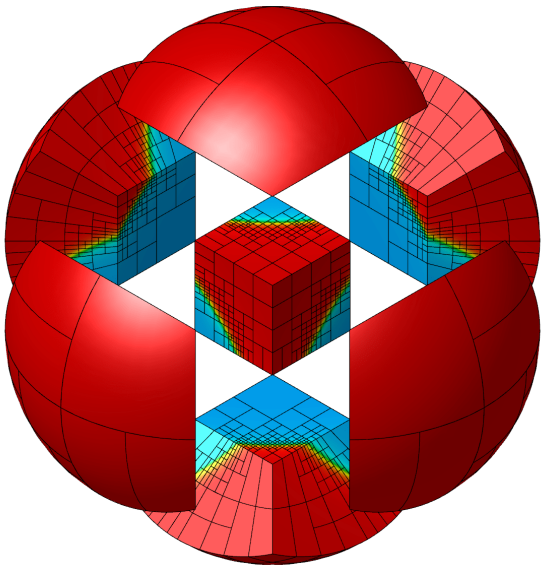
NOE REYES RIVAS

A GUIDE TO MFEM



Preface

Approximation has, in one form or another, been a part of human culture for thousands of years. The ancient Egyptians, for example, used a simple form of linear interpolation to estimate the area of a circle. The ancient Greeks used a similar method to estimate the value of π. In the 17th century, Isaac Newton and Gottfried Wilhelm Leibniz independently developed calculus, which allowed them to approximate the area under a curve by summing the areas of rectangles. In the 18th century, Leonhard Euler developed the method of least squares, which allowed him to approximate the orbits of planets. In the 19th century, Carl Friedrich Gauss developed the method of finite differences, which allowed him to approximate the solution of differential equations. In the 20th century, John von Neumann and Stanisław Ulam developed the method of Monte Carlo, which allowed them to approximate the solution of partial differential equations. In the 21st century, the *method of finite elements* or the *finite element method* has become a popular method for approximating the solution of partial differential equations.

MFEM[1] is a free, open-source, lightweight, scalable C++ library for the finite element method. It is designed to be fast, accurate, efficient, portable, parallel, and scalable. MFEM was once described as simply "a really fast linear system solver." The current method for learning MFEM is through its plentiful thorough and well-documented examples. As well-documented as they are however, such a learning strategy may seem daunting to some. This guide aims to provide a more structured and comprehensive introduction to MFEM, and is intended for students, researchers, and practitioners who are new to MFEM and the finite element method.

[1] mfem. MFEM: Modular finite element methods [Software]. https://mfem.org

The goals of this work are to give a thorough introduction to the tools provided by MFEM and to expose the reader to functionality perhaps not previously encountered. A few *anti-goals* of this work are: to provide an end-all resource for MFEM and to substitute for the official documentation.

We assume the reader is familiar with C++ and has some experience with numerical methods; a good resource for learning C++

is Lippman, Lajoie, and Moo's book[2]. The reader is encouraged to follow along with the examples in this guide by running provided code snippets into their text editor. Furthermore, we use 🗋 and 🗀 to denote a file and folder, respectively, included in the MFEM repository, e.g. 🗋 `examples/ex0.cpp` and 🗀 `examples`. Lastly, the reader is encouraged to visit the Appendix for information that may be useful for setting up their development environment.

[2] Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo. *C++ Primer*. Addison-Wesley, Upper Saddle River, NJ, 5 edition, 2013. ISBN 9780321714114

List of Listings

List of Tables

Contents

Introduction

Before we begin our investigation of MFEM, we first discuss the strong and weak forms of partial differential equations, Galerkin's method, and then include a general discussion on the Finite Element Method. This chapter may be skipped by readers who are already well-familiar with these concepts.

Strong Forms, Weak Forms

We will first discuss the general form of a PDE and then introduce the Finite Element Method as a method to solve these equations. Several definitions are in order: let X be a normed space. A bilinear form $b : X \times X \to \mathbb{R}$ is a function that is linear in both of its arguments. We say that b is *bounded* if there exists a constant $\alpha > 0$ such that

$$|b(u, v)| \leq \alpha \|u\|_X \|v\|_X \quad \text{for all} \quad u, v \in X,$$

and we say that b is *coercive* if there exists a constant $\beta > 0$ such that

$$\beta \|u\|_X^2 \leq b(u, u) \quad \text{for all} \quad u \in X.$$

The Lebesgue space $L^p(X)$ is the space of measurable functions $f : X \to \mathbb{R}$ such that

$$\|f\|_{L^p(X)} = \left(\int_X |f(x)|^p \, \mathrm{d}x \right)^{1/p} < \infty,$$

and the Sobolev space $W^{k,p}(X)$ is the space of functions f such that f has weak derivatives of order k in $L^p(X)$. Finally, we define $H^k(X) = W^{k,2}(X)$.

