FELICITY

Finite ELement Implementation and
Computational Interface Tool for You

Shawn W. Walker

August 4, 2014

Contents

1 Overview **1**
- 1.1 Installation . . . 1
- 1.2 Quick Start Guide . . . 2
- 1.3 General Philosophy . . . 2
- 1.4 Other FEM Packages . . . 3
- 1.5 To Do List . . . 3

2 Basic Mesh Manipulation **5**
- 2.1 Introduction . . . 5
- 2.2 Tutorial . . . 5
- 2.3 Details of The Mesh Class . . . 10
- 2.4 Local Mesh and Topological Entities . . . 14
- 2.5 Mesh Class Methods . . . 16

3 Defining Finite Elements **19**
- 3.1 Introduction . . . 19
- 3.2 Finite Element M-File Format . . . 19
- 3.3 Nodal Topological Arrangement . . . 22

4 Assembling Matrices **27**
- 4.1 Automatically Generating Code . . . 27
- 4.2 Example 1-D Problem . . . 27
- 4.3 Some Details of Transforming an Input .m File Into a MEX File . . . 35
- 4.4 How To Call The MEX File . . . 37
- 4.5 Advanced Features . . . 37
- 4.6 In-Depth Examples . . . 40
- 4.7 Conclusion . . . 42

5 Automatically Generating Degree-of-Freedom Maps **43**
- 5.1 Introduction . . . 43
- 5.2 Tutorial . . . 43
- 5.3 Further Notes . . . 44

6 Helper Routines For Finite Element Codes **47**
- 6.1 Introduction . . . 47

6.2 Read Out Reference Finite Element Information 48
6.3 Access to Finite Element Matrices 48
6.4 Manipulate Degrees-of-Freedom 49

7 Interpolating Finite Element Data **59**
7.1 Automatically Generating Code 59
7.2 Example 2-D Interpolation 59
7.3 Some Details of Transforming an Input .m File Into a MEX File 65
7.4 Other Things You Can Do 65

8 Point Searching On A Mesh **67**
8.1 Automatically Generating Code 67
8.2 Example Of Point Searching In 2-D 67
8.3 Some Details of Transforming an Input .m File Into a MEX File 71
8.4 Other Things You Can Do 72

9 Miscellaneous Tools **75**
9.1 Eikonal Equation Solver 75
9.2 Unstructured Mesh Generation: 2-D and 3-D 75
9.3 Hierarchical Search Trees 76

A Sparse Matrices **77**
A.1 Sparse Matrix Storage 77
A.2 C++ Class For Sparse Matrix Assembly 78

B Notes on Generating Code **79**
B.1 Computing The Local Finite Element Matrix 79
B.2 Local-to-Global Transformations 87

Chapter 1

Overview

1.1 Installation

Installing **FELICITY** is very easy and automatic.

1.1.1 Requirements

FELICITY depends rather heavily on the MATLAB Symbolic Math Toolbox and will **not** run without it. You also need a `C++` compiler that MATLAB can use with its `mex` command. The Microsoft Visual C++ Express Edition is free and works well (Note: you must also install Microsoft SDK; see MATLAB online help for MS Visual C++ installation directions). If you use `linux`, then `gcc` will fulfill your needs.

You can configure the `C++` compiler with MATLAB by simply typing

```
mex -setup
```

at the MATLAB prompt.

1.1.2 Unpack

Unzip the file `FELICITY.zip` to a directory on your computer. This will create a subdirectory called `.\FELICITY`.

1.1.3 Set Path

Open MATLAB, and change to the `.\FELICITY` directory created above. Execute the m-file script `FELICITY_paths` to add **FELICITY** to your MATLAB path.

1.1.4 Test It!

Now that **FELICITY** is in your MATLAB path, run the m-file script `test_FELICITY` to execute all unit tests. This will tell you if **FELICITY** is installed properly.

1.1.5 User Help

You can type `FELICITY_user_help` at the MATLAB prompt to get a listing of relevant classes and m-files in the **FELICITY** package.

1.2 Quick Start Guide

To learn **FELICITY**, the Google-Code page is the best place to start:

`http://code.google.com/p/felicity-finite-element-toolbox/`

Tutorials can be found on the wiki section of the Google-Code page.

NEW: there is also a quick reference guide (in PDF format) in the main directory of **FELICITY**.

1.3 General Philosophy

1.3.1 Why Another Finite Element Toolbox?

There are many finite element packages available, both commercial and free. **FELICITY** is designed for simulating problems where sub-domains interact in non-trivial ways, i.e. Partial Differential Equations (PDEs) on surfaces (e.g. Laplace-Beltrami) interacting with PDEs in the bulk, systems with Lagrange multipliers that are only defined on boundaries, multi-physics problems, etc. In particular, **FELICITY** can be used rather effectively with a front-tracking approach for simulating moving-domain/free boundary problems. Moreover, **FELICITY** is a MATLAB toolbox, so it inherits the advantages of the MATLAB interface. Therefore, in my opinion, **FELICITY** fills a niche within the FEM software community.

FELICITY is not a GUI! If you want a plug-and-play package, this may not be for you. **FELICITY** provides a flexible high-level interface, a decent level of modularity, and some low-level access to the underlying code. In my experience, any real computational research problem will require some kind of non-standard modification of a pre-existing software package or you write your own from scratch. No single package will be able to handle **every** finite element formulation *and* remain simple to modify. Thus, **FELICITY** was developed in a way that is not *too* high level so that modifications will remain moderately easy. Of course, there is a trade-off here. You should choose a package that is right for you.

1.3.2 How Is FELICITY Implemented?

FELICITY is a mix of MATLAB and C++ code. Basically, MATLAB is used for most of the computations, while C++ is used for speed in certain areas (i.e. matrix assembly). All matrix manipulations (such as with sparse matrices) are automatically handled by MATLAB. In addition, MATLAB allows for easy manipulation of arrays and other data structures.

Some parts of **FELICITY** use automatic code generation, such as for assembling finite element matrices. This feature is high level, straightforward to use, and is based on a Domain-Specific-Language (DSM) implemented in **FELICITY**. In this case, **FELICITY** takes

inspiration from the FEniCS project [8, 9], which is a multi-purpose high-level Python/C++ finite element code for solving many different types of problems. However, at the time of this writing, FEniCS has not implemented low-dimensional domains in higher dimensions, e.g. 3-D space curves or 2-D surfaces in 3-D. **FELICITY** has this capability and can solve the Laplace-Beltrami operator on 2-D surfaces in 3-D. Moreover, **FELICITY** has the ability to use meshes with higher order geometric information, such as with curved quadratic elements. In addition, **FELICITY** can handle assembly of forms (i.e. computing integrals) over *any* sub-domains with co-dimension ≥ 0.

1.3.3 What Should You Know to Use FELICITY?

It is assumed that you are familiar with weak and variational formulations of PDE problems, and you have some experience in the finite element method. You should be familiar with at least one of the following references: [2, 3, 4, 5, 7].

1.4 Other FEM Packages

Here is an incomplete list of other popular (and free) finite element toolboxes that you may be interested in if **FELICITY** does not satisfy your need:

- `deal.II`: `http://www.dealii.org/`
- `feelpp`: `http://code.google.com/p/feelpp/`
- `FEniCS`: `http://fenicsproject.org/`
- `freeFEM++`: `http://www.freefem.org/`
- `GetFEM++`: `http://download.gna.org/getfem/html/homepage/`
- `LifeV`: `http://cmcsforge.epfl.ch/projects/lifev`
- `mfem`: `http://code.google.com/p/mfem/`

In addition, there is

- `milamin`: `http://milamin.sourceforge.net/`

which is also a MATLAB based finite element toolbox. Their web page provides some useful MATLAB tips on solving linear systems, applying boundary conditions, etc., that also apply to **FELICITY**. In fact, many of the `milamin` tools (e.g. quad-tree based interpolation on 2-D triangular meshes) can be used interchangeably with **FELICITY**.

1.5 To Do List

FELICITY is now fairly mature. Many of its components can be used interchangeably with each other or with user created code. For example, if all you want is the matrix assembly, then you can just use that. Some remaining items to be implemented in **FELICITY** are the following.

- Functionality for Discontinuous-Galerkin (DG) methods.
- Adaptive refinement/bisection in 3-D.
- Access to special purpose solvers.

If you use **FELICITY**, please send me an email at

`walker@math.lsu.edu`

I welcome all comments and/or suggestions, including bug reports.

Chapter 2

Basic Mesh Manipulation

2.1 Introduction

FELICITY provides methods for accessing and performing basic mesh queries and operations. The following mesh classes are implemented:

- (1-D) `MeshInterval`: meshes composed of line segments.
- (2-D) `MeshTriangle`: meshes composed of triangles.
- (3-D) `MeshTetrahedron`: meshes composed of tetrahedra.

Most of the functions (methods) within each class are the same across all three classes (see Section 2.5).

Note: the classes `MeshTriangle` and `MeshTetrahedron` are sub-classes of MATLAB's built-in class `TriRep` (see MATLAB documentation). The class `MeshInterval` is a sub-class of `EdgeRep`, which is a custom class (for 1-D meshes) designed to mimic the `TriRep` class.

2.2 Tutorial

2.2.1 2-D Square Mesh

We start with a simple 2-D mesh example. This tutorial can be found in the MATLAB script `test_Square_2D_Mesh.m` located in the sub-directory `..\FELICITY\Classes\Mesh\Unit_Test\Dim_2\Square`.

First define a simple square mesh composed of two triangles. At the MATLAB prompt (or in a script) type the following commands:

```
% define simple square mesh
Vtx = [0 0; 1 0; 1 1; 0 1]; % coordinates
Tri = [1 2 3; 3 4 1];       % triangle connectivity

% create a mesh object
Mesh = MeshTriangle(Tri,Vtx,'Square');
```

This will create a mesh object of class `MeshTriangle`. This mesh object gives access to many built-in routines. For example, you can compute the "qualities" of all triangles in the mesh and plot them:

```
% plot histogram of the "quality" of each simplex in the mesh
figure;
Qual = Mesh.Quality(5);
% "5" specifies the number of bins; "Qual" is a vector of quality values
```

If no argument is given to `Mesh.Quality`, then no plot is generated.

Next, we define *subdomains* that are embedded in the mesh. For instance, we will define the upper right vertex of the square mesh to be a 0-D subdomain:

```
% define a 0D subdomain of the square mesh
Mesh = Mesh.Append_Subdomain('0D','Upper Right Vtx',[3]);
% data representing subdomain can be accessed via:  Mesh.Subdomain(1).Data
```

The "`[3]`" in the argument specifies the global vertex index; we could also specify a vector of vertex indices.

The next subdomain will be the 1-D boundary of the mesh:

```
% get the boundary edges of the mesh
Bdy_Edge = Mesh.freeBoundary;
% define a 1D subdomain of the square mesh
Mesh = Mesh.Append_Subdomain('1D','Boundary',Bdy_Edge);
```

`Bdy_Edge` is an $M \times 2$ array that specifies the edge connectivity (M is the number of edges). Incidentally, the second argument `'Boundary'` is just an identifier (name) for the subdomain (so that we can access it easily later).

The last subdomain will be a 2-D mesh consisting of just one triangle:

```
% define a 2D subdomain of the square mesh
Mesh = Mesh.Append_Subdomain('2D','Lower Right Tri',[1]);
```

The "`[1]`" in the argument specifies the global triangle index; we could also specify a vector of triangle indices.

We are done defining the mesh and its subdomains. But we may want to perform operations on the mesh. For example, you can renumber the global vertex indices:

```
% reorder the nodes of the mesh automatically
% note: this will NOT affect the subdomains
Mesh = Mesh.Reorder;
```

You can also display useful information about the mesh by executing:

```
% display info about Mesh
display(Mesh); % you can also just type: >> Mesh <ENTER>
```

Plotting the mesh is easy:

```
% plot the mesh
figure;
Mesh.Plot;
```

And you can add plots of the subdomains:

```
% add plots of the subdomains
hold on;
Mesh.Plot_Subdomain('Lower Right Tri');
Mesh.Plot_Subdomain('Boundary');
Mesh.Plot_Subdomain('Upper Right Vtx');
hold off;
title('Initial Mesh');
```

Here, you must specify the subdomain *name* so the mesh object knows which to plot. Mesh refinement is also something you can do:

```
% uniformly refine the entire mesh by
%           dividing each triangle into 4 triangles
Mesh = Mesh.Refine;
% reorder again
Mesh = Mesh.Reorder;
% refine again
Mesh = Mesh.Refine;
% reorder again
Mesh = Mesh.Reorder;
```

Here, we refined (and reordered) twice. The mesh class is able to *manage the subdomains through refinement*, i.e. the geometric structure of the subdomains stays the same, but refinement causes new elements to be added. The reordering is not necessary but is done to show that reordering does not affect the subdomains. Everything was automatically done.

To prove that is indeed the case, plot the new mesh (and its subdomains) and visually check the consistency:

```
% plot everything again
figure;
Mesh.Plot;
hold on;
Mesh.Plot_Subdomain('Lower Right Tri');
Mesh.Plot_Subdomain('Boundary');
Mesh.Plot_Subdomain('Upper Right Vtx');
hold off;
title('After Uniform Refinement');
```

Everything should match up.

2.2.2 Selective Mesh Bisection in 2-D

Here we give some practice in adaptively refining 2-D triangular meshes via Rivara Bisection [10]. This tutorial can be found in the MATLAB script `test_Bisection_2D_Square.m` located in the sub-directory:
`..\FELICITY\Classes\Mesh\Unit_Test\Dim_2\Bisection_2D`.

We proceed similarly to Section 2.2.1. At the MATLAB prompt (or in a script) type the following commands:

```
% define simple square mesh
Vtx = [0 0; 1 0; 1 1; 0 1]; % coordinates
Tri = [1 2 3; 3 4 1];       % triangle connectivity

% create a mesh object
Mesh = MeshTriangle(Tri,Vtx,'Square');

% define a 0D subdomain of the square mesh
Mesh = Mesh.Append_Subdomain('0D','Upper Right Vtx',[3]);
% data representing subdomain can be accessed via:  Mesh.Subdomain(1).Data

% define a 1D subdomain of the square mesh
Diag_Edge = [1 3]; % edge between vertices 1 and 3
Mesh = Mesh.Append_Subdomain('1D','Diagonal',Diag_Edge);

% define a 2D subdomain of the square mesh
Mesh = Mesh.Append_Subdomain('2D','Lower Right Region',[1]);
```

And you can plot everything just as you did in the previous tutorial:

```
% plot the mesh
figure;
Mesh.Plot;
% add plots of the subdomains
hold on;
Mesh.Plot_Subdomain('Lower Right Region');
Mesh.Plot_Subdomain('Diagonal');
Mesh.Plot_Subdomain('Upper Right Vtx');
hold off;
title('Initial Mesh');
```

Now let us bisect one triangle. This requires making a list of "marked" triangles and inputting it to the refinement routine. For instance, we bisect triangle #1 by the following command:

```
% bisect upper left triangle into two triangles
% of course, this will cause the other triangle to get bisected to maintain
% conformity
Mesh = Mesh.Refine('bisection',[1]);
```

Next, let us bisect a subset of triangles in the new mesh:

```
% bisect the (new) #2 and #4 triangles
Mesh = Mesh.Refine('bisection',[2; 4]);
```

We can bisect *all* triangles simply by **not** specifying a "marked" list:

```
% refine all triangles by bisection
Mesh = Mesh.Refine('bisection');
```

Now lets use the sub-domains as a way to mark the triangles to bisect. We can do this by finding all triangles (cells) that a particular sub-domain references (see Section 2.3.2). We then feed this list to the refinement routine. For example, this can be done for the "`Upper Right Vtx`" subdomain:

```
% refine all triangles that reference the ''Upper Right Vtx'' by bisection
Mark_Tri = Mesh.Get_Subdomain_Cells('Upper Right Vtx');
Mesh = Mesh.Refine('bisection',Mark_Tri);
```

We can do the same for a 2-D sub-domain:

```
% refine all triangles contained in the ''Lower Right Region'' by bisection
Mark_Tri = Mesh.Get_Subdomain_Cells('Lower Right Region');
Mesh = Mesh.Refine('bisection',Mark_Tri);
```

And, of course, we can do this for a 1-D subdomain:

```
% refine all triangles that reference the ''Diagonal'' by bisection
Mark_Tri = Mesh.Get_Subdomain_Cells('Diagonal');
Mesh = Mesh.Refine('bisection',Mark_Tri);
```

While we are it, lets refine again:

```
% refine again
Mark_Tri = Mesh.Get_Subdomain_Cells('Diagonal');
Mesh = Mesh.Refine('bisection',Mark_Tri);
```

Now plot the mesh and sub-domains, just as we did above, and verify that the sub-domains are preserved.

Remark 2.1.

- *WARNING: make sure you run* `test_FELICITY.m` *before doing this tutorial because the bisection routine uses a* `C++/MEX` *file to implement the Rivara Longest-Edge-Propagation-Path (LEPP) Bisection method.* `test_FELICITY.m` *will automatically compile the* `MEX` *file for you.*
- *The LEPP Bisection* `C++` *code can be found in* `../FELICITY/Static_Codes/Lepp_Bisection_2D/src_code/`.

2.3 Details of The Mesh Class

2.3.1 Mesh Data Structure

All of the mesh classes in **FELICITY** use a standard data structure to store meshes. The layout is:

```
Mesh.Name
Mesh.X
Mesh.Triangulation
Mesh.Subdomain
```
(2.1)

where `Mesh` is an object whose type is a mesh class (see Section 2.1), and each variable is described below.

- `Name`: a string variable to label what the mesh is.
- `X`: list of vertex coordinates. It is an $N \times d$ array, where N is the number of vertices and d is the geometric dimension (i.e. $d = 1, 2, \text{or } 3$).
- `Triangulation`: element connectivity data. It is an $M \times (t+1)$ array, where M is the number of elements and t is the topological dimension (i.e. $t = 1, 2, \text{or } 3$). The topological and geometric dimension must satisfy $d \geq t$.
- `Subdomain`: an array of `struct` variables that represent a subdomain embedded in the underlying mesh. Each subdomain can be accessed by `Mesh.Subdomain(ind)` for a fixed index `ind`.

The most important parts are `X` and `Triangulation`, which look like:

$$\texttt{X} = \underbrace{\begin{bmatrix} x_1 & \cdots & z_1 \\ x_2 & \cdots & z_2 \\ \vdots & \vdots & \vdots \\ x_N & \cdots & z_N \end{bmatrix}}_{d}, \qquad \texttt{Triangulation} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1,t+1} \\ c_{21} & c_{22} & \cdots & c_{2,t+1} \\ \vdots & \vdots & \vdots & \vdots \\ c_{M,1} & c_{M,2} & \cdots & c_{M,t+1} \end{bmatrix}, \tag{2.2}$$

where c_{ij} are positive integers indexing into the rows of `X`. Note: MATLAB's `TriRep` class also uses `X` and `Triangulation`; same goes for the custom class `EdgeRep`.

2.3.2 Subdomain Data Structure

Subdomain information is stored in the following way.

- `Mesh.Subdomain(ind)` has three sub-fields defined as
  - `Name`: identifier that the user specifies when the subdomain is created.
  - `Dim`: topological dimension of the subdomain, which must be less than or equal to the topological dimension of the underlying mesh.

– `Data`: a $K \times (1 \text{ or } 2)$ array (see below).

The information that defines the subdomain is stored indirectly in the array `Subdomain.Data`, and its format depends on the topological dimension of the underlying mesh *that contains the subdomain* and the topological dimension of the subdomain itself. This is best described on a case-by-case basis.

Subdomains of 1-D Meshes

For subdomains of 1-D meshes, there are two possibilities:

- 0-D subdomain: set of individual vertices contained in the underlying 1-D mesh.
- 1-D subdomain: subset of the edge elements in the underlying 1-D mesh.

The main idea is to store the appropriate *cell* indices (in this case, edge element indices) with a *pointer* to the *local* mesh entity. For example,

$$\text{storing 0-D subdomain of 1-D mesh} \rightarrow \texttt{Subdomain.Data} = \begin{bmatrix} C_1 & 1 \\ C_2 & 2 \\ \vdots & \vdots \\ C_K & 1 \end{bmatrix}, \tag{2.3}$$

where $\{C_i\}$ are the edge element (cell) indices (in the 1-D mesh that contains the subdomain). The second column entries are either 1 or 2, depending on the *local* vertex (in the corresponding cell) to which it refers (see Section 2.4.2).

Example 2.1. *For instance, if one row of* (2.3) *is the row vector "* $[12, 2]$*", then it refers to the #2 local vertex on edge #12 (see Section 2.4.2). Then all that is needed to "find" the global index of that vertex is to look at edge #12 in the underlying mesh and read the index of* local *vertex #2.*

Remark 2.2. *This* ***same scheme*** *is used for higher dimensions, e.g. 2-D subdomains in a 3-D mesh (see the following sections). This indirect storage of subdomains is useful for two reasons: (1) you can always reconstruct the global index data and (2) more importantly, it* attaches *a cell (of the original mesh) to each part of the subdomain. In other words,* this storage scheme captures how the subdomain is embedded in the underlying mesh*. This has advantages for higher dimension subdomains, such as for assembling bilinear forms over domains of codimension 1 (or greater).*

Remark 2.3. *The way in which a subdomain is embedded is not unique, meaning that more than one cell could be used to store a particular vertex (or edge or face in higher dimensions).*

Note that the storage of subdomains of equal topological dimension as the underlying mesh is simplified (e.g. *subset* of edge elements in a 1-D mesh). In this case,

$$\text{storing 1-D subdomain of 1-D mesh} \rightarrow \texttt{Subdomain.Data} = \begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_K \end{bmatrix}, \tag{2.4}$$

where $\{C_i\}$ are the edge element (cell) indices (in the underlying 1-D mesh).

Subdomains of 2-D Meshes

Proceeding similarly, for subdomains of 2-D meshes, there are three possibilities:

- 0-D subdomain: set of individual vertices contained in the underlying 2-D mesh.
- 1-D subdomain: set of directed (oriented) edges in the underlying 2-D mesh.
- 2-D subdomain: subset of triangle elements in the underlying 2-D mesh.

For example,

$$\text{storing 0-D subdomain of 2-D mesh} \rightarrow \texttt{Subdomain.Data} = \begin{bmatrix} C_1 & 1 \\ C_2 & 3 \\ C_3 & 2 \\ \vdots & \vdots \\ C_K & 2 \end{bmatrix}, \tag{2.5}$$

where $\{C_i\}$ are the triangle (cell) indices (in the 2-D mesh that contains the subdomain). The second column entries are either 1, 2, or 3, depending on the *local* vertex (in the corresponding cell) to which it refers (see Section 2.4.3).

Next, for

$$\text{storing 1-D subdomain of 2-D mesh} \rightarrow \texttt{Subdomain.Data} = \begin{bmatrix} C_1 & -1 \\ C_2 & 3 \\ C_3 & 2 \\ \vdots & \vdots \\ C_K & -2 \end{bmatrix}, \tag{2.6}$$

where the first column is similarly defined as before. The second column entries are either ± 1, ± 2, or ± 3, and they refer to the *local* edge index (in the corresponding cell) with a sign to indicate its *orientation* (or direction). This differs from the vertex case, because vertices do not have an orientation. Also see Section 2.4.3.

Lastly, we have for

$$\text{storing 2-D subdomain of 2-D mesh} \rightarrow \texttt{Subdomain.Data} = \begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_K \end{bmatrix}, \tag{2.7}$$

where $\{C_i\}$ are the triangle (cell) indices (in the underlying 2-D mesh).

Subdomains of 3-D Meshes

The general pattern should be clear now. For subdomains of 3-D meshes, there are four possibilities:

- 0-D subdomain: set of individual vertices contained in the underlying 3-D mesh.

