The Journal of Open Source Software

FEM_2D: A Rust Package for 2D Finite Element Method Computations with Extensive Support for *hp*-refinement

Jeremiah Corrado[1], **Jake J. Harmon**[1], **Milan M. Ilic**[1,2], **and Branislav M. Notaroš**[1]

1 Department of Electrical and Computer Engineering, Colorado State University, USA **2** School of Electrical Engineering, University of Belgrade, Serbia

DOI: 10.21105/joss.04172

Software

- Review
- Repository
- Archive

Editor: Jed Brown

Reviewers:

- @jeremylt
- @YohannDudouit

Submitted: 13 February 2022
Published: 21 April 2023

Introduction

The Finite Element Method (FEM) is a powerful computation framework used to solve Partial Differential Equations (PDEs) on arbitrary geometries. While physical systems behave in a continuous manner (both in space and time), FEM solvers are able to model these dynamics with a high fidelity by decomposing a physical model into a finite set of elements. Each element supports a finite number of degrees of freedom, which are used to describe the behavior of the system. This way, other mathematical tools, such as linear solvers, can be used to compute highly accurate approximations of the continuous dynamics. Solutions are constructed such that the PDE is satisfied along with some boundary conditions (on the border of the domain) and some continuity conditions (between neighboring elements).

Some common PDEs include the Navier-Stokes equations, which characterize the behavior of fluids, Schrödinger's equation, which governs the evolution of quantum systems, and Maxwell's Equations, which are a macroscopic description of essentially all electromagnetic phenomena. The ability to accurately and efficiently model these differential equations and others is imperative to the success of many engineering projects and scientific endeavors. Most of the technology that engineers are interested in developing has far exceeded the reach of direct mathematical analysis, and thus computational tools such as FEM are used ubiquitously to drive technological development forward.

As such, innovations in FEM have a direct impact on essentially all engineering disciplines. The more efficient, accurate, and feature-rich we can make simulation tools, the more beneficial they will be to industrial and scientific applications. This is the motivation behind academic work in the field of FEM. The `FEM_2D` library is a Rust package that aims to enable further research into a particular FEM innovation called Refinement-by-Superposition (RBS). The related research papers (Corrado et al., 2021; Harmon et al., 2021) explore benefits of RBS using the 2D Maxwell Eigenvalue Problem as an experimental test case.

FEM codes based on RBS differ from more traditional FEM codes in two primary ways: (1) The discretization data structure supports a set of hierarchical trees of elements (a "forest" data structure) rather than a "flat" set of elements. `FEM_2D`'s `Mesh` data structure aims to expose a wide array of functionality for instantiating and manipulating a tree of elements both with *h*- and *p*-refinements. Here, *h*-refinement refers to the improvement of the spatial discretization by superimposing smaller elements over existing elements, and *p*-refinement refers to the process of increasing the polynomial expansion order of the basis set associated with a particular element. (2) The integration API, used to populate the system matrices, supports inter-layer integration that can handle integrals of overlapping basis functions defined on different layers of element trees. In traditional implementations, integrals are computed strictly on individual elements. FEM_2D contains all the necessary integration functionality to solve the Maxwell Eigenvalue Problem or other H(curl) conforming problems.

In addition to the centrally important *hp*-refinement functionality, FEM_2D is supported by a rich set of surrounding features. This includes two eigensolvers: a dense solver that is entirely native to Rust, and a sparse solver implemented using an external C++ library. There is also a solution plotting API, and an external Mesh plotting tool to assist in future research work based on the FEM_2D Library.

Statement of Need

In some application domains such as high-frequency structure analysis, efficiently computing FEM solutions over geometries with sharp edges or stark material discontinuities necessitates *hp*-refinement (whether isotropic or anisotropic) because these situations tend to introduce multi-scale solution behavior that is challenging to model with pure *p*- or pure *h*-refinements. This motivates the use of solvers with combined *hp*-refinements where an initially coarse mesh is progressively refined both spatially and in terms of polynomial expansion order. Within this framework, at each refinement step, small-scale behavior is localized into smaller elements using *h*-refinement, and modelling of higher-order behavior is improved using *p*-refinements. When intelligently applied to problems with sharp or singular solutions, combined *hp*-refinements can achieve exponential rates of convergence, where *h*- or *p*-refinements alone would only be able to achieve algebraic convergence. FEM_2D exposes an *hp*-refinement API that can be used to achieve such results (Harmon et al., 2021).