A fundamental PDE, often used as a toy-example, is *Poisson's equation*. First, we define the domain $\Omega \subset \mathbb{R}^d$ and its boundary $\partial\Omega$. For a given functions f, we seek to find a function u such that

$$\begin{aligned} -\Delta u &= f \qquad \text{in } \Omega, \\ u &= 0 \qquad \text{on } \partial\Omega. \end{aligned} \tag{1}$$

Of course, such a task may be impossible, as we have made no assumptions on the function f and the solution u. Thus, we must find conditions that make this problem *well-posed*—that is, so that

- a solution exists,
- the solution is unique, and
- the solution depends continuously on the data.

We will not derive these conditions here, and instead simply state them; the reader is encouraged to consult e.g. Evans[3] for a more detailed discussion. For Poisson's equation, we require that $f \in L^2(\Omega)$.

[3] L.C. Evans. *Partial Differential Equations*. Graduate studies in mathematics. American Mathematical Society, 2010. ISBN 9780821849743

As stated, (1) is the *strong form* of the PDE. To solve it with the FEM, we must first convert it to the *weak form*. This is typically done by multiplying both sides of the equation by a (sufficiently smooth) test function v and integrating over the domain Ω:

$$\int_\Omega -v\Delta u \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x.$$

We can then use integration by parts to move some derivatives from u to v,

$$\int_\Omega -v\Delta u \,\mathrm{d}x = \int_\Omega \nabla v \cdot \nabla u \,\mathrm{d}x - \int_{\partial\Omega} v\frac{\partial u}{\partial n} \,\mathrm{d}S,$$

where $\frac{\partial u}{\partial n} := \nabla u \cdot n$ is the normal derivative of u on the boundary $\partial\Omega$. *If we assume*[4] $v = 0$ *on* $\partial\Omega$, we can then rewrite the weak form of Poisson's equation as

[4] It's reasonable to make this assumption, as $u = 0$ on $\partial\Omega$ in (1).

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x \quad \text{for all} \quad v \in H_0^1(\Omega), \tag{2}$$

where

$$H_0^1(\Omega) := \{v \in H^1(\Omega) \mid v = 0 \text{ on } \partial\Omega\}$$

is defined in the trace sense. In order to show that (2) is well-posed, we require the following theorem[5].

[5] L.C. Evans. *Partial Differential Equations*. Graduate studies in mathematics. American Mathematical Society, 2010. ISBN 9780821849743

Theorem (Lax-Milgram)**.** *Let $\mathcal{H}$ be a Hilbert space and $\mathcal{H}^*$ denote its dual. Suppose $a : \mathcal{H} \times \mathcal{H} \to \mathbb{R}$ is a bounded and coercive bilinear form and $b \in \mathcal{H}^*$. Then there exists a unique element $u \in \mathcal{H}$ such that*

$$a(u, v) = b(v) \quad \textit{for all} \quad v \in \mathcal{H}.$$

Through Lax-Milgram, we can show that (2) is well-posed. In this case, the bilinear form $a : H_0^1(\Omega) \times H_0^1(\Omega) \to \mathbb{R}$ and linear functional $b : H_0^1(\Omega) \to \mathbb{R}$ are given by

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x, \quad b(v) = \int_\Omega fv \,\mathrm{d}x.$$

One can verify that a, b are bounded through Cauchy-Schwarz and that a is coercive through Poincaré's inequality.

