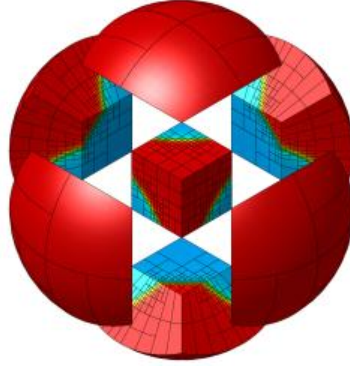

MFEM: A Modular Finite Element Methods Library

Robert Anderson[1], Andrew Barker[1], Jamie Bramwell[1], Jakub Cerveny[2], Johann Dahm[3], Veselin Dobrev[1], Yohann Dudouit[1], Aaron Fisher[1], Tzanio Kolev[1], Mark Stowell[1], and Vladimir Tomov[1]

[1]Lawrence Livermore National Laboratory
[2]University of West Bohemia
[3]IBM Research

July 2, 2018

Abstract

MFEM is a free, lightweight, flexible and scalable C++ library for modular finite element methods that features arbitrary high-order finite element meshes and spaces, support for a wide variety of discretization approaches and emphasis on usability, portability, and high-performance computing efficiency. Its mission is to provide application scientists with access to cutting-edge algorithms for high-order finite element meshing, discretizations and linear solvers. MFEM also enables researchers to quickly and easily develop and test new algorithms in very general, fully unstructured, high-order, parallel settings. In this paper we describe the underlying algorithms and finite element abstractions provided by MFEM, discuss the software implementation, and illustrate various applications of the library.

Contents

1 Introduction **3**

2 Overview of the Finite Element Method **4**

3 Meshes **9**
- 3.1 Conforming Meshes 10
- 3.2 Non-Conforming Meshes 11
- 3.3 NURBS Meshes 12
- 3.4 Parallel Meshes 12
- 3.5 Supported Input and Output Formats 13

4 Finite Element Spaces 13
4.1 Finite Elements . . . 14
4.2 Discrete de Rham Complex . . . 16
4.3 High-Order Spaces . . . 17
4.4 Visualization . . . 18

5 Finite Element Operators 18
5.1 Discretization Methods . . . 18
5.2 Finite Element Linear Systems . . . 19
5.3 Operator Decomposition . . . 23
5.4 High-Order Partial Assembly . . . 25

6 High-Performance Computing 27
6.1 Parallel Meshes, Spaces, and Operators . . . 27
6.2 Scalable Linear Solvers . . . 28
6.3 Vectorization and GPU Acceleration . . . 30
6.4 Matrix-Free Preconditioning . . . 31

7 Finite Element Adaptivity 32
7.1 Conforming Adaptive Mesh Refinement . . . 32
7.2 Non-Conforming Adaptive Mesh Refinement . . . 33
7.3 Mesh Optimization . . . 38

8 Applications 39
8.1 Example Codes . . . 40
8.2 Electromagnetics . . . 41
8.3 Compressible Hydrodynamics . . . 43
8.4 Other Applications . . . 44

9 Conclusions 44

Appendix A Bilinear Form Integrators 55
A.1 Scalar Field Operators . . . 55
A.2 Vector Finite Element Operators . . . 56
A.3 Nodal Vector Field Operators . . . 56
A.4 Discontinuous Galerkin Operators . . . 57
A.5 Special Purpose Integrators . . . 57

Appendix B Ordinary Differential Equation Integrators 59
B.1 Explicit Runge-Kutta Methods . . . 59
B.2 Implicit Methods . . . 60
B.3 Time-Dependent Operator . . . 62
B.4 Symplectic Integration Solvers . . . 62

Appendix C Letters of Support for Wilkinson Prize Nomination 63

Acknowledgments 78

1 Introduction

The Finite Element Method (FEM) is a powerful discretization technique that uses general unstructured grids to approximate the solutions of many partial differential equations (PDEs). It has been exhaustively studied, both theoretically and in practice, in the past several decades [34, 108, 39, 11, 41, 36]. High-order finite elements in particular are ideally suited to take advantage of the changing computational landscape because their order can be used to adjust the algorithm for different hardware or to fine tune the performance by increasing the computations performed with data transferred from memory (the FLOPs/byte ratio).

The MFEM project performs mathematical research and software development that aims to enable application scientists to take advantage of cutting-edge algorithms for high-order finite element meshing, discretizations, and linear solvers. MFEM also enables researchers and computational mathematicians to quickly and easily develop and test new research algorithms in very general, fully unstructured, high-order, parallel settings. The MFEM software library [88] features arbitrary high-order finite element meshes and spaces, support for a wide variety of discretization approaches and emphasis on usability, portability, and high-performance computing (HPC) efficiency. The MFEM code is freely available via OpenHPC, Spack and GitHub, `https://github.com/mfem`.

In this paper we provide an overview of some of the key mathematical algorithms and software design choices that have enabled MFEM to be widely applicable and highly performant from a relatively small and lightweight code base (Section 6 and Section 8). MFEM's main capabilities and their corresponding sections in the paper are outlined in the following paragraphs.

Conceptually, MFEM can be viewed as a finite element toolbox that provides the building blocks for developing finite element algorithms in a manner similar to that of MATLAB for linear algebra methods (Section 2). MFEM includes support for the full high-order de Rham complex: H^1-conforming, discontinuous (L^2), $H(\text{div})$-conforming, $H(\text{curl})$-conforming and NURBS finite element spaces in 2D and 3D (Subsection 4.2), as well as many bilinear, linear, and nonlinear forms defined on them (Appendix A) including linear operators such as gradient, curl, and embeddings between these spaces. It enables the quick prototyping of various finite element discretizations including: Galerkin methods, mixed finite elements, discontinuous Galerkin (DG), isogeometric analysis, hybridization, and discontinuous Petrov-Galerkin approaches (Subsection 5.1).

MFEM contains classes for dealing with a wide range of mesh types: triangular, quadrilateral, tetrahedral, and hexahedral, as well as surface and topologically periodic meshes (Section 3). It has general support for mesh refinement and optimization including local conforming and non-conforming adaptive mesh refinement (AMR) with arbitrary order hanging nodes, powerful node-movement mesh optimization, anisotropic refinement, derefinement, and parallel load balancing (Section 7). Arbitrary element transformations allowing for high-order mesh elements with curved boundaries are also supported. Some commonly used linear solvers, nonlinear methods, eigensolvers, and some explicit and implicit Runge-Kutta time integrators are also available (Appendix B).

MFEM supports Message Passing Interface (MPI)-based parallelism throughout the library and can readily be used as a scalable unstructured finite element problem generator (Subsection 6.1). MFEM-based applications have been scaled to hundreds of thousands of cores. The library supports efficient operator partial assembly and evaluation for tensor-product high-order elements (Subsection 5.4). A serial MFEM application typically requires minimal changes to transition to a scalable parallel version of the code where it can take advantage of the integrated scalable linear solvers from the *hypre* library, including the BoomerAMG, AMS, and ADS solvers (Subsection 6.2). Both versions can be further transitioned to high-performing templated variants where operator assembly/evaluation is fully inlined for particular runtime parameters.

Comprehensive support for a number of external numerical libraries, e.g., PETSc, SuperLU, STRUMPACK, SuiteSparse, SUNDIALS, and PUMI is also included, which gives access to many additional linear and nonlinear solvers, preconditioners, and time integrators. MFEM's meshes and solutions can be visualized with its lightweight native visualization tool GLVis [55], as well as with the VisIt [32, 114] visualization and analysis tool (Subsection 4.4).

MFEM is used in a number of applications in the U.S. Department of Energy, academia, and industry (Section 8). The object-oriented design of the library separates the mesh, finite element, and linear algebra abstractions, making it easy to extend and adapt to the needs of different simulations. The MFEM code base is relatively small and is written in highly portable C++ (e.g., with very limited use of templates and the STL). The serial version of MFEM has no required external dependencies and is straightforward to build on Linux, Mac, and Windows. The MPI-parallel version uses two third-party libraries (*hypre* [67] and METIS [71, 87]) and is easy to build with an MPI compiler. On most machines, both the serial and the parallel versions can be built in under a minute by typing: `make serial` and `make parallel`, respectively.

Some of the distinguishing features of MFEM compared to other finite element packages, such as deal.II [10], FEniCS [5], DUNE [22], FreeFem++ [60], Hermes [107], libMesh [72], FETK [65], etc., are its massively parallel scalability, HPC efficiency, support for arbitrary high-order finite elements, generality in mesh type and discretization methods, and the focus on maintaining a clean, lightweight code base. The continued development of MFEM is motivated by close work with a variety of researchers and application scientists resulting in a combination of features that is not available in any other finite element package. The wide applicability of the library is illustrated by the fact that over the last two years (2017-2018) it has been cited in journal articles, conference papers, and preprints covering topology optimization for additive manufacturing, compressible shock hydrodynamics, reservoir modeling, fusion-relevant electromagnetic simulations, space propulsion thrusters, radiation transport, space-time discretizations, PDEs on surfaces, parallelization in time, and algebraic multigrid methods [80, 33, 95, 53, 83, 18, 46, 16, 6, 96, 2, 8, 101, 86, 115, 56, 118, 77, 116, 106, 63, 21, 81, 102, 59, 104].

2 Overview of the Finite Element Method

This section is intended as a concise introduction to the finite element method. We define some of the basic finite element terminology and illustrate the corresponding MFEM software abstractions on a simple model problem. These are discussed in more detail in the following sections; consequently, computational scientists already familiar with finite elements may choose to skip this section.

The finite element method builds on concepts from the calculus of variations and linear algebra to discretize field variables and PDE operators into finite vector and matrix approximations. These vectors and matrices can then be assembled to create a system of algebraic equations with solutions that approximate the solutions to the PDE. While the theory of finite elements can be highly abstract, the application of the method to a given PDE is relatively straightforward and systematic.

The method begins by creating a geometric mesh of elements to represent the domain of the PDE. These elements can have a variety of shapes and can even have curved edges and faces. Next, the PDE is translated into a variational integral form (weak formulation), which along with a choice of basis functions can describe the solution locally on the elements with matrix and vector representations of the fields and operators of the PDE. These local matrices and vectors are then assembled together to create a system of equations that approximates the PDE. We will demonstrate this process in more detail by way of a basic example and explain all of the components and their

analogues in MFEM in the following paragraphs.

As an illustrative first example, consider Poisson's equation

$$-\Delta u = f \tag{1}$$

where u is an unknown scalar field and f is a given forcing function. This partial differential equation can be used to model a wide variety of physical phenomena, including steady-state heat conduction, electric potentials and gravitational fields. Adding simple homogeneous Dirichlet boundary conditions, we can state the *strong form* of the boundary value problem:

$$\begin{aligned} &\text{Find } u : \Omega \to \mathbb{R} \text{ such that} \\ -\Delta u &= f \quad \text{in } \Omega \\ u &= 0 \quad \text{on } \Gamma \end{aligned} \tag{2}$$

where $\Omega \subset \mathbb{R}^d$ is the domain of interest, Γ is its boundary, and $f : \Omega \to \mathbb{R}$ is the given source.

The finite element method employs a weak (integral) form of the governing equations instead of the above strong (differential) form. This approach enlarges the set of admissible solutions and allows the modeling of discontinuous fields which arise frequently in practice. One characteristic feature of the integral formulation is the introduction of *test functions* into the governing equations, which serves the purpose of locally weighting or *testing* the approximation. The use of weighting functions with local support allows us to solve the equations using an integral formulation and still retain local accuracy.

Multiplying (2) by an arbitrary test function v and integrating over the domain Ω yields

$$-\int_\Omega \Delta u \cdot v = \int_\Omega f\, v. \tag{3}$$

We then integrate by parts, i.e. apply the divergence theorem, to obtain

$$\int_\Omega \nabla u \cdot \nabla v - \int_\Gamma (\nabla u \cdot \mathbf{n}) v = \int_\Omega f\, v \tag{4}$$

where $\mathbf{n}$ is the outward normal of the boundary Γ. If we impose the boundary condition on the test function, $v = 0$ on Γ, the second term vanishes and we obtain the *weak form* of the boundary value problem

$$\begin{aligned} &\text{Find } u \in V \text{ such that} \\ &\int_\Omega \nabla u \cdot \nabla v = \int_\Omega f\, v \quad \forall v \in V \end{aligned} \tag{5}$$

where V is the space of admissible functions. In this example, that means all functions which are zero-valued on the boundary Γ and have integrate ("well-defined") first derivatives. In other words,

$$V = \left\{ v \in H^1(\Omega), v = 0 \text{ on } \Gamma \right\}, \tag{6}$$

where H^1 is a well-known Sobolev space [34, 25]. Introducing the notation

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \tag{7}$$

$$l(v) = \int_\Omega f\, v, \tag{8}$$

we can rewrite the above problem abstractly as

$$\begin{aligned}&\text{Find } u \in V \text{ such that}\\ &a(u,v) = l(v) \quad \forall v \in V\,.\end{aligned} \tag{9}$$

Here $a(u,v)$ is an example of a *bilinear form* since it is a mapping from $V \times V \to \mathbb{R}$ that is linear in both arguments separately. Similarly, $l(v)$ is an example of a *linear form* since it is a mapping from $V \to \mathbb{R}$ that is also linear in its argument. The form $a(\cdot,\cdot)$ is also symmetric in this case, $a(u,v) = a(v,u)$, but it is possible to have weak formulations where the space for u, the so-called *trial* space, is different than the space for v (the test space). It is also possible to apply variational formulations to nonlinear problems and handle other types of boundary conditions (Neumann, Robin).[1]

The above problem (5) is continuous, meaning that the solution space V is infinite dimensional. In order to compute an approximate solution on a computer using linear algebra, we approximate the solution u as

$$u \approx u_h = \sum_j^N c_j \varphi_j \tag{10}$$

where φ_j are basis functions of a finite dimensional subspace $V_h \subset V$ and c_j are scalar unknowns. We call φ_j *trial basis functions* and c_j *degrees of freedom* (DOFs). Endowing a mesh with a set of finite elements creates a space of functions. In MFEM, the objects corresponding to these concepts are the `Mesh`, the `FiniteElementCollection`, and the corresponding `FiniteElementSpace`.[2]

To define the basis functions φ_i, we subdivide the domain Ω into a finite number of *elements*. This subdivision of the domain is known as the computational *mesh*. The discrete points that define the mesh are the mesh *nodes*, e.g. the vertices of the mesh in the low-order case. Standard first order nodal basis functions are defined as having a value of one at a single node and a value of zero at all other nodes. The function is then linearly interpolated from the non-zero node to its neighbors. An example of a mesh and basis function is shown on the left in Figure 1. Higher-order functions have DOFs associated with more points in the element, not just the vertices, but also points on the edges, faces and the element interior[3]. The DOFs of some finite element spaces also may not correspond to values at particular points, e.g. scalar DOFs may represent a vector-valued function[4].

Substituting our approximation (10) into the weak form (5) we get

$$\sum_j c_j \int_\Omega \nabla\varphi_j \cdot \nabla v = \int_\Omega f\, v \quad \forall v \in V. \tag{11}$$

Choosing $\varphi_i \in V_h \subset V$ as our test functions, known as the Galerkin method, we obtain the finite dimensional problem

$$\sum_j c_j \int_\Omega \nabla\varphi_j \cdot \nabla\varphi_i = \int_\Omega f\, \varphi_i \quad \forall \varphi_i \in V_h. \tag{12}$$

[1] See Subsection 5.1 for further examples.
[2] These are discussed in detail in Section 3 and Section 4.
[3] See Subsection 4.3.
[4] See Subsection 4.2.

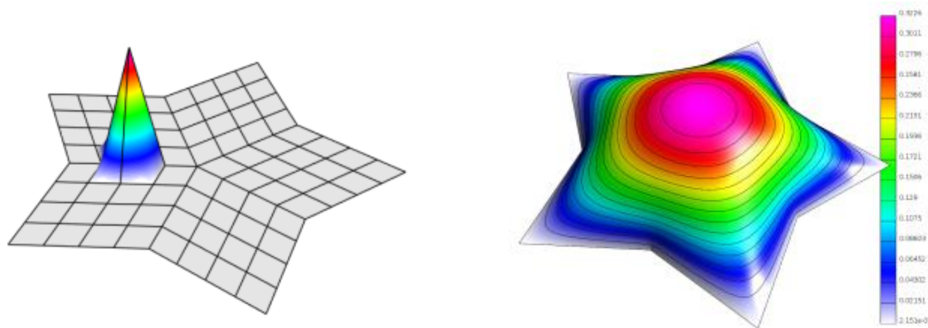

Figure 1: Left: 2D computational mesh and one of the first order nodal basis functions φ_i. Right: The finite element discrete approximation visualized with GLVis [55].

Defining the linear algebra objects

$$\int_\Omega \nabla\varphi_j \cdot \nabla\varphi_i \to A_{ij} \tag{13}$$

$$\int_\Omega f\,\varphi_i \to b_i \tag{14}$$

$$c_j \to x_j, \tag{15}$$

we arrive at the final discrete system of linear equations

$$Ax = b \tag{16}$$

which we can solve on a computer for x to approximate the solution of the original PDE. The size of the above linear system depends on the number of DOFs in the finite element spaces, and generally the finer the mesh (or higher the order), the larger the linear system will be, and the closer the discrete approximate to the continuous solution of the weak form of the PDE. These types of finite element linear systems are discussed in detail in Subsection 5.2.

Mappings (13) and (14) represent key components of the MFEM library. The `BilinearForm` object, defined over a `FiniteElementSpace`, supplied with a `BilinearFormIntegrator`, generates a `Matrix`. Similarly, in (14), a `LinearForm` object, defined over a `FiniteElementSpace`, supplied with a `LinearFormIntegrator`, generates a `Vector`. In this way, it is useful to think about MFEM as a library for building *finite element objects* and using them to generate *linear algebra objects* which can be used to solve the resulting systems of equations.

As an example of the implementation of the Poisson problem in MFEM, consider the file `examples/ex1.cpp` in the MFEM distribution. We first read the mesh from the given mesh file. We can handle triangular, quadrilateral, tetrahedral, hexahedral, surface and volume meshes with the same code.[5]

```
75  Mesh *mesh = new Mesh(mesh_file, 1, 1);
76  int dim = mesh->Dimension();
```

We then refine our coarse mesh to improve the accuracy of our solution. In this example we do `ref_levels` of uniform refinement. We choose `ref_levels` to be the largest number that gives a final mesh with no more than $50,000$ elements.[6]

[5] See Section 3 for discussion of more general meshes.

[6] Advanced forms of refinement are discussed in Section 7.

```
83  int ref_levels = (int)floor(log(50000./mesh->GetNE())/log(2.)/dim);
84  for (int l = 0; l < ref_levels; l++)
85  {
86     mesh->UniformRefinement();
87  }
```

We now define a FiniteElementCollection which determines the type of basis functions we want on a reference unit master element, e.g. a unit square or triangle. For this example, we use the H^1 Sobolev space defined in (6). Note that this collection can be of arbitrary order and dimension.

```
94  FiniteElementCollection *fec;
95  fec = new H1_FECollection(order, dim);
```

Once the FiniteElementCollection is defined, we can define a FiniteElementSpace which maps the basis functions on a reference element to each physical element in our mesh.

```
108 FiniteElementSpace *fespace = new FiniteElementSpace(mesh, fec);
```

We can then determine the list of essential (known-value) boundary degrees of freedom. In this example, the boundary conditions are defined by marking all the boundary attributes from the mesh as essential (Dirichlet or fixed) and converting them to a list of DOFs.

```
116 Array<int> ess_tdof_list;
117 if (mesh->bdr_attributes.Size())
118 {
119    Array<int> ess_bdr(mesh->bdr_attributes.Max());
120    ess_bdr = 1;
121    fespace->GetEssentialTrueDOFs(ess_bdr, ess_tdof_list);
122 }
```

Finally, we can define our finite element objects given in (7). First, we set up the linear form $l(\cdot)$ which corresponds to the right-hand side of the FEM linear system. In this case it is $\int_\Omega 1\, \varphi_i$ where φ_i are the basis functions in the finite element space, fespace.

```
127 LinearForm *b = new LinearForm(fespace);
128 ConstantCoefficient one(1.0);
129 b->AddDomainIntegrator(new DomainLFIntegrator(one));
130 b->Assemble();
```

Second, we set up the bilinear form $a(\cdot,\cdot)$ on the finite element space corresponding to the Laplacian operator $(-\Delta)$ by adding the diffusion domain integrator with coefficient 1.

```
141 BilinearForm *a = new BilinearForm(fespace);
142 a->AddDomainIntegrator(new DiffusionIntegrator(one));
```

The next stage is transforming these finite element objects into linear algebra objects. We first define the solution vector x, which is a GridFunction initialized with the known essential boundary condition of zero.

```
135 GridFunction x(fespace);
136 x = 0.0;
```

We then assemble the bilinear form and the corresponding linear system, applying any necessary transformations such as eliminating boundary conditions.[7]

[7]See Subsection 5.2.

```
149   a->Assemble();
150
151   SparseMatrix A;
152   Vector B, X;
153   a->FormLinearSystem(ess_tdof_list, x, *b, A, X, B);
```

We have now formed the linear algebra system of equations $Ax = b$. To solve the system, we use an iterative conjugate gradient scheme with a simple Gauss-Seidel preconditioner.[8]

```
160   GSSmoother M(A);
161   PCG(A, M, B, X, 1, 200, 1e-12, 0.0);
```

After computing the solution, we can map it back from the linear algebra to the finite element representation using (10).

```
171   a->RecoverFEMSolution(X, *b, x);
```

We can save the refined mesh and the computed solution.

```
175   ofstream mesh_ofs("refined.mesh");
176   mesh_ofs.precision(8);
177   mesh->Print(mesh_ofs);
178   ofstream sol_ofs("sol.gf");
179   sol_ofs.precision(8);
180   x.Save(sol_ofs);
```

And we can finally visualize the results, e.g., using MFEM's native visualization tool, GLVis [55] as illustrated on the right in Figure 1.[9]

```
183   if (visualization)
184   {
185      char vishost[] = "localhost";
186      int  visport   = 19916;
187      socketstream sol_sock(vishost, visport);
188      sol_sock.precision(8);
189      sol_sock << "solution\n" << *mesh << x << flush;
190   }
```

Extensions to parallel settings[10], high-performance computing algorithms[11], and various applications[12] are discussed in the following sections. Those sections assume closer familiarity with finite elements, so we recommend the following excellent as additional references: [69, 52, 108, 41, 24, 34, 25, 109, 119, 54, 89].

3 Meshes

The main mesh classes in MFEM are: `Mesh` for a serial mesh and `ParMesh` for an MPI-distributed parallel mesh. The class `ParMesh` is derived from `Mesh` and extends the local mesh representation (corresponding to the inherited `Mesh` data and interface) with data and functionality representing the mesh connections across MPI ranks (see Subsection 6.1).

In this section we describe the internal representation aspects of these two classes. The mesh manipulation capabilities (refinement, derefinement, etc.) will be described later in Section 7.

[8] More powerful preconditioners are discussed in Subsection 6.2.
[9] See also Subsection 4.4.
[10] Subsection 6.1.
[11] Section 6.
[12] Section 8.

3.1 Conforming Meshes

The definition of a serial (or a local component in parallel), unstructured, conforming mesh in MFEM consists of two parts: *topological* (connectivity) data and *geometrical* (coordinates) data.

The *primary* topology data are: number of vertices, number of elements, number of boundary elements; vertices have no topological description; each element has a type (triangle, quad, tetrahedron, etc.), an attribute (an integer), and a tuple of vertex indices; boundary elements are described in the same way, with the assumption that they define elements with dimension one less than the dimension of the regular elements. Any additional topological data — such as edges, faces, and their connections to the elements, boundary elements and vertices — is derived internally from the primary data. Most of the basic topological and geometrical data for the reference elements can be found in the files `fem/geom.hpp` and `fem/geom.cpp`. For example, the canonical orderings of the vertices of the reference elements are defined in the constructor of class `Geometry`.

