# MFEM

## A modular finite element methods library

Anderson, Robert; Andrej, Julian; Barker, Andrew; Bramwell, Jamie; Camier, Jean Sylvain; Cerveny, Jakub; Dobrev, Veselin; Dudouit, Yohann; Akkerman, Ido; More Authors

**DOI**
10.1016/j.camwa.2020.06.009

**Publication date**
2021

**Document Version**
Accepted author manuscript

**Published in**
Computers and Mathematics with Applications

Computers and Mathematics with Applications xxx (xxxx) xxx

# MFEM: A modular finite element methods library☆

Robert Anderson [a], Julian Andrej [a], Andrew Barker [a], Jamie Bramwell [a], Jean-Sylvain Camier [a], Jakub Cerveny [a], Veselin Dobrev [a], Yohann Dudouit [a], Aaron Fisher [a], Tzanio Kolev [a,*], Will Pazner [a], Mark Stowell [a], Vladimir Tomov [a], Ido Akkerman [b], Johann Dahm [c], David Medina [d], Stefano Zampini [e]

[a] *Lawrence Livermore National Laboratory, Livermore, USA*
[b] *Delft University of Technology, The Netherlands*
[c] *Vulcan Inc., Seattle, USA*
[d] *Occalytics LLC, NY, USA*
[e] *King Abdullah University of Science and Technology, Saudi Arabia*

ARTICLE INFO

*Article history:*
Available online xxxx

*Keywords:*
Finite element methods
Numerical PDEs
Open-source scientific software
High-order methods
Matrix-free algorithms
High-performance computing

ABSTRACT

MFEM is an open-source, 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. MFEM's goal is to provide application scientists with access to cutting-edge algorithms for high-order finite element meshing, discretizations and linear solvers, while enabling researchers to quickly and easily develop and test new algorithms in very general, fully unstructured, high-order, parallel and GPU-accelerated 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.

Published by Elsevier Ltd.

## 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 [1–8].

*MFEM* is an open-source, lightweight, modular and scalable software library for finite elements, featuring 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 [9]. The MFEM project performs mathematical research and software development 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 source code is freely available via Spack, OpenHPC, and GitHub, https://github.com/mfem, under the open source BSD license.

☆ This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344, LLNL-JRNL-795849.

* Corresponding author.
*E-mail addresses:* dobrev1@llnl.gov (V. Dobrev), tzanio@llnl.gov (T. Kolev).

https://doi.org/10.1016/j.camwa.2020.06.009
0898-1221/Published by Elsevier Ltd.

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, while retaining relatively small and lightweight code base (see Sections 6 and 8). MFEM's main capabilities and their corresponding sections in the paper are outlined in the following text.

MFEM is distinguished from other finite element packages, such as deal.II [10], FEniCS [11], DUNE [12], FreeFem++ [13], Hermes [14], libMesh [15], FETK [16], NGSolve [17], etc., by a unique combination of features, including its massively parallel scalability, HPC efficiency, support for arbitrary high-order finite elements, generality in mesh type and discretization methods, support for GPU acceleration 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. The wide applicability of the library is illustrated by the fact that in recent years 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. A comprehensive list of publications making use of MFEM can be found at https://mfem.org/publications.

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 [18]: $H^1$-conforming, discontinuous ($L^2$), $H(\mathrm{div})$-conforming, $H(\mathrm{curl})$-conforming and NURBS finite element spaces in 2D and 3D (Section 4.3), as well as many bilinear, linear, and nonlinear forms defined on them, including linear operators such as gradient, curl, and embeddings between these spaces. It enables fast prototyping of various finite element discretizations including: Galerkin methods, mixed finite elements, discontinuous Galerkin (DG), isogeometric analysis, hybridization, and discontinuous Petrov–Galerkin approaches (Section 5.1).

MFEM contains classes for dealing with a wide range of mesh types: triangular, quadrilateral, tetrahedral, hexahedral, prismatic as well as mixed meshes, surface meshes 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 a variety of explicit and implicit Runge–Kutta time integrators are also available.

MFEM supports Message Passing Interface (MPI)-based parallelism throughout the library and can readily be used as a scalable unstructured finite element problem generator (Section 6.1). Starting with version 4.0, MFEM offers initial support for GPU acceleration, and programming models, such as CUDA, OCCA, RAJA and OpenMP (Section 6.3). 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 (Section 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 (Section 6.2). Both the serial and parallel versions can make use of high-performance, *partial assembly* kernels, described in further detail in Section 6.3.

Comprehensive support for a number of external numerical libraries, e.g., PETSc [19], SuperLU [20], STRUMPACK [21], SuiteSparse [22], SUNDIALS [23], and PUMI [24] 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 [25], as well as with ParaView and the VisIt [26,27] visualization and analysis tool (Section 4.5).

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++, using a limited subset of the language. This reduces the entry barrier for new contributors and makes it easy to build the library on early-access HPC systems with vendor compilers that may not be mature. The serial version of MFEM has no required external dependencies and is straightforward to build on Linux, Mac, and Windows. The MPI-parallel version requires only two third-party libraries (*hypre* [28] and METIS [29,30]) and is easy to build with an MPI compiler.

MFEM's development grew out of a need for robust, flexible, and efficient simulation algorithms for physics and engineering applications at Lawrence Livermore National Laboratory (LLNL). The initial open-source release of the library was in 2010, followed by version 1.2 in 2011 that added MPI parallelism. Versions 2.0, 3.0 and 3.4 released in 2011, 2015 and 2018 added new features such as arbitrary high-order spaces, non-conforming AMR, HPC miniapps and mesh optimization. An important milestone was the initial GPU support added in MFEM-4.0, which was released in May 2019. The latest version is 4.1, which was released in March 2020.

MFEM is being actively developed on GitHub with contributions from many users and developers worldwide. Users can report bugs and connect with the MFEM developer community via the GitHub issue tracker at https://github.com/mfem/mfem/issues. Details on testing, continuous integration, and how to contribute to the project can be found in the top-level `README` and `CONTRIBUTING.md` files in the MFEM repository.

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

## 2. Finite element abstractions

To illustrate some of the functionality of MFEM, we consider the model Poisson problem with homogeneous boundary conditions:

$$\begin{aligned} &\text{Find } u : \Omega \to \mathbb{R} \text{ such that} \\ &\quad -\Delta u = f \quad \text{in } \Omega \\ &\quad\quad u = 0 \quad \text{on } \Gamma \end{aligned} \tag{1}$$

where $\Omega \subset \mathbb{R}^d$ is the domain of interest, $\Gamma$ is its boundary, and $f : \Omega \to \mathbb{R}$ is the given source. The solution to this problem lies in the infinite dimensional space of admissible solutions (cf. e.g. [4])

$$V = \{v \in H^1(\Omega),\, v = 0 \text{ on } \Gamma\}. \tag{2}$$

To discretize (1), we begin by defining a mesh of the physical domain $\Omega$. The mesh is represented in MFEM using a `Mesh` object. Once the mesh is given, we may define a finite dimensional subspace $V_h \subset V$, represented in MFEM by `FiniteElementSpace`. The approximate solution $u_h \in V_h$ is found by solving the corresponding finite element problem:

$$\begin{aligned} &\text{Find } u_h \in V_h \text{ such that} \\ &\int_\Omega \nabla u_h \cdot \nabla v_h = \int_\Omega f\, v_h \qquad \forall v_h \in V_h. \end{aligned} \tag{3}$$

This can be written equivalently as

$$\begin{aligned} &\text{Find } u_h \in V_h \text{ such that} \\ &a(u_h, v_h) = l(v_h) \qquad \forall v_h \in V_h, \end{aligned} \tag{4}$$

where the bilinear form $a(\cdot,\cdot)$ and linear form $l(\cdot)$ are defined by

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v, \tag{5}$$

$$l(v) = \int_\Omega f\, v. \tag{6}$$

These types of forms are represented in MFEM by the classes `BilinearForm` and `LinearForm`, respectively. These forms are expressed as sums of terms defined by classes derived from `BilinearFormIntegrator` and `LinearFormIntegrator`, respectively (see Section 5.1). In the example considered here, the bilinear form (5) has one term of type `DiffusionIntegrator` and the linear form (6) has one term of type `DomainLFIntegrator`. Functions such as $f$, and any material coefficients, are represented as classes derived from `Coefficient`, `VectorCoefficient`, or `MatrixCoefficient`. Note that due to performance considerations, linear and bilinear forms in MFEM are described using sub-classes of the above classes and not with a domain-specific language.

After defining basis functions $\varphi_j$ for the space $V_h$, the finite element problem (3) may be rewritten as

$$\begin{aligned} &\text{Find coefficients } c_j \text{ such that} \\ &\sum_j c_j \int_\Omega \nabla\varphi_j \cdot \nabla\varphi_i = \int_\Omega f\, \varphi_i. \end{aligned} \tag{7}$$

Defining the linear algebra objects

$$A_{ij} = \int_\Omega \nabla\varphi_j \cdot \nabla\varphi_i, \tag{8}$$

$$b_i = \int_\Omega f\varphi_i, \tag{9}$$

$$x_i = c_i, \tag{10}$$

we arrive at the discrete system of linear equations

$$Ax = b. \tag{11}$$

By calling the `FormLinearSystem` method, the `BilinearForm` object representing $a(\cdot,\cdot)$ is transformed into an `Operator` object representing the linear operator $A$, and the `LinearForm` object representing $l(\cdot)$ is transformed into a `Vector` object representing $b$. After the linear system (11) has been solved, the resulting `Vector` object may be used to define a `GridFunction` representing the discrete solution $u_h \in V_h$ by means of the method `RecoverFEMSolution` (see Section 5.2).

This simple example illustrates some of the core concepts and classes in the MFEM example. A more comprehensive description of MFEM's capabilities, including extensions to other discretization techniques, parallelization, mesh adaptivity and GPU acceleration are described in the following sections.

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

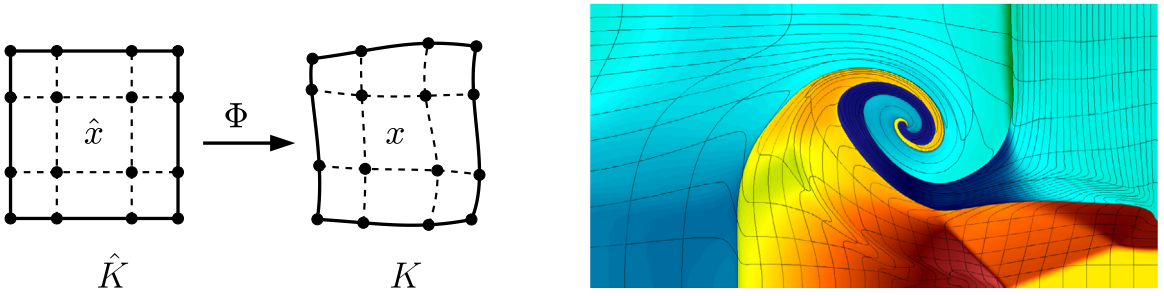



**Fig. 1.** Left: The mapping $\Phi$ from the reference element $\hat{K}$ to a bi-cubic element $K$ in physical space with high-order nodes shown as black dots. Right: Example of a highly deformed high-order mesh from a Lagrangian hydrodynamics simulation (see Section 8.3).

## 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 Section 6.1).

In this section we describe the internal representation aspects of these two classes. Mesh input and output functionality is described in Section 4.5, and mesh manipulation capabilities (refinement, derefinement, etc.) will be described later in Section 7.

### 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 *geometric* (coordinates) data.

The *primary* topology data are: a list of vertices, list of elements, and list of boundary elements. Each element has a type (triangle, quad, tetrahedron, etc.), an attribute (an integer used to identify subdomains and physical boundaries), 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.

The geometric locations of the mesh entities can be described in one of two ways: (1) by the coordinates of all vertices, and (2) by 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 (DOFs) of the mesh. The nodal geometric description is much richer than the one based only on the 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 (see Fig. 1).

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 Fig. 1. The mapping $\Phi$ itself is defined in terms of the basis functions $\{w_i(\hat{x})\}_{i=1}^N$, typically polynomials, and the local nodal coordinates $\mathbf{x}_K$ which are extracted/derived from the global nodal vector $\mathbf{x}$,

$$x(\hat{x}) = \Phi(\hat{x}) = \sum_{i=1}^{N} \mathbf{x}_{K,i} w_i(\hat{x}) \,. \tag{12}$$

Both $\{w_i\}$ and $\{\mathbf{x}_K\}$ are defined from the geometric mesh description — either the vertex coordinates with linear (bilinear for quadrilaterals, or trilinear for hexahedra) 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 $\Phi$, for a particular element $K$, is represented by the class `ElementTransformation`. 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 the 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 dimension of 2 and space dimension of 3, see e.g. Fig. 19.

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

### 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. Assuming a linear mesh, 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 Section 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. In the case of high-order spaces, the goal is to 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 Section 4.4.

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 [8].

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 $(k+2)\times(k+2)$ (2D) and $(k+2)\times(k+2)\times(k+2)$ (3D) elements, with $k$ the continuity 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).

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

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

partitioner [29,30] 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 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. In this case, the parallel partitioning is performed using a space-filling curve instead of using METIS. This is discussed in more detail later in Section 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.

## 4. Finite element discretization

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`. In the `FiniteElement` class, this information is represented by a specifier of type `Geometry::Type`. 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 $\hat{d} \leq d$ 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 $K$.

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

`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 $H(\mathrm{div})$-conforming basis functions [32]. 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 $H(\mathrm{curl})$-conforming basis functions [32]. 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 interpolatory finite elements (referred to as *nodal* finite elements in MFEM), evaluating the $j$th basis function at the $i$th node gives $\delta_{ij}$ (the Kronecker delta). However, this is not true in general for all finite element types.

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 $\delta_{ij}$. 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 $H(\mathrm{curl})$-conforming (with class names beginning with `ND`, short for *Nedelec*), and the $H(\mathrm{div})$-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. These elements can be used with several types of bases, including the nodal Lagrange basis at Gauss–Lobatto or uniform points (or Gauss–Legendre points for $L^2$ finite elements) and the Bernstein basis. For an illustration, see Fig. 2.