- 1-D subdomain: set of oriented edges in the underlying 3-D mesh.
- 2-D subdomain: set of oriented faces in the underlying 3-D mesh.
- 3-D subdomain: subset of tetrahedral elements in the underlying 3-D mesh.

For example,

$$\text{storing 0-D subdomain of 3-D mesh} \rightarrow \texttt{Subdomain.Data} = \begin{bmatrix} C_1 & 1 \\ C_2 & 3 \\ C_3 & 4 \\ C_4 & 1 \\ \vdots & \vdots \\ C_K & 2 \end{bmatrix}, \tag{2.8}$$

where $\{C_i\}$ are the tetrahedral (cell) indices (in the 3-D mesh that contains the subdomain). The second column entries are either 1, 2, 3, or 4, depending on the *local* vertex (in the corresponding cell) to which it refers (see Section 2.4.4).

Next, for

$$\text{storing 1-D subdomain of 3-D mesh} \rightarrow \texttt{Subdomain.Data} = \begin{bmatrix} C_1 & 1 \\ C_2 & -6 \\ C_3 & 3 \\ \vdots & \vdots \\ C_K & -4 \end{bmatrix}, \tag{2.9}$$

where the first column is similarly defined as before. The second column entries take values in $\pm\{1, 2, 3, 4, 5, 6\}$ and they refer to the *local* edge index (in the corresponding cell) with a sign to indicate its *orientation* (see Section 2.4.4).

We also have for

$$\text{storing 2-D subdomain of 3-D mesh} \rightarrow \texttt{Subdomain.Data} = \begin{bmatrix} C_1 & 4 \\ C_2 & 1 \\ C_3 & -2 \\ \vdots & \vdots \\ C_K & -3 \end{bmatrix}, \tag{2.10}$$

where the first column is similarly defined as before. The second column entries take values in $\pm\{1, 2, 3, 4\}$ and they refer to the *local* face index (in the corresponding cell) with a sign to indicate its *orientation* (see Section 2.4.4).

Lastly, we have for

$$\text{storing 3-D subdomain of 3-D mesh} \rightarrow \texttt{Subdomain.Data} = \begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_K \end{bmatrix}, \tag{2.11}$$

where $\{C_i\}$ are the tetrahedral (cell) indices (in the underlying 3-D mesh).

2.4 Local Mesh and Topological Entities

It only remains to define what we mean by local edge #5, local face #3, and so on. In other words, we need to fix the definitions of the local topological entities (vertices, edges, faces, and cells).

In the following sections, we lay out the assumptions that **FELICITY** uses to define the standard reference cells and how the local topological entities are labeled.

2.4.1 Vertices

This is the most trivial case (vertices can never be cells). Vertices are referenced via a global index that points to a row of `X`, or by a local index of a given cell in the underlying mesh.

2.4.2 Reference Interval

In 1-D, the reference element is the unit interval $[0,1] \subset \mathbb{R}$. An *edge* E in a mesh is a directed line segment joining two vertices v_1 and v_2 (see Figure 2.1). The reference element is denoted by $\widehat{E} \equiv [0,1] \subset \mathbb{R}$. For any edge E, there always exists an affine map from the unit interval to E (for all geometric dimensions), i.e. $\Phi_E : \widehat{E} \to E$.

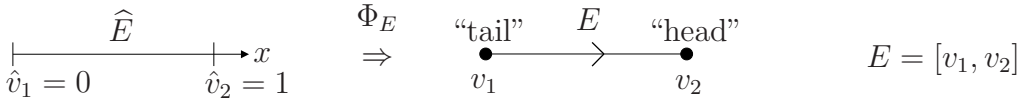

Figure 2.1: Reference **edge** and topology definition. An edge is an ordered pair of vertices; note that E has an orientation/direction (the arrow is pointing from v_1 to v_2). v_j is a positive integer (*global* index) that references the vertex coordinate list (see (2.2)); the *local* index of v_j is j. Note that E is not an interval (despite similarity in the notation); E is a line segment. $\widehat{E}$ can be identified with $[\hat{v}_1, \hat{v}_2]$ where $\hat{v}_1 = 0$ and $\hat{v}_2 = 1$ are the reference coordinates.

Remark 2.4. *One dimensional meshes consist of a set of edges $\{E_i\}$ as described in Figure 2.1, i.e. $E_i = [v_{i1}, v_{i2}]$.*

2.4.3 Reference Triangle

In 2-D, the reference element is the logical triangle $\widehat{T} \subset \mathbb{R}^2$, (see Figure 2.2). A *triangle* T in a mesh is a simplex defined by three vertices v_1, v_2, and v_3 (see Figure 2.2). For any triangle T, there always exists an affine map from $\widehat{T}$ to T (for all geometric dimensions greater than 1), i.e. $\Phi_T : \widehat{T} \to T$.

Note that the local edges e_j (as defined in Figure 2.2) are assumed to be *positively* oriented. Thus, a *negatively* oriented edge is denoted with a minus sign and indicates to swap the two vertex indices (with respect to the positive case), e.g.

$$\text{if } e_j = [5, 13], \quad \text{then} \quad -e_j = [13, 5]. \tag{2.12}$$

This sign notation is useful when storing 1-D subdomains of 2-D meshes (recall Section 2.3.2).

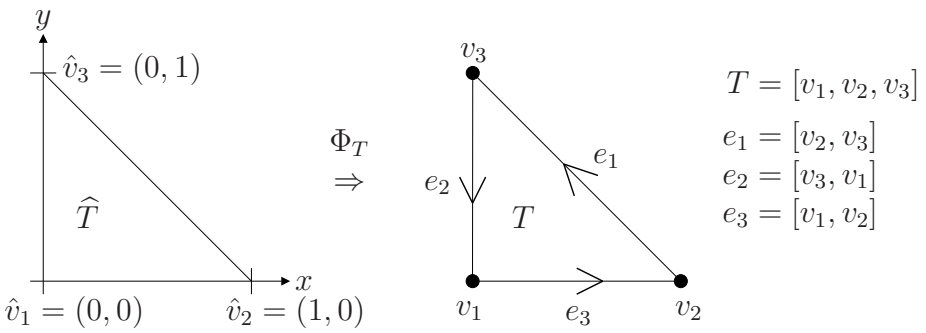

Figure 2.2: Reference **triangle** and topology definition. A triangle is an ordered triple of vertices; note that T has an orientation/direction (the direction of the edges e_j of T, and the direction of the unit normal vector of T, are determined by the "right-hand-rule"). e_j is a *local* edge of T that references two vertices in the global coordinate list (see (2.2)) and is always opposite the vertex v_j; the *local* index of e_j is j. $\widehat{T}$ can be identified with $[\hat{v}_1, \hat{v}_2, \hat{v}_3]$ where $\hat{v}_1 = (0,0)$, $\hat{v}_2 = (1,0)$, and $\hat{v}_3 = (0,1)$ are the reference coordinates.

2.4.4 Reference Tetrahedron

In 3-D, the reference element is the logical tetrahedron $\widehat{T} \subset \mathbb{R}^3$, (see Figure 2.3). A *tetrahedron* T in a mesh is a simplex defined by four vertices v_1, v_2, v_3, and v_4 (see Figure 2.3). For any tetrahedron T, there always exists an affine map from $\widehat{T}$ to T (for all geometric dimensions greater than 2), i.e. $\Phi_T : \widehat{T} \to T$.

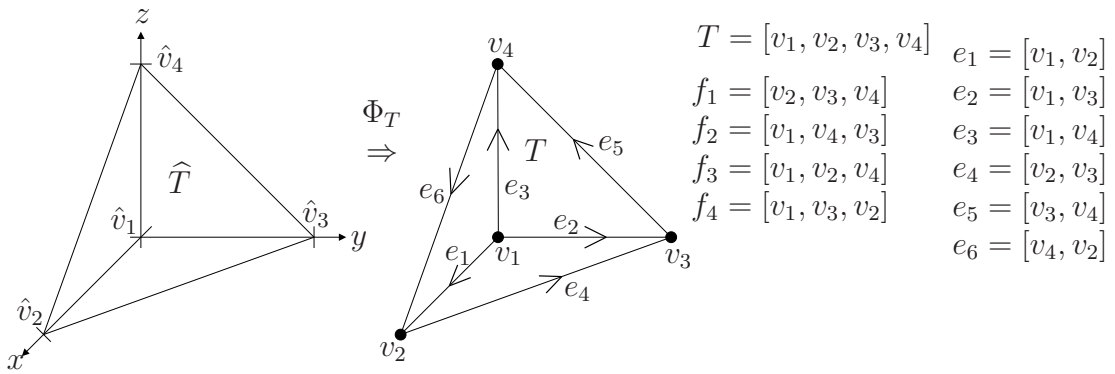

Figure 2.3: Reference **tetrahedron** and topology definition. A tetrahedron is an ordered quadruple of vertices; note that T has an orientation/direction. The faces f_j of T have an outward pointing normal vector with respect to the "right-hand-rule" (recall Figure 2.2). f_j is a *local* face of T that references three vertices in the global coordinate list (see (2.2)) and is always opposite the vertex v_j; the *local* index of f_j is j. Note that the faces are not labeled in the figure but the directed edges are labeled. $\widehat{T}$ can be identified with $[\hat{v}_1, \hat{v}_2, \hat{v}_3, \hat{v}_4]$ where $\hat{v}_1 = (0,0,0)$, $\hat{v}_2 = (1,0,0)$, $\hat{v}_3 = (0,1,0)$, and $\hat{v}_4 = (0,0,1)$ are the reference coordinates.

Note that the local edges e_j and faces f_j (as defined in Figure 2.3) are assumed to be *positively* oriented. Thus, a *negatively* oriented edge or face is denoted with a minus sign. In the case of a face, a minus sign indicates to swap any two vertex indices (with respect to

the positive case). But for the sake of specificity, **FELICITY** *assumes the second and third vertices are swapped*, e.g.

$$\text{if } f_j = [5, 24, 17], \quad \text{then} \quad -f_j = [5, 17, 24]. \tag{2.13}$$

This sign notation is useful when storing 2-D subdomains of 3-D meshes (recall Section 2.3.2).

Remark 2.5 (Local Edges of Faces)**.** *In addition to the above topology definitions, there are also the local edges* of each local face. *For example, local edge #1 of face #3 is the directed edge defined by* $f_3(2:3) \equiv [v_2, v_4]$. *It is left as an exercise for you to work out the rest.*

2.5 Mesh Class Methods

For the classes `MeshTriangle` and `MeshTetrahedron`, the available methods are:

```
Angles                              Plot                      edgeAttachments
Append_Subdomain                    Plot_Subdomain            edges
Create_Embedding_Data               Quality                   faceNormals
Create_Subdomain                    Refine                    featureEdges
Generate_Subdomain_Embedding_Data   Remove_Unused_Vertices    freeBoundary
Geo_Dim                             Reorder                   incenters
Get_Adjacency_Matrix                Set_X                     isEdge
Get_Global_Subdomain                Top_Dim                   neighbors
Get_Subdomain_Cells                 Volume                    refToBary
Get_Subdomain_Index                 baryToCart                size
MeshTriangle/MeshTetrahedron        baryToRef                 vertexAttachments
Num_Cell                            cartToBary
Num_Vtx                             circumcenters
Output_Subdomain_Mesh               display
```

Note: `Refine` is only implemented for `MeshTriangle`.

For the class `MeshInterval`, the available methods are:

```
Angles                              Output_Subdomain_Mesh     circumcenters
Append_Subdomain                    Plot                      display
Create_Embedding_Data               Plot_Subdomain            edgeTangents
Create_Subdomain                    Quality                   edges
Generate_Subdomain_Embedding_Data   Refine                    freeBoundary
Geo_Dim                             Remove_Unused_Vertices    isEdge
Get_Adjacency_Matrix                Reorder                   neighbors
Get_Global_Subdomain                Set_X                     refToBary
Get_Subdomain_Cells                 Top_Dim                   size
Get_Subdomain_Index                 Volume                    vertexAttachments
MeshInterval                        baryToCart
Num_Cell                            baryToRef
Num_Vtx                             cartToBary
```

You can see the list of methods in MATLAB by applying the `methods` command (see MATLAB help for this) to the class of interest. The best way to gain familiarity with the different methods is to use the `help` command in MATLAB (applied to the method of interest).

Chapter 3

Defining Finite Elements

3.1 Introduction

Most things in **FELICITY** depend on a "finite element m-file." This file defines a finite element (close to the Ciarlet sense [5]) by specifying the reference domain, shape functions (i.e. local basis functions), and the corresponding nodal Degree-of-Freedom (DoF) topological arrangement. This m-file is basically a configuration file that does the following:

- Puts all of the essential information that defines the finite element into an easily editable form.
- Allows for easily implementing new/different finite elements.

The format of the m-file is explained below (examples are included with **FELICITY**).

3.2 Finite Element M-File Format

Here is a sample `.m` file that is explained in detail in the subsequent sections. It can be found in the `..\FELICITY\Elem_Defn` sub-directory of **FELICITY**. This example defines a continuous piecewise linear element over the 1-D reference interval $[0, 1]$.

```
function Elem = lagrange_deg1_dim1()
%lagrange_deg1_dim1
%
%   This defines a finite element to be used by FELICITY.

% name it
Elem.Name = mfilename;

% continuous galerkin space
Elem.Type = 'CG';

% intrinsic dimension and domain
Elem.Dim = 1;
```

```
Elem.Domain = 'interval';

% basis function definitions
Elem.Basis.Func =...
    {{'1 - x'};
     {'x'}};
% local mapping transformation to use
Elem.Basis.Transformation = 'H1_Trans';

% nodal variables (dual basis)
% barycentric coordinates (point evaluation)
% Note: this part is experimental
Elem.Nodal_Var.Type  = 'point evaluation';
Elem.Nodal_Var.Basis =...
    {{'phi'}, [1,0];
     {'phi'}, [0,1]};

%%% nodal (topological) arrangement

% nodes attached to vertices
Elem.Nodal_Top.V = {[1;
                     2]};

% nodes attached to (directed) edges
Elem.Nodal_Top.E = {[]};

% nodes attached to (triangles) faces
Elem.Nodal_Top.F = {[]}; % no triangle in 1-D

% nodes attached to tetrahedra
Elem.Nodal_Top.T = {[]}; % no tetrahedra in 1-D

end
```

Note that anything *after* a percent sign `%` is considered a comment by MATLAB.

3.2.1 Basic Items

The element m-file creates a struct variable `Elem` with sub-fields. Some of those fields are

- `Name`: a unique identifier (string) for the element.
- `Type`: specifies either continuous-Galerkin (CG) or discontinuous-Galerkin (DG).
- `Dim`: topological dimension of the reference domain for the element.
- `Domain`: type of domain, i.e. `interval`, `triangle`, `tetrahedron`.

3.2.2 Basis Functions

The next sub-field is `Basis`, which has two sub-fields `Func` and `Transformation`. `Func` is an $N \times 1$ cell array (see MATLAB for documentation on cell arrays), where N is the number of basis functions for the finite element.

Each entry of `Func` is *also a cell array* of size $r \times l$, and is used to represent (in general) a matrix-valued basis function *symbolically* (with dimensions $r \times l$). So each entry of the sub-cell array is a string variable representing the shape function for a particular component of the basis function. Hence, each row of `Func` completely specifies each basis function symbolically.

For example, if $r = l = 1$, then the basis functions are just scalars. If $r = 2$ and $l = 1$, then each basis function is a 2-D vector.

Remark 3.1. *The Degree-of-Freedom (DoF) index for each basis function in* `Func` *is its row index.*

The other sub-field, `Transformation`, is a string indicating which transformation to use when mapping the basis functions to an element in a mesh:

- `Transformation = 'H1_Trans'` indicates to use the standard transformation for H^1 functions.
- `Transformation = 'Hdiv_Trans'` indicates to use the Piola transformation that preserves the divergence operator.

Note that the field `Basis` can also be an array, which allows for defining tensor product *mixed* finite elements (but this is not yet implemented).

3.2.3 Dual Basis

The next field is `Nodal_Var`, but this is experimental. Please ignore it for now.

3.2.4 Nodal Topology

The last field is `Nodal_Top`, and it is used to specify the topological arrangement of the basis functions/dual basis (see Section 3.3).

3.2.5 How To Output Information About A Finite Element

FELICITY has a MATLAB class that will output important information for any given finite element that is implemented as in Section 3.2. For example, type the following at the MATLAB prompt:

```
Elem_Info = FELOutputElemInfo(lagrange_deg2_dim2);

Elem_Info.Print_Basis_Functions('latex');
Elem_Info.Print_Basis_Functions;
Elem_Info.Print_DoFs;
```

The first line defines the MATLAB object with the given element being piecewise quadratic Lagrange polynomials in 2-D (i.e. on a triangle). The second line prints (to the MATLAB window) a snippet of LaTeX code that defines the basis functions of the element. The third line simply "pretty" prints the basis function definitions (i.e. plain text). The last line prints information about how the Degrees-of-Freedom (DoFs) are arranged on the reference simplex (i.e. which nodes are attached to vertices, edges, faces, or tetrahedral cells).

3.3 Nodal Topological Arrangement

No matter what the basis functions are in your particular finite element, when it comes to allocating the Degrees-of-Freedom (DoF) on a given mesh, all that matters is the arrangement of the nodal DoFs on a single element in the mesh.

This section describes how to define the nodal DoF arrangement on a simplex so that **FELICITY** can automatically generate code to (consistently) allocate DoFs over an entire mesh. In other words, to ensure certain continuity requirements of the elements, care must be taken in the placement of the nodes [3]. This is accomplished by properly defining `Nodal_Top`, which provides a convenient way of partitioning the DoFs of a finite element.

Remark 3.2. *The consistent arrangement of the nodal DoFs only matters for continuous finite elements. For discontinuous-Galerkin (DG), allocating DoFs is trivial.*

The arrangement is specified by the struct `Nodal_Top` under `Elem` and could be an array of structs, which is useful for tensor product elements (e.g. velocity and pressure). But this is not yet implemented, so we will assume `Nodal_Top` is a single struct.

$$\text{The sub-fields } \texttt{V}, \texttt{E}, \texttt{F}, \texttt{T} \text{ are MATLAB cell arrays of size } 1 \times M, \tag{3.1}$$

where M may be different for each field (familiarize yourself with MATLAB's cell array capabilities before proceeding). The sub-fields contain the following information.

- `V`: stores which DoFs are attached to the vertices of the simplex.
- `E`: stores which DoFs are attached to the edges of the simplex.
- `F`: stores which DoFs are attached to the faces of the simplex.
- `T`: stores which DoFs are attached to the tetrahedra of the simplex.

Note: **FELICITY** only goes up to 3-D. The format of `Nodal_Top` depends on the topological dimension of the reference domain on which the finite element space is defined (as well as the finite element space itself). This is described in the following sections.

3.3.1 Nodes Attached to Vertices

The data structure of `V` is

$$\texttt{V} = \{S_1, ..., S_M\}, \text{ where } S_j \text{ is a matrix for all } j; \tag{3.2}$$

Each S_j is a set of DoFs (attached to vertices) and looks like this:

$$S_j = \begin{bmatrix} n_1 & \cdots & n_{1+K} \\ n_2 & \cdots & n_{2+K} \\ \vdots & \vdots & \vdots \\ n_{t+1} & \cdots & n_{t+K} \end{bmatrix}, \text{ where } n_l \text{ are local DoF indices,} \tag{3.3}$$

t is the topological dimension of the reference simplex, and Row k of S_j gives the set of DoFs attached to local vertex k (v_k). The number of columns in S_j depends on the particular finite element space you are defining.

Example 3.1. *On a mesh of edge segments there are two vertices per simplex (topological dimension is $t = 1$). Suppose the finite element space was piecewise linear on each edge. Then* `V` $= \{S_j\}_{j=1}^{1}$ *and*

$$S_1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, \tag{3.4}$$

which indicates that DoF #1 is attached to v_1 and DoF #2 is attached to v_2. Of course, this depends on how you numbered the basis functions (see Remark 3.1).

Remark 3.3. *Thus, when you define your finite element, you just need to fill in the* `V` *data structure with the correct DoF indices that are attached to each vertex (see examples). And similarly for* `E`*,* `F`*, and* `T` *(see the following sections).*

Remark 3.4. *The cell array storage scheme may seem too general, but consider this: as you increase the polynomial degree of a finite element, the set of DoFs can become quite complicated with different DoFs attached to vertices, edges, and simplex interiors, for example. Hence, we need a general way of defining the layout of the DoFs such that* ***FELICITY*** *can use it to automatically generate code for* ***any*** *nodal arrangement. In other words, this provides a way of easily extending and implementing new and different finite elements, without digging through code.*

3.3.2 Nodes Attached to Edges

The data structure of `E` is also a cell array just like in (3.2). But in this case, each S_j is a set of DoFs (attached to edges) and looks like this

$$S_j = \begin{bmatrix} n_1 & \cdots & n_{1+K} \\ n_2 & \cdots & n_{2+K} \\ \vdots & \vdots & \vdots \\ n_q & \cdots & n_{q+K} \end{bmatrix}, \text{ where } n_l \text{ are local DoF indices,} \tag{3.5}$$

q is determined by (3.6), and Row k of S_j gives the set of DoFs attached to local edge k (e_k). The number of columns in S_j depends on the particular finite element space you are defining. The number of rows q of (3.5) depends on the topological dimension t by

$$q(t) = \begin{array}{l} 1, \text{ if } t = 1 \\ 3, \text{ if } t = 2 \\ 6, \text{ if } t = 3 \end{array}, \quad \text{(see Section 2.4).} \tag{3.6}$$

Remark 3.5. *The edge case is different from the vertex case. In each row of S_j, the* ***order*** *of DoF indices matters. You must order the DoFs along the directed edge (see Section 2.4) to ensure the correct continuity requirements [3]. In short, you should be well-versed in the finite element method and know what you are doing before attempting this.*

Remark 3.6. Exception: *when the edge* is *the reference simplex, the order of DoFs does not matter because they are* interior *degrees of freedom. However, it is good form to order them in a sensible way.*

Example 3.2. *On a mesh of triangles there are three vertices and three edges per simplex (topological dimension is $t = 2$). Suppose the finite element space was piecewise quadratic on each triangle. Then there is only one entry each in the cell arrays* `V` *and* `E`*:*

$$\texttt{V\{1\}} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \qquad \texttt{E\{1\}} = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix}, \tag{3.7}$$

which indicates that DoF #k is attached to v_k for $1 \leq k \leq 3$ and DoF #$k+3$ is attached to e_k for $1 \leq k \leq 3$. Of course, this depends on how you numbered the basis functions (see Remark 3.1). Note: the `F` *struct is empty in this case because there are no internal DoFs for the $\mathcal{P}_2$ element on a triangle.*

3.3.3 Nodes Attached to Faces

The data structure of `F` is also a cell array just like in (3.2). But each S_j is a set of DoFs (attached to faces) and looks like this

$$S_j = \begin{bmatrix} n_1 & \cdots & n_{1+K} \\ n_2 & \cdots & n_{2+K} \\ \vdots & \vdots & \vdots \\ n_q & \cdots & n_{q+K} \end{bmatrix}, \text{ where } n_l \text{ are local DoF indices,} \tag{3.8}$$

q is determined by (3.9), and Row k of S_j gives the set of DoFs attached to local face k (f_k). The number of rows q of (3.8) depends on the topological dimension t by

$$q(t) = \begin{matrix} 1, \text{ if } t = 2 \\ 4, \text{ if } t = 3 \end{matrix}, \quad \text{(see Section 2.4).} \tag{3.9}$$

Remark 3.7. *Similar to the previous section, the order of DoFs (in each row) matters for continuity purposes. However, the face case adds an additional complication. For instance, the DoFs* attached *to a face of a tetrahedron (i.e. the interior DoFs of that face) must satisfy a* 3-fold symmetry *or be a* single *DoF at the barycenter of the face. For the 3-fold symmetry case, each row of S_j is further partitioned into three sections, i.e.*

$$\text{DoFs for face } f_k \rightarrow S_j(k,:) = [D_1, D_2, D_3], \tag{3.10}$$

where D_1, D_2, and D_3 are all row vectors of equal length and D_l contains the DoFs associated with the ***local*** *edge l of face k. Note: the local edge referred to here is with respect to the local face (****not*** *the six edges of the tetrahedron); recall Remark 2.5.*

For the single DoF case, the situation is simpler and S_j is simply a column vector.

