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

SUBMISSION DATE / POSTED DATE

13-02-2022 / 10-05-2022

CITATION

Corrado, Jeremiah; Harmon, Jake; Notaros, Branislav; Ilic, Milan M. (2022): FEM_2D: A Rust Package for 2D Finite Element Method Computations with Extensive Support for hp-refinement. TechRxiv. Preprint. https://doi.org/10.36227/techrxiv.19166339.v2

DOI

10.36227/techrxiv.19166339.v2

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 Colorado State University; Department of Electrical and Computer Engineering **2** University of Belgrade; School of Electrical Engineering

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Introduction

The Finite Element Method (FEM) is a powerful computation framework used to solve Partial Differential Equations (PDE)s on arbitrary geometries. In reality, physical systems behave in a continuous manner (both in space and time); however 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, the mathematically continuous dynamics can be expressed in terms of a system of linear equations. Linear algebra tools are then used to solve the problem 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 PDE's 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 force behind academic work within 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 a proving ground.

Although `FEM_2D` focuses on the Maxwell Eigenvalue problem specifically, it's functionality is intended to extend easily to other domains using a generic interface over basis function evaluation and integration. The module-structure of the library is also designed to be open to new features.

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 which 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.

The Journal of Open Source Software

Statement of Need

Efficiently computing FEM solutions over geometries with sharp edges or stark material discontinuities necessitates *hp*-refinement (whether isotropic or anisotropic). These situations tend to introduce multi-scale solution behavior which is challenging to model with pure *p*- or pure *h*-refinements, motivating combined hp-refinements (Harmon et al., 2021). Within the class of hp-refinements, the addition of anisotropic *hp*-refinements (over isotropic ones) presents a significantly larger capacity for solution efficiency, as small-scale behavior is targeted more directly and ineffectual Degrees of Freedom are left out of the system (Corrado et al., 2021). Increased efficiency in terms of the number of degrees of freedom is a key factor in the speed of large scale simulations, as well as the applicability of the method to smaller scale hardware (such as personal computers). Thus, a feature-rich anisotropic *hp*-refinement API is needed to enable efficient solution of challenging FEM problems. This is directly afforded by the underlying RBS methodology.

For research purposes, it is also important that the implementation is straightforward and easy to understand. This way, other researchers can quickly read the code to validate the methodology itself or they can use it as a starting point for additional investigation and software development. This is yet another benefit of the RBS approach, as it greatly simplifies the enforcement of continuity conditions, which is typically the most challenging aspect of an *h*-refinement implementation over quadrilateral or hexahedral elements.

Thus, we conclude that `FEM_2D`'s RBS implementation gives it a distinct advantage over other FEM libraries such as Deal.II (Arndt et al., 2021); specificity as a research package. The succinctness of the continuity enforcement algorithm removes much of the difficulty of implementing new features. This is a major barrier to entry for contributing to larger and more complex packages. Additionally, the generic Trait-Based interface makes it easy to leverage the advanced *hp*-refinement API against other domains of computational physics.

Features

hp-Refinement API:

`FEM_2D`'s primary offering to the FEM research community is its highly dynamic and expressive *hp*-refinement API. Unlike many other quadrilateral-element FEM packages, `FEM_2D` supports n-irregular anisotropic *h*-refinement as well as anisotropic *p*-refinement. In other words, there are far fewer limitations on the shape, location, or orientation of new elements when adding them to the Mesh. The polynomial expansion orders of the Basis Functions associated with each element can also be modified separately in each direction. This level of freedom would not be possible without the underlying RBS methodology.

The following example shows how some of the h-refinement methods may be used to modify a mesh structure. It is important to note that there are three primary h-refinement types which 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 which invoke a subsequent anisotropic refinement on one of the two child elements in the opposite direction. These are constructed with `HRef::U(Some(child_index))` and `HRef::V(Some(child_index))` respectively, where `child_index` must be either 0 or 1.

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 error types is provided in the documentation.

```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)
}
```

The following example shows how some of the p-refinement methods may be used. Here, the `Mesh` is provided as an argument rather than being loaded from a file. The p-refinements are constructed from the `PRef` Type 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, while the more explicit methods allow for failure.

```rust
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));
```

```rust
    // 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 example juxtaposes some of the functionality with the above p-refinement API.

Here, both methods can return an error, as it is possible to specify an invalid expansion order. These methods take a length-two array of u8's (8-bit unsigned integers), and thus preemptively remove the possibility of setting negative expansion orders, however, they still Err on expansion orders that are zero or too large.

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

fn set_some_expansion_orders(mesh: &mut Mesh) -> Result<(), PRefError> {
    // set the expansion order on all elems to (3, 3)
    mesh.set_global_expansion_orders([3, 3])?;

    // set the expansion orders to (4, 4) on all "leaf" elems
    // set the expansion orders to (2, 2) on all other elems
    mesh.set_expansions_with_filter(|elem| {
        if elem.has_children() {
            Some([2, 2])
        } else {
            Some([4, 4])
        }
    })?;

    Ok(())
}
```

The Journal of Open Source Software

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 depiction of how one might translate a mathematical problem into an `FEM_2D` implementation.

