MORIS
Mesh Generation

Last updated:
March 8, 2023

Preface This document serves as a user guide to using MORIS' mesh generation abilities to output body-fitted finite element meshes alongside extraction operators projecting the basis of the foreground mesh into a tensor-product B-spline basis. The mesh data generated is intended for *interpolation-based immersed finite element analysis* as introduced by Fromm et al. [1].

The document is split into two parts. The first part, Chapter 1, provides an overview of some theoretical and algorithmic aspects, the second part, Chapter 2, lays out all currently supported features and how to use them with an XML input file.

This guide assumes that MORIS has already been installed and compiled. An installation guide is provided in the MORIS GitHub repository: github.com/kkmaute/moris (🔗). Note that the features outlined in this document currently undergo expansion and changes. Please check back regularly for any updates.

If you have questions, comments, feature requests, or find bugs, please contact Nils Wunsch ✉

Topology Optimiation Research Group
Aerospace Mechanics Research Center (AMReC)
Ann & H. J. Smead Department of Aerospace Engineering Sciences
University of Colorado Boulder

Contents

1 Overview **3**
1.1 Geometry 4
1.2 Background Basis and Enrichment 5
1.3 Foreground Mesh 10
1.4 Lagrange Extraction 12

2 User Guide **14**
2.1 Geometry 15
2.2 Background Meshes 20
2.3 Foreground Mesh 21
2.4 Extraction Operators 24

Bibliography **26**

1 Overview

The goal of this chapter is to give an overview of the theoretical and algorithmic pieces needed to understand the inputs for, and outputs of MORIS' mesh generation workflow. As mentioned in the foreword, the output data is intended to perform interpolation-based immersed finite element analysis, the theory for which, with the exception of a brief introduction to Lagrange extraction, will not be discussed in this document. Instead the reader is advised to consult the publication by Fromm et al. [1]

MORIS ("Multi-physics Optimization Research and Innovation System") is a standalone finite element and topology optimization software framework. The analysis method utilized inside MORIS itself is a quadrature-based immersed isogeometric method. The interpolation function basis is provided by a rectangular tensor-product B-spline mesh, the **background mesh**, that can be hierarchically refined. Next, geometries are immersed into said mesh. The geometry is described implicitly by level-set fields $\Phi(\boldsymbol{x})$ and their iso-contours $\Phi(\boldsymbol{x}) = c$. A quadrature rule for evaluating the discretized weak form is found by tessellating, or "*decomposing*", the background mesh's elements intersected by the iso-contours into triangles or tetrahedrons whose Gauss points and weights provide the integration rule. Additionally, the background basis is Heaviside enriched to solve problems involving multiple materials. For the interested reader, the isogeometric analysis framework inside MORIS is explained in more detail by Noël et al. [2] and Schmidt et al. [3].

The set of non-intersected background elements and triangular elements within intersected elements together form a body-fitted mesh which from here on will be referred to as the **foreground mesh**. This body-fitted foreground mesh and its known relation to the background mesh together form the mesh information needed for interpolation-based immersed analysis. Hence, the mesh generation abilities of MORIS can directly be exploited to provide all mesh information needed to perform interpolation-based immersed finite element analysis.

The meshing apparatus inside MORIS consists of three modules: a hierarchical B-spline mesh generator called HMR, a geometry engine (GEN), and an extended finite element tool kit (XTK). To obtain the desired meshes, the inputs for these three modules need to be configured. More specifically, HMR needs to be fed what the size, dimensions, and order of its tensor-product meshes is, and how they're to be refined. GEN needs level-set fields to determine interface locations and material membership. XTK works based off the information provided by the other modules, apart from minor configuration input. A chart of the workflow is shown below.

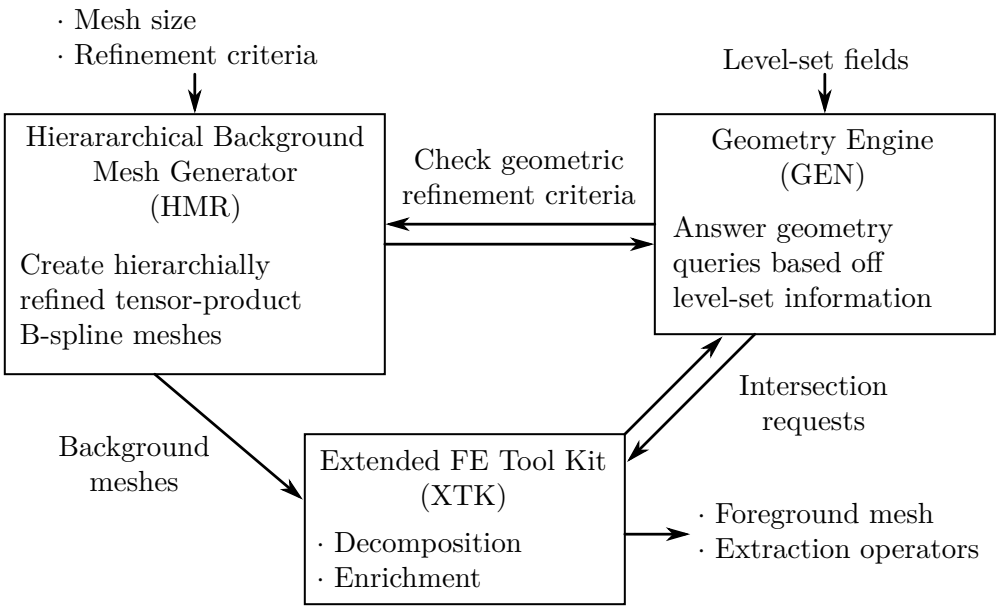

Figure 1.1: Workflow for mesh generation inside MORIS

The remainder of this chapter is structured as follows. First, a short introduction on how geometries can be defined with level-set fields will be given. Next, the interpolation basis used by MORIS is defined alongside the enrichment strategy used. Lastly, the algorithm for generating the foreground mesh is shown and how the extraction operators are computed from it.

Note: The features currently supported in the pure mesh generation workflow do not include all features supported by MORIS internally, but are undergoing expansion to do so. Until then, some parts of this chapter may seem unnecessary.