Remark 3.8. Exception: *when the face* is *the reference simplex, the order of DoFs does not matter because they are* interior *degrees of freedom. However, it is good form to order them in a sensible way.*

Example 3.3. *On a mesh of tetrahedra there are four vertices, six edges, and four faces (triangles) per simplex (topological dimension is $t = 3$). Suppose the finite element space is BDM_1, which consists of three (independent) linear 3-D vector valued functions on each triangle/face. Then there are no DoFs associated with the vertices or edges, so* `V` *and* `E` *are empty. On the other hand, the cell array* `F` *is given by:*

$$\texttt{F\{1\}} = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix}. \tag{3.11}$$

For instance, DoF #2, #6, #10 are all attached to face #2. Moreover, DoF #7 is associated with ***local*** *edge #2 of face #3. Of course, you can number the basis functions differently (see Remark 3.1). Note: the* `T` *struct is empty in this case because there are no internal DoFs for the BDM_1 element on a tetrahedron.*

Remark 3.9. *If the set of DoFs on faces can be partitioned into two sets, one set with* 3-fold symmetry *and the other set consisting of a* single *DoF at the barycenter of each face, then* `F` *must be a cell array with two entries:*

$$\texttt{F} = \{S_1, S_2\}, \tag{3.12}$$

where S_1, S_2 have the form of (3.8)*, with S_1 corresponding to the 3-fold symmetry DoFs and S_2 corresponding to the barycenter DoF. If there are multiple sets of each, then* `F` *must have more entries accordingly.*

3.3.4 Nodes Attached to Tetrahedra

The data structure of `T` is also a cell array just like in (3.2). But each S_j is a row vector containing a set of interior DoFs (attached to a tetrahedron) and looks like this

$$S_j = \begin{bmatrix} n_1 & \cdots & n_K \end{bmatrix}, \text{ where } n_l \text{ are local DoF indices.} \tag{3.13}$$

Remark 3.10. *If tetrahedra are present in the mesh, then the topological dimension must be 3 (****FELICITY*** *is limited to this case). Thus, only* interior *degrees of freedom are possible (in the tetrahedron) so the order of the indices does not matter. However, it is good form to order them in a sensible way.*

3.3.5 Conclusion

We emphasize that all of the indices stored under `Nodal_Top` must be distinct. No single index can appear twice. If it does, then **FELICITY** will give you an error message if you try to use that element m-file. Remember that each DoF index points to a row in `Elem.Basis.Func` (see Remark 3.1).

Chapter 4

Assembling Matrices

4.1 Automatically Generating Code

For assembling finite element matrices, **FELICITY** takes the following approach. You specify an input M-file that defines certain things:

- The quadrature order.
- The finite element spaces you want to use.
- The bilinear forms (matrices) you want to create.
- Etc.

Writing the script file is straightforward. Next, use **FELICITY** to process the script file into a stand-alone matrix assembly `C++` code. Then, compile it with a `C++` compiler. The following sections walk you through this process (example files are included).

4.2 Example 1-D Problem

Consider the boundary value problem (classical formulation)

$$\begin{aligned} -u''(s) &= -1, \text{ for all } s \text{ in } \Omega := (0,1) \\ u(0) &= 0, \leftarrow \text{ essential (Dirichlet) condition} \\ u'(1) &= \alpha, \leftarrow \text{ natural (Neumann) condition,} \end{aligned} \tag{4.1}$$

where α is a number. The corresponding *weak* formulation for (4.1) is: find u in $\mathbb{V} := \{v \in H^1(\Omega) : v(0) = 0\}$ such that

$$\int_0^1 u'v' = \alpha v(1) - \int_0^1 v, \text{ for all } v \text{ in } \mathbb{V}. \tag{4.2}$$

Remark 4.1. *Notice in* (4.2) *that we dropped the s and ds notation. This is done to simplify notation. The context will always make it clear.*

In order to compute a solution to (4.1) using the finite element method, we first need to compute the matrices $\int_0^1 u'v'$ and $-\int_0^1 v$ (see [2, 3, 4, 5, 7] for more details). The next section shows how to generate code to do this automatically for any given input mesh. This may seem like overkill for such a simple problem, but this is just for illustration. Once you master the basics, it fairly easy to solve more complicated problems.

4.2.1 Writing The Input File

Here is a sample file `MatAssem_Laplace_1D.m` that is explained in detail in the subsequent sections. It can be found in the `..\FELICITY\Demo\Laplace_1D` sub-directory of **FELICITY**. To run this demo, execute `test_Laplace_1D` at the MATLAB prompt.

```
function MATS = MatAssem_Laplace_1D()

% define domain (1-D)
Omega = Domain('interval');

% define finite element spaces
Scalar_P1    = Element(Omega,lagrange_deg1_dim1,1);

% define functions on FE space
v  = Test(Scalar_P1);
u  = Trial(Scalar_P1);

% define FE matrices
Mass_Matrix = Bilinear(Scalar_P1,Scalar_P1);
I1 = Integral(Omega,v.val * u.val);
Mass_Matrix = Mass_Matrix.Add_Integral(I1);

Stiff_Matrix = Bilinear(Scalar_P1,Scalar_P1);
I2 = Integral(Omega,v.grad' * u.grad);
Stiff_Matrix = Stiff_Matrix + I2;

RHS = Linear(Scalar_P1);
RHS = RHS + Integral(Omega,-1.0 * v.val);

% set the minimum order of accuracy for the quad rule
Quadrature_Order = 10;
% define geometry representation - Domain, (default to piecewise linear)
G1 = GeoElement(Omega);
% define a set of matrices
MATS = Matrices(Quadrature_Order,G1);

% collect all of the matrices together
MATS = MATS.Append_Matrix(Mass_Matrix);
```

```
MATS = MATS.Append_Matrix(Stiff_Matrix);
MATS = MATS.Append_Matrix(RHS);
```

Note that anything *after* a percent sign `%` is considered a comment.

Initial Items

The first line defines this as a MATLAB function:

```
function MATS = MatAssem_Laplace_1D()
```

where `MATS` is a MATLAB object of class `Matrices` (discussed later in this section).

Next, we look at:

```
% define domain (1-D)
Omega = Domain('interval');
```

This line specifies a MATLAB class object `Domain`. The argument of `Domain` refers to a *reference domain* used to "triangulate" the domain. In this case, `'interval'` (a keyword) indicates that the domain `Omega` is a mesh of 1-D segments, where each segment has the unit interval $[0, 1]$ as its reference domain.

Finite Element Space

Defining finite element spaces is achieved by:

```
% define finite element spaces
Scalar_P1    = Element(Omega,lagrange_deg1_dim1,1);
```

The class `Element` is used to specify the finite element space. `Element` is defined through its 3 arguments, which are (in order)

- name of the domain on which the element is defined,
- name of the m-file that defines the element (see Chapter 3),
- the number of tensor components (see Remark 4.2).

In this case, we have defined a scalar valued piecewise linear finite element space on the domain `Omega`. This is recorded by the object `Scalar_P1`. The mathematical notation for this is

$$\mathbb{V} := \{v \in H^1(\Omega) : v|_E \in \mathcal{P}_1(E), v \in C^0(\Omega)\}, \tag{4.3}$$

where $\mathbb{V}$ is the space, Ω is the domain, E is an edge segment contained in Ω, and $\mathcal{P}_1(E)$ is the space of linear functions on E.

Finite Element Functions

Defining functions on finite element spaces is straightforward:

```
% define functions on finite element space
v  = Test(Scalar_P1);
u  = Trial(Scalar_P1);
```

The classes here are `Test` and `Trial`; this indicates whether the function you are defining is a *test* or *trial* function. The argument of `Test` and `Trial` is a previously defined finite element space. In this case, `v` (a MATLAB object) is a *test* function in the finite element space `Scalar_P1` (defined earlier). `u` (also a MATLAB object) is a *trial* function in `Scalar_P1`. Clearly, `v,u` are functions in $\mathbb{V}$ (recall (4.3)).

Bilinear and Linear Forms (Finite Element Matrices)

Defining finite element matrices is done in the following way:

```
% define finite element matrices
Mass_Matrix = Bilinear(Scalar_P1,Scalar_P1);
I1 = Integral(Omega,v.val * u.val);
Mass_Matrix = Mass_Matrix.Add_Integral(I1);

Stiff_Matrix = Bilinear(Scalar_P1,Scalar_P1);
I2 = Integral(Omega,v.grad' * u.grad);
Stiff_Matrix = Stiff_Matrix + I2;

RHS = Linear(Scalar_P1);
RHS = RHS + Integral(Omega,-1.0 * v.val);
```

The class `Bilinear` indicates that you are specifying a *Bilinear Form*. The arguments of `Bilinear` are the test and trial *spaces*, respectively. In this case, the first matrix is an object with name `Mass_Matrix` and is defined on `Scalar_P1` $\times$ `Scalar_P1` (see Section 4.2.1).

The next line defines an integral, `I1`:

```
I1 = Integral(Omega,v.val * u.val);
```

The class `Integral` requires two arguments. The first argument is the domain on which to compute the integral. The domain could be a subset of `Omega` of codimension ≥ 0 (this is particularly useful in higher dimensional problems). The second argument is a symbolic representation of the integrand. The mathematical notation for the integral `I1` is

$$\int_{\Omega} v(x)u(x)\,dx, \qquad \text{where } \Omega \equiv \texttt{Omega}.$$

The objects `v,u` have various methods that can be called. For example, `u.val` calls the method `val` that outputs a MATLAB symbolic variable representing the direct evaluation of `u` (i.e. $u(x)$ with no derivatives). In this case, it looks like

$$\texttt{u.val} = \texttt{u\_v1\_t1},$$

which is just a symbolic label that allows **FELICITY** to keep track of this particular basis function evaluation. The notation `u_v1_t1` indicates that we are referring to *tensor* component 1 (`t1`) of *vector* component 1 (`v1`) of the function `u`. Recall that `u` was defined as a trial function on a scalar finite element space (i.e. it is a scalar-valued function with only one tensor component).

Remark 4.2. *We make a distinction here between* vector *and* tensor *components. Recall the* `Element` *class that allows you to define a finite element space with multiple tensor components. This means we take repeated tensor/cartesian products of the same finite element space; this is one way to represent vector-valued functions (i.e. as a rank-1 tensor [6, 11, 12]).*

However, one can have finite element spaces that are intrinsically vector-valued, such as Raviart-Thomas (RT) or Brezzi-Douglas-Marini (BDM) elements, i.e. each irreducible basis function is vector-valued. Thus, we need a distinction between "intrinsic vectors" and tensor products. This is mainly due to Degree-of-Freedom (DoF) ordering (see Section 3.3).

The final part of the definition of `Mass_Matrix` is to *add* the integral `I1`, i.e.

```
Mass_Matrix = Mass_Matrix.Add_Integral(I1);
```

Note that the integrand of `I1` must only involve test and trial functions from the correct finite element spaces; however, it may also include coefficient functions (see Section 4.5.3). If this is violated, then **FELICITY** will issue an error message when you try to compile.

Remark 4.3. ***FELICITY*** *allows you to add* multiple *integrals when defining forms. For example, you could have*

$$a(v,u) = \int_\Omega vu + \int_{\partial\Omega} vu,$$

and the corresponding ***FELICITY*** *code would be*

```
I1 = Integral(Omega,v.val * u.val);
Mass_Matrix = Mass_Matrix.Add_Integral(I1);
I2 = Integral(d_Omega,v.val * u.val);
Mass_Matrix = Mass_Matrix.Add_Integral(I2);
```

where `d_Omega` *is a subset of* `Omega` *that represents its boundary.*

The next matrix is `Stiff_Matrix` and is also a Bilinear form on `Scalar_P1` $\times$ `Scalar_P1`. The associated integral `I2` is `I2 = Integral(Omega,v.grad' * u.grad);`. Here, we use the `grad` method. This outputs a MATLAB symbolic variable representing the gradient of `u` (i.e. $(\nabla u)(x)$). In this case, it looks like

$$\texttt{u.grad} = \texttt{u\_v1\_t1\_grad1}.$$

This indicates evaluation of the gradient of the trial function `u`. Note that the gradient only has one component because this example is only in 1-D. For higher dimensions, the output of `u.grad` is a vector-valued symbolic variable. The mathematical notation for the integral `I2` is

$$\int_\Omega \nabla v(x) \cdot \nabla u(x)\, dx \equiv \int_\Omega v'(x) \cdot u'(x)\, dx.$$

To complete the definition of `Stiff_Matrix`, we add the integral:

```
Stiff_Matrix = Stiff_Matrix + I2;
```

where we have used an alternative syntax that is more efficient to write.

The last matrix `RHS` is a *Linear Form*, i.e. `RHS = Linear(Scalar_P1);` (the "right-hand-side" data). The syntax is similar as before, except it only takes one argument to create it, i.e. `RHS` is defined on `Scalar_P1`. Just as before, we define an integral

```
Integral(Omega,-1.0 * v.val);
```

which, in mathematical notation, is

$$\int_\Omega -v(x)\,dx.$$

But we use another alternative syntax when adding the integral:

```
RHS = RHS + Integral(Omega,-1.0 * v.val);
```

Here, we do not introduce an intermediate variable (the particular syntax you use is not important). Note: for a Linear form, the integral may only include *test* functions and coefficient functions.

The equivalent mathematical notation for the finite element matrices is given by

$$M(v,u) = \int_\Omega v(s)u(s)ds, \quad K(v,u) = \int_\Omega v'(s)u'(s)ds, \quad L(v) = \int_\Omega -v(s)ds, \tag{4.4}$$

where v, u are in $\mathbb{V}$, M is the mass matrix, K is the stiffness matrix, and L is the right-hand-side.

Set Quadrature Order

Set the (minimum) quadrature order to use in evaluating the integrals during matrix assembly:

```
% set the minimum order of accuracy for the quad rule
Quadrature_Order = 10;
```

This is used later (see below).

Geometry Representation

Next, we specify how the geometry is represented:

```
% define geometry representation - Domain, (default to piecewise linear)
G1 = GeoElement(Omega);
```

This indicates how the local geometry is discretized, using the MATLAB class `GeoElement`. One advantage of **FELICITY** is that you can use any implemented finite element type to describe the local geometry of each element. In this case, you would add a second argument to `GeoElement`, such as `lagrange_deg2_dim1`, which is the element definition for piecewise quadratic functions. By omitting the argument, as is done here, indicates that the domain geometry is locally piecewise linear (the default standard case); furthermore, the topological and geometric dimensions are inherited from the Domain `Omega` (in this case, both are 1).

Collect All Matrices

First, create a MATLAB object of class `Matrices`:

```
% define a set of matrices
MATS = Matrices(Quadrature_Order,G1);
```

Then include all of the previously defined matrices:

```
% collect all of the matrices together
MATS = MATS.Append_Matrix(Mass_Matrix);
MATS = MATS.Append_Matrix(Stiff_Matrix);
MATS = MATS.Append_Matrix(RHS);
```

This combines all of the information needed by **FELICITY** to automatically generate code that computes the matrices defined above.

4.2.2 Generate Executable (MEX) to Assemble Matrices

If **FELICITY** is installed (see Section 1.1), you can execute the following command at the MATLAB prompt:

```
Main_Dir = 'C:\Your_Favorite_Directory\';
[status, Path_To_Mex] =...
    Convert_Mscript_to_MEX(Main_Dir,'MatAssem_Laplace_1D','Assemble_1D');
```

Note that `MatAssem_Laplace_1D.m` should be in the directory specified by `Main_Dir`. The last argument specifies what the executable will be called, and it will be placed in `Main_Dir`. Make sure that `Main_Dir` is in your MATLAB path.

4.2.3 Solving The PDE

First, create the domain (i.e. the mesh). At the MATLAB prompt (or in a script) type the following commands:

```
Num_Vertices   = 101;
Indices        = (1:1:Num_Vertices)';
% you must use unsigned integer
Omega_Mesh     = uint32([Indices(1:end-1,1), Indices(2:end,1)]);
Omega_Vertices = linspace(0,1,Num_Vertices)';
```

This will define a mesh data structure for the 100 uniform edge segments in the mesh of the interval $[0, 1]$. Next, specify the Degree-of-Freedom map (DoFmap) for the finite element space used to represent the solution. Since we are using continuous piecewise linear "hat" functions, we can do the following:

```
P1_DoFmap = Omega_Mesh;
```

Now use the executable that was generated before. Run the following at the MATLAB prompt (or insert after the above code in a script):

```
FEM = Assemble_1D([],Omega_Vertices,Omega_Mesh,[],[],P1_DoFmap);
```

Recall that we named the MEX file `Assemble_1D`. In general, if you try to run the MEX file with the incorrect number of arguments, it will give you an error message but it will also *tell you* what the inputs should be. The output is a MATLAB `struct` of the following form:

```
FEM(1).Type : 'Mass_Matrix'
FEM(1).MAT  : sparse matrix representing the
              discrete bilinear form for the mass matrix

FEM(2).Type : 'RHS'
FEM(2).MAT  : a column vector representing the
              discrete linear form for the right-hand-side data

FEM(3).Type : 'Stiff_Matrix'
FEM(3).MAT  : sparse matrix representing the
              discrete bilinear form for the stiffness matrix
```

Note that the `Type` field contains the same name that was used in the `MatAssem_Laplace_1D.m` file. Also note that the matrices are *output in alphabetical order*.

Compute the numerical solution of (4.2) by using these commands:

```
Soln = zeros(size(FEM(3).MAT,1),1); % init
RHS  = FEM(2).MAT;
A    = FEM(3).MAT;

% add in the Neumann condition at s = 1
alpha    = 0;
RHS(end) = RHS(end) + alpha;

% use backslash to solve (impose Dirichlet condition at s=0)
Soln(2:end,1) = A(2:end,2:end) \ RHS(2:end,1);
```

Now plot the solution

```
figure;
h1 = plot(Omega_Vertices,Soln,'b-','LineWidth',2.0);
title('Solution of PDE: -d^2/ds^2 u = -1, u(0) = 0, d/ds u(1) = 0',...
                        'FontSize',14);
xlabel('x (domain = [0, 1])','FontSize',14);
ylabel('u (solution value)','FontSize',14);
set(gca,'FontSize',14);
axis equal;
```

The solution is shown in Figure 4.1.

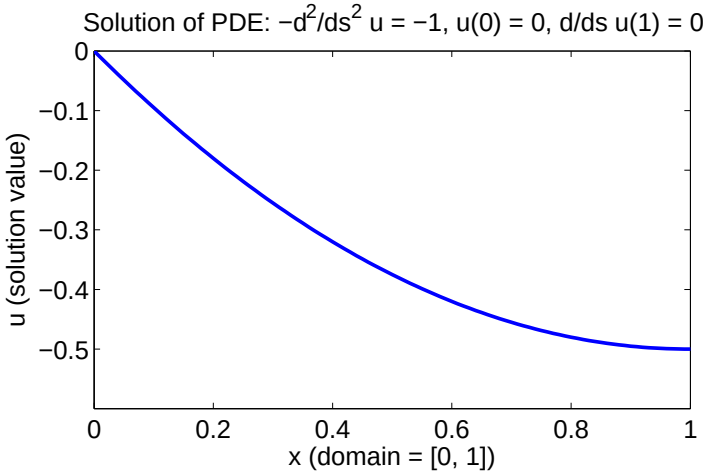

Figure 4.1: Plot of solution of (4.2).

4.2.4 Concluding Remarks

The procedure is the same for problems in 2-D and 3-D. Of course, the main difficulty will be in creating a mesh, which is a separate task. **FELICITY** does not have mesh *generation* yet, but it can manipulate and refine meshes (Chapter 2). **FELICITY** also provides tools for creating DoF maps for any implemented finite element space on a given input mesh (Chapter 5). And imposing boundary conditions requires finding the associated DoF nodes. See the Google-Code page (Section 1.2) for more tutorials.

4.3 Some Details of Transforming an Input .m File Into a MEX File

A sample MATLAB script (`test_Laplace_1D.m`) will convert `MatAssem_Laplace_1D.m` into a stand-alone `C++` code with MEX interface, compile it, and execute it on a sample problem (i.e. it will do what is described in Sections 4.2.2 and 4.2.3). This MATLAB script is located in the `..\FELICITY\Demo\Laplace_1D` sub-directory of **FELICITY**. That script file calls two other files. The first is called `Convert_Mscript_to_MEX.m` (presented below). The second is `Execute_Laplace_1D.m` (located in the same directory) and is straightforward to understand (i.e. it implements what is described in Section 4.2.3).

The conversion routine (m-file) is given here:

```
function [status, Path_To_Mex] =...
                  Convert_Mscript_to_MEX(Main_Dir,Mscript,MEX_FileName)
%Convert_Mscript_to_MEX
%
%   Generic compile procedure for generating MEX files that compute finite
```

```
%   element matrices using FELICITY.  NO changes are necessary here.

FI                = FELInterface(Main_Dir); % do NOT change
Snippet_SubDir    = 'Scratch_Dir'; % no change necessary
GenCode_SubDir    = 'Assembly_Code_AutoGen'; % no change necessary
MEX_Dir           = Main_Dir; % where the MEX-file goes

%%%%%%%% do NOT change %%%%%%%%
[status, Path_To_Mex] =...
FI.Compile_Matrix_Assembly(Mscript,Snippet_SubDir,GenCode_SubDir,...
                           MEX_Dir,MEX_FileName);

end
```

4.3.1 FELInterface MATLAB Class

This part of the code:

```
FI           = FELInterface(Main_Dir);
```

defines a simple object that interfaces to the **FELICITY** code generation capabilities. You then generate *and* compile the matrix assembly code by running:

```
[status, Path_To_Mex] =...
FI.Compile_Matrix_Assembly(Mscript,Snippet_SubDir,GenCode_SubDir,...
                           MEX_Dir,MEX_FileName);
```

The inputs here are filenames and directories (see next section).

Note: make sure you have a `C++` compiler that MATLAB can use with its MEX compiler. I have used Microsoft's Visual `C++` compiler. It is easy to configure MATLAB to use a compatible C++ once it is installed (just type `mex -setup` at the MATLAB prompt). Also, make sure your MATLAB path is set correctly to find **FELICITY** and its sub-directories.

4.3.2 Directories

- `Mscript` is the name of the MATLAB script to run (i.e. the one that defines the finite element matrices).

- `Snippet_SubDir` specifies a sub-directory (under `Main_Dir`) where **FELICITY** can store auto-generated pieces of `C++` code. This is a scratch work directory for **FELICITY** and can be deleted afterwords.

- `GenCode_SubDir` specifies where **FELICITY** should place the auto-generated matrix assembly code. The generated code is a stand-alone `C++` code interfaced to MATLAB through a MEX interface. You are free to modify this code directly at your own risk. You can also delete it after the MEX file is generated.

- `MEX_Dir` specifies where to place the compiled MATLAB MEX file. Make sure this directory is in the MATLAB path so that you can run it from MATLAB.
- `MEX_FileName` defines the filename of the MEX executable.