The geometrical locations of the mesh entities can be described in two ways: (1) define the coordinates of all vertices, and (2) define a `GridFunction` called nodal grid function, or simply *nodes*. Clearly, the first approach can only be used when describing a *linear* mesh. In the second case, the `GridFunction` class is the same class that MFEM uses to describe any finite element function/solution. In particular, it defines (a) the basis functions mapping each reference element to physical space, and (b) the coefficients multiplying the basis functions in the finite element expansion – we refer to these as nodal coordinates, control points, or nodal degrees of freedom of the mesh. The nodal geometric description is much richer than the first case of vertex coordinates: it allows nodes to be associated not only with the mesh vertices but with the edges, faces, and the interiors of the elements as well, see Figure 2.

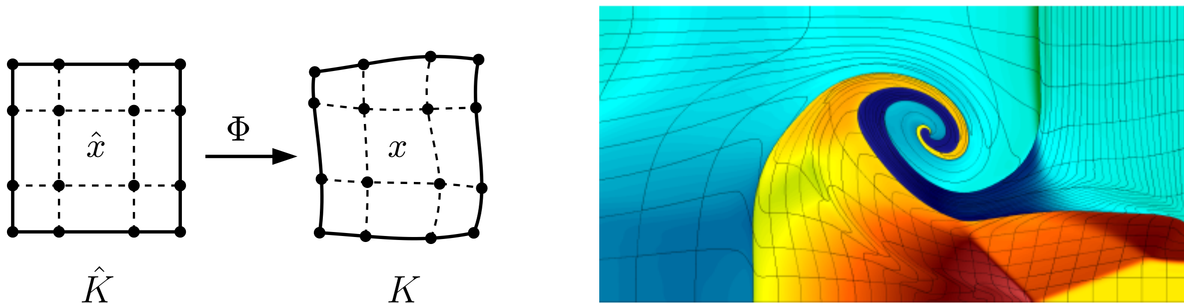

Figure 2: Left: The mapping Φ from the reference element $\hat{K}$ to a bi-cubic element K in physical space with high-order nodes shown as black dots. Right: Example of a highly deformed high-order mesh from a Lagrangian hydrodynamics simulation (see Subsection 8.3).

The exact shape of an element is defined through a mapping $\Phi \equiv \Phi_K : \hat{K} \to K$ from the reference element $\hat{K}$, as shown in Figure 2. The mapping Φ itself is defined in terms of the basis functions $\{w_i(\hat{x})\}_{i=1}^N$ (typically polynomials) and the local nodal coordinates $\mathbf{x}_K$ (extracted/derived from the global nodal vector $\mathbf{x}$)

$$x(\hat{x}) = \Phi(\hat{x}) = \sum_{i=1}^{N} \mathbf{x}_{K,i} w_i(\hat{x}) \,. \tag{17}$$

Both $\{w_i\}$ and $\{\mathbf{x}_K\}$ are defined from the geometric mesh description — either the vertex coordinates with linear (bi-linear for quads, or tri-linear for hexes) polynomials, or the nodal `GridFunction` with its respective definition of basis functions and node coordinates. Typically, the basis functions

$\{w_i\}$ are scalar functions and the coefficients $\{\mathbf{x}_{K,i}\}$ are small vectors of the same dimension as $x \in K \equiv \Phi(\hat{K})$. In MFEM, the mapping Φ, for a particular element K, is represented by the class `ElementTransormation`. The element transformation for an element K can be obtained directly from its `Mesh` object using the method `GetElementTransformation(k)`, where `k` is the index of the element K in the mesh. Once constructed, the `ElementTransformation` object can be used for computing the physical coordinates of any reference point, the Jacobian matrix of the mapping, the integration weight associated with the change of the variables from K to $\hat{K}$, etc. All of these operations generally depend on a reference point of interest which is typically a quadrature point in a quadrature rule. This motivates the use of class `IntegrationPoint` to represent reference points.

Note that MFEM meshes distinguish between the dimension of the reference space of all regular elements (reference dimension) and the dimension of the space into which they are mapped (spatial dimension). This way, surface meshes are naturally supported with reference and space dimensions of 2 and 3, respectively.

3.2 Non-Conforming Meshes

Non-conforming meshes, also referred to as meshes with hanging nodes, can be viewed as conforming meshes (as described above) with a set of constraints imposed on some of their vertices. Typically, the requirement is that each constrained vertex has to be the convex combination of a set of parent vertices. Note that, in general, the parent vertices of a constrained vertex can be constrained themselves. However, it is usually required that all the dependencies can be uniquely resolved and all constrained vertices can be expressed as linear combinations of non-constrained ones, see Subsection 7.2 for more details.

The need for such non-conforming meshes arises most commonly in the local refinement of quadrilateral and hexahedral meshes. In such scenarios, an element that is refined shares a common entity (edge or face that the first element needs to refine) with another element that does not need to refine the shared entity. To restrict the propagation of the refinement, the first element introduces one or more constrained vertices on the shared entity and constrains them in terms of the vertices of the shared entity. The goal of the constraint is to ensure that the refined sub-entities introduced by the refinement of the first element are completely contained inside the original shared entity. In simpler terms, the goal is to make sure that the mesh remains "watertight", i.e. there are no gaps or overlaps in the refined mesh.

When working with high-order curved meshes, or high-order finite element spaces on linear non-conforming meshes, one has to replace the notion of constrained vertices with constrained degrees of freedom. The goal of the constraints is still the same — ensure there are no gaps or overlaps in the refined mesh; or, in the case of high-order spaces — ensure that the constrained non-conforming finite element space is still a subspace of the discretized continuous space, H^1, $H(\mathrm{div})$, etc. High-order finite elements are further discussed in Subsection 4.3.

The observation that a non-conforming mesh can be represented as a conforming mesh plus a set of linear constraints on some of its nodes, is the basis for the handling of non-conforming meshes in MFEM. Specifically, the `Mesh` class represents the topology of the conforming mesh (which we refer to as the "cut" mesh) while the constraints on the mesh nodes are explicitly imposed on the nodal `GridFunction` which contains both the unconstrained and the constrained degrees of freedom. In order to store the additional information about the fact that the mesh is non-conforming, the `Mesh` class stores a pointer to an object of class `NCMesh`. For example, `NCMesh` stores the full refinement hierarchy along with all parent-child relations for non-conforming edges and faces, while `Mesh` simply represents the current mesh consisting of the leaves of the full hierarchy, see [31].

Notable features of the `NCMesh` class include its ability to perform both isotropic and anisotropic

refinement of quadrilateral and hexahedral meshes while supporting an arbitrary number of refinements across a single edge or face (i.e. arbitrary level of hanging nodes).

3.3 NURBS Meshes

Non-Uniform Rational B-Splines (NURBS) are often used in geometric modeling. In part, this is due to their capability to represent conic sections exactly. In the last decade, the use of NURBS discrete functions for PDE discretization has also become popular and is often referred to as IsoGeometric Analysis (IGA), see [36].

In principle, the construction of NURBS meshes and discrete spaces is very similar to the case of high-order polynomials. For example, a NURBS mesh can be viewed as a quadrilateral (in 2D) or hexahedral (in 3D) mesh where the basis functions are tensor products of 1D NURBS basis functions. However, an important distinction is that the nodal degrees of freedom are no longer associated with edges, faces, or vertices. Instead, the nodal degrees of freedom (usually called control points in this context) can participate in the description of multiple layers of elements — a fact that follows from the observation that NURBS basis functions have support (i.e. are non-zero) inside of blocks of $(p+1)\times(p+1)$ (2D) and $(p+1)\times(p+1)\times(p+1)$ (3D) elements (here p is the degree/order of the NURBS space).

In MFEM, NURBS meshes are represented internally through the class `NURBSExtension` which handles all NURBS-specific implementation details such as constructing the relation between elements and their degrees of freedom. However, from the user perspective, a NURBS mesh is still represented by the class `Mesh` (with quadrilateral or hexahedral elements) which, in this case, has a pointer to an object of type `NURBSExtension` and a nodal `GridFunction` that defines the appropriate NURBS basis functions and control points. Most MFEM examples can directly run on NURBS meshes, and some of them also support IGA discretizations. As of version 3.4, MFEM can also handle variable-order NURBS, see the examples in the `miniapps/nurbs` directory.

3.4 Parallel Meshes

As mentioned in the beginning of this section, an MPI-distributed parallel mesh is represented in MFEM by the class `ParMesh` which is derived from class `Mesh`. The data structures and functionality inherited from class `Mesh` represent the local (to the MPI task) portion of the mesh. Note that each element in the global mesh is assigned to exactly one MPI rank, so different processors cannot own/share the same element; however they can share mesh entities of lower dimensions: faces (in 3D), edges (in 2D and 3D), and vertices (in 3D, 2D, and 1D). In MFEM, such mesh entities are called *shared*.

The standard way to construct a `ParMesh` in MFEM is to start with a serial `Mesh` object and a partitioning array that assigns an MPI rank to each element in the mesh. By default, the partitioning array is constructed using the METIS graph partitioner [71, 87] where mesh elements are the vertices of the partitioned graph, and the graph edges correspond to the internal faces (3D), edges (2D) and vertices (1D) connecting two adjacent mesh elements.

Given the partitioning array, each shared entity can be associated with a unique set of processors, namely, the set of processors that share that entity. Such sets of processors, corresponding to at least one shared entity, are called *processor groups* or simply *groups*. Each MPI rank constructs its own set of groups and represents it with an object of class `GroupTopology` which represents the communication connections of each rank with its (mesh) neighbors. Inside each group one of the processors is selected as the *master* for the group. This choice must be made consistently by all processors in the group. For example, MFEM assigns the processor with the lowest rank in the

group to be the master.

In order to maintain a consistent mesh description across processors, it is important to ensure that shared entities are described uniformly across all MPI tasks in the shared entity group. For example, since `ParMesh` does not define a global numbering of all vertices, a shared triangle with local vertex indices (a, b, c) on processor A must be described on processor B as (x, y, z) such that the shared vertex with index x on processor B is the same as the shared vertex with index a on processor A, and similarly for the indices y and z. This uniformity must be ensured during the construction of the `ParMesh` object and maintained later, e.g. during mesh refinement.

For this reason, shared entities are stored explicitly (as tuples of local vertex indices) on each processor. In addition, the shared entities are ordered by their dimension (vertices, edges, faces) and by their group, making it easier to maintain consistency across processors.

The case of parallel non-conforming meshes is treated similarly to the serial case: the `ParMesh` object is augmented by an object of class `ParNCMesh` which inherits from `NCMesh` and provides all required parallel functionality. This is discussed in more detail later in Subsection 7.2.

The case of parallel NURBS meshes is also treated similarly to the serial case: the `ParMesh` object is augmented with an object of class `ParNURBSExtension` which inherits from `NURBSExtension`. Note that, currently, MFEM does not support parallel refinement of NURBS meshes.

3.5 Supported Input and Output Formats

There are two MFEM native ASCII formats: the first one is for generic meshes (non-NURBS) and a second one which is specific for NURBS meshes. These are the default formats used when writing a mesh to a C++ output stream (`std::ostream`) or when calling the `Print()` method of class `Mesh` or `ParMesh`. Note that the cross-processor connectivity in a parallel mesh is lost when using the `Print()` method which, however, is not required for visualization purposes. To save a parallel mesh with all cross-processor connections, one needs to use the method `ParMesh::ParPrint()`.

Other *input* formats supported by class `Mesh` are: Netgen, TrueGrid, unstructured VTK, Gmsh (linear elements only), and Genesis format (produced by Cubit, supported when NetCDF support is enabled in MFEM). Class `Mesh` also provides output support for the unstructured VTK format through the method `PrintVTK()`.

For more comprehensive input/output, where a mesh is stored with any number of finite element solution fields, MFEM defines the class `DataCollection` along with several derived classes: `VisItDataCollection`: writes an additional `.mfem_root` file that can be opened by the MFEM plugin in VisIt [32, 114]; `SidreDataCollection`: a set of data formats based on the Sidre component of LLNL's Axom library which, in particular, supports binary I/O and can also be opened by VisIt; and `ConduitDataCollection`: a set of data formats based on LLNL's Conduit library [35] which also supports binary I/O and can be opened by VisIt. Note that both classes `DataCollection` and `VisItDataCollection` use the default ASCII format to save the mesh and finite element solution fields.

4 Finite Element Spaces

In this section, we introduce and describe the main classes (in addition to the mesh classes described in Section 3) required for the full definition of any finite element discretization space: the class `FiniteElement` with its derived classes, the class `FiniteElementCollection` with its derived classes, and finally the class `FiniteElementSpace`. In addition, we describe the class `GridFunction` which represents a particular discrete function in a finite element space.

4.1 Finite Elements

The concept of a *finite element* is represented in MFEM by the abstract base class `FiniteElement`. The main characteristics of the class are the following.

Reference element. This is the precise definition of the reference geometric domain along with descriptions of its vertices, edges, faces, and how they are ordered. As previously mentioned, this information is included in class `Geometry` implemented in the files `fem/geom.hpp` and `fem/geom.cpp`. In the `FiniteElement` class this information is represented by an integer referring to one of the constants in the enumeration `Geometry::Type`: `POINT`, `SEGMENT`, `TRIANGLE`, `SQUARE`, `TETRAHEDRON`, or `CUBE`. This data member can be accessed via the method `GetGeomType()`. The respective dimension of the reference element can be accessed via the method `GetDim()`.

Map type. This is an integer given by one of the constants: `VALUE`, `INTEGRAL`, `H_DIV`, and `H_CURL` defined in the `FiniteElement` class. These constants represent one of the four ways a function on the reference element $\hat{K}$ can be transformed into a function on any physical element K through a transformation $\Phi : \hat{K} \to K$. The four choices are:

`VALUE` This map-type can be used with both scalar- and vector-valued functions on the reference element: assume that $\hat{u}(\hat{x})$, $\hat{x} \in \hat{K}$ is a given function, then the transformed function $u(x)$, $x \in K$ is defined by

$$u(x) = \hat{u}(\hat{x}) \,, \qquad \text{where} \qquad x = \Phi(\hat{x}) \,.$$

`INTEGRAL` This map-type can be used with both scalar- and vector-valued functions on the reference element: assume that $\hat{u}(\hat{x})$, $\hat{x} \in \hat{K}$ is a given function, then the transformed function $u(x)$, $x \in K$ is defined by

$$u(x) = \frac{1}{w(\hat{x})} \hat{u}(\hat{x}) \,, \qquad \text{where} \qquad x = \Phi(\hat{x}) \,,$$

and $w(\hat{x})$ is the transformation weight factor derived from the Jacobian $J(\hat{x})$ of the transformation $\Phi(\hat{x})$, which is a matrix of dimensions $d \times \hat{d}$ (where $\hat{d} \le d$ are the dimensions of the reference and physical spaces, respectively):

$$w(\hat{x}) = \begin{cases} \det(J(\hat{x})) & \text{when } \hat{d} = d, \text{ i.e. } J \text{ is square} \\ \det(J(\hat{x})^T J(\hat{x}))^{\frac{1}{2}} & \text{otherwise.} \end{cases}$$

This mapping preserves integrals over mapped subsets of $\hat{K}$ and K.

`H_DIV` This map-type can be used only with vector-valued functions on the reference element where the number of the vector components is $\hat{d}$, i.e. the reference element dimension: assume that $\hat{u}(\hat{x})$, $\hat{x} \in \hat{K}$ is such a function, then the transformed function $u(x)$, $x \in K$ is defined by

$$u(x) = \frac{1}{w(\hat{x})} J(\hat{x}) \hat{u}(\hat{x}) \,, \qquad \text{where} \qquad x = \Phi(\hat{x}) \,,$$

and $w(\hat{x})$ and $J(\hat{x})$ are as defined above. This is the *Piola* transformation used for mapping $H(\text{div})$-conforming basis functions. This mapping preserves the integrals of the normal component over mapped $(\hat{d}-1)$-dimensional submanifolds of $\hat{K}$ and K.

`H_CURL` This map-type can be used only with vector-valued functions on the reference element where the number of the vector components is $\hat{d}$, i.e. the reference element dimension: assume that $\hat{u}(\hat{x})$, $\hat{x} \in \hat{K}$ is such a function, then the transformed function $u(x)$, $x \in K$ is defined by

$$u(x) = \begin{cases} J(\hat{x})^{-T} \hat{u}(\hat{x}) & \text{when } \hat{d} = d, \text{ i.e. } J \text{ is square} \\ J(\hat{x}) [J(\hat{x})^T J(\hat{x})]^{-1} \hat{u}(\hat{x}) & \text{otherwise,} \end{cases} \qquad \text{where} \qquad x = \Phi(\hat{x}) \,,$$

and $w(\hat{x})$ and $J(\hat{x})$ are as defined above. This is the *Piola* transformation used for mapping $H(\text{curl})$-conforming basis functions. This mapping preserves the integrals of the tangential component over mapped 1D paths.

There is a connection between the way a function is mapped and how its gradient, curl or divergence is mapped: if a function is mapped with the `VALUE` map type, then its gradient is mapped with `H_CURL`; if a vector function is mapped with `H_CURL`, then its curl is mapped with `H_DIV`; and finally, if a vector function is mapped with `H_DIV`, then its divergence is mapped with `INTEGRAL`.

The map type can be accessed with the method `GetMapType()`. In MFEM, the map type also determines the type of basis functions used by the `FiniteElement`: scalar (for `VALUE` or `INTEGRAL` map types) or vector (for `H_CURL` or `H_DIV` map types).

Degrees of freedom. The number of the degrees of freedom in a `FiniteElement` can be obtained using the method `GetDof()` which is also the number of basis functions defined by the finite element. Each degree of freedom i has an associated point in reference space, called its node (i-th node). For many scalar finite elements, evaluating the j-th basis function at the i-th node gives δ_{ij} (the Kronecker delta). However, that is generally not required.

The basis functions can all be evaluated simultaneously at a single reference point, given as an `IntegrationPoint`, using the virtual method `CalcShape()` for scalar finite elements or `CalcVShape()` for vector finite elements. Similarly, based on the specific finite element type, the gradient, curl, or divergence of the basis functions can be evaluated with the method `CalcDShape()`, `CalcCurlShape()`, or `CalcDivShape()`, respectively.

In order to simplify the construction of a global enumeration for the DOFs, each local DOF in a `FiniteElement` is associated with one of its vertices, edges, faces, or the element interior. Then the local DOFs are ordered in the following way: first all DOFs associated with the vertices (in the order defined by the reference element), then all edge DOFs following the order and orientation of the edges in the reference element, and then similarly the face DOFs, and finally, the interior DOFs. This local ordering is then easier to translate to the global mesh level where global DOFs are numbered in a similar manner but now traversing all mesh vertices first, then all mesh edges, then all mesh faces, and finally all element interiors.

For vector finite elements, in addition to the node, each DOF i has an associated $\hat{d}$-dimensional vector, $\vec{r}_i$. For DOF i, associated with a non-interior entity (usually edge or face) the vector $\vec{r}_i$ is chosen to be either normal or tangential to the face/edge based on its map type: `H_DIV` or `H_CURL`, respectively. The role of these associated vectors is to define the basis functions on the reference element, so that evaluating the j-th vector basis function at the i-th node and then computing the dot product with the vector $\vec{r}_i$ gives δ_{ij}. Note that the vectors $\vec{r}_i$ have to be scaled appropriately in order to preserve the rotational symmetries of the basis functions.

The main classes derived from the base `FiniteElement` class are the arbitrary order H^1-conforming (with class names beginning with `H1`), the L^2-conforming (i.e. discontinuous, with class names beginning with `L2`), the $H(\text{curl})$-conforming (with class names beginning with `ND`, short for *Nedelec*), and the $H(\text{div})$-conforming (with class names beginning with `RT`, short for *Raviart-Thomas*) finite elements. All of these elements are defined for all reference element types where they make sense. For an illustration see Figure 3.

In addition to the methods for evaluating the basis functions and their derivatives, class `FiniteElement` introduces a number of other useful methods. Among these are: methods to support mesh refinement: `GetLocalInterpolation()` and `GetTransferMatrix()`; methods to support finite element interpolation/projection: `Project()` (scalar and vector version), `ProjectMatrixCoefficient()`; and methods to support the evaluation of discrete operators such as embedding, gradient, curl, and divergence: `ProjectGrad()`, `ProjectCurl()`, etc.

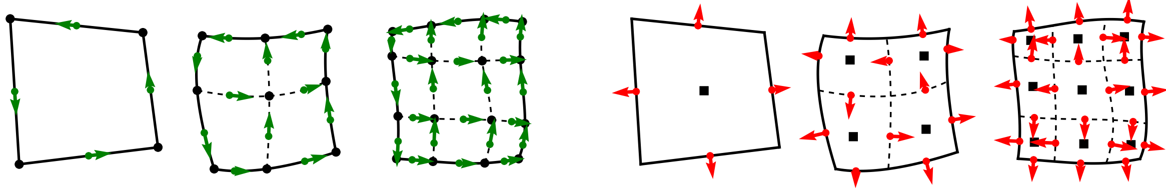

Figure 3: Linear, quadratic and cubic H^1 finite elements and their respective $H(\text{curl})$, $H(\text{div})$ and L^2 counterparts in 2D. Left: The black dots and the green arrows represent the H^1 and the $H(\text{curl})$ DOFs, respectively. Right: The red arrows and the black squares represent the $H(\text{div})$ and the L^2 DOFs, respectively.

In order to facilitate mesh-type independent programming, on one hand, and to define any required permutations of DOFs shared by neighbor elements in the process of mapping global DOFs to local DOFs, on the other hand, MFEM introduces the abstract base class `FiniteElementCollection`. Its main functionality is to (1) define a specific finite element for every mesh entity type, and (2) define a permutation for the DOFs on any mesh entity type, based on the *orientation* of that entity relative to any other orientation of that entity; these orientations correspond to the different permutations of the vertices of the entity, as seen from the points of view of adjacent elements.

The main classes derived from `FiniteElementCollection` are the arbitrary order `*_FECollection` classes where the `*`-prefix is one of `H1`, `L2`, `ND`, or `RT` which combine the appropriate finite element classes with the respective prefix for all different types of reference elements. Note that in the case of `RT_FECollection`, the regular (non-boundary) elements use `RT_*` finite elements, however, the edges (2D) or faces (3D) use `L2_*` elements with `INTEGRAL` map-type. In addition to these "standard" `FiniteElementCollection`s, MFEM defines also *interfacial* collections used for defining spaces on the mesh skeleton/interface which consists of all lower-dimensional mesh entities, excluding the regular full-dimension mesh elements. These collections can be used to define discrete spaces for the traces (on the mesh skeleton) of the regular H^1, $H(\text{curl})$, and $H(\text{div})$ spaces.

4.2 Discrete de Rham Complex

In MFEM, the mathematical concept (or definition) of a discrete finite element function space is encapsulated in the class `FiniteElementSpace`. The two main components for constructing this class are a `Mesh` and a `FiniteElementCollection` which provides sufficient information in order to determine global characteristics such as the total number of DOFs and the enumeration of all the global DOFs. In the `FiniteElementSpace` constructor, this enumeration is generated and stored as an object of class `Table` which represents the mapping: for any given element index i, return the ordered list of global DOF indices associated with element i. The order of these global DOFs in the list corresponds exactly to the local ordering of the local DOFs as described by the `FiniteElement`. The specific `FiniteElement` object associated with an element i can be obtained by first looking up the reference element type in the `Mesh` and then querying the `FiniteElementCollection` for the respective `FiniteElement` object. Thus, the `FiniteElementSpace` can produce the basis functions for any mesh element and the global indices of the respective local DOFs.