In addition to the methods for evaluating the basis functions and their derivatives, the 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.

In order to facilitate programming independent of the mesh type, while simultaneously defining any required permutations of DOFs shared by neighbor elements in the process of mapping global DOFs to local DOFs, MFEM introduces the abstract base class `FiniteElementCollection`. Its main functionality is to (1) define a specific finite element for every

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

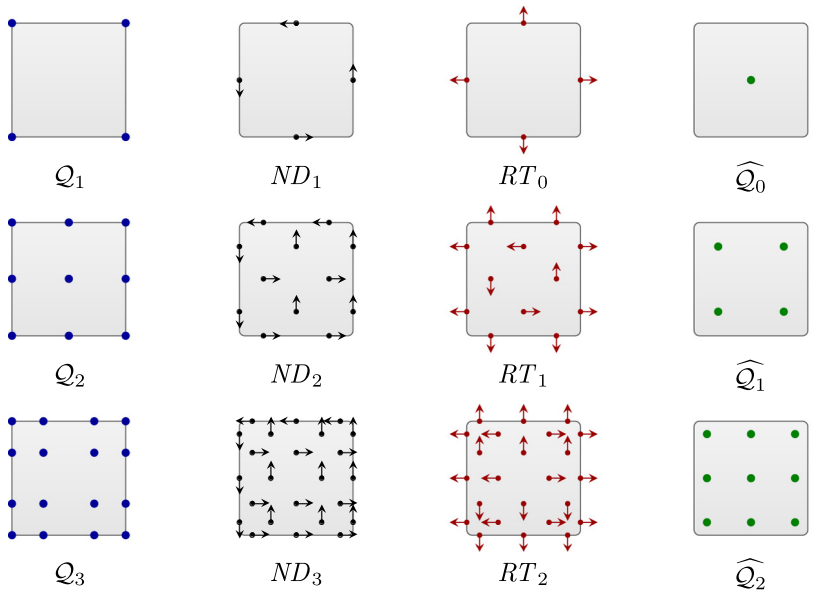



**Fig. 2.** Linear, quadratic and cubic $H^1$ finite elements and their respective $H(\text{curl})$, $H(\text{div})$ and $L^2$ counterparts in 2D. Note that the MFEM degrees of freedom for the Nedelec (*ND*) and Raviart–Thomas (*RT*) spaces are not integral moments, but dot products with specific vectors in specific points as shown above.

mesh entity type, and (2) define a permutation for the DOFs on any mesh entity type (face, edge), based on the *orientation* of that entity relative to any other possible orientations; 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 also defines *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. Finite element spaces

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, 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. Various renumbering schemes, such as [33], are also supported to improve the cache locality.

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).

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

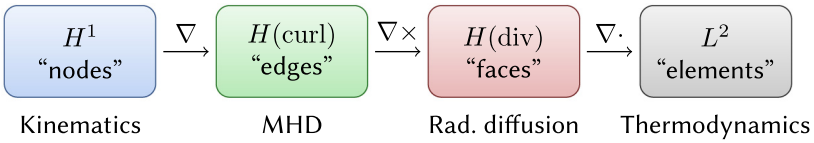



**Fig. 3.** Continuous de Rham complex in 3D and example physical fields that can be represented in the respective spaces.

### 4.3. Discrete de Rham complex

The *de Rham complex* [34,35] is a compatible multi-physics discretization framework that naturally connects the solution spaces for many common PDEs. It is illustrated in Fig. 3. 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 $H^1$, $H(\mathrm{curl})$, $H(\mathrm{div})$, or $L^2$ space, respectively. Note that the order of the space is simply a parameter in the constructor of the respective `*_FECollection`, see Fig. 2.

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 Section 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 Section 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.

### 4.4. High-order spaces

High-order methods are playing an increasingly important role in computational science due to their potential for better simulation accuracy and favorable scaling on modern architectures [3,5,6,36,37]. MFEM supports arbitrary-order elements, and provides efficient implementations of specialized algorithms designed to control the algorithmic complexity with respect to the polynomial order, see Section 5.4.

### 4.5. Input/output and 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`. (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 [25] and VisIt [26,27]. 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 so all VisIt functionality can be used directly. Various file formats are supported, including in-memory remote visualization via socket connection in the case of GLVis.

For mesh I/O, there are two MFEM native ASCII formats: one for generic (non-NURBS) meshes, and one that 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 can use the method `ParMesh::ParPrint()`.

Other *input* formats supported by class `Mesh` are: Netgen [17,38], TrueGrid [39], unstructured VTK [40], Gmsh (linear elements only) [41], and Exodus format (produced by the Cubit mesh generator, among others) [42]. 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 base class `DataCollection` along with several derived classes: `VisItDataCollection`: writes an additional `.mfem_root` file that can be opened by the MFEM plugin in VisIt [26,27]; `SidreDataCollection`: a set of data formats

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

based on the Sidre component of LLNL's Axom library [43] 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 [44] which also supports binary I/O and can be opened by VisIt. Note that the class `VisItDataCollection` uses the default ASCII format to save the mesh and finite element solution fields. The class `ParaViewDataCollection` can be used to output XML data in ParaView's "VTU" format, using either ASCII or compressed binary format. In addition to standard low-order output, `ParaViewDataCollection` also supports ParaView's high-order Lagrangian elements.

## 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. As discussed in Section 2, before discretizing a linear PDE using the finite element method, it is converted into a variational form like (4) consisting of a bilinear and a linear form. In MFEM, they are represented by the classes `BilinearForm` and `LinearForm`, respectively. Depending on the PDE, each of these forms consists of one or more terms, called *integrators* in MFEM. The process of describing the PDE in MFEM consists of defining a `BilinearForm` and a `LinearForm` and then adding integrators to them by calling their `Add*Integrator` methods, e.g. `AddDomainIntegrator` or `AddBoundaryIntegrator`. The main parameter for these methods is an instance of an integrator: a subclass of the abstract base classes `BilinearFormIntegrator` and `LinearFormIntegrator`. An extensive list of the integrators defined in MFEM can be found at https://mfem.org/fem. Note that this design is extensible since it allows users to implement and use their own integrators.

There are many different approaches for expressing a given PDE in variational form which, in turn, give rise to different finite element methods for the same given PDE. MFEM's included examples illustrate some of these different methods. For example, a very common approach for discretizing Poisson's equation is to use $H^1$ elements of any order and spatial dimension, where the basis functions are continuous across element interfaces. This is illustrated in Example 1. 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) [45] and discontinuous Galerkin (DG) discretizations, respectively, see Section 8.1. The examples include interactive documentation (in `examples/README.html` or online at https://mfem.org/examples) organized by the different discretization methods available in the library and are the fastest route to learn about MFEM's capabilities.

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 $2 \times 2$ 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 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 Section 3.3. This allows for isogeometric analysis, where the basis is refined without changing the geometry or its parametrization [8].

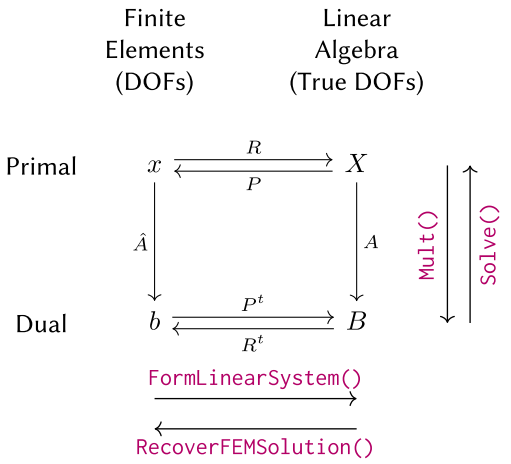
MFEM includes various ordinary differential equation (ODE) solvers that can be used in conjunction with the finite elements and bilinear forms to discretize the time derivative terms. Many ODE solvers are distributed with the library: various implicit and explicit Runge–Kutta (RK) methods including singly-diagonal 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. Explicit and fully-implicit time stepper solvers (TS) from PETSc [46] are also supported. Finally, MFEM's ODE solvers can be extended by inheriting from the abstract base class `ODESolver`.

### 5.2. Finite element linear systems

One of the main operations that MFEM performs is the construction of a linear system of the form (11) given a finite element description of problem such as in (4). Performing this task while supporting distributed memory architectures, high-order basis functions, non-conforming meshes, or more general basis function types introduces complications that require careful treatment. To manage these complexities 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. Parallel linear algebra via PETSc is supported via the classes `PetscParVector` and `PetscParMatrix`; the latter also provides on-the-fly conversion routines between *hypre* and PETSc parallel data formats. The finite element objects include `(Par)GridFunction`, `(Par)LinearForm`, and `(Par)BilinearForm`. For convenience `(Par)GridFunction` and

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.




**Notation**

- $x$: `ParGridFunction` (DOF primal vector)
- $\hat{A}$: `ParBilinearForm` (DOF primal-to-dual)
- $b$: `ParLinearForm` (DOF dual vector)
- $X$: `Vector`/`HypreParVector` (TDOF primal vector)
- $A$: `HypreParMatrix` (TDOF primal-to-dual)
- $B$: `Vector`/`HypreParVector` (TDOF dual vector)
- $P$: `HypreParMatrix` (TDOF-to-DOF rectangular)
- $R$: `SparseMatrix` (DOF-to-TDOF rectangular)

**Fig. 4.** 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.

`(Par)LinearForm` inherit from the `Vector` class and can therefore be used as vectors, and similarly `(Par)BilinearForm` can be used as a matrix.

Fig. 4 illustrates the relationship between the finite element and linear algebra objects in MFEM. 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, denoted by $x$ in Fig. 4. $X$ in Fig. 4 is a linear algebra `Vector` (or `HypreParVector`) related to this `ParGridFunction` but potentially quite different. $X$ represents a non-overlapping, parallel decomposition of the *true* degrees of freedom of the `ParGridFunction` $x$. 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. Thus, the linear algebra `Vector` represented by $X$ may be much smaller than the `ParGridFunction` $x$. The $P$ and $R$ operators shown in Fig. 4, 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.

The `ParGridFunction`, labeled $x$ in Fig. 4, 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 $f_i$ in

$$f(\vec{x}) = \sum_i f_i \varphi_i(\vec{x}). \tag{13}$$

Conversely, a `ParLinearForm`, labeled $b$ in Fig. 4, 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 [1]. More importantly, they can be used to map a `ParGridFunction` to a physical quantity of interest. For example, if we have a `ParGridFunction` $\rho$ 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 $v \cdot \rho$ 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 complementary 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 $P^t$ 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 Fig. 4, the linear algebra operator $A$ can be computed from the `(Par)BilinearForm` $\hat{A}$ as $A = P^t \hat{A} P$; however, many finite element linear systems require boundary conditions to ensure that they are non-singular. To

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

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:

$$A = P^t \hat{A} P,$$
$$X = Rx,$$
$$B = P^t b.$$

Further modifications are also performed in order to impose essential boundary conditions.

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*, see e.g. [5,32]. Note that hybridization in MFEM is applied to a single bilinear form, see [47], instead of the more classical hybridization approach applied to mixed finite element discretizations. 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 [47].

### 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 Fig. 5 for the simple case of a symmetric linear operator on second order ($Q_2$) scalar continuous ($H^1$) elements, where we use the notions **T-vector** (true vector), **L-vector** (local vector), **E-vector** (element vector) and **Q-vector** (quadrature 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. Note that class `(Par)GridFunction` represents an L-vector, and T-vector is typically represented by either `HypreParVector` or `Vector`, cf. Fig. 4. We remark that although the decomposition presented in Fig. 5 is appropriate for square, symmetric linear operators, the generalization of this finite element decomposition to rectangular and nonlinear operators is straightforward.

One of the challenges with high-order methods is that a global sparse matrix is no longer an efficient representation of a high-order linear operator, both with respect to the FLOPs needed for its evaluation [48], as well as the memory transfer needed for a matrix–vector product (matvec) [49,50]. 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 quadrature points) evaluator $B$.
- Operator at quadrature points $D$.

More generally, when the test and trial space differ, each space has its own versions of $P$, $G$ and $B$.

Note that in the case of adaptive mesh refinement (AMR), the restriction $P$ will involve not just extracting sub-vectors, but evaluating values at constrained degrees of freedom through the AMR interpolation, see Section 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 Fig. 5. This is discussed further in Section 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

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

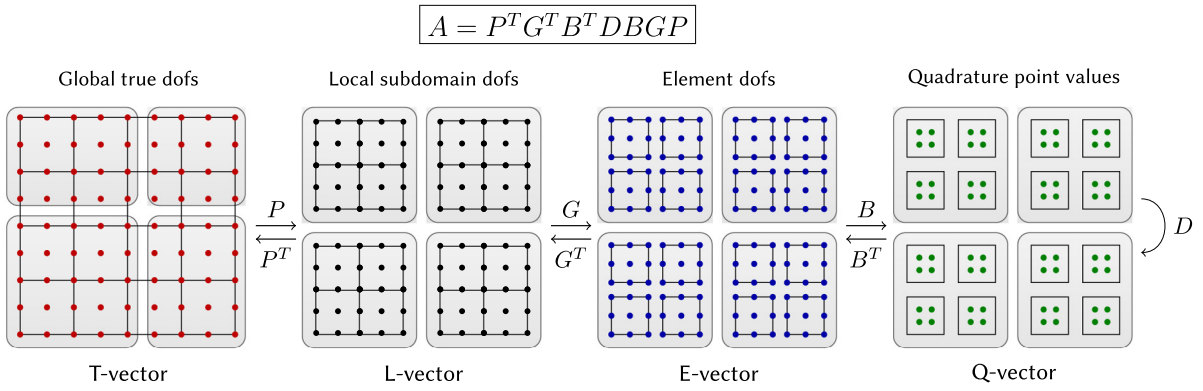



**Fig. 5.** Fundamental finite element operator decomposition. This algebraically factored form is a much better description than a global sparse matrix for high-order methods and is easy to incorporate in a wide variety of applications. See also the libCEED library in [37].