4.4 How To Call The MEX File

First, try to execute the MEX file in MATLAB without any arguments. It will give you an error message, but it will also tell you what the inputs should be for your specialized matrix assembly application.

FELICITY provides some mesh tools (Chapter 2), but the initial mesh *creation* must be done by an external source. Eventually, **FELICITY** will include meshing tools. You must also provide the local-to-global Degree-of-Freedom (DoF) map for the finite element spaces (Chapter 5). At this stage of **FELICITY**, you must still piece these things together to solve a real PDE problem by the FEM (see the Google-Code page for more examples). Future releases of **FELICITY** will further automate this as well.

To see how to call the MEX file, look at the M-file `Execute_Laplace_1D.m` (in the same directory as `MatAssem_Laplace_1D.m`) or look at Section 4.2.3.

4.5 Advanced Features

What are the limits of the MATLAB script file? What exactly can you put in the integrand? This section describes the various options.

4.5.1 Operations on Finite Element Functions

Suppose `v` is either a `Test`, `Trial`, or `Coef` (coefficient function), and let `k,j` be indices. Then one can apply the following operators:

- `v.val(k)` gives the value of the `kth` component of `v`.
- `v.grad(k,j)` gives the `jth` component of the gradient of the `kth` component of `v`.
- `v.ds` gives the derivative of `v` with respect to arc-length; only useable when the topological dimension is 1.

Note: all of the above operations output symbolic variable representations (see MATLAB's Symbolic Toolbox).

Eventually, **FELICITY** will have the Hessian operator implemented also.

4.5.2 Operations on Geometric Quantities

FELICITY as been developed with applications involving geometric quantities in mind (see Section 4.6 for examples). This can be accessed by defining a MATLAB object of class `GeoFunc`:

```
% define geometric function on 'Sigma' domain
gf = GeoFunc(Sigma);
```

You can then call various methods to output symbolic variables that represent a geometric quantity. For instance, `gf.X` is a symbolic variable that represents the coordinate map (i.e. parameterization).

Let `q` be the topological dimension, and let `i` be an index between 1 and `q`. Likewise, let `d` be the ambient geometric dimension, and let `k,j` be an index between 1 and `d`. You can apply the following operators to the parameterization:

- `gf.X(k)` gives the value of the `kth` coordinate, i.e. the identity map. This is especially useful for, say, a 2-D surface in 3-D.

- `gf.deriv_X(k,i)` gives the derivative of the `kth` component of the map with respect to the `ith` *local* coordinate.

- `gf.Tangent_Space_Proj(k,j)` gives the `(k,j)` component of the "tangent space projection" matrix. This (symmetric) matrix maps `d` dimensional vectors onto the tangent space of the domain. Note: this is only useful if the domain is a manifold embedded in higher dimensions.

We also have the following geometric quantities at our disposal:

- `gf.N(k)` gives the value of the `kth` component of the normal vector of the geometric element; only useful when the geometric dimension is 1 higher than the topological dimension.

- `gf.T(k)` gives the value of the `kth` component of the tangent vector of the geometric element; only useful for topological dimension 1 and higher geometric dimensions.

- `gf.VecKappa(k)` gives the value of the `kth` component of the (total) curvature vector of the geometric element; only useful if the geometry is represented by higher order elements (like piecewise quadratics).

- `gf.Kappa` gives the value of the signed (total) curvature of the geometric element; only useful if the geometry is represented by higher order elements (like piecewise quadratics).

- `gf.Kappa_Gauss` gives the value of the Gaussian curvature of the geometric element; only useful if the geometry is a 2-D surface in 3-D and is represented by higher order elements (like piecewise quadratics).

FELICITY also includes the mesh size as a function, i.e. `gf.Mesh_Size` gives the local mesh size of the geometric element (i.e. it is constant on each element). This is defined to be the longest edge of the local element/simplex.

4.5.3 Defining Coefficient Functions

Sometimes it can be useful to have extra finite element functions in the definition of a discrete bilinear (or linear) form. These could be weighting factors or they may come from linearizing a non-linear problem about a given solution. This is done by using the following syntax:

```
my_f     = Coef(Vector_P1);
old_soln = Coef(Scalar_P2);
```

where `my_f` is a finite element function in the space `Vector_P1` and `old_soln` is a finite element function in the space `Scalar_P2`. The keyword `Coefficient` means that these functions can be arbitrarily set and used in evaluating an integrand. See the tutorials on the wiki of the Google-Code page.

4.5.4 Defining Useful Constants

Defining constants is done by standard MATLAB syntax.

```
% define some constants
C1 = pi;
C2 = exp(C1);
```

You can then use `C1`, `C2` when defining the integrals.

4.5.5 Including Simple C-Code

Sometimes the integrands of bilinear (or linear) forms may involve complicated problem-specific functions. This can be accomplished by the following syntax to include extra `C` code:

```
% example integrand
Body_Force_Matrix = Linear(v);
ffoo = sym('foo(F)'); % symbolic variable
ffoo = subs(ffoo,'F', C1 * gf.X(3) );
Body_Force_Matrix = Body_Force_Matrix.Integral(Sigma,v.val * ffoo);

.......

MATS = Matrices(Quadrature_Order,G1);

% include the definition of "foo" (see "Body_Force_Matrix")
MATS = MATS.Include_C_Code('my_foo_code.c');
```

This causes **FELICITY** to include the file `my_foo_code.c` in the generated code.

If you only indicate the file name, then it should reside in the same directory as the MATLAB script. Otherwise, you need to specify the entire path. Moreover, the file should only contain simple `C` functions that are used in evaluating the integrands. For example, here `my_foo_code.c` contains the definition of a `C` function named `foo` that accepts a single argument (`const double&` data type) and returns an argument of type `double`. See Section 4.6.1 for an example that uses this.

4.6 In-Depth Examples

Here, we will use some of the advanced features introduced in the previous section. All of these examples are in the `..\FELICITY\Code_Generation\Matrix_Assembly\Unit_Test` sub-directory.

In fact, you can run all of these examples by executing `test_matrix_assembly.m` in the **FELICITY** subdirectory `..\FELICITY\Code_Generation\Matrix_Assembly\`.

4.6.1 Topological Dimension 1, Geometric Dimension 3

See the file `..\Unit_Test\Dim_1\MatAssem_interval.m`.

The mathematical notation for the finite element matrices implemented by `MatAssem_interval.m` is

$$\begin{aligned}
\texttt{Mass_Matrix} &= \int_\Sigma v(s)u(s)ds, \quad \texttt{Stiff_Matrix} = \int_\Sigma \partial_s v(s)\partial_s u(s)ds,\\
\texttt{Body_Force_Matrix} &= \int_\Sigma v(s)\sin(\pi z(s))ds,\\
\texttt{Vector_Mass_Matrix} &= \int_\Sigma \mathbf{v}(s)\cdot\mathbf{u}(s)ds,\\
\texttt{Weighted_Mass_Matrix} &= \int_\Sigma \partial_s\mathbf{f}_1(s)v(s)u(s)ds,\\
\texttt{Tangent_Matrix} &= \int_\Sigma \mathbf{v}(s)\cdot\boldsymbol{\tau}(s)ds,\\
\texttt{Curv_Matrix} &= \int_\Sigma v(s)\kappa(s)ds,\\
\texttt{L2_Norm_Sq_Error} &= \int_\Sigma \{\exp(x(s)) - g_{\text{old}}(s)\}^2 ds,
\end{aligned} \tag{4.5}$$

where Σ is a helix, $\mathbf{f}$ refers to `my_f` (a vector function), g_{old} refers to `old_soln`, and $\boldsymbol{\tau}$ is the oriented unit tangent vector. Note that `L2_Norm_Sq_Error` is just a 1x1 matrix. Also note that `Body_Force_Matrix`, `Tangent_Matrix`, and `Curv_Matrix` are *linear forms* (not bilinear forms); thus they are actually just finite element column vectors. See the `.m` file for more information. Also, see the unit test files for more details.

Matrices Involving Tensor-Valued Variables

The matrix structure of `Vector_Mass_Matrix` looks like:

$$\texttt{Vector\_Mass\_Matrix} = \begin{bmatrix} \texttt{M} & 0 & 0 \\ 0 & \texttt{M} & 0 \\ 0 & 0 & \texttt{M} \end{bmatrix}, \tag{4.6}$$

where `M` is a *scalar* mass matrix. This is because the test and trial functions are tensor-valued functions. The ordering of the tensor components is done in the obvious way. See Section 6.4.1 for more information.

4.6.2 Topological Dimension 2, Geometric Dimension 3

See the file `..\Unit_Test\Dim_2\MatAssem_triangle.m`.

The mathematical notation for the finite element matrices implemented by `MatAssem_triangle.m` is

$$\begin{aligned}
\texttt{Mass_Matrix} &= \int_\Gamma v(\mathbf{x})u(\mathbf{x})dS(\mathbf{x}), \quad \texttt{Stiff_Matrix} = \int_\Gamma \nabla_\Gamma v(\mathbf{x}) \cdot \nabla_\Gamma u(\mathbf{x})dS(\mathbf{x}), \\
\texttt{Body_Force_Matrix} &= \int_\Gamma v(\mathbf{x}) \sin(x)\cos(y)\sin(z)dS(\mathbf{x}), \\
\texttt{Vector_Mass_Matrix} &= \int_\Gamma \mathbf{v}(\mathbf{x}) \cdot \mathbf{u}(\mathbf{x})dS(\mathbf{x}), \\
\texttt{Normal_Matrix} &= \int_\Gamma \mathbf{v}(\mathbf{x}) \cdot \boldsymbol{\nu}(\mathbf{x})dS(\mathbf{x}), \\
\texttt{Curv_Matrix} &= \int_\Gamma v(\mathbf{x})\kappa(\mathbf{x})dS(\mathbf{x}), \\
\texttt{L2_Norm_Sq_Error} &= \int_\Gamma \{\exp(x) - g_{\text{old}}(\mathbf{x})\}^2 dS(\mathbf{x}),
\end{aligned} \tag{4.7}$$

where Γ is the surface of a unit sphere, g_{old} refers to `old_soln`, $\mathbf{x} = (x, y, z)$, and $\boldsymbol{\nu}$ is the outward unit normal vector. Note that ∇_Γ is the *surface gradient* on Γ and $\nabla_\Gamma v$ is a vector of length 3 because the geometric dimension is 3. See the `.m` file for more information. Also, see the unit test files for more details.

4.6.3 Topological Dimension 3, Geometric Dimension 3

See the file `..\Unit_Test\Dim_3\MatAssem_tet.m`.

The mathematical notation for the finite element matrices implemented by `MatAssem_tet.m` is

$$\begin{aligned}
\texttt{Mass_Matrix} &= \int_\Omega v(\mathbf{x})u(\mathbf{x})d\mathbf{x}, \quad \texttt{Stiff_Matrix} = \int_\Omega \nabla v(\mathbf{x}) \cdot \nabla u(\mathbf{x})d\mathbf{x}, \\
\texttt{Body_Force_Matrix} &= \int_\Omega v(\mathbf{x}) \sin(x)\cos(y)\sin(z)d\mathbf{x}, \\
\texttt{Vector_Mass_Matrix} &= \int_\Omega \mathbf{v}(\mathbf{x}) \cdot \mathbf{u}(\mathbf{x})d\mathbf{x}, \\
\texttt{Weighted_Mass_Matrix} &= \int_\Omega [\nabla \mathbf{f}(\mathbf{x})]_{2,2}\mathbf{f}_1(\mathbf{x})v(\mathbf{x})u(\mathbf{x})d\mathbf{x}, \\
\texttt{L2_Norm_Sq_Error} &= \int_\Omega \{\exp(x) - g_{\text{old}}(\mathbf{x})\}^2 d\mathbf{x},
\end{aligned} \tag{4.8}$$

where Ω is a right circular cylinder, $\mathbf{x} = (x, y, z)$, $\mathbf{f}$ refers to `my_f` (a vector function), and g_{old} refers to `old_soln`. See the `.m` file for more information. Also, see the unit test files for more details.

4.7 Conclusion

Other examples can be found under the `Unit_Test` sub-directory of `Matrix_Assembly`, as well as the demos directory: `..\FELICITY\Demo\`.

Chapter 5

Automatically Generating Degree-of-Freedom Maps

5.1 Introduction

In order to assemble any finite element matrix, you must provide the local-to-global Degree-of-Freedom (DoF) map for each element in the mesh. **FELICITY** assumes an $N \times L$ MATLAB matrix to represent this:

$$\texttt{DoFmap} = \begin{bmatrix} c_{1,1} & \cdots & c_{1,L} \\ c_{2,1} & \cdots & c_{2,L} \\ \vdots & \vdots & \vdots \\ c_{N,1} & \cdots & c_{N,L} \end{bmatrix}, \tag{5.1}$$

where N is the number of elements in the mesh, L is the number of basis functions on each element, and $c_{i,j}$ is the global DoF index for local basis function j on element i.

For a complicated element, it can be tedious to implement DoF allocation on a general mesh. However, **FELICITY** can do this automatically by using the element definition file described in Chapter 3. The following tutorial gives a walk-through on how to do this.

5.2 Tutorial

This is a tutorial that you can type in sequentially at the MATLAB prompt. Or you can view the demo in this directory:
`..\FELICITY\Demo\DoFmap_Generation\Dim_2`

5.2.1 Generate Executable

If **FELICITY** is installed (see Section 1.1), you can execute the following command at the MATLAB prompt:

```
% define finite elements
Elem    = lagrange_deg1_dim2(); % P1 space
```

```
Elem(2) = lagrange_deg2_dim2(); % P2 space

Main_Dir = 'C:\Your_Favorite_Directory\';
[status, Path_To_Mex] =...
         FEL_Compile_DoF_Allocate(Main_Dir,'mexDoF_Example_2D',Elem);
```

You must declare the finite elements for which to generate the DoF allocation code. The files `lagrange_deg1_dim2` and `lagrange_deg2_dim2` are in this directory:
`..\FELICITY\Elem_Defn`

The second argument specifies what the executable will be called, and it will be placed in `Main_Dir`. Make sure that `Main_Dir` is in your MATLAB path.

5.2.2 Using The Executable

First, load up a sample triangulation. At the MATLAB prompt (or in a script) type the following commands:

```
% load up the 'standard' triangulation
[Vtx, Tri] = Standard_Triangle_Mesh_Test_Data();
```

`Standard_Triangle_Mesh_Test_Data` is an m-file provided by **FELICITY** that defines a simple triangulation of the square $[0, 2] \times [0, 2]$. This example refrains from using the mesh classes that **FELICITY** provides because each tutorial (and chapter) should be as self-contained as possible. Next, we plot the mesh:

```
figure;
p1 = trimesh(Tri,Vtx(:,1),Vtx(:,2),0*Vtx(:,2));
view(2);
axis equal;
shading interp;
set(p1,'edgecolor','k'); % make mesh black
title('Sample Mesh for Automatic DoF Allocation Test');
```

Now run the generated executable from Section 5.2.1:

```
% allocate DoFs
[P1_DoFmap, P2_DoFmap] = mexDoF_Example_2D(uint32(Tri));
```

That's it. You can now look at the output arrays in MATLAB or feed them to a matrix assembler (Chapter 4). Note that the input arguments must be `uint32`.

5.3 Further Notes

In general, the way you call the generated executable depends on how many finite elements you told it to allocate for. If you try to call the executable with the wrong number of input or output arguments, **FELICITY** will give you an error message but it will also *tell you* what the inputs and outputs should be.

But the generated MEX file only requires **one** input argument, which depends on the topological dimension of the mesh:

- 1-D: edge connectivity data.
- 2-D: triangle connectivity data.
- 3-D: tetrahedron connectivity data.

Note: the input must be `uint32`.

Remark 5.1 (Recent Change)**.** *The generated DoF allocation code* ***only requires*** *the mesh connectivity. You do* ***not*** *need to pass in the global edges (in 2-D) or the tetrahedral neighbors in (3-D).*

Chapter 6

Helper Routines For Finite Element Codes

6.1 Introduction

Writing a finite element code involves the following fundamental tasks.

- Creating a mesh of the problem domain with annotation data specifying any sub-domains.
- Defining finite element spaces on sub-domains of the mesh and allocating Degrees-of-Freedom (DoFs).
- Computing and assembling discrete matrix versions of bilinear and linear forms.
- Solving the resulting linear system, possibly within a loop for non-linear problems.

These tasks were covered in the previous chapters. Any other tasks require more-or-less straightforward manipulations of the numerical data, such as:

- Manipulating finite element matrices to enforce boundary conditions, concatenate with other matrices, couple with other discrete problems, etc.
- Managing DoFs with respect to mesh sub-domains or dirichlet data, such as getting the subset of DoFs that lie on a sub-domain.
- Plotting solution data.

Implementing these other tasks is fairly easy in MATLAB, especially since most of them can be made efficient by vectorization. You are free to do this yourself with your own MATLAB scripts. However, some of these procedures can become rather repetitive and tedious, such as when finding DoFs attached to sub-domains.

The following sections describe some built-in **FELICITY** classes and routines that can help automate some of these basic tasks, as well as some other classes that provide convenience routines.

6.2 Read Out Reference Finite Element Information

Chapter 3 describes how **FELICITY** defines a finite element on a reference domain. But **FELICITY** also provides a MATLAB class that can "pretty print" information about the element, like the basis function definitions and the location of the nodal degrees of freedom.

6.2.1 The FELOutputElemInfo Class

This is best described by an example using the degree 2 Lagrange finite element space on a triangle. Type the following at the MATLAB prompt.

```
Elem_Lag = FELOutputElemInfo(lagrange_deg2_dim2);

% output LaTeX code for the basis function definitions
OUT_STR = Elem_Lag.Print_Basis_Functions('latex');

% pretty print the basis function definitions to the MATLAB display
Elem_Lag.Print_Basis_Functions;

% pretty print the location of the nodal degrees-of-freedom
%        to the MATLAB display and a MATLAB figure
Elem_Lag.Print_DoFs;
```

If you want to look at another element, just change the argument of `FELOutputElemInfo` to be one of the filenames contained in `.\FELICITY\Elem_Defn\`.

6.3 Access to Finite Element Matrices

In chapter 4, we learned how to automatically generate code for assembling finite element matrices. The output of the MEX-file is a `FEM` structure array (see Section 4.2.3). The names of the matrices are embedded in the `FEM` variable. But it is a little annoying to sift through the array looking at the matrix names to find the one you need. So, there is a class to do that for you.

6.3.1 The FEMatrixAccessor Class

This is best described by an example. Type the following at the MATLAB prompt.

```
% we are assuming that FEM has already been created from
%    a FELICITY generated matrix assembly code

FEM_Data = FEMatrixAccessor('my test',FEM);
MASS  = FEM_Data.Get_Matrix('Mass_Matrix');
```

The name that is passed to `Get_Matrix` is the same name that was used in the input M-file (see Chapter 4). `Get_Matrix` returns the sparse matrix data corresponding to the

name that was given. Note: this does not duplicate the data because MATLAB uses the copy-on-write convention. In other words, `MASS` really just points to the data that is held in the `FEM_Data` object. If you modify any entries of `MASS`, *then* MATLAB will copy the data and overwrite the entries accordingly in the new copy.

6.4 Manipulate Degrees-of-Freedom

Dealing with boundary conditions, and their associated DoFs, is a typical task that can be very annoying. You must figure out which DoFs are "attached" to the boundary and possibly their physical coordinates (so you can interpolate a function to set the boundary condition). This is especially irritating if you are using a higher order method where the DoFs are arranged in a non-trivial way.

For low order methods, say piecewise linear hat functions, you can get by with your own custom subroutines. But even in this case, it can be tedious if the finite element space is tensor valued, i.e. defined as a cartesian product of scalar valued spaces. For example, you may have *tensor valued*, continuous piecewise quadratic polynomials defined over a triangular mesh; for instance, this is used in the Taylor-Hood element for the Stokes equations. Knowing the DoFs associated with each tensor component is crucial for writing a finite element code.

6.4.1 Some Conventions

Suppose we have a simple mesh of a square domain with a piecewise quadratic finite element space $\mathbb{V}$ defined over it (see Figure 6.1). There are six local basis functions on each triangle, so there are six indices associated with each triangle:

T_1:	1	2	3	8	6	5
T_2:	1	3	4	9	7	6

(6.1)

The table in (6.1) is the so-called Degree-of-Freedom map (DoFmap) for the finite element space $\mathbb{V}$ defined on the mesh in Figure 6.1. In this case, the total number of *global* DoFs is nine. Hence, a global basis for $\mathbb{V}$ would look like

$$\mathbb{V} = \{\phi_1, \phi_2, ..., \phi_9\}, \tag{6.2}$$

such that

$$\phi_i(x_j) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}, \tag{6.3}$$

where x_j are the coordinates of node j.

DoFmaps are stored as matrices in MATLAB. Each row of the DoFmap corresponds to a mesh element (e.g. triangle), and each column corresponds to the *local* DoFs for that element. For example, column #2 of the DoFmap gives the *global* DoF indices for *local* basis function #2 of all the elements.

The ordering of the local DoFs is linked to the reference domain and depends on how the local basis functions are ordered (see Chapter 3). On the other hand, the global indices are just bookkeeping; each global index refers to a unique basis function in (6.2). One

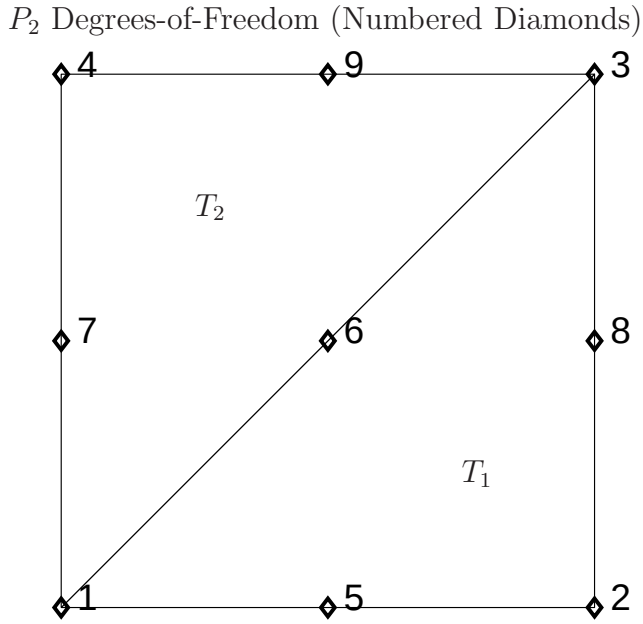

Figure 6.1: Illustration of finite element space Degrees-of-Freedom (DoFs). The domain is a square and is meshed by two triangles (denoted T_1, T_2). The finite element space $\mathbb{V}$ is the set of globally continuous piecewise quadratic polynomials defined over each triangle. Each basis function of $\mathbb{V}$ corresponds directly to one of the numbered diamonds. Since $\mathbb{V}$ is globally continuous, both triangles T_1, T_2 share the *same* DoFs $\{1, 3, 6\}$ along their common edge.

could permute the numbering in Figure 6.1 as long as the indices are fixed throughout the simulation. See these references [1, 7] for more information.

When applying boundary conditions, such as Dirichlet conditions, one must identify which DoFs are attached to the boundary. Then the system matrix can be modified accordingly to enforce the conditions. Thus, one must "sift through" the DoFs to identify which ones belong to the boundary. You can either write your own code to do this, or use the facilities in **FELICITY** (Section 6.4.2).

