Mesh

Nicholas K. Burgess

April 5, 2014

1 Introduction

For the purposes of this document a mesh denotes a collection of nodes and elements. nodes define points in physical space and elements describe how these nodes are connected. The FEM library only supports certain types elements i.e. 1-D lines, 2-D triangles, 2-D quadrilaterals, 3-D tetrahedra, 3-D prisms, and 3-D Hexahedra. First we describe how elements are defined and how they are combined to form unstructured meshes. Then with the theory how meshes are defined in hand we will describe the data structures used in the code to define them and how to acess the data of the mesh using the implemented class functions of **UnstGrid**.

Note that mesh generation is a topic unto itself and the goal of this section is to describe how the numerical integration library interacts with a pre-existing mesh. A pre-existing mesh is one where the nodal coordinates in the physical space (i.e. the one in which we wish to solve the PDE's) represented notionally by the coordinates (x,y,z) are known. Additionally a mesh must supply how these nodes are connected into elements along with how the boundary faces of the domain are connected to the nodes that lie on the domain boundary. A few conditions on the mesh are that is must be completely fill the domain of interest Ω, mathematicaly this is

$$\Omega = \bigcup_{e} \Omega_e \tag{1}$$

2 Element Types

This section describes the types of elements that are supported by the solver library. In general elements define how nodes in the mesh are connected to each other. Elements are normally defineds in a know standard space defined by greek letters ξ, η, ζ. The positions of the nodes of the elements are known in this standard space and the physical locations of the nodes in (x,y,z) spaces are determined by a mapping function $f_m :^d \rightarrow^d$ where d is the number of physical dimensions of the domain. In this work the mapping functions f_m take on the form of polynomial interpolations, that are nodally exact at all nodes of the element.

2.1 1-D Bar

In one spatial dimension only one type of element is available. This element in known as the “bar” element. The “bar” element (shown in Figure 1) is constructed by taking two nodes of the mesh and connecting them by a line segment.

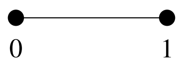

Figure 1: 1-D “bar” element.

Recalling that each element has a definition in the standard space of equivalent dimension the coordinates of the nodes are node$_1$: $\xi = -1$ and node$_2$: $\xi = 1$ in the standard space spanning $\xi \in (-1, 1)$.

A 1-D “bar” element may have any number of solution unknowns associated with it but it requires a minimum of two uknowns: one for each element of the element. For finite-element methods this results in a linear data representation and an asymptotically second-order accurate solution.

2.2 2-D Triangle

In two spatial dimensions the simplex element shape is the triangle. The triangle is defined by connecting 3 of the mesh nodes as in Figure 2. Triangles require a minimum of three nodes to define the element. For a finite-element discretization these three nodes can be used to for a linear representation of the data within the element resulting in an asymptotically second order accurate approximation.

2.3 3-D Tetrahedron

In three spatial dimensions the simplex element shape is the tetrahedron. The tetrahedron is defined by a minimum of 4 nodes which are numbered 0 through 3. These four nodes can be used for a linear representation of the data within an element.

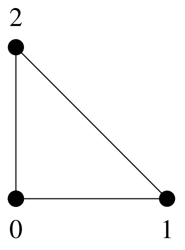

Figure 2: 2-D Triangle’ element.

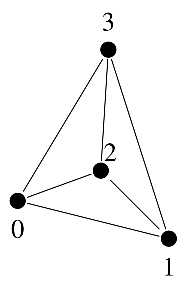

Figure 3: 3-D Tetrahadron.

3 Mesh Connectivity

The mesh connectivity data is stored in a linked list structure to facilitate optimal memory footprint. Each connectivity of the mesh is stored as a separate linked list. For example the nodes belonging to an element is one mesh connectivity and is stored in one linked list while the elements surrounding a node is stored in yet another linked list. Each linked list is comprised of two arrays an "index" array and a "data" array. The index array is normally named with an "i" at the end of the variable name and is composed on integer.