1.1 Geometry

As mentioned above, geometry in MORIS is described implicitly by a level-set field $\Phi_i(\boldsymbol{x})$ and a threshold c, which is assumed to be $c = 0$ from here on, determining the locations of interfaces. This leads to a natural definition of what is *inside* and *outside* a given geometry.

$$\Phi_i(\boldsymbol{x}) \begin{cases} < 0 & \boldsymbol{x} \notin \Omega_i \\ = 0 & \boldsymbol{x} \in \partial\Omega_i \\ > 0 & \boldsymbol{x} \in \Omega_i \end{cases} \tag{1.1}$$

If multiple level-set fields, and therefore geometries, are introduced, each point $\boldsymbol{x}$ is defined to be either inside or outside with respect to each of the geometries. Each region of points that is a certain combination of inside/outside with respect to each of the geometries is called a *phase*. This leads to there being up to $n_p = 2^{n_{LSF}}$ phases inside a given domain, when n_{LSF} geometries are defined. *Material* sub-domains can simply be defined by merging these phases, as illustrated in Figure 1.2

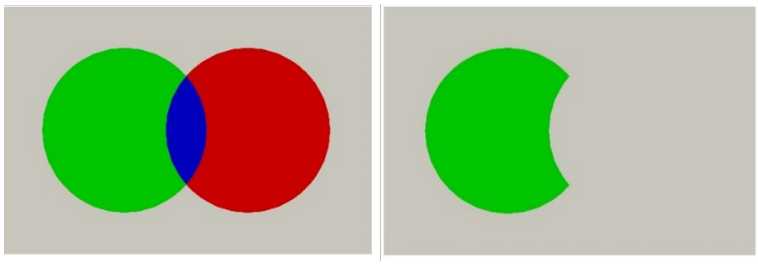

Figure 1.2: Introducing two discs as geometries into an ambient domain results in four phases (marked in different colors). Material domains can be defined by merging phases.

A particularly notable type of level-set field is a so-called signed-distance field, which every point has a value whose magnitude corresponds to the minimum distance to an interface. Any geometry definition that has a notion of "inside" and "outside" can be converted to a signed-distance field and therefore be used with MORIS, if a pre-processor for the conversion exists.

1.2 Background Basis and Enrichment

Tensor-Product B-spline Basis B-spline basis functions are used to define the function basis of the background mesh. The B-spline basis functions $N_{i,p}$ of polynomial order p are defined by a knot vector $\{\xi_i\}_{i=1}^{n}$ containing a set of control points, and the Cox-de Boor recursive formula. The definition of the i-th basis function starts with piece-wise constant parts for zeroth order $p = 0$ B-spline basis function.

$$N_{i,0}(\xi) = \begin{cases} 1 \; if \; \xi_i \leq \xi < \xi_{i+1} \\ 0 \; else \end{cases} \tag{1.2}$$

B-spline basis functions of higher order can be constructed by evaluating the following equation recursively:

$$N_{i,p}(\xi) = \frac{\xi - \xi_i}{\xi_{i+p} - \xi_i} N_{i,p-1}(\xi) + \frac{\xi_{i+p+1} - \xi}{\xi_{i+p+1} - \xi_{i+1}} N_{i+1,p-1}(\xi) \tag{1.3}$$

This recursive definition leads to a piece-wise polynomial definition of the basis functions.

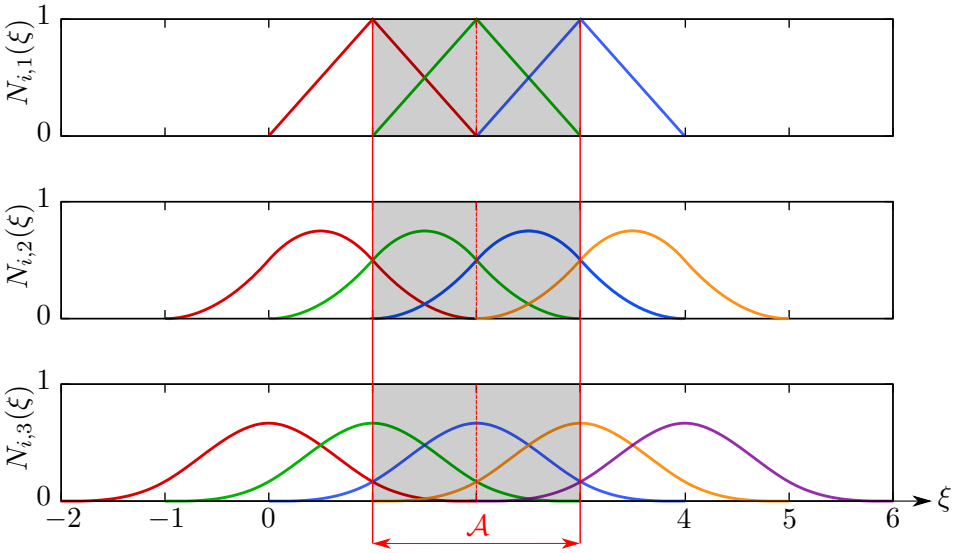

Figure 1.3: C^{p-1} continuous B-spline basis functions constructed from unique and uniform knot vectors or orders $p = 1, 2, 3$ defining an interpolation basis for the computational domain $\mathcal{A} = [1, 3]$

Using a knot vector of unique and equispaced control points ξ_i results in C^{p-1} **continuous** basis functions that are translated and repeated. The intervals between the control points can be considered elements (i.e. **the control points act as mesh lines**), as in each of these regions a unique set of $p + 1$ basis functions interpolate into. With exception of the first and last p knot intervals, called the *padding*, each of these sets of basis functions form a complete polynomial basis and a partition of unity. Hence, the padding is not part of the ambient domain $\mathcal{A}$ on which a complete interpolation basis is defined. The polynomial basis and the domain they cover is shown in Figure 1.3. A notable feature of basis functions defined this way is their **increasing support size for increasing polynomial orders** p.

The definition of the basis is extended into multiple (d) spatial dimensions by forming tensor products from these basis functions.

$$N_{i_1,\cdots,i_d}(\boldsymbol{\xi}) = \prod_{d}^{n_d} N_{i_d}(\xi_d) \tag{1.4}$$

The basic tensor-product B-spline basis used inside MORIS is constructed from an equispaced grid whose mesh lines represent the unique control points used to define the B-spline basis functions in each spatial dimension.

Hierarchical Refinement H-refinement could be performed by inserting additional knot points and therefore mesh lines. However, this would the additional mesh lines created would propagate throughout the whole domain. Instead, a hierarchical basis definition is employed to perform local refinement of the tensor-product B-spline background mesh.

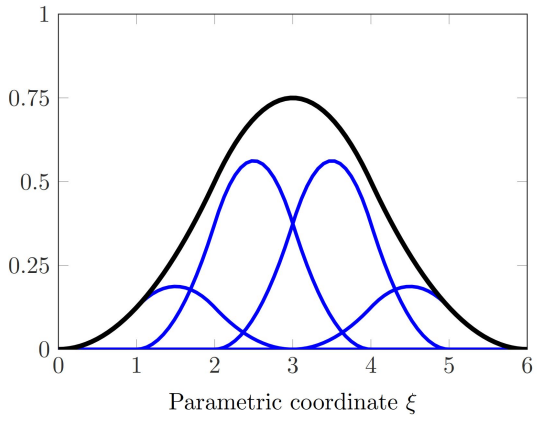

Figure 1.4: The black basis function is a sum of the blue translated, scaled, and weighted versions of itself.