If the finite element space is built from taking tensor products of other spaces, then the DoFmap gets a little more complicated. For example, suppose we have the vector-valued space

$$\mathbb{W} = \mathbb{V} \times \mathbb{V}. \tag{6.4}$$

Clearly, the dimension of $\mathbb{W}$ is twice that of $\mathbb{V}$, i.e. there are now *two times* as many basis functions as before. There is a choice here in what basis set to take and how to order the basis functions. A simple choice (assumed by **FELICITY**) is

$$\mathbb{W} = \left\{ \underbrace{\begin{bmatrix} \phi_1 \\ 0 \end{bmatrix}}_{\boldsymbol{\phi}_1}, \underbrace{\begin{bmatrix} \phi_2 \\ 0 \end{bmatrix}}_{\boldsymbol{\phi}_2}, ..., \underbrace{\begin{bmatrix} \phi_9 \\ 0 \end{bmatrix}}_{\boldsymbol{\phi}_9}, \underbrace{\begin{bmatrix} 0 \\ \phi_1 \end{bmatrix}}_{\boldsymbol{\phi}_{10}}, \underbrace{\begin{bmatrix} 0 \\ \phi_2 \end{bmatrix}}_{\boldsymbol{\phi}_{11}}, ..., \underbrace{\begin{bmatrix} 0 \\ \phi_9 \end{bmatrix}}_{\boldsymbol{\phi}_{18}} \right\}, \tag{6.5}$$

where each basis function is tensor-valued (i.e. a rank-1 tensor of length 2). So the first nine basis functions of $\mathbb{W}$ are basically the same as $\mathbb{V}$ (i.e. the first component of $\mathbb{W}$ is $\mathbb{V}$). Thus, the DoFmap for $\mathbb{V}$ (6.1) can also be used to reference the *first* component of $\mathbb{W}$.

But each basis function of $\mathbb{W}$ has its own unique global index. So we need another DoFmap to reference the *second* component of $\mathbb{W}$, which is given by:

$$\text{adding } \dim(\mathbb{V}) \text{ to (6.1)},$$

i.e.

T_1:	10	11	12	17	15	14
T_2:	10	12	13	18	16	15

(6.6)

Thus, (6.6) references the second component of $\mathbb{W}$, i.e. the basis functions $\boldsymbol{\phi}_{10}$, ..., $\boldsymbol{\phi}_{18}$ (note bold-faced notation).

For storage purposes, **FELICITY** only needs the DoFmap for the first component. The DoFmap for the second component can always be built on-the-fly from the first when needed. Higher order tensor spaces are handled in an analogous way.

6.4.2 The FiniteElementSpace Class

FELICITY has the class `FiniteElementSpace` that allows for access to and manipulation of the DoFmap. See the demo `test_Finite_Element_Space_2D` in the directory: `.\FELICITY\Demo\Finite_Element_Space_2D`.

Basic DoF Routines

Let's do an example. First create the mesh in Figure 6.1 by typing the following at the MATLAB prompt.

```
% define a simple square mesh
Vtx = [0 0; 1 0; 1 1; 0 1];
Tri = [1 2 3; 1 3 4];

% create a mesh object
Mesh = MeshTriangle(Tri,Vtx,'Square');
```

Now define several sub-domains of the square (see Chapter 2).

```
% define the boundary of the square
Bdy_Edge = Mesh.freeBoundary;
Mesh = Mesh.Append_Subdomain('1D','Boundary',Bdy_Edge);

% define some more subdomains
Mesh = Mesh.Append_Subdomain('0D','Bottom Left Corner',[1]);
Mesh = Mesh.Append_Subdomain('0D','Bottom Right Corner',[2]);
Mesh = Mesh.Append_Subdomain('0D','Top Right Corner',[3]);
Mesh = Mesh.Append_Subdomain('0D','Top Left Corner',[4]);
Mesh = Mesh.Append_Subdomain('1D','Bottom Side',[1 2]);
Mesh = Mesh.Append_Subdomain('1D','Top Side',[3 4]);
Mesh = Mesh.Append_Subdomain('1D','Diag',[1 3]);
Mesh = Mesh.Append_Subdomain('2D','Bottom Tri',[1]);
```

Next, load up the degree 2, Lagrange element of dimension 2 (defined on a triangle), and create a `ReferenceFiniteElement` object that is tensor-valued with 2 components:

```
% declare reference element
P2_Elem = lagrange_deg2_dim2();
RE = ReferenceFiniteElement(P2_Elem,2); % 2 components
```

Now create the `FiniteElementSpace` object for the space $\mathbb{W}$ (6.4):

```
% declare a finite element space
FES = FiniteElementSpace('P2',RE,Mesh,[]);
clear RE; % not needed anymore
```

The arguments of `FiniteElementSpace` (in order) are:

1. A name for the finite element space, e.g. `'P2'`.

2. The reference finite element for the space.

3. The mesh data for the finite element space.

4. The name of the sub-domain that the finite element space is defined on. In this case, the space is defined on the entire mesh (no sub-domain), so just pass the empty matrix `[]`. If we wanted to define the space on `'Bottom Tri'` (i.e. T_1 in Figure 6.1), we would instead pass the string `'Bottom Tri'`.

We must still set the DoFmap for this space. So we store (6.1) by the following commands:

```
% set DoFmap
DFM = [1     2     3     8     6     5;
       1     3     4     9     7     6];
FES = FES.Set_DoFmap(Mesh,DFM);
clear DFM; % not needed anymore
```

Note: if the mesh has lots of elements, then you may want to automatically generate the code for creating the DoFmap (see Chapter 5).

Now lets look at the object. We can get a summary about the finite element space by typing:

```
FES
```

We can look at the DoFmap:

```
FES.DoFmap
```

We can get a list of the DoF indices for the *first component*:

```
DoF_Indices = FES.Get_DoFs
```

And we can get the global coordinates of the DoFs (for the first component):

```
XC = FES.Get_DoF_Coord(Mesh)
```

Note that the mesh geometry is needed to get the coordinates. We can plot the mesh and DoFs from Figure 6.1 by the following commands:

```
figure;
Mesh.Plot;
AX = [-0.05 1.05 -0.05 1.05];
axis(AX);
hold on;
for ii = 1:size(XC,1)
    plot(XC(ii,1),XC(ii,2),'kd','LineWidth',1.7);
    DoFstr = [num2str(ii)];
    text(XC(ii,1)+0.03,XC(ii,2)+0.02,DoFstr,'FontSize',16);
end
text(0.75,0.25,'T1');
text(0.25,0.75,'T2');
hold off;
```

```
axis equal;
axis(AX);
axis off;
title('P_2 Degrees-of-Freedom (Numbered Diamonds)');
```

The DoFs for the second component of the space $\mathbb{W}$ can be retrieved by:

```
% get the 2nd component DoF indices
FES.Get_DoFs(2)
```

We can also get a matrix of DoF indices, where the k-th column is a list of the DoF indices for the k-th component:

```
FES.Get_DoFs('all')
```

Note: each row of this matrix corresponds to one of the diamonds in Figure 6.1, i.e. each row stores the first and second component DoF indices both of which have the *same* global position coordinates.

Restricting DoFs To Sub-domains

When sub-domains are present, you can retrieve the *subset* of DoFs that lie on a given sub-domain. For example, we can get the `Boundary` DoFs by:

```
% get list of the (1st component) DoFs on the Boundary
Bdy_DoFs = FES.Get_DoFs_On_Subdomain(Mesh,'Boundary')
```

You can also get the DoFs for all tensor components on a given sub-domain by passing an additional argument `'all'`, e.g.

```
% get list of all the (tensor component) DoFs on the Boundary
FES.Get_DoFs_On_Subdomain(Mesh,'Boundary','all')
```

Having defined `XC` in the previous section, we can get the coordinates of the DoFs on a sub-domain. For example, the coordinates of the DoFs on `Boundary` are obtained by

```
% get the coordinates of the Boundary DoFs
XC(Bdy_DoFs,:)
```

We can also store in the `FES` object which sub-domains have their DoFs fixed (i.e. for Dirichlet conditions). For example, we can fix the `Bottom Side` and `Top Side` sub-domains (defined earlier):

```
FES = FES.Append_Fixed_Subdomain(Mesh,'Bottom Side');
FES = FES.Append_Fixed_Subdomain(Mesh,'Top Side');
```

Note: you must pass the mesh because the `FiniteElementSpace` object does some internal consistency checks to make sure the mesh actually contains the sub-domain. You can now list the DoFs that are *fixed* by

```
FES.Get_Fixed_DoFs(Mesh)
FES.Get_Fixed_DoFs(Mesh,2) % if you want the fixed DoFs for component 2
```

Similarly, you can get the DoFs that are *free* by

```
FES.Get_Free_DoFs(Mesh)
```

If you want to clear the list of fixed sub-domains, just type

```
FES = FES.Set_Fixed_Subdomains(Mesh,{});
```

The argument `{}` is an empty cell array. Alternatively, you could pass a cell array of strings, where each string is a name of a sub-domain. This is another way to set the fixed sub-domains, instead of using `Append_Fixed_Subdomain`.

Remark 6.1. *To make the most of the tools in this section, it is highly recommended that you familiarize yourself with MATLAB's indexing capabilities. These are used a lot in the manipulation of DoFs.*

6.4.3 Defining Finite Element Spaces on *Sub-domains*

Finite element spaces can be *defined over sub-domains* of a global mesh. For example, you may have a piecewise linear space on a domain that is a polygonal curve *embedded* in a triangular mesh. Allocating DoFs in this case is no more difficult than what was done in the previous sections. But there is some potential for confusion here, so we shall give an example.

See the demo `test_Finite_Element_Space_On_1D_Subdomain` in the directory: `.\FELICITY\Demo\Finite_Element_Space_On_1D_Subdomain`.

Allocating DoFs On An Embedded Sub-domain

First create the mesh in Figure 6.2 by typing the following at the MATLAB prompt.

```
% define a simple square mesh
Vtx = [0 0; 1 0; 1 1; 0 1; 0.5 0.5];
Tri = [1 2 5; 2 3 5; 3 4 5; 4 1 5];

% create a mesh object
Mesh = MeshTriangle(Tri,Vtx,'Square');
```

Now define the "diagonal" sub-domain of the square (see Chapter 2).

```
% define a subdomain to be one of the diagonals of the square
Mesh = Mesh.Append_Subdomain('1D','Diag',[1 5; 5 3]);
```

Next, let $\mathbb{V}$ be the set of continuous piecewise linear basis functions over the domain `Diag` (see Figure 6.2). There are two local basis functions on each edge, so there are two indices associated with each edge. This suggests using the following DoFmap:

E_1:	1	2
E_2:	2	3

(6.7)

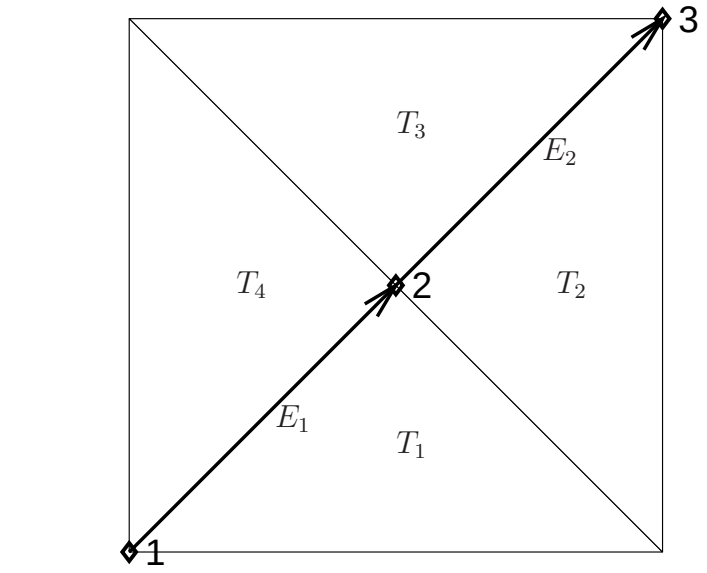

Figure 6.2: Illustration of finite element space Degrees-of-Freedom (DoFs) defined on a subdomain of a global mesh. The global mesh is the square, consisting of four triangles (denoted T_1, ..., T_4). The domain of the finite element space is the diagonal of the square which consists of two edges (denoted E_1, E_2) plotted as directed arrows. The finite element space $\mathbb{V}$ is the set of globally continuous piecewise linear polynomials defined over each edge. Each basis function of $\mathbb{V}$ corresponds directly to one of the numbered diamonds. Since $\mathbb{V}$ is globally continuous, both edges E_1, E_2 share the *same* DoF #2 on their common vertex.

Remark 6.2 (DoF Numbering)**.** *We emphasize that the DoF indices in the finite element space have* ***no relation*** *to the numbering of the vertices in the mesh. In this example, the center mesh vertex of the square has index #5, but the finite element space DoF index at the center vertex is #2.*

When defining any *DoFmap, there must not be any "gaps" in the DoF numbering in the DoFmap. For example, we could not use this:*

E_1:	1	5
E_2:	5	3

,

which is simply the connectivity of `Diag` *referenced to the global mesh vertices. If we use this, then* ***FELICITY*** *will give an error message when calling* `Set_DoFmap`*.*

In conclusion, if N is the number of distinct basis functions in the finite element space (i.e. the dimension of $\mathbb{V}$), then the DoFmap must contain all indices *from* 1 *to* N*.*

We can define this space by the following commands:

```
% declare reference element
P1_Elem = lagrange_deg1_dim1();
RE = ReferenceFiniteElement(P1_Elem,1); % 1 component

% declare a finite element space that is defined on 'Diag'
FES = FiniteElementSpace('P1',RE,Mesh,'Diag');
clear RE;
```

Note that the reference element is 1-D, because `Diag` has topological dimension 1.

Next, set the DoFmap for this space. Since the domain consists of only two edges and the space is piecewise linear, we type the following commands:

```
% set DoFmap
DFM = [1     2;
       2     3];
FES = FES.Set_DoFmap(Mesh,DFM);
clear DFM;
```

Again, if the mesh has lots of elements, then you may want to automatically generate the code for creating the DoFmap (see Chapter 5).

We can plot the mesh and DoFs from Figure 6.2 by the following commands:

```
figure;
Mesh.Plot;
AX = [-0.05 1.05 -0.05 1.05];
axis(AX);
hold on;
for ii = 1:size(XC,1)
    plot(XC(ii,1),XC(ii,2),'kd','LineWidth',1.7);
    DoFstr = [num2str(ii)];
```

```
    text(XC(ii,1)+0.03,XC(ii,2)+0.0,DoFstr,'FontSize',16);
end
text(0.5,0.2,'T1');
text(0.8,0.5,'T2');
text(0.5,0.8,'T3');
text(0.2,0.5,'T4');
text(0.27,0.25,'E1');
text(0.77,0.75,'E2');
Mesh.Plot_Subdomain('Diag');
hold off;
axis equal;
axis(AX);
axis off;
title('P_1 DoFs of Finite Element Space (Numbered Diamonds)');
```

Other types of sub-domains (e.g. sets of faces in a tetrahedral mesh) are handled in a similar way.

Chapter 7

Interpolating Finite Element Data

7.1 Automatically Generating Code

A common task in finite element codes is interpolation, which is used for plotting solution data, transferring data to different grids, etc. **FELICITY** can generate code to do this for you. This is accomplished by writing an input M-file that defines certain things:

- The domains on which to interpolate.
- The finite element spaces you want to use.
- The coefficient functions and geometry to interpolate.
- The expressions to interpolate (i.e. the interpolants).

Writing the script file is straightforward. Next, use **FELICITY** to process the script file into a stand-alone `C++` code. Then, compile it with a `C++` compiler. The following sections walk you through this process (example files are included).

7.2 Example 2-D Interpolation

Let Ω be a 2-D domain partitioned into a set of triangles (i.e. the mesh). Consider the finite element space:

$$\mathbb{V} := \{v \in H^1(\Omega) : v|_T \in \mathcal{P}_2(T), v \in C^0(\Omega)\}, \tag{7.1}$$

i.e. the space of piecewise quadratic polynomials, where T is a triangle contained in Ω, and $\mathcal{P}_2(T)$ is the space of quadratic polynomials on T.

We wish to interpolate the following expression:

$$\nabla p \cdot \mathbf{x}, \quad \text{on } \Omega,$$

where p is a function in $\mathbb{V}$ and $\mathbf{x}$ is the coordinate function on Ω. The following sections describe how to generate code to do this.

7.2.1 Writing The Input File

Here is a sample file `Interpolate_Grad_P_X_2D.m` that is explained in detail in the subsequent sections. It can be found in the `..\FELICITY\Demo\Interp_2D` sub-directory of **FELICITY**. To run this demo, execute `test_Interp_2D` at the MATLAB prompt.

```
function INTERP = Interpolate_Grad_P_X_2D()

% define domain
Omega = Domain('triangle');

% define finite element spaces
Scalar_P2  = Element(Omega,lagrange_deg2_dim2,1);

% define functions on FE spaces
p = Coef(Scalar_P2);

% define geometric function on 'Omega' domain
gf = GeoFunc(Omega);

% define expressions to interpolate
I_grad_p_X = Interpolate(Omega,p.grad' * gf.X);

% define geometry representation - Domain, reference element
G1 = GeoElement(Omega);

% define a set of interpolations to perform
INTERP = Interpolations(G1);

% collect all of the interpolations together
INTERP = INTERP.Append_Interpolation(I_grad_p_X);
```

Initial Items

The first line defines this as a MATLAB function:

```
function INTERP = Interpolate_Grad_P_X_2D()
```

where `INTERP` is a MATLAB object of class `Interpolations` (discussed later in this section).

The next few lines are similar to what is done when generating code for matrix assembly (Ch. 4). So you should review Sections 4.2.1, 4.5.2, and 4.5.3. Thus, the following lines:

```
% define domain
Omega = Domain('triangle');

% define finite element spaces
Scalar_P2  = Element(Omega,lagrange_deg2_dim2,1);
```

```
% define functions on FE spaces
p = Coef(Scalar_P2);

% define geometric function on 'Omega' domain
gf = GeoFunc(Omega);
```

define the domain `Omega` on which to interpolate, the finite element space `Scalar_P2`, a finite element coefficient function `p`, and a geometric function `gf` on `Omega`.

Interpolate

Next, define the expression to interpolate:

```
% define expressions to interpolate
I_grad_p_X = Interpolate(Omega,p.grad' * gf.X);
```

The class `Interpolate` requires two arguments. The first argument is the domain on which to interpolate the expression. The domain could be a subset of `Omega` of codimension ≥ 0 (this is particularly useful in higher dimensional problems). The second argument is a symbolic representation of the expression. In this case, the expression is $\nabla p \cdot \mathbf{x}$. Note: this is similar to how `Integral`(s) are defined (see Section 4.2.1).

The next line

```
% define geometry representation - Domain, reference element
G1 = GeoElement(Omega);
```

defines the geometry of the triangulation. In this case, the triangulation is assumed to consist of straight triangles (i.e. piecewise linear elements); see Section 4.2.1 for more information.

Collect All Interpolations

Create a MATLAB object of class `Interpolations`:

```
% define a set of interpolations to perform
INTERP = Interpolations(G1);
```

Then include all of the previously defined interpolants:

```
% collect all of the interpolations together
INTERP = INTERP.Append_Interpolation(I_grad_p_X);
```

This combines all of the information needed by **FELICITY** to automatically generate code that computes the interpolations defined above.

7.2.2 Generate Executable (MEX) to Compute Interpolations

If **FELICITY** is installed (see Section 1.1), you can execute the following command at the MATLAB prompt:

```
Main_Dir = 'C:\Your_Favorite_Directory\';
[status, Path_To_Mex] =...
    Convert_Interp_script_to_MEX(Main_Dir,...
                            'Interpolate_Grad_P_X_2D','Interp_2D');
```

Note that `Interpolate_Grad_P_X_2D.m` should be in the directory specified by `Main_Dir`. The last argument specifies what the executable will be called, and it will be placed in `Main_Dir`. Make sure that `Main_Dir` is in your MATLAB path.

7.2.3 Interpolating Data

First, create the domain (triangulation) for the unit square. At the MATLAB prompt (or in a script) type the following commands:

```
% create mesh (Omega is the unit square)
Vtx = [0 0; 1 0; 1 1; 0 1];
Tri = [1 2 3; 1 3 4];
Mesh = MeshTriangle(Tri, Vtx, 'Omega');
```

i.e. the mesh consists of two triangles. Next, define the piecewise quadratic finite element space:

```
% define FE space
P2_RefElem = ReferenceFiniteElement(lagrange_deg2_dim2(),1,true);
P2_Lagrange_Space = FiniteElementSpace('Scalar_P2', P2_RefElem, Mesh, 'Omega');
P2_DoFmap = uint32(Setup_Lagrange_P2_DoFmap(Mesh.Triangulation,[]));
P2_Lagrange_Space = P2_Lagrange_Space.Set_DoFmap(Mesh,P2_DoFmap);
P2_X = P2_Lagrange_Space.Get_DoF_Coord(Mesh); % get coordinates of P2 DoFs
```

See Section 6.4.2 for more info on how the `FiniteElementSpace` class works.

Set the coefficient function $p(x, y) = \sin(x + y)$:

```
% define coefficient function
p_func  = @(x,y) sin(x + y);
px_func = @(x,y) cos(x + y);
py_func = @(x,y) cos(x + y);
p_val = p_func(P2_X(:,1),P2_X(:,2)); % coefficient values
```

Next, define analytic function for the expression $\nabla p \cdot \mathbf{x}$:

```
% exact interpolant function
I_grad_p_X = @(x,y) px_func(x,y) .* x + py_func(x,y) .* y;
```

Now, define the coordinates of the interpolation points. In this example, we will only have two interpolation points:

```
% define interpolation points

% specify the triangle indices to interpolate within
Tri_Indices = [1;
               2];
% specify interpolation coordinates w.r.t. reference triangle
Ref_Coord = [(1/3), (1/3);
             (1/3), (1/3)];
% collect into a cell array
Omega_Interp_Data = {uint32(Tri_Indices), Ref_Coord};
```

The format requires the interpolation point data to be stored in a MATLAB cell array:

- The first element of `Omega_Interp_Data` is a MATLAB vector of unsigned integers, of length R, which are triangle indices. Note: in order to interpolate, you must specify the mesh elements to interpolate within.
- The second element of `Omega_Interp_Data` is a MATLAB matrix of size $R \times T$; R is the number of interpolation points and T is the topological dimension of the mesh element. In this example, $T = 2$.

Now use the executable that was generated before. Run the following at the MATLAB prompt (or insert after the above code in a script):

```
INTERP = Interp_2D(Mesh.X,uint32(Mesh.Triangulation),[],[],...
                   Omega_Interp_Data,P2_DoFmap,p_val);
```

Recall that we named the MEX file `Interp_2D`. In general, if you try to run the MEX file with the incorrect number of arguments, it will give you an error message but it will also *tell you* what the inputs should be. The output is a MATLAB `struct` of the following form:

```
INTERP(1).Name : 'I_grad_p_X'
INTERP(1).DATA : cell array containing interpolation data
                 for the expression
```

The `Name` field contains the same name that was used in the `Interpolate_Grad_P_X_2D.m` file. If there is more than one interpolation, they are *output in alphabetical order*.

Interpolating two points is not so complicated. Lets create a grid of points on the unit square and interpolate at those:

```
% now interpolate at a lot more points
x_vec = (0:0.1:1);
y_vec = (0:0.1:1);
[XX, YY] = meshgrid(x_vec,y_vec);
Interp_Pts_2 = [XX(:), YY(:)];
```

Remember that interpolating at a given point requires **FELICITY** to know the mesh element that the point belongs to. Thus, we must identify which triangle the above grid points belong to:

```
% find which triangle the points belong to
BOT_TRI = (Interp_Pts_2(:,1) >= Interp_Pts_2(:,2));
Tri_Indices_2 = zeros(size(Interp_Pts_2,1),1);
Tri_Indices_2(BOT_TRI)  = 1; % bottom triangle cell index
Tri_Indices_2(~BOT_TRI) = 2; % top triangle cell index
```

Then, we must convert the global grid coordinates to local reference coordinates:

```
BaryCentric = Mesh.cartToBary(Tri_Indices_2, Interp_Pts_2);
Ref_Coord_2 = Mesh.baryToRef(BaryCentric);
Omega_Interp_Data_2 = {uint32(Tri_Indices_2), Ref_Coord_2};
```

Note: `cartToBary` is a built-in MATLAB command under `TriRep`; `baryToRef` is a command built into the **FELICITY** mesh classes.