Galerkin's Method

We now have a unique solution u to (2). If we wish to obtain u computationally, we can not work with (2) directly as the Hilbert space $H_0^1(\Omega)$ is infinite-dimensional. Thus, it is natural to consider using *finite-dimensional* spaces in our approximation scheme. Galerkin's method[6] is one such method. The idea is to approximate the solution u by a function u_h in a finite-dimensional subspace $V_h \subset H_0^1(\Omega)$, where V_h is spanned by a finite number of basis functions $\{\varphi_i\}_{i=1}^N$. Here, $h > 0$ denotes the fineness of the approximation; we aim to decrease h to obtain a more accurate approximation. The approximation u_h is then given by

[6] Galerkin's method is named after the Soviet mathematician Boris Galerkin. There are a multitude of "Galerkin-type" methods, such as the Petrov-Galerkin method and Ritz-Galerkin method.

$$u_h = \sum_{i=1}^{N} c_i \varphi_i \tag{3}$$

for some coefficients $c_i \in \mathbb{R}$ and $N = N(h)$. We hence state Galerkin's method:

$$\text{Find } u_h \in V_h \text{ such that } a(u_h, v) = b(v) \text{ for all } v \in V_h. \tag{4}$$

Under what circumstances is (4) well-posed? This is a trivial matter if our approximation scheme is *conforming*—that is, if our finite-dimensional space V_h is a subset of the Hilbert space $\mathcal{H} := H_0^1(\Omega)$, which is the case in Galerkin's method. In this case, we can apply the Lax-Milgram theorem to show that (4) indeed has a unique solution $u_h \in V_h$.

If a basis $\{\varphi_i\}_{i=1}^N$ for V_h is selected, the goal is then to find the coefficients c_i in (3). To do this, choose $v = \varphi_j$ in (4) for $j = 1, \dots, N$ and substitute (3) into (4) to obtain

$$\sum_{i=1}^{N} a(\varphi_i, \varphi_j) c_i = \sum_{i=1}^{N} a(c_i \varphi_i, \varphi_j) = a\left(\sum_{i=1}^{N} c_i \varphi_i, \varphi_j \right) = b(\varphi_i),$$

where we have applied the linearity of a. Defining the *stiffness matrix* $A \in \mathbb{R}^{N \times N}$ and the *load vector*[7] $b \in \mathbb{R}^N$ as

[7] This terminology arises from the engineering field.

$$A_{ij} = a(\varphi_j, \varphi_i) \quad \text{and} \quad b_i = b(\varphi_i),$$

the problem (4) reduces to solving the linear system

$$A^T c = b,$$

where $c = (c_1, \dots, c_N) \in \mathbb{R}^N$ is the vector of coefficients.

The Finite Element Method

The Finite Element Method (FEM) is a numerical technique used to solve partial differential equations in a variety of fields, including fluid dynamics, structural mechanics, and electromagnetics. We take Demkowicz's[8] definition of the Finite Element Method as a starting point.

[8] Leszek F. Demkowicz. *Mathematical Theory of Finite Elements*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2023. DOI: 10.1137/1.9781611977738

Definition (Finite Element Method)**.** The Finite Element Method is a special case of the Galerkin method where the basis functions are constructed by "gluing" together polynomials defined on individual elements.

Thus, there are two components to the Finite Element Method: the elements and the piecewise-polynomials defined on them. For a domain $\Omega \subset \mathbb{R}^d$, we choose the elements K_i such that

- $\operatorname{int} K_i \neq \varnothing$,
- $\operatorname{int} K_i \cap \operatorname{int} K_j = \varnothing$ for $i \neq j$, and
- $\bigcup_i K_i = \Omega$.

We typically call the collection $\{K_i\} =: \mathcal{T}$ a *mesh* for Ω.

A simple choice for the elements is to use a triangulation of the domain Ω into N-dimensional simplices. In the case $N = 2$, the elements are *triangles*, and in the case $N = 3$, the elements are *tetrahedra*.

After a mesh is chosen, we are ready to define the basis functions. First, we define the space of polynomials.

Definition (Polynomial Space)**.** The real vector space $\mathcal{P}_{k,d}$ is the space of all d-variate polynomial functions of total degree at most k. In particular,

$$\mathcal{P}_{k,d} \coloneqq \operatorname{span}\left\{ x_1^{\alpha_1} \cdots x_d^{\alpha_d} \mid \alpha_1 + \cdots + \alpha_d \leq k, \alpha_i \in \mathbb{N}_0 \right\}.$$

It is not hard to show that

$$\dim \mathcal{P}_{k,d} = \binom{k+d}{d}.$$

This formula will be useful in determining the number of *degrees of freedom* in a finite element space.