The global DOF numbering is created by first enumerating all DOFs associated with all vertices in the mesh; then enumerating all DOFs associated with all edges in the mesh — this is done, edge by edge, choosing a fixed direction on each edge and listing the DOFs on the edge following the chosen direction; next, the DOFs associated with faces are enumerated — this is done face by face,

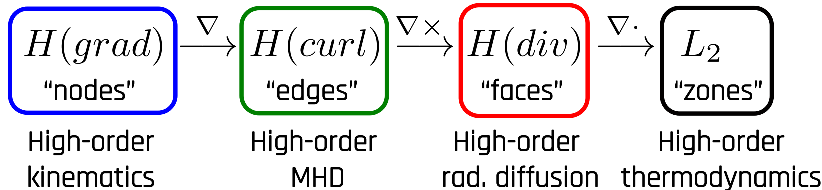

Figure 4: Continuous de Rham complex in 3D and example physical fields that can be represented in the respective spaces.

choosing a fixed orientation for each face and following it when listing the DOFs on the face; finally, all DOFs associated with the interiors of all mesh elements are enumerated, element by element.

An additional parameter in the construction of a `FiniteElementSpace` is its *vector dimension* which represents, mathematically, a Cartesian power (i.e. number of components) applied to the space defined by the `FiniteElement` basis functions. The additional optional parameter, `ordering`, of the `FiniteElementSpace` constructor, determines how the components are ordered globally: either `Ordering::byNODES` (default) or `Ordering::byVDIM`; the *vector* DOF (`vdof`) index k corresponding to the (scalar) DOF i in component j is given by $k = i + jN_d$ in the first case (N_d is the number of DOFs in one component), and $k = j + iN_c$, in the second (N_c is the number of components).

The *de Rham complex* [12, 91] is a compatible multi-physics discretization framework that naturally connects the solution spaces for many common PDEs. It is illustrated in Figure 4. The finite element method provides a compatible approach to preserve the de Rham complex properties on a fully discrete level. In MFEM, constructing a `FiniteElementSpace` using the `*_FECollection` with `*` replaced by `H1`, `ND`, `RT`, or `L2`, creates the compatible discrete finite element space for the continuous H^1, $H(\text{curl})$, $H(\text{div})$, or L^2 space, respectively. Note that the order of the space is simply a parameter in the constructor of the respective `*_FECollection`, see Figure 3.

The finite element spaces in the de Rham sequence are the natural discretization choices respectively for: kinematic variables (e.g., position, velocity), electromagnetic fields (e.g., electric field in magnetohydrodynamics (MHD)), diffusion fluxes (e.g., in flux-based radiation-diffusion) and thermodynamic quantities (e.g., internal energy, density, pressure). MFEM includes full support for the de Rham complex at arbitrary high order, on arbitrary order meshes, as illustrated for example in the first four example codes that come with the MFEM distribution, see Subsection 8.1.

Finite element functions are represented by the class `GridFunction`. A `GridFunction` is the list of DOFs for a discrete function in a particular `FiniteElementSpace`, so it could be used both on a linear algebra level (as a `Vector` object), or on the finite element level (as a piecewise-smooth function on the computational mesh). Grid functions are primal vectors, see Subsection 5.2, that are used to represent the finite element approximate solution. They contain methods for interpolation of continuous data (`ProjectCoefficient`), evaluation of integrals and errors (`ComputeL2Error`), as well as many linear algebra operations that are inherited from the `Vector` class. See file `fem/gridfunc.hpp` for more details.

4.3 High-Order Spaces

High-order methods are becoming increasingly important in computational science due to their potential for better simulation accuracy and favorable scaling on modern architectures [108, 41, 39, 26, 30]. Next-generation parallel computing architectures will favor simulation algorithms that exhibit high data parallelism and perform many floating-point operations (FLOPs) per memory

access (per byte). Traditional low-order discretization approaches, developed in an era when FLOPs were the performance-limiting factor, will no longer be an attractive option, since the cost of FLOPs will be considerably smaller than the cost of data motion.

High-order methods are a win-win proposition with respect to both mathematical accuracy and performance efficiency on current and future HPC architectures. It has long been known that high-order methods provide increased accuracy at low cost for problems with large regions of smooth, but non-trivial variation, such as vortices, boundary layers, and mixing regions. Our recent work on Arbitrary Lagrangian Eulerian (ALE) hydrodynamics shows that high-order finite elements on high-order meshes can also have mathematical benefits with respect to symmetry preservation, energy conservation, robustness in Lagrangian flow, and sub-zonal resolution for shock problems. These improvements come at the cost of additional floating-point operations, which map well to future computer architectures that will favor algorithms able to perform many FLOPs per memory access.

High-order finite elements use high-order polynomials in reference space for approximating physics fields and potentially the geometry of mesh elements, see Figure 3 and Figure 2, and require specialized algorithms for controlling algorithmic complexity not just with respect to problem size and parallelism, but also with respect to the polynomial order, see Subsection 4.3.

4.4 Visualization

MFEM provides integrations with several external tools for easy and accurate visualization of finite element meshes and grid functions, including arbitrary high-order meshes and fields. These integrations are based on sampling of the geometry and grid function data on a reference space lattice via the `GeometryRefiner` class in `fem/geom.hpp`. (One example of its use is the **Shaper** miniapp in `miniapps/meshing`.) MFEM can also provide accurate gradients enabling better surface normal vector computations.

Two of the visualization tools with which MFEM has been integrated are GLVis [55] and VisIt [32, 114]. GLVis is MFEM's lightweight *in-situ* visualization tool that directly uses MFEM classes for OpenGL visualization supporting interactive refinement of the reference-space sampling and uses accurate gradients for surface normals. VisIt is a comprehensive data analysis framework developed at LLNL, which includes native MFEM support via an embedded copy of the library. The sampled data in this case is controlled by a *multi-resolution* slider and is treated as low-order refined information (cf. Subsection 6.4) so all VisIt functionality can be used directly. Various file formats are supported, see Subsection 3.5, including in-memory remote visualization via socket connection in the case of GLVis.

5 Finite Element Operators

5.1 Discretization Methods

MFEM includes the abstractions and building blocks to *discretize* equations; that is, the process by which the linear system is formed from a PDE, choice of basis functions, and mesh. There are many different approaches for doing this in finite element methods, even for the same given PDE. One approach already outlined in Section 2 uses H^1 elements of any order and spatial dimension, where the basis functions are continuous across element interfaces, to discretize Poisson's equation. This is the most straightforward discretization of the equation, but there are many other approaches possible. For instance, Example 8 and Example 14 solve the same PDE with discontinuous Petrov-Galerkin (DPG) [17] and discontinuous Galerkin (DG) discretizations, respectively,

see Subsection 8.1. The examples include interactive documentation organized by the different discretization methods available in the library and are the fastest route to learn about MFEM's capabilities. A comprehensive list of the bilinear form integrators that come pre-packaged with MFEM is included in Appendix A.

Examples 3-5 show a wide range of the discretization capability of MFEM and many of the possible finite elements. Example 3 solves the second-order definite Maxwell equation using the $H(\text{curl})$ Nedelec finite elements with the curl-curl and mass bilinear form integrators. Example 4 progresses down the de Rham sequence, and solves a second order definite equation with a Neumann boundary condition using $H(\text{div})$ Raviart-Thomas finite elements and div-div and mass bilinear form integrators. Example 5 uses a mixed $H(\text{div})$ and L^2 (DG) discretization of a Darcy problem, solving these together in a 2×2 block bilinear form. These three examples are just a few of the many examples included with the library, but they show a wide range of the finite elements and discretization approaches possible along the de Rham sequence.

On the meshing side, there are also many different approaches. As described above in Section 3, MFEM supports arbitrary-order H^1 and L^2 meshes, which can be topologically periodic or assigned boundary tags. However, MFEM also includes an extension to its `Mesh` class to generate basis functions from non-uniform Rational B-splines (NURBS), see Subsection 3.3 and the `NURBSExtension` class in the `mesh/nurbs.hpp` file. This allows for isogeometric analysis, where the basis is refined without changing the geometry or its parametrization [36].

MFEM includes various ordinary differential equation (ODE) solvers that can be used in conjunction with the finite elements and bilinear forms to discretize the time derivative terms. These work by writing code for a `TimeDependentOperator` that defines the map $(\mathbf{x}, t) \to f(\mathbf{x}, t)$ that the ODE solvers, both implicit and explicit, use to integrate the approximate solution in time, see Appendix B. Many ODE solvers are distributed with the library: various implicit and explicit Runge-Kutta (RK) methods including singly-diagonal implicit versions (SDIRK), and symplectic methods. Additionally, MFEM supports time integration with the SUNDIALS (SUite of Nonlinear and DIfferential/ALgebraic equation Solver) library, which provides many additional ODE solvers. Finally, MFEM's ODE solvers can be extended by inheriting from the base abstract class `ODESolver`.

5.2 Finite Element Linear Systems

At the heart of any finite element application is a finite element linear system which, as discussed in Section 2, can be written in the canonical form

$$Ax = b$$

where the matrix A is square and non-singular. Such systems are well-suited for either iterative or direct linear solvers but creating this linear system is non-trivial. MFEM's primary task is to build this linear system and interpret the resulting solution vector x. When using first order nodal basis functions on a conforming mesh existing on only one processor this is a relatively straight forward task. However, supporting distributed memory architectures, high-order basis functions, non-conforming meshes, or more general basis function types each introduce complications that require more care. To manage these complications MFEM makes use of abstractions which clearly separate finite element concepts from linear algebra concepts.

MFEM's linear algebra objects include `Vector` and `SparseMatrix` in serial and `HypreParVector` and `HypreParMatrix` in parallel, as well as various linear solvers derived from the `Solver` class. The finite element objects include `(Par)GridFunction`, `(Par)LinearForm`, and `(Par)BilinearForm`. For convenience `(Par)GridFunction` and `(Par)LinearForm` inherit from the `Vector` class and can

therefore be used as vectors, and similarly `(Par)BilinearForm` can be used as a matrix, but these convenience features should be used with care.

The `ParGridFunction` object contains, among other things, all of the degrees of freedom needed to interpolate field values within every element contained in the local portion of the computational mesh, see the lowercase `x` in Figure 5. The uppercase `X` in Figure 5 is a linear algebra `Vector` related to this `ParGridFunction` but potentially quite different. The uppercase `X` contains only the *true* degrees of freedom from `x`. For example, some of the degrees of freedom in the `ParGridFunction` may be subject to constraints if they happen to be shared with neighboring elements in a non-conforming portion of the mesh, or they may be constrained to match degrees of freedom owned by elements found on another processor. Some of the degrees of freedom in the `ParGridFunction` may not even directly contribute to the linear system if static condensation or hybridization is being used. For all of these reasons, the linear algebra `Vector` stored in `X` might be much smaller than the `ParGridFunction` stored in `x`. The `P` and `R` operators shown in Figure 5, called the *prolongation* and *restriction* operators, respectively, are created and managed by the `ParFiniteElementSpace` and can be used to map data between the finite element representation of a field and its linear algebra representation.

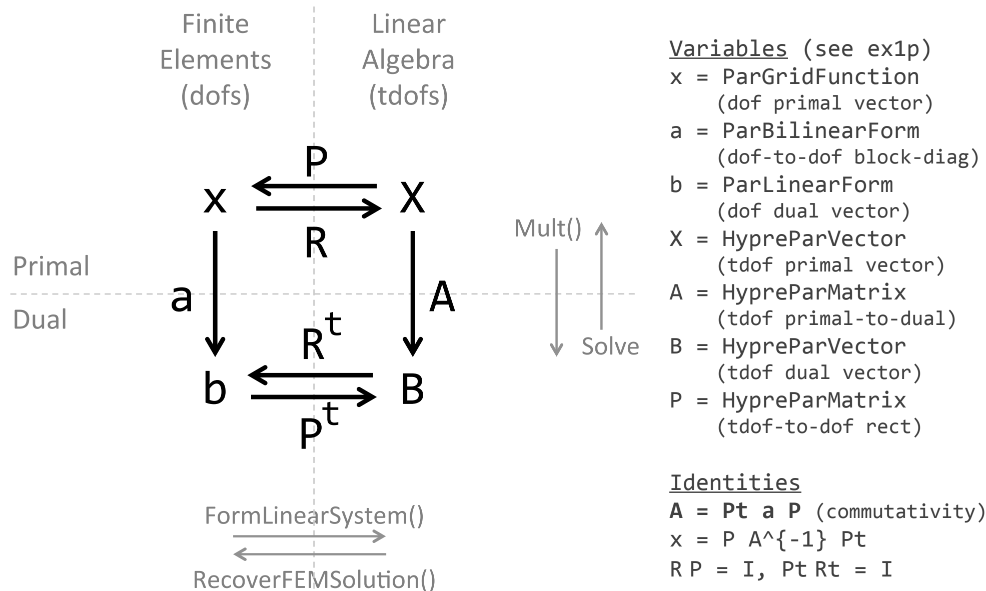

Figure 5: Graphical depiction of the relationship between the finite element bilinear and linear form objects, and the linear algebra matrices and primal/dual vectors in MFEM.

The `ParGridFunction`, labeled `x` in Figure 5, and its linear algebra counterpart `X`, are called *primal* vectors because of their direct relationship with the finite element expansion of a field. Indeed the values stored in `x` are the expansion coefficients f_i needed in Equation 18.

$$f(\vec{x}) = \sum_i f_i \varphi_i(\vec{x}). \tag{18}$$

Conversely, a `ParLinearForm`, labeled `b` in Figure 5, or the vector labeled `B` are *dual* vectors. In this context duality refers to the fact that *dual* vectors map *primal* vectors to the set of real numbers. More importantly, they can be used to map a `ParGridFunction` to a physical quantity of interest. For example, if we have a `ParGridFunction` ρ representing the mass density of a fluid, and

a `ParLinearForm` v such that $v_i = \int_\Omega \varphi_i$, i.e. a `ParLinearForm` representing the constant function 1, then $v \circ \rho$ would approximate the integral of the density over the computational domain which would equal the total mass of the fluid in this illustration. Dual vectors will be of the same length as their primal counterparts but their entries have very different meanings. The relationship between `b` and `B` is complimentary to that between `x` and `X`. Whereas the restriction operator removes dependent entries from `x` to produce the shorter vector `X`, the transpose of the prolongation operator is used to coalesce entries from `b` to form those of `B`. For example $\texttt{P}^t$ will add together entries from `b` to sum the contributions from different elements to the basis function integral over its entire support which will be stored in `B`.

Dual vectors can be created directly by integrating a function times the appropriate basis functions as occurs inside a `(Par)LinearForm` or indirectly by applying a `(Par)BilinearForm` or a system matrix to a primal vector. The resulting dual vector should be identical in either case. Which scheme is used to create a particular dual vector is usually determined by how the source terms in the PDE arise. If the sources are determined by known functions it is generally most efficient to provide these functions to a `(Par)LinearForm` object and compute the dual vector directly. If, on the other hand, the source term is the result of a field represented by a `(Par)GridFunction` it could be more efficient to simply apply a `(Par)BilinearForm` to the appropriate primal vector.

As implied in Figure 5, the linear algebra operator `A` can be computed from the `(Par)BilinearForm` `a` as $A = P^t a P$; however, many finite element linear systems require boundary conditions to ensure that they are non-singular. To facilitate the application of boundary conditions the `(Par)BilinearForm` class has a `FormLinearSystem` method which prepares the three linear algebra objects, as well as applying boundary conditions. In the simplest case this method performs the following operations:

$$\begin{aligned} A &= P^t a P, \\ X &= R x, \\ B &= P^t b. \end{aligned}$$

If a set of essential degrees of freedom have also been provided the system matrix and right-hand side vector `B` are further modified as follows:

$$\begin{aligned} A' &= (I - D)A(I - D) + D, \\ B' &= (I - D)(B - ADX) + DX. \end{aligned}$$

Here I is the identity matrix and D is a diagonal matrix with ones on the diagonals corresponding to essential degrees of freedom and zeros otherwise. The first equation eliminates the rows and columns corresponding to essential degrees of freedom and places a one at the corresponding diagonal entry. The second equation strategically places the desired boundary values from `X`, scaled by `A` when necessary, in the appropriate entries in `B`.

The `FormLinearSystem` method also supports two more advanced and closely related techniques for reducing the size of a finite element linear system: *Hybridization* and *Static Condensation*. These approaches are briefly reviewed below, see [48] for more details.

Hybridization can be viewed as a technique for solving linear systems obtained through finite element assembly. The assembled matrix A can be written as:

$$A = P^T \hat{A} P,$$

where P is the matrix mapping the conforming finite element space to the purely local finite element space without any inter-element constraints imposed, and $\hat{A}$ is the block-diagonal matrix of all element matrices. We assume that:

- $\hat{A}$ is invertible,
- P has a left inverse R, such that $RP = I$,
- a constraint matrix C can be constructed, such that $\mathrm{Ker}(C) = \mathrm{Im}(P)$.

Under these conditions, the linear system $AX = B$ can be solved using the following procedure:

- Solve for λ in the linear system:

$$(C\hat{A}^{-1}C^T)\lambda = C\hat{A}^{-1}R^T B.$$

- Compute

$$X = R\hat{A}^{-1}(R^T B - C^T\lambda). \tag{19}$$

Hybridization is advantageous when the matrix $H = (C\hat{A}^{-1}C^T)$ of the hybridized system is either smaller than the original system, or is simpler to invert with a known method.

In some cases, e.g. high-order elements, the matrix C can be written as

$$C = \begin{pmatrix} 0 & C_b \end{pmatrix},$$

and then the hybridized matrix H can be assembled using the identity

$$H = C_b S_b^{-1} C_b^T,$$

where S_b is the Schur complement of $\hat{A}$ with respect to the same decomposition as the columns of C:

$$S_b = \hat{A}_b - \hat{A}_{bf}\hat{A}_f^{-1}\hat{A}_{fb}.$$

Hybridization can also be viewed as a discretization method for imposing (weak) continuity constraints between neighboring elements.

Static condensation is a technique for solving linear systems by eliminating groups/blocks of unknowns and reducing the original system to the remaining interfacial unknowns. The assumption is that unknowns in one group are connected (in the graph of the matrix) only to unknowns in the same group or to interfacial unknowns but not to other groups.

For finite element systems, the groups correspond to degrees of freedom associated with the interior of the elements. The rest of the DOFs (associated with the element boundaries) are interfacial.

In block form the matrix of the system can be written as

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{array}{l} \text{– groups: element interior/private DOFs} \\ \text{– interface: element boundary/exposed DOFs} \end{array}$$

where the block A_{11} is itself block diagonal with small local blocks and it is, therefore, easily invertible.

Starting with the block system

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} = \begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$$

the reduced, statically condensed system is given by

$$S_{22}X_2 = B_2 - A_{21}A_{11}^{-1}B_1,$$

where the Schur complement matrix S_{22} is given by

$$S_{22} = A_{22} - A_{21}A_{11}^{-1}A_{12}.$$

After solving the Schur complement system, the X_1 part of the solution can be recovered using the formula

$$X_1 = A_{11}^{-1}(B_1 - A_{12}X_2). \tag{20}$$

Static condensation can be beneficial when using sufficiently high-order basis functions, i.e. when the number of internal degrees of freedom outweighs the number of degrees of freedom on element interfaces.

These more advanced techniques, which compute only portions of the solution vector, necessitate a further step of reconstructing the entire solution vector. The `(Par)BilinearForm` class provides a `RecoverFEMSolution` method for exactly this purpose. Given the partial solution vector `X` and the `(Par)LinearForm` `b` this method computes the full degree of freedom vector `x` needed to properly represent the solution field throughout the mesh. Additional details can be found in [48].

5.3 Operator Decomposition

Finite element operators are typically defined through weak formulations of partial differential equations that involve integration over a computational mesh. The required integrals are computed by splitting them as a sum over the mesh elements, mapping each element to a simple reference element (e.g. the unit square) and applying a quadrature rule in reference space, see Section 2.

This sequence of operations highlights an inherent hierarchical structure present in all finite element operators where the evaluation starts on *global (trial) degrees of freedom* on the whole mesh, restricts to *degrees of freedom on subdomains* (groups of elements), then moves to independent *degrees of freedom on each element*, transitions to independent *quadrature points* in reference space, performs the integration, and then goes back in reverse order to global (test) degrees of freedom on the whole mesh.

This is illustrated in Figure 6 for the simple case of a symmetric linear operator on third order (Q_3) scalar continuous (H^1) elements, where we use the notions **T-vector**, **L-vector**, **E-vector** and **Q-vector** to represent the sets corresponding to the (true) degrees of freedom on the global mesh, the split local degrees of freedom on the subdomains, the split degrees of freedom on the mesh elements, and the values at quadrature points, respectively.

One of the challenges with high-order methods is that a global sparse matrix is no longer a good representation of a high-order linear operator, both with respect to the FLOPs needed for its evaluation, as well as the memory transfer needed for a matvec. Thus, high-order methods require a new "format" that still represents a linear (or more generally, nonlinear) operator, but not through a sparse matrix.

We refer to the operators that connect the different types of vectors as:

- Subdomain restriction P.
- Element restriction G.
- Basis (Dofs-to-Qpts) evaluator B.
- Operator at quadrature points D.

More generally, when the test and trial space differ, they get their own versions of P, G and B.

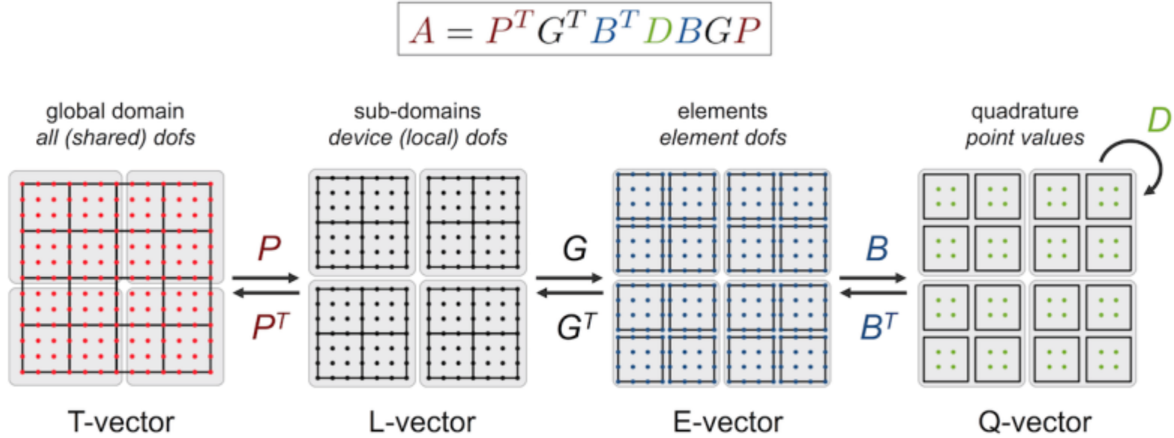

Figure 6: Fundamental finite element operator decomposition. This algebraically factored form is a much better description than a global sparse matrix for high-order methods and is easy to incorporate in a wide variety of applications. See also the libCEED library in [30].

Note that in the case of adaptive mesh refinement (AMR), the restrictions P and G will involve not just extracting sub-vectors, but evaluating values at constrained degrees of freedom through the AMR interpolation, see Subsection 7.2. There can also be several levels of subdomains (P_1, P_2, etc.), and it may be convenient to split D as the product of several operators (D_1, D_2, etc.).

After the application of each of the first three transition operators, P, G and B, the operator evaluation is decoupled on their ranges, so P, G and B allow us to "zoom-in" to subdomain, element, and quadrature point level, ignoring the coupling at higher levels. Thus, a natural mapping of A on a parallel computer is to split the **T-vector** over MPI ranks in a non-overlapping decomposition, as is typically used for sparse matrices, and then split the rest of the vector types over computational devices (CPUs, GPUs, etc.) as indicated by the shaded regions in Figure 6. This is discussed further in Subsection 6.1.