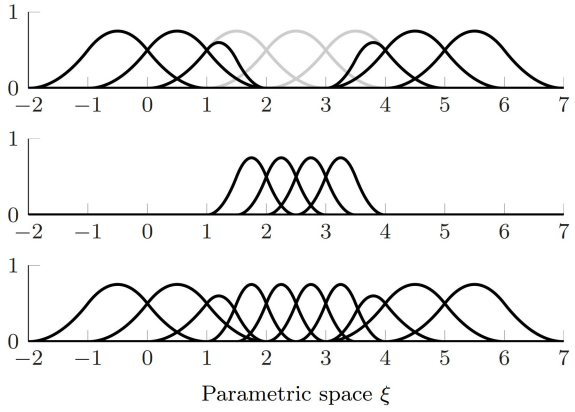

Figure 1.5: Construction of a 1D hierarchical B-spline basis.

Each B-spline basis function can be represented as a sum of translated, scaled and weighted versions of itself, like shown in Figure 1.4. These *finer* versions of a *coarse* basis function are constructed from the coarse knot vectors that have additional knots inserted at the midpoints of their knot intervals.

This allows a coarse basis function at a refinement level l to be replaced by a weighted set of basis functions of refinement level $l+1$. The underlying tensor-product elements can be understood to undergo **quadtree or octree refinement** due to the introduction of additional mesh lines splitting them into four or eight elements in 2D and 3D respectively.

The domain of the elements on a given *level* l marked for refinement is denoted as Ω^{l+1}. On this domain Ω^{l+1}, e.g. the single element domain $\Omega^1 = [2,3]$ shown in Figure 1.5, a set of finer basis functions interpolating into it is constructed on the refined level $l+1$. Assuming

that all coarse basis functions N^l are represented by a weighted sum of finer basis functions N^{l+1}, the contribution of the finer basis functions actually constructed can be subtracted from them. This leads to either **truncation** or removal of the coarse basis functions as shown in the top row of Figure 1.5. The definition of the resulting truncated coarse basis functions can be stated as:

$$trunc^{l+1}(N^l) = \sum_{j:N_j \not\subseteq \Omega^{l+1}} \alpha_j N_j^{l+1} \tag{1.5}$$

The new, locally refined basis is the sum of the constructed fine basis and truncated coarse basis, as shown in the third row of Figure 1.5. The truncated hierarchical basis can be stated as a sum of recursively defined components:

$$\begin{aligned} \mathcal{V}^h &= \bigcup_{l=0}^{L} \mathcal{V}^l \\ where \quad \mathcal{V}^{l+1} &= \left\{ trunc^{l+1}(N) \mid B \in \mathcal{V}^l \wedge supp(N) \not\subseteq \Omega^{l+1} \right\} \\ &\cup \left\{ N \in \mathcal{V}^{l+1} \mid supp(N) \subseteq \Omega^{l+1} \right\} \end{aligned} \tag{1.6}$$

where $\mathcal{V}^0$ is the tensor-product B-spline basis described at the beginning of this section.

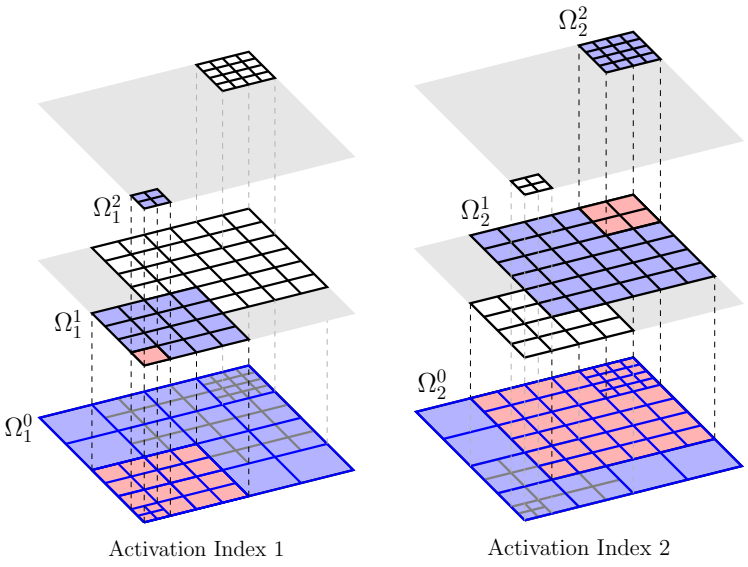

Figure 1.6: Definition of multiple background mesh grids by marking sub-domains Ω_i^l for refinement.

MORIS enables multiple hierarchically refined B-spline meshes to be created by refining a single mesh grid in different ways, like shown in Figure 1.6. Additionally, B-spline basis functions of various polynomial degrees p can then be defined on the resulting grids.

Enrichment Individual B-spline elements may contain multiple materials within them. Meanwhile, the state variable fields within these materials are decoupled. This is achieved by employing a generalized Heaviside enrichment strategy.

For each basis function $N_j(\boldsymbol{x})$ a set of degrees of freedom (DoFs) $\left\{d_j^e\right\}_{e=1}^{n_e}$ are introduced. Indicator functions $\psi_j^e(\boldsymbol{x})$ provide the information whether a combination of DoF and basis function is active or $= 0$ at a given point $\boldsymbol{x}$.

$$u^h(\boldsymbol{x}) = \sum_{j=1}^{n_B} \sum_{e=1}^{n_{e,j}} \psi_j^e(\boldsymbol{x}) \cdot N_j(\boldsymbol{x}) \cdot d_j^e \tag{1.7}$$

The index e would traditionally correspond to the material index of a given point. However, for small scale features, like the protrusion shown in Figure 1.7 (a), artificial coupling in the solution between the material regions on either side of it could still be observed due to the same basis B interpolating into either separated region.

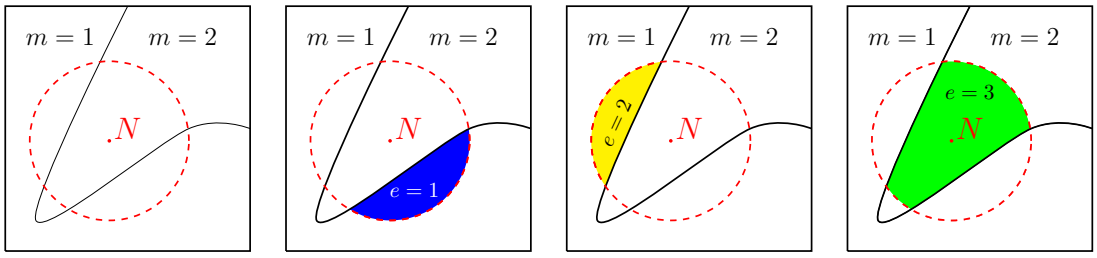

Figure 1.7: Basis function enrichment for a two material problem. The support of a basis function B with arbitrary support is illustrated in red. The basis function is enriched three times based on the three topologically disconnected regions of the two materials it is interpolating into.