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 (i.e. transformations $Y \to X^T Y X$) with $B$, $G$ and $P$, starting from its point-wise kernel $D$, a matrix–vector product 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 the **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 by the use of *partial assembly* algorithms, as described in the following section. In this case, 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 quadrilateral and hexahedral 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. The action of the operator is computed by multiplying with this matrix. At high orders this requires both a large amount of memory to store the matrix, as well as many floating point operations to compute and apply it. By exploiting the structure shown in Section 5.3 as well as the basis functions structure, there are options for creating operators that require much less storage and scale better at high orders. This section introduces partial assembly and sum factorization [6,48], 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[1] 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, $B$, from the assembled operator. This leaves only the $D$ operator to store for every element, see Section 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, consider the decomposition of the mass matrix evaluated on a single element $E$

$$(M_E)_{ij} = \int_E \rho\, \varphi_j \varphi_i \, dx, \tag{14}$$

[1] Domain-based operators correspond to bilinear forms which use integrals over the problem domain, as opposed to its boundary, for example.

where $\rho$ is a given density coefficient and $\{\varphi_i\}$ are the finite element basis functions on the element $E$. Changing the variables in the integral from $E$ to the reference element $\hat{E}$ and applying a quadrature rule with points $\{\hat{x}_k\}$ and weights $\{\alpha_k\}$ yields

$$(M_E)_{ij} = \sum_k \alpha_k \left(\rho \circ \Phi\right)(\hat{x}_k)\, \hat{\varphi}_j(\hat{x}_k)\hat{\varphi}_i(\hat{x}_k)\, \det(J(\hat{x}_k)). \tag{15}$$

In the last expression, $\Phi$ is the mapping from the reference element $\hat{E}$ to the physical element $E$, $J$ is its Jacobian matrix, and $\{\hat{\varphi}_i\}$ are the finite element basis functions on the reference element. Defining the matrix $B$ of basis functions evaluated at quadrature points as $B_{ki} = \hat{\varphi}_i(\hat{x}_k)$, the above equation can be rewritten as

$$(M_E)_{ij} = \sum_k B_{ki}(D_E)_{kk}B_{kj}, \qquad \text{where} \quad (D_E)_{kk} = \alpha_k \det(J(\hat{x}_k))\left(\rho \circ \Phi\right)(\hat{x}_k), \quad (D_E)_{kl} = 0,\ k \neq l. \tag{16}$$

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 $M_E$.

For general $B$, its application requires the same order of floating point operations as applying the fully-assembled $M_E$ matrix: $\mathcal{O}(p^{2d})$ (assuming that the number of quadrature points is $\mathcal{O}(p^d)$). 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

$$B_{ki} = \hat{\varphi}^{1d}_{i_1}\left(\hat{x}^{1d}_{k_1}\right)\ldots\hat{\varphi}^{1d}_{i_d}\left(\hat{x}^{1d}_{k_d}\right), \qquad k = (k_1, \ldots, k_d), \quad i = (i_1, \ldots, i_d), \tag{17}$$

with $d$ the number of dimensions. In this case the matrix $B$ itself is decomposed as a tensor product of smaller one-dimensional matrices $B^{1d}_{lj} = \hat{\varphi}^{1d}_j\left(\hat{x}^{1d}_l\right)$ so that

$$B_{ki} = B^{1d}_{k_1 i_1}\ldots B^{1d}_{k_d i_d}. \tag{18}$$

Applying the series of $B^{1d}$ matrices reduces the overall number of floating point operations when applying $M_E$ to $\mathcal{O}(p^{d+1})$ (assuming that the number of 1D quadrature points is $\mathcal{O}(p)$). This evaluation strategy 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

$$(Bv)_k = \sum_i B_{ki}v_i = \sum_i \hat{\varphi}_i(\hat{x}_k)v_i, \tag{19}$$

which requires $\mathcal{O}(p^{2d})$ ($d = 2$) storage and operations for the matrix–vector product. When using the alternative form (18) the operation can be rewritten as

$$(Bv)_k = \sum_i B_{ki}v_i = \sum_{i_1,i_2} B^{1d}_{k_1 i_1}B^{1d}_{k_2 i_2}V_{i_1 i_2} = \left[B^{1d}V\left(B^{1d}\right)^t\right]_{k_1 k_2}, \tag{20}$$

where $V$ is the vector $v$ viewed as a square matrix: $V_{i_1 i_2} = v_i$. This highlights an interesting aspect of sum factorization: with each smaller matrix product with $B^{1d}$, an additional axis is converted from basis ($i_j$) to quadrature ($k_j$) indices. The same reasoning can also be applied to three spatial dimensions. Using the sum factorization approach, the storage was reduced to $\mathcal{O}(p^d)$ and the number of operations to $\mathcal{O}(p^{d+1})$.

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 algebraic multigrid solvers are difficult to apply. These issues are discussed further in Section 6.2.

The storage and asymptotic number of floating point operations required for assembly and evaluation using the different methods are recorded in Table 1. Sum factorization can be utilized to reduce the cost of assembling the local element matrices and thus the cost of full assembly (**T-vector** level) – this is shown in the second row of the table. Furthermore, partial assembly has improved the asymptotic scaling for high orders in both storage and number of floating point operations for assembly and evaluation. Therefore, partial assembly is well-suited for high orders.

**Table 1**
Comparison of storage and Assembly/Evaluation FLOPs required for full and partial assembly algorithms on tensor-product element meshes (quadrilaterals and hexahedra). 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.

| Method | Storage | Assembly | Evaluation |
|---|---|---|---|
| Traditional full assembly + matvec | $\mathcal{O}(p^{2d})$ | $\mathcal{O}(p^{3d})$ | $\mathcal{O}(p^{2d})$ |
| Sum factorized full assembly + matvec | $\mathcal{O}(p^{2d})$ | $\mathcal{O}(p^{2d+1})$ | $\mathcal{O}(p^{2d})$ |
| Partial assembly + matrix-free action | $\mathcal{O}(p^d)$ | $\mathcal{O}(p^d)$ | $\mathcal{O}(p^{d+1})$ |

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 parallel, which makes partial assembly

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

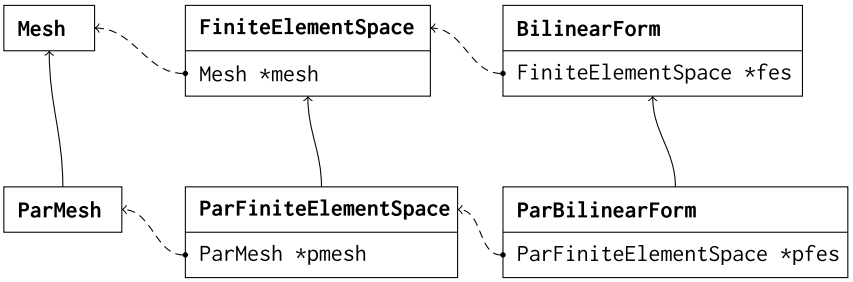



**Fig. 6.** Parallel classes inherit from, and partially override, serial classes.

with sum factorization a promising portion of the finite element algorithm to offload to co-processors, such as GPUs. In MFEM, partial assembly and sum factorization are implemented in the bilinear and nonlinear form integrators themselves. Specifically, in the base class `BilinearFormIntegrator`, the assembly and evaluation are performed by the virtual methods `AssemblePA` and `AddMultPA`, respectively. MFEM supports partial assembly for the entire de Rham complex, including $H^1$, $H(\text{curl})$, $H(\text{div})$, and $L^2$ spaces. MFEM currently supports partial assembly for tensor-product elements (quadrilaterals and hexahedra), for which sum factorization is most efficient. Partial assembly on simplices and mixed meshes are partially supported through MFEM's integration with the libCEED library. In the case of simplices, sum factorization cannot be used for the evaluation of the action of the $B$ operator, however, other efficient algorithms exist, for example using the Bernstein basis [51].

## 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 an 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 Fig. 6, occasionally overriding small parts of the code using virtual functions.

If $K$ is the number of MPI tasks, MFEM decomposes the problem domain (i.e. the mesh) into $K$ parts, with the goal of processing the parts as locally as possible, see Fig. 7. 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 Section 3.4 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*. 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 Section 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.

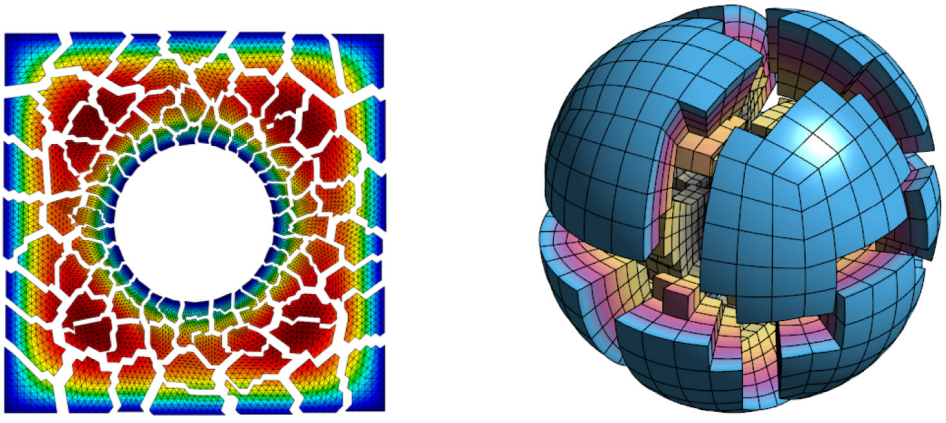


**Fig. 7.** Left: Solving a Poisson problem (parallel example 1, `examples/ex1p.cpp`) in parallel on 100 processors with a relatively coarse version of `data/square-disc.mesh`. Right: Unstructured parallel decomposition of a fourth order NURBS mesh of the unit ball on 16 processors.

The finite element stiffness matrix at the **L-vector** level, $A_L \equiv G^T B^T D B G$, has $K$ diagonal blocks and can be assembled without any parallel communication. The prolongation matrix $P$ is parallel and its construction requires communication,

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

however part of that communication can be overlapped with the computation of $A_L$. 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 Section 5.3, the final parallel assembly is computed with a parallel $P^t A_L P$ triple matrix product, which is performed either with the *hypre* library [28] (making use of the `RAP` triple-product kernel which *hypre* provides internally for the coarse-grid operator construction in its algebraic multigrid solvers), or via PETSc routines, depending on the underlying operator type set via the `SetOperatorType` method of the `ParBilinearForm` class.

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.

### 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 Section 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

```
205   OperatorPtr A;
206   Vector B, X;
207   a->FormLinearSystem(ess_tdof_list, x, *b, A, X, B);
```

and then *hypre*'s BoomerAMG preconditioner can be used with the preconditioned conjugate gradient (CG) method to solve it simply with

```
212   Solver *prec = NULL;
213   if (!pa) { prec = new HypreBoomerAMG; }
214   CGSolver cg(MPI_COMM_WORLD);
215   cg.SetRelTol(1e-12);
216   cg.SetMaxIter(2000);
217   cg.SetPrintLevel(1);
218   if (prec) { cg.SetPreconditioner(*prec); }
219   cg.SetOperator(*A);
220   cg.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) [52] which is specifically designed for second-order definite Maxwell problems discretized with Nedelec $H(\text{curl})$-conforming elements, see Section 4.3. The AMS algorithm needs the discrete gradient operator between the nodal $H^1$ 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`:

```
2856  ParDiscreteLinearOperator *grad;
2857  grad = new ParDiscreteLinearOperator(vert_fespace, edge_fespace);
2858  if (trace_space)
2859  {
2860     grad->AddTraceFaceInterpolator(new GradientInterpolator);
2861  }
2862  else
2863  {
2864     grad->AddDomainInterpolator(new GradientInterpolator);
2865  }
2866  grad->Assemble();
2867  grad->Finalize();
2868  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:

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

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

Different preconditioning options are also easy to combine as illustrated in Example 4p which solves an $H(\mathrm{div})$ problem discretized with Raviart–Thomas finite elements. Depending on the dimension, and the use of hybridization or static condensation, see Section 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);
```

MFEM provides easy access to a variety of other iterative and direct solvers. For example, discretization-enhanced Balancing Domain Decomposition by Constraints (BDDC) solvers from PETSc [53] are exposed via the `PetscBDDCSolver` class. These methods provide customizable, multilevel preconditioning for various finite element discretizations, as well as for isogeometric analysis, see [54] and [55] for a recent review. Examples `examples/petsc/ex3p.cpp` and `examples/petsc/ex4p.cpp` construct the BDDC solver for the second-order definite Maxwell equations [56,57] as well as for the $H(\mathrm{div})$ [58] problem, as shown in the below code snippet:

```
PetscParMatrix A;
a->SetOperatorType(Operator::PETSC_MATIS);
a->FormLinearSystem(ess_tdof_list, x, *b, A, X, B);

ParFiniteElementSpace *prec_fespace =
   (a->StaticCondensationIsEnabled() ? a->SCParFESpace() : fespace);
PetscPCGSolver *pcg = new PetscPCGSolver(A);
PetscPreconditioner *prec = NULL;

