# MEAN-DG

**Mixed Elements Applications with Nodal Discontinuous Galerkin methods**

## Lab Manual

by

**Subodh Joshi, Tapan Mankodi, Prof. S. Gopalakrishnan**


Indian Institute of Technology, Bombay
Mumbai
2018

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 What

MEAN-DG (**M**ixed **E**lements **A**pplications with **N**odal **D**iscontinuous **G**alerkin method) aims to be a versatile, industry-applications oriented efficient Discontinuous Galerkin code. However, all efforts have been taken to keep it as a flexible framework such that in addition to the Galerkin family of schemes, other schemes such as the FV scheme, spectral methods etc. can be written as required. The code in its finished form is expected to cater four types of 3D elements: hexahedral, tetrahedral, prism and pyramid cells. It can be customized for a wide array of equations ranging from as simple as the linear advection equation, Laplace equation to as complicated as the Navier Stokes (N-S) equations and Magnetohydrodynamics (MHD) equations. The code is mostly a c++ code with some utilities written in Python language. The data-structure and the file reading format is inspired from the OpenFOAM file format. Thus, test-cases and applications for OpenFOAM can be readily solved using MEAN-DG (depending on the solvers coded). After the current phase the code will have following features:

- Solve advection dominated flows efficiently
- Achieve arbitrary higher-order accuracy in space
- Open-MP parallelization
- Capability to handle flexible geometries and different mesh element types
- Capability to code other systems of equations

## 1.2 Why

Higher-order accuracy in space is extremely important for many applications such as boundary layer flows, vortex dominated flows, linear propagating waves (such as in electromagnetics and aeroacoustics) etc. Finite volume (FV) methods are well established and mature enough to cater to the industry demands. However, FV methods suffer from certain limitations, such as:

- Poor hardware scalability and parallelization capacity (because of wider stencils used and the inter-cell communications it demands)

- Limitations on order of accuracy because of non-compact nature of the reconstruction
- Difficult boundary treatments

On the other hand the spectral methods and global collocation methods are extremely accurate but lack flexibility and simplicity of the finite volume methods. The DG methods fall right in the sweet spot between the element based Galerkin methods and the finite volume methods. As of now, not many large-scale codes are available based on the DG framework. A few codes are available such as,

1. Deal.II (http://www.dealii.org/)
2. Dune (https://www.dune-project.org/about/dune/)
3. Fenix (https://fenicsproject.org/documentation/)

The idea behind MEAN-DG is to have a DG code similar to the OpenFOAM FV code for industry level applications. It also serves the purpose of capacity building, indigenous technology building and have an in-house computation-platform.

## 1.3 Who

The PI of the project is Prof. S. Gopalakrishnan. Primary contributions to the project are made by Tapan Mankodi and Subodh Joshi. The project is funded by Siemens Technologies, Bangalore.

## 1.4 How to refer this manual

Chapter 2 covers the necessary mathematical background to get started with the code. Chapter 3 explains the structure of the code. Chapter 4 explains the work-flow for a few applications solved with MEAN-DG. Chapter 5 consists of the license. User is strongly recommended to go through it before using the code.

# Chapter 2

# Mathematical Background

## 2.1 Introduction

Discontinuous Galerkin finite element method is an amalgamation of the element based Galerkin schemes and the finite volume method. It is a compact, higher-order accurate scheme where the solution can be discontinuous between two elements. The DG scheme requires the physical domain ($\Omega$) discretized with $Ne$ finite elements (written as $\Omega^k$). The finite element cells are assumed to be one of the following types.

1. Hexahedral (8 edges, 8 nodes, 6 faces)
2. Tetrahedral (6 edges, 4 nodes, 4 faces)
3. Prism (9 edges, 6 nodes, 5 faces)
4. Pyramid (8 edges, 5 nodes, 5 faces)

The solution is assumed to be continuous (along with continuous derivatives depending on the governing equations) inside the element. However, the solution can be discontinuous at the edges and faces. The value of the conserved variable (and the numerical fluxes) at the faces and edges are found out by solving the Riemann problem which arises because of the discontinuity.

The DG method essentially starts with the 'variational' formulation of the problem. Consider a differential equation given by,

$$\mathbf{L}(u) = f(u), u \in H^1(\Omega) \tag{2.1}$$

(and suitable boundary conditions, not mentioned here). For example, $\mathbf{L} = \frac{d}{dx}$ and $f(u) = u$ results in,

$$\frac{du}{dx} = u \tag{2.2}$$

We will be using the general notation given by equation 2.1 henceforth. Then, if $u^k$ is the approximate solution inside of the element, then instead of the solving equation 2.1 with $u^k$, we solve the following variational formulation of equation 2.1.

$$\int_{\Omega_k} \phi \mathbf{L}(u^k) d\Omega = \int_{\Omega_k} \phi f(u^k) d\Omega \tag{2.3}$$

$\phi$ is the test function. The approximate solution $u^k$ is obtained by polynomial approximation of $u$ inside $\Omega^k$. A modal interpolation (basis functions consisting of modes, i.e. frequencies) or a nodal interpolation (basis functions consist of the cardinal polynomials such as the Lagrange polynomials) can be used for this. In the DG methods, the basis functions serve as the test functions. The integrations in equation 2.2 are solved numerically using quadrature formulas. The resulting algebraic system is solved iteratively (or ODE is solved numerically if time dependant). This process is repeated till convergence (for steady state problems) or till the final time is reached (for transient problems). It is easier to solve the problem on a reference cell (canonical element) and then map the solution on to the physical cell. This requires a mapping to be defined between the reference cell and the physical cell. Based on this general process, the following mathematical concepts are revised in this chapter.