Within the class of *hp*-refinements, the addition of anisotropic *hp*-refinements (over isotropic ones) presents a larger capacity for solution efficiency, as small-scale behavior can be targeted more directly and ineffectual Degrees of Freedom (DoFs) can be left out of the system (Corrado et al., 2021). In other words, these directionally specific refinements can reduce the introduction of superfluous entropy into the system by directly targeting inaccuracies that are specific to only one direction. The resultant improvement in per-DoF efficiency can be used to reduce the memory requirements for a given solution accuracy or to increase the accuracy achievable with a given amount of memory. Thus, providing a feature-rich anisotropic *hp*-refinement API is an additional goal of the FEM_2D library.

FEM_2D aims to expose some features similar to those in other FEM libraries such as Deal.II (Arndt et al., 2021) and MFEM (Anderson et al., 2021) that are designed to be general purpose frameworks for implementing FEM codes. Although it is not nearly as feature-rich as these libraries, FEM_2D's *hp*-refinement API aims to provide the basic functionality needed to iteratively solve challenging 2D FEM problems as described in the associated work (Corrado et al., 2021, 2022; Harmon et al., 2021). These features will be discussed in detail in the following sections. Additionally, FEM_2D differs from other common FEM libraries in that its *h*-refinement functionality is built on a Refinement by Superposition (RBS) framework, whereas most FEM libraries with support for *h*-refinement use Refinement by Replacement (RBR).

For research purposes, it can also be helpful to design software libraries that are straightforward to use and understand. As such, we note that FEM_2D is available on Rust's package manager Cargo, making it straightforward to download, compile, and run using only a few commands. It can also be included as a dependency in any Rust project in order to develop new code on top of the library. Additionally, the RBS approach that underpins FEM_2D lends itself to a desirable level of simplicity with respect to its *h*-refinement implementation. This is because RBS is designed such that continuity conditions between neighboring elements are enforced by construction: no explicit handling of hanging nodes is required. As such, some of the typical difficulties with implementing *h*-refinements over quadrilateral elements for H(curl) or H(div) conforming boundary conditions are avoided entirely. Our hope is that the straightforward nature of the RBS approach will allow other researchers to easily contribute to FEM_2D, or to use it as a starting point for software development in adjacent research.

Features

hp-Refinement API:

FEM_2D's most notable feature is its highly dynamic and expressive *hp*-refinement API. Notably, FEM_2D supports n-irregular anisotropic *h*-refinement, meaning that elements can be refined in each direction individually, and there is no limitation on the number of hanging nodes introduced along an edge. As such, refinement algorithms built on top of FEM_2D do not have to check that an *h*-refinement is valid before applying it, and no transition elements are needed between regions of coarse and dense *h*-refinement. This level of freedom is facilitated by the underlying RBS methodology. FEM_2D also supports anisotropic *p*-refinements, meaning that the polynomial expansion orders of the Basis Functions associated with each element can be modified separately in each direction. Lastly, there are various methods for querying properties of individual elements and the relationships between them (adjacency/descendancy). Overall, the Mesh Refinement API is intended to provide enough abstraction and feature-richness to make the implementation of refinement algorithms, like those used in Corrado et al. (2022), as straightforward and unencumbered as possible.

h-refinement:

It is important to note that there are three primary *h*-refinement types that are designated by the HRef enum:

- T - isotropic: produces 4 child elements
- U - anisotropic in the u-direction: produces 2 child elements
- V - anisotropic in the v-direction: produces 2 child elements

There are also two sub-types associated with the U and V refinements that invoke a subsequent anisotropic refinement on one of the two child elements in the opposite direction. These are constructed by passing an additional optional index to the relevant constructors: HRef::U(Some(child_index)) and HRef::V(Some(child_index)), where child_index must be either 0 or 1.

Note that u and v represent the parametric x and y dimensions. Curvilinear elements are not yet supported; thus these symbols can generally be considered to by synonymous with x and y respectively.

It is also important to note that the global_h_refinement and h_refine_with_filter methods will only apply refinements to elements that are eligible for *h*-refinement (i.e., they must be leaf elements and the length of each of their edges must be above a minimum threshold). Alternatively, the methods that expose more explicit control (h_refine_elems and execute_h_refinements) can return an error if one of the specified elements is not eligible for *h*-refinement. A detailed explanation of the possible error types is provided in the documentation.