To alleviate this issue, MORIS considers the material layout within a given basis functions support. Enrichment levels e and corresponding DoFs d_j^e are created for each disconnected material region within the support. This implies that a different set of non-zero B-spline basis functions interpolates into a point on a material interface depending not only on the material index with respect to which it is considered, but also the material layout in its vicinity. An example is provided by Figure 1.7.

1.3 Foreground Mesh

As mentioned in the overview of MORIS, the foreground mesh is generated by decomposing the background mesh, or more precisely one of the background grids. This process will be referred to as the *decomposition*.

The Decomposition consists of the following steps which are also visualized in Figure 1.8

- The first step is to identify which particular background elements are intersected and need to be decomposed. To do so, the values of all level-set fields $\Phi_i(\boldsymbol{x})$ are checked at the vertices, i.e. the "corners", of the background element. If a sign change is detected between level-set values, the element is marked for decomposition. Note, due to this particular way of intersection detection, material regions completely contained within a single background element, or protrusions through individual facets may not be detected.
- In a next step, each intersected background element is subdivided into triangles or tetrahedrons, depending on dimensionality. This step will be referred to as the *regular subdivision*.
- The edges of the triangular elements created in the regular subdivision can again be checked for sign changes in the level-set fields. Depending on which edges of a given triangle or tetrahedron are intersected, the element can be further subdivided into a pre-defined set of triangles or tetrahedrons. This step is subsequently repeated for each geometry in the order that the level-set fields are defined. Hence, the resulting foreground mesh may differ slightly depending on the order in which the level-set fields are defined.
- At this point the elements are just geometric entities. By placing nodes at the vertices linear elements are obtained. Higher order elements are obtained by positioning equispaced (with respect to a standardized parametric element) nodes along the edges and the inside of the resulting triangles and tetrahedrons.
- All elements are assigned a material membership based on the signs of the level-set functions, as discussed previously.

The decomposition process layed out above leads to a foreground mesh with the properties listed below.

- The element facets created during the templated triangulation are straight. Therefore, the **approximation of the geometry is piecewise linear**.
- Since the foreground mesh constructed by decomposition of a background grid, it is **background-fitted** as defined by Fromm et al. [1].
- Further, the **quality of the geometry approximation is influenced by the resolution of the background grid** that is decomposed. Quadtree or octree refinement around the interfaces may be performed to improve the geometry representation.
- The simple algorithmic decomposition may lead to **triangular elements of extremely poor quality** (large aspect ratios, very high maximum angles). For interpolation-based

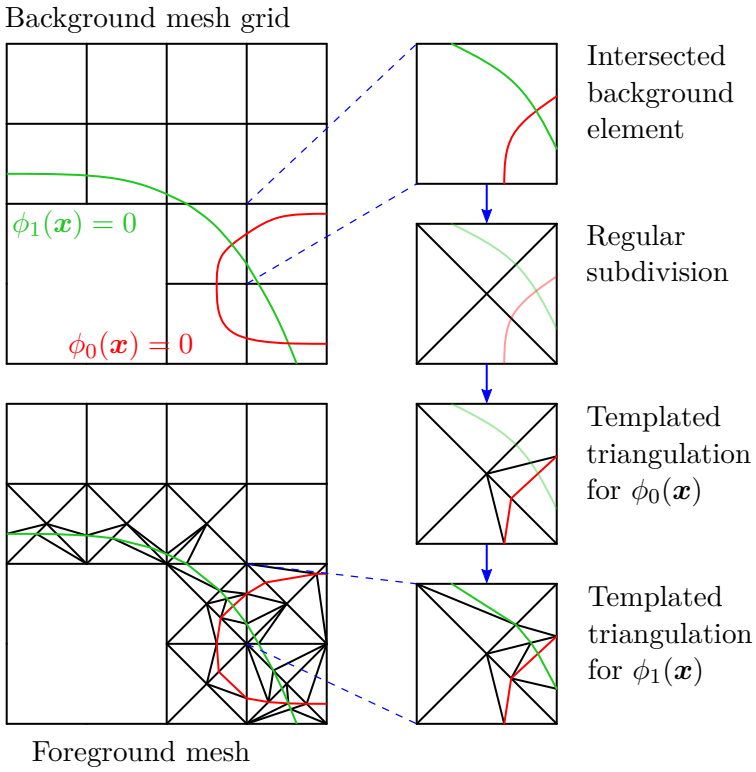

Figure 1.8: Decomposition of a background mesh grid into a geometry-fitted foreground mesh.

immersed analysis the poor element quality is generally not of concern though.

- Non-intersected quadrilateral or hexahedral background elements are not decomposed. The resulting foreground mesh therefore contains a **mix of triangular and rectangular elements**. Non-intersected elements may additionally be marked for regular subdivision, though.
- If a hierarchically, i.e. locally, refined background mesh grid is used for decomposition, the **foreground mesh may contain hanging nodes**, like shown in Figure 1.8.

1.4 Lagrange Extraction

Lastly, it should be useful to recap the basics of Lagrange extraction. The key to performing interpolation-based analysis is the fact that Lagrange basis functions $\phi_j(\boldsymbol{x})$ are *interpolatory*, i.e. they are $= 1$ at exactly one nodal point, and $= 0$ at every other nodal point $\hat{\boldsymbol{x}}_i$. An illustration for linear Lagrange basis function inside a linear triangular element is provided on the right of Figure 1.9.

$$\phi_i(\hat{\boldsymbol{x}}_i) = \begin{Bmatrix} 1 & j = i \\ 0 & j \neq i \end{Bmatrix} = \delta_{ij} \tag{1.8}$$

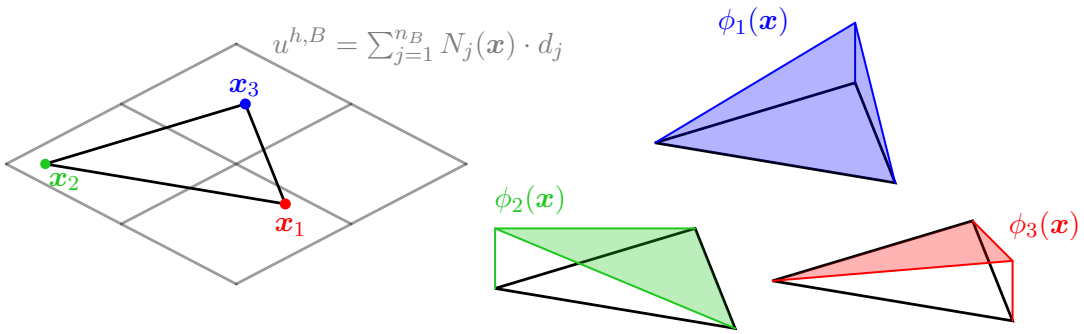

Figure 1.9: Lagrange extraction.