1. Mapping functions (Jacobian ($\mathbf{J}$) of transformation, inverse Jacobian ($\mathbf{J}^{-1}$) and ($|\mathbf{J}|$))
2. Modal and nodal basis functions
3. Polynomial interpolation and differentiation
4. Numerical quadrature (integration)
5. Cell matrices (Vandermonde matrix, Mass matrix, differentiation matrix and flux matrix)
6. Other mathematical functions

The discussion is kept brief and concise and only those concepts concerned with MEAN-DG are discussed. The user is encouraged to go through the references for in-depth theory of DG schemes.

**Note 2.1.** *We use the c++ style indexing. i.e. indices start at* 0 *(not* 1*).*

## 2.2 Mapping on the Reference Element

Corresponding to each physical cell, there exists a canonical (reference) element. The physical cell is expressed in terms of the physical coordinate system $(x, y, z)$. The reference coordinate system is written in terms of $(r, s, t)$. Thus, each of the physical coordinates can be written in terms of the $(r, s, t)$ coordinate system. In the code, we describe the physical coordinate system as $XYZ$, and the reference system as $RST$. The mapping from the physical coordinate system to the reference coordinate system is characterized by the Jacobian of transformation $\mathbf{J}$. In this section, we describe the Jacobian of transformation for the four types of the cells.

### 2.2.1 Hexahedral cells

Hexahedral, or Hex in short, cells are mapped to a cube in 3D. The Hex cell has 8 nodes given by the array, $((x_0, y_0, z_0), (x_1, y_1, z_1), ..., (x_7, y_7, z_7))$. We define the linear shape functions on the reference element as follows:

$$N_0 = \frac{1}{8}(1-r)(1-s)(1-t), \quad \{r, s, t \in V | \forall v \in V \subset \mathbb{R},\ v \in [-1, 1]\} \tag{2.4}$$

$$N_1 = \frac{1}{8}(1+r)(1-s)(1-t) \tag{2.5}$$

$$N_2 = \frac{1}{8}(1+r)(1+s)(1-t) \tag{2.6}$$

$$N_3 = \frac{1}{8}(1-r)(1+s)(1-t) \tag{2.7}$$

$$N_4 = \frac{1}{8}(1-r)(1-s)(1+t) \tag{2.8}$$

$$N_5 = \frac{1}{8}(1+r)(1-s)(1+t) \tag{2.9}$$

$$N_6 = \frac{1}{8}(1+r)(1+s)(1+t) \tag{2.10}$$

$$N_7 = \frac{1}{8}(1-r)(1+s)(1+t) \tag{2.11}$$

The mapping then is simply given as,

$$x(r,s,t) = \sum_{i=0}^{7} N_i x_i \tag{2.12}$$

$$y(r,s,t) = \sum_{0}^{7} N_i y_i \tag{2.13}$$

$$z(r,s,t) = \sum_{0}^{7} N_i z_i \tag{2.14}$$

**Definition 1.** *The Jacobian of transformation (***J***) is defined as :*

$$\mathbf{J} = \begin{bmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial s} & \frac{\partial x}{\partial t} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial s} & \frac{\partial y}{\partial t} \\ \frac{\partial z}{\partial r} & \frac{\partial z}{\partial s} & \frac{\partial z}{\partial t} \end{bmatrix} = \begin{bmatrix} J_{00} & J_{01} & J_{02} \\ J_{10} & J_{11} & J_{12} \\ J_{20} & J_{21} & J_{22} \end{bmatrix} \tag{2.15}$$

The partial derivatives in equation 2.15 can be easily computed from equations 2.4-2.11 and 2.12, 2.13, 2.14. In the code, the above implementation is found in Cell.cpp.

`Cell::calculateJacobian3DTensor(double r,double s,double t)`

**Note 2.2.** *The first name indicates the name of the class (Cell), the second symbol (::) indicates the scope (in this case, indicating 'belongs to Cell' the third name is the name of the class method (calculateJacobian3DTensor) and terms in the bracket being arguments passed to this function.*

It is to be noted that the name 'Tensor' in the above function indicates that the functional space for a Hex cell is obtained by a tensor product (outer product) of 1D functional spaces.