The following example depicts a variety of *h*-refinement methods that could be used to manipulate a Mesh data structure:

```rust
use fem_2d::prelude::*;
use std::error::Error;

fn do_some_h_refinements(mesh_file_path: &str) -> Result<Mesh, Box<dyn Error>> {
    let mut mesh = Mesh::from_file(mesh_file_path)?;

    // isotropically h-refine all elems
    mesh.global_h_refinement(HRef::T);

    // anisotropically h-refine all elems connected to some target node
    let target_node_id = 5;
    mesh.h_refine_with_filter(|elem| {
        if elem.nodes.contains(&target_node_id) {
            Some(HRef::u())
        } else {
            None
        }
    });

    // anisotropically h-refine a list of elems by id
    mesh.h_refine_elems(vec![3, 4, 8, 12], HRef::v())?;

    // directly apply a list of refinements to the mesh
    mesh.execute_h_refinements(vec![
        (1, HRef::T),
        (5, HRef::U(Some(0))),
        (6, HRef::U(Some(1))),
        (10, HRef::V(None)),
    ])?;

    Ok(mesh)
}
```

***p*-refinement:**

The following example shows how some of the *p*-refinement methods may be used. Here, the `Mesh` is provided as an argument to the function rather than being loaded from a file. The *p*-refinement objects are constructed from a static method on `PRef` using a pair of `i8`'s (8-bit signed integers). As such, any element's u- and v-directed expansion orders can be modified independently in either the positive or negative direction.

The behavior of these methods is straightforward with the slight caveat that the `global_p_refinement` and `p_refine_with_filter` methods will guard against any refinement pushing an element outside of its valid expansion order range. Specifically, refinements are clamped element-wise to ensure that the final expansion order is in the range [1, 20]. The *p*-refinement methods that can return an error (those followed by a `?` in the example) do not exhibit this behavior. This is in keeping with the design of the *h*-refinement API in the sense that methods with less explicit control are safer—and are intended for simpler use cases—while the more explicit methods allow for failure and are intended for more advanced use.

```
use fem_2d::prelude::*;

fn do_some_p_refinements(mesh: &mut Mesh) -> Result<(), PRefError> {
    // isotropically p-refine all elems (with a magnitude-2 refinement)
    mesh.global_p_refinement(PRef::from(2, 2));

    // positively p-refine all "leaf" elems (with a magnitude 1 refinement)
    // negatively p-refine all other elems (with a magnitude -1 refinement)
    mesh.p_refine_with_filter(|elem| {
        if elem.has_children() {
            Some(PRef::from(-1, -1))
        } else {
            Some(PRef::from(1, 1))
        }
    });
    // anisotropically p-refine a list of elems by id
    mesh.p_refine_elems(vec![3, 4, 8, 12], PRef::from(4, 2))?;

    // directly apply a list of refinements to the mesh
    mesh.execute_p_refinements(vec![
        (1, PRef::from(3, 2)),
        (5, PRef::from(0, 1)),
        (6, PRef::from(-1, -1)),
        (10, PRef::from(4, -2)),
    ])?;

    Ok(())
}
```

The Mesh data structure also has an alternative set of methods to modify expansion orders by setting them directly rather than additively. These methods can be very useful in scenarios where the current expansion orders are irrelevant, and elements require a specific expansion order which is either known beforehand or computed ad-hoc. The following shows how this API may be used in practice.

Here, both methods take an Orders object that specifies the expansion order in the u and v directions. In the first method, try_new is used to construct an Orders. This can fail if the expansion order specified is either 0 or greater than the maximum allowed expansion order: MAX_POLYNOMIAL_ORDER. The second interface takes a closure which creates an Option<Orders> for each element (in this example, the closure uses Orders::new which will panic when provided with invalid expansion orders).

```
use fem_2d::prelude::*;

fn set_some_expansion_orders(mesh: &mut Mesh, order: u8) -> Result<(), PRefError> {
    // set the expansion order on all elems to 'order'
    mesh.set_global_expansion_orders(Orders::try_new(order, order)?);

    // set the expansion orders to (4, 4) on all "leaf" elems
    // set the expansion orders to (2, 2) on all base layer elems
    // leave all other elems unchanged
    mesh.set_expansions_with_filter(|elem| {
        if !elem.has_children() {
            Some(Orders::new(4, 4))
        } else if elem.parent_id().is_none() {
            Some(Orders::new(2, 2))
        } else {
            None
        }
    });

    Ok(())
}
```

Problem Formulation and Solution

The following example shows how a simplified formulation of the Maxwell Eigenvalue Problem maps to the corresponding code in the library. This is intended provide a general overview of the libraries available functionality. It is not comprehensive, but does aim to provide a good starting point.