Execute the interpolation code again:

```
% interpolate!
INTERP_2 = Interp_2D(Mesh.X,uint32(Mesh.Triangulation),[],[],...
                     Omega_Interp_Data_2,P2_DoFmap,p_val);
Interp_Values = INTERP_2(1).DATA{1,1};
```

Now plot the grid data:

```
figure;
surf(XX, YY, I_grad_p_X(XX,YY)); % plot "exact" surface
hold on;
% plot the FE interpolated data as black dots
plot3(Interp_Pts_2(:,1),Interp_Pts_2(:,2),Interp_Values,'k.','MarkerSize',18);
hold off;
axis equal;
AZ = 70;
EL = 10;
view(AZ,EL);
title('Surface is Exact. Points Are Interpolated From FE Approximation',...
      'FontSize',12);
xlabel('x','FontSize',12);
ylabel('y','FontSize',12);
set(gca,'FontSize',12);
```

Note that the black dots do not appear exactly on the surface. This makes sense because we are interpolating a finite element *approximation* of the surface.

Remark 7.1. *The procedure is the same for interpolation in 1-D, 2-D, and 3-D. See the Google-Code page (Section 1.2) for more tutorials.*

7.3 Some Details of Transforming an Input .m File Into a MEX File

A sample MATLAB script (`test_Interp_2D.m`) will convert `Interpolate_Grad_P_X_2D.m` into a stand-alone `C++` code with MEX interface, compile it, and execute it on a sample problem (i.e. it will do what is described in Sections 7.2.2 and 7.2.3). This MATLAB script is located in the `..\FELICITY\Demo\Interp_2D` sub-directory of **FELICITY**. That script file calls two other files. The first is called `Convert_Interp_script_to_MEX.m` (presented below). The second is `Execute_Interp_2D.m` (located in the same directory) and is straightforward to understand (i.e. it implements what is described in Section 7.2.3).

The conversion routine (m-file) is given here:

```
function [status, Path_To_Mex] =...
          Convert_Interp_script_to_MEX(Main_Dir,Mscript,MEX_FileName)
%Convert_Interp_script_to_MEX
%
%   Generic compile procedure for generating MEX files that interpolate FEM
%   functions in FELICITY.  NO changes are necessary here.

FI                = FELInterface(Main_Dir); % do NOT change
Snippet_SubDir    = 'Scratch_Dir'; % no change necessary
GenCode_SubDir    = 'Interpolation_Code_AutoGen'; % no change necessary
MEX_Dir           = Main_Dir; % where the MEX-file goes

%%%%%%%% do NOT change %%%%%%%%
[status, Path_To_Mex] =...
   FI.Compile_Interpolation_Code(Mscript,Snippet_SubDir,GenCode_SubDir,...
                                 MEX_Dir,MEX_FileName);

end
```

The format and interface used here is similar to the matrix assembly code generation interface (see Section 4.3).

Remark 7.2 (How To Call The MEX File)**.** *First, try to execute the MEX file in MATLAB without any arguments. It will give you an error message, but it will also tell you what the inputs should be for your specialized interpolation application.*

Section 7.2 describes how to call the MEX file. You can also look at the M-file `Execute_Interp_2D.m` *(in the same directory as* `Interpolate_Grad_P_X_2D.m`*).*

7.4 Other Things You Can Do

Some more advanced features of automatic finite element interpolation code generation are the following.

- Interpolation can be done on sub-domains of co-dimension ≥ 0. In fact, one can do multiple interpolations on several different sub-domains when writing the input file (see Section 7.2.1). The only thing that changes for the MEX file is that you must supply multiple sets of interpolation points: one set for each distinct sub-domain. Remember, if you call the MEX function with no arguments, it will output an error message and tell you what the input arguments should be.

- All of the geometric function and coefficient function options listed in Section 4.5 apply here as well. Moreover, you can interpolate multiple geometric functions from different sub-domains in a *single interpolation expression*. Of course, in order to interpolate (on domain S) the geometric data from domain D, it is necessary that $S \subset D$.

Chapter 8

Point Searching On A Mesh

8.1 Automatically Generating Code

In Chapter 7, a MEX file was generated to interpolate finite element data. This required knowing the location of each interpolation point, which was specified by (i) the mesh cell that contains the point, and (ii) the local reference domain coordinates within that cell. But it is often the case that one starts with a set of *arbitrary* points with coordinates known only in the global space. In order to use the interpolation code in Chapter 7, one must translate the set of *global* coordinates into a set of mesh cells with corresponding *local* reference domains coordinates.

This is not such a simple task! Essentially, one must "search" the mesh for the specific cell that contains a given point. Once the cell is found, the corresponding local coordinates must be found. If the mesh is higher order (e.g. piecewise quadratic), then this requires an iterative procedure. Fortunately, **FELICITY** can generate code to do this for you. This is accomplished by writing an input M-file that defines certain things:

- The mesh sub-domains on which to search.
- The definition of how the mesh geometry is represented, e.g. piecewise linear elements, piecewise quadratic triangles, etc.

Writing the script file is straightforward. Next, use **FELICITY** to process the script file into a stand-alone `C++` code. Then, compile it with a `C++` compiler. The following sections walk you through this process (example files are included).

8.2 Example Of Point Searching In 2-D

Let Ω be a 2-D domain partitioned into a set of triangles (i.e. the mesh). Given a set of points with coordinates in $\mathbb{R}^2$, for each point we want to find the specific mesh triangle in Ω that contains that point. In addition, we want the corresponding local reference domain coordinates. The following sections describe how to generate code to do this.

8.2.1 Writing The Input File

Here is a sample file `Point_Search_Planar_Domain.m` that is explained in detail in the subsequent sections. It can be found in the `..\FELICITY\Demo\Point_Search_Domain_2D` subdirectory of **FELICITY**. To run this demo, execute `test_Point_Search_Planar_Domain` at the MATLAB prompt.

```
function DOM = Point_Search_Planar_Domain()

% define domain
Omega = Domain('triangle');

% define geometry representation of the global mesh domain
G1 = GeoElement(Omega); % arguments: Domain, reference element (optional)

% define a set of domains to point search in
DOM = PointSearches(G1);

% input the domain to be searched
DOM = DOM.Append_Domain(Omega);

end
```

Initial Items

The first line defines this as a MATLAB function:

```
function DOM = Point_Search_Planar_Domain()
```

where `DOM` is a MATLAB object of class `PointSearches` (discussed later in this section).

The next line defines the domain (recall Section 4.2.1):

```
% define domain
Omega = Domain('triangle');
```

i.e. define the domain `Omega` on which to point search.

Collect All Domains

The next line

```
% define geometry representation of the global mesh domain
G1 = GeoElement(Omega);
```

defines the geometry of the triangulation (of `Omega`). In this case, the triangulation is assumed to consist of straight triangles (i.e. piecewise linear elements); see Section 4.2.1 for more information.

Now create a MATLAB object of class `PointSearches`:

```
% define a set of domains to point search in
DOM = PointSearches(G1);
```

Then include all of the domains you want to search:

```
% input the domain to be searched
DOM = DOM.Append_Domain(Omega);
```

This combines all of the information needed by **FELICITY** to automatically generate code that will search a mesh for a given set of points.

8.2.2 Generate Executable (MEX) For Point Searching

If **FELICITY** is installed (see Section 1.1), you can execute the following command at the MATLAB prompt:

```
Main_Dir = 'C:\Your_Favorite_Directory\';
[status, Path_To_Mex] =...
    Convert_PtSearch_script_to_MEX(Main_Dir,...
                            'Point_Search_Planar_Domain','Search_2D');
```

Note that `Point_Search_Planar_Domain.m` should be in the directory specified by `Main_Dir`. The last argument specifies what the executable will be called, and it will be placed in `Main_Dir`. Make sure that `Main_Dir` is in your MATLAB path.

8.2.3 Finding Random Points

First, create a mesh (triangulation) for a square domain. At the MATLAB prompt (or in a script) type the following commands:

```
% load test mesh
[Omega_Vtx, Omega_Tri] = Standard_Triangle_Mesh_Test_Data();
% domain is a square [0, 2] x [0, 2]
```

Note: the mesh consists of 16 triangles. Next, define a mesh object and compute the local neighbor data structure:

```
MT = MeshTriangle(Omega_Tri, Omega_Vtx, 'Omega');
% make a triangulation so we can get neighbors
Omega_Neighbors = uint32(MT.neighbors);
```

See Section 2.1 for more info on how the `MeshTriangle` class works.

Define a set of random points in $[0, 2] \times [0, 2]$ to be the points to search for in the mesh:

```
% define the global points
NP = 20;
GX = 2*rand(NP,2);
```

Next, we need an initial guess for the cells (i.e. triangles) of the mesh that contain the given points `GX`:

```
% create random initial guesses as to which
%        mesh triangle contains each global point
Num_Cells = MT.Num_Cell();
Cell_Indices = uint32(randi(Num_Cells,NP,1));
```

The format of the MEX file requires the given point data to be stored in a MATLAB cell array:

```
% collect all the data needed to search the domain Omega
Omega_Given_Points = {Cell_Indices, GX, Omega_Neighbors};
```

In other words,

- The first element of `Omega_Given_Points` is a MATLAB vector of unsigned integers, of length R, which are triangle indices. This is the initial cells to try when searching the mesh for the given points.
- The second element of `Omega_Given_Points` is a MATLAB matrix of size $R \times D$; R is the number of points and D is the space dimension of the points. In this example, $D = 2$.
- The third element of `Omega_Given_Points` is a MATLAB matrix of size $M \times (T + 1)$; M is the number of cells in the sub-domain `Omega`, and T is the topological dimension of `Omega`. In this example, $T = 2$.

Now use the executable that was generated before. Run the following at the MATLAB prompt (or insert after the above code in a script):

```
SEARCH = Search_2D(MT.X,uint32(MT.Triangulation),[],[],Omega_Given_Points);
```

Recall that we named the MEX file `Search_2D`. In general, if you try to run the MEX file with the incorrect number of arguments, it will give you an error message but it will also *tell you* what the inputs should be. The output is a MATLAB `struct` of the following form:

```
SEARCH(1).Name : 'Omega'
SEARCH(1).DATA : cell array containing the found point data
```

The `Name` field contains the same name that was used in the `Point_Search_Planar_Domain.m` file for the domain to be searched. If there is more than one domain (or sub-domain) to search, they are *output in alphabetical order*.

Let us extract the "found" data from `SEARCH(1).DATA`:

```
CI = double(SEARCH.DATA{1}); % make it type double for convenience
Local_Ref_Coord = SEARCH.DATA{2};
```

The first one is the actual mesh cell indices that contain the given points. The second is the local reference domain coordinates of the given point in its corresponding mesh cell.

We can check the error between the original given points, and the points "found" by the MEX file. First,

```
% map local coordinates (of the found points)
%     to their corresponding global coordinates
BC = MT.refToBary(Local_Ref_Coord);
XC = MT.baryToCart(CI, BC);
```

Now compute the difference:

```
DIFF1 = GX - XC;
ERROR = max(abs(DIFF1(:)));
```

Note: `ERROR` should be very small (approximately equal to machine precision $< 10^{-15}$). Finally, we can plot the original points and the found points together as a visual check that the algorithm worked.

```
% plot the given points and found points
F1 = trimesh(MT.Triangulation,MT.X(:,1),MT.X(:,2),0*MT.X(:,1));
set(F1,'edgecolor','k');
hold on;
HG = plot(GX(:,1),GX(:,2),'kp','MarkerSize',15);
HF = plot(XC(:,1),XC(:,2),'r.','MarkerSize',15);
hold off;
legend([HG, HF],'Given Points','Found Points');
axis([0 2 0 2]);
axis square;
title('Point Search On A Planar Triangulation');
view(2);
```

Remark 8.1. *The procedure is the same for point searching in 1-D, 2-D, and 3-D. See the Google-Code page (Section 1.2) for more tutorials.*

8.3 Some Details of Transforming an Input .m File Into a MEX File

A sample MATLAB script (`test_Point_Search_Planar_Domain.m`) will convert `Point_Search_Planar_Domain.m` into a stand-alone `C++` code with MEX interface, compile it, and execute it on a sample problem (i.e. it will do what is described in Sections 8.2.2 and 8.2.3). This MATLAB script is located in the `..\FELICITY\Demo\Point_Search_Domain_2D` sub-directory of **FELICITY**. That script file calls two other files. The first is called `Convert_PtSearch_script_to_MEX.m` (presented below). The second is `Execute_Point_Search_Planar_Domain.m` (located in the same directory) and is straightforward to understand (i.e. it implements what is described in Section 8.2.3).

The conversion routine (m-file) is given here:

```
function [status, Path_To_Mex] =...
          Convert_PtSearch_script_to_MEX(Main_Dir,Pscript,MEX_FileName)
%Convert_PtSearch_script_to_MEX
```

```
%
%   Generic compile procedure for generating MEX files that perform
%   point searches of meshes via FELICITY's code generation.
%   NO changes are necessary here.
%
%   [status, Path_To_Mex] = Convert_PtSearch_script_to_MEX(...
%                                   Main_Dir,Pscript,MEX_FileName);
%
%   Main_Dir     = (string) full path to the location of interest.
%   Pscript      = FELICITY script (m-file) that defines
%                  the point searching to perform.
%   MEX_FileName = (string) name of MEX file.
%
%   status      = success == 0 or failure ~= 0.
%   Path_To_Mex = (string) full path to the MEX file.

FI                = FELInterface(Main_Dir); % do NOT change
GenCode_SubDir    = 'Pt_Search_Code_AutoGen'; % no change necessary
MEX_Dir           = Main_Dir; % where the MEX-file goes

%%%%%%%% do NOT change %%%%%%%%
[status, Path_To_Mex] =...
   FI.Compile_Point_Search_Code(Pscript,GenCode_SubDir,...
                                MEX_Dir,MEX_FileName);
end
```

The format and interface used here is similar to the matrix assembly code generation interface (see Section 4.3).

Remark 8.2 (How To Call The MEX File)**.** *First, try to execute the MEX file in MATLAB without any arguments. It will give you an error message, but it will also tell you what the inputs should be for your specialized point search application.*

Section 8.2.3 describes how to call the MEX file. You can also look at the M-file `Execute_Point_Search_Planar_Domain.m` *(in the same directory as* `Point_Search_Planar_Domain.m`*).*

8.4 Other Things You Can Do

Some more advanced features of automatic point search code generation are the following.

- Searching can be done on sub-domains of co-dimension ≥ 0. In fact, one can search several different sub-domains when writing the input file (see Section 8.2.1). The only thing that changes for the MEX file is that you must supply multiple sets of points (to be searched): one set for each distinct sub-domain. Remember, if you call the MEX function with no arguments, it will output an error message and tell you what the input arguments should be.

- Moreover, the domains to be searched can be curves or surfaces embedded in $\mathbb{R}^3$ (e.g. 3-D curves and surfaces). In this case, the generated MEX file searches for the **closest point**, in the surface, to the given point. Calling the MEX file is essentially identical to the example given above. See the demo in `..\FELICITY\Demo\Closest_Point_Sphere` for an example. Also see the Google-Code page (Section 1.2) for more tutorials.

Chapter 9

Miscellaneous Tools

FELICITY includes some additional utilities and modules. Each is stand-alone and can be used with all other components.

9.1 Eikonal Equation Solver

In many applications, it is crucial to solve the Eikonal equation:

$$\begin{aligned} |\nabla u| &= 1, \text{ in } \Omega \\ u &= 0, \text{ on } \partial\Omega, \end{aligned} \tag{9.1}$$

i.e. u is the distance function on Ω to the boundary $\partial\Omega$. A solver for this problem is available as a `C++` code, with MATLAB class interface, in the sub-directory:
`./FELICITY/Static_Codes/Eikonal_2D/`

The solver works on arbitrary 2-D triangulations and allows for a non-Euclidean metric, i.e. $|\nabla u|_M = 1$. See the `README.txt` in the `Eikonal_2D` directory for more details. Also see the unit test:
`./FELICITY/Static_Codes/Eikonal_2D/Unit_Test/test_Eikonal2D_Solve.m`

9.2 Unstructured Mesh Generation: 2-D and 3-D

Finite elements need mesh generation. Thus, **FELICITY** provides an implementation of the method in [13] for generating simplex meshes of 2-D and 3-D closed, smooth iso-surfaces. Note: this method of mesh generation is only appropriate for smooth bounding surfaces. This is implemented as a mixed `C++` and MATLAB interface in the sub-directory:
`./FELICITY/Static_Codes/Isosurface_Meshing/`.

A tutorial on creating meshes is available on the Google-Code page:

`http://code.google.com/p/felicity-finite-element-toolbox/`

Also see the unit tests in:
`./FELICITY/Static_Codes/Isosurface_Meshing/Unit_Test/`

9.3 Hierarchical Search Trees

Interpolating finite element functions at specific points in space requires finding the enclosing cell and the corresponding local reference cell coordinates (see Chapter 7). **FELICITY** provides a way to search for points on meshes (see Chapter 8), but this requires an initial guess for the enclosing cell. Furthermore, if the initial guess is "bad," then the point search code can get "stuck." So having a good initial guess is crucial.

One way to get a good initial guess is to do a brute force search of all cells in the mesh, i.e. find the cell that is closest to a given point and use that as the initial guess. But this can be expensive! An efficient way to do this is to use a hierarchical search tree, such as a quadtree or octree. This is implemented as a mixed `C++` and MATLAB interface in the sub-directory:
`./FELICITY/Static_Codes/Search_Trees/`.

A tutorial on using search trees is available on the Google-Code page:

`http://code.google.com/p/felicity-finite-element-toolbox/`

Also see the unit tests in:
`./FELICITY/Static_Codes/Search_Trees/Unit_Test/`

Appendix A

Sparse Matrices

A.1 Sparse Matrix Storage

Consider a sparse matrix M:

$$M = \begin{bmatrix} 2.3 & 0 & 0 & 0 & 95.3 & 0 \\ 0 & 1.4 & 0 & 0 & -13.3 & 12.6 \\ 0 & 7.66 & 0 & 15.6 & 0 & 0 \\ 0 & -9.9 & 0 & -34.3 & 0 & -17.1 \\ -4.8 & 0 & 0 & 0 & 0 & 23.7 \end{bmatrix}. \tag{A.1}$$

Let nnz be the number of non-zero elements in M (nnz = 12 in this case). Let $m = 5$ and $n = 6$ be the number of rows and columns of M. Then the corresponding Compressed Sparse Column (CSC) format is given in Table A.1. This is the format used by MATLAB.

Table A.1: The contents of the sparse data arrays associated with the sparse matrix in (A.1). ir and pr both have length nnz; jc has length $n + 1$. Note that the rows and columns of M are indexed between 0 and $m - 1$ and 0 and $n - 1$, respectively.

C-index	0	1	2	3	4	5	6	7	8	9	nnz − 2	nnz − 1
ir	0	4	1	2	3	2	3	0	1	1	3	4
pr	2.3	-4.8	1.4	7.66	-9.9	15.6	-34.3	95.3	-13.3	12.6	-17.1	23.7

C-index	0	1	2	3	4	n-1	n					
jc	0	2	5	5	7	nnz − 3	nnz					

Let j be a column of the matrix. Then,

$$\begin{aligned} &\text{non-zero elements in column j are between row indices ir[jc[j]] and ir[jc[j + 1] − 1],} \\ &\qquad \text{with corresponding data values given between pr[jc[j]] and pr[jc[j + 1] − 1].} \end{aligned} \tag{A.2}$$

Note: if jc[j] = jc[j + 1], then the entries of column j are all zero.

A.2 C++ Class For Sparse Matrix Assembly

I would like to thank David Bindel for providing an easy to use and efficient sparse matrix assembly class that I could integrate with MATLAB's MEX interface. For those interested in obtaining a generic class for inserting local matrices into a global sparse matrix, the files can be found in the following sub-directory:
`..\FELICITY\Code_Generation\@FELCodeHdr\private\Matrix_Assembler`
The necessary files are:

- `block_alloc.h`
- `abstract_assembler.cc`
- `assembler.cc`
- `reassembler.cc`
- `simple_assembler.cc`

Note: you can also find this on David Bindel's website:
`http://www.cs.cornell.edu/~bindel/`

Appendix B

Notes on Generating Code

Here are some notes on ways to implement code generation for finite element methods. You may find them useful as maxims to follow in your own work.

B.1 Computing The Local Finite Element Matrix

One of the most fundamental aspects of a finite element (FE) code is the evaluation of the local FE matrix. If you can do that, you are well on your way to numerically solving a PDE. You just need a way of inserting the matrix into the global matrix (there are libraries available for this in a variety of sparse matrix formats). Of course, you also need a solver (not a trivial matter for large problems, but solver libraries are also available). But implementing local FE matrix evaluation for a particular problem can be tedious if you use higher order methods, non-standard bilinear forms, or you have a nonlinear map that describes the domain shape.

The goal here is to build up a systematic approach for *automatically* generating code that implements the local matrix computation. In doing this, we will capitalize on MATLAB's symbolic toolbox (Maple).

B.1.1 Notation; Definitions

Domain Geometry

One fundamental aspect of solving PDEs is specifying the *geometry of the domain*, regardless of the solution method. Typically, this is done by an explicit parametrization, i.e. some mapping from a *reference* domain to the physical domain. The reference domain is usually chosen to be something simple (e.g. a box) for computational convenience. The parametrization could be a globally defined smooth function, but this may not be easy to implement if the domain geometry is complex.

Instead, the mapping is usually defined piecewise using "local charts" (c.f. differential geometry), where each local map is defined over a simple reference domain. Moreover, the collection of all the local maps defines a continuous *global* map, possibly with higher differentiability (e.g. NURBS).

Standard Mapping

One method for generating these local charts is the following. Given a domain Ω, we decompose it into a set of (small) sub-domains that are all topologically identical, i.e. a mesh. For example, if $\Omega \subset \mathbb{R}^2$ is a polygon, it can be decomposed into a set of triangles (a triangulation). The sub-domains are called *elements*. Let $\mathcal{T}$ be the set of elements obtained by "meshing" Ω, i.e. the mesh $\mathcal{T}$ satisfies

$$\Omega = \cup_{T \in \mathcal{T}} T, \tag{B.1}$$

where $T \subset \Omega$ denotes a particular element in the mesh. Since all the elements are topologically identical, we can choose a single *reference element* to identify them with. Denote the reference element by $\widehat{T}$. Now, given $T \in \mathcal{T}$, define $\mathbf{\Phi} : \widehat{T} \to T \subset \mathbb{R}^d$ (a map) such that $T = \mathbf{\Phi}(\widehat{T})$. (Really $\mathbf{\Phi}$ should be subscripted with T, but we drop it for simplicity.) Each $\mathbf{\Phi}$ is a local chart that completely defines the geometry of an element of Ω. Thus, taking the entire collection $\{\mathbf{\Phi}\}$ defines Ω.

Therefore, regardless of the finite element method used, we must do one of the following:

- Given a domain Ω, decompose its geometry via a mesh $\mathcal{T}$. This induces a collection of local maps, i.e. $\{\mathbf{\Phi}_T\}_{T \in \mathcal{T}}$.
- First, specify a collection of local maps $\{\mathbf{\Phi}_T\}_{T \in \mathcal{T}}$, where each $\mathbf{\Phi}_T : \widehat{T} \to T$. Then define Ω by (B.1).