The inverse of the Jacobian is found out by the standard process of inversion of a matrix.

$$\mathbf{J^{-1}} = \begin{bmatrix} \frac{\partial r}{\partial x} & \frac{\partial r}{\partial y} & \frac{\partial r}{\partial z} \\ \frac{\partial s}{\partial x} & \frac{\partial s}{\partial y} & \frac{\partial s}{\partial z} \\ \frac{\partial t}{\partial x} & \frac{\partial t}{\partial y} & \frac{\partial t}{\partial z} \end{bmatrix} \tag{2.16}$$

These are given as:

$$\frac{\partial r}{\partial x} = \frac{1}{|\mathbf{J}|}\left(J_{22}J_{11} - J_{21}J_{12}\right) \tag{2.17}$$

$$\frac{\partial r}{\partial y} = \frac{-1}{|\mathbf{J}|}\left(J_{22}J_{01} - J_{21}J_{02}\right) \tag{2.18}$$

$$\frac{\partial r}{\partial z} = \frac{1}{|\mathbf{J}|}\left(J_{12}J_{01} - J_{11}J_{02}\right) \tag{2.19}$$

$$\frac{\partial s}{\partial x} = \frac{-1}{|\mathbf{J}|}\left(J_{22}J_{10} - J_{20}J_{12}\right) \tag{2.20}$$

$$\frac{\partial s}{\partial y} = \frac{1}{|\mathbf{J}|}\left(J_{22}J_{00} - J_{20}J_{02}\right) \tag{2.21}$$

$$\frac{\partial s}{\partial z} = \frac{-1}{|\mathbf{J}|}\left(J_{12}J_{00} - J_{10}J_{02}\right) \tag{2.22}$$

$$\frac{\partial t}{\partial x} = \frac{1}{|\mathbf{J}|}\left(J_{21}J_{10} - J_{20}J_{11}\right) \tag{2.23}$$

$$\frac{\partial t}{\partial y} = \frac{-1}{|\mathbf{J}|}\left(J_{21}J_{00} - J_{20}J_{01}\right) \tag{2.24}$$

$$\frac{\partial t}{\partial z} = \frac{1}{|\mathbf{J}|}\left(J_{11}J_{00} - J_{10}J_{01}\right) \tag{2.25}$$

where, the determinant of the Jacobian $|\mathbf{J}|$ is calculated as:

$$|\mathbf{J}| = J_{00}(J_{22}J_{11} - J_{21}J_{12}) - J_{01}(J_{22}J_{10} - J_{20}J_{12}) + J_{02}(J_{21}J_{10} - J_{20}J_{11}) \tag{2.26}$$

The inverse of Jacobian is implemented in

`Cell::calculateInverseJacobianMatrix3DTensor(arguments)`

**Note 2.3.** *The inverse of Jacobian is called as* **dRST_by_dXYZ** *in the code. Each cell stores these matrices for all the quadrature points within that cell. The metric terms given by 2.17–2.25 are used in computations involving derivatives as seen in the subsequent sections and chapters.*

The inverse Jacobian is stored for all the quadrature points for all the cells. It is an attribute of the cell (i.e. Cell::dRST_by_dXYZ) and has dimensions $nQuad \times 3 \times 3$.

**Use of the Jacobian** Consider the following integration:

$$I = \int_{\Omega^k} \phi(x, y, z) d\Omega^k \tag{2.27}$$

The integration can be written as,

$$I = \int_{\Omega^s} \phi(r, s, t)|\mathbf{J}|d\Omega^s \tag{2.28}$$

where, $\Omega^s$ is a reference element and $\mathbf{J}$ is the Jacobian matrix. The main benefit of performing the above conversion is that, the function $\phi(r, s, t)$ needs to be computed only once for the standard element, each cell will have its own Jacobian matrix, so number of computations reduce drastically.

In the current code, the vertices and faces are numbered in a standardized fashion as shown in Fig.2.1.

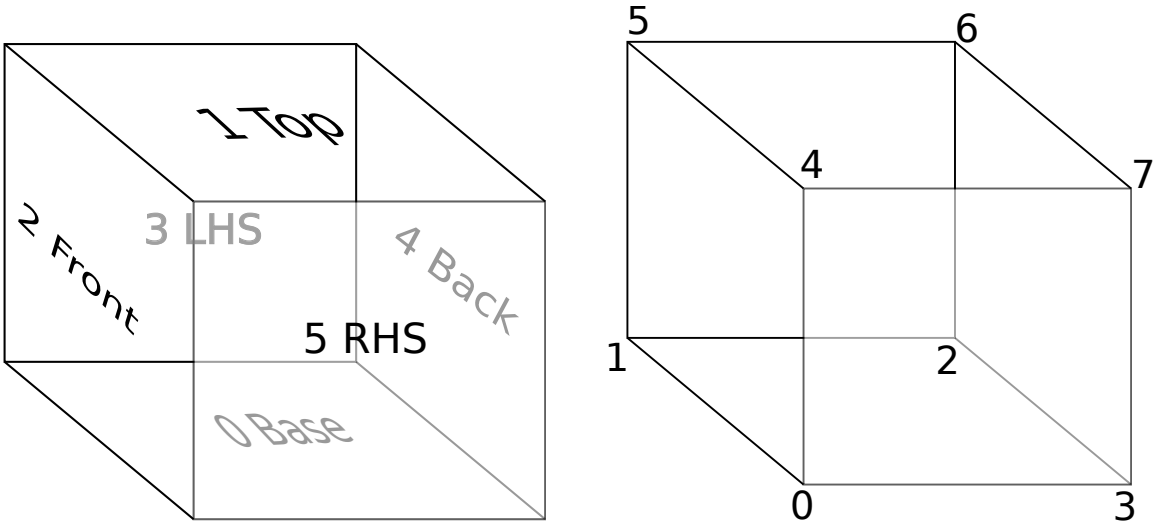


Figure 2.1: a. Reference cell and faces relative numbering b. Reference cell and vertices numbering