One of the advantages of the decomposition perspective in these settings is that the operators P, G, B and D clearly separate the MPI parallelism in the operator (P) from the unstructured mesh topology (G), the choice of the finite element space/basis (B) and the geometry and point-wise physics D. These components also naturally fall in different classes of numerical algorithms – parallel (multi-device) linear algebra for P, sparse (on-device) linear algebra for G, dense/structured linear algebra (tensor contractions) for B and parallel point-wise evaluations for D.

Since the global operator A is just a series of variational restrictions with B, G and P, starting from its point-wise kernel D, a "matvec" with A can be performed by evaluating and storing some of the innermost variational restriction matrices, and applying the rest of the operators "on-the-fly". For example, one can compute and store a global matrix on **T-vector** level. Alternatively, one can compute and store only the subdomain (**L-vector**) or element (**E-vector**) matrices and perform the action of A using matvecs with P or P and G. While these options are natural for low-order discretizations, they are not a good fit for high-order methods due to the amount of FLOPs needed for their evaluation, as well as the memory transfer needed for a matvec.

Much higher performance can be achieved instead by the use of *partial assembly* algorithms, see Subsection 5.4, where we compute and store only D (or portions of it) and evaluate the actions of P, G and B on-the-fly. Critically for performance, we take advantage of the tensor-product structure of the degrees of freedom and quadrature points on quad and hex elements to perform the action of B without storing it as a matrix. Implemented properly, the partial assembly algorithm requires

the optimal amount of memory transfers (with respect to the polynomial order) and near-optimal FLOPs for operator evaluation. It consists of an operator *setup* phase, that evaluates and stores D and an operator *apply* (evaluation) phase that computes the action of A on an input vector. When desired, the setup phase may be done as a side-effect of evaluating a different operator, such as a nonlinear residual. The relative costs of the setup and apply phases are different depending on the physics being expressed and the representation of D.

5.4 High-Order Partial Assembly

In the traditional finite element setting, the operator is assembled in the form of a matrix, whose action is the operator evaluation. At high orders this requires both a large amount of memory to store the matrix, as well as many floating point operations to apply it. By exploiting the structure shown in Section 5 as well as the basis functions structure, there are options for creating operators that require much less storage space and scale better at high orders. This section introduce partial assembly and sum factorization [94, 41], which reduce both the assembly storage and number of floating point operations required to apply the operator, and discusses general algorithm opportunities and challenges in the MFEM code.

Removing the finite element space restriction operator from the assembly for domain-based operators yields the element-local matrices at the **E-vector** level. This storage can lead to faster data access, since the block is stored contiguously in memory, and applications of the block can be designed to maximally use the cache.

Partial assembly operates at the **Q-vector** level, after additionally removing the basis functions and gradients from the assembled operator. This leaves on the D operator to store for every element, see Subsection 5.3. This by itself reduces the storage but not the number of floating point operations required for evaluation. As will be discussed later, this is key to offloading the operator action to a co-processor that may have less memory.

As an illustration of partial assembly, the decomposition of the mass matrix evaluated on a single element E

$$(M_E)_{ij} = \int_E \rho\, \varphi_j \varphi_i \, dx \tag{21}$$

when mapped to a reference element and integrated with a quadrature rule of points $\{\hat{x}\}$ and weights $\{\alpha_k\}$ yields

$$(M_E)_{ij} = \sum_k \alpha_k \left(\rho \circ \Phi\right)(\hat{x}_k)\, \hat{\varphi}_j(\hat{x}_k)\hat{\varphi}_i(\hat{x}_k)\, |J(\hat{x}_k)|. \tag{22}$$

Defining the tensor of basis functions at quadrature points as $B_{ki} = \hat{\varphi}_i(\hat{x}_k)$, the above equation can be rewritten as

$$(M_E)_{ij} = \sum_k B_{ki}(D_E)_{kk}B_{kj}, \qquad (D_E)_{kk} = \alpha_k |J(\hat{x}_k)| \left(\rho \circ \Phi\right)(\hat{x}_k). \tag{23}$$

Using this definition, the matrix operator can be written simply as $M_E = B^T D_E B$. Matrix-vector evaluations are computed as the series of products by B, D_E, and B^T without explicitly forming M_E.

Applying B requires the same number of floating point operations as applying the fully-assembled M_E matrix ($\mathcal{O}(p^{2d})$), but as described in Subsection 5.3 when taking advantage of the tensor-product structure of the basis functions and quadrature points on quad and hex elements, B_{ki} can be written as

$$B_{ki} = \hat{\varphi}_{i_1}(\hat{x}_{k_1}) \ldots \hat{\varphi}_{i_d}(\hat{x}_{k_d}) \tag{24}$$

Method	Storage	Assembly	Evaluation
Traditional full assembly + Matvec	$\mathcal{O}(p^{2d})$	$\mathcal{O}(p^{3d})$	$\mathcal{O}(p^{2d})$
Sum factorization full assembly + Matvec	$\mathcal{O}(p^{2d})$	$\mathcal{O}(p^{2d+1})$	$\mathcal{O}(p^{2d})$
Partial assembly + Matrix-free action	$\mathcal{O}(p^{d})$	$\mathcal{O}(p^{d})$	$\mathcal{O}(p^{d+1})$

Table 1: Comparison of storage and Assembly/Evaluation FLOPs required for full and partial assembly algorithms on tensor-product element meshes (quads and hexes). Here, p represents the polynomial order of the basis functions and d represents the number of spatial dimensions. The number of DOFs on each element is $\mathcal{O}(p^d)$ so the "sum factorization full assembly" and "partial assembly" algorithms are nearly optimal.

with d the number of dimensions. In this case the tensor B itself can be decomposed as a series of smaller one-dimensional matrices, for instance represented for the first dimension as $\tilde{B}_{i_1 k_1}$. The one-dimensional index sets $\{i_1 \dots i_d\}$ and $\{k_1 \dots k_d\}$ form a multi-index to represent a single value of the multi-dimensional index i and k, respectively. The equivalent expression using this decomposition is

$$B_{ki} = \tilde{B}_{k_1 i_1} \dots \tilde{B}_{k_d i_d}. \tag{25}$$

Applying the series of $\tilde{B}$ tensors reduces the overall number of floating point operations when applying M_E to $\mathcal{O}(p^{d+1})$. This formulation is often referred to as *sum factorization*.

To make this point concrete, consider the application of a quad basis to a vector v for interpolation at a tensor product of quadrature points. Without taking advantage of the structure of the basis, the product takes the form

$$B_{ki} v_i = \varphi_{i_1}(\hat{x}_1)\varphi_{i_2}(\hat{x}_2) v_i, \tag{26}$$

which requires $\mathcal{O}(p^{2d})$ $(d = 2)$ storage and operations for the tensor-vector product. When using the alternative form (25) the operation can be rewritten as

$$\tilde{B}_{k_1 i_1} \tilde{B}_{k_2 i_2} \tilde{v}_{i_1 i_2} = \tilde{B} \tilde{v} \tilde{B}^T. \tag{27}$$

This highlights an interesting aspect of sum factorization: with each smaller matrix product with $\tilde{B}$, an additional axis is converted from basis (i) to quadrature (k) indices. The reasoning applies to three spatial dimensions as well, but additional reshaping is required. Using the sum factorization approach, the storage was reduced to $\mathcal{O}(p^2)$ and the number of operations to $\mathcal{O}(p^3)$.

Choosing to store the partially assembled operator instead of the full matrix affects the solvers that can be used, since the full matrix is not available to be queried. This means for instance that traditional multigrid solvers are difficult to apply. However, the operator does have structure which can be exploited to expose the diagonal of the matrix, making it possible to use diagonal preconditioners. This is discussed further in Subsection 6.4.

The storage and asymptotic number of floating point operations per second (FLOP/s) required for assembly and evaluation using the different methods are recorded in Table 1. Sum factorization reduces the cost of assembling the operator, even the cost of full assembly (**T-vector** level). Furthermore, partial assembly has improved asymptotic scaling for high orders in both storage and number of floating point operations for assembly and evaluation. Therefore, partial assembly is ideal for high orders.

There are many opportunities and challenges for parallelization with partial assembly using sum factorization. At the **E-vector** level the products can be applied independently for every element in

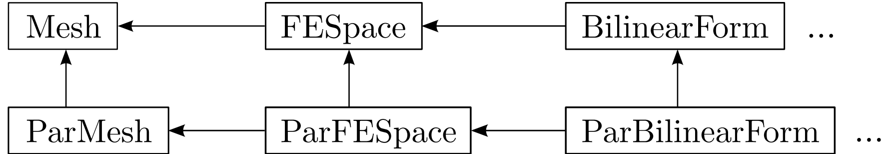

Figure 7: Parallel classes inherit from, and partially override, serial classes.

parallel, which makes partial assembly with sum factorization an ideal portion of the finite element algorithm to offload to co-processors, such as GPUs. However, since these are small, algorithms will assuredly remain bandwidth bound if looping over the elements on the inner-most loop. In order to increase efficiency, the series of matrix contractions needs to be fused into a specialized kernel.

In MFEM these algorithms are in the bilinear and nonlinear form integrators themselves. In the construction phase of the integrator the partially assembled operator is usually constructed, and the `Mult` method implements the matrix-free evaluation using the operator.

6 High-Performance Computing

6.1 Parallel Meshes, Spaces, and Operators

The MFEM design handles large scale parallelism by utilizing the Message Passing Interface (MPI) library in additional layer that reuses as much of the serial code as possible. In terms of object-oriented design, this is done by sub-classing the serial classes to augment them with parallel logic, see Figure 7, occasionally overriding small parts of the code using virtual functions.

If K is the number of MPI tasks, we decompose the problem domain into K parts and try to process the parts locally as much as possible, see Figure 8. The parallel mesh object, `ParMesh`, is just a regular serial `Mesh` on each MPI task plus additional information that describes the geometric entities (faces, edges, vertices) that are shared with other processors. See Subsection 3.4 and the file `mesh/pmesh.hpp` for more details. The parallel finite element space, `ParFiniteElementSpace` is just a regular serial `FiniteElementSpace` on each task plus a description of the shared degrees of freedom, grouped in *communication groups*, see `fem/pfespace.hpp`. As in the serial case, one of the main responsibilities of the parallel finite element space is to provide, via `GetProlongationMatrix()`, the prolongation matrix P, see Subsection 5.3, which is used for parallel assembly (see below) or adaptive mesh refinement, see Section 7. Parallel grid functions, `ParGridFunction`, are just regular `GridFunction` objects on **L-vector** level which can be mapped back and forth to **T-vectors**, e.g. with the `ParallelAverage` and `Distribute` methods. See `fem/pgridfunc.hpp` for details.

The global finite element stiffness matrix A has K diagonal blocks and can be assembled without any parallel communication. The prolongation matrix P is parallel and its construction requires communication, however part of that communication can be overlapped with the computation of A. As a general rule, we try to keep MPI messages to a minimum and only communicate with immediate neighbors in the parallel mesh, ideally overlapping communication with computation using asynchronous MPI calls.

Based on the variational restriction perspective presented in Subsection 5.3, the final parallel assembly can be computed with a parallel P^TAP triple matrix product, which in MFEM is based on the *hypre* library [67]. We make use of the highly optimized `RAP` triple-product kernel which *hypre* provides internally for the coarse-grid operator construction in its algebraic multigrid solvers.

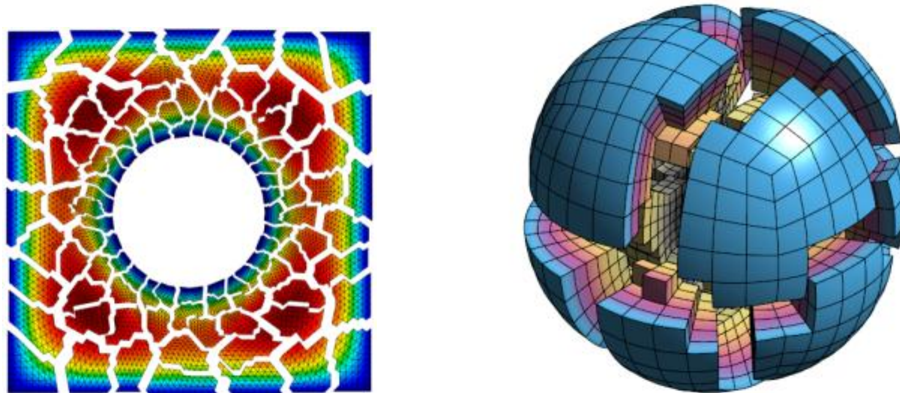

Figure 8: Left: Parallel Example 1 (examples/ex1p.cpp) on 100 processors with a relatively coarse version of data/square-disc.mesh. Right: Unstructured parallel decomposition of a fourth order NURBS mesh of the unit ball obtained in the solution of Example 1 on 16 processors.

One of the advantages of handling parallelism by sub-classing the serial finite element classes is that serial MFEM-based application codes are easily converted to highly-scalable parallel versions by simply adding the Par prefix to the types of finite element variables. To emphasize this point, the MFEM distribution includes serial and parallel versions of most of its example codes, so the changes needed to transition between the two are easy to compare.

Here are, for example, the main changes between examples/ex1.cpp and examples/ex1p.cpp.

```
94    ParMesh *pmesh = new ParMesh(MPI_COMM_WORLD, *mesh);
95    {
96       int par_ref_levels = 2;
97       for (int l = 0; l < par_ref_levels; l++)
98       {
99          pmesh->UniformRefinement();
100      }
101   }
102   ParFiniteElementSpace *fespace = new ParFiniteElementSpace(pmesh, fec);
103   ParLinearForm *b = new ParLinearForm(fespace);
104   ParGridFunction x(fespace);
105   ParBilinearForm *a = new ParBilinearForm(fespace);
```

The results can be visualized with GLVis in parallel (each MPI tasks sends its data independently) where different regions can be identified/split to emphasize the parallelism in the computation, see Figure 8 and Subsection 4.4.

6.2 Scalable Linear Solvers

Parallel matrices in MFEM are computed and stored directly in the ParCSR format of the *hypre* library, which gives the user direct access to high-performance parallel linear algebra algorithms. For example, MFEM uses *hypre*'s matvec routines, as well as the RAP function, see Subsection 6.1, which has been optimized in *hypre* for the construction of coarse-grid operators in a multigrid hierarchy.

This tight integration with *hypre* enables MFEM applications to easily access the powerful algebraic multigrid (AMG) preconditioner in the library, which has demonstrated scalability to millions of parallel tasks. All parallel MFEM examples are using these scalable preconditioners, which only

take a line of code in MFEM. For example the parallel linear system in `examples/ex1p.cpp` is defined by

```
183    HypreParMatrix A;
184    Vector B, X;
185    a->FormLinearSystem(ess_tdof_list, x, *b, A, X, B);
```

hypre's BoomerAMG preconditioner can then be used with Conjugate Gradient to solve it simply with

```
194    HypreSolver *amg = new HypreBoomerAMG(A);
195    HyprePCG *pcg = new HyprePCG(A);
196    pcg->SetTol(1e-12);
197    pcg->SetMaxIter(200);
198    pcg->SetPrintLevel(2);
199    pcg->SetPreconditioner(*amg);
200    pcg->Mult(B, X);
```

In addition to general black-box solvers, such as BoomerAMG, the MFEM interface enables access to *discretization-enhanced* AMG methods such as the auxiliary-space Maxwell solver (AMS) [74] which is specifically designed for second-order definite Maxwell problems discretized with Nedelec $H(\text{curl})$-conforming elements, see Subsection 4.2. The AMS algorithm needs the discrete gradient operator between the nodal H^1-type and the Nedelec spaces, which in MFEM is represented as a `DiscreteLinearOperator` corresponding to an embedding between spaces. This operator is constructed in general parallel settings (including on surfaces and mesh skeletons) with the following code from `linalg/hypre.cpp`.

```
2815    ParDiscreteLinearOperator *grad;
2816    grad = new ParDiscreteLinearOperator(vert_fespace, edge_fespace);
2817    if (trace_space)
2818    {
2819       grad->AddTraceFaceInterpolator(new GradientInterpolator);
2820    }
2821    else
2822    {
2823       grad->AddDomainInterpolator(new GradientInterpolator);
2824    }
2825    grad->Assemble();
2826    grad->Finalize();
2827    G = grad->ParallelAssemble();
```

From the user perspective, this is handled automatically given a `FiniteElementSpace` object, and the use of AMS is also a one-liner in MFEM. This is illustrated in the following excerpt from `examples/ex3p.cpp`, which also shows how static condensation is seamlessly handled by the preconditioner.

```
195    ParFiniteElementSpace *prec_fespace =
196       (a->StaticCondensationIsEnabled() ? a->SCParFESpace() : fespace);
197    HypreSolver *ams = new HypreAMS(A, prec_fespace);
198    HyprePCG *pcg = new HyprePCG(A);
199    pcg->SetTol(1e-12);
200    pcg->SetMaxIter(500);
201    pcg->SetPrintLevel(2);
202    pcg->SetPreconditioner(*ams);
203    pcg->Mult(B, X);
```

Finally, different preconditioning options are also easy to combine as illustrated in Example 4p which solves an $H(\text{div})$ problem discretized with Raviart-Thomas finite elements. Depending on the

dimension, and the use of hybridization or static condensation, see Subsection 5.2, several different preconditioning options could be appropriate. All of them can be handled with the following simple code segment.

```
221    if (hybridization) { prec = new HypreBoomerAMG(A); }
222    else
223    {
224       ParFiniteElementSpace *prec_fespace =
225          (a->StaticCondensationIsEnabled() ? a->SCParFESpace() : fespace);
226       if (dim == 2)   { prec = new HypreAMS(A, prec_fespace); }
227       else            { prec = new HypreADS(A, prec_fespace); }
228    }
229    pcg->SetPreconditioner(*prec);
```

6.3 Vectorization and GPU Acceleration

A high-performance implementation of the partial assembly algorithms targeting high-order finite element discretizations (see Subsection 5.4) is available in MFEM via the HPC versions of Example 1 under `miniapps/performance`. These codes use a set of templated classes to efficiently implement the inner-most portion $B^T DB$ of the fundamental finite element operator decomposition (see Subsection 5.3). The C++ template approach allows the element evaluation kernel to be fully inlined, facilitating fixed loop bounds and compiler vectorization without intrinsics. These classes are fully documented at `http://mfem.org/performance/`; we summarize some of the major classes below.

Fixed-size container classes `TVector`, `TMatrix`, `TTensor3` and `TTensor4` for 1D/2D/3D/4D tensors stored in column-major layout are defined in `linalg/ttensor.hpp`. The following tensor contraction/product operations are supported:

- `Mult_1_2` : $C_{i,j,k} = \sum_s A_{s,j} B_{i,s,k}$,
- `Mult_2_1` : $C_{i,j,k} = \sum_s A_{i,s} B_{s,j,k}$,
- `TensorAssemble` : $C_{i,k,j,l} = \sum_s A_{s,i} A_{s,j} B_{k,s,l}$ and $D_{i,k,j,l} = \sum_s A_{i,s} B_{s,j} C_{k,s,l}$,
- `TensorProduct` : $C_{i,j,k,l} = A_{i,j,k} B_{j,l}$.

Several kernel classes (e.g. mass, diffusion) that represent the matrix D from the above finite element operator decomposition can be found in `fem/tbilininteg.hpp`. These classes also specify the type of the local operator that needs to be applied before and after the D matrix - these are the B_{in} and B_{out}^T matrices, respectively. The product $B_{out}^T D B_{in}$ is the local element matrix, which is the result when using the `BilinearFormIntegrator` classes. The kernel classes provide the following methods:

- `Action`: Evaluate the action of D without explicitly storing the partially assembled data; this is needed for matrix-free action.
- `Assemble`: Evaluate the partially assembled data, D, which is kernel-specific: e.g., for mass, the data is one scalar per quadrature point; for diffusion, the data is one $d \times d$ matrix (in d-dimensions) per quadrature point.
- `MultAssembled`: Perform the action of D using the pre-computed partially assembled data.

The overall bilinear form operator, templated on the mesh, finite element space, integration rule and bilinear form integrator can be found in `fem/tbilinearform.hpp`. The following assembly and evaluation schemes are supported:

- `MultUnassembled`: Matrix-free action using the mesh nodes and the input vector.
- `Assemble`, `MultAssembled`: Partial assembly and operator action using the partially assembled data at quadrature points and the input vector.
- `AssembleMatrix(DenseTensor &)`: Assemble the local element matrices and store them as a `DenseTensor`.
- `AssembleMatrix(SparseMatrix &)`: Assemble the operator in a global (CSR) `SparseMatrix`.
- `AssembleBilinearForm(BilinearForm &)`: Assemble element matrices and add them to the bilinear form.

The MFEM project is also actively engaged with exploring GPU acceleration for partial assembly high-order algorithms. A lot of these efforts take place in the CEED co-design center [30], where for example a GPU accelerated version of the library has been developed based on OCCA [92]. Additional efforts in this area include the AcroTensor library [1], collaboration with the MAGMA team at UTK, the development of CUDA kernels using warp shuffle intrinsics and the use of the RAJA performance portability layer [66]. The MFEM team is working on each of these approaches, and we expect to have a unifying GPU interface in the next major release of the library.

6.4 Matrix-Free Preconditioning

With high-order methods the explicit assembly of finite element matrices becomes a bottleneck, as discussed in Subsection 5.4. While matrix-free (partially assembled) high-order operators offer a lot of benefits, one of their drawbacks is that the entries of the matrix are not easily available, so we cannot use algebraic preconditioners for such operators.

Most work on linear solvers and preconditioning focuses on robustness and scalability with respect to the mesh spacing h, insuring that iteration counts, algorithmic complexity, and solve times behave nicely as h goes to 0. Here we are concerned instead with how solvers behave as the polynomial order p increases. As one example of the importance of this perspective, naive finite element assembly of a system matrix has a complexity of $\mathcal{O}(p^{3d})$ arithmetic operations, where d is the spatial dimension. As shown in Subsection 5.4, for tensor-product finite elements, this same operator can be applied in $\mathcal{O}(p^{d+1})$ operations. This consideration alone argues against assembling high-order finite element operators explicitly.

Multigrid solvers for high-order discretizations are a fairly well-studied area [90, 93, 110]. However, these works rely on traditional matrix assembly, which is not practical for very high p. Matrix-free multigrid solvers are also fairly common in a geometric multigrid context—see [85] for a recent example, but the application of algebraic multigrid techniques to matrix-free implementations of high-order discretizations has, to our knowledge, not been attempted.

The basic idea of the low-order refined (LOR) technique is to put a first order mesh on the nodal points of a high-order mesh, as in Figure 9. For a nodal discretization this induces a very close relation between the high-order and the low-order spaces, where, in particular, the identity matrix can be thought of as the interpolation between them. As a result, a finite-element matrix assembled on the low-order mesh has the same size as the corresponding high-order operator but is much sparser, and so can be explicitly assembled and stored for preconditioning.

The templated HPC Example 1p, discussed in the previous section, includes a LOR preconditioning option where we set up *hypre*'s BoomerAMG based on the LOR sparse matrix, and then use this AMG to precondition CG with the "matrix-free" tensor-based application of the high-order operator.

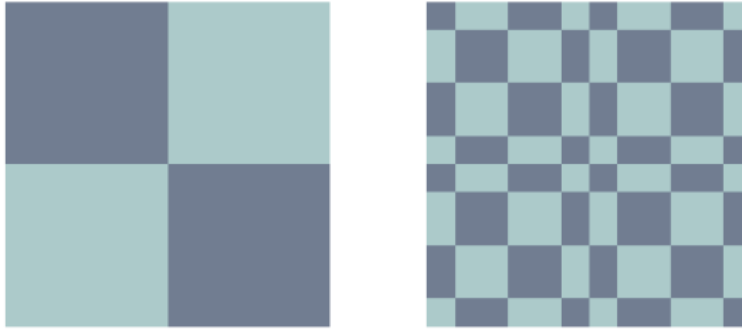

Figure 9: A four-element high-order mesh on the left and its corresponding low-order refined mesh on the right. The high-order mesh is fourth order, and the low-order refined mesh has a vertex for each nodal point of the high-order mesh.