PetscBDDCSolverParams opts;
opts.SetSpace(prec_fespace);
prec = new PetscBDDCSolver(A,opts);
pcg->SetPreconditioner(*prec);
```

In addition, the `PetscBDDCSolver` class provides support for preconditioning symmetric indefinite linear systems [59], as shown in `examples/petsc/ex5p.cpp` for the mixed $H(\mathrm{div})$-$L^2$ formulation of the Poisson equation. The same example showcases MFEM's interface to the generic field-split solver `PetscFieldSplitSolver` in PETSc, which can be used to quickly and easily prototype block-preconditioning techniques for complicated multi-physics problems.

With high-order methods, the explicit assembly of finite element matrices becomes a bottleneck, as discussed in Section 5.4. While matrix-free (partially assembled) high-order operators offer many benefits, one of their drawbacks is that the entries of the matrix are not readily available, and thus purely algebraic preconditioners cannot be used. An ongoing area of research pursued by the MFEM team is the development of matrix-free preconditioners for high-order operators. These include matrix-free $h$- and $p$-multigrid methods, as well as low-order refined preconditioning, which is based on the idea of preconditioning a spectrally equivalent low-order refined operator obtained by meshing the nodes of each of the high-order elements [48,60–63].

### 6.3. GPU acceleration

Version 4.0 of MFEM introduced initial support for hardware accelerators, such as GPUs, as well as programming models and libraries, such as CUDA, OCCA [64], libCEED [65], RAJA [66] and OpenMP in the library. This support is based on new backends and kernels working seamlessly with a new lightweight memory spaces manager. Several of the MFEM example codes and the Laghos miniapp [67] (see Section 8.3) can now take advantage of this GPU acceleration.

Given the rapidly changing computing landscape, the MFEM performance portability approach has been to not commit to a single framework, but instead to support a variety of different backends, which may differ in the set of features they actually implement, the technology they use (OCCA, external library such as libCEED, OpenMP, CUDA, RAJA, HIP),

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

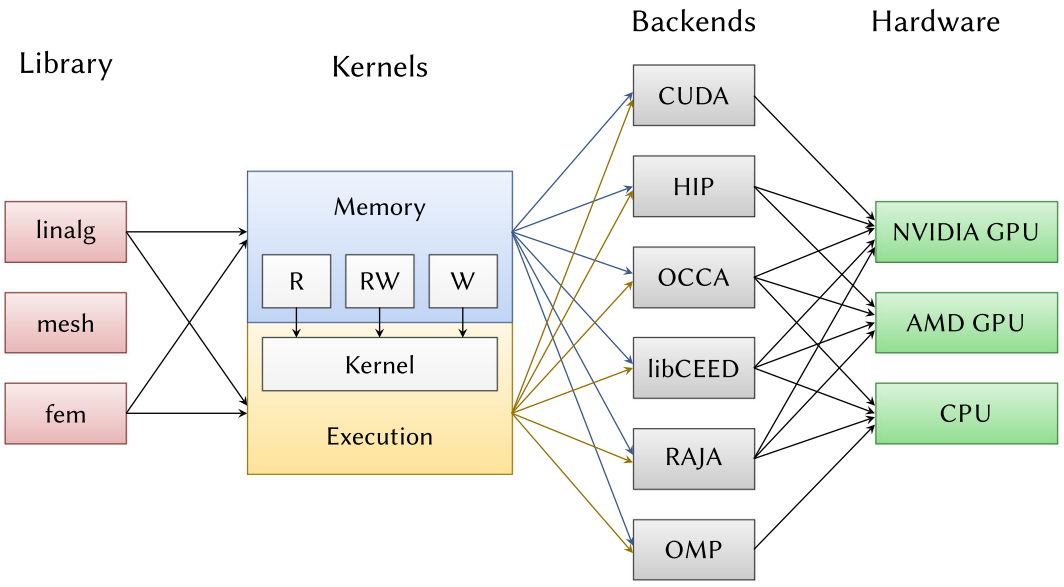



**Fig. 8.** Diagram of MFEM's modular design for accelerator support, combining flexible memory management with runtime-selectable backends for executing key finite element and linear algebra kernels.

the targeted architecture (Intel, IBM, AMD, Nvidia), the algorithms to achieve performance, and the implementations of these algorithms. This flexibility allows generic backends like the core backend of MFEM, using the macro `MFEM_FORALL` described below, to target all architectures with a performance emphasis on GPU architectures. The OCCA backend serves a similar purpose of generic backend using the OCCA just-in-time compilation technology, but varies in the algorithms used, and more significantly in the implementation ideas. The libCEED backend uses the libCEED library that itself contains numerous backends. This modularity over backends increases both the portability and performance of MFEM algorithms, as different backends provide the best performance in different scenarios, see [68,69].

One main feature of the MFEM performance portability approach is the ability to select the backends at runtime: e.g. different MPI ranks can choose different backends (like CPU or GPU) allowing applications to take full advantage of heterogeneous architectures. Another important aspect of MFEM's approach is the ability to easily mix CPU-only code with code that utilizes the new backends, thus allowing for selective gradual transition of existing capabilities.

Most of the kernels are based on a single source, while still offering good efficiency. For performance critical kernels, where single source does not provide the best performance, the implementation introduces dispatch points based on the selected backend and, in some cases, on kernel parameters such as the finite element order. Many of the linear algebra and finite element operations can now benefit fully from the new GPU acceleration.

Fig. 8 illustrates the main components of MFEM's modular design for accelerator support. The *Library* side of MFEM (on the left) represents the software components where new kernels have been added. The following components have been extended with new accelerated *kernels*:

- The `linalg` directory: most operations in class `Vector` and some operations (e.g. matvec) in class `SparseMatrix`. Other classes, such as the Krylov solvers and time-stepping methods, are automatically executed on the device because they are written in terms of `Vector` operations.
- The `mesh` directory: the computation of the so-called *geometric factors*.
- The `fem` directory: the *mass*, *diffusion*, *convection* (including DG), gradient, divergence, and some $H(\text{curl})$ `BilinearFormIntegrator`s; the *element restriction* and *quadrature interpolator* operators ($G$ and $B$ on Fig. 5) associated with class `FiniteElementSpace`; the matrix-free action of the `BilinearForm`, `MixedBilinearForm` and `NonlinearForm` classes.

Note, however, that many of the capabilities in the library are still not ported to GPU including the mesh refinement/derefinement, a number of the `BilinearFormIntegrator` classes, sparse matrix assembly, error estimation, integration with external libraries, etc. Some of these missing parts are currently under development and will become available in the near future.

The integration of the kernels has been made at the *for-loop* level. Existing code has been transformed to use a new *for-loop* abstraction defined as a set of new `MFEM_FORALL` macros, in order to take advantage of various *backends* supported via the new macros. This approach allows for gradual code transformations that are not too disruptive for both, MFEM developers and users. Existing applications based on MFEM should be able to continue to work as before and have an easy

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

way to transition to accelerated kernels. Another requirement is to allow interoperability with other software components or external libraries that MFEM could be used in conjunction with, for instance *hypre*, PETSc, and SUNDIALS, among others.

The main challenge in this transition to *kernel-centric* implementation is the need to transform existing algorithms to take full advantage of the increased levels of parallelism in the accelerators while maintaining good performances on standard CPU architectures. Another important aspect is the need to manage memory allocation and transfers between the CPU (host) and the accelerator (device). In MFEM, this is achieved using a new `Memory` class that manages a pair of host and device pointers and provides a simple interface for copying or moving the data when needed. An important feature of this class is the ability to work with externally allocated host and/or device pointers which is essential for interoperability with other libraries.

*Lambda-capturing* for-loop *bodies*

There are multiple ways to write kernels, but one of the easiest ways, from the developer's point of view, is to turn *for-loop* bodies into *kernels* by keeping the bodies unchanged and having a way to *wrap* and *dispatch* them toward native backends. This can be easily done for the first outer for-loop using standard C++11 features. However, additional care is required when one wants to address deeper levels of parallelism. The following listing illustrates a possible implementation in MFEM of the diffusion setup (partial assembly) kernel in 2D.