The corresponding implementation can be found in:

`Cell::orderDefiningPoints()`

## 2.3 Mapping onto the Reference Face

DG method involves evaluation of surface integrations in order to find the numerical flux at the cell-interface. The surface integrations in turn require mapping of a given arbitrary surface on the reference surface. The parameterization of a plane surface requires two parameters $r$ and $s$.

### 2.3.1 Quadrilateral Faces

For a quadrilateral faces, the parameters assume the following values: $(r, s) \in [-1, 1]$. However, the given surface can be arbitrarily oriented in the space, and thus requires $(x, y, z)$ position to be completely defined. We thus need to orient the surface such as one of the variables becomes redundant. This means rotating the surface such as to make it parallel to $xy$ plane, for example. Thus, each surface needs to be re-oriented in space such that the face normal $\hat{n}$ becomes parallel to the $z$ axis. After the we can map the rotated face onto the standard element. Let $\theta_z$ be the angle made by the face normal with the $z$

axis and $\theta_x$ be the angle made by the face normal with the $x$ axis. The rotation matrix rotates the face through $\theta_x$ first (counterclockwise) and the through $\theta_z$ counterclockwise. The resulting rotation matrix $R$ is given as:

$$R =< R_z, R_x > \tag{2.29}$$

where,

$$R_x = \begin{bmatrix} cos(\theta_x) & sin(\theta_x) & 0 \\ -sin(\theta_x) & cos(\theta_x) & 0 \\ 0 & 0 & 1 \end{bmatrix}; \quad R_z = \begin{bmatrix} cos(\theta_z) & 0 & -sin(\theta_z) \\ 0 & 1 & 0 \\ sin(\theta_z) & 0 & cos(\theta_z) \end{bmatrix}; \tag{2.30}$$

and $< a, b >$ indicates the dot product between matrices $a$ and $b$. Matrix $R^{-1}$ indicates the reverse transformation. The rotation matrix is calculated in,

```
Face::calculateRotationMatrixParallelToXY()
```

and the inverse matrix is computed in,

```
Face::calculateInverseRotationMatrix()
```

In order to describe the procedure to compute the Jacobian of transformation ($|\mathbf{J}^f|$) for face, we use the following nomencleture:

- $(x, y, z)$: physical coordinate system (actual coordinates)
- $(x', y', z')$: Rotated coordinates such that the face is $||$ to the $xy$ plane
- $(r, s)$: Reference coordinates such that $r, s \in [-1, 1]$

There are two ways the Jacobian can be computed.

### Method 1

First, rotate each vertex of the face using $R$. i.e.

$$\begin{bmatrix} x_i' \\ y_i' \\ z_i' \end{bmatrix} = R \left( \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} \right), \quad i \in \{0, 1, 2, 3\} \tag{2.31}$$

Each of the rotated points is then mapped to the standard element. After rotation of the face, $\frac{\partial z'}{\partial r} = 0$, $\frac{\partial z'}{\partial s} = 0$ since $z'$ remains constant. The mapping from the rotated face $f'$ to the standard face $f^s$ can be defined through the shape functions.

$$N_0 = \frac{1}{4}(1 - r)(1 - s), \quad \{r, s \in V | \forall v \in V \subset \mathbb{R},\ v \in [-1, 1]\} \tag{2.32}$$

$$N_1 = \frac{1}{4}(1 + r)(1 - s) \tag{2.33}$$

$$N_2 = \frac{1}{4}(1+r)(1+s) \tag{2.34}$$

$$N_3 = \frac{1}{4}(1-r)(1+s) \tag{2.35}$$

Then,

$$x' = \sum_{i=0}^{3} N_i x'_i, \quad y' = \sum_{i=0}^{3} N_i y'_i, \tag{2.36}$$

Equation2.36 can be used to find partial derivatives of $x'$ and $y'$ with respect to $r$ and $s$. The required face Jacobian is:

$$\mathbf{J}^f = \begin{bmatrix} \frac{\partial x}{\partial r} & \frac{\partial x}{\partial s} \\ \frac{\partial y}{\partial r} & \frac{\partial y}{\partial s} \\ \frac{\partial z}{\partial r} & \frac{\partial z}{\partial s} \end{bmatrix} \tag{2.37}$$

which can be written as,

$$\mathbf{J}^f = \begin{bmatrix} x \\ y \\ z \end{bmatrix} . \begin{bmatrix} \frac{\partial}{\partial r} & \frac{\partial}{\partial s} \end{bmatrix} = R^{-1} \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} . \begin{bmatrix} \frac{\partial}{\partial r} & \frac{\partial}{\partial s} \end{bmatrix} \tag{2.38}$$

i.e.