7 Finite Element Adaptivity

MFEM includes extensive support for serial and parallel finite element adaptivity on general high-order unstructured meshes, including: local conforming mesh refinement on triangular and tetrahedral meshes (conforming h-adaptivity), non-conforming adaptive mesh refinement on quadrilateral and hexahedral meshes (non-conforming h-adaptivity), and support for mesh optimization by node movement (r-adaptivity). The unified support for local refinement on simplex and tensor-product elements is one of the distinguishing features of the MFEM library.

These capabilities are described in the following subsections. Additional parallel conforming mesh adaptivity and modification algorithms are available via the integration with RPI's parallel unstructured mesh infrastructure (PUMI) [68], see the `mesh/pumi.hpp` file and the PUMI example codes in the `examples/pumi` directory.

7.1 Conforming Adaptive Mesh Refinement

The conforming h-adaptivity algorithm in MFEM is based on the bisection procedure for tetrahedral meshes proposed in [13]. This approach supports both uniform refinement of all elements in the mesh, as well as local refinement of only elements of interest with additional (forced) refinement of nearby elements to ensure a conforming mesh. Note that in parallel these forced refinements may propagate to neighboring processors, which MFEM handles automatically for the user.

When a tetrahedral mesh is marked for refinement with `Mesh::MarkForRefinement()` the vertices of each tetrahedron are permuted so that the longest edge of the tetrahedron becomes the edge between vertices 0 and 1. MFEM ensures that the longest edge in each tetrahedron is chosen consistently in neighbor tetrahedrons based on a global sort of all edges (by length). The edge between vertices 0 and 1 becomes the marked edge, i.e. the edge that will be bisected during refinement. Initially, this is the longest edge in the element (with equal length edges ordered according to the global sort). However, later, the bisection algorithm may choose to mark an edge that is not the longest. When a tetrahedron is bisected, its type (M, A, etc., see [13]) determines which edges in the two children become marked, as well as what types are assigned to them. The initial type of the tetrahedron is also determined based on the globally sorted edges, see the method `Tetrahedron::MarkEdge()` in `tetrahedron.cpp` for more details.

The bisection algorithm consists of several passes. For example, during *green* refinement, every tetrahedron is checked if it "needs refinement" by calling the method `Tetrahedron::NeedRefinement()`

and if it does, the element is bisected once. The method `NeedRefinement()` returns true if any of its edges have been refined. When a tetrahedron is bisected, it is replaced (in the list of elements) by one of its children and the other child is appended at the end of the element list. That way, the children will be checked if they need refinement in the next loop over the elements. If no elements "need refinement", the green refinement step is done.

We note that the algorithm implemented in MFEM is not exactly the algorithm from [13]. Specifically, after refinement (including propagation of forced refinements) we replace the type of the Pf tetrahedrons with type Pu. The reason for the modification is to avoid cases where more than three levels of bisection need to be performed in a single call to `Mesh::LocalRefinement()`. (To support that case, one needs a more flexible data structure than the `DSTable` class, e.g. `DSTable` does not allow insertion of newly created vertices and the corresponding newly created edges.) When performing local refinement (i.e. when not all elements in the mesh are refined) one can observe some deterioration of the quality of the tetrahedrons after consecutive local refinements. With uniform refinement (i.e. when all elements are refined), the quality of the tetrahedrons may get worse only after the first two levels of uniform refinement. After that no new shapes of tetrahedrons can be generated through uniform refinement.

In parallel, the tetrahedrons are marked consistently across processors, as inherited from the serial mesh before the parallel partitioning. The consistently marked tetrahedrons guarantee that a face between any two tetrahedrons will be refined the same way from both sides. This implies in particular that uniform refinement can be performed in parallel without communication. In the case of local refinement we need to know which of the five possible cases of face refinement was actually performed on the other side of a shared face. These five cases are as described in the comment before the implementation of `ParMesh::GetFaceSplittings()`.

7.2 Non-Conforming Adaptive Mesh Refinement

Many high-order applications can be enriched by parallel adaptive mesh refinement (AMR) on unstructured quadrilateral and hexahedral meshes. Quadrilateral and hexahedral elements are attractive for their tensor product structure (enabling efficiency) and for their refinement flexibility (enabling e.g., *anisotropic* refinement). However, as opposed to the bisection-based methods for simplices considered in the previous section, *hanging* nodes that occur after local refinement of quadrilaterals and hexahedra are not easily avoided by further refinement [40, 105, 61]. We are thus interested in *non-conforming* (irregular) meshes, in which adjacent elements need not share a complete face or edge and where some finite element degrees of freedom (DOFs) need to be constrained to obtain a conforming solution.

In this section we review MFEM's software abstractions and algorithms for handling parallel non-conforming meshes on a general discretization level, independent of the physics simulation. These methods support the entire de Rham sequence of finite element spaces (see Subsection 4.2), at arbitrarily high-order, and can support high-order curved meshes, as well as finite element techniques such as hybridization and static condensation (see Subsection 5.2). They are also highly scalable, easy to incorporate into existing codes, and can be applied to complex (anisotropic, n-irregular) 3D meshes. While MFEM's approaches can be extended to general hp-refinement, the current implementation focuses exclusively on non-conforming h-refinement with a fixed $p \geq 1$.

A lot of the above benefits stem from the variational restriction approach to AMR that we describe below. For more details, see [31].

Consider the weak variational formulation (9) where for simplicity we assume that the bilinear form $a(\cdot,\cdot)$ is symmetric. To discretize the problem, we cover the computational domain Ω with a mesh consisting of mutually disjoint elements K_i, their vertices V_j, edges E_m, and faces F_n. Except

for the vertices, we consider these entities as open sets, so that $\Omega = (\cup_i K_i) \cup (\cup_j V_j) \cup (\cup_m E_m) \cup (\cup_n F_n)$. In the case of non-conforming meshes, there exist faces F_s that are strict subsets of other faces, $F_s \subsetneq F_m$, see Figure 10. We call F_s *slave faces* and F_m *master faces*. The remaining standard faces F_c are disjoint with all other faces and will be referred to as *conforming faces*. Similarly, we define *slave edges*, *master edges* and *conforming edges*.

Non-conforming meshes in MFEM are represented by the `NCMesh` and `ParNCMesh` classes described in files `mesh/ncmesh.hpp` and `mesh/pncmesh.hpp` respectively. We use an octree data structure to represent refinements which has been optimized to rely only on the following information: 1) edges are identified by pairs of vertices; 2) faces are identified by four vertices; 3) elements contain indices of eight vertices, or indices of eight child elements if refined. Edges and faces are tracked by associative maps (see below), which reduce both code complexity and memory footprint. In the case of a uniform hexahedral mesh, our data structure requires about 290 bytes per element, counting the complete refinement hierarchy and including vertices, edges, and faces.

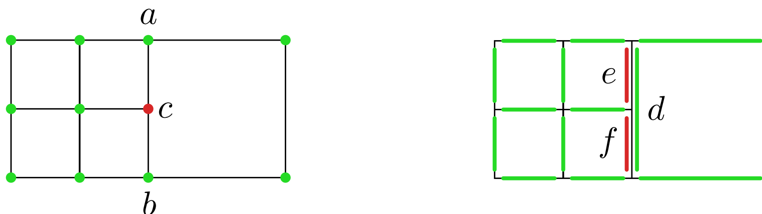

Figure 10: Illustration of conformity constraints for lowest order nodal elements in 2D. Left: Nodal elements (subspace of H^1), constraint $c = (a+b)/2$. Right: Nedelec elements (subspace of $H(\text{curl})$, constraints $e = f = d/2$. In all cases, fine degrees of freedom on a coarse-fine interface are linearly interpolated from the values at coarse degrees of freedom on that interface.

To construct a standard finite dimensional FEM approximation space $V_h \subset V$ on a given non-conforming mesh, we must ensure that the necessary conformity requirements are met between the slave and master faces and edges so that we get V_h that is a (proper) subspace of V. For example, if V is the Sobolev space H^1, the solution values in V_h must be kept continuous across the non-conforming interfaces. In contrast, if V is an $H(\text{curl})$ space, the tangential component of the finite element vector fields in V_h needs to be continuous across element faces. More generally, the conformity requirement can be expressed by requiring that values of V_h functions on the slave faces (edges) are interpolated from the finite element function values on their master faces (edges). Finite element degrees of freedom on the slave faces (and edges) are thus effectively constrained and can be expressed as linear combinations of the remaining degrees of freedom. The simplest constraints for finite element subspaces of H^1 and $H(curl)$ in 2D are illustrated in Figure 10.

The degrees of freedom can be split into two groups: *unconstrained (or true)* degrees of freedom and *constrained (or slave)* degrees of freedom. If z is a vector on all slave DOFs, then z can be expressed as $z = Wx$, where x is a vector on all true DOFs and W is a global interpolation matrix, handling indirect constraints and arbitrary differences in refinement levels of adjacent elements. Introducing the *conforming prolongation matrix*

$$P = \begin{pmatrix} I \\ W \end{pmatrix},$$

we observe that the coupled AMR linear system can be written as

$$P^T A P x_c = P^T b, \tag{28}$$

where A and b are the finite element stiffness matrix and load vector corresponding to discretization of (9) on the broken space $\widehat{V}_h = \cup_i(V_h|_{K_i})$. After solving for the true degrees of freedom x_c we recover the complete set of degrees of freedom, including slaves, by calculating $x = Px_c$. Note that in MFEM this is handled automatically for the user via `FormLinearSystem()` and `RecoverFEMSolution()`, see Subsection 5.2. An illustration of this process is provided in Figure 11.

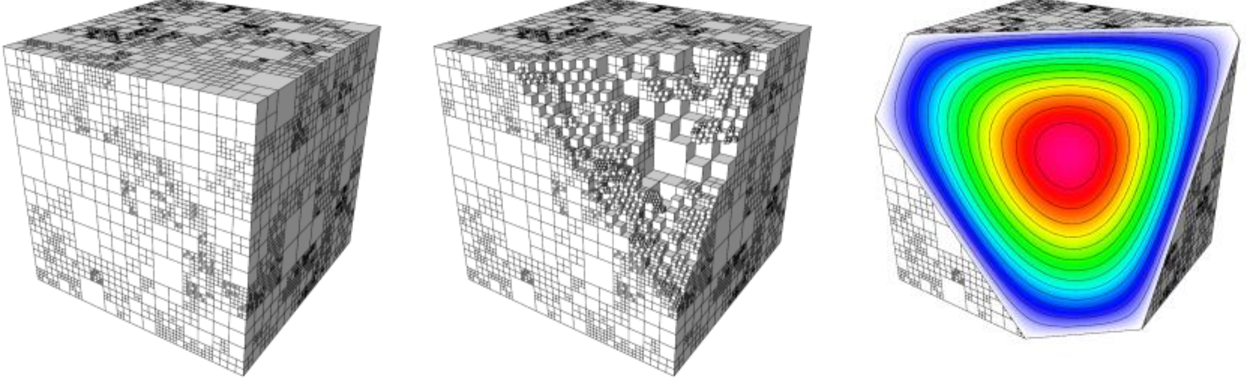

Figure 11: Illustration of the variational restriction approach to forming the global AMR problem. Randomly refined non-conforming mesh (left and center) where we assemble the matrix A and vector b independently on each element. The interpolated solution $x = Px_c$ (right) of the system (28) is globally conforming (continuous for an H^1 problem).

In MFEM, given an `NCMesh` object, the conforming prolongation matrix can be defined for each `FiniteElementSpace` class and accessed with the `GetConformingProlongation()` method. The algorithm for constructing this operator can be interpreted as a sequence of interpolations $P = P_k P_{k-1} \cdots P_1$, where for a k-irregular mesh the DOFs in $\widehat{V}_h$ are indexed as follows: 0 – true DOFs, 1 – first generation of slaves that only depend on true DOFs, 2 – second generation of slaves that only depend on true DOFs and first generation of slaves, ..., k – last generation of slaves, and

$$P_1 = \begin{pmatrix} I \\ W_{10} \end{pmatrix},\ P_2 = \begin{pmatrix} I & 0 \\ 0 & I \\ W_{20} & W_{21} \end{pmatrix},\ \ldots,\ P_k = \begin{pmatrix} I & 0 & \cdots & 0 \\ 0 & I & \cdots & 0 \\ & & \ddots & \\ 0 & 0 & \cdots & I \\ W_{k0} & W_{k1} & \cdots & W_{k(k-1)} \end{pmatrix}$$

are the local interpolation matrices defined only in terms of the edge-to-edge and face-to-face constraining relations.

The basis for determining face-to-face relations between hexahedra is the function `FaceSplitType`, sketched below. Given a face (v_1, v_2, v_3, v_4), it tries to find mid-edge and mid-face vertices and determine if the face is split vertically, horizontally (relative to its reference domain), or not split.

```
1 Split FaceSplitType(v1, v2, v3, v4)
2 {
3    v12 = FindVertex(v1 , v2);
4    v23 = FindVertex(v2 , v3);
5    v34 = FindVertex(v3 , v4);
6    v41 = FindVertex(v4 , v1);
7
8    midf1 = (v12 != NULL && v34 != NULL) ? FindVertex(v12, v34) : NULL;
```

```
   midf2 = (v23 != NULL && v41 != NULL) ? FindVertex(v23, v41) : NULL;

   if (midf1 == NULL && midf2 == NULL)
      return NotSplit;
   else
      return (midf1 != NULL) ? Vertical : Horizontal;
}
```

The function FindVertex uses a hash table to map end-point vertices to the vertex in the middle of their edge. This algorithm naturally supports anisotropic refinement, as illustrated in Figure 12.

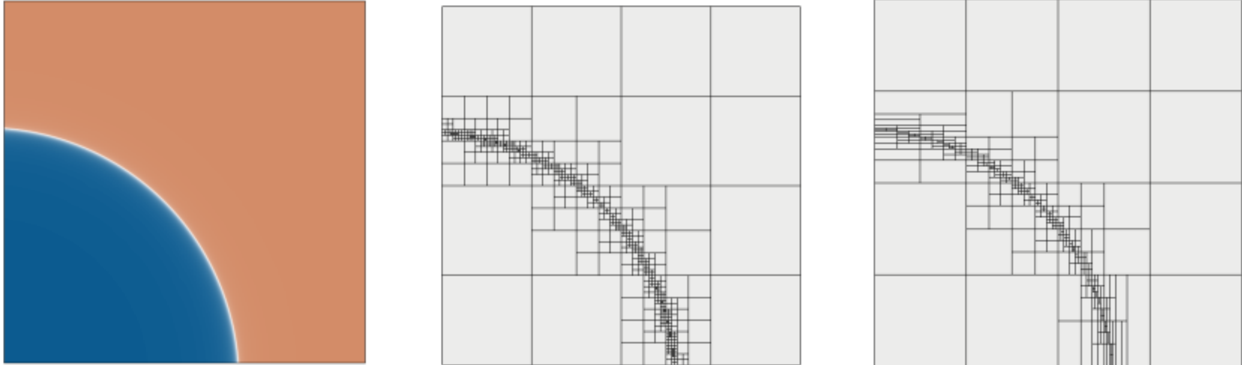

Figure 12: Left: 2D benchmark problem for a Poisson problem with a known exact solution. Center: Isotropic AMR mesh at 2197 DOFs. Right: Anisotropic AMR mesh at 1317 DOFs. Even though the wave front in the solution is not really aligned with the mesh, many elements could still be refined in one direction only, which saved up to 48% DOFs in this problem for similar error.

The algorithm to build the P matrix in parallel is more complex, but conceptually similar to the serial algorithm. We still express slave DOF rows of P as linear combinations of other rows, however some of them may be located on other MPI tasks and may need to be communicated first. The mesh is partitioned between MPI tasks by splitting the space-filling curve obtained by enumerating depth-first all leaf elements of all refinement octrees. The simplest traversal with a fixed order of children at each octree node leads to the well-known Morton ordering, or the Z-curve. We use instead the more efficient Hilbert curve that can be obtained just by changing the order of visiting subtrees at each level of the octree [23]. The use of space-filling curve partitioning ensures that balancing the mesh so that each MPI task has the same number of elements (±1 if the total number of elements is not divisible by the number of tasks) is relatively straightforward.

These algorithms have been heavily optimized for both weak and strong parallel scalability as illustrated in Figure 13, where we report results from a 3D Poisson problem on the unit cube with exact solution having two shock-like features. We initialize the mesh to 32^3 hexahedra and repeat the following steps, measuring their wall clock times (averaged over all MPI ranks): 1) Construct the finite element space for the current mesh (create the P matrix); 2) Assemble locally the stiffness matrix A and right hand side b; 3) Form the products P^TAP, P^Tb; 4) Eliminate Dirichlet boundary conditions from the parallel system; 5) Project the exact solution u to u_h by nodal interpolation; 6) Integrate the exact error $e_i = ||u_h - u||_{E,K_i}$ on each element; 7) Refine elements with $e_j > 0.9 \max\{e_i\}$; 8) Load balance so each process has the same number of elements (±1). Out of the approximately 100 iterations of the AMR loop, we select 8 iterations that have approximately 0.5, 1, 2, 4, 8, 16, 32 and 64 million elements in the mesh at the beginning of the iteration and plot their times as if they were 8 independent problems. We run from 64 to 393216

(384K) cores on LLNL's Vulcan BG/Q machine. The solid lines in Figure 13 show strong scaling, i.e. we follow the same AMR iteration and its total time as the number of cores doubles. The dashed lines skip to a double-sized problem when doubling the number of cores showing weak scaling, and should ideally be horizontal. To the best of our knowledge, these are some of the best weak and strong scaling results for a fully unstructured AMR implementation.

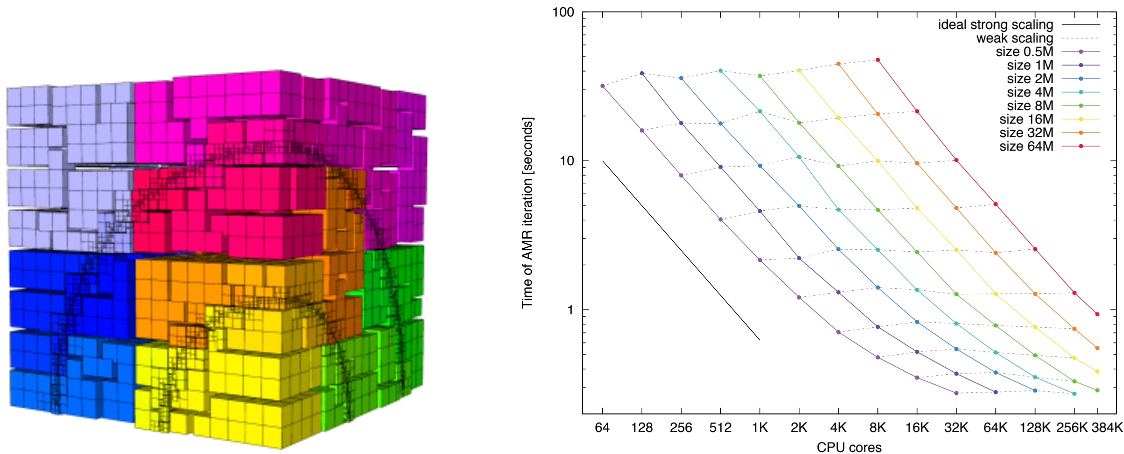

Figure 13: Left: One octant of the parallel test mesh partitioned by the Hilbert curve (2048 domains shown). Right: Overall parallel weak and strong scaling for selected iterations of the AMR loop. Note that these results test only the AMR infrastructure, no physics computations are being timed.

MFEM's variational restriction-based AMR approach can be remarkably unintrusive when it comes to integration in a real finite element application code. To illustrate this point we show two results from the **Laghos** miniapp (see Subsection 8.3) which required minimal changes for static refinement support, see Figure 14 and about 550 new lines of code for full dynamic AMR, including derefinement, see Figure 15.

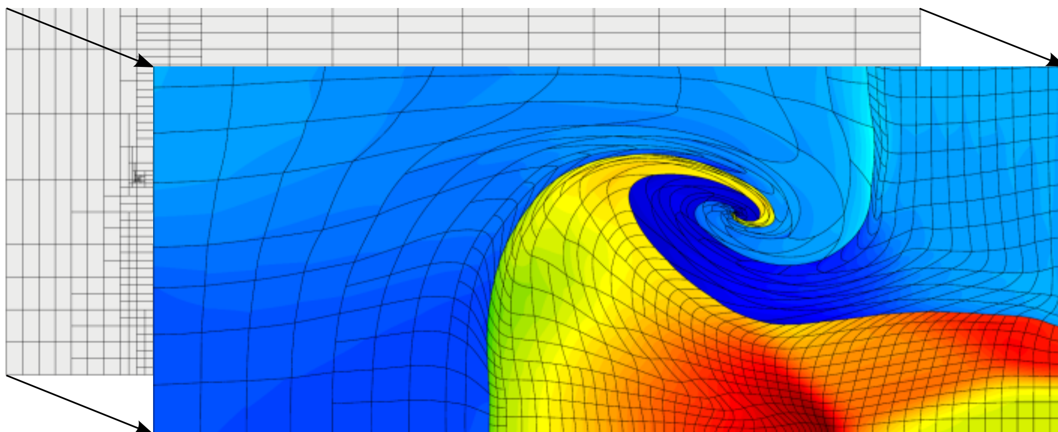

Figure 14: MFEM-based static refinement in a triple point shock-interaction problem. Initial mesh at $t = 0$ (background) refined anisotropically in order to obtain more regular element shapes at target time (foreground).

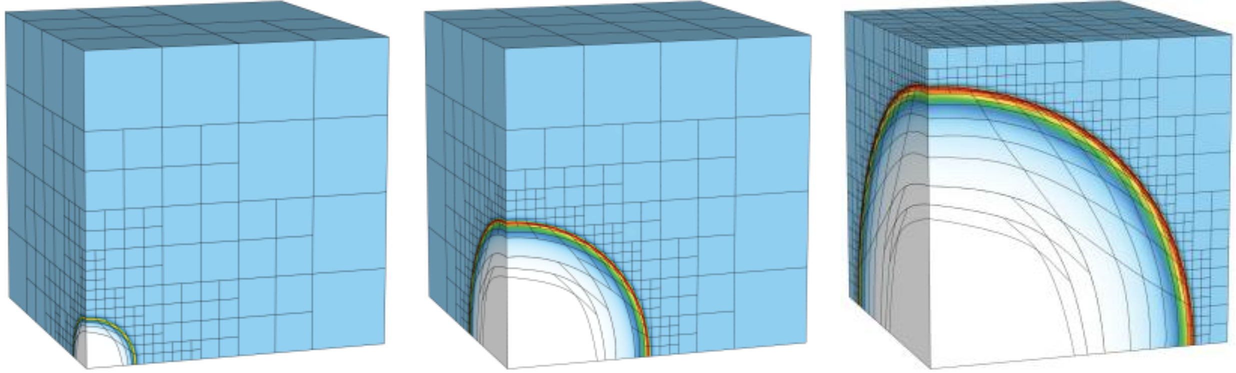

Figure 15: MFEM-based dynamic refinement/derefinement in the 3D Sedov blast problem. Mesh and density shown at $t = 0.0072$ (left), $t = 0.092$ (center) and $t = 0.48$ (right). Q3Q2 elements ($p = 3$ kinematic, $p = 2$ thermodynamic quantities).

7.3 Mesh Optimization

A vital component of high-order methods is the use of high-order representation not just for the *physics* fields, but also for the geometry, represented by a high-order computational mesh. High-order meshes can be relatively coarse and still capture curved geometries adequately, leading to equivalent simulation quality for a smaller number of elements. High-order meshes can also be very beneficial in a wide range of applications, where e.g. radial symmetry preservation, or alignment with physics flow or curved model boundary is important [103, 49, 82]. Such applications can utilize *static* meshes, where a good-quality high-order mesh needs to be generated only as an input to the simulation, or *dynamic* meshes, where the mesh evolves with the problem and its quality needs to be constantly controlled. In both cases, the quality of high-order meshes can be difficult to control, because their properties vary in space on a sub-zonal level. Such control is critical in practice, as poor mesh quality leads to small time steps or simulation failures.