The Maxwell eigenvalue problem has the following Continuous-Galerkin formulation for an arbitrary Domain terminated with Dirichlet boundary conditions, (constraining the solution to TE modes only):

Find a solution:

$$\mathsf{U} = \{\mathbf{u}, \lambda\} \in B_{hp} \times \mathbb{R} \tag{1}$$

which satisfies:

$$b(\mathbf{u}, \phi) = \lambda a(\mathbf{u}, \phi) \quad \forall \phi \in B_{hp} \tag{2}$$

$$\text{where: } \begin{cases} B_{hp} \subset H_0(\text{curl}; \Omega) \\ a(\mathbf{u}, \phi) = \langle \nabla_t \times \mathbf{u}, \nabla_t \times \phi \rangle \\ b(\mathbf{u}, \phi) = \langle \mathbf{u}, \phi \rangle \end{cases} \tag{3}$$

The Generalized Eigenvalue Problem is built from a `Mesh` with the following code. This example assumes that `mesh` has already been refined to the desired state.

```rust
use fem_2d::prelude::*;
use rayon::prelude::*;

fn problem_from_mesh(mesh: Mesh) -> Result<GEP, GalerkinSamplingError> {
  // Setup a global thread-pool for parallelizing Galerkin Sampling
  rayon::ThreadPoolBuilder::new().num_threads(8).build_global().unwrap();

  // Generate a Domain (Ω) from a Mesh with H(curl) Continuity Conditions
  let domain = Domain::from(mesh, ContinuityCondition::HCurl);

  // Compute a Generalized Eigenvalue Problem
  let gep = galerkin_sample_gep_hcurl::<
      HierPoly, // Basis Space
      CurlCurl, // Stiffness Integral
      L2Inner,  // Mass Integral
  >(&domain, Some([8, 8]))
}
```

The `Domain` structure represents the entire FEM domain, including the discretization and the basis space that conforms to the provided continuity condition. (Only H(curl) is currently implemented but a framework is in place for implementing H(div) and other continuity conditions.)

Galerkin sampling is then executed in parallel over the Domain, yielding a Generalized Eigenvalue Problem composed of two sparse matrices. The Domain and a Gauss-Legendre-Quadrature grid size are provided as arguments. This function may also return an Error if the Galerkin Sampling fails due to an ill-posed problem.

The three generic arguments – designated with the turbofish operator (`::<>`) – correspond to the three lines of Equation 3. The basis space can be swapped for any other space that implements the `HierCurlBasisFnSpace` Trait. `HierPoly` is a relatively simple implementation composed of products of polynomial functions. A more sophisticated basis space: `HierMaxOrtho` can be included using the `max_ortho_basis` Feature Flag. Custom Basis Spaces can also be created by implementing the same Trait.

The `CurlCurl` and `L2Inner` integrals, which correspond to the Stiffness and Mass matrices respectively, can be swapped for any other structure that implements the `HierCurlIntegral` Trait. This generic interface allows users to leverage the Galerkin Sampling functionality against other curl-conforming problems.[1]

[1]The provided functionality is obviously somewhat incomplete, as only Curl Conforming problems can be solved; however, the library's module-structure and trait-hierarchy provide a clear template for the analogous H(div) implementation. There is also room for other Galerkin sampling and integration functionality associated with alternate continuity conditions. These methods, structures, and traits would require additions to the `Domain` structure, and few changes to the `Mesh` structure if any.

The Generalized Eigenvalue Problem, can then be solved using one of the available solvers:

```
// Dense solution (not recommended for large problems)
let eigenpair = nalgebra_solve_gep(gep, target_eigenvalue).unwrap();

// OR: Sparse solution (requires external SLEPc solver)
let eigenpair = slepc_solve_gep(gep, target_eigenvalue).unwrap();
```

The dense solver, implemented using Nalgebra ("Nalgebra," 2021), converts the eigenproblem's sparse matrices into dense matrices. This is an expensive operation, and should be avoided for large problems. The sparse solver, implemented using SLEPc (Balay et al., 1997, 2021a, 2021b; Hernandez et al., 2005), is a direct interface to a generalized eigensolver. This is a relatively fast operation, but requires an external solver to be installed and compiled. It also avoids directly inverting the B-matrix, which is numerically advantageous for ill-conditioned problems.

Both solvers look for the eigenvalue closest to the provided `target_eigenvalue`. They can return errors if the solution does not converge. Upon success, the returned eigenpair contains the eigenvalue nearest to the target, and the corresponding eigenvector with length equal to the number of degrees of freedom in the domain.