$$\mathbf{J}^f = R^{-1}\mathbf{J}^{f'} \tag{2.39}$$

where,

$$\mathbf{J}^{f'} = \begin{bmatrix} \frac{\partial x'}{\partial r} & \frac{\partial x'}{\partial s} \\ \frac{\partial y'}{\partial r} & \frac{\partial y'}{\partial s} \\ 0 & 0 \end{bmatrix} \tag{2.40}$$

The procedure is summarized as,

1. Rotate the face so as to make it parallel to the $xy$ plane.
2. Find the Jacobian of transformation from the rotated face to the standard face ($\mathbf{J}^{f'}$).
3. Find the Jacobian of the face as given by equation 2.39.
4. $|\mathbf{J}^f|$ is given as $\sqrt{|\left(\mathbf{J}^f\right)^T\left(\mathbf{J}^f\right)|}$

**Method 2**

In this method, we parametarize the surface $x, y, z$ with parameters $r, s$. Let

$$X = x\hat{i} + y\hat{j} + z\hat{k}$$

We use the standard shape functions to map the physical element to parameters $r, s$. i.e.

$$x = \sum_{i=0}^{3} N_i x_i, \quad y = \sum_{i=0}^{3} N_i y_i, \quad z = \sum_{i=0}^{3} N_i z_i, \tag{2.41}$$

where same shape functions are used as earlier.

**Note 2.4.** *In order to find the vertex mapping of a randomly oriented face with the standard element, it would be advisable to first rotate the face to make it parallel to the xy plane and then find the vertex mapping. Then find the shape function and proceed with equation 2.41. However, we do not use the coordinates of the rotated face here.*

After that, we find $\frac{\partial X}{\partial r}$ and $\frac{\partial X}{\partial s}$. Then $|\mathbf{J}^f|$ is simply $\left|\left|\frac{\partial X}{\partial r} \times \frac{\partial X}{\partial s}\right|\right|$ Both of these methods are implemented in

`Face::calculateJacobianQuadFace(double r, double s)`

Any given quadrature point $(r, s)$ can be mapped into $(x, y, z)$ space using the two operations:

1. First find mapping from $(r, s)$ to $(x', y', z')$ using equations 2.32-2.35 and 2.36.
2. Then map from $(x', y', z')$ to physical coordinates using $R^{-1}$.

The global positioning of all the quadrature points on each face is stored in array

`Face::quadPointsGlobalLocation`

Index $i$ runs over the surface quadrature points and index $j$ runs over the $x, y, z$ axes.

**Note 2.5.** *Some of the volume quadrature points coincide with the surface quadrature points. Function*

`DG::mapFaceQuadPointsToCellQuadPoints()`

*performs the mapping by direct comparison of global positioning of surface and volume quadrature points.*

The mapping of surface to volume quadrature points is stored in arrays Face::mapOwnerQuadPoints[ ] and Face::mapNeighbourQuadPoints[ ].

Thus, if we are finding a $2^{nd}$-order polynomial for a face, then it should have 9 quadrature points (QP). The corresponding cell will meanwhile have 27 quadrature points, 9 of which will coincide with the face QP. Then if

```
Face::mapOwnerQuadPoints = [3, 2, 21, 14, ... , 7]
```

Then, it means the $0^{th}$ quad point on face is same as the $3^{rd}$ quad point of owner cell and so on.

**Note 2.6.** *In the improved version of the code, the face-vertex numbering is standardized, such that the vertex numbering follows the right-hand rule. i.e. if the curled fingers indicate the sequence of vertices, then the stretched thumb points in the direction of hte face-normal. In that case, the rotation matrices are not required. i.e. both method 1 and method 2 are obsolete. Refer:*

`Face::orderDefiningPoints()`

## 2.4 Modal and Nodal Basis Functions

Basis functions are the independant dimensions on which any function can be projected. In this chapter, we present a few modal and nodal basis functions.

### 2.4.1 Modal Basis Functions

These are the set of basis functions, where each dimension corresponds to a 'mode' or frequency. A simple example is a set of monomials defined on the interval $[a, b]$

**Note 2.7.** *This is just an example, in reality $x$ could be valid over $(-\infty, \infty)$*

$$V^K_{mono} = \{1, x, x^2, x^3, ..., x^{K-1}\}, \quad x \in C^0[a, b] \tag{2.42}$$

A polynomial up to order $K-1$ can be represented exactly using $K$ modal functions. Thus, any general polynomial can be written as

$$P^K(x) = \sum_{i=0}^{i=K-1} \alpha_i x^i, , \quad x \in C^0[a, b] \tag{2.43}$$

The monomials is not a 'good' choice of the basis functions as we will see later, thus we need a set of basis functions where each of the functions is 'orthogonal' (or better 'orthonormal') to every other functions. One such important set of orthogonal functions is called as 'Jacobi' polynomials. A special case of 'Jacobi' polynomials is 'Legendre' polynomials. In our code, we have extensively used Legendre polynomials. A general $i^{th}$ modal basis function is indicated by symbol $\psi_i$. Other popular example of modal basis functions is a set of Chebyshev polynomials.