The MFEM project has developed a general framework for the optimization of high-order curved meshes based on the node-movement techniques of the Target-Matrix Optimization Paradigm (TMOP) [73]. This enables applications to have very precise control over local mesh quality, while still optimizing the mesh globally. Note that while our new methods are targeting high-order meshes, they are general, e.g., they can also be applied to low-order mesh applications that use linear meshes.

A brief description of the MFEM mesh optimization framework is provided in the rest of this section. For more details see [42, 43]. To the best of our knowledge this is the first ever user-friendly implementation of general parallel high-order mesh optimization tools in open-source software.

TMOP is a general approach for controlling mesh quality, where mesh nodes (vertices in the low-order case) are moved so-as to optimize a multi-variable objective function that quantifies global mesh quality. Specifically, at a given point of interest (inside each mesh element), TMOP uses three Jacobian matrices:

- The Jacobian matrix $A_{d \times d}$ of the transformation from reference to physical coordinates, where d is the space dimension.

- The *target matrix*, $W_{d \times d}$, which is the Jacobian of the transformation from the reference to the *target* coordinates. The target matrices are defined according to a user-specified method prior to the optimization; they define the desired properties in the optimal mesh.

- The *weighted Jacobian* matrix, $T_{d\times d}$, defined by $T = AW^{-1}$, represents the Jacobian of the transformation between the target and the physical coordinates.

The T matrix is used to define the *local quality measure*, $\mu(T)$. The quality measure can evaluate shape, size, or alignment of the region around the point of interest. The combination of targets and quality metrics is used to optimize the node positions, so that they are as close as possible to the shape/size/alignment of their targets. This is achieved by minimizing a global *objective function*, $F(x)$, that depends on the local quality measure throughout the mesh:

$$F(x) := \sum_{E\in\mathcal{E}} \int_{E_t} \mu(T(x))dx_t = \sum_{E\in\mathcal{E}} \sum_{x_q\in Q_E} w_q \det(W(x_q))\, \mu(T(x_q))\,, \tag{29}$$

where E_t is the target element corresponding to the physical element E, Q_E is the set of quadrature points for element E, w_q are the corresponding quadrature weights, and both $T(x_q)$ and $W(x_q)$ are evaluated at the quadrature point x_q of element E. The objective function can be extended by using combinations of quality metrics, space-dependent weights for each metric, and limiting the amount of allowed mesh displacements. As $F(x)$ is nonlinear, MFEM utilizes Newton's method to solve the critical point equations, $\partial F(x)/\partial \boldsymbol{x} = 0$, where $\boldsymbol{x}$ is the vector that contains the current mesh positions. This approach involves the computation of the first and second derivatives of $\mu(T)$ with respect to T. Furthermore, boundary nodes are enforced to stay fixed or move only in the boundary's tangential direction. Additional modifications are performed to guarantee that the Newton updates do not lead to inverted meshes, see [42].

The current MFEM interface provides access to 12 two-dimensional mesh quality metrics, 7 three-dimensional metrics, and 5 target construction methods, together with the first and second derivatives of each metric with respect to the matrix argument. The quality metrics are defined by the inheritors of the class `TMOP_QualityMetric`, and target construction methods are defined by the class `TargetConstructor`, which are all contained in the source file `fem/tmop.hpp`. The file `linalg/invariants.hpp` contains computations of matrix invariants and their first and second derivatives (with respect to the matrix), which are used by MFEM's `NewtonSolver` class to solve $\partial F(x)/\partial \boldsymbol{x} = 0$. The library interface allows users to choose between various options concerning target construction methods and mesh quality metrics and adjust various parameters depending on their particular problem. The mesh optimization module can be easily extended by additional mesh quality metrics and target construction methods. Illustrative examples are presented in the form of a simple mesh optimization miniapp, `mesh-optimizer`, in the `miniapps/meshing` directory, which includes both serial and parallel implementations. Some examples of simulations that can be performed by this miniapp are shown in Figure 16. MFEM's mesh optimization capabilities are also routinely used in production runs for many of the ALE simulation problems in the BLAST code, see Figure 17.

Work to extending MFEM's mesh optimization capabilities to simulation-driven adaptivity (a.k.a. r-adaptivity) [43], and coupling h- and r-adaptivity of high-order meshes by combining the TMOP and AMR concepts is ongoing. See Figure 18 for some preliminary results in that direction.

8 Applications

MFEM has been used in many applications and research publications, including [75, 31, 48, 14, 8, 17, 42, 49, 112, 113, 3, 97, 70, 80, 33, 95, 53, 83, 18, 46, 101, 78, 45, 44, 37, 47, 9, 111, 64, 50, 57, 76, 16, 6, 96, 2, 86, 115, 27, 28, 51, 100, 84, 117, 56, 118, 77, 116, 106, 63, 21, 81, 102]. The project is a member of the FASTMath institute in the SciDAC program and is also a major

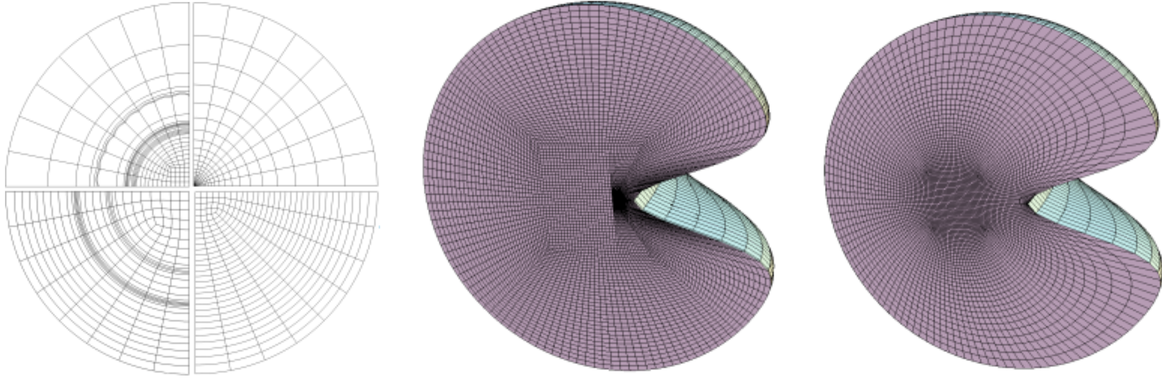

Figure 16: In the left panel, a perturbed (top left) fourth order 2D mesh is being optimized by targeting shape-only optimization (top right), shape and equal size (bottom right), shape and space-dependent size (bottom left). In the middle and right panels, a second order 3D mesh is being optimized with respect to shape.

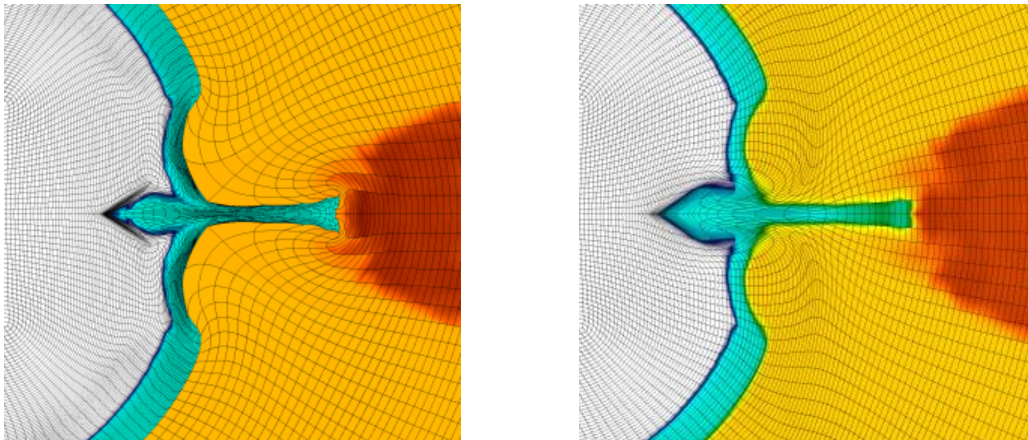

Figure 17: Optimized meshes in parallel inertial confinement fusion simulation, see [8] and Subsection 8.3. Shown is the region around the capsule's fill tube. Both meshes are optimized with respect to shape, size, and amount of mesh displacement. On the left, material interfaces are kept fixed. On the right, interfaces are relaxed later in the simulation.

participant in the co-design Center for Efficient Exascale Discretizations (CEED) in the Exascale Computing Project (ECP) where MFEM's software is being upgraded to exploit increasing on-node concurrency targeting multiple complex architectures, including GPUs, see Subsection 6.3.

8.1 Example Codes

The MFEM codebase includes a wide array of example applications that utilize numerous MFEM features and demonstrate the finite element discretization of many PDEs. The goal of these well documented example codes is to provide a step-by-step introduction to MFEM in simple model settings. Most of the examples have both serial and parallel versions (indicated by a `p` appended to the filename) which illuminate the straightforward transition to parallel code and the use of the *hypre* solvers and preconditions. There are also variants of many example codes in the `petsc`, `sundials` and `pumi` subdirectories that display integration with those packages. Each example code

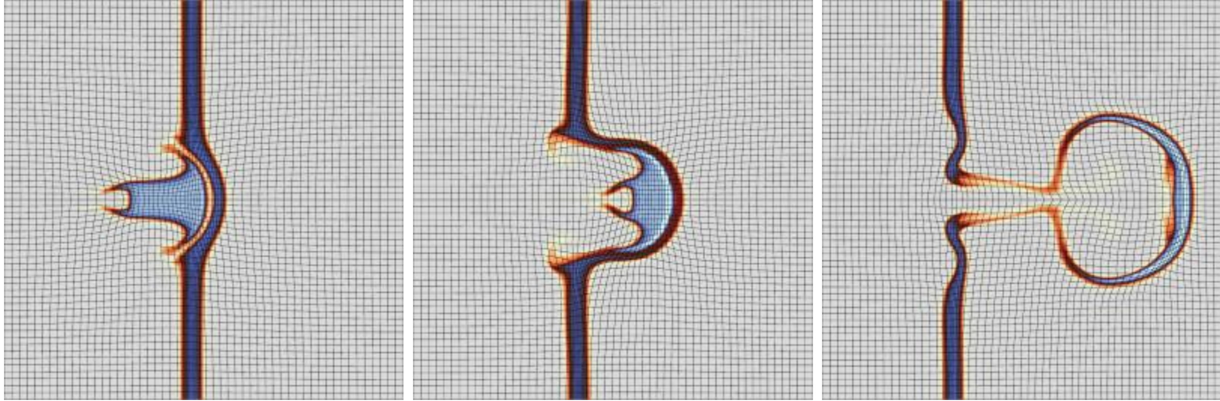

Figure 18: Example of MFEM r-adaptivity to align the mesh with materials in a multi-material ALE simulation of high velocity gas impact, cf. [19].Time evolution of the materials and mesh positions at times 2.5 (left), 5 (center), and 10 (right). See [43] for details.

has the flexibility to change the order of the calculations, switch various finite element features on or off, and utilize different meshes through command line options. Once the example codes are built using the accompanying `makefile`, their options can be displayed by running the code with `--help` as a command line option. The outputs of the examples can be visualized with GLVis, see Subsection 4.4.

The example codes are simply named `ex1`–`ex19`, roughly in order of complexity, so it is recommended that users start with earlier numbered examples in order to learn the basics of interfacing with MFEM before moving on to more complicated examples. Here we will provide a brief overview of these examples and the topics they elucidate. More details can be found in our online documentation `http://mfem.org/examples/`.

The first example `ex1` begins with the solution of the Laplace problem with homogeneous Dirichlet (fixed) boundaries utilizing nodal H^1 elements. This parallels the problem solved in Section 2. Examples `ex6`, `ex8`, and `ex14` also solve the Poisson problem, but they also highlight AMR, DPG and DG formulations, respectively. Examples `ex2` and `ex17` solve the equations of linear elasticity with Galerkin and DG formulations, respectively, while `ex10` provides an implementation of nonlinear elasticity utilizing a Newton solver. An elementary introduction to utilizing $H(\mathrm{curl})$ vector elements to solve problems arising from Maxwell's equations can be found in `ex3` and `ex13`. An example of utilizing surface meshes embedded in a 3D space can be found in `ex7` while more advanced dynamic AMR is explored in `ex15`. Finally, examples `ex9`, `ex10`, `ex16`, and `ex17` show the solution of time dependent problems, while `ex11`, `ex12`, and `ex13` tackle frequency domain problems solving for eigenvalues of their respective systems. Sample results from some of the example runs are shown in Figure 19.

8.2 Electromagnetics

The electromagnetic miniapps in MFEM are designed to provide a jumping-off point for developing real-world electromagnetic applications. As such, they cover a few common problem domains and attempt to support a variety of boundary conditions and source terms. This way application scientists can easily adapt these miniapps to solve particular problems arising in their research.

The **Volta** miniapp solves Poisson's equation with boundary conditions and sources tailored to electrostatic problems. This miniapp supports fixed voltage or fixed charge density boundary

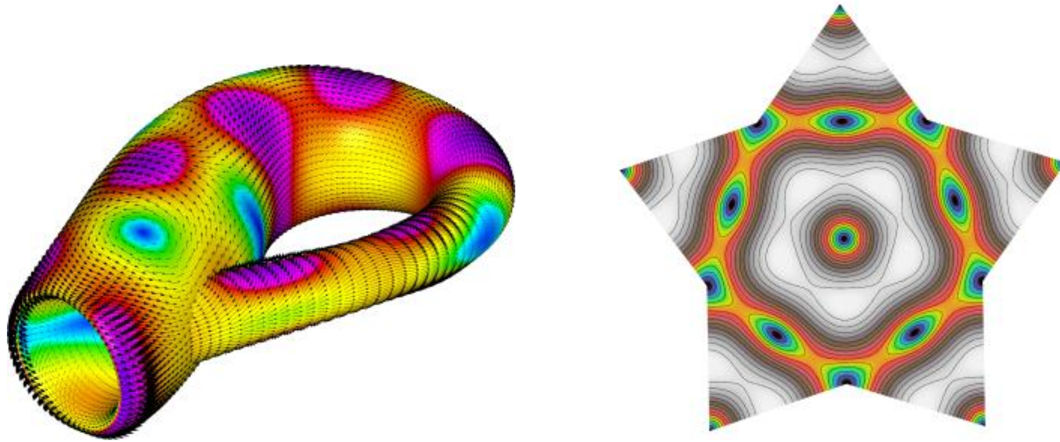

Figure 19: Left: Solution of a Maxwell problem on a Klein bottle surface with `ex3`; mesh generated with the `klein-bottle` miniapp in `miniapps/meshing`. Right: An electromagnetic eigenmode of a star-shaped domain computed with 3rd order finite elements computed with `ex13p`.

conditions which correspond to the usual Dirichlet or Neumann boundary conditions, respectively. The volumetric source terms can be derived from either a prescribed charge density or a fixed polarization field.

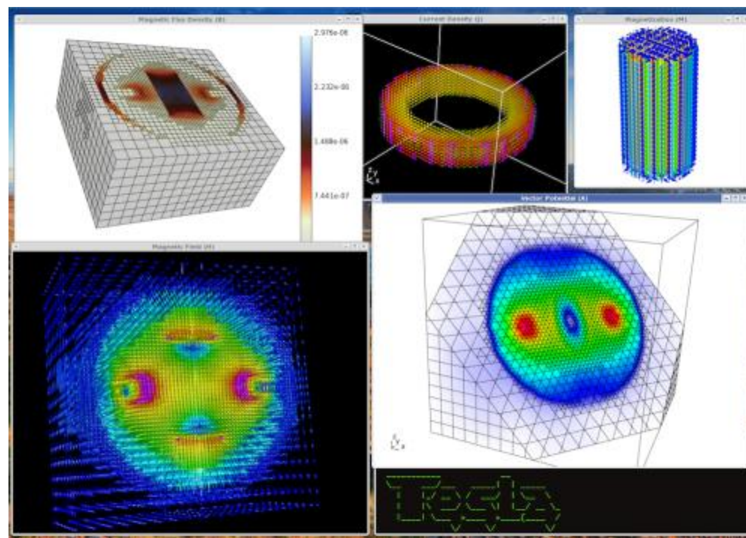

Figure 20: Screen shot showing some of the fields computed using the **Tesla** miniapp.

The **Tesla** miniapp models magnetostatic problems. Magnetostatic boundary conditions are more complicated than those for electrostatic problems due to the nature of the curl-curl operator. We support two types of boundary conditions: the first leads to a constant magnetic field at the boundary, the second arises from a surface current. The surface current is itself the solution of a Poisson problem restricted to the surface of the problem domain. The motivation for this surface current boundary condition is to approximate the magnetic fields surrounding current carrying conductors. **Tesla** also supports volumetric sources due to current densities or materials with a fixed magnetization (i.e. permanent magnets). Note that the curl-curl operator cannot be solved with an arbitrary source term; the source must be a solenoidal vector field. To ensure this, the

Tesla miniapp must remove any irrotational components by performing a projection operation known as "divergence cleaning".

The **Maxwell** miniapp simulates full-wave time-domain electromagnetic wave propagation. This miniapp solves the Maxwell equations as a pair of coupled first order partial differential equations using a symplectic time-integration algorithm, see Subsection B.4, designed to conserve energy (unless, of course, lossy materials are present.) The simulation can be driven by a time-varying applied electric field boundary condition or by a volumetric current density. Perfect electrical conductor, perfect magnetic conductor, and first order Sommerfeld absorbing boundary conditions are also available. A frequency-domain version of this miniapp is currently under development.

One of the most practical applications of electromagnetics is the approximation of the Joule heating caused by an alternating electrical current in an imperfect conductor. The MFEM miniapp **Joule** models this behavior with a system of coupled partial differential equations which approximate low-frequency electromagnetics and thermal conduction. The boundary conditions consist of a time-varying voltage, used to drive a volumetric current, and a thermal flux boundary condition, which can approximate a thermal insulator. This miniapp is a good example of a simple multiphysics application which could be modified to simulate a variety of important real-world problems in electrical engineering.

8.3 Compressible Hydrodynamics

The MFEM-based **Laghos** miniapp models time-dependent, compressible, inviscid gas dynamics via the Euler equations in the Lagrangian form. The Euler equations are a system which describe the conservation of mass, momentum, and energy of an inviscid fluid. In the Lagrangian picture, the elements represent regions of fluid of fixed identity that move with the flow, resulting in a moving and deforming mesh. The high-order curved mesh capabilities of MFEM provide a great advantage in this context, since curved meshes can describe larger deformations more robustly than meshes using only straight segments, and mitigate problems with the mesh intersecting itself when it becomes highly deformed.

The **Laghos** miniapp uses continuous finite element spaces to describe the position and velocity fields, and a discontinuous space to describe the energy field. The order of these fields is determined by runtime parameters, making the code arbitrarily high order. The assembly of the finite elements in **Laghos** can be accomplished using either standard *full assembly*, or by a method which is more efficient in the high-order case known as *partial assembly*, see Subsection 5.4. With partial assembly, the global matrices are never fully created and stored, but rather only the local action of these operators is required. This reduces both memory footprint and computational cost.

Laghos is also a simplified model for a more complex multi-physics MFEM-based code system known as **BLAST** [8, 49, 47], which contains not only Lagrangian gas dynamics capabilities, but also has mesh remapping or ALE capabilities, solid mechanics, and multi-material zones. The remapping capability allows arbitrarily large deformations to be modeled, since the mesh can be regularized at intervals sufficient to continue a simulation indefinitely. This capability to vary smoothly between Lagrangian and Eulerian meshes is known as the Arbitrary Lagrangian-Eulerian method, or ALE. The remap capability in **BLAST** is accomplished with a high-order Discontinuous Galerkin method, which is both conservative and monotonic [7]. (The DG component of the remap algorithm is very close to the DG advection in Example 9.)

BLAST uses a general stress tensor formulation which allows for the simulation of elastic-plastic flows in 2D, 3D, and in axisymmetric coordinates. Multi-material elements are described using high-order material indicator functions, which describe the volume fractions of materials at all points in the domain. A new, high-order multi-material closure model was developed to solve

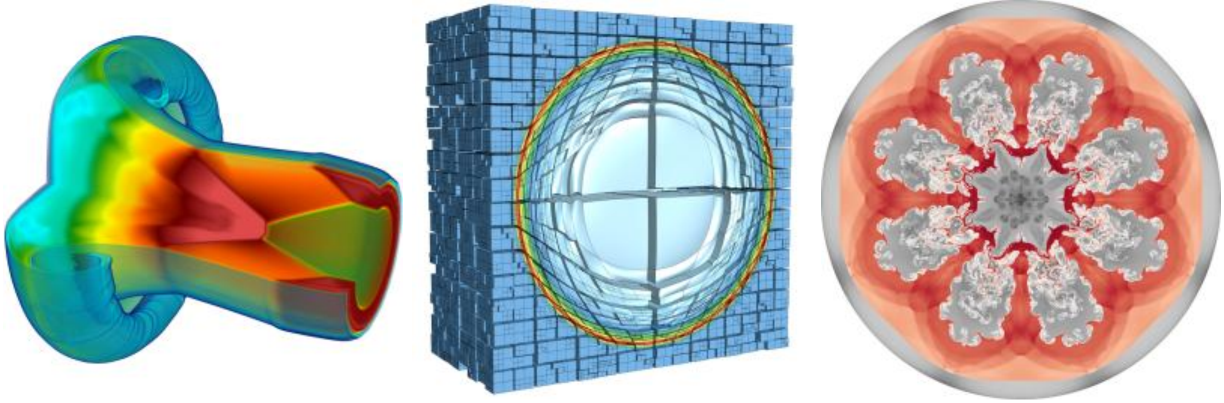

Figure 21: Left: A shock interface 3D hydrodynamics calculation on 16,384 processors with BLAST. Center: Static non-conforming AMR in a Sedov blast simulation, see Subsection 7.2. Right: High-order axisymmetric multi-material inertial confinement fusion (ICF)-like implosion in BLAST.

the resulting multi-material system of equations [50]. This capability has been used to model many types of hydrodynamic systems, such as Rayleigh-Taylor instability, shock-interface interactions, solid impact problems, and inertial confinement fusion dynamics. Some examples of **BLAST** calculations are shown in Figure 21.

8.4 Other Applications

MFEM has been applied successfully to a variety of applications including radiation-diffusion, additive manufacturing, topology optimization, heart modeling applications, linear and nonlinear elasticity, reaction-diffusion, time-domain electromagnetics, DG advection problems, Stokes/Darcy flow, and more. Two examples of such applications are the Cardioid and ParElag project described below.

The Cardioid project at LLNL recently used MFEM to rewrite and simplify two cardiac simulation tools. The first is a fiber generation code which solves a series of Poisson problems to compute cardiac fiber orientations on a given mesh. See Figure 22 for sample output. Additionally, a deformable cardiac mechanics code which solves incompressible anisotropic hyperelasticity equations with active tension has also been developed. The methods implemented are outlined in [20] and [58]. A second MFEM-based code to generate electrocardiograms using simulated electrophysiology data is also under development.

Another project that is built extensively around MFEM is ParElag [98], which is a library organized around the idea of algebraic coarsening of the de Rham complex introduced in Subsection 4.2. ParElag leverages MFEM's high-order Lagrange, Nedelec, and Raviart-Thomas finite element spaces as well as its auxiliary space solvers (Subsection 6.2) to systematically provide a de Rham complex on a coarse level, even for unstructured grids with no geometric hierarchy. Its algorithms and approach are described in [79, 99].