Either way, we must know how each element in Ω is related to the reference element.

Local-to-Global Mapping Identities

In later sections, we use the following identities. Let $\mathbf{x} = \mathbf{\Phi}(\hat{\mathbf{x}})$ for all $\hat{\mathbf{x}} \in \widehat{T}$; obviously, $\hat{\mathbf{x}} = \mathbf{\Phi}^{-1}(\mathbf{x})$ for all $\mathbf{x} \in T$. Moreover, $(\mathbf{\Phi}^{-1} \circ \mathbf{\Phi})(\hat{\mathbf{x}}) = \hat{\mathbf{x}}$, for all $\hat{\mathbf{x}} \in \widehat{T}$, and $(\mathbf{\Phi} \circ \mathbf{\Phi}^{-1})(\mathbf{x}) = \mathbf{x}$, for all $\mathbf{x} \in T$. This leads to the following inverse function theorem identity:

$$\begin{aligned} \mathbf{I} &= \nabla_{\hat{\mathbf{x}}}(\mathbf{\Phi}^{-1} \circ \mathbf{\Phi})(\hat{\mathbf{x}}) = [\nabla_{\mathbf{x}} \mathbf{\Phi}^{-1}(\mathbf{x})][\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})], \\ \implies (\nabla_{\mathbf{x}} \mathbf{\Phi}^{-1} \circ \mathbf{\Phi})(\hat{\mathbf{x}}) &= [\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})]^{-1}, \end{aligned} \tag{B.2}$$

where the term on the end is the matrix inverse of $\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})$.

We also have an identity involving second derivatives. First, rewrite the first line in (B.2) with indices

$$\delta_{ij} = \hat{\partial}_j (\mathbf{\Phi}_i^{-1} \circ \mathbf{\Phi})(\hat{\mathbf{x}}) = \sum_k [\partial_k \mathbf{\Phi}_i^{-1}(\mathbf{x})][\hat{\partial}_j \mathbf{\Phi}_k(\hat{\mathbf{x}})],$$

and take another derivative

$$\begin{aligned} 0 &= \hat{\partial}_q \hat{\partial}_j (\mathbf{\Phi}_i^{-1} \circ \mathbf{\Phi})(\hat{\mathbf{x}}) \\ &= \sum_k \sum_l [\partial_l \partial_k \mathbf{\Phi}_i^{-1}(\mathbf{x})][\hat{\partial}_j \mathbf{\Phi}_k(\hat{\mathbf{x}})][\hat{\partial}_q \mathbf{\Phi}_l(\hat{\mathbf{x}})] + \sum_k [\partial_k \mathbf{\Phi}_i^{-1}(\mathbf{x})][\hat{\partial}_q \hat{\partial}_j \mathbf{\Phi}_k(\hat{\mathbf{x}})]. \end{aligned} \tag{B.3}$$

Next, replace $\mathbf{x}$ with $\mathbf{\Phi}(\hat{\mathbf{x}})$ and use the identity in (B.2) to get

$$\begin{aligned}\sum_k \sum_l (\partial_l \partial_k \mathbf{\Phi}_i^{-1} \circ \mathbf{\Phi})(\hat{\mathbf{x}})[\hat{\partial}_j \mathbf{\Phi}_k(\hat{\mathbf{x}})][\hat{\partial}_q \mathbf{\Phi}_l(\hat{\mathbf{x}})] &= -\sum_k (\partial_k \mathbf{\Phi}_i^{-1} \circ \mathbf{\Phi})(\hat{\mathbf{x}})[\hat{\partial}_q \hat{\partial}_j \mathbf{\Phi}_k(\hat{\mathbf{x}})] \\ \sum_k \sum_l (\partial_l \partial_k \mathbf{\Phi}_i^{-1} \circ \mathbf{\Phi})(\hat{\mathbf{x}})[\hat{\partial}_j \mathbf{\Phi}_k(\hat{\mathbf{x}})][\hat{\partial}_q \mathbf{\Phi}_l(\hat{\mathbf{x}})] &= -\sum_k \left([\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})]^{-1}\right)_{ik} [\hat{\partial}_q \hat{\partial}_j \mathbf{\Phi}_k(\hat{\mathbf{x}})].\end{aligned} \tag{B.4}$$

Assume that $[\hat{\partial}_j \mathbf{\Phi}_k(\hat{\mathbf{x}})]$ is a square matric and let $G_{kj} := [\hat{\partial}_j \mathbf{\Phi}_k(\hat{\mathbf{x}})] = [\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})]_{kj}$ and $G^{kl} := \left([\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})]^{-1}\right)_{kl}$ and note the Kronecker property $\delta_{sr} = \sum_i G_{si} G^{ir}$. Thus, the last line of (B.4) becomes

$$\begin{aligned}\sum_k \sum_l (\partial_l \partial_k \mathbf{\Phi}_i^{-1} \circ \mathbf{\Phi})(\hat{\mathbf{x}}) G_{kj} G_{lq} &= -\sum_k G^{ik} [\hat{\partial}_q \hat{\partial}_j \mathbf{\Phi}_k(\hat{\mathbf{x}})] \\ \sum_j \sum_q \sum_k \sum_l (\partial_l \partial_k \mathbf{\Phi}_i^{-1} \circ \mathbf{\Phi})(\hat{\mathbf{x}}) G_{kj} G^{jz} G_{lq} G^{qw} &= -\sum_r G^{ir} \sum_j \sum_q G^{jz} G^{qw} [\hat{\partial}_q \hat{\partial}_j \mathbf{\Phi}_r(\hat{\mathbf{x}})] \\ (\partial_w \partial_z \mathbf{\Phi}_i^{-1} \circ \mathbf{\Phi})(\hat{\mathbf{x}}) &= -\sum_r G^{ir} \sum_j \sum_q G^{jz} G^{qw} [\hat{\partial}_q \hat{\partial}_j \mathbf{\Phi}_r(\hat{\mathbf{x}})].\end{aligned} \tag{B.5}$$

Finally, we obtain (when $[\hat{\partial}_j \mathbf{\Phi}_k(\hat{\mathbf{x}})]$ is square)

$$\begin{aligned}(\partial_w \partial_z \mathbf{\Phi}_i^{-1} \circ \mathbf{\Phi})(\hat{\mathbf{x}}) &= -\sum_r \left([\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})]^{-1}\right)_{ir} \left(\left([\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})]^{-1}\right)^{\dagger} \underbrace{[\nabla^2_{\hat{\mathbf{x}}} \mathbf{\Phi}_r(\hat{\mathbf{x}})]}_{\text{matrix}} [\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})]^{-1} \right)_{wz} \\ (\nabla^2_{\mathbf{x}} \mathbf{\Phi}_i^{-1} \circ \mathbf{\Phi})(\hat{\mathbf{x}}) &= -\sum_r \left([\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})]^{-1}\right)_{ir} \left([\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})]^{-1}\right)^{\dagger} [\nabla^2_{\hat{\mathbf{x}}} \mathbf{\Phi}_r(\hat{\mathbf{x}})][\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})]^{-1}.\end{aligned} \tag{B.6}$$

Local Basis Function

Having a mesh provides a convenient way to represent the solution of the PDE. For the sake of example, we restrict ourselves to the 2-D Euclidean plane. Let $\hat{f} : \widehat{T} \to \mathbb{R}$ be a scalar valued function defined on the *reference* element $\widehat{T} \subset \mathbb{R}^2$ (e.g. logical reference triangle). Given $T \in \mathcal{T}$, we have the map $\mathbf{\Phi} : \widehat{T} \to T \subset \mathbb{R}^2$, such that $T = \mathbf{\Phi}(\widehat{T})$. Lastly, let $f : T \to \mathbb{R}$ be the function obtained by

$$f(\mathbf{x}) = \hat{f}(\mathbf{\Phi}^{-1}(\mathbf{x})), \quad \text{for all } \mathbf{x} \in T. \tag{B.7}$$

Ultimately, f represents some quantity written with respect to the *natural ambient coordinates* of the problem that we want to compute by solving a PDE. More specifically, f plays the role of a *global* basis function used to represent the solution of the PDE.

The function $\hat{f}$ is a local representation of the function that is easy to define and compute with because the reference element $\widehat{T}$ is simple. We call $\hat{f}$ a *local* basis function. Note that, because we have a set of local charts $\{\mathbf{\Phi}\}$, we can use the *same* local basis functions for every element $T \in \mathcal{T}$.

Change of Variables; Chain Rule

Computing derivatives of global quantities (i.e. f) is a typical operation in many finite element codes. However, it is more convenient to compute in terms of *local coordinates* because f is defined in terms of $\hat{f}$ and $\mathbf{\Phi}$. Of course, the complication is in dealing with the chain rule while differentiating (B.7), that old friend (fiend?) from calculus.

From (B.7), it is clear that $f(\mathbf{x}) = \hat{f}(\hat{\mathbf{x}})$. The transformation rule for the gradient is:

$$\begin{aligned} \partial_{x_j} f(\mathbf{x}) &= \partial_{x_j} \hat{f}(\mathbf{\Phi}^{-1}(\mathbf{x})) = \sum_k \partial_{\hat{x}_k} \hat{f}(\hat{\mathbf{x}})\Big|_{\hat{\mathbf{x}}=\mathbf{\Phi}^{-1}(\mathbf{x})} \partial_{x_j}(\mathbf{\Phi}^{-1}(\mathbf{x}) \cdot \mathbf{e}_k) \\ \Rightarrow \quad \underbrace{(\nabla_{\mathbf{x}} f)(\mathbf{x})}_{\text{row vector}} &= \underbrace{\nabla_{\hat{\mathbf{x}}} \hat{f}(\mathbf{\Phi}^{-1}(\mathbf{x}))}_{\text{row vector}} \underbrace{(\nabla_{\mathbf{x}} \mathbf{\Phi}^{-1})(\mathbf{x})}_{\text{matrix}}, \end{aligned} \tag{B.8}$$

which implies that

$$\begin{aligned} (\nabla_{\mathbf{x}} f) \circ \mathbf{\Phi}(\hat{\mathbf{x}}) &= \nabla_{\hat{\mathbf{x}}} \hat{f}(\hat{\mathbf{x}}) (\nabla_{\mathbf{x}} \mathbf{\Phi}^{-1} \circ \mathbf{\Phi})(\hat{\mathbf{x}}) \\ \text{using (B.2)} \rightarrow &= \underbrace{\nabla_{\hat{\mathbf{x}}} \hat{f}(\hat{\mathbf{x}})}_{\text{row vector}} \underbrace{[\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})]^{-1}}_{\text{matrix}}, \end{aligned} \tag{B.9}$$

where we took advantage of mapping to the reference element $\widehat{T}$.

We have the standard formula for applying a change of variables to an integral:

$$\int_T f(\mathbf{x})\, d\mathbf{x} = \int_{\widehat{T}} f(\mathbf{\Phi}(\hat{\mathbf{x}})) \det\left(\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})\right)\, d\hat{\mathbf{x}} = \int_{\widehat{T}} \hat{f}(\hat{\mathbf{x}}) \det\left(\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})\right)\, d\hat{\mathbf{x}}. \tag{B.10}$$

And when a gradient is present, you get

$$\begin{aligned} \int_T (\nabla_{\mathbf{x}} f)(\mathbf{x})\, d\mathbf{x} &= \int_{\widehat{T}} (\nabla_{\mathbf{x}} f) \circ \mathbf{\Phi}(\hat{\mathbf{x}}) \det\left(\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})\right)\, d\hat{\mathbf{x}} \\ \text{using (B.9)} \rightarrow &= \int_{\widehat{T}} \nabla_{\hat{\mathbf{x}}} \hat{f}(\hat{\mathbf{x}}) [\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})]^{-1} \det\left(\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})\right)\, d\hat{\mathbf{x}}. \end{aligned} \tag{B.11}$$

As an example, here is how the stiffness matrix transforms (for Laplace's equation) on a single element:

$$\begin{aligned} \int_T \nabla f_i \cdot \nabla f_j &= \int_T \nabla_{\mathbf{x}} f_i(\mathbf{x}) [\nabla_{\mathbf{x}} f_j(\mathbf{x})]^\dagger\, d\mathbf{x} \\ &= \int_{\widehat{T}} [\nabla_{\hat{\mathbf{x}}} \hat{f}_i(\hat{\mathbf{x}}) [\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})]^{-1}] [\nabla_{\hat{\mathbf{x}}} \hat{f}_j(\hat{\mathbf{x}}) [\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})]^{-1}]^\dagger \det\left(\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})\right)\, d\hat{\mathbf{x}}. \end{aligned} \tag{B.12}$$

We now derive the transformation rule for the Hessian. From (B.8), the product rule gives

$$\begin{aligned} \partial_{x_i} \partial_{x_j} f(\mathbf{x}) &= \partial_{x_i} \sum_k \partial_{\hat{x}_k} \hat{f}(\hat{\mathbf{x}})\Big|_{\hat{\mathbf{x}}=\mathbf{\Phi}^{-1}(\mathbf{x})} \partial_{x_j}(\mathbf{\Phi}^{-1}(\mathbf{x}) \cdot \mathbf{e}_k) \\ &= \sum_k \Big\{ \partial_{x_i} \left(\partial_{\hat{x}_k} \hat{f}(\hat{\mathbf{x}})\Big|_{\hat{\mathbf{x}}=\mathbf{\Phi}^{-1}(\mathbf{x})} \right) \partial_{x_j}(\mathbf{\Phi}^{-1}(\mathbf{x}) \cdot \mathbf{e}_k) \\ &\qquad + \left(\partial_{\hat{x}_k} \hat{f}(\hat{\mathbf{x}})\Big|_{\hat{\mathbf{x}}=\mathbf{\Phi}^{-1}(\mathbf{x})} \right) \partial_{x_i} \partial_{x_j}(\mathbf{\Phi}^{-1}(\mathbf{x}) \cdot \mathbf{e}_k) \Big\}. \end{aligned} \tag{B.13}$$

For the moment, consider the first term:

$$\partial_{x_i}\left(\partial_{\hat{x}_k}\hat{f}(\hat{\mathbf{x}})\Big|_{\hat{\mathbf{x}}=\mathbf{\Phi}^{-1}(\mathbf{x})}\right) = \sum_l \left(\partial_{\hat{x}_l}\partial_{\hat{x}_k}\hat{f}(\hat{\mathbf{x}})\Big|_{\hat{\mathbf{x}}=\mathbf{\Phi}^{-1}(\mathbf{x})}\right)\partial_{x_i}(\mathbf{\Phi}^{-1}(\mathbf{x})\cdot\mathbf{e}_l), \tag{B.14}$$

so we can rewrite (B.13) as

$$\begin{aligned}\partial_{x_i}\partial_{x_j}f(\mathbf{x}) = &\sum_k\sum_l \left(\partial_{\hat{x}_l}\partial_{\hat{x}_k}\hat{f}(\hat{\mathbf{x}})\Big|_{\hat{\mathbf{x}}=\mathbf{\Phi}^{-1}(\mathbf{x})}\right)\partial_{x_i}(\mathbf{\Phi}^{-1}(\mathbf{x})\cdot\mathbf{e}_l)\,\partial_{x_j}(\mathbf{\Phi}^{-1}(\mathbf{x})\cdot\mathbf{e}_k)\\ &+\sum_k\left(\partial_{\hat{x}_k}\hat{f}(\hat{\mathbf{x}})\Big|_{\hat{\mathbf{x}}=\mathbf{\Phi}^{-1}(\mathbf{x})}\right)\partial_{x_i}\partial_{x_j}(\mathbf{\Phi}^{-1}(\mathbf{x})\cdot\mathbf{e}_k),\end{aligned} \tag{B.15}$$

which can be written more succinctly as

$$\begin{aligned}[\nabla^2_{\mathbf{x}}f(\mathbf{x})]_{ij} = &(\partial_{x_i}\mathbf{\Phi}^{-1})(\mathbf{x})\cdot\underbrace{[\nabla^2_{\hat{\mathbf{x}}}\hat{f}(\mathbf{\Phi}^{-1}(\mathbf{x}))]}_{\text{matrix}}\underbrace{(\partial_{x_j}\mathbf{\Phi}^{-1})(\mathbf{x})}_{\text{col. vector}}\\ &+\underbrace{\nabla_{\hat{\mathbf{x}}}\hat{f}(\mathbf{\Phi}^{-1}(\mathbf{x}))}_{\text{vector}}\cdot\underbrace{[\partial_{x_i}\partial_{x_j}\mathbf{\Phi}^{-1}(\mathbf{x})]}_{\text{vector}}.\end{aligned} \tag{B.16}$$

Using (B.2) and (B.9), we get

$$\begin{aligned}[\nabla^2_{\mathbf{x}}f]\circ\mathbf{\Phi}(\hat{\mathbf{x}}) = &\left([\nabla_{\hat{\mathbf{x}}}\mathbf{\Phi}(\hat{\mathbf{x}})]^{-1}\right)^{\dagger}[\nabla^2_{\hat{\mathbf{x}}}\hat{f}(\hat{\mathbf{x}})][\nabla_{\hat{\mathbf{x}}}\mathbf{\Phi}(\hat{\mathbf{x}})]^{-1}\\ &+\sum_k\left(\nabla_{\hat{\mathbf{x}}}\hat{f}(\hat{\mathbf{x}})\cdot\mathbf{e}_k\right)(\nabla^2_{\mathbf{x}}\mathbf{\Phi}_k^{-1}\circ\mathbf{\Phi})(\hat{\mathbf{x}}),\end{aligned} \tag{B.17}$$

which can be reduced further by using (B.6).

B.1.2 Evaluating Expressions

In many applications, such as (B.12), $\hat{f}_i$ represents a basis function (i.e. a polynomial defined over $\widehat{T}$). So at this stage, everything in (B.12) is directly computable (assuming you have an explicit formula for $\mathbf{\Phi}$). Therefore, it is tempting to throw the entire formula (B.12) into a symbolic computing toolbox and evaluate for each pair (i,j). Besides, we just want the left-hand-side of (B.12).

However, I caution against this! It is true that a symbolic toolbox could generate code to compute the right-hand-side of (B.12). But as of today, these toolboxes are not yet smart enough to do so in the most efficient way possible. For instance:

A. You will need to loop over the indices (i,j). This means certain sub-calculations will be redone each time (e.g. the mapping of ∇f_i). I am not aware that a symbolic toolbox can reliably detect this and account for it (yet).

B. There are quadrature loops to consider, which can further compound the cost of redundant calculations.

C. $\mathbf{\Phi}$ may be complicated, so computing the jacobian $[\nabla_{\hat{\mathbf{x}}}\mathbf{\Phi}(\hat{\mathbf{x}})]$ and its inverse should be done as a separate calculation. In other words, it is beneficial to reuse calculations of the jacobian, inverse, and its determinant.

D. You may not need the full gradient of $\hat{f}$ in all situations. There are certainly examples of this in FEM.

E. You may want to assemble more than one FE matrix at once. In this case, it is advantageous to reuse the mappings of ∇f_i when it appears in multiple bilinear forms (FE matrices).

Hence, it is advantageous to break the calculation up into the smallest pieces possible. For example, first order derivatives will require computations like in (B.9). So it seems reasonable that ∇f_i should be "pre-computed" before considering the local FE matrix. Note: higher order derivatives of f_i (e.g. the Hessian) will require higher order derivatives of the map $\mathbf{\Phi}$.

Computing The Jacobian And Other Terms

It is also evident from (B.9), that we need to compute the jacobian. The most fundamental part of matrix assembly is the handling of the local-to-global map $\mathbf{\Phi}$.

Now we finally get to the first instance of *code automation*. To describe this, we define some simple data structures:

$$\begin{aligned} \texttt{double Phi\_value\_i[Nq]} &= \text{ array for each } \texttt{i} = 1, 2, \\ \texttt{double Phi\_grad\_ij[Nq]} &= \text{ array for each } \texttt{i,j} = 1, 2, \\ \texttt{double Phi\_grad\_det[Nq]} &= \text{ array}, \\ \texttt{double Phi\_grad\_inv\_ij[Nq]} &= \text{ array for each } \texttt{i,j} = 1, 2, \end{aligned} \tag{B.18}$$

where `Nq` is the number of quadrature points used in computing the integrals. Conceptually, you can think of $\hat{\mathbf{x}}$ as being a quadrature point in $\widehat{T}$ and $\mathbf{x} = \mathbf{\Phi}(\hat{\mathbf{x}})$ is the location of the quadrature point in T. Thus, we have

A. `Phi_value_i` contains the evaluation of $\mathbf{\Phi}(\hat{\mathbf{x}})$ at each quadrature point. Typically, it is not needed, unless you must evaluate expressions that depend on the coordinate $\mathbf{x}$.

B. `Phi_grad_ij` contains the evaluation of $[\nabla_{\hat{\mathbf{x}}}\mathbf{\Phi}(\hat{\mathbf{x}})]$ at each quadrature point.

C. `Phi_grad_det` contains the evaluation of $\det([\nabla_{\hat{\mathbf{x}}}\mathbf{\Phi}(\hat{\mathbf{x}})])$ at each quadrature point. This can be computed directly from `Phi_grad_ij`.

D. `Phi_grad_inv_ij` contains the evaluation of $[\nabla_{\hat{\mathbf{x}}}\mathbf{\Phi}(\hat{\mathbf{x}})]^{-1}$ at each quadrature point. This can be computed directly from `Phi_grad_ij` and `Phi_grad_det`.

Of course, you must specify $\mathbf{\Phi}$ in some way; this is part of defining the problem that you want to solve. This can be done in many ways. If you have an explicit formula for $\mathbf{\Phi}$, then it is trivial to insert the expression into a symbolic toolbox and have it evaluated at the quadrature points of the integration rule you chose. The standard approach in FE

computations is to use the mesh data structure to define an affine map (linear) from $\widehat{T}$ to T. In this case, $\nabla\mathbf{\Phi}$ is constant, so you can even simplify the data structures.

However, we will take a more general approach and assume that $\mathbf{\Phi}$ depends on a finite number of parameters (coefficients), e.g.

$$\mathbf{\Phi}(\hat{\mathbf{x}};\boldsymbol{\alpha}) = \begin{bmatrix} (1+e^{\alpha_1\hat{x}_1})(1+\alpha_2\sin(2\pi\hat{x}_2)) \\ \hat{x}_2 \end{bmatrix}, \text{ where } \hat{\mathbf{x}} = (\hat{x}_1,\hat{x}_2), \tag{B.19}$$

or something more linear

$$\mathbf{\Phi}(\hat{\mathbf{x}};\boldsymbol{\alpha}) = \sum_{j=1}^{M} \boldsymbol{\alpha}_j\phi_j(\hat{\mathbf{x}}), \quad \phi_j : \widehat{T} \to \mathbb{R}, \text{ for all } j, \tag{B.20}$$

where the functions $\{\phi_j\}$ are user specified functions. (B.20) is called iso-parametric domain approximation in some cases.

With the mappings in (B.19) and (B.20), it is straightforward to evaluate the data structures in (B.18) using a symbolic toolbox for the "heavy lifting." You then only have to pass the coefficient data $\boldsymbol{\alpha}$ to the appropriate subroutine. Moreover, if you use (B.20) you can store the function evaluations (and derivatives) outside of the assembly loop; thus, evaluating (B.18) only requires summing over j (for each quadrature point).

Remark B.1. *The form* (B.20) *is especially convenient because it matches the usual form of a finite element basis function expansion on $\widehat{T}$. This means that you can potentially use any finite element space for approximating the spatial domain. Other forms of mapping can be used, such as NURBS.*

The exact details of implementation are not really important, as long as you end up with something like (B.18). Just make sure the implementation is automatic.