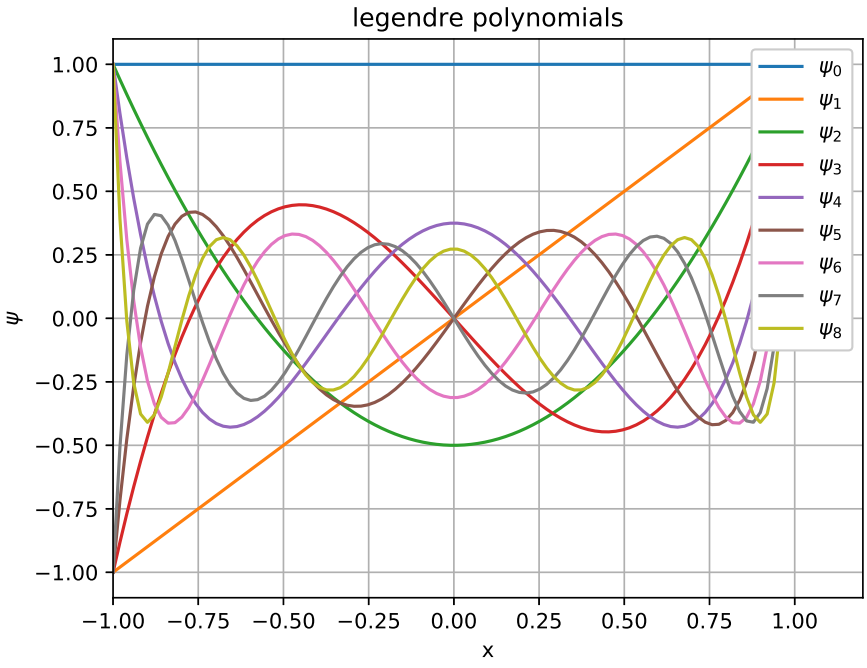


Figure 2.2: First 9 Legendre polynomials

Fig.2.2 shows first 9 Legendre polynomials. Observe that the 'frequency' of the polynomials increases (for increasing count) indicating higher modes. Fig.2.3 shows Chebyshev polynomials.

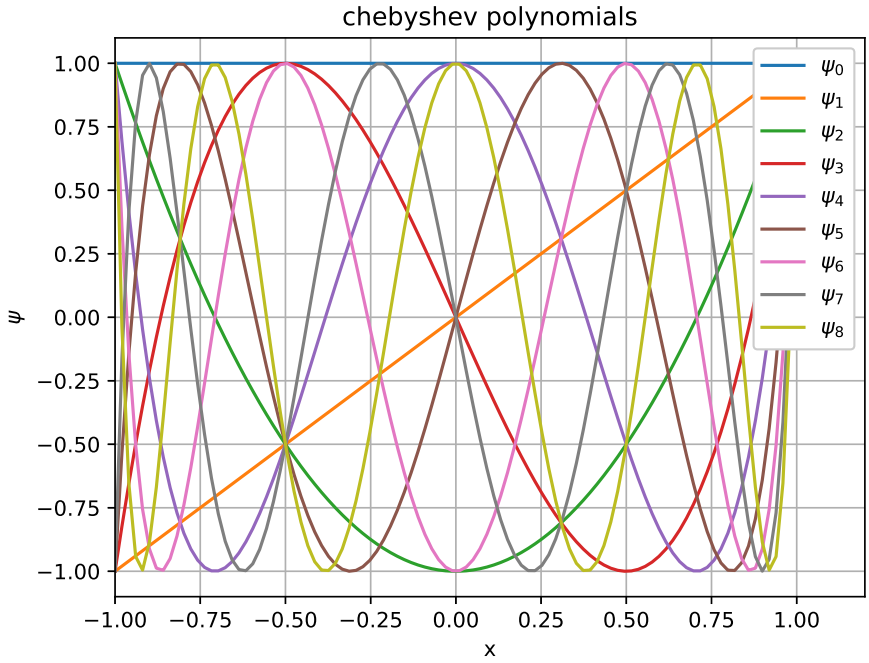


Figure 2.3: First 9 Chebyshev polynomials

The 1D modal functions are implemented in:

`FunctionalSpace::monomials1D(args)`,
`FunctionalSpace::legendrePolynomial1D(args)`,
`FunctionalSpace::chebyshevPolynomial1D(args)` and
`FunctionalSpace::jacobiPolynomial1D(args)`

It is indeed difficult to comprehend how the polynomials shown in Fig.2.2 and Fig.2.3 be 'orthogonal'. The main reason for this is that we try to stick to our understanding of the 3D space and the notion of orthogonality that comes with it, however, the polynomial space is essentially $\infty$ dimensional space. For this space, the definition of orthogonality depends on the definition of the Inner-Product and 'metric' of that space. For $L^2$ space, the orthogonality is defined as,

$$\int_\Omega \psi_i \psi_j d\Omega = \alpha \delta_{ij}, \quad \forall \psi_i, \psi_j \in L^2\{\Omega\} \;\&\; \alpha \in \mathbb{R} \tag{2.44}$$

where, $\delta_{ij}$ is a 'Kronecker Delta' which means $\delta_{ij} = 1$ if $i = j$ and 0 otherwise. The functions $\psi_i$ and $\psi_j$ are orthonormal if $\alpha = 1$.