9 Conclusions

In this paper we provided an overview of the algorithms, capabilities and applications of the MFEM finite element library as of version 3.4, released in May 2018. Our goal was to emphasize the

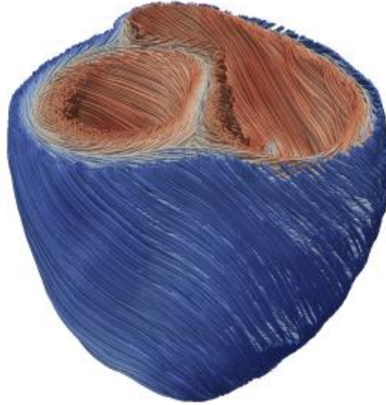

Figure 22: Heart fiber orientations computation in Cardioid using MFEM.

mathematical ideas and software design choices that enable MFEM to be widely applicable and highly performant from a relatively small and lightweight code base.

While we tried to cover all major MFEM components, this paper is really just an introduction to MFEM, and readers interested in learning more should consult the additional material available on the website `http://mfem.org` and in the MFEM code distribution.

In particular, new users should start with the interactive documentation of the example codes, available online as well as in the `examples/` directory, and may be interested in reading some of the references in Section 8, e.g. [8, 115, 86, 64, 101, 18].

Researchers interested in learning mathematical details about MFEM's finite element algorithms and potentially contributing to the library can follow up with [31, 42, 17, 48, 43] and the instructions/developer documentation in the `CONTRIBUTING.md` file.

References

[1] AcroTensor: Fast and flexible large scale tensor operations. `https://github.com/LLNL/acrotensor`. {31}

[2] J. H. Adler, I. Lashuk, and S. P. MacLachlan. Composite-grid multigrid for diffusion on the sphere. *Numerical Linear Algebra with Applications*, 25(1):e2115, 2017. {4, 39}

[3] J. H. Adler and P. S. Vassilevski. Error analysis for constrained first-order system least-squares finite-element methods. *SIAM Journal on Scientific Computing*, 36(3):A1071–A1088, 2014. {39}

[4] R. Alexander. Diagonally implicit Runge-Kutta methods for stiff O.D.E.s. *SIAM Journal on Numerical Analysis*, 14(6):1006–1021, 1977. {61}

[5] M. S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M. E. Rognes, and G. N. Wells. The FEniCS project version 1.5. *Archive of Numerical Software*, 3(100):9–23, 2015. {4}

[6] R. Anderson, V. Dobrev, T. Kolev, D. Kuzmin, M. Q. de Luna, R. Rieben, and V. Tomov. High-order local maximum principle preserving (MPP) discontinuous Galerkin finite element method for the transport equation. *Journal of Computational Physics*, 334:102–124, 12 2016. {4, 39}

[7] R. W. Anderson, V. A. Dobrev, T. V. Kolev, and R. N. Rieben. Monotonicity in high-order curvilinear finite element ALE remap. *International Journal for Numerical Methods in Fluids*, 77(5):249–273, 2014. {43}

[8] R. W. Anderson, V. A. Dobrev, T. V. Kolev, R. N. Rieben, and V. Z. Tomov. High-order multi-material ALE hydrodynamics. *SIAM Journal on Scientific Computing*, 40(1):B32–B58, 2018. {4, 39, 40, 43, 45}

[9] A. Aposporidis, P. S. Vassilevski, and A. Veneziani. Multigrid preconditioning of the non-regularized augmented Bingham fluid problem. *ETNA. Electronic Transactions on Numerical Analysis*, 41, 01 2014. {39}

[10] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, and D. Wells. The `deal.II` library, version 8.5. *Journal of Numerical Mathematics*, 25(3):137–146, 2017. {4}

[11] D. N. Arnold, F. Brezzi, B. Cockburn, and L. D. Marini. Unified analysis of discontinuous Galerkin methods for elliptic problems. *SIAM Journal on Numerical Analysis*, 39(5):1749–1779, May 2001. {3}

[12] D. N. Arnold, R. S. Falk, and R. Winther. Finite element exterior calculus, homological techniques, and applications. *Acta Numerica*, 15:1–155, 2006. {17}

[13] D. N. Arnold, A. Mukherjee, and L. Pouly. Locally adapted tetrahedral meshes using bisection. *SIAM Journal on Scientific Computing*, 22(2):431–448, 2000. {32, 33}

[14] H. Auten. The high value of open source software. *Science & Technology Review*, January/February 2018:5–11, 2018. {39}

[15] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, R. T. Mills, T. Munson, K. Rupp, P. Sanan, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang. PETSc users manual. Technical Report ANL-95/11 - Revision 3.9, Argonne National Laboratory, 2018. {59}

[16] R. E. Bank, P. S. Vassilevski, and L. T. Zikatanov. Arbitrary dimension convection-diffusion schemes for space-time discretizations. *Journal of Computational and Applied Mathematics*, 310:19–31, 2017. {4, 39}

[17] A. T. Barker, V. Dobrev, J. Gopalakrishnan, and T. Kolev. A scalable preconditioner for a primal discontinuous Petrov-Galerkin method. *SIAM Journal on Scientific Computing*, 40(2):A1187–A1203, 2018. {18, 39, 45}

[18] A. T. Barker, C. S. Lee, and P. S. Vassilevski. Spectral upscaling for graph Laplacian problems with application to reservoir simulation. *SIAM Journal on Scientific Computing*, 39(5):S323–S346, 2017. {4, 39, 45}

[19] A. Barlow, R. Hill, and M. J. Shashkov. Constrained optimization framework for interface-aware sub-scale dynamics closure model for multimaterial cells in Lagrangian and arbitrary Lagrangian-Eulerian hydrodynamics. *Journal of Computational Physics*, 276(0):92–135, 2014. {41}

[20] J. D. Bayer, R. C. Blake, G. Plank, and N. A. Trayanova. A novel rule-based algorithm for assigning myocardial fiber orientation to computational heart models. *Annals of Biomedical Engineering*, 40(10):2243–2254, Oct 2012. {44}

[21] J. Billings, A. McCaskey, G. Vallee, and G. Watson. Will humans even write code in 2040 and what would that mean for extreme heterogeneity in computing? *arXiv:1712.00676*, 12 2017. {4, 39}

[22] M. Blatt, A. Burchardt, A. Dedner, C. Engwer, J. Fahlke, B. Flemisch, C. Gersbacher, C. Gräser, F. Gruber, C. Grüninger, D. Kempf, R. Klöfkorn, T. Malkmus, S. Müthing, M. Nolte, M. Piatkowski, and O. Sander. The distributed and unified numerics environment, version 2.4. *Archive of Numerical Software*, 4(100):13–29, 2016. {4}

[23] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring. *Scientific Programming*, 20(2):129–150, 2012. {36}

[24] D. Braess. *Finite elements: theory, fast solvers, and applications in elasticity theory*, volume 166. Cambridge University Press, 2007. {9}

[25] S. Brenner and L. Scott. *The Mathematical Theory of Finite Element Methods*. Springer, New York, NY, USA, 1994. {5, 9}

[26] J. Brown. Efficient nonlinear solvers for nodal high-order finite elements in 3D. *Journal of Scientific Computing*, 45:48–63, 2010. {17}

[27] T. A. Brunner. Mulard: A multigroup thermal radiation diffusion mini-application. In *DOE Exascale Research Conference, Portland, Oregon*, 2012. {39}

[28] T. A. Brunner, T. V. Kolev, T. S. Bailey, and A. T. Till. Preserving spherical symmetry in axisymmetric coordinates for diffusion. In *International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering*, 2013. {39}

[29] J. Candy and W. Rozmus. A symplectic integration algorithm for separable Hamiltonian functions. *Journal of Computational Physics*, 92(1):230 – 256, 1991. {62}

[30] CEED: Center for Efficient Exascale Discretizations in the U.S. Department of Energy's Exascale Computing Project. `http://ceed.exascaleproject.org`. {17, 24, 31}

[31] J. Ceverny, V. Dobrev, and T. Kolev. Non-conforming mesh refinement for high-order finite elements. *SIAM Journal on Scientific Computing*, 2018. submitted. {11, 33, 39, 45}

[32] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil. VisIt: An end-user tool for visualizing and analyzing very large data. In *High Performance Visualization—Enabling Extreme-Scale Scientific Insight*, pages 357–372. CRC Press/Francis–Taylor Group, Oct. 2012. {4, 13, 18}

[33] M. L. C. Christensen, U. Villa, A. Engsig-Karup, and P. S. Vassilevski. Numerical multilevel upscaling for incompressible flow in reservoir simulation: An element-based algebraic multigrid (AMGe) approach. *SIAM Journal on Scientific Computing*, 39, 08 2016. {4, 39}

[34] P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*. North-Holland, 1978. {3, 5, 9}

[35] Conduit: Scientific data exchange library for HPC simulations. [Computer Software] `http://software.llnl.gov/conduit`, Oct. 2014. {13}

[36] J. Cottrell, T. Hughes, and Y. Bazilevs. *Isogeometric analysis: toward integration of CAD and FEA*. Wiley, 2009. {3, 12, 19}

[37] P. D'Ambra and P. S. Vassilevski. Adaptive AMG with coarsening based on compatible weighted matching. *Computing and Visualization in Science*, 16(2):59–76, Apr 2013. {39}

[38] J. De Basabe. *High-Order Finite Element Methods for Seismic Wave Propagation*. PhD thesis, University of Texas at Austin, 2009. {57}

[39] L. Demkowicz. *Computing with hp-Adaptive Finite Elements. Volume I: One and Two Dimensional Elliptic and Maxwell Problems*. Chapman Hall CRC, 2006. {3, 17}

[40] L. Demkowicz, J. Oden, W. Rachowicz, and O. Hardy. Toward a universal h-p adaptive finite element strategy, part 1. constrained approximation and data structure. *Computer Methods in Applied Mechanics and Engineering*, 77(1):79 – 112, 1989. {33}

[41] M. Deville, P. Fischer, and E. Mund. *High-order methods for incompressible fluid flow*. Cambridge University Press, 2002. {3, 9, 17, 25}

[42] V. Dobrev, P. Knupp, T. Kolev, K. Mittal, and V. Tomov. The Target-Matrix Optimization Paradigm for high-order meshes. *SIAM Journal on Scientific Computing*, 2018. in review. {38, 39, 45}

[43] V. Dobrev, P. Knupp, T. Kolev, and V. Tomov. Towards simulation-driven optimization of high-order meshes by the Target-Matrix Optimization Paradigm. In *27th International Meshing Roundtable, Oct 1-8, 2018, Albuquerque*, 2018. submitted. {38, 39, 41, 45}

[44] V. A. Dobrev, T. E. Ellis, T. V. Kolev, and R. N. Rieben. Curvilinear finite elements for Lagrangian hydrodynamics. *International Journal for Numerical Methods in Fluids*, 65(11-12):1295–1310, 2011. {39}

[45] V. A. Dobrev, T. E. Ellis, T. V. Kolev, and R. N. Rieben. High-order curvilinear finite elements for axisymmetric Lagrangian hydrodynamics. *Computers & Fluids*, 83:58–69, 2013. {39}

[46] V. A. Dobrev, T. Kolev, N. A. Peterson, and J. B. Schroder. Two-level convergence theory for multigrid reduction in time (MGRIT). *SIAM Journal on Scientific Computing*, 39:S501–S527, 5 2017. {4, 39}

[47] V. A. Dobrev, T. Kolev, and R. Rieben. High order curvilinear finite elements for elastic-plastic Lagrangian dynamics. *Journal of Computational Physics*, 257:1062–1080, 01 2014. {39, 43}

[48] V. A. Dobrev, T. V. Kolev, C. S. Lee, V. Z. Tomov, and P. S. Vassilevski. Algebraic hybridization and static condensation with application to scalable H(div) preconditioning. *SIAM Journal on Scientific Computing*, 2018. in review. {21, 23, 39, 45}

[49] V. A. Dobrev, T. V. Kolev, and R. N. Rieben. High-order curvilinear finite element methods for Lagrangian hydrodynamics. *SIAM Journal on Scientific Computing*, 34(5):B606–B641, 2012. {38, 39, 43}

[50] V. A. Dobrev, T. V. Kolev, R. N. Rieben, and V. Z. Tomov. Multi-material closure model for high-order finite element Lagrangian hydrodynamics. *International Journal for Numerical Methods in Fluids*, 82(10):689–706, 2016. {39, 44}

[51] T. Dong, V. Dobrev, T. Kolev, R. Rieben, S. Tomov, and J. Dongarra. A step towards energy efficient computing: Redesigning a hydrodynamic application on CPU-GPU. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 972–981, May 2014. {39}

[52] A. Ern and J.-L. Guermond. *Theory and practice of finite elements*, volume 159 of *Applied Mathematical Sciences*. Springer-Verlag, New York, 2004. {9}

[53] R. D. Falgout, T. A. Manteuffel, B. O'Neill, and J. B. Schroder. Multigrid reduction in time for nonlinear parabolic problems: A case study. *SIAM Journal on Scientific Computing*, 39:298–322, 5 2017. {4, 39}

[54] V. Girault and P. Raviart. *Finite Element Approximation of the Navier-Stokes Equations*, volume 749 of *Lecture Notes in Mathematics*. Springer-Verlag, New York, 1981. {9}

[55] GLVis: OpenGL finite element visualization tool. [Computer Software] `http://glvis.org`. {4, 7, 9, 18}

[56] J. Gopalakrishnan, M. Neumüller, and P. Vassilevski. The auxiliary space preconditioner for the de Rham complex. *arXiv:1710.07840*, 10 2017. {4, 39}

[57] J.-L. Guermond, B. Popov, and V. Tomov. Entropy-viscosity method for the single material Euler equations in Lagrangian frame. *Computer Methods in Applied Mechanics and Engineering*, 300:402 – 426, 2016. {39}

[58] V. Gurev, P. Pathmanathan, J.-L. Fattebert, H.-F. Wen, J. Magerlein, R. A. Gray, D. F. Richards, and J. J. Rice. A high-resolution computational model of the deforming human heart. *Biomechanics and Modeling in Mechanobiology*, 14(4):829–849, Aug 2015. {44}

[59] T. S. Haut, P. G. Maginot, V. Z. Tomov, T. A. Brunner, and T. S. Bailey. An efficient sweep-based solver for the S_N equations on high-order meshes. In *American Nuclear Society 2018 Annual Meeting, June 14-21, Philadelphia, PA*, 2018. {4}

[60] F. Hecht. New development in FreeFem++. *Journal of Numerical Mathematics*, 20(3-4):251–265, 2012. {4}

[61] V. Heuveline and F. Schieweck. H1-interpolation on quadrilateral and hexahedral meshes with hanging nodes. *Computing*, 80(3):203–220, Jul 2007. {33}

[62] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396, 2005. {59}

[63] C. Hofer, U. Langer, M. Neumuller, and I. Toulopoulos. Time-multipatch discontinuous Galerkin space-time isogeometric analysis of parabolic evolution problems. *RICAM-Report 2017-26*, 2017. {4, 39}

[64] M. Holec, J. Limpouch, R. Liska, and S. Weber. High-order discontinuous Galerkin nonlocal transport and energy equations scheme for radiation hydrodynamics. *International Journal for Numerical Methods in Fluids*, 83(10):779–797, 2016. {39, 45}

[65] M. Holst. Adaptive numerical treatment of elliptic systems on manifolds. *Advances in Computational Mathematics*, 15:139–191, 2001. {4}

[66] R. D. Hornung and J. A. Keasler. The RAJA Portability Layer: Overview and Status. Technical Report LLNL-TR-661403, LLNL, Sept. 2014. {31}

[67] hypre: High performance preconditioners. `http://www.llnl.gov/casc/hypre`, 2018. {4, 27}

[68] D. A. Ibanez, E. S. Seol, C. W. Smith, and M. S. Shephard. PUMI: Parallel Unstructured Mesh Infrastructure. *ACM Transactions on Mathematical Software*, 42(3):17:1–17:28, 2016. {32}

[69] C. Johnson. *Numerical solution of partial differential equations by the finite element method*. Cambridge University Press, 1987. {9}

[70] D. Z. Kalchev, C. S. Lee, U. Villa, Y. Efendiev, and P. S. Vassilevski. Upscaling of mixed finite element discretization problems by the spectral AMGe method. *SIAM Journal on Scientific Computing*, 38:A2912–A2933, 09 2016. {39}

[71] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998. {4, 12}

[72] B. S. Kirk, J. W. Peterson, R. H. Stogner, and G. F. Carey. `libMesh`: A C++ library for parallel adaptive mesh refinement/coarsening simulations. *Engineering with Computers*, 22(3–4):237–254, 2006. {4}

[73] P. Knupp. Introducing the target-matrix paradigm for mesh optimization by node movement. *Engineering with Computers*, 28(4):419–429, 2012. {38}

[74] T. Kolev and P. Vassilevski. Parallel auxiliary space AMG for H(curl) problems. *J. Comput. Math.*, 27:604–623, 2009. Special issue on Adaptive and Multilevel Methods in Electromagnetics. UCRL-JRNL-237306. {29}

[75] T. V. Kolev and P. S. Vassilevski. Parallel auxiliary space AMG solver for H(div) problems. *SIAM Journal on Scientific Computing*, 34(6):A3079–A3098, 2012. {39}

[76] T. V. Kolev, J. Xu, and Y. Zhu. Multilevel preconditioners for reaction-diffusion problems with discontinuous coefficients. *Journal of Scientific Computing*, 67(1):324–350, Apr 2016. {39}

[77] M. Larsen, J. Ahrens, U. Ayachit, E. Brugger, H. Childs, B. Geveci, and C. Harrison. The ALPINE in situ infrastructure: Ascending from the ashes of strawman. In *ISAV 2017: In Situ Infrastructures for Enabling Extreme-scale Analysis and Visualization*, pages 42–46, 11 2017. {4, 39}

[78] I. V. Lashuk and P. S. Vassilevski. Element agglomeration coarse Raviart–Thomas spaces with improved approximation properties. *Numerical Linear Algebra with Applications*, 19(2):414–426, 2012. {39}

[79] I. V. Lashuk and P. S. Vassilevski. The construction of coarse de Rham complexes with improved approximation properties. *Computational Methods in Applied Mathematics*, 14(2):257–303, 2014. {44}

[80] R. Li and Y. Saad. Low-rank correction methods for algebraic domain decomposition preconditioners. *SIAM Journal on Matrix Analysis and Applications*, 38, 05 2015. {4, 39}

[81] R. Li, Y. Xi, L. Erlandson, and Y. Saad. The eigenvalues slicing library (EVSL): Algorithms, implementation, and software. *arXiv:1802.05215*, 02 2018. {4, 39}

[82] X.-J. Luo, M. Shephard, L.-Q. Lee, L. Ge, and C. Ng. Moving curved mesh adaptation for higher-order finite element simulations. *Engineering with Computers*, 27(1):41–50, 2011. {38}

[83] T. A. Manteuffel, L. N. Olson, J. B. Schroder, and B. S. Southworth. A root-node based algebraic multigrid method. *SIAM Journal on Scientific Computing*, 39:723–756, 5 2017. {4, 39}

[84] O. Marques, A. Druinsky, X. S. Li, A. T. Barker, P. Vassilevski, and D. Kalchev. Tuning the coarse space construction in a spectral AMG solver. *Procedia Computer Science*, 80:212 – 221, 2016. International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA. {39}

[85] D. A. May, J. Brown, and L. LePourhiet. A scalable, matrix-free multigrid preconditioner for finite element discretizations of heterogeneous Stokes flow. *Computer Methods in Applied Mechanics and Engineering*, 290:496–523, 2015. {31}

[86] A. Mazuyer, P. Cupillard, R. Giot, M. Conin, Y. Leroy, and P. Thore. Stress estimation in reservoirs using an integrated inverse method. *Computers & Geosciences*, 114:30 – 40, 2018. {4, 39, 45}

[87] METIS: Serial graph partitioning and fill-reducing matrix ordering. [Computer Software] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview. {4, 12}

[88] MFEM: Modular finite element methods. http://mfem.org, 2018. {3}

[89] P. Monk. *Finite Element Methods for Maxwell's Equations*. Numerical Mathematics and Scientific Computation. Oxford University Press, Oxford, UK, 2003. {9}

[90] A. Napov and Y. Notay. Algebraic multigrid for moderate order finite elements. *SIAM Journal on Scientific Computing*, 36(4):A1678–A707, 2014. {31}

[91] J. C. Nedelec. Mixed finite elements in $\mathbb{R}^3$. *Numerische Mathematik*, 35(3):315–341, Sept. 1980. {17}

[92] OCCA: Open Concurrent Compute Abstraction. http://libocca.org. {31}

[93] L. Olson. Algebraic multigrid preconditioning of high-order spectral elements for elliptic problems on a simplicial mesh. *SIAM Journal on Scientific Computing*, 29(5):2189–2209, 2007. {31}

[94] S. A. Orszag. Spectral methods for problems in complex geometries. *Journal of Computational Physics*, 37(1):70 – 92, 1980. {25}

[95] S. Osborn, P. S. Vassilevski, and U. Villa. A multilevel, hierarchical sampling technique for spatially correlated random fields. *SIAM Journal on Scientific Computing*, 39, 03 2017. {4, 39}

[96] S. Osborn, P. Zulian, T. Benson, U. Villa, R. Krause, and P. S. Vassilevski. Scalable hierarchical PDE sampler for generating spatially correlated random fields using non-matching meshes. *Numerical Linear Algebra with Applications*, 25:e2146, 12 2017. {4, 39}

[97] D. Osei-Kuffuor, R. Li, and Y. Saad. Matrix reordering using multilevel graph coarsening for ILU preconditioning. *SIAM Journal on Scientific Computing*, 37(1):A391–A419, 2015. {39}

[98] ParElag: Element agglomeration algebraic multigrid and upscaling. https://www.github.com/llnl/parelag, 2018. {44}

[99] J. E. Pasciak and P. S. Vassilevski. Exact de Rham sequences of spaces defined on macro-elements in two and three spatial dimensions. *SIAM Journal on Scientific Computing*, 30(5):2427–2446, 2008. {44}

[100] M. Reberol and B. Lévy. Low-order continuous finite element spaces on hybrid non-conforming hexahedral-tetrahedral meshes. *CoRR*, abs/1605.02626, 2016. {39}

[101] M. Reberol and B. Lévy. Computing the distance between two finite element solutions defined on different 3D meshes on a GPU. *SIAM Journal on Scientific Computing*, 40(1):C131–C155, 2018. {4, 39, 45}

[102] S. Rhebergen and S. N. Wells. Preconditioning of a hybridized discontinuous Galerkin finite element method for the Stokes equations. *arXiv:1801.04707*, 01 2018. {4, 39}

[103] O. Sahni, X. Luo, K. Jansen, and M. Shephard. Curved boundary layer meshing for adaptive viscous flow simulations. *Finite Elements in Analysis and Design*, 46(1):132–139, 2010. {38}

[104] A. Sánchez-Villar and M. Merino. Advances in wave-plasma modelling in ecr thrusters. In *2018 Space Propulsion Conference, May 14-18, Seville, Spain*, 2018. {4}

[105] T. Schönfeld. Adaptive mesh refinement methods for three-dimensional inviscid flow problems. *International Journal of Computational Fluid Dynamics*, 4(3-4):363–391, 1995. {33}

[106] Shiraiwa, S., Wright, J. C., Bonoli, P. T., Kolev, T., and Stowell, M. RF wave simulation for cold edge plasmas using the MFEM library. In *22 Topical Conference on Radio-Frequency Power in Plasmas*, volume 157, page 03048, 2017. {4, 39}

[107] P. Solin, J. Cerveny, and I. Dolezel. Arbitrary-level hanging nodes and automatic adaptivity in the hp-FEM. *Mathematics and Computers in Simulation*, 77:117–132, 2008. {4}

[108] P. Solin, K. Segeth, and I. Dolezel. *Higher-Order Finite Element Methods*. Chapman Hall CRC, 2002. {3, 9, 17}

[109] G. Strang and G. J. Fix. *An analysis of the finite element method*, volume 212. Prentice-Hall Englewood Cliffs, NJ, 1973. {9}

[110] H. Sundar, G. Stadler, and G. Biros. Comparison of multigrid algorithms fro high-order continuous finite element discretizations. *Numerical Linear Algebra with Applications*, 22(4):664–680, 2015. {31}

[111] P. S. Vassilevski and Y. U. Meier. Reducing communication in algebraic multigrid using additive variants. *Numerical Linear Algebra with Applications*, 21(2):275–296, 2014. {39}

[112] P. S. Vassilevski and U. Villa. A block-diagonal algebraic multigrid preconditioner for the Brinkman problem. *SIAM Journal on Scientific Computing*, 35(5):S3–S17, 2013. {39}

[113] P. S. Vassilevski and U. Villa. A mixed formulation for the Brinkman problem. *SIAM Journal on Numerical Analysis*, 52(1):258–281, 2014. {39}

[114] VisIt: A distributed, parallel visualization and analysis tool. [Computer Software] `http://visit.llnl.gov`. {4, 13, 18}

[115] D. A. White, M. Stowell, and D. A. Tortorelli. Toplogical optimization of structures using Fourier representations. *Structural and Multidisciplinary Optimization*, pages 1–16, 04 2018. {4, 39, 45}

[116] J. Wright and S. Shiraiwa. Antenna to core: A new approach to RF modelling. In *22 Topical Conference on Radio-Frequency Power in Plasmas*, volume 157, 2017. {4, 39}

[117] J. S. Yeom, J. J. Thiagarajan, A. Bhatele, G. Bronevetsky, and T. Kolev. Data-driven performance modeling of linear solvers for sparse matrices. In *2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 32–42, Nov 2016. {39}

[118] S. Zampini, P. S. Vassilevski, V. Dobrev, and T. Kolev. Balancing domain decomposition by constraints algorithms for curl-conforming spaces of arbitrary order. In *Domain Decomposition Methods in Science and Engineering XXIV*, 2017. {4, 39}

[119] O. C. Zienkiewicz, R. L. Taylor, and J. Z. Zhu. *The Finite Element Method: Its Basis and Fundamentals*. Butterworth-Heinemann, 2005. {9}

Appendix A Bilinear Form Integrators

MFEM includes numerous types of bilinear form integrators which are listed here to provide a quick reference for developers. For more information regarding the appropriate trial and test spaces, supported coefficient types, etc. see the MFEM online documentation at `http://mfem.org/bilininteg/`. While we have tried to include support for a wide variety of operators, this list can be easily extended by adding customized bilinear form integrators which would inherit from the `BilinearFormIntegrator` abstract base class.