A Lagrange element can be placed into another mesh that may provide another type of interpolation $u^{h,B} = \sum_{j=1}^{n_B} N_j(\boldsymbol{x}) \cdot d_j$. The two interpolations u^h can be assumed to be approximately equal to each other (in a mathematically crude fashion).

$$\begin{aligned} u^{h,B} &\approx u^{h,L} \\ \sum_{j=1}^{n_B} N_j(\boldsymbol{x}) \cdot d_j &\approx \sum_{i=1}^{\nu} \phi_j(\boldsymbol{x}) \cdot c_j \end{aligned} \tag{1.9}$$

Substituting identity (1.8) into the equation at the nodal points $\hat{\boldsymbol{x}}_i$ yields the extraction operator M_{ij} relating the degrees of freedom of the two bases to each other.

$$\begin{aligned}\sum_{j=1}^{n_B} N_j(\hat{\boldsymbol{x}}_i)\cdot d_j &\approx \sum_{i=1}^{\nu}\phi_j(\hat{\boldsymbol{x}}_i)\cdot c_j\\ \sum_{j=1}^{n_B} M_{ij}\cdot d_j &\approx \sum_{i=1}^{\nu}\delta_{ij}\cdot c_j\\ \sum_{j=1}^{n_B} M_{ij}\cdot d_j &\approx c_i\end{aligned} \tag{1.10}$$

Hence, the extraction operator can be obtained simply by evaluating the background basis functions $N_j(\boldsymbol{x})$ at the nodal points of the foreground mesh $\hat{\boldsymbol{x}}_i$.

$$M_{ij} = N_j(\hat{\boldsymbol{x}}_i) \tag{1.11}$$

Note, due to the enrichment of the basis of the background mesh, nodes positioned along an interface will have different non-zero enriched background basis functions associated with them depending on with respect to which material equation (1.11) is evaluated.

$$\mathcal{V}^h = span\left\{\widehat{N}_j(\boldsymbol{x}) = \sum_{i=1}^{\nu} M_{ij}\cdot\phi_i\right\}_{j=1}^{n} \tag{1.12}$$

Using (1.10), the elemental stiffness matrices $A^{(e)}$ and force vectors $B^{(e)}$ assembled in a "standard" finite element procedure can easily be projected into the space of the background mesh, or more precisely, the *interpolated* background basis (1.12) as stated in equation (1.13).

$$\begin{aligned}K^{(e)} &= \sum_{j,k} M_{ji}\cdot A^{(e)}_{jk}\cdot M_{kl}\\ F^{(e)} &= \sum_{j} M_{ji}\cdot B^{(e)}_{jk}\end{aligned} \tag{1.13}$$

2 User Guide

This chapter will provide instructions on how to use MORIS' mesh generation workflow with a simplified XML input file and list all available options and parameters. Below a quick overview on how to run the mesh generation is given. As discussed in Chapter 1, conceptually the code requires definitions for the geometry, background, and foreground meshes, the parameters for which are explained in Section 2.1, Section 2.2, and Section 2.3 respectively. Lastly, an explanation of the extraction operator output is provided in Section 2.4.

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- All parameters are nested under this root -->
<MeshGenerationParameterList>

    <!-- Parameters defining the geometry -->
    <Geometries>
        ...
    </Geometries>

    <!-- Parameters defining the background grids and meshes -->
    <BackgroundMeshes>
        ...
    </BackgroundMeshes>

    <!-- Parameters defining the foreground mesh and the output -->
    <ForegroundMesh>
        ...
    </ForegroundMesh>

</MeshGenerationParameterList>
```

Listing 2.1: Structure of the XML input file

Initial Setup Firstly, it is convenient to define environment variables to run the MORIS executable from any directory. This can be done by setting environment variables manually, or by adding them to the `.cshrc_moris`, if it is not already set. For this documentation the environment variables `MRD` and `MRO` are used, referencing versions of MORIS compiled with and without debug flags respectively.

```
> setenv MRD $MORISROOT/build_dbg/projects/mains/moris
> setenv MRO $MORISROOT/build_opt/projects/mains/moris
```

The environment variable `MORISROOT` should already be defined in the `.cshrc_moris`.

Next, the input file needs to be defined. The structure of the input file is shown in Listing 2.1. All parameters listed in the following sections are to be nested under their respective tags. Example input files are provided in the GitHub repository alongside this documentation (↗).

Finally, to generate the desired mesh and extraction operators, run the MORIS executable and provide the flag `--meshgen` with the input file name.

```
> $MRO --meshgen Input_File.xml
```

2.1 Geometry

As discussed in Section 1.1, geometry is defined using multiple Level-Set Functions (LSFs). The different regions defined by the zero level-set iso-contours can be merged using a phase map. Currently, there are two ways LSFs can be defined:

1. Pre-defined geometries
2. Image files
3. WaveFront .obj files

Each new LSF (and hence geometry) is defined using a `Geometry` tag and nested under the `Geometries` tag (see Listing 2.1). The `type` attribute is used to indicate how the LSF is defined. The parameters defining the particular geometry are then nested within. Examples are provided below with each of the respective geometry types.

Currently the signed distance fields for five basic geometries are built-in. An additional `geom` attribute is used to initialize each basic geometry.

- circle
- sphere
- ellipse
- ellipsoid
- line/plane

A Circle is defined by a center point in the xy-plane and its radius using the tags `Point` and `Radius` respectively. The vector defining the center point must have two entries defining the (x, y)-position of the point; the array entries are comma-separated. Example parameters are shown in Listing 2.2. The level-set function is defined to be positive inside the circle, and negative outside.

```
<Geometry type="pre_defined" geom="circle">
    <Point>1.3,3.7</Point>
    <Radius>4.2</Radius>
</Geometry>
```

Listing 2.2: Example parameters for defining a circle.

A Sphere is defined analogous to the circle using a `Point` and a `Radius` respectively. The vector defining the center point must have three entries defining the (x, y, z)-position of the point. The level-set function is defined to be positive inside the sphere, and negative outside.

```
<Geometry type="pre_defined" geom="sphere">
    <Point>1.3,3.7,0.8</Point>
    <Radius>4.2</Radius>
</Geometry>
```

Listing 2.3: Example parameters for defining a circle.

An Ellipse is defined by a center point in the xy-plane and its semi-diameters using the tags `Point` and `SemiDiameters` respectively. Both need to be defined by a comma-separated arrays of length two defining the center point's (x, y)-position and semi-diameters in the x- and y-directions respectively. Additionally, an exponent may be defined by the user to construct a super-ellipse (↗) using the tag `Exponent`. This value is set to 2 by default, which resembles a standard ellipse, but can be changed to make the ellipse more convex or concave. Example parameters are shown in Listing 2.4. The level-set function is defined to be positive inside the ellipse, and negative outside.