### 2.4.2 Nodal Basis Functions

The 'nodal' basis functions are the set of functions which are 'Cardinal' in nature. What that means is, at the interpolation point, the basis function will have value 1 and for all

other collocation points, the value of the basis function will be zero. Thus, the interpolated value of the function agrees perfectly at the collocation points but may have some error at other places. The general symbol used for nodal basis functions is $\phi$. The interpolation itself is given as,

$$f^N(x) = \sum_{i=0}^{N-1} f(x_i)\phi_i(x), \quad x \in C^0[a,b] \tag{2.45}$$

and, $\phi_i(x_j) = \delta_{ij}$. We use Lagrange polynomials as the nodal basis functions. The polynomials in 1D are give as,

$$\phi_i(x) = \prod_{j=0, j\neq i}^{N-1} \frac{(x - x_j)}{(x_i - x_j)} \tag{2.46}$$

it is clear that at $x = x_i$, $\phi_i(x_i) = 1$. One crucial point that needs to be taken care of is selection of the collocation points $[x_0, x_1, ..., x_{N-1}]$. It is seen that the equi-spaced arrangement of collocation points gives poor interpolation for very high orders of accuracies. Therefore we take these points as roots of orthonormal modal polynomials. Popular choices for collocation points are Gauss-Legendre (LG) collocation points, Legendre-Gauss-Lobatto (LGL) points or Chebyshev points. An important distinction between LG and LGL points is that, LGL points include the 'end-points' whereas LG points include only the internal points. For DG methods, this doesn't have a significant difference since solving a Riemann problem at the face remains unchanged.

### 2.4.3 Multidimensional Interpolation

For number of dimensions greater than 1, the interpolating polynomials are obtained depending on the type of the element we wish to interpolate on. For hex elements, these are obtained using tensor product of 1D polynomials. i.e.

$$\phi(x, y, z) = \phi_{1D}(x)\phi_{1D}(y)\phi_{1D}(z) \tag{2.47}$$

Whereas, for other types of elements, special multidimensional polynomials are used. However, the concept of modal and nodal basis functions remains essentially the same irrespective of the number of dimensions.

# Chapter 3

# Code Structure

Roughly the code structure is displayed in Fig.3.1. For more accurate and detailed description, run Doxygen utility and refer the generated documentation.

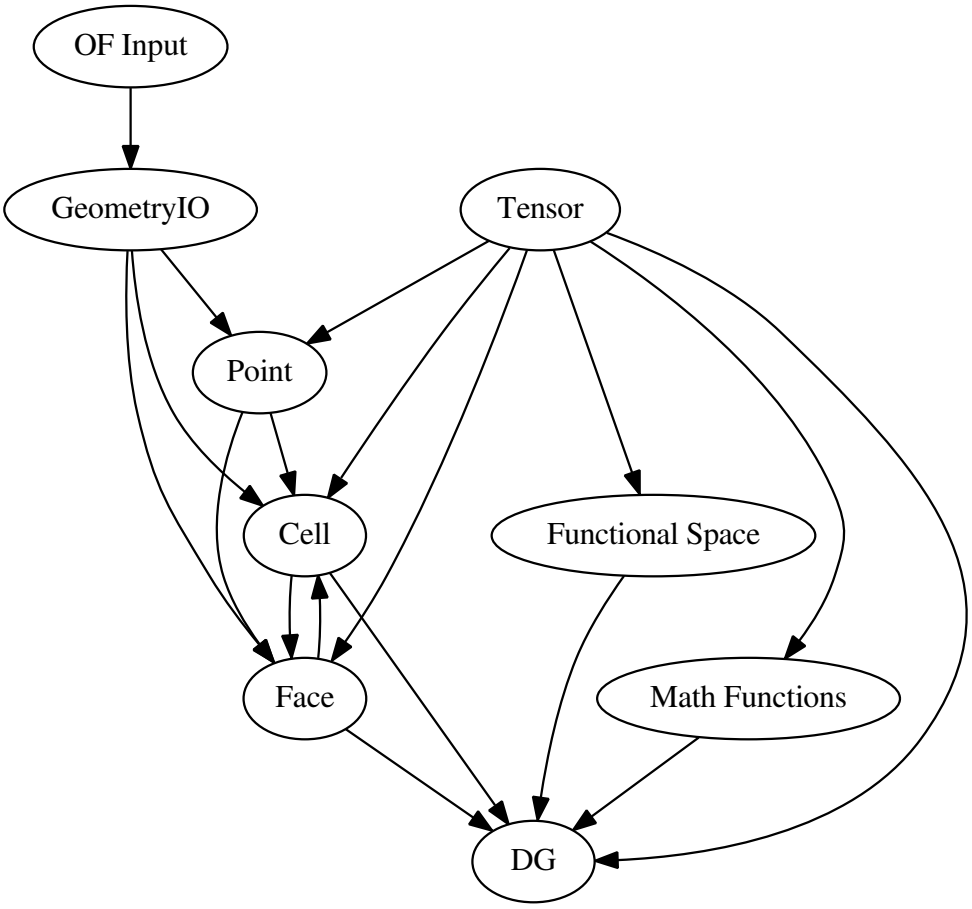


Figure 3.1: Class dependency (crudely)

Each of the the constituent elements are briefly described next.