A.1 Scalar Field Operators

These operators, described in Table 2, require scalar-valued trial spaces. Many of these operators will work with either H^1 or L^2 basis functions but some, those that require a gradient operator, should be used with H^1.

Class Name	**Operator**	**Continuous Operator**
Square Operators		
`MassIntegrator`	$(\lambda u, v)$	λu
`DiffusionIntegrator`	$(\lambda \nabla u, \nabla v)$	$-\nabla \cdot (\lambda \nabla u)$
General Operators		
`MixedScalarMassIntegrator`	$(\lambda u, v)$	λu
`MixedScalarWeakDivergenceIntegrator`	$(-\vec{\lambda} u, \nabla v)$	$\nabla \cdot (\vec{\lambda} u)$
`MixedScalarWeakDerivativeIntegrator`	$(-\lambda u, \frac{dv}{dx})$	$\frac{d}{dx}(\lambda u)$
`MixedScalarWeakCurlIntegrator`	$(\lambda u, \nabla \times \vec{v})$	$\nabla \times (\lambda\, u\, \hat{z})$
`MixedVectorProductIntegrator`	$(\vec{\lambda} u, \vec{v})$	$\vec{\lambda} u$
`MixedScalarWeakCrossProductIntegrator`	$(\vec{\lambda} u\, \hat{z}, \vec{v})$	$\vec{\lambda} \times\, \hat{z}\, u$
`MixedScalarWeakGradientIntegrator`	$(-\lambda u, \nabla \cdot \vec{v})$	$\nabla(\lambda u)$
`MixedDirectionalDerivativeIntegrator`	$(\vec{\lambda} \cdot \nabla u, v)$	$\vec{\lambda} \cdot \nabla u$
`MixedScalarCrossGradIntegrator`	$(\vec{\lambda} \times \nabla u, v)$	$\vec{\lambda} \times \nabla u$
`MixedScalarDerivativeIntegrator`	$(\lambda \frac{du}{dx}, v)$	$\lambda \frac{du}{dx}$
`MixedGradGradIntegrator`	$(\lambda \nabla u, \nabla v)$	$-\nabla \cdot (\lambda \nabla u)$
`MixedCrossGradGradIntegrator`	$(\vec{\lambda} \times \nabla u, \nabla v)$	$-\nabla \cdot (\vec{\lambda} \times \nabla u)$
`MixedVectorGradientIntegrator`	$(\lambda \nabla u, \vec{v})$	$\lambda \nabla u$
`MixedCrossGradIntegrator`	$(\vec{\lambda} \times \nabla u, \vec{v})$	$\vec{\lambda} \times \nabla u$
`MixedCrossGradCurlIntegrator`	$(\vec{\lambda} \times \nabla u, \nabla \times \vec{v})$	$\nabla \times (\vec{\lambda} \times \nabla u)$
`MixedGradDivIntegrator`	$(\vec{\lambda} \cdot \nabla u, \nabla \cdot \vec{v})$	$-\nabla(\vec{\lambda} \cdot \nabla u)$
Special Purpose Operators		
`DerivativeIntegrator`	$(\lambda \frac{\partial u}{\partial x_i}, v)$	$\lambda \frac{\partial u}{\partial x_i}$
`ConvectionIntegrator`	$(\vec{\lambda} \cdot \nabla u, v)$	$\vec{\lambda} \cdot \nabla u$
`GroupConvectionIntegrator`	$(\alpha \vec{\lambda} \cdot \nabla u, v)$	$\alpha \vec{\lambda} \cdot \nabla u$
`BoundaryMassIntegrator`	$(\lambda\, u, v)$	$\lambda\, u$

Table 2: Scalar field integrators.

A.2 Vector Finite Element Operators

These operators, described in Table 3, require vector-valued basis functions in the trial space. Many of these operators will work with either Nedelec or Raviart-Thomas finite elements.

Class Name	Operator	Continuous Operator
Square Operators		
VectorFEMassIntegrator	$(\lambda\vec{u}, \vec{v})$	$\lambda\vec{u}$
CurlCurlIntegrator	$(\lambda\nabla\times\vec{u}, \nabla\times\vec{v})$	$\nabla\times(\lambda\nabla\times\vec{u})$
DivDivIntegrator	$(\lambda\nabla\cdot\vec{u}, \nabla\cdot\vec{v})$	$-\nabla(\lambda\nabla\cdot\vec{u})$
General Operators		
MixedDotProductIntegrator	$(\vec{\lambda}\cdot\vec{u}, v)$	$\vec{\lambda}\cdot\vec{u}$
MixedScalarCrossProductIntegrator	$(\vec{\lambda}\times\vec{u}, v)$	$\vec{\lambda}\times\vec{u}$
MixedVectorWeakDivergenceIntegrator	$(-\lambda\vec{u}, \nabla v)$	$\nabla\cdot(\lambda\vec{u})$
MixedWeakDivCrossIntegrator	$(-\vec{\lambda}\times\vec{u}, \nabla v)$	$\nabla\cdot(\vec{\lambda}\times\vec{u})$
MixedVectorMassIntegrator	$(\lambda\vec{u}, \vec{v})$	$\lambda\vec{u}$
MixedCrossProductIntegrator	$(\vec{\lambda}\times\vec{u}, \vec{v})$	$\vec{\lambda}\times\vec{u}$
MixedVectorWeakCurlIntegrator	$(\lambda\vec{u}, \nabla\times\vec{v})$	$\nabla\times(\lambda\vec{u})$
MixedWeakCurlCrossIntegrator	$(\vec{\lambda}\times\vec{u}, \nabla\times\vec{v})$	$\nabla\times(\vec{\lambda}\times\vec{u})$
MixedScalarWeakCurlCrossIntegrator	$(\vec{\lambda}\times\vec{u}, \nabla\times\vec{v})$	$\nabla\times(\vec{\lambda}\times\vec{u})$
MixedWeakGradDotIntegrator	$(-\vec{\lambda}\cdot\vec{u}, \nabla\cdot\vec{v})$	$\nabla(\vec{\lambda}\cdot\vec{u})$
MixedScalarCurlIntegrator	$(\lambda\nabla\times\vec{u}, v)$	$\lambda\nabla\times\vec{u}$
MixedCrossCurlGradIntegrator	$(\vec{\lambda}\times\nabla\times\vec{u}, \nabla v)$	$-\nabla\cdot(\vec{\lambda}\times\nabla\times\vec{u})$
MixedVectorCurlIntegrator	$(\lambda\nabla\times\vec{u}, \vec{v})$	$\lambda\nabla\times\vec{u}$
MixedCrossCurlIntegrator	$(\vec{\lambda}\times\nabla\times\vec{u}, \vec{v})$	$\vec{\lambda}\times\nabla\times\vec{u}$
MixedScalarCrossCurlIntegrator	$(\vec{\lambda}\times\hat{z}\,\nabla\times\vec{u}, \vec{v})$	$\vec{\lambda}\times\hat{z}\,\nabla\times\vec{u}$
MixedCurlCurlIntegrator	$(\lambda\nabla\times\vec{u}, \nabla\times\vec{v})$	$\nabla\times(\lambda\nabla\times\vec{u})$
MixedCrossCurlCurlIntegrator	$(\vec{\lambda}\times\nabla\times\vec{u}, \nabla\times\vec{v})$	$\nabla\times(\vec{\lambda}\times\nabla\times\vec{u})$
MixedScalarDivergenceIntegrator	$(\lambda\nabla\cdot\vec{u}, v)$	$\lambda\nabla\cdot\vec{u}$
MixedDivGradIntegrator	$(\vec{\lambda}\nabla\cdot\vec{u}, \nabla v)$	$-\nabla\cdot(\vec{\lambda}\nabla\cdot\vec{u})$
MixedVectorDivergenceIntegrator	$(\vec{\lambda}\nabla\cdot\vec{u}, \vec{v})$	$\vec{\lambda}\nabla\cdot\vec{u}$
Special Purpose Operators		
VectorFEDivergenceIntegrator	$(\lambda\nabla\cdot\vec{u}, v)$	$\lambda\nabla\cdot\vec{u}$
VectorFEWeakDivergenceIntegrator	$(-\lambda\vec{u}, \nabla v)$	$\nabla\cdot(\lambda\vec{u})$
VectorFECurlIntegrator	$(\lambda\nabla\times\vec{u}, \vec{v})$ or $(\lambda\vec{u}, \nabla\times\vec{v})$	$\lambda\nabla\times\vec{u}$ or $\nabla\times(\lambda\vec{u})$

Table 3: Vector field integrators.

A.3 Nodal Vector Field Operators

These operators, described in Table 4, require vector-valued basis functions constructed by using multiple copies of scalar fields. Specifically, the trial basis functions used with these operators

should be either $(H^1)^d$ or $(L^2)^d$. In each of these integrators, the scalar basis function index increments most quickly followed by the vector index. This leads to local element matrices which have a block structure.

Class Name	Operator	Continuous Operator
Square Operators		
VectorMassIntegrator	$(\lambda\vec{u}, \vec{v})$	$\lambda\vec{u}$
VectorCurlCurlIntegrator	$(\lambda\nabla\times\vec{u}, \nabla\times\vec{v})$	$\nabla\times(\lambda\nabla\times\vec{u})$
VectorDiffusionIntegrator	$(\lambda\nabla u_i, \nabla v_i)$	$-\nabla\cdot(\lambda\nabla u_i)$
ElasticityIntegrator	$(c_{ikjl}\nabla u_j, \nabla v_i)$	$-\nabla_k\cdot(c_{ikjl}\nabla_l u_j)$
General Operators		
VectorDivergenceIntegrator	$(\lambda\nabla\cdot\vec{u}, v)$	$\lambda\nabla\cdot\vec{u}$

Table 4: Nodal vector field integrators.

A.4 Discontinuous Galerkin Operators

The notation used for the Discontinuous Galerkin (DG) operators shown in Table 5 may be unfamiliar to some readers so we provide a few definitions in Table 6. The details of the operator implemented in the DGElasticityIntegrator can be found in [38].

Class Name	Operator
DGTraceIntegrator	$\alpha\left\langle\rho_u(\vec{u}\cdot\hat{n})\{v\},[w]\right\rangle+\beta\left\langle\rho_u\lvert\vec{u}\cdot\hat{n}\rvert[v],[w]\right\rangle$
DGDiffusionIntegrator	$-\left\langle\{Q\nabla u\cdot\hat{n}\},[v]\right\rangle+\sigma\left\langle[u],\{Q\nabla v\cdot\hat{n}\}\right\rangle+\kappa\left\langle\{h^{-1}Q\}[u],[v]\right\rangle$
DGElasticityIntegrator	$-\left\langle\{\sigma(u)\cdot\vec{n}\},[v]\right\rangle+\alpha\left\langle\{\sigma(v)\cdot\vec{n}\},[u]\right\rangle+\kappa\left\langle h^{-1}\{\lambda+2\mu\}[u],[v]\right\rangle$
TraceJumpIntegrator	$\langle v,[w]\rangle$
NormalTraceJumpIntegrator	$\langle v,[\vec{w}\cdot\vec{n}]\rangle$

Table 5: Discontinuous Galerkin operators.

Description	Notation
Inner Product	$\langle u,v\rangle=\int_F u\cdot v$ for face F
Average	$\{u\}=\frac{1}{2}\left(u\rvert_{K_1}+u\rvert_{K_2}\right)$ for elements K_1 and K_2
Jump	$[u]=u\rvert_{K_1}-u\rvert_{K_2}$ for elements K_1 and K_2

Table 6: Discontinuous Galerkin definitions.

A.5 Special Purpose Integrators

These "integrators", described in Table 7, do not actually perform integrations, they merely alter the results of other integrators. As such, they provide a convenient and easy way to reuse existing integrators in special situations rather than needing to re-implement their functionality.

Class Name	**Description**
Square Operators	
`LumpedIntegrator`	Returns a diagonal local matrix where each entry is the sum of the corresponding row of a local matrix computed by another `BilinearFormIntegrator`
`InverseIntegrator`	Returns the inverse of the local matrix computed by another `BilinearFormIntegrator` which produces a square local matrix
`SumIntegrator`	Returns the sum of a series of integrators with compatible dimensions
General Operators	
`TransposeIntegrator`	Returns the transpose of the local matrix computed by another `BilinearFormIntegrator`

Table 7: Special purpose integrators.

Appendix B Ordinary Differential Equation Integrators

Ordinary Differential Equation (ODE) integrators are not an integral part of a finite element library, but we include them in MFEM because they can be an important component of many finite element applications. In general, ODEs of any order can be written as systems of first order ODEs, therefore, solution methods focus on the solution equations of the type:

$$\frac{d\vec{y}}{dt} = f(t, \vec{y}) \tag{30}$$

MFEM comes pre-packaged with a handful of explicit and implicit Runge-Kutta methods (as well as some specialized methods, see below). Easy-to-use interfaces to the time integrators in the SUNDIALS [62] and PETSc [15] suites are also provided.

B.1 Explicit Runge-Kutta Methods

Explicit Runge-Kutta methods can be written in the form:

$$y_{n+1} = y_n + h\sum_{i=1}^{s} b_i k_i$$

where

$$\begin{aligned}
k_1 &= f(t_n, y_n) \\
k_2 &= f(t_n + c_2 h, y_n + h(a_{21}k_1)) \\
k_3 &= f(t_n + c_3 h, y_n + h(a_{31}k_1 + a_{32}k_2)) \\
&\vdots \\
k_s &= f(t_n + c_s h, y_n + h(a_{s1}k_1 + a_{s2}k_2 + \cdots + a_{s,s-1}k_{s-1}))
\end{aligned}$$

The number of stages and the choice of coefficients determines the particular method being employed. In the general explicit case the coefficients are often displayed in a Butcher tableau such as shown in Table 8.

$$\begin{array}{c|ccccc}
0 & & & & & \\
c_2 & a_{21} & & & & \\
c_3 & a_{31} & a_{32} & & & \\
\vdots & \vdots & \vdots & \ddots & & \\
c_s & a_{s1} & a_{s2} & \dots & a_{s,s-1} & \\
\hline
 & b_1 & b_2 & \dots & b_{s-1} & b_s
\end{array}$$

Table 8: General explicit Butcher tableau.

The MFEM class `ExplicitRKSolver` can be used, along with user defined a_{ij}, b_j, and c_i coefficients, to perform an arbitrary explicit Runge-Kutta iteration. Other explicit Runge-Kutta methods provided by MFEM are shown in Table 9. Butcher tableau for many of these methods are shown in Tables 10–13.

Method	Order	Stages
`ForwardEulerSolver`	1	1
`RK2Solver`	2	2
`RK3SSPSolver`	3	3
`RK4Solver`	4	4
`RK6Solver`	6	8
`RK8Solver`	8	12

Table 9: Explicit Runge-Kutta methods and their numbers of stages and orders.

$$
\begin{array}{c|c}
0 & \\
\hline
 & 1
\end{array}
$$

Table 10: Butcher tableau for `ForwardEulerSolver`.

$$
\begin{array}{c|cc}
0 & & \\
a & a & \\
\hline
 & (1-\frac{1}{2a}) & \frac{1}{2a}
\end{array}
$$

Table 11: Butcher tableau for `RK2Solver` parameterized with user-defined a. Common choices would be: the midpoint method with $a = \frac{1}{2}$, Heun's method with $a = 1$, and a minimum truncation error method with $a = 2/3$.

$$
\begin{array}{c|ccc}
0 & & & \\
1 & 1 & & \\
1/2 & 1/4 & 1/4 & \\
\hline
 & 1/6 & 1/6 & 2/3
\end{array}
$$

Table 12: Butcher Tableau for `RK3SSPSolver`, a third order, strong stability preserving (SSP) method.

$$
\begin{array}{c|cccc}
0 & & & & \\
1/2 & 1/2 & & & \\
1/2 & 0 & 1/2 & & \\
1 & 0 & 0 & 1 & \\
\hline
 & 1/6 & 1/3 & 1/3 & 1/6
\end{array}
$$

Table 13: Butcher Tableau for `RK4Solver` which implements the classical forth order Runge-Kutta algorithm.

B.2 Implicit Methods

Implicit Runge-Kutta methods can be written in the form:

$$
y_{n+1} = y_n + h \sum_{i=1}^{s} b_i k_i
$$

where

$$
k_i = f(t_n + c_i h, y_n + h \sum_{j=1}^{s} a_{ij} k_j)
$$

The available implicit Runge-Kutta methods and their stability are shown in Table 14. Butcher tableau for each of these methods are shown in Tables 15–19. For more information about the SDIRK methods see [4].

Method	**A-Stable**	**L-Stable**
`BackwardEulerSolver`	✓	✓
`ImplicitMidpointMethod`	✓	
`SDIRK23Solver(option)`	for `option` ≥ 1	for `option` ≥ 2
`SDIRK34Solver`	✓	
`SDIRK33Solver`	✓	✓

Table 14: Stability of the available implicit Runge-Kutta methods.

1	1
	1

Table 15: Butcher tableau for `BackwardEulerSolver`.

1/2	1/2
	1

Table 16: Butcher tableau for `ImplicitMidpointSolver`.

γ	γ	
$(1-\gamma)$	$(1-2\gamma)$	γ
	$1/2$	$1/2$

Table 17: Butcher tableau for `SDIRK23Solver(option)` where γ depends upon the value of `option`: $0 \rightarrow \gamma = \frac{1}{2} - \frac{\sqrt{3}}{6}$, $1 \rightarrow \gamma = \frac{1}{2} + \frac{\sqrt{3}}{6}$, $2 \rightarrow \gamma = 1 - \frac{\sqrt{2}}{2}$, $3 \rightarrow \gamma = 1 + \frac{\sqrt{2}}{2}$.

a	a		
$1/2$	$(1/2-a)$	a	
$(1-a)$	$2a$	$(1-4a)$	a
	b	$(1-2b)$	b

Table 18: Butcher tableau for `SDIRK34Solver` where $a = \frac{1}{2} + \frac{\cos(\pi/18)}{\sqrt{3}}$ and $b = \frac{1}{6(2a-1)^2}$.

a	a		
c	$(c-a)$	a	
1	b	$(1-a-b)$	a
	b	$(1-a-b)$	a

Table 19: Butcher tableau for `SDIRK33Solver` where a is the root of $x^3 - 3x + \frac{3}{2}x - \frac{1}{6} = 0$ which lies within the interval $(\frac{1}{6}, \frac{1}{2})$, $b = (6a - (8 - \sqrt{58}))((8 + \sqrt{58}) - 6a)/24$ and $c = (1 + a)/2$.

B.3 Time-Dependent Operator

The implementation of Runge-Kutta methods requires a means to evaluate the ODE function $f(t, \vec{y})$ appearing in (30). In MFEM these functions are implemented as `TimeDependentOperator` objects. Let $\vec{k} \equiv f(t, \vec{y})$, a `TimeDependentOperator` must find $\vec{k}$ such that $F(\vec{y}, \vec{k}, t) = G(\vec{y}, t)$ where F and G represent the *implicit* and *explicit* parts of the operator $f(t, \vec{y})$, respectively.

Two special cases of `TimeDependentOperator` objects exist: **explicit** when $F(\vec{y}, \vec{k}, t) = \vec{k}$, and **homogeneous** when $G(\vec{y}, t) = 0$. In general, such operators are called **implicit**.

MFEM's `TimeDependentOperator` objects contain several methods, but two are of particular importance for Runge-Kutta methods. The `TimeDependentOperator::Mult(y,k)` method finds the vector $\vec{k}$ which satisfies

$$F(\vec{y}, \vec{k}, t) = G(\vec{y}, t) \qquad \text{or equivalently} \qquad \vec{k} = f(t, \vec{y})$$

and is needed by explicit Runge-Kutta methods. The `TimeDependentOperator::ImplicitSolve(`Δt`,y,k)` method finds the vector $\vec{k}$ which satisfies

$$F(\vec{y} + \Delta t\, \vec{k}, \vec{k}, t) = G(\vec{y} + \Delta t\, \vec{k}, t) \qquad \text{or equivalently} \qquad \vec{k} = f(t, \vec{y} + \Delta t\, \vec{k})$$

and is only needed by implicit Runge-Kutta methods.

B.4 Symplectic Integration Solvers

An important class of problems in physics, including wave phenomena, can be described by second order ODEs which can be derived from a Hamiltonian operator

$$H(\vec{q}, \vec{p}, t) \quad = \quad T(\vec{p}) + V(\vec{q}, t).$$

Which leads to Hamilton's equations

$$\begin{aligned} \frac{d\vec{p}}{dt} &= -\frac{\partial H}{\partial \vec{q}} \equiv F(\vec{q}, t)\,, \\ \frac{d\vec{q}}{dt} &= \frac{\partial H}{\partial \vec{p}} \equiv P(\vec{p}). \end{aligned}$$

Such ODEs could be reduced to a first order system of equations as in Equation 30 and solved with a Runge-Kutta method. However, such a procedure would fail to preserve quantities, such as total energy, which should be conserved during the evolution of the system. Symplectic Integration algorithms [29] were designed to preserve such quantities.

MFEM contains implementations of the four symplectic integration algorithms described in [29] which perform numerical integration of Hamiltonian systems with orders 1–4. These integrators make use of the same `TimeDependentOperator` mentioned in Subsection B.3 to define the function $F(\vec{q}, t)$. On the other hand, the operator $P(\vec{p})$ is a standard MFEM `Operator` object.