```
1  void PADiffusionSetup2D(const int Q, const int N, const Array<double> &w,
2                          const Vector &j, const double alpha, Vector &y) {
3     auto W = w.Read();
4     auto J = Reshape(j.Read(), Q*Q, 2, 2, N);
5     auto Y = Reshape(y.Write(), Q*Q, 3, N);
6     MFEM_FORALL_2D(e, N, Q, Q,
7        MFEM_FOREACH_THREAD(qx, x, Q)
8           MFEM_FOREACH_THREAD(qy, y, Q) {
9              const int q = qx + qy * Q;
10             const double J11 = J(q,0,0,e), J21 = J(q,1,0,e);
11             const double J12 = J(q,0,1,e), J22 = J(q,1,1,e);
12             const double c_detJ = alpha * W[q] / ((J11*J22)-(J21*J12));
13             Y(q,0,e) =  c_detJ * (J12*J12 + J22*J22);
14             Y(q,1,e) = -c_detJ * (J12*J11 + J22*J21);
15             Y(q,2,e) =  c_detJ * (J11*J11 + J21*J21);
16          });
17 }
```

The kernel is structured as follows:

- Lines 3 to 5 are the portion of the kernel where the pointers are requested from the memory manager (presented in the next paragraph) and turned into tensors with given shapes.
- Line 6 holds the `MFEM_FORALL_2D` *wrapper* of the first outer *for-loop*, with the iterator, the range, and the *for-loop* body.
- Lines 7 and 8 allow inner for-loops to be mapped to blocks of threads with arbitrary sizes (from 1 to thousands): it uses another level of parallelism within the lambda body for each mesh element.
- Lines 9 to 15 are the core of the computation and show how to use the tensors declared before entering the kernel. This portion may use *shared* memory as a fast scratch memory shared within the thread block when supported by the respective backend. This kernel is the one used both for the OpenMP and the CUDA backends.

*Memory management*

Before entering each kernel, the pointers that will be used in it have to be requested from the new `Memory` class which acts as the *frontend* of the internal lightweight MFEM memory manager. Access to the pointers stored by the `Memory` class is requested using three modes: `Read`-only, `Write`-only, and `ReadWrite`. These access types allow the memory manager to seamlessly copy or move data to the device when needed. Portions of the code that do not use acceleration (i.e. run on CPU) need to request access to the `Memory` using the *host* versions of the three access methods: `HostRead`, `HostWrite`, and `HostReadWrite`. The use of these access types allows the memory manager to minimize memory transfers between the host and the device. The pointers returned by the three access methods can be reshaped as *tensors* with given dimensions using the function `Reshape` which then allows for easy multi-dimensional indexing inside the computational kernels.

In addition to holding the host and device pointers, the `Memory` class keeps extra metadata in order to keep track of the usage of the different memory spaces. For example, if a vector currently residing in device memory is temporarily needed on the host where it will not be modified (e.g. to save the data to a file), the host code can use `HostRead` to tell the memory manager to copy the data to the host while also telling it that the copied data will not be modified; using this information, the memory manager knows that a subsequent call to, say, `Read` will not require a memory copy from host to device.

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

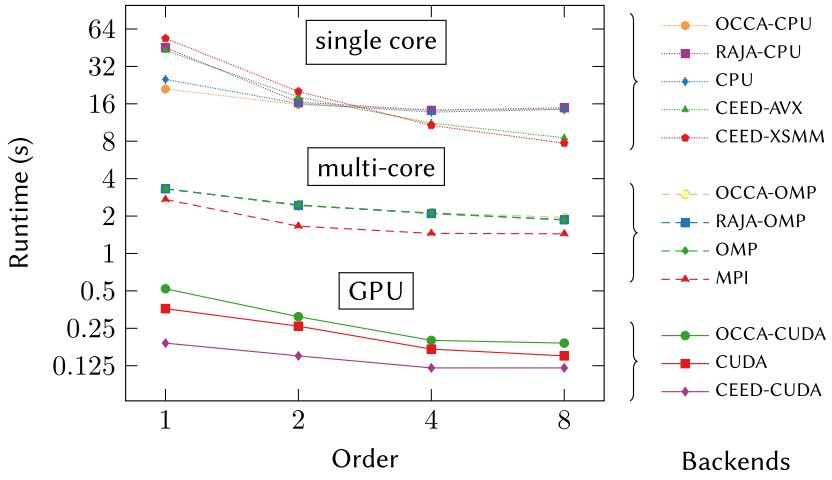



**Fig. 9.** Performance results with MFEM-4.0: Poisson problem (Example 1), 200 conjugate gradient iterations using partial assembly, 2D, 1.3 M dofs, GV100, sm_70, CUDA 10.1, Intel Xeon Gold 6130@2.1 GHz.

*Transitioning applications to GPUs*

Porting existing codes to GPUs can be relatively simple in some cases. The first step is to configure an `mfem::Device` object, e.g. using a string from a command-line option. The next step is typically to enable the partial assembly mode in the `(Par)BilinearForm` object(s). Since in this mode the fully assembled sparse matrix is not available, one has to switch to suitable matrix-free solvers. In cases when an application uses MFEM at a lower level, e.g. to implement some algorithm on an element level, porting to GPU will be more involved. For such cases, the user will typically need to learn more about the `MFEM_FORALL` macros and the memory management.

Some current limitations in the GPU support are: not all pre-defined integrators in MFEM have been ported to GPU; full assembly on GPU (which may be of interest for low-orders) is also not available. As pointed out earlier, these and many other missing components are being actively developed and will become immediately available in the MFEM source repository when completed.

*Results*

Fig. 9 and Table 2 present initial performance results with MFEM v4.0 measured on a Linux desktop with a Quadro GV100 GPU (Volta, 5120 cuda cores, 7.4 TFLOPS FP64 peak; 32 GB HBM2, 870 GB/s peak), CUDA 10.1, and Intel Xeon Gold 6130 CPU (Skylake, 16 cores/32 threads, 970 GFLOPS FP64 peak; 128 GB/s memory bandwidth peak) @ 2.10 GHz.

Single-core, multi-core CPU, and single-GPU performance for different discretization orders is shown, keeping the total number of degrees of freedom (DOFs) constant at 1.3 million in 2D. Results from backends supported in MFEM 4.0, as well as recent results based on the libCEED library (integrated with MFEM) are included. The libCEED library itself includes several backends, targeting, for example, CPUs using AVX instructions, Intel CPUs taking advantage of the LIBXSMM library [70], and GPUs using CUDA. Fig. 9 shows that GPU acceleration offers a significant gain in performance relative to multi-core CPU.

We emphasize that these results are preliminary and additional performance improvements in several of the backends are under active development. Therefore these results illustrate only the current state of the MFEM backends, and should not be viewed as a fair and exhaustive comparison of the specific CPU and GPU hardware.

## 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) [24].

### 7.1. Conforming adaptive mesh refinement

The conforming *h*-adaptivity algorithm in MFEM is based on the bisection procedure for tetrahedral meshes proposed in [71]. This approach supports both uniform refinement of all elements in the mesh, as well as local refinement of only

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

**Table 2**
Performance results with MFEM-4.0: Poisson problem (Example 1), 200 conjugate gradient iterations using partial assembly, 2D, 1.3M dofs, GV100, sm_70, CUDA 10.1, Intel Xeon Gold 6130@2.1 GHz. The best performing backends in each category (GPU, multicore, and CPU) are shown in bold.

| | | $p = 1$ | $p = 2$ | $p = 4$ | $p = 8$ |
|---|---|---|---|---|---|
| GPU | OCCA-CUDA | 0.52 | 0.31 | 0.20 | 0.19 |
| | RAJA-CUDA | 0.38 | 0.30 | 0.28 | 0.45 |
| | CUDA | 0.36 | 0.26 | 0.17 | 0.15 |
| | CEED-CUDA | **0.19** | **0.15** | **0.12** | **0.12** |
| Multicore | OCCA-OMP | 3.34 | 2.41 | 2.13 | 1.95 |
| | RAJA-OMP | 3.32 | 2.45 | 2.10 | 1.87 |
| | OMP | 3.30 | 2.46 | 2.10 | 1.86 |
| | MPI | **2.72** | **1.66** | **1.45** | **1.44** |
| CPU | OCCA-CPU | **21.05** | **15.77** | 14.23 | 14.53 |
| | RAJA-CPU | 45.42 | 16.53 | 14.22 | 14.88 |
| | CPU | 25.18 | 16.11 | 13.73 | 14.45 |
| | CEED-AVX | 43.04 | 18.16 | 11.20 | 8.53 |
| | CEED-XSMM | 53.80 | 20.13 | **10.73** | **7.72** |

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 tetrahedra 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 [71]) 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.

The bisection algorithm consists of several passes. For example, during *green* refinement (cf. [71]), 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.

In parallel, the tetrahedra are marked consistently across processors, as inherited from the serial mesh before the parallel partitioning. The consistently marked tetrahedra guarantee that a face between any two tetrahedra 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.

### 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, see Section 5.4) 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 [72–74]. 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 Section 4.3), at arbitrarily high-order, and can support high-order curved meshes, as well as finite element techniques such as hybridization and static condensation (see Section 5.2). They are also highly scalable, easy to incorporate into existing codes, and can be applied to complex, anisotropic 3D meshes with arbitrary levels of non-conforming refinement. While MFEM's approaches can be exclusively on non-conforming $h$-refinement with fixed polynomial degree.

These approaches are based on a variational restriction approach to AMR, described below. For more details, see [31]. Consider the weak variational formulation (4) where for simplicity we assume that the bilinear form $a(\cdot, \cdot)$ is symmetric. To discretize the problem, we cover the computational domain $\Omega$ with a mesh consisting of mutually disjoint elements

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

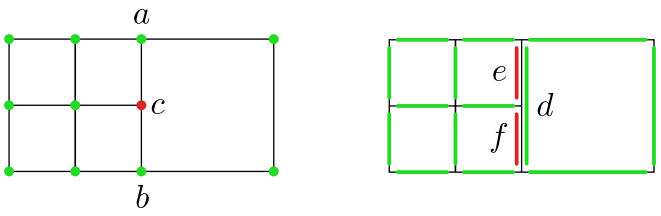



**Fig. 10.** Illustration of conformity constraints for lowest order nodal elements in 2D. Left: Nodal elements (subspace of $H^1$), constraint $c = (a+b)/2$. Right: Nedelec elements (subspace of $H(\mathrm{curl})$, constraints $e = f = d/2$. In all cases, fine degrees of freedom on a coarse-fine interface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

$K_i$, 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 Fig. 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. We use a tree-based data structure to represent refinements which has been optimized to rely only on the following information: (1) elements contain indices of eight vertices, or indices of eight child elements if refined; (2) edges are identified by pairs of vertices; (3) faces are identified by four vertices. 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.

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 $H(\mathrm{curl})$ 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 $H(\mathrm{curl})$ in 2D are illustrated in Fig. 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 of all slave DOFs, then $z$ can be expressed as $z = Wx$, where $x$ is a vector of 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

$$P^t A P x_c = P^t b, \tag{21}$$

where $A$ and $b$ are the finite element stiffness matrix and load vector corresponding to discretization of (4) on the "cut" space (see Section 3.2) $\widehat{V}_h = \cup_i (V_h|_{K_i})$. 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 Section 5.2. An illustration of this process is provided in Fig. 11.

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 corresponds to true DOFs, 1 corresponds to the first generation of slaves that only depend on true DOFs, 2 corresponds to second generation of slaves that only depend on true DOFs and first generation of slaves, and so on. $k$ corresponds to the last generation of slaves. We have

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

are the local interpolation matrices defined only in terms of the edge-to-edge and face-to-face constraining relations. Note that while MFEM supports meshes of arbitrary irregularity ($k \geq 1$), the user can specify a limit on $k$ when refining elements, if necessary (an example of a 1-irregular mesh is shown in Fig. 15).

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

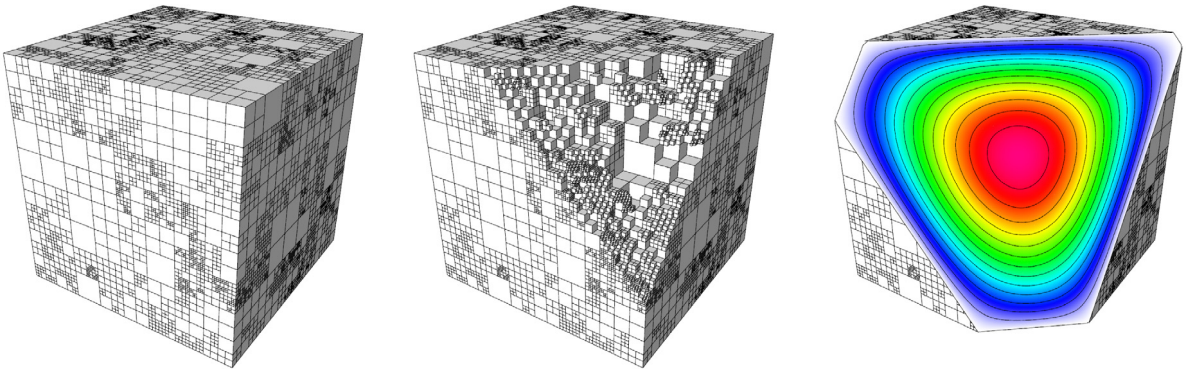


**Fig. 11.** Illustration of the variational restriction approach to forming the global AMR problem. Randomly refined non-conforming mesh (left and center) where we assemble the matrix $A$ and vector $b$ independently on each element. The interpolated solution $x = Px_c$ (right) of the system (21) is globally conforming (continuous for an $H^1$ problem).

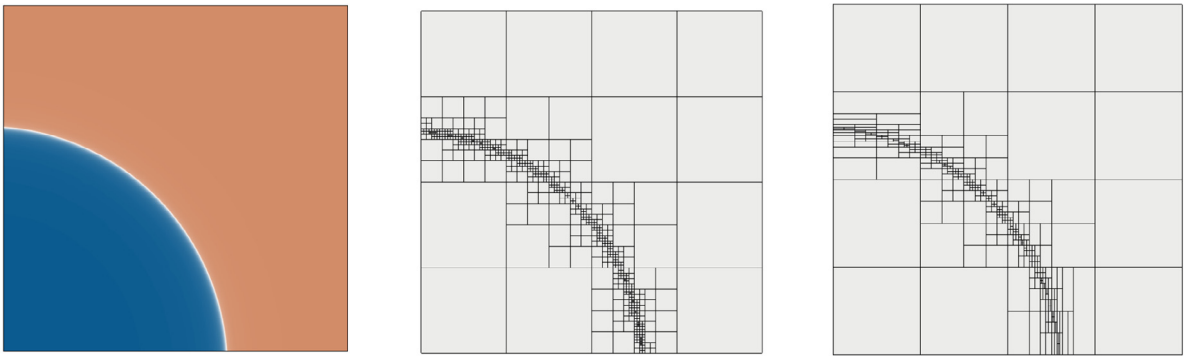


**Fig. 12.** Left: 2D benchmark problem for a Poisson problem with a known exact solution. Center: Isotropic AMR mesh with 2197 DOFs. Right: Anisotropic AMR mesh with 1317 DOFs. Even though the wave front in the solution is not perfectly aligned with the mesh, many elements could still be refined in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

The 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.

```
Split FaceSplitType(v1, v2, v3, v4)
{
   v12 = FindVertex(v1 , v2);
   v23 = FindVertex(v2 , v3);
   v34 = FindVertex(v3 , v4);
   v41 = FindVertex(v4 , v1);

   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 Fig. 12.

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.

Unlike the conforming `ParMesh` class, which is partitioned with METIS, the `ParNCMesh` is partitioned between MPI tasks by splitting a space-filling curve obtained by enumerating depth-first all leaf elements of all refinement trees [75]. The simplest traversal with a fixed order of children at each tree level leads to the well-known Morton ordering, or the Z-curve.

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

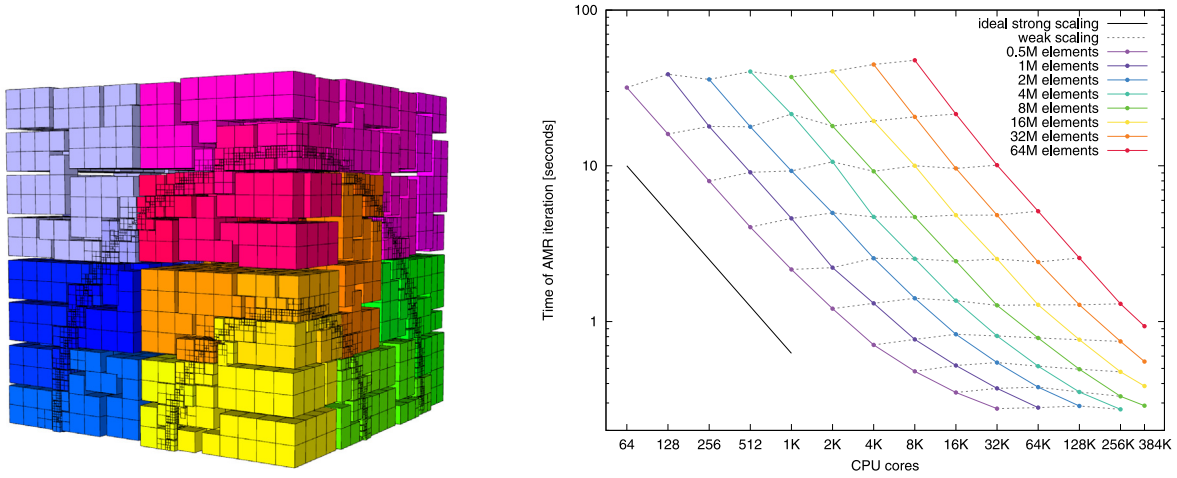



**Fig. 13.** Left: One octant of the parallel test mesh partitioned by the Hilbert curve (2048 domains shown). Right: Overall parallel weak and strong scaling for selected iterations of the AMR loop. Note that these results test only the AMR infrastructure, no physics computations are being timed.

We use instead the more efficient Hilbert curve that can be obtained just by changing the order of visiting subtrees at each level [76]. The use of space-filling curve partitioning ensures that balancing the mesh so that each MPI task has the same number of elements ($\pm 1$ 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 Fig. 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 with $32^3$ 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 ($\pm 1$). We run about 100 iterations of the AMR loop and select iterations that happen to have approximately 0.5, 1, 2, 4, 8, 16, 32 and 64 million elements in the mesh at the beginning. We then plot the times of the selected iterations as if they were 8 independent problems. We run from 64 to 393,216 (384 K) cores on LLNL's Vulcan BG/Q machine. The solid lines in Fig. 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.

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 Section 8.3) which required minimal changes for static refinement support (see Fig. 14) and about 550 new lines of code for full dynamic AMR, including derefinement (see Fig. 15).

### 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 with high-resolution, 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 [77–79]. 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 (e.g. following the motion of a material) 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 step restrictions 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) [80,81]. This enables applications to have 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, and can also be applied to low-order mesh applications that use linear meshes.

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

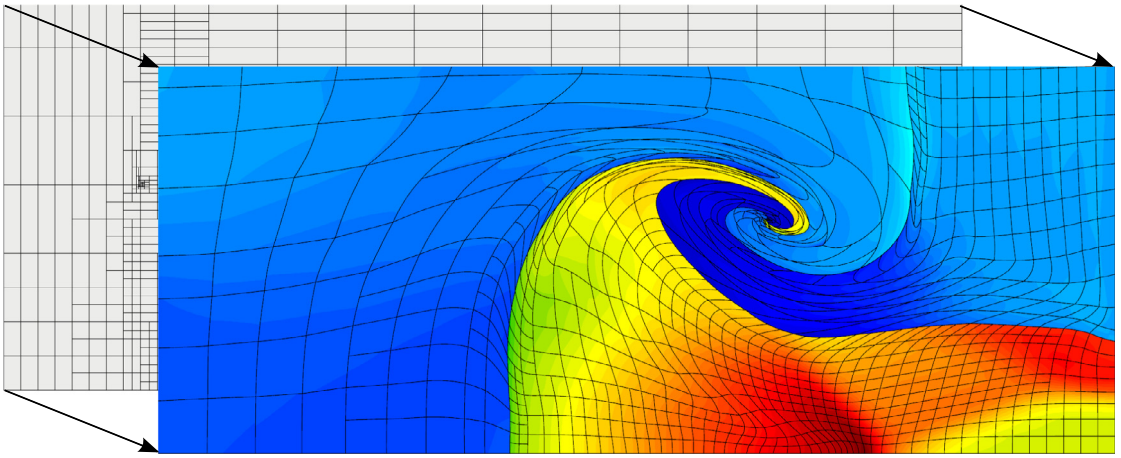


**Fig. 14.** MFEM-based static refinement in a triple point shock-interaction problem. Initial mesh at $t = 0$ (background) refined anisotropically in order to obtain more regular element shapes at target time (foreground).