Field Visualization

The Fields API exposes functionality to generate a solution-field using an eigenvector and associated domain. It also allows functions of field solutions to be computed using basic mathematical operations. The following example shows how electric field solutions are generated and exported to a VTK file.

```
use fem_2d::prelude::*;
use std::error::Error;

fn compute_some_solution_fields(
    eigenpair: EigenPair,
    domain: &Domain
) -> Result<(), Box<dyn Error>> {
    // build a solution field space
    let mut field_space = UniformFieldSpace::new(domain, [16, 16]);

    // compute the x and y directed electric fields
    let [ex_name, ey_name] =
        field_space.xy_fields::<HierPoly>("E", eigenpair.vector)?;

    // compute the magnitude of the electric field
    field_space.expression_2arg([&ex_name, &ey_name], "E_mag", |ex, ey| {
        (ex.powi(2) + ey.powi(2)).sqrt()
    })?;

    // compute the absolute value of the x and y directed electric fields
    field_space.map_to_quantity(ex_name, "E_x_abs", |e| e.abs())?;
    field_space.map_to_quantity(ey_name, "E_y_abs", |e| e.abs())?;

    // print E_x, E_y, E_x_abs, E_y_abs, and E_mag to a VTK file
    field_space.print_all_to_vtk("path/to/file.vtk")
}
```

Here, we are using a `UniformFieldSpace` to define our solution space over the domain. This structure defines a grid of points such that the density is uniform across leaf-elements.[2] Here, we use a 16x16 grid. The parent elements will have a larger density because the leaf-element's points are projected "downwards" onto their ancestors. So, in this case, an element that has four children (who are all leafs) would evaluate its local solution using a 32x32 point grid such that the points align with the grids on its descendants.

On the following line, we compute the X- and Y-directed fields using the eigenvector (and the same basis-space as before). The `UniformFieldSpace` maintains an internal table of solution components designated by name. The names for the fields are returned from the `xy_fields` method.

The next line uses the X- and Y-components to compute the magnitude of the electric field using a two-argument expression. This solution component is stored in the provided name `"E_mag"`. We also compute the absolute value of both components.

Finally, the fields are exported to a VTK file for plotting. Multiple external tools are available to generate high-quality plots from the VTK data. Figure 1 shows an electric field magnitude generated using `FEM_2D` and VISIT.