```
<Geometry type="pre_defined" geom="ellipse">
    <Point>1.3,3.7</Point>
    <SemiDiameters>1.3,3.7</SemiDiameters>
    <!-- Default value is 2, if not specified -->
    <Exponent>4.0</Exponent>
</Geometry>
```

Listing 2.4: Example parameters for defining a (super-) ellipse.

The Ellipsoid is defined analogous to the ellipse using the tags `Point`, `SemiDiameters`, and, if needed, `Exponent`. The former two each need an additional third value specified for the z-position of the center point and the semi-diameter in z-direction respectively.

```
<Geometry type="pre_defined" geom="ellipsoid">
    <Point>1.3,3.7,4.2</Point>
    <SemiDiameters>1.3,3.7,0.8</SemiDiameters>
    <!-- Default value is 2, if not specified -->
    <Exponent>4.0</Exponent>
</Geometry>
```

Listing 2.5: Example parameters for defining a (super-) ellipsoid.

A Plane is defined by a point and a normal which are parsed using the tags `Point` and `Normal` respectively. Both are vectors and must have as many entries as there are spatial dimensions. Example parameters are shown in Listing 2.6. The level-set function is defined to be positive on that side of the plane towards which the normal is pointing.

```
<Geometry type="pre_defined" geom="plane">
    <Point>1.3,3.7</Point>
    <Normal>4.0,2.0</Normal>
</Geometry>
```

Listing 2.6: Example parameters for defining a plane.

Image Files can be converted to a signed distance field (SDF) using the MatLab scripts provided in the GitHub repository under share/matlab/. For three-dimensional image information, a stack of images needs to be parsed with the image index before the file ending, e.g. a stack of images called `Cube.000.png`, `Cube.001.png`, ..., `Cube.127.png` could be used. The generated SDF by each script is stored in .hdf5 format.

The generated SDF is assumed to occupy a rectangular or cuboidal domain depending on the number of spatial dimensions. The SDF needs to be positioned relative to the ambient domain by defining its side lengths and origin using the `ImageDimensions` and `ImageOrigin` tags respectively. The origin of the SDF domain is assumed to be the vertex the the minimum x-, y-, and, if applicable, z-values. An example for defining a geometry based off an image file is provided in Listing 2.7 as well as in the "Bear" example input file (↗).

```
<Geometry type="image_file">
    <FileName>Cali_Bear.hdf5</FileName>
    <ImageDimensions>3.81,1.95</ImageDimensions>
    <ImageOrigin>0.0,0.0</ImageOrigin>
</Geometry>
```

Listing 2.7: Example parameters for an image file as provided in one of the examples.

Wavefront OBJ Files are used to store the geometry of 3D objects. More details how the file format stores geometry data can be found here (↗). They can easily be generated from more common STL files, e.g. in ParaView, as long as the geometry used has a closed surface. The OBJ file the geometry is to be loaded from is specified using the tag `FileName`. The position of the coordinate system which the geometry is defined in within the OBJ file is set using the tag `ObjectOrigin`. Additionally, the signed distance field generated from the OBJ data can be offset by a constant using the tag `SdfShift`. This can be used in, e.g., situations where small scale features merge on a coarse mesh. An example for defining a geometry based off an OBJ file is provided in Listing 2.8 as well as in the "Dragon" example input file (↗).

```
<Geometry type="object_file">
    <FileName>Dragon.obj</FileName>
    <ObjectOrigin>0.0,0.0,0.0</ObjectOrigin>
    <SdfShift>0.00</SdfShift>
</Geometry>
```

Listing 2.8: Example parameters for an OBJ file as provided in one of the examples.

Future Extensions to the geometry capabilities are more options for pre-defined geometries and patterns thereof. Further, symbolic expressions for level-set functions will be enabled. Please reach out with any feature suggestions and requests.

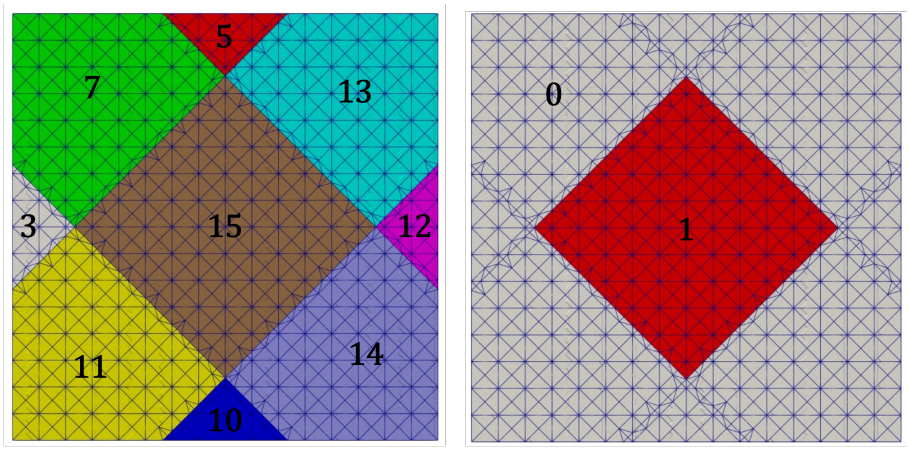

Figure 2.1: Default phase assignment for a rotated square constructed from four planes (left), and phase assignment after using the phase map shown in Listing 2.9

A Phase Map is used to assign materials to the phases. As mentioned in Section 1.1, up to $n_M = 2^{n_{LSF}}$ phases and therefore material sub-domains arise from n_{LSF} level-set functions, assuming that each sign combination from the LSFs corresponds to one phase. For the default material assignment each phase, i.e. regions with a certain sign combination, correspond to a material. The index of a phase is assumed to be the value of a bitset where each digit corresponds to the sign of one of the level-set function. Negative values of the LSFs correspond to zeros, positive values to ones. The phase assignment is provided in Table 2.1.

m_i	ϕ_0	ϕ_1	ϕ_2
0	-	-	-
1	-	-	+
2	-	+	-
3	-	+	+
4	+	-	-
5	+	-	+
6	+	+	-
7	+	+	+

Table 2.1: Default phase assignment for three level-set functions

The default material assignment for the rotated square example (↗) using four planes to define the geometry is shown on the right of Figure 2.1.

Using the tag `PhaseMap` each phase index can be mapped to a material index. The syntax

consists of comma-separated pairs of indices, the first indicating the index of the phase to map from, and the second the the index of the material to map to. The pairs are separated by semicolon. The example input is shown in Listing 2.9. If the phase assignment for one's problem is not clear, it is advised to first generate a foreground mesh without a phase map specified and visualize the phase assignment before applying a phase map. To speed up this process, the extraction operator output can be suppressed by specifying the tag `OutputExtractionOperators` in the `ForegroundMesh` parameter list and setting it to `false`.