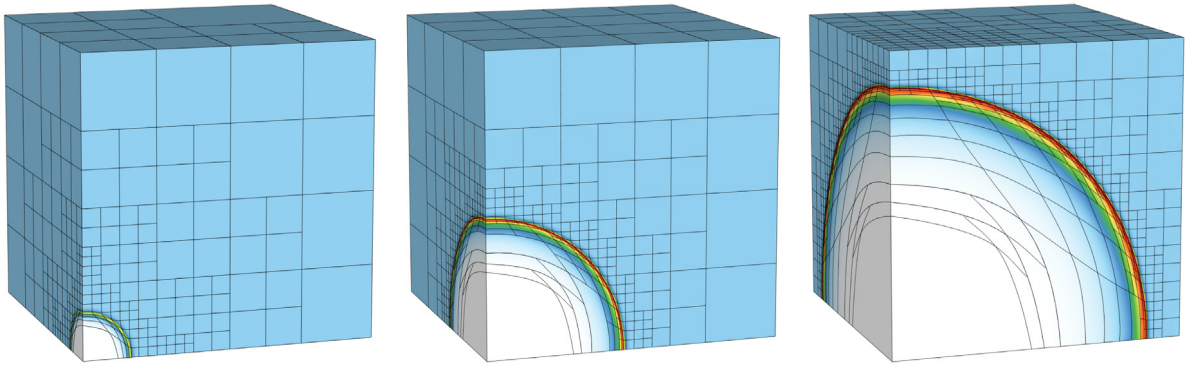


**Fig. 15.** MFEM-based dynamic refinement/derefinement in the 3D Sedov blast problem. Mesh and density shown at $t = 0.0072$ (left), $t = 0.092$ (center) and $t = 0.48$ (right). Q3Q2 elements ($p = 3$ kinematic, $p = 2$ thermodynamic quantities).

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 (current) 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:

$$F(x) := \sum_{E\in\mathcal{E}} \int_{E_t} \mu(T(x))dx_t = \sum_{E\in\mathcal{E}} \sum_{x_q\in Q_E} w_q \det(W(x_q))\,\mu(T(x_q))\,, \tag{22}$$

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

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

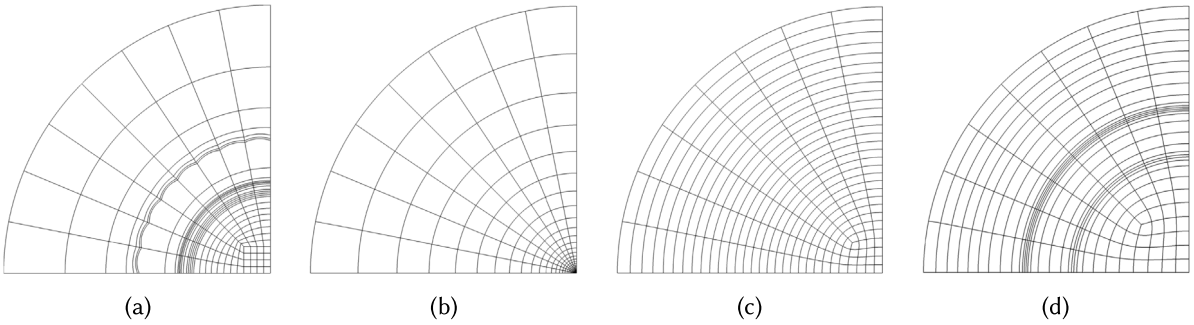



**Fig. 16.** A perturbed fourth order 2D mesh (a) is being optimized by targeting shape-only optimization (b), shape and equal size (c), and finally shape and space-dependent size (d).

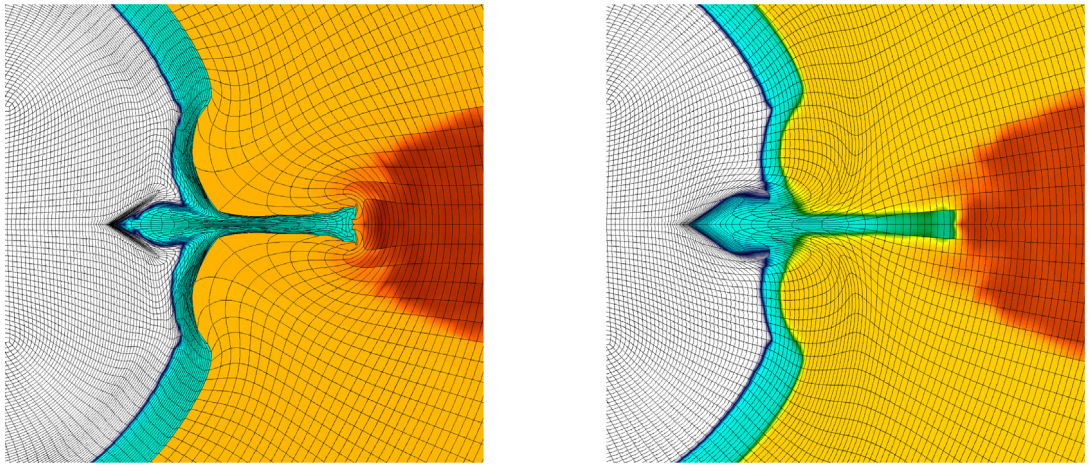


**Fig. 17.** Optimized meshes in parallel inertial confinement fusion simulation, see [82] and Section 8.3. Shown is the region around the capsule's fill tube. Both meshes are optimized with respect to shape, size, and amount of mesh displacement. On the left, material interfaces are kept fixed. On the right, interfaces are relaxed later in the simulation.

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 [81].

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`. MFEM supports the computation of matrix invariants and their first and second derivatives (with respect to the matrix), which are then used by the `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 Fig. 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 Section 8.3, and the example in Fig. 17.

Work to extend MFEM's mesh optimization capabilities to simulation-driven adaptivity (a.k.a. $r$-adaptivity) [83], and coupling $h$- and $r$-adaptivity of high-order meshes by combining the TMOP and AMR concepts is ongoing. See Fig. 18 for some preliminary results in that direction.

## 8. Applications

MFEM has been used in numerous applications and research publications, a comprehensive list of which is available on the project website at https://mfem.org/publications. In this section we illustrate a small sample of these applications.

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

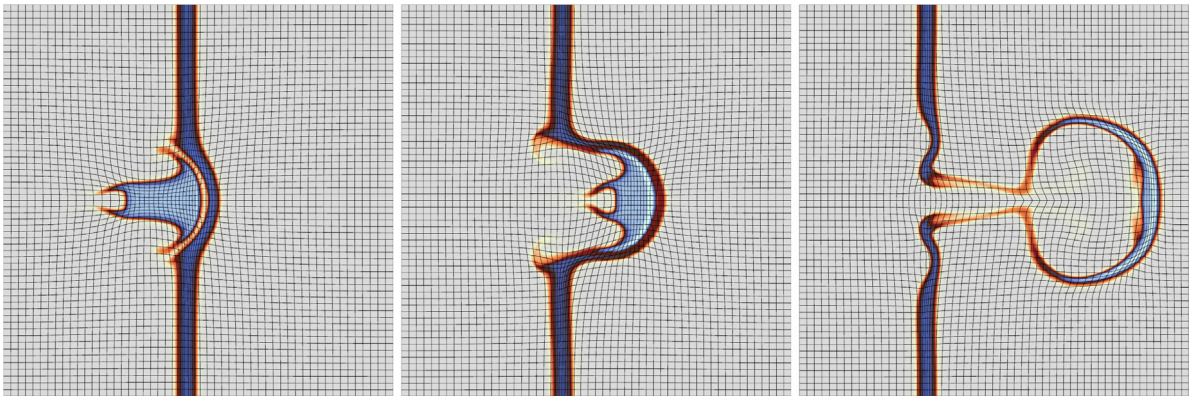


**Fig. 18.** Example of MFEM *r*-adaptivity to align the mesh with materials in a multi-material ALE simulation of high velocity gas impact, cf. [84]. Time evolution of the materials and mesh positions at times 2.5 (left), 5 (center), and 10 (right). See [83] for details.

### 8.1. Examples and miniapps

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 illustrate the straightforward transition to parallel code and the use of the *hypre* solvers and preconditioners. There are also variants of many example codes in the `petsc`, `sundials` and `pumi` subdirectories that display integration with those packages. Each example code 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, 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 Section 4.5 and https://glvis.org. Basic tutorials for running the examples can be found online at https://mfem.org under the links "Serial Tutorial" and "Parallel Tutorial".

The example codes are simply named `ex1`–`ex21`, 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. More details can be found in our online documentation at https://mfem.org/examples.

The first example `ex1` begins with the solution of the Laplace problem with homogeneous Dirichlet boundaries utilizing nodal $H^1$ elements. 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; the interface to PETSc's nonlinear solvers is described in `petsc/ex10p`, which also showcases the support for a Jacobian-free Newton Krylov approach. 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`. Time-dependent simulations are considered in examples `ex9`, `ex10`, `ex16`, and `ex17`; users interested in the usage of the SUNDIALS and PETSc ODE solvers are referred to examples `sundials/ex9p` and `petsc/ex9p` respectively. Finally, `ex11`, `ex12`, and `ex13` tackle frequency domain problems solving for eigenvalues of their respective systems. Results from some of the example runs are shown in Fig. 19.

### 8.2. Electromagnetics

The electromagnetic miniapps in MFEM are designed to provide a starting 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 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.

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

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

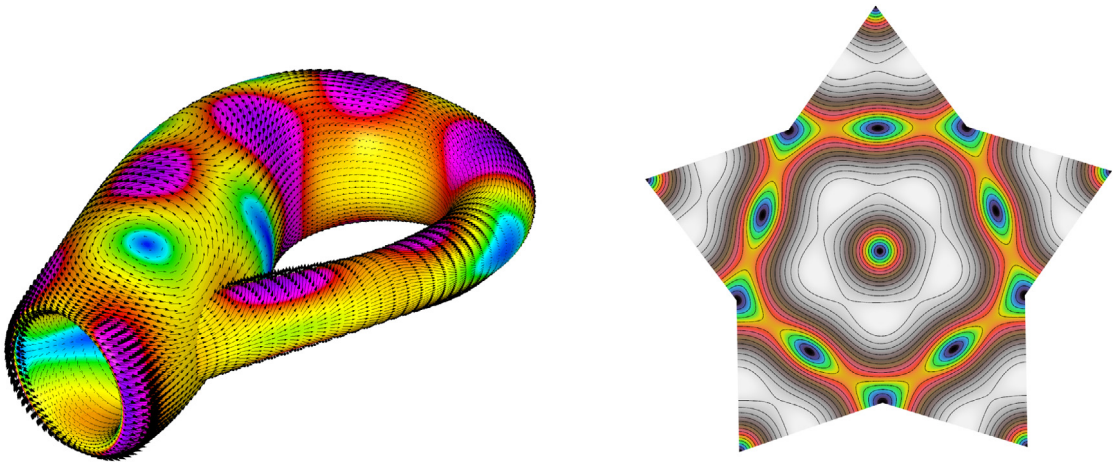


**Fig. 19.** Left: Solution of a Maxwell problem on a Klein bottle surface with `ex3`; mesh generated with the `klein-bottle` miniapp in `miniapps/meshing`. Right: An electromagnetic eigenmode of a star-shaped domain computed with 3rd order finite elements computed with `ex13p`.

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 designed to conserve energy (in the absence of lossy materials). 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 multi-physics 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* [67,68] (short for Lagrangian high-order solver) miniapp models time-dependent, compressible, inviscid gas dynamics via the Euler equations in the Lagrangian form. The Euler equations describe the conservation of mass, momentum, and energy of an inviscid fluid. In the Lagrangian setting, the elements represent regions of fluid that move with the flow, resulting in a moving and deforming mesh. The high-order curved mesh capabilities of MFEM provide a significant advantage in this context, since curved meshes can describe larger deformations more robustly than meshes using only straight segments. This in turn mitigates 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 as *partial assembly*, see Section 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 code known as *BLAST* [78,82,85], which features mesh remapping, arbitrary Lagrangian–Eulerian (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. The remap capability in *BLAST* is accomplished with a high-order discontinuous Galerkin method, which is both conservative and monotonic [86]. The DG component of the remap algorithm is very similar to the DG advection in Example 9.

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

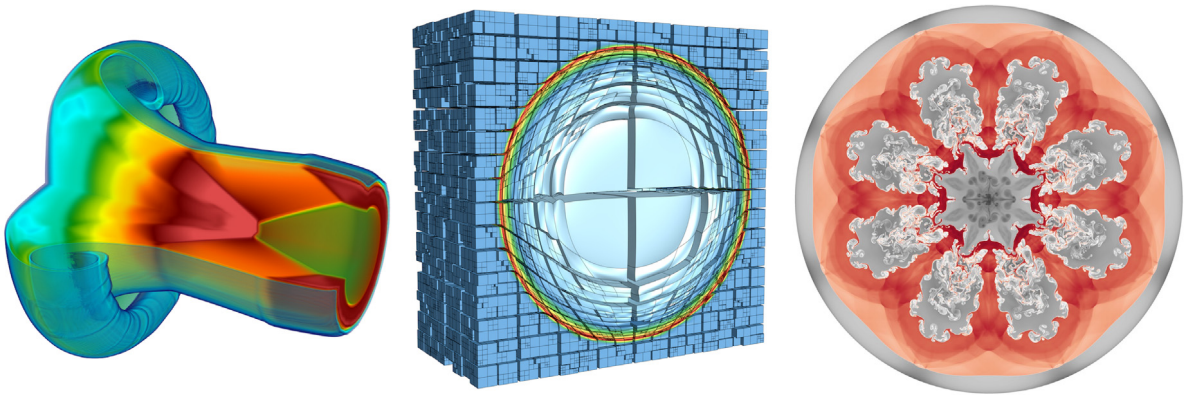


**Fig. 20.** Left: A shock interface 3D hydrodynamics calculation on 16,384 processors with BLAST. Center: Static non-conforming AMR in a Sedov blast simulation, see Section 7.2. Right: High-order axisymmetric multi-material inertial confinement fusion (ICF)-like implosion in BLAST.