As an example consider the connectivity defining the node numbers of each element in a mesh. To make the explanation concrete consider the simple mesh in Figure 4. One should note that this 2-D example contains a mesh of mostly triangles with

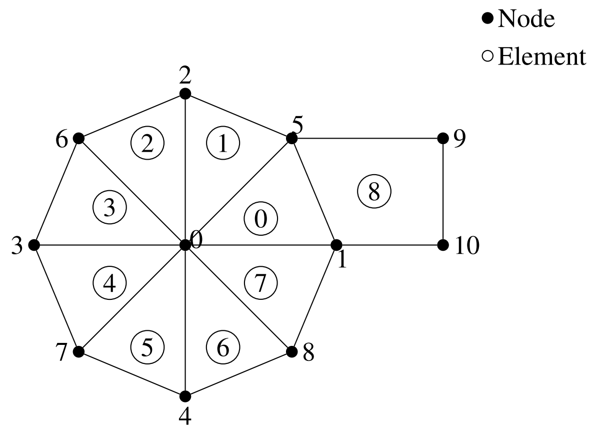

Figure 4: Sample Mesh.

one quadrilateral. The mixed element nature of the mesh prevents the mesh connectivity taking the form of a table (which can be stored in the computer a 2-D array) because some rows need more columns than others. For example all rows of the table corrsponding to triangular elements require 3 columns and rows corresponding to quadrilaterals require 4 columns. Clearly a more elegant data structure is required. In

	0	1	2	3	4	5	6	7	8	9
element2nodei:	0	3	6	9	12	15	18	21	24	28

	0-2	3-5	6-8	9-11	12-14	15-17	18-20	21-23	24-27
element2node:	0 1 5	0 5 2	0 2 6	0 6 3	0 3 7	0 7 4	0 4 8	0 8 1	1 10 9 5

this work a linked list is employed to facilitate the mesh connectivity storage.

Let *element2node* denote the data array that stores the nodes that make up each element and *element2nodei* denote the index array for *elem2node*. The size of *element2node* is the sum of the nodes attached to each element over all elements of the mesh and the size of *element2nodei* is the number of elements + 1. For an element *e* the nodes that makeup *e* are stored in the following locations

$$\begin{aligned} &is = elem2nodei[e] \\ &ie = elem2nodei[e+1]-1 \\ &elem2node[is:ie] : \text{is where the nodes for element } e \text{ are stored} \end{aligned} \tag{2}$$

In order to make this concrete consider the sample mesh in Figure 4. For this mesh the arrays *element2nodei* and *element2node* take the following values:

4 Auxiliary Connectivity

Solving partial differential equations (PDEs) numerically requires additional data structures beyond the nodes that make up an element. In general the edges and or faces of the mesh must be extracted. Additionally, one may need to form a list of nodes neighboring nodes or elements neighboring elements. In general these structures are not available from the mesh generator but must be extracted by an algorithm. Many of these data structures are formed as part of the mesh initialization process.

5 UnstGrid

UnstGrid is a c++ class implements the methods required to store and work with an unstructured mesh of arbitrary and mixed element types. The class assumes that the nodal coordinates and the mapping of which nodes make up an element will be provided in some form. The basic usage of UnstGrid is described in the following algorithm

Algorithm 1 :Using UnstGrid

```
{ First instantiate an object of type UnstGrid called grid. This will have integer data of type int and real number data of type double. }
$UnstGrid < int, double >$ grid(3) { Note that the class constructor is called with the number 3 indicating a 3-D grid }
Require: nnode, nelement, nbc_face, nbc_id;
for i = 0; i ¡ nnode; i++ do
    { Get values of x, y, z from wherever they are, these could be in a file or in some other data structure }
    grid.set_node_coord(i, 0, x);
    grid.set_node_coord(i, 1, y);
    grid.set_node_coord(i, 2, z);
end for
for e= 0; e ¡ nelement; e++ do
    { Get the nodes on the element and put them into the mesh data structure }
    for n = 0; n ¡ nnode_on_element; n++ do
        { Knowing the nodes on an element from your mesh source populate the grid like this }
        grid.set_node_on_element(e, n, node) { Where node is the node number you want to assign as the nth node of element e }
    end for
end for
for f= 0; f ¡ nbc_face; f++ do
    { Get the nodes on the element and put them into the mesh data structure }
    for n = 0; n ¡ nnode_on_bc_face; n++ do
        { Knowing the nodes on an element from your mesh source populate the grid like this }
        grid.set_node_on_bc_face(f, n, node) { Where node is the node number you want to assign as the nth node of boundary face f }
    end for
end for
{ Now use the init connectivity function to form all connectivity }
grid.init_connectivity(); { Grid is now prepared for use }
{ Mesh is ready for use }
```