```
<PhaseMap>3,0;5,0;7,0;10,0;11,0;12,0;13,0;14,0;15,1</PhaseMap>
```

Listing 2.9: Phase map for the material re-assignment shown in Figure 2.1

2.2 Background Meshes

As discussed in Section 1.2, a background mesh is understood to be a grid with a certain type of basis functions defined on it. The definition of all background meshes, with respect to which extraction operators will be computed, starts of by defining a `BaseGrid` as shown in Listing 2.10. The base grid is defined by the number of elements in each spatial dimension using the tag `Size`, the dimensions of the rectangular or cuboidal ambient domain to be meshed using the tag `Dimensions`, and the position of the origin of said domain using the tag `Origin`.

```
<BaseGrid>
    <Size>12,8</Size>
    <Dimensions>3.4,2.1</Dimensions>
    <Origin>-1.2,-0.3</Origin>
</BaseGrid>
```

Listing 2.10: Definition of a base grid

From the base grid, multiple refined mesh grids can be defined under the tag `MeshGrids` as shown in Listing 2.11. Each mesh grid needs an index, starting with 0, which is set using the attribute `ind`. The grids are quadtree or octree refined versions of the base grid. The number or repeated refinements is set using the `InitialRefinements`.

```
<MeshGrids>
    <MeshGrid ind="1">
        <InitialRefinements>0</InitialRefinements>
    </MeshGrid>
    <MeshGrid ind="0">
        <InitialRefinements>1</InitialRefinements>
    </MeshGrid>
</MeshGrids>
```

Listing 2.11: Definition of a multiple mesh grids

Finally, the B-spline background meshes are defined under the tag `BsplineMeshes` by combining the mesh grid, identified by its index using the `MeshGridIndex` tag, and a polynomial order which is set using the `PolynomialOrder`. An index, starting with 0, must be assigned to each B-spline mesh using the attribute `ind`. This index will be used for indexing the extraction operators to be outputted.

```
<BsplineMeshes>
    <BsplineMesh ind="0">
        <MeshGridIndex>0</MeshGridIndex>
        <PolynomialOrder>2</PolynomialOrder>
    </BsplineMesh>
    <BsplineMesh ind="1">
        <MeshGridIndex>1</MeshGridIndex>
        <PolynomialOrder>1</PolynomialOrder>
    </BsplineMesh>
</BsplineMeshes>
```

Listing 2.12: Definition of a multiple mesh grids

Future Developments to the background mesh capabilities are to enable hierarchical refinements around interfaces in the B-spline background mesh. Other refinement criteria for the background meshes can be enabled upon request.

2.3 Foreground Mesh

To generate the foreground mesh, one of the mesh grids is chosen by its index under the tag `DecompositionGrid`. This grid **needs to be at least as refined as the finest grid used for defining the B-spline meshes**. Additionally, the foreground mesh can then be quadtree or octree refined around the interfaces using the tag `InterfaceRefinements`.

As mentioned in the overview, the foreground mesh is a hybrid TRI/QUAD or TET/HEX

mesh and may contain hanging nodes. By setting the flag `TriangulateAllFgElems` to true, the regular subdivision can also be applied to non-intersected background elements. Note, that this option is currently not supported for locally refined meshes due to the code not being able to handle hanging nodes along facets shared by triangular and rectangular elements. The polynomial order of the foreground mesh is chosen using the tag `FgPolynomialOrder`. Currently orders $p = 1$ and $p = 2$ are supported.