*BLAST* uses a general stress tensor formulation which allows for the simulation of elasto-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 the resulting multi-material system of equations [87]. 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 Fig. 20.

### 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 projects described below.

The Cardioid project at LLNL [88] 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 Fig. 21 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 [89] and [90]. A second MFEM-based code to generate electrocardiograms using simulated electrophysiology data is also under development.

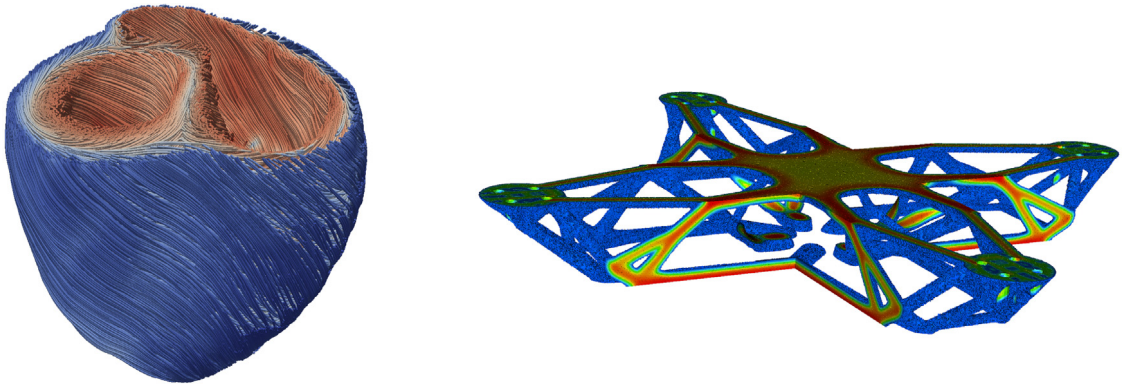


**Fig. 21.** Left: Heart fiber orientations computed in Cardioid using MFEM. Cardioid is being developed for virtual drug screening and modeling heart activity in clinical settings. Right: Drone body optimized for maximum strength in a given mass based on MFEM discretizations in LLNL's LiDO code. LiDO enables engineers to optimize immensely complex systems in HPC environments — in this case using 100 million elements, well beyond the capability of commercial software..

MFEM has also been applied to topology optimization for additive manufacturing (3D printing) by LLNL's Center for Design and Optimization, which develops the Livermore Design Optimization (LiDO) software. LiDO is used to solve challenging structural engineering problems consisting of millions of design variables. See [91] and Fig. 21. Another project that is built extensively around MFEM is ParElag [92], which is a library organized around the idea of algebraic coarsening

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

of the de Rham complex introduced in Section 4.3. ParElag leverages MFEM's high-order Lagrange, Nedelec, and Raviart–Thomas finite element spaces as well as its auxiliary space solvers (Section 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 [93,94].

## 9. Conclusions

In this paper we provided an overview of the algorithms, capabilities and applications of the MFEM finite element library as of version 4.1, released in March 2020. Our goal was to emphasize the 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 this manuscript covers all major MFEM components, it is really just an introduction to MFEM, and readers interested in learning more should consult the additional material available on the website https://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. [82,95–99].

Researchers interested in learning mathematical details about MFEM's finite element algorithms and potentially contributing to the library can follow up with [31,45,47,81,83] and the instructions/developer documentation in the `CONTRIBUTING.md` file.

## CRediT authorship contribution statement

**Robert Anderson:** Software, Investigation. **Julian Andrej:** Software, Investigation. **Andrew Barker:** Software, Investigation. **Jamie Bramwell:** Software, Investigation. **Jean-Sylvain Camier:** Methodology, Software, Investigation. **Jakub Cerveny:** Methodology, Software, Investigation. **Veselin Dobrev:** Conceptualization, Methodology, Software, Investigation. **Yohann Dudouit:** Software, Investigation. **Aaron Fisher:** Software, Investigation. **Tzanio Kolev:** Conceptualization, Methodology, Software, Investigation. **Will Pazner:** Software, Investigation. **Mark Stowell:** Software, Investigation. **Vladimir Tomov:** Methodology, Software, Investigation. **Ido Akkerman:** Methodology, Software, Investigation. **Johann Dahm:** Software, Investigation. **David Medina:** Software, Investigation. **Stefano Zampini:** Software, Investigation.

## Acknowledgments

The MFEM project would not have been possible without the help and advice of Joachim Schöberl, Panayot Vassilevski, and all the contributors in the MFEM open-source community, see https://mfem.org/about and https://github.com/orgs/mfem/people.

MFEM has been supported by a number of U.S. Department of Energy (DOE) grants, including the Applied Math and SciDAC programs in the DOE Office of Science, and the ASC and LDRD programs in NNSA. MFEM is also a major participant in the co-design Center for Efficient Exascale Discretizations (CEED) in the DOE's Exascale Computing Project.

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

## References

[1] P.G. Ciarlet, The Finite Element Method for Elliptic Problems, North-Holland, 1978.
[2] C. Johnson, Numerical Solution of Partial Differential Equations By the Finite Element Method, Cambridge University Press, 1987.
[3] L. Demkowicz, Computing with Hp-Adaptive Finite Elements. Volume I: One and Two Dimensional Elliptic and Maxwell Problems, Chapman Hall CRC, 2006.
[4] S.C. Brenner, L.R. Scott, The Mathematical Theory of Finite Element Methods, Springer New York, 2008, http://dx.doi.org/10.1007/978-0-387-75934-0.
[5] P. Solin, K. Segeth, I. Dolezel, Higher-Order Finite Element Methods, Chapman Hall CRC, 2002.
[6] M. Deville, P. Fischer, E. Mund, High-Order Methods for Incompressible Fluid Flow, Cambridge University Press, 2002.
[7] D.N. Arnold, F. Brezzi, B. Cockburn, L.D. Marini, Unified analysis of discontinuous Galerkin methods for elliptic problems, SIAM J. Numer. Anal. 39 (5) (2001) 1749–1779, http://dx.doi.org/10.1137/S0036142901384162.
[8] J. Cottrell, T. Hughes, Y. Bazilevs, Isogeometric Analysis: Toward Integration of CAD and FEA, Wiley, 2009.
[9] MFEM: Modular finite element methods, [Computer Software], 2020, http://dx.doi.org/10.11578/dc.20171025.1248, http://mfem.org.
[10] D. Arndt, W. Bangerth, D. Davydov, T. Heister, L. Heltai, M. Kronbichler, M. Maier, J.-P. Pelteret, B. Turcksin, D. Wells, The `deal.ii` library, version 8.5, J. Numer. Math. 25 (3) (2017) 137–146, http://dx.doi.org/10.1515/jnma-2016-1045.

[11] M.S. Alnæs, J. Blechta, J. Hake, A. Johansson, B. Kehlet, A. Logg, C. Richardson, J. Ring, M.E. Rognes, G.N. Wells, The FEniCS project version 1.5, Arch. Numer. Softw. 3 (100) (2015) 9–23, http://dx.doi.org/10.11588/ans.2015.100.20553.
[12] 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, O. Sander, The distributed and unified numerics environment, version 2.4, Arch. Numer. Softw. 4 (100) (2016) 13–29, http://dx.doi.org/10.11588/ans.2016.100.26526.
[13] F. Hecht, New development in FreeFem++, J. Numer. Math. 20 (3–4) (2012) 251–265.
[14] P. Solin, J. Cerveny, I. Dolezel, Arbitrary-level hanging nodes and automatic adaptivity in the hp-FEM, Math. Comput. Simulation 77 (2008) 117–132, http://dx.doi.org/10.1016/j.matcom.2007.02.011.
[15] B.S. Kirk, J.W. Peterson, R.H. Stogner, G.F. Carey, `libmesh`: A C++ library for parallel adaptive mesh refinement/coarsening simulations, Eng. Comput. 22 (3–4) (2006) 237–254, http://dx.doi.org/10.1007/s00366-006-0049-3.
[16] M. Holst, Adaptive numerical treatment of elliptic systems on manifolds, Adv. Comput. Math. 15 (2001) 139–191.
[17] J. Schöberl, C++11 Implementation of Finite Elements in NGSolve, Tech. Rep. ASC 30/2014, Institute for Analysis and Scientific Computing — Vienna University of Technology, 2014, URL https://www.asc.tuwien.ac.at/~schoeberl/wiki/publications/ngs-cpp11.pdf.
[18] D.N. Arnold, R.S. Falk, R. Winther, Differential complexes and stability of finite element methods I. The de Rham complex, in: Compatible Spatial Discretizations, Springer New York, 2006, pp. 23–46, http://dx.doi.org/10.1007/0-387-38034-5_2.
[19] 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, H. Zhang, PETSc Users Manual, Tech. Rep. ANL-95/11 - Revision 3.9, Argonne National Laboratory, 2018, URL http://www.mcs.anl.gov/petsc.
[20] X.S. Li, An overview of SuperLU, ACM Trans. Math. Software 31 (3) (2005) 302–325, http://dx.doi.org/10.1145/1089014.1089017.
[21] P. Ghysels, X.S. Li, F.-H. Rouet, S. Williams, A. Napov, An efficient multicore implementation of a novel HSS-structured multifrontal solver using randomized sampling, SIAM J. Sci. Comput. 38 (5) (2016) S358–S384, http://dx.doi.org/10.1137/15m1010117.
[22] T.A. Davis, Direct Methods for Sparse Linear Systems, Society for Industrial and Applied Mathematics, 2006, http://dx.doi.org/10.1137/1.9780898718881.
[23] A.C. Hindmarsh, P.N. Brown, K.E. Grant, S.L. Lee, R. Serban, D.E. Shumaker, C.S. Woodward, SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers, ACM Trans. Math. Softw. 31 (3) (2005) 363–396.
[24] D.A. Ibanez, E.S. Seol, C.W. Smith, M.S. Shephard, PUMI: Parallel unstructured mesh infrastructure, ACM Trans. Math. Software 42 (3) (2016) 17:1–17:28.
[25] GLVis: OpenGL finite element visualization tool, [Computer Software] https://glvis.org, http://dx.doi.org/10.11578/dc.20171025.1249.
[26] 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, P. Navrátil, VisIt: An end-user tool for visualizing and analyzing very large data, in: High Performance Visualization—Enabling Extreme-Scale Scientific Insight, CRC Press/Francis–Taylor Group, 2012, pp. 357–372.
[27] VisIt: A distributed, parallel visualization and analysis tool, [Computer Software] https://visit.llnl.gov. http://dx.doi.org/10.11578/dc.20171025.on.1019.
[28] hypre: High performance preconditioners, [Computer Software], 2018, http://www.llnl.gov/casc/hypre.
[29] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. 20 (1) (1998) 359–392, http://dx.doi.org/10.1137/S1064827595287997.
[30] METIS: Serial graph partitioning and fill-reducing matrix ordering, [Computer Software] http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.
[31] J. Cerveny, V. Dobrev, T. Kolev, Non-conforming mesh refinement for high-order finite elements, SIAM J. Sci. Comput. 41 (4) (2019) C367–C392, http://dx.doi.org/10.1137/18M1193992.
[32] D. Boffi, F. Brezzi, M. Fortin, Mixed Finite Element Methods and Applications, Springer Berlin Heidelberg, 2013, http://dx.doi.org/10.1007/978-3-642-36519-5.
[33] P. Lindstrom, The Minimum Edge Product Linear Ordering Problem, Tech. Rep. LLNL-TR-496076, Lawrence Livermore National Laboratory, 2011.
[34] D.N. Arnold, R.S. Falk, R. Winther, Finite element exterior calculus, homological techniques, and applications, Acta Numer. 15 (2006) 1–155, URL http://ima.umn.edu/arnold/papers/acta.pdf.
[35] J.C. Nedelec, Mixed finite elements in $\mathbb{R}^3$, Numer. Math. 35 (3) (1980) 315–341, http://dx.doi.org/10.1007/bf01396415.
[36] J. Brown, Efficient nonlinear solvers for nodal high-order finite elements in 3D, J. Sci. Comput. 45 (2010) 48–63, http://dx.doi.org/10.1007/s10915-010-9396-8.
[37] CEED: Center for Efficient Exascale Discretizations in the U.S. Department of Energy's Exascale Computing Project, https://ceed.exascaleproject.org.
[38] Netgen/NGSolve: A high performance multiphysics finite element software, [Computer Software] https://ngsolve.org.
[39] XYZ Scientific Applications, Inc., TrueGrid, [Computer Software], http://truegrid.com.
[40] W. Schroeder, K. Martin, B. Lorensen, The Visualization Toolkit–an Object-Oriented Approach to 3D Graphics, fourth ed., Kitware, Inc., 2006.
[41] C. Geuzaine, J.-F. Remacle, Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities, Internat. J. Numer. Methods Engrg. 79 (11) (2009) 1309–1331, http://dx.doi.org/10.1002/nme.2579.
[42] E. Shemon, C. Attaway, User Manual for EXODUS II Mesh Converter, Tech. Rep., 2014, http://dx.doi.org/10.2172/1165335.
[43] R.D. Hornung, A. Black, A. Capps, B. Corbett, N. Elliott, C. Harrison, R. Settgast, L. Taylor, K. Weiss, C. White, G. Zagaris, Axom, [Computer Software] https://github.com/LLNL/axom.
[44] Conduit: Scientific Data Exchange Library for HPC Simulations, 2014, http://dx.doi.org/10.11578/dc.20171025.1563, [Computer Software] http://software.llnl.gov/conduit.
[45] A.T. Barker, V. Dobrev, J. Gopalakrishnan, T. Kolev, A scalable preconditioner for a primal discontinuous Petrov-Galerkin method, SIAM J. Sci. Comput. 40 (2) (2018) A1187–A1203, http://dx.doi.org/10.1137/16M1104780.
[46] S. Abhyankar, J. Brown, E.M. Constantinescu, D. Ghosh, B.F. Smith, H. Zhang, PETSc/TS: A modern scalable ODE/DAE solver library, 2018, arXiv:1806.01437.
[47] V.A. Dobrev, T.V. Kolev, C.S. Lee, V.Z. Tomov, P.S. Vassilevski, Algebraic hybridization and static condensation with application to scalable H(div) preconditioning, SIAM J. Sci. Comput. 41 (3) (2019) B425–B447, http://dx.doi.org/10.1137/17M1132562.
[48] S.A. Orszag, Spectral methods for problems in complex geometries, J. Comput. Phys. 37 (1) (1980) 70–92, http://dx.doi.org/10.1016/0021-9991(80)90005-4, URL http://www.sciencedirect.com/science/article/pii/0021999180900054.
[49] S. Müthing, M. Piatkowski, P. Bastian, High-performance implementation of matrix-free high-order discontinuous Galerkin methods, 2017.
[50] M. Kronbichler, K. Kormann, Fast matrix-free evaluation of discontinuous Galerkin finite element operators, ACM Trans. Math. Software 45 (3) (2019) http://dx.doi.org/10.1145/3325864.
[51] M. Ainsworth, G. Andriamaro, O. Davydov, Bernstein–Bézier finite elements of arbitrary order and optimal assembly procedures, SIAM J. Sci. Comput. 33 (6) (2011) 3087–3109, http://dx.doi.org/10.1137/11082539x.
[52] T. Kolev, P. Vassilevski, Parallel auxiliary space AMG for H(curl) problems, J. Comput. Math. 27 (2009) 604–623, http://dx.doi.org/10.4208/jcm.2009.27.5.013, special issue on Adaptive and Multilevel Methods in Electromagnetics. UCRL-JRNL-237306.
[53] S. Zampini, PCBDDC: a class of robust dual-primal methods in PETSc, SIAM J. Sci. Comput. 38 (5) (2016) S282–S306.