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:

$$\mathsf{U} = \{\mathbf{u}, \lambda\} \in B_0 \times \mathbb{R} \tag{1}$$

which satisfies:

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

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

The Generalized Eigenvalue Problem is built from a `Mesh` with the following code.

```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 which conforms to the provided continuity condition (only H(Curl) is currently implemented; however, 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 exponential 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]

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 (Hernandez et al., 2005) (Balay et al., 2021a) (Balay et al., 2021b) (Balay et al., 1997), 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 and eigenvector with length equal to the number of degrees of freedom in the domain.

Field Visualization

The Fields API allows us to compute a solution-field with an eigenvector and associated domain. It also allows functions of field solutions to be computed. 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_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())?;
```

[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 galerking sampling and integration functionality associated with alternate continuity conditions. These methods, structures, and traits should require minimal additions to the `Domain` structure, and no changes to the `Mesh` structure.

```
    // 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.

The following 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.

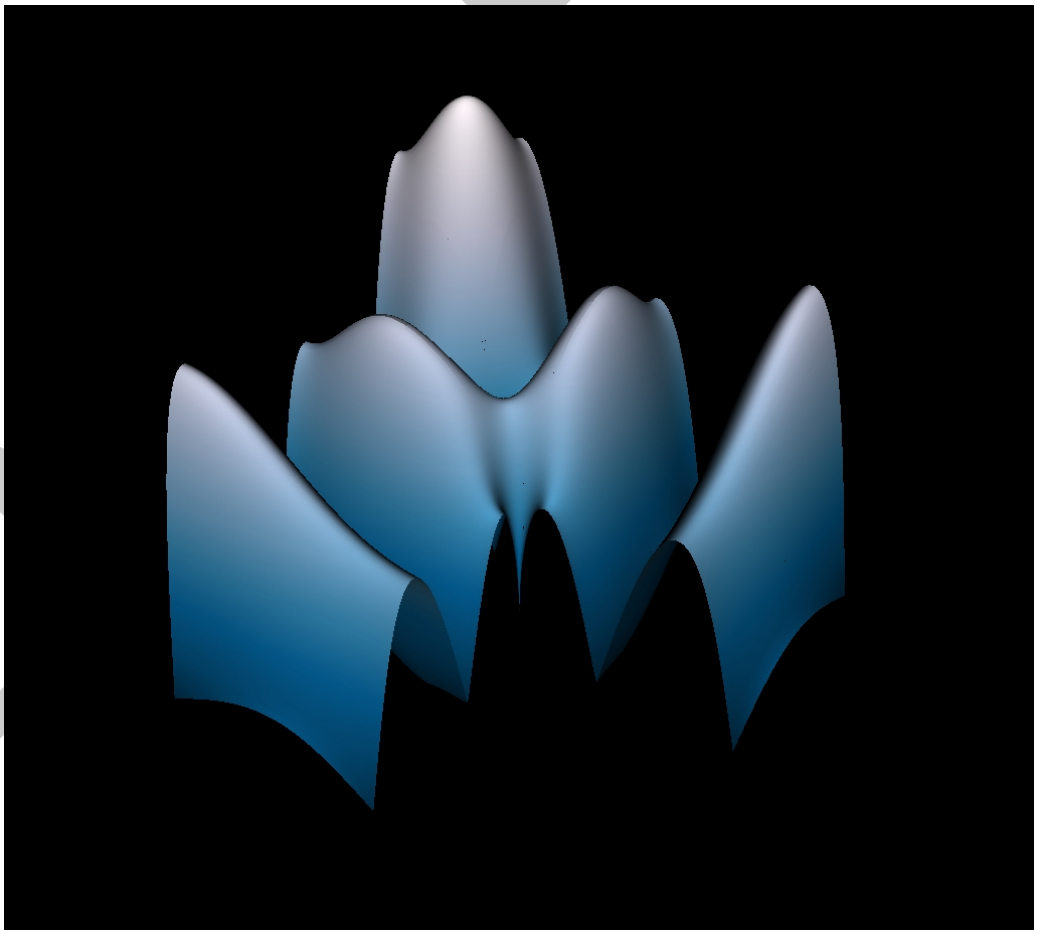

Figure 1: Example of an Electric Field Magnitude of an Eigenfunction

[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

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/. 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

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