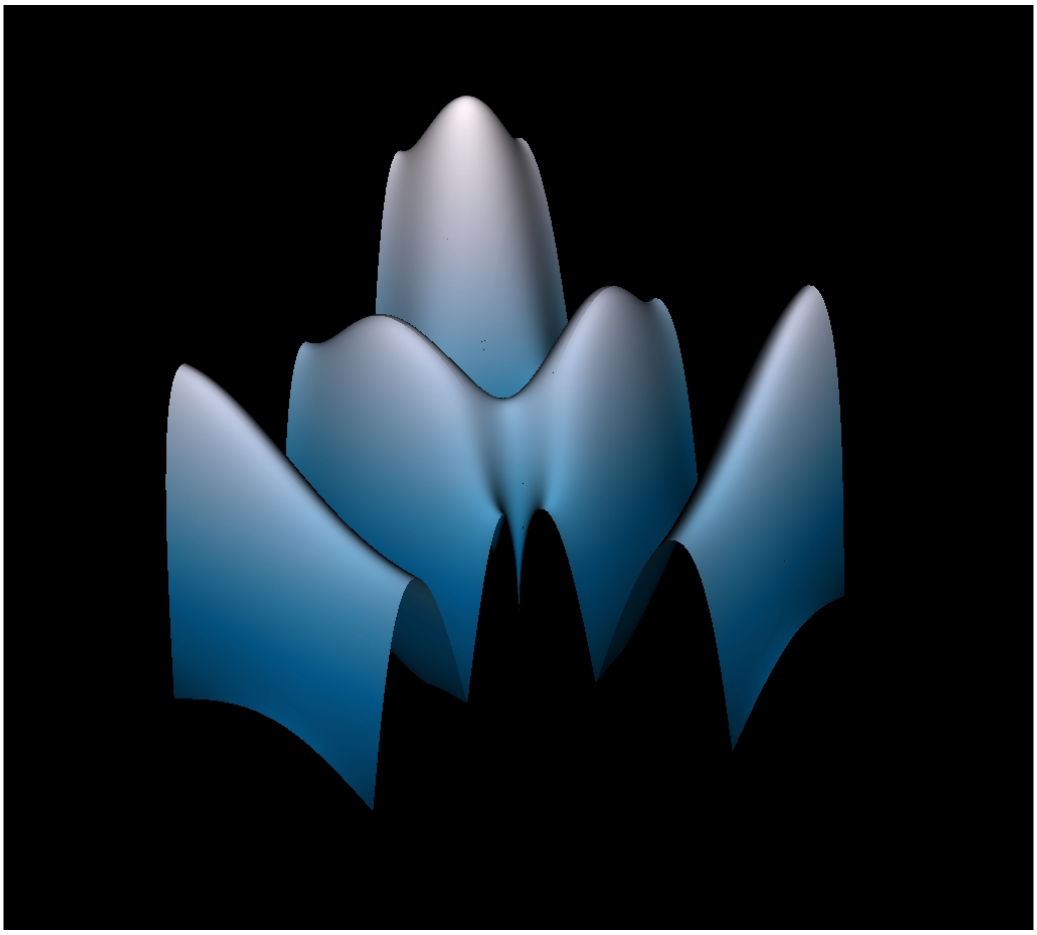

Figure 1: Example of an Electric Field Magnitude of an Eigenfunction

[2]There is also a need for an implementation with densities proportional to the size of the elements. This would be useful for generating images of the fields, as the overall point-density would be globally uniform across the domain.

References

Anderson, R., Andrej, J., Barker, A., Bramwell, J., Camier, J.-S., Dobrev, J. C. V., Dudouit, Y., Fisher, A., Kolev, Tz., Pazner, W., Stowell, M., Tomov, V., Akkerman, I., Dahm, J., Medina, D., & Zampini, S. (2021). MFEM: A modular finite element methods library. *Computers & Mathematics with Applications*, *81*, 42–74. https://doi.org/10.1016/j.camwa.2020.06.009

Arndt, D., Bangerth, W., Blais, B., Fehling, M., Gassmöller, R., Heister, T., Heltai, L., Köcher, U., Kronbichler, M., Maier, M., Munch, P., Pelteret, J.-P., Proell, S., Simon, K., Turcksin, B., Wells, D., & Zhang, J. (2021). The `deal.II` library, version 9.3. *Journal of Numerical Mathematics*, *29*(3), 171–186. https://doi.org/10.1515/jnma-2021-0081

Balay, S., Abhyankar, S., Adams, M. F., Benson, S., Brown, J., Brune, P., Buschelman, K., Constantinescu, E. M., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W. D., Hapla, V., Isaac, T., Jolivet, P., Karpeev, D., Kaushik, D., Knepley, M. G., Kong, F., … Zhang, J. (2021a). *PETSc Web page*. https://petsc.org/

Balay, S., Abhyankar, S., Adams, M. F., Benson, S., Brown, J., Brune, P., Buschelman, K., Constantinescu, E., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W. D., Hapla, V., Isaac, T., Jolivet, P., Karpeev, D., Kaushik, D., Knepley, M. G., Kong, F., … Zhang, J. (2021b). *PETSc/TAO users manual* (ANL-21/39 - Revision 3.16). Argonne National Laboratory.

Balay, S., Gropp, W. D., McInnes, L. C., & Smith, B. F. (1997). Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, & H. P. Langtangen (Eds.), *Modern software tools in scientific computing* (pp. 163–202). Birkhäuser Press. https://doi.org/10.1007/978-1-4612-1986-6_8

Corrado, J., Harmon, J., & Notaros, B. (2021). *A refinement-by-superposition approach to fully anisotropic hp-refinement for improved efficiency in CEM*. https://doi.org/10.36227/techrxiv.16695163.v1

Corrado, J., Harmon, J., & Notaros, B. (2022). *An adaptive anisotropic hp-refinement algorithm for the 2D Maxwell eigenvalue problem*. https://doi.org/10.36227/techrxiv.19636770.v1

Harmon, J., Corrado, J., & Notaros, B. (2021). *A refinement-by-superposition hp-method for H(curl)- and H(div)-conforming discretizations*. https://doi.org/10.36227/techrxiv.14807895.v1

Hernandez, V., Roman, J. E., & Vidal, V. (2005). SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Trans. Math. Software*, *31*(3), 351–362. https://doi.org/10.1145/1089014.1089019

Nalgebra: Linear algebra library for the Rust programming language. (2021). In *GitHub repository*. GitHub. https://github.com/dimforge/nalgebra