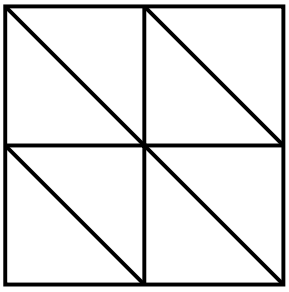


Figure 1: A triangulation of a square domain $\Omega \coloneqq (0,1)^2$. Each triangle is an element.

Using the polynomial space $\mathcal{P}_{k,d}$, we can define the *shape functions* $\psi_i = \psi_{i,K}$ on each element $K \in \mathcal{T}$ and the *basis functions* φ_i defined on Ω. We "glue" the shape functions together to form the basis functions.

Courant's Method

The shape functions are usually associated with (and defined through) the vertices of their respective elements. For reasons that will soon become apparent, we assume (unless stated otherwise) that the shape

functions are linear, i.e., $\psi_{i,K} \in \mathcal{P}_{1,d}$, and that the elements are d-simplices. Let $\{v_{j,K}\}_{j=1}^{N_K}$ be the vertices of a simplex K. The shape functions $\psi_i = \psi_{i,K}$ are chosen such that

$$\psi_i(v_{j,K}) = \delta_{ij}, \quad i,j = 1,\dots,N_K, \tag{5}$$

where δ_{ij} is the Kronecker delta. This gives rise to a system of equations: for each shape function ψ_i, we have N_K equations, one for each vertex $v_{j,K}$ of the simplex K. In particular, as each $\psi_i \in \mathcal{P}_{1,d}$ and

$$\dim \mathcal{P}_{1,d} = d + 1 = N_K, \tag{6}$$

there are the same number of equations as unknowns, and thus the system (5) is uniquely solvable. What we have described here is known as *Courant's method*[9], which laid the foundation for more general finite element methods.

[9] Richard Courant. Variational methods for the solution of problems of equilibrium and vibrations. *Bulletin of the American Mathematical Society*, 49(1): 1–23, 1943. DOI: 10.1090/s0002-9904-1943-07818-4

Now let $\{v_i\}_{i=1}^{d+1}$ be the set of all vertices in the mesh $\mathcal{T}$ and define the *finite element star*

$$D_{v_i} \coloneqq \{K \in \mathcal{T} \mid v_i \in K\}.$$

The basis functions $\varphi_i : \Omega \to \mathbb{R}$ for $i = 1,\dots,d+1$ are then defined as

$$\varphi_i(x) \coloneqq \begin{cases} \psi_{i,K}(x) & \text{if } x \in K \in D_{v_i}, \\ 0 & \text{otherwise.} \end{cases}$$

where K is the element that contains x.

There is such an element K due to our assumption that $\bigcup_i K_i = \Omega$, and *only one element* K due to our assumption that the K_i are pairwise disjoint, which makes φ_i well-defined.

Note the importance of the assumption that each shape function is linear and the elements are simplices. The crucial condition that $\dim \mathcal{P}_{1,d}$ is equal to the number of vertices in a d-simplex allowed us to obtain unique shape functions satisfying (5).

Generalizing Courant's Method

One desire in our attempt to generalize Courant's method is to allow for higher-order polynomials as shape functions and arbitrary elements (e.g. hexahedra).

We would like to extend (5) to arbitrary elements, but in order to do so, we require an equality like (6) to guarantee a unique solution to the resulting system of equations. Asserting (6) for arbitrary elements and polynomial orders by using the vertices of the element is in general not possible—take for example, a triangular element K in $\mathbb{R}^2$ and the polynomial space $\mathcal{P}_{3,2}$. Then

$$\dim \mathcal{P}_{3,2} = \binom{3+2}{2} = 10, \quad \text{but} \quad \operatorname{card} N_K = 3.$$

The element K does not possess sufficiently many vertices to define unique shape functions. However, we can rectify this issue by

injecting additional nodes into the element. For a triangular element K and the polynomial space $\mathcal{P}_{3,2}$, we require $\dim \mathcal{P}_{3,2} - \operatorname{card} N_K = 7$ additional nodes, which we place in a lattice-like manner within the element; cf. Figure 2.