## 3.1 Code structure

### 3.1.1 Tensor

MEANDG has a central datastructure called as 'Tensor', which forms all the important arrays, matrices, multidimensional storage lists etc. Refer 'include/Tensor' for details. Tensor class is made of 4 sub-classes.

```
Tensor01<dType>, Tensor02<dType>, Tensor03<dType>, Matrix<dType>
```

From these, $O(1), O(2), O(3)$ tensors can be created of arbitrary dimensions. The tensor objects interact with one another and have several in-build methods such as $L_1, L_2, L_\infty$ norms etc. All the maths functions accept objects of Tensor library.

Following example code shows usage of tensor class:

```
Tensor01<double> A(5); // creates a O(1) tensor of size 5 with
// data-members of the type double
Tensor01<double> C(5);

for (int i=0; i<5; i++){
// generate a random number between 0 and 100
double valueA = getRandom(0.0,100.0);
// generate another random number between 0 and 100
double valueC = getRandom(0.0,100.0);

// set value at ith index
A.setValue(i, valueA);
C.setValue(i, valueC);
};
// now A and C have random values

double dotAnswer = Math::dot(A,C); // perform math product between two arrays
cout << dotAnswer << endl; // print the answer
```

For further details and the available functions, refer the code documentation.

### 3.1.2 Point

Point is a physical point in $(x, y, z)$ coordiante system. It has a unique id, and description of its location.

### 3.1.3 Cell

If the domain is indicated by $\Omega$, then the descritization of the domain $(\Omega_h)$ is described as,

$$\Omega_h = \bigcup_{l=1}^{n} S_i \tag{3.1}$$

where $S_i$ are the finite volume cells. These are non-overlapping pieces of the domain. Each cell has the following important attributes (among others):

- Vertices (of the type 'Point')
- Faces (i.e. boundary of the cell)
- Degrees of Freedom (i.e. locations where unkowns are evaluated)
- Data (such as the conserved variables)
- Volume
- Mathematical constructs (Mass matrix, Stiffness matrix etc.)
- Mapping onto the standart cell (Jacobian etc.)

Thus, each of these parameters (among others) can be accessed using the accessor (get, set) functions of the class 'Cell'. For further details, refer Cell.h and Cell.cpp.

### 3.1.4 Face

A face is the boundary of a Cell. The face has following important attributes:

- Geometrical information (area, normal, vertices etc.)
- Neighbour and Owner cells
- Data (numerical flux and variables)
- Mapping (Jacobian etc.)

### 3.1.5 GeometryIO

This module reads the input file in the OpenFOAM format and fills the data arrays. Refer:

`GeometryIO::fillDataArrays(args)`

### 3.1.6 Functional Space

This module is responsible for everything related to $L^2, H^1$ functional spaces. It contains nodal, modal basis functions and derivatives. It also computes various matrices used in DG formulation.

### 3.1.7 Math functions

These are various mathematical functions.

### 3.1.8 Gasdynamics, Riemann solvers etc.

Depending on the system of equations which one wants to solve, include the physics-specific functions and files. For example, for Euler's equations of Gasdynamics, we have Gasdyanmics.cpp module and corresponding Riemann solvers. Apart from these, there are test functions, display functions etc. Refer the code files in 'include/' and 'src/' folders.

### 3.1.9 DG

This is the core (central) module of the code. It performs following operations:

- Memory management
- Cells, faces creation
- Calling appropriate functions for computation of cell matrices
- Time integration
- PDE solving

Currently it is configured to solve the hyperbolic system of PDEs of the type:

$$\frac{\partial Q}{\partial t} + \nabla.\mathbf{F}(Q) = 0 \tag{3.2}$$

# Chapter 4

# Applications

# Chapter 5

# License

MEANDG is licensed under BSD-3 clause license. Refer:

`https://en.wikipedia.org/wiki/BSD_licenses`

The library Eigen which is used in this software (see ./include/Eigen/) is licensed under MPL2 (see ./include/Eigen/COPYING.README).
Use preprocessor flag -DEIGEN_MPL2_ONLY to ensure that only MPL2 (and possibly BSD) licensed code from Eigen is compiled. Refer

`http://eigen.tuxfamily.org/index.php?title=Main\_Page#License`

for details. No files of the Eigen library are modified for use with MEANDG.

## 5.1 LICENSE