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

[54] L.B. da Veiga, L.F. Pavarino, S. Scacchi, O.B. Widlund, S. Zampini, Isogeometric BDDC preconditioners with deluxe scaling, SIAM J. Sci. Comput. 36 (3) (2014) A1118–A1139.
[55] S. Zampini, D.E. Keyes, On the robustness and prospects of adaptive BDDC methods for finite element discretizations of elliptic PDEs with high-contrast coefficients, in: Proceedings of the Platform for Advanced Scientific Computing Conference, ACM, 2016, p. 6.
[56] S. Zampini, Adaptive BDDC deluxe methods for H (curl), in: Domain Decomposition Methods in Science and Engineering XXIII, Springer, Cham, 2017, pp. 285–292.
[57] S. Zampini, P. Vassilevski, V. Dobrev, T. Kolev, Balancing domain decomposition by constraints algorithms for curl-conforming spaces of arbitrary order, in: International Conference on Domain Decomposition Methods, Springer, Cham, 2017, pp. 103–116.
[58] D.-S. Oh, O. Widlund, S. Zampini, C. Dohrmann, BDDC algorithms with deluxe scaling and adaptive selection of primal constraints for Raviart–Thomas vector fields, Math. Comp. 87 (310) (2018) 659–692.
[59] S. Zampini, X. Tu, Multilevel balancing domain decomposition by constraints deluxe algorithms with adaptive coarse spaces for flow in porous media, SIAM J. Sci. Comput. 39 (4) (2017) A1389–A1415.
[60] M. Deville, E. Mund, Chebyshev pseudospectral solution of second-order elliptic equations with finite element preconditioning, J. Comput. Phys. 60 (3) (1985) 517–533, http://dx.doi.org/10.1016/0021-9991(85)90034-8, URL http://www.sciencedirect.com/science/article/pii/0021999185900348.
[61] C. Canuto, A. Quarteroni, M.Y. Hussaini, T.A. Zang, Spectral Methods, Springer Berlin Heidelberg, 2007, http://dx.doi.org/10.1007/978-3-540-30728-0.
[62] W. Pazner, Efficient low-order refined preconditioners for high-order matrix-free continuous and discontinuous Galerkin methods, 2019, arXiv preprint 1908.07071. URL https://arxiv.org/abs/1908.07071.
[63] M. Franco, J.-S. Camier, J. Andrej, W. Pazner, High-order matrix-free incompressible flow solvers with GPU acceleration and low-order refined preconditioners, 2019, arXiv preprint 1910.03032. URL https://arxiv.org/abs/1910.03032.
[64] OCCA: Open Concurrent Compute Abstraction, [Computer Software] https://libocca.org.
[65] libCEED: Code for efficient extensible discretization, [Computer Software] https://github.com/CEED/libCEED.
[66] R.D. Hornung, J.A. Keasler, The RAJA Portability Layer: Overview and Status, Tech. Rep. LLNL-TR-661403, LLNL, 2014.
[67] Laghos: LAGrangian High-Order Solver, [Computer Software] https://github.com/CEED/Laghos.
[68] V. Dobrev, J. Dongarra, J. Brown, P. Fischer, A. Haidar, I. Karlin, T. Kolev, M. Min, T. Moon, T. Ratnayaka, S. Tomov, V. Tomov, CEED ECP Milestone Report: Identify Initial Kernels, Bake-Off Problems (Benchmarks) and Miniapps, Tech. Rep. WBS 1.2.5.3.04, Milestone CEED-MS6, 2017, http://dx.doi.org/10.5281/zenodo.2542333, URL https://zenodo.org/record/2542333.
[69] P. Fischer, M. Min, T. Rathnayake, S. Dutta, T. Kolev, V. Dobrev, J.S. Camier, M. Kronbichler, T. Warburton, K. Swirydowicz, J. Brown, Scalability of high-performance PDE solvers, Int. J. High Perform. Comput. Appl. (2020) in print.
[70] A. Heinecke, G. Henry, M. Hutchinson, H. Pabst, LIBXSMM: Accelerating small matrix multiplications by runtime code generation, in: SC16: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2016, http://dx.doi.org/10.1109/sc.2016.83.
[71] D.N. Arnold, A. Mukherjee, L. Pouly, Locally adapted tetrahedral meshes using bisection, SIAM J. Sci. Comput. 22 (2) (2000) 431–448, http://dx.doi.org/10.1137/S1064827597323373.
[72] L. Demkowicz, J. Oden, W. Rachowicz, O. Hardy, Toward a universal h-p adaptive finite element strategy, part 1. constrained approximation and data structure, Comput. Methods Appl. Mech. Engrg. 77 (1) (1989) 79–112, http://dx.doi.org/10.1016/0045-7825(89)90129-1, URL http://www.sciencedirect.com/science/article/pii/0045782589901291.
[73] T. Schönfeld, Adaptive mesh refinement methods for three-dimensional inviscid flow problems, Int. J. Comput. Fluid Dyn. 4 (3–4) (1995) 363–391, http://dx.doi.org/10.1080/10618569508904530, URL http://iahr.tandfonline.com/doi/abs/10.1080/10618569508904530.
[74] V. Heuveline, F. Schieweck, H1-interpolation on quadrilateral and hexahedral meshes with hanging nodes, Computing 80 (3) (2007) 203–220, http://dx.doi.org/10.1007/s00607-007-0233-3.
[75] S. Aluru, F. Sevilgen, Parallel domain decomposition and load balancing using space-filling curves, in: Proceedings Fourth International Conference on High-Performance Computing, IEEE Comput. Soc, 1997, http://dx.doi.org/10.1109/hipc.1997.634498.
[76] E.G. Boman, U.V. Catalyurek, C. Chevalier, K.D. Devine, The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring, Sci. Program. 20 (2) (2012) 129–150.
[77] O. Sahni, X. Luo, K. Jansen, M. Shephard, Curved boundary layer meshing for adaptive viscous flow simulations, Finite Elem. Anal. Des. 46 (1) (2010) 132–139.
[78] V.A. Dobrev, T.V. Kolev, R.N. Rieben, High-order curvilinear finite element methods for Lagrangian hydrodynamics, SIAM J. Sci. Comput. 34 (5) (2012) B606–B641, http://dx.doi.org/10.1137/120864672.
[79] X.-J. Luo, M. Shephard, L.-Q. Lee, L. Ge, C. Ng, Moving curved mesh adaptation for higher-order finite element simulations, Eng. Comput. 27 (1) (2011) 41–50, http://dx.doi.org/10.1007/s00366-010-0179-5.
[80] P. Knupp, Introducing the target-matrix paradigm for mesh optimization by node movement, Eng. Comput. 28 (4) (2012) 419–429.
[81] V. Dobrev, P. Knupp, T. Kolev, K. Mittal, V. Tomov, The Target-Matrix Optimization Paradigm for high-order meshes, SIAM J. Sci. Comput. 41 (1) (2019) B50–B68.
[82] R.W. Anderson, V.A. Dobrev, T.V. Kolev, R.N. Rieben, V.Z. Tomov, High-order multi-material ALE hydrodynamics, SIAM J. Sci. Comput. 40 (1) (2018) B32–B58, http://dx.doi.org/10.1137/17M1116453.
[83] V. Dobrev, P. Knupp, T. Kolev, V. Tomov, Towards simulation-driven optimization of high-order meshes by the Target-Matrix Optimization Paradigm, in: X. Roca, A. Loseille (Eds.), 27th International Meshing Roundtable, in: Lecture Notes in Computational Science and Engineering, vol. 127, Springer International Publishing, Cham, 2019, pp. 285–302, http://dx.doi.org/10.1007/978-3-030-13992-6_16.
[84] A. Barlow, R. Hill, M.J. Shashkov, Constrained optimization framework for interface-aware sub-scale dynamics closure model for multimaterial cells in Lagrangian and arbitrary Lagrangian-Eulerian hydrodynamics, J. Comput. Phys. 276 (2014) 92–135.
[85] V.A. Dobrev, T. Kolev, R. Rieben, High order curvilinear finite elements for elastic–plastic Lagrangian dynamics, J. Comput. Phys. 257 (2014) 1062–1080.
[86] R.W. Anderson, V.A. Dobrev, T.V. Kolev, R.N. Rieben, Monotonicity in high-order curvilinear finite element ALE remap, Internat. J. Numer. Methods Fluids 77 (5) (2014) 249–273, http://dx.doi.org/10.1002/fld.3965, URL https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.3965.
[87] V.A. Dobrev, T.V. Kolev, R.N. Rieben, V.Z. Tomov, Multi-material closure model for high-order finite element Lagrangian hydrodynamics, Internat. J. Numer. Methods Fluids 82 (10) (2016) 689–706.
[88] J. Cranford, D. Richards, X. Zhang, S. Laudenschlager, J. Glosli, T. O'Hara, A. Mirin, E. Draeger, J.-L. Fattebert, R. Blake, T. Oppelstrup, Cardioid, 2018, http://dx.doi.org/10.11578/DC.20181218.2, https://github.com/llnl/cardioid.
[89] J.D. Bayer, R.C. Blake, G. Plank, N.A. Trayanova, A novel rule-based algorithm for assigning myocardial fiber orientation to computational heart models, Ann. Biomed. Eng. 40 (10) (2012) 2243–2254.
[90] V. Gurev, P. Pathmanathan, J.-L. Fattebert, H.-F. Wen, J. Magerlein, R.A. Gray, D.F. Richards, J.J. Rice, A high-resolution computational model of the deforming human heart, Biomech. Model. Mechanobiol. 14 (4) (2015) 829–849.
[91] A. Heller, Leading a revolution in design, Sci. Technol. Rev. (2018) 4–11.
[92] ParElag: Element agglomeration algebraic multigrid and upscaling, [Computer Software], 2018, https://www.github.com/llnl/parelag.

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.

[93] I.V. Lashuk, P.S. Vassilevski, The construction of coarse de Rham complexes with improved approximation properties, Comput. Methods Appl. Math. 14 (2) (2014) 257–303.
[94] J.E. Pasciak, P.S. Vassilevski, Exact de Rham sequences of spaces defined on macro-elements in two and three spatial dimensions, SIAM J. Sci. Comput. 30 (5) (2008) 2427–2446.
[95] D.A. White, M. Stowell, D.A. Tortorelli, Topological optimization of structures using Fourier representations, Struct. Multidiscip. Optim. (2018) 1–16.
[96] A. Mazuyer, P. Cupillard, R. Giot, M. Conin, Y. Leroy, P. Thore, Stress estimation in reservoirs using an integrated inverse method, Comput. Geosci. 114 (2018) 30–40, http://dx.doi.org/10.1016/j.cageo.2018.01.004, URL http://www.sciencedirect.com/science/article/pii/S0098300417305010.
[97] M. Holec, J. Limpouch, R. Liska, S. Weber, High-order discontinuous Galerkin nonlocal transport and energy equations scheme for radiation hydrodynamics, Internat. J. Numer. Methods Fluids 83 (10) (2016) 779–797, http://dx.doi.org/10.1002/fld.4288, https://onlinelibrary.wiley.com/doi/pdf/10.1002/fld.4288..
[98] M. Reberol, B. Lévy, Computing the distance between two finite element solutions defined on different 3D meshes on a GPU, SIAM J. Sci. Comput. 40 (1) (2018) C131–C155, http://dx.doi.org/10.1137/17M1115976.
[99] A.T. Barker, C.S. Lee, P.S. Vassilevski, Spectral upscaling for graph Laplacian problems with application to reservoir simulation, SIAM J. Sci. Comput. 39 (5) (2017) S323–S346, http://dx.doi.org/10.1137/16M1077581.

Please cite this article as: R. Anderson, J. Andrej, A. Barker et al., MFEM: A modular finite element methods library, Computers and Mathematics with Applications (2020), https://doi.org/10.1016/j.camwa.2020.06.009.