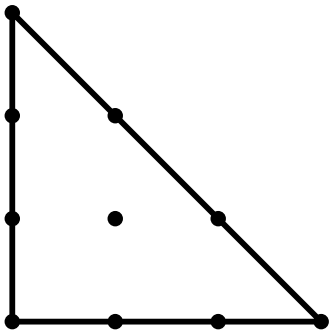


Figure 2: Langrage simplicial element of order 3 in the case $d = 2$. Note that there are $\dim \mathcal{P}_{3,2} = \binom{3+2}{2} = 10$ many nodes.

We construct the *Lagrange simplicial elements* of order k in a similar manner and acquire a set of shape functions $\psi_{i,K} \in \mathcal{P}_{k,d}$ such that

$$\psi_{i,K}(x^{(j)}) = \delta_{ij} \quad \text{for } 1 \leq i, j \leq n,$$

where $n = \dim \mathcal{P}_{k,d}$ and $x^{(1)}, \ldots, x^{(n)} \in \mathbb{R}^d$ are the nodes in a Lagrange simplicial element K.

An Example

We close this chapter by combining our results and using the FEM to solve Poisson's equation (1). We define our domain Ω to be the unit square in $\mathbb{R}^2$, our mesh $\mathcal{T}_h$ to be a triangulation of Ω with four triangles K_1, K_2, K_3, K_4, the function $f : \Omega \to \mathbb{R}$ to be $f \equiv 1$, and our finite dimensional subspace $V_h \subset H_0^1(\Omega)$ to be

In other words, we apply Courant's method.

$$V_h := \{v \in C^0(\overline{\Omega}) \mid v|_K \in \mathcal{P}_{1,2}, K \in \mathcal{T}_h \text{ and } v|_{\partial\Omega} = 0\}.$$

A visual representation of the mesh $\mathcal{T}_h$ is shown in Figure 3.

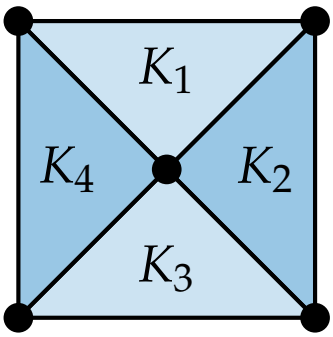


Figure 3: A triangulation of the unit square $\Omega = (0,1)^2$ with four triangles K_1, K_2, K_3, K_4.

We must find a basis $\{\varphi_i\}_{i=1}^N$ for V_h by finding the shape functions $\psi_{i,K}$ associated to each triangle K. There are a total of

$$(\dim \mathcal{P}_{1,2}) \cdot (\operatorname{card} \mathcal{T}_h) = (1+2) \cdot 4 = 12$$

shape functions,

$$\begin{aligned} &\psi_{1,K_1}, \psi_{2,K_1}, \psi_{3,K_1}, \quad \psi_{1,K_2}, \psi_{2,K_2}, \psi_{3,K_2}, \\ &\psi_{1,K_3}, \psi_{2,K_3}, \psi_{3,K_3}, \quad \psi_{1,K_4}, \psi_{2,K_4}, \psi_{3,K_4}, \end{aligned}$$

defined by $\psi_{i,K_j}(x_1, x_2) = a_{i,j} + b_{i,j}x_1 + c_{i,j}x_2$. For ease of notation, define

$$M_1 = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1/2 & 1/2 \end{bmatrix}, \quad M_2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1/2 & 1/2 \\ 1 & 1 & 0 \end{bmatrix},$$
$$M_3 = \begin{bmatrix} 1 & 1/2 & 1/2 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{bmatrix}, \quad M_4 = \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1/2 & 1/2 \\ 1 & 0 & 0 \end{bmatrix},$$

and let $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ denote the standard basis vectors for $\mathbb{R}^3$. The condition (5) implies

$$M_j \begin{bmatrix} a_{i,j} \\ b_{i,j} \\ c_{i,j} \end{bmatrix} = \mathbf{e}_i \quad \text{for } 1 \le i \le 3,\ 1 \le j \le 4.$$

