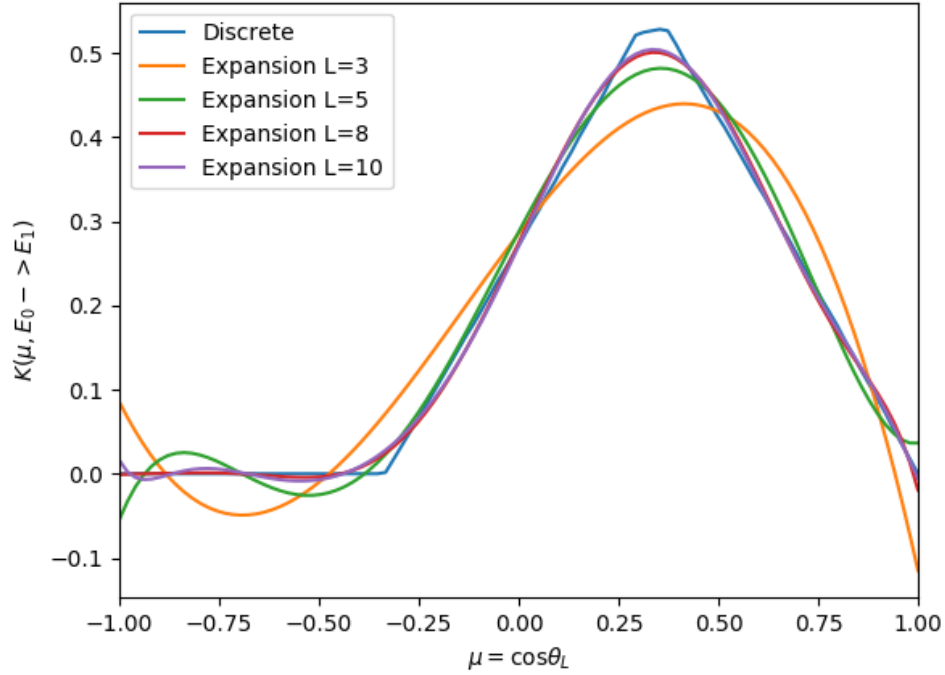


Figure C.4: Kernel function for a neutron scattering off of a stationary carbon nuclear ($A = 12$) and scattering from group 0 to 1.

Appendix D Quadrature rule for integration of triangle space

We seek an integral of a function in triangle space T_{sp} in the form

$$\int \int_{T_{sp}} f(x, y).dx.dy = \sum_{i=0}^{N-1} w_i f(x_i, y_i).$$

Furthermore we know that in the finite element method with only linear shape functions we will at most have polynomials of degree 2 therefore we can devise a set of test functions

$$\begin{aligned} f(x, y) = 1 & \quad \int_0^1 \int_0^{1-y} 1.dx.dy = \frac{1}{2} = \sum_{i=0}^{N-1} w_i \\ f(x, y) = x & \quad \int_0^1 \int_0^{1-y} x.dx.dy = \frac{1}{6} = \sum_{i=0}^{N-1} w_i x_i \\ f(x, y) = y & \quad \int_0^1 \int_0^{1-y} y.dx.dy = \frac{1}{6} = \sum_{i=0}^{N-1} w_i y_i \\ f(x, y) = xy & \quad \int_0^1 \int_0^{1-y} xy.dx.dy = \frac{1}{24} = \sum_{i=0}^{N-1} w_i x_i y_i \\ f(x, y) = x^2 & \quad \int_0^1 \int_0^{1-y} x^2.dx.dy = \frac{1}{12} = \sum_{i=0}^{N-1} w_i x_i^2 \\ f(x, y) = y^2 & \quad \int_0^1 \int_0^{1-y} y^2.dx.dy = \frac{1}{12} = \sum_{i=0}^{N-1} w_i y_i^2 \end{aligned}$$

With $N = 3$ a symmetric solution is obtained with

$$\begin{aligned} w_i &= \frac{1}{6} \\ x_0, y_0 &= \left(\frac{1}{6}, \frac{1}{6}\right) \\ x_1, y_1 &= \left(\frac{4}{6}, \frac{1}{6}\right) \\ x_2, y_2 &= \left(\frac{1}{6}, \frac{4}{6}\right) \end{aligned}$$

which is not a unique solution.

Appendix E Quadrature rule for integration of tetrahedron space

The study of quadratures for tetrahedrons is a deeply mathematical topic one that is outside the scope of this study. As with the two dimensional case, and since we will limit our study to piece-wise linear shape functions we will limit our quadrature set to a minimum degree of precision of 2. Meaning we only need to exactly integrate polynomials of to the second degree. For tetrahedons, in natural coordinates, quadrature sets are available in [7]. For this study the weights and quadrature points as shown in Table below will be used.

Point	weights	X	Y	Z
0	0.25	0.585410197	0.138196601	0.138196601
1	0.25	0.138196601	0.138196601	0.138196601
2	0.25	0.138196601	0.138196601	0.585410197
3	0.25	0.138196601	0.585410197	0.138196601

Table 1: Quadrature points and weights used for tetrahedron elements.

References

- [1] Barrera-Figueroa V., et al. *Multiple root finder algorithm for Legendre and Chebyshev polynomials via Newton's method*, Annales Mathematicae et Informaticae, volume 33, pages 3-13, 2006
- [2] Lewis E.E, Miller W.F. *Computational Methods of Neutron Transport*, John Wiley & Sons, 1984, ISBN 0-471-09245-2
- [3] Duderstadt J.J., Hamilton L.J., *Nuclear Reactor Analysis*, John Wiley & Sons, 1976.
- [4] Bailey T.S., Chang J.H., Adams M.L., *A Piecewise Linear Discontinuous Finite Element spatial discretization of the transport equation in 2D Cylindrical Geometry*, 2009 International Conference on Advances in Mathematics, Computational Methods, and Reactor Physics, 2008.
- [5] Bailey T.S., Warsa J.S., Chang J.H., Adams M.L., *A Piecewise Bi-linear Discontinuous Finite Element spatial discretization of the S_n transport equation*, International Conference on Mathematics and Computational Methods Applied to Nuclear Science and Engineering, 2011.
- [6] Pierce, Rod. *Inverse of a Matrix using Minors, Cofactors and Adjugate* Math Is Fun. Ed. Rod Pierce. 22 Nov 2018. 1 Jan 2019 <http://www.mathsisfun.com/algebra/matrix-inverse-minors-cofactors-adjugate.html>
- [7] Engels H., Zienkiewicz O., *Quadrature Rules for Tetrahedrons*, http://people.sc.fsu.edu/~jburkardt/datasets/quadrature_rules_tet/quadrature_rules_tet.html, accessed January 1, 2019.
- [8] Bailey T.S., Adams M.L., Yang B., Zika M.R., *A piecewise linear finite element discretization of the diffusion equation for arbitrary polyhedral grids*, Journal of Computational Physics 227 (2008) 3738–3757, 2007
- [9] Cheng et al, *Delaunay Mesh Generation*, Chapman & Hall/CRC Computer & Information Science Series, 2013
- [10] Adams M.P., Hawkins W.D., Adams M.L., *Managing Information Flow in Graph Traversals*, Texas A&M University, November 2018