Evaluating Finite Element Basis Functions and Derivatives

We proceed to use the same idea for computing, say mappings like in (B.9), and we will take advantage of the pre-computed data in (B.18). First, we need some more basic data structures to hold the basis function evaluations:

$$\begin{aligned} \texttt{double f\_k[Nb].value[Nq]} &= \text{ array,} \\ \texttt{double f\_k[Nb].grad\_j[Nq]} &= \text{ array for each } \texttt{j} = 1,2, \end{aligned} \tag{B.21}$$

where `Nb` is the number of basis functions and `k` indexes the vector component of the basis function $\mathbf{e}_k \cdot \mathbf{f}_i \equiv$`f_k[i]`. So we have:

A. `f_k[i].value` contains the evaluation of $\mathbf{e}_k \cdot \mathbf{f}_i(\mathbf{\Phi}(\hat{\mathbf{x}})) = \mathbf{e}_k \cdot \hat{\mathbf{f}}_i(\hat{\mathbf{x}})$ at each quadrature point. Note that $\hat{\mathbf{f}}_i$ is a known function.

B. `f_k[i].grad_j` contains the evaluation of $(\nabla_{\mathbf{x}}\mathbf{e}_k \cdot \mathbf{f}_i) \circ \mathbf{\Phi}(\hat{\mathbf{x}})$ at each quadrature point (recall (B.9)).

Even though the calculation of (B.9) may seem trivial, it is best left to the symbolic toolbox to avoid errors. One way to do this is to write (B.9) (in MATLAB syntax)

$$\left[\begin{array}{cc} \texttt{fhat\_k\_grad\_1} & \texttt{fhat\_k\_grad\_2} \end{array}\right] * \left[\begin{array}{cc} \texttt{Jinv\_11} & \texttt{Jinv\_12} \\ \texttt{Jinv\_21} & \texttt{Jinv\_22} \end{array}\right] \tag{B.22}$$

where each term is a symbolic variable. You then let the symbolic toolbox handle the algebra when expanding the expression. You can also do an easy search and replace of each text string before generating the code snippet (the symbolic toolbox can do this).

The automation may seem trivial so far, but it connects with Section B.1.3 that greatly extends the power of code generation.

B.1.3 Evaluating Forms

Evaluating bilinear forms can be complicated for general problems, especially if there are tensor valued variables. So we want to open the door for using a symbolic toolbox that handles the algebra of vectors and matrices for us. Therefore, we need a framework to do this.

Basic Interface

We will assume that the finite element functions are just scalars or tensor products of scalars (for now). Let $\mathbf{v} : \mathbb{R}^2 \to \mathbb{R}^2$, and suppose we have a OOP class that implements various symbolic operations with $\mathbf{v}$. Let `vv` be the code syntax for this object. The idea is to define various methods for the class that output the appropriate symbolic representation. For instance,

$$\texttt{vv.val} = \left[\begin{array}{c} \texttt{vv\_t1} \\ \texttt{vv\_t2} \end{array}\right] \equiv \mathbf{v} = \left[\begin{array}{c} \mathbf{e}_1 \cdot \mathbf{v} \\ \mathbf{e}_2 \cdot \mathbf{v} \end{array}\right], \tag{B.23}$$

where `vv_t1` is a symbolic variable for the first (tuple) component and `vv_t2` is for the second component. The "`t`" is for tuple. Similarly,

$$\begin{aligned} \texttt{vv.grad} = \left[\begin{array}{cc} \texttt{vv\_t1\_grad\_1} & \texttt{vv\_t1\_grad\_2} \\ \texttt{vv\_t2\_grad\_1} & \texttt{vv\_t2\_grad\_2} \end{array}\right] \equiv \\ \nabla \mathbf{v} = \left[\begin{array}{cc} (\mathbf{e}_1 \cdot \nabla)(\mathbf{e}_1 \cdot \mathbf{v}) & (\mathbf{e}_2 \cdot \nabla)(\mathbf{e}_1 \cdot \mathbf{v}) \\ (\mathbf{e}_1 \cdot \nabla)(\mathbf{e}_2 \cdot \mathbf{v}) & (\mathbf{e}_2 \cdot \nabla)(\mathbf{e}_2 \cdot \mathbf{v}) \end{array}\right]. \end{aligned} \tag{B.24}$$

One can then compute (with the symbolic toolbox) quantities like $\mathbf{v} \cdot \mathbf{u}$ and $\nabla \mathbf{v} : \nabla \mathbf{u}$, where $\mathbf{u}$ is similarly defined. You can then identify the individual variables in the resulting expression. Now you know what needs to be computed, e.g. the second component of the gradient of the first (tuple) component of the function $\texttt{vv\_t1\_grad\_2} \equiv (\mathbf{e}_2 \cdot \nabla)(\mathbf{e}_1 \cdot \mathbf{v})$. Once you have evaluated those terms (at the quadrature points), then the symbolic toolbox can automatically generate a code snippet to evaluate the expression.

More Tricks

There are other things one can take advantage of here. You can check for symmetry of the variables in the expression, which could possibly allow you to copy sub-matrices of the FE matrix over (more efficient). You can determine if sub-matrices of the FE matrix are identical (possibly with a transpose); again, it is more efficient to just copy over. You can isolate the different sub-matrices by setting some of the variables in the expression to 0; the part left over is a sub-matrix. You can easily add more operators, such as

$$\begin{aligned} \texttt{vv.div} = \texttt{vv\_t1\_grad\_1} + \texttt{vv\_t2\_grad\_2} \equiv \\ \nabla \cdot \mathbf{v} = (\mathbf{e}_1 \cdot \nabla)(\mathbf{e}_1 \cdot \mathbf{v}) + (\mathbf{e}_2 \cdot \nabla)(\mathbf{e}_2 \cdot \mathbf{v}). \end{aligned} \tag{B.25}$$

You can even add the Hessian operator. You can also pass integer arguments to `val` and `grad` in order to pick out particular components.

Something Harder

Let $\mathbf{v}, \mathbf{u}$ be 2-D vectors (like an $H(\text{div})$ element) and define $\mathbf{g} = \mathbf{v} \otimes \mathbf{u}$ which is a 2-tuple of 2-D vectors. Then the `val` operator gives

$$\texttt{vg.val} = \begin{bmatrix} \texttt{vg\_t1\_v1} & \texttt{vg\_t2\_v1} \\ \texttt{vg\_t1\_v2} & \texttt{vg\_t2\_v2} \end{bmatrix} \equiv \mathbf{g} = \begin{bmatrix} \mathbf{e}_1 \cdot \mathbf{g}_1 & \mathbf{e}_1 \cdot \mathbf{g}_2 \\ \mathbf{e}_2 \cdot \mathbf{g}_1 & \mathbf{e}_2 \cdot \mathbf{g}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{e}_1 \cdot \mathbf{v} & \mathbf{e}_1 \cdot \mathbf{u} \\ \mathbf{e}_2 \cdot \mathbf{v} & \mathbf{e}_2 \cdot \mathbf{u} \end{bmatrix}, \tag{B.26}$$

where `vg_t1` corresponds to $\mathbf{g}_1$ and the extra `_v1` notation indicates to take the first *vector* component of the function. Hence, we make a distinction between tuples and vectors. A vector is some intrinsically defined object, whereas a tuple is just a tensor product of objects. In this example, the 2 objects in the tuple are of identical type, but they don't have to be.

Continuing, if we apply `grad`, then `vg.grad` would give a multi-matrix. As comparison, `vg.div` gives

$$\begin{aligned} &\texttt{vg.div} = \left[\begin{array}{cc} \texttt{vg\_t1\_v1\_grad\_1} + \texttt{vg\_t1\_v2\_grad\_2}, & \texttt{vg\_t2\_v1\_grad\_1} + \texttt{vg\_t2\_v2\_grad\_2} \end{array} \right] \\ &\equiv \nabla \cdot \mathbf{g} = \left[\begin{array}{cc} \nabla \cdot \mathbf{g}_1, & \nabla \cdot \mathbf{g}_2 \end{array} \right]. \end{aligned} \tag{B.27}$$

B.2 Local-to-Global Transformations

We now discuss different types of local mappings (transformations) that one typically uses in finite element methods. Recall the discussion from Section B.1.1.

B.2.1 Natural Mapping For Scalar Valued Functions

As we saw earlier, a scalar function f can be defined in terms of $\hat{f}$ by the mapping

$$f(\mathbf{x}) = \hat{f}(\mathbf{\Phi}^{-1}(\mathbf{x})), \quad \text{for all } \mathbf{x} \in T. \tag{B.28}$$

Furthermore, we can use a similar mapping for a vector valued function $\mathbf{f}$ if it can be written as a tensor product of scalar functions, i.e. map each vector component just like in the above equation.

B.2.2 Transformations of Vectors

For various reasons, we need different transformations for vector valued functions that are not tensor products of scalar functions.

Transformation For $H(\mathrm{div})$

For vector functions $\mathbf{v}$ in $H(\mathrm{div}, \Omega)$ [4], we must use the following transformation (Piola)

$$\mathbf{v}(\mathbf{x}) = \frac{1}{\det(\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}}))} [\nabla_{\hat{\mathbf{x}}} \mathbf{\Phi}(\hat{\mathbf{x}})] \hat{\mathbf{v}}(\hat{\mathbf{x}}), \quad \text{for all } \mathbf{x} \in T, \tag{B.29}$$

where $\hat{\mathbf{x}} = \mathbf{\Phi}^{-1}(\mathbf{x})$ and $\hat{\mathbf{v}} : \widehat{T} \to \mathbb{R}^d$ is a (local) vector basis function in $H(\mathrm{div}, \widehat{T})$. This transformation preserves the divergence and the normal vector on ∂T.

Derivatives of Piola Transformation

Let $J_{ij}(\hat{\mathbf{x}}) := \partial_{\hat{x}_j} \left(\mathbf{\Phi}(\hat{\mathbf{x}}) \cdot \mathbf{e}_i \right)$; so J is a square matrix. Then (B.29) becomes

$$\mathbf{v}(\mathbf{x}) \cdot \mathbf{e}_i = \left((\det J)^{-1} \circ \mathbf{\Phi}^{-1}(\mathbf{x}) \right) \sum_{j=1}^{N} \left[J_{ij} \circ \mathbf{\Phi}^{-1}(\mathbf{x}) \right] \left(\hat{\mathbf{v}} \circ \mathbf{\Phi}^{-1}(\mathbf{x}) \cdot \mathbf{e}_j \right). \tag{B.30}$$

Differentiating (B.30) gives

$$\begin{aligned} \partial_{x_k} \mathbf{v}(\mathbf{x}) \cdot \mathbf{e}_i = {} & \partial_{x_k} \left((\det J)^{-1} \circ \mathbf{\Phi}^{-1}(\mathbf{x}) \right) \sum_{j=1}^{N} \left[J_{ij} \circ \mathbf{\Phi}^{-1}(\mathbf{x}) \right] \left(\hat{\mathbf{v}} \circ \mathbf{\Phi}^{-1}(\mathbf{x}) \cdot \mathbf{e}_j \right) + \\ & \left((\det J)^{-1} \circ \mathbf{\Phi}^{-1}(\mathbf{x}) \right) \sum_{j=1}^{N} \left(\partial_{x_k} [J_{ij} \circ \mathbf{\Phi}^{-1}(\mathbf{x})] \right) \left(\hat{\mathbf{v}} \circ \mathbf{\Phi}^{-1}(\mathbf{x}) \cdot \mathbf{e}_j \right) + \\ & \left((\det J)^{-1} \circ \mathbf{\Phi}^{-1}(\mathbf{x}) \right) \sum_{j=1}^{N} [J_{ij} \circ \mathbf{\Phi}^{-1}(\mathbf{x})] \partial_{x_k} \left(\hat{\mathbf{v}} \circ \mathbf{\Phi}^{-1}(\mathbf{x}) \cdot \mathbf{e}_j \right). \end{aligned} \tag{B.31}$$

Proceeding in steps gives

$$\begin{aligned} \partial_{x_k} \left((\det J)^{-1} \circ \mathbf{\Phi}^{-1}(\mathbf{x}) \right) &= - \left((\det J)^{-2} \circ \mathbf{\Phi}^{-1}(\mathbf{x}) \right) \partial_{x_k} (\det J \circ \mathbf{\Phi}^{-1}(\mathbf{x})) \\ &= - \left((\det J)^{-2} \circ \mathbf{\Phi}^{-1}(\mathbf{x}) \right) \sum_m \left(\partial_{\hat{x}_m} \det J \right) \left(\partial_{x_k} \mathbf{\Phi}^{-1}(\mathbf{x}) \cdot \mathbf{e}_m \right), \end{aligned} \tag{B.32}$$

which turns into

$$\begin{aligned} & \partial_{x_k} \left((\det J)^{-1} \circ \mathbf{\Phi}^{-1} \right) \circ \mathbf{\Phi}(\hat{\mathbf{x}}) \\ & \quad = -(\det J(\hat{\mathbf{x}}))^{-2} \sum_m \left(\partial_{\hat{x}_m} \det J(\hat{\mathbf{x}}) \right) \left(\partial_{x_k} \mathbf{\Phi}^{-1}(\mathbf{x}) \cdot \mathbf{e}_m \right) \circ \mathbf{\Phi}(\hat{\mathbf{x}}), \\ & \quad = -(\det J(\hat{\mathbf{x}}))^{-2} \sum_m \left(\partial_{\hat{x}_m} \det J(\hat{\mathbf{x}}) \right) J^{-1}_{mk}(\hat{\mathbf{x}}). \end{aligned} \tag{B.33}$$

Next, we have

$$\partial_{x_k}(J_{ij}\circ\mathbf{\Phi}^{-1}(\mathbf{x})) = \sum_m \left((\partial_{\hat{x}_m}J_{ij})\circ\mathbf{\Phi}^{-1}(\mathbf{x})\right)\left(\partial_{x_k}\mathbf{\Phi}^{-1}(\mathbf{x})\cdot\mathbf{e}_m\right) \tag{B.34}$$

which turns into

$$\partial_{x_k}(J_{ij}\circ\mathbf{\Phi}^{-1})\circ\mathbf{\Phi}(\hat{\mathbf{x}}) = \sum_m \left(\partial_{\hat{x}_m}J_{ij}(\hat{\mathbf{x}})\right)J^{-1}_{mk}(\hat{\mathbf{x}}). \tag{B.35}$$

Finally, we have

$$\partial_{x_k}\left(\hat{\mathbf{v}}\circ\mathbf{\Phi}^{-1}(\mathbf{x})\cdot\mathbf{e}_j\right) = \sum_m(\partial_{\hat{x}_m}\hat{\mathbf{v}}\cdot\mathbf{e}_j)\circ\mathbf{\Phi}^{-1}(\mathbf{x})\left(\partial_{x_k}\mathbf{\Phi}^{-1}(\mathbf{x})\cdot\mathbf{e}_m\right) \tag{B.36}$$

which turns into

$$\partial_{x_k}\left(\mathbf{e}_j\cdot\hat{\mathbf{v}}\circ\mathbf{\Phi}^{-1}\right)\circ\mathbf{\Phi}(\hat{\mathbf{x}}) = \sum_m(\partial_{\hat{x}_m}\hat{\mathbf{v}}(\hat{\mathbf{x}})\cdot\mathbf{e}_j)J^{-1}_{mk}(\hat{\mathbf{x}}). \tag{B.37}$$

Going back to (B.31), and plugging the above relations in, we obtain

$$\begin{aligned}(\mathbf{e}_i\cdot\partial_{x_k}\mathbf{v})\circ\mathbf{\Phi}(\hat{\mathbf{x}}) = &\left[-(\det J(\hat{\mathbf{x}}))^{-2}\sum_m\left(\partial_{\hat{x}_m}\det J(\hat{\mathbf{x}})\right)J^{-1}_{mk}(\hat{\mathbf{x}})\right]\sum_{j=1}^N\left[J_{ij}(\hat{\mathbf{x}})\right](\mathbf{e}_j\cdot\hat{\mathbf{v}}(\hat{\mathbf{x}}))\\ &+(\det J(\hat{\mathbf{x}}))^{-1}\sum_{j=1}^N\left[\sum_m\left(\partial_{\hat{x}_m}J_{ij}(\hat{\mathbf{x}})\right)J^{-1}_{mk}(\hat{\mathbf{x}})\right]\mathbf{e}_j\cdot\hat{\mathbf{v}}(\hat{\mathbf{x}})\\ &+(\det J(\hat{\mathbf{x}}))^{-1}\sum_{j=1}^N J_{ij}(\hat{\mathbf{x}})\left[\sum_m(\partial_{\hat{x}_m}\hat{\mathbf{v}}(\hat{\mathbf{x}})\cdot\mathbf{e}_j)J^{-1}_{mk}(\hat{\mathbf{x}})\right].\end{aligned} \tag{B.38}$$

Note the classic relation

$$\partial_{\hat{x}_j}\det J = (\det J)\text{trace}\left([\partial_{\hat{x}_j}J]J^{-1}\right) = \det J\sum_k\sum_l(\partial_{\hat{x}_j}J_{kl})J^{-1}_{lk}, \tag{B.39}$$

and $\partial_{\hat{x}_m}J_{ij} = \partial_{\hat{x}_j}J_{im}$ by the definition of J and the commutativity of derivatives. Then, (B.38) simplifies to

$$\begin{aligned}(\mathbf{e}_i\cdot\partial_{x_k}\mathbf{v})\circ\mathbf{\Phi}(\hat{\mathbf{x}}) = &\left[-(\det J(\hat{\mathbf{x}}))^{-1}\sum_m\text{trace}\left([\partial_{\hat{x}_m}J]J^{-1}\right)J^{-1}_{mk}(\hat{\mathbf{x}})\right](\mathbf{e}_i\cdot J\hat{\mathbf{v}})\\ &+(\det J(\hat{\mathbf{x}}))^{-1}\sum_{j=1}^N\left[\sum_m\left(\partial_{\hat{x}_m}J_{ij}(\hat{\mathbf{x}})\right)J^{-1}_{mk}(\hat{\mathbf{x}})\right]\mathbf{e}_j\cdot\hat{\mathbf{v}}(\hat{\mathbf{x}})\\ &+(\det J(\hat{\mathbf{x}}))^{-1}\sum_{j=1}^N J_{ij}(\hat{\mathbf{x}})\left[\sum_m(\partial_{\hat{x}_m}\hat{\mathbf{v}}(\hat{\mathbf{x}})\cdot\mathbf{e}_j)J^{-1}_{mk}(\hat{\mathbf{x}})\right].\end{aligned} \tag{B.40}$$

The divergence simplifies dramatically. Set $i = k$, sum, and commute derivatives:

$$\begin{aligned}(\nabla_{\mathbf{x}} \cdot \mathbf{v}) \circ \mathbf{\Phi}(\hat{\mathbf{x}}) = &-(\det J(\hat{\mathbf{x}}))^{-1} \sum_m \operatorname{trace}\left([\partial_{\hat{x}_m} J] J^{-1}\right) \sum_j \sum_k J^{-1}_{mk}(\hat{\mathbf{x}}) J_{kj}(\mathbf{e}_j \cdot \hat{\mathbf{v}}(\hat{\mathbf{x}})) \\ &+ (\det J(\hat{\mathbf{x}}))^{-1} \sum_j \sum_k \left[\sum_m \left(\partial_{\hat{x}_j} J_{km}(\hat{\mathbf{x}})\right) J^{-1}_{mk}(\hat{\mathbf{x}})\right] (\mathbf{e}_j \cdot \hat{\mathbf{v}}(\hat{\mathbf{x}})) \\ &+ (\det J(\hat{\mathbf{x}}))^{-1} \sum_j \sum_m \sum_k J^{-1}_{mk}(\hat{\mathbf{x}}) J_{kj}(\hat{\mathbf{x}}) (\partial_{\hat{x}_m} \hat{\mathbf{v}}(\hat{\mathbf{x}}) \cdot \mathbf{e}_j).\end{aligned} \tag{B.41}$$

This leads to

$$\begin{aligned}(\nabla_{\mathbf{x}} \cdot \mathbf{v}) \circ \mathbf{\Phi}(\hat{\mathbf{x}}) &= -(\det J(\hat{\mathbf{x}}))^{-1} \sum_m \operatorname{trace}\left([\partial_{\hat{x}_m} J] J^{-1}\right) \sum_j \delta_{mj} (\mathbf{e}_j \cdot \hat{\mathbf{v}}(\hat{\mathbf{x}})) \\ &\quad + (\det J(\hat{\mathbf{x}}))^{-1} \sum_j \operatorname{trace}\left([\partial_{\hat{x}_j} J] J^{-1}\right) (\mathbf{e}_j \cdot \hat{\mathbf{v}}(\hat{\mathbf{x}})) \\ &\quad + (\det J(\hat{\mathbf{x}}))^{-1} \sum_j \sum_m \delta_{mj} (\partial_{\hat{x}_m} \hat{\mathbf{v}}(\hat{\mathbf{x}}) \cdot \mathbf{e}_j) \\ &= (\det J(\hat{\mathbf{x}}))^{-1} \sum_j (\partial_{\hat{x}_j} \hat{\mathbf{v}}(\hat{\mathbf{x}}) \cdot \mathbf{e}_j) = (\det J(\hat{\mathbf{x}}))^{-1} \nabla_{\hat{\mathbf{x}}} \cdot \hat{\mathbf{v}}(\hat{\mathbf{x}}),\end{aligned} \tag{B.42}$$

which is true whether $\mathbf{\Phi}$ is linear or not.

References

[1] J. Alberty, C. Carstensen, and S. A. Funken. Remarks around 50 lines of matlab: Short finite element implementation. *Numerical Algorithms*, 20:117–137, 1998.

[2] D. Braess. *Finite Elements: Theory, Fast Solvers, and Applications in Solid Mechanics*. Cambridge University Press, 2nd edition, 2001.

[3] S. C. Brenner and L. R. Scott. *The Mathematical Theory of Finite Element Methods*, volume 15 of *Texts in Applied Mathematics*. Springer, New York, NY, 3rd edition, 2008.

[4] F. Brezzi and M. Fortin. *Mixed and Hybrid Finite Element Methods*. Springer-Verlag, New York, NY, 1991.

[5] P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*. Classics in Applied Mathematics. SIAM, Philadelphia, PA, 2nd edition, 2002. ISBN: 978-0898715149.

[6] G. A. Holzapfel. *Nonlinear Solid Mechanics: A Continuum Approach For Engineering*. John Wiley & Sons, Inc., 2000.

[7] T. J. R. Hughes. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Dover Publications, Mineola, NY, 1st edition, 2000.

[8] A. Logg. Automating the finite element method. *Arch. Comput. Methods Eng.*, 14(2):93–138, 2007.

[9] A. Logg and G. N. Wells. Dolfin: Automated finite element computing. *ACM Trans. Math. Softw.*, 37(2):1–28, 2010.

[10] M.-C. Rivara. Lepp-bisection algorithms, applications and mathematical properties. *Applied Numerical Mathematics*, 59(9):2218 – 2235, 2009. Second Chilean Workshop on Numerical Analysis of Partial Differential Equations (WONAPDE 2007).

[11] R. M. Temam and A. M. Miranville. *Mathematical Modeling in Continuum Mechanics*. Cambridge University Press, 2nd edition, 2005.

[12] C. A. Truesdell. *A First Course in Rational Continuum Mechanics*. Pure and applied mathematics, a series of monographs and textbooks. Academic Press, 1976.

[13] S. W. Walker. Tetrahedralization of isosurfaces with guaranteed-quality by edge rearrangement (TIGER). *SIAM Journal on Scientific Computing*, 35(1):A294–A326, 2013.