```
<ForegroundMesh>
    <DecompositionGrid>0</DecompositionGrid>
    <InterfaceRefinements>1</InterfaceRefinements>
    <FgPolynomialOrder>2</FgPolynomialOrder>
    <TriangulateAllFgElems>false</TriangulateAllFgElems>
</ForegroundMesh>
```

Listing 2.13: Definition of a multiple mesh grids

The generated foreground mesh is outputted in the EXODUS file format with the filename `foreground_mesh.exo`. More details on the file format in general can be found at sandialabs.github.io/seacas-docs (↗).

A noteworthy feature of EXODUS is its organization of elements into *blocks* and *sets*. Volume elements in the output mesh are sorted into blocks based on their assigned material. Blocks are however limited to elements of the same type which is why elements composing one material may be split up into a block of triangular elements stemming from intersected background elements and non-intersected rectangular background elements. Blocks composed of the former are marked by a "**c**" for cut, e.g. the block of triangular elements for material 0 is called `HMR_dummy_c_p0`; the latter are marked by an "**n**" for non-cut, e.g. `HMR_dummy_n_p0`. An example for the blocks found in the rotated square example (↗) are shown in Figure 2.2.

Further, interface elements are sorted into sets. There are two types of interface sets.

- **Sets of elements that interface between a given material pair.** For example, all interface elements connecting the materials 0 and 1 can be found in the set `iside_b0_0_b1_1`; "`b0_0`" and "`b1_1`" indicate that material 0 is considered the primary, and material 1 the secondary material. Sets with flipped primary and secondary materials, but equal interface elements, are available. For this example the name would be `iside_b0_1_b1_0`. This grouping of interface elements means that the number of geometries and materials defined has an impact on how easy it is to grab sets of interface elements to apply boundary conditions. For example, a square block can be defined using a single geometry or using four planes. The former results necessarily in only one interface set, the latter allows each side to be sorted into an individual interface set allowing different boundary conditions to be imposed on each side of the square.

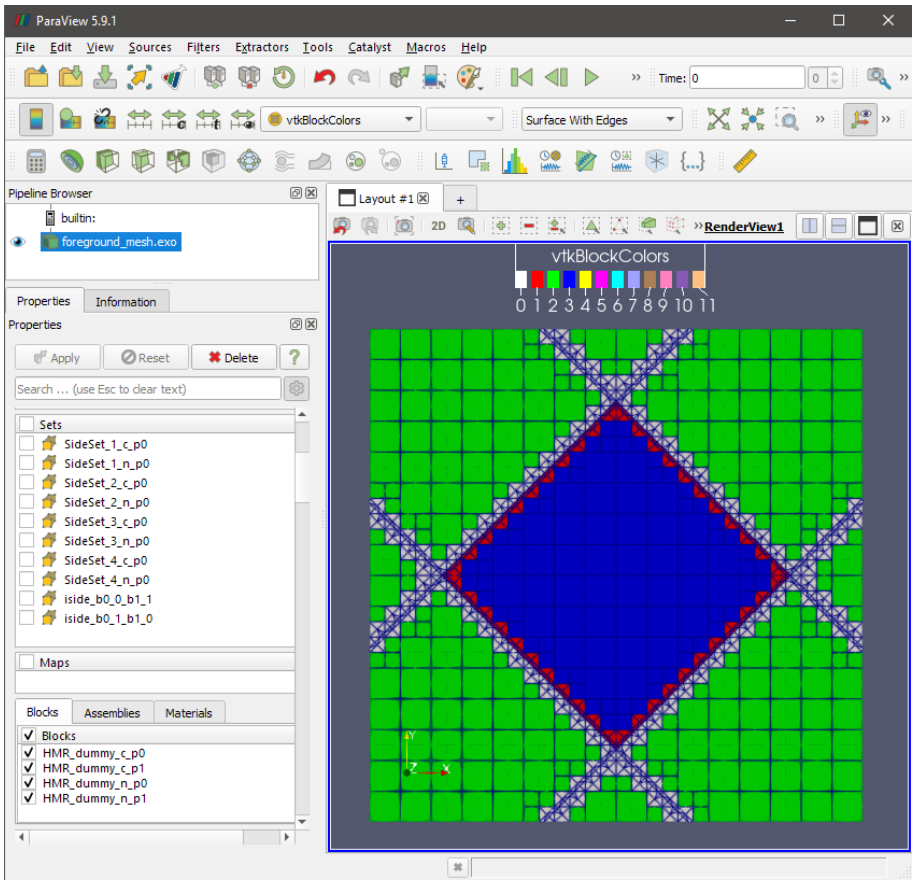

Figure 2.2: Mesh blocks and sets written to the foreground mesh output for the rotated square example and how they are listed in ParaView.

- **Sets of facets of elements of a given Block that make up the boundary of the domain.** Each side of a domain is denoted using a *side ordinal*. The convention for naming side ordinals, alongside other numbering conventions used in the EXODUS file format, can be found here (). The facets of the elements inside the block `HMR_dummy_c_p0` that are part of the bottom, i.e. side ordinal 1, of the domain are collected in the set `SideSet_1_c_p0`. This allows boundary conditions to easily be imposed onto a given material along the various sides of the domain.

Future Developments to the foreground mesh generation capabilities are to enable the triangulation of all elements even for locally refined foreground meshes. Further, enabling cubic foreground elements is possible.

2.4 Extraction Operators

The extraction operators are computed per foreground element and for each B-spline mesh defined. For each B-spline mesh a separate HDF5 file is created that contains the B-spline mesh index with a prefix `"B"` in its name,
e.g. the files `Elemental_Extraction_Operators_B0.hdf5` and `Elemental_Extraction_Operators_B1.hdf5` contain the extraction operators with respect to B-spline mesh indices 0 and 1 respectively.

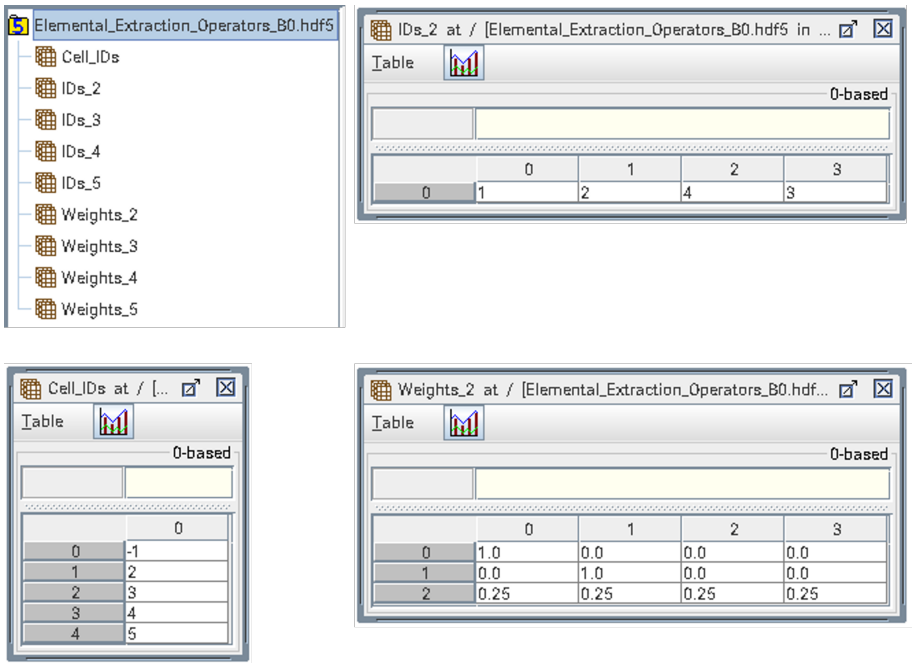

Figure 2.3: Output of the extraction operators in HDF5 format for a foreground mesh consisting of four elements. The file contains a list of element IDs (bottom left), and for each element the extraction operator (bottom right) and an assembly map (top right).

Each contains an array called `Cell_IDs` that is an element index to ID array. An example is shown in Figure 2.3, bottom left. The index of an array entry represents the index of a foreground cell, the value of the entry is the ID. Note, the array may contain IDs whose value is -1 which indicates the corresponding element is not part of the foreground mesh outputted to the EXODUS file and can be ignored.

For each foreground element ID the extraction operator named `Weights_<ID>` and a list of background basis function IDs named `IDs_<ID>` are written to the HDF5 files. Examples are shown on the right in Figure 2.3. The weights are the extraction operators that are supposed

to be applied to the elemental Jacobians, residuals, stiffness matrices, etc. as stated in equation (1.13) and demonstrated in Figure 2.4. Once this operation is completed the ID vector is used to assemble the elemental matrix, or vector into the global system. In the elemental stiffness matrix example in Figure 2.4, the entry in the second row and column, i.e. position $(2, 3)$ would be assembled into position $(2, 4)$ of the global stiffness matrix.

$$\underset{\mathbf{M}^T}{\begin{bmatrix} 1 & 0 & 0.25 \\ 0 & 1 & 0.25 \\ 0 & 0 & 0.25 \\ 0 & 0 & 0.25 \end{bmatrix}} \cdot \underset{\mathbf{A}^{(e)}}{\begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix}} \cdot \underset{\mathbf{M}}{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0.25 & 0.25 & 0.25 & 0.25 \end{bmatrix}} = \begin{matrix} 1 \\ 2 \\ 4 \\ 3 \end{matrix} \underbrace{\begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{bmatrix}}_{\mathbf{K}^{(e)}} \\ \begin{matrix} 1 & 2 & 4 & 3 \end{matrix}$$

Figure 2.4: Projection of the elemental stiffness matrix $\boldsymbol{A}^{(e)}$ assembled on the Lagrange element into the background basis using the example extraction operators showm in in Figure 2.3. The blue arrays are the `IDs_2` vector and serve as an assembly map.

Bibliography

[1] J. E. Fromm, N. Wunsch, R. Xiang, H. Zhao, K. Maute, J. A. Evans, and D. Kamensky. Interpolation-based immersed finite element and isogeometric analysis. *Computer Methods in Applied Mechanics and Engineering*, 405:115890, 2022.

[2] L. Noël, M. Schmidt, K. Doble, J. Evans, and K. Maute. Xiga: An extended isogeometric analysis approach for multi-material problems. *Computational Mechanics*, 70(6):1281–1308, 2022.

[3] M. Schmidt, L. Noel, K. Doble, J. A. Evans, and K. Maute. Extended isogeometric analysis of multi-material and multi-physics problems using hierarchical B-splines. *Computer Methods in Applied Mechanics and Engineering*, 2022.