After solving these systems, we find that

$$\begin{aligned} \psi_{1,K_1}(x_1, x_2) &= -x_1 + x_2, & \psi_{1,K_2}(x_1, x_2) &= -1 + x_1 + x_2, \\ \psi_{2,K_1}(x_1, x_2) &= -1 + x_1 + x_2, & \psi_{2,K_2}(x_1, x_2) &= 2 - 2x_1, \\ \psi_{3,K_1}(x_1, x_2) &= 2 - 2x_2, & \psi_{3,K_2}(x_1, x_2) &= x_1 - x_2, \\ \psi_{1,K_3}(x_1, x_2) &= 2x_2, & \psi_{1,K_4}(x_1, x_2) &= -x_1 + x_2, \\ \psi_{2,K_3}(x_1, x_2) &= 1 - x_1 - x_2, & \psi_{2,K_4}(x_1, x_2) &= 2x_1, \\ \psi_{3,K_3}(x_1, x_2) &= x_1 - x_2, & \psi_{3,K_4}(x_1, x_2) &= 1 - x_1 - x_2. \end{aligned}$$

Note that due to the boundary condition $v|_{\partial\Omega} = 0$ in the definition of V_h, there is only one basis function φ for the space V_h, namely

Hence, it was not necessary to compute all other shape functions.

$$\begin{aligned} \varphi(x_1, x_2) &= \begin{cases} \psi_{3,K_1}(x_1, x_2) & \text{if } (x_1, x_2) \in K_1, \\ \psi_{2,K_2}(x_1, x_2) & \text{if } (x_1, x_2) \in K_2, \\ \psi_{1,K_3}(x_1, x_2) & \text{if } (x_1, x_2) \in K_3, \\ \psi_{2,K_4}(x_1, x_2) & \text{if } (x_1, x_2) \in K_4 \end{cases} \\ &= \begin{cases} 2 - 2x_2 & \text{if } (x_1, x_2) \in K_1, \\ 2 - 2x_1 & \text{if } (x_1, x_2) \in K_2, \\ 2x_2 & \text{if } (x_1, x_2) \in K_3, \\ 2x_1 & \text{if } (x_1, x_2) \in K_4. \end{cases} \end{aligned} \tag{7}$$

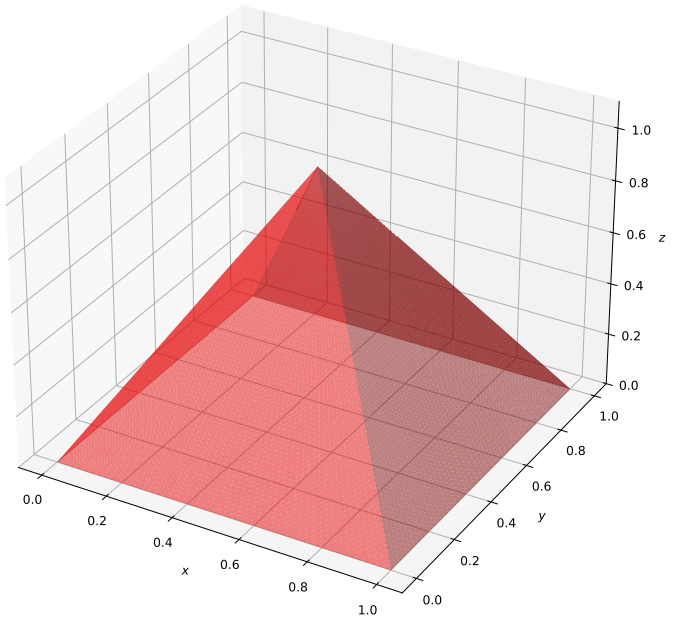


Figure 4: A graphical representation of the basis function φ in (7).

The basis function φ is shown in Figure 4.

Now that we have chosen a basis for V_h, we can compute our approximation u_h to the solution of (1). Recall that in Galerkin's method, we must find $c \in \mathbb{R}^N$ such that $A^T c = b$, where

$$A_{ij} = a(\varphi_j, \varphi_i) := \int_\Omega \nabla\varphi_j \cdot \nabla\varphi_i \,\mathrm{d}x \quad \text{and} \quad b_i = b(\varphi_i) := \int_\Omega f\varphi_i \,\mathrm{d}x,$$

for $1 \leq i, j \leq N$. In our case, we have $N = 1$ and $f \equiv 1$, so we abuse notation and write

$$A = \int_\Omega |\nabla\varphi|^2 \,\mathrm{d}x \quad \text{and} \quad b = \int_\Omega \varphi \,\mathrm{d}x.$$

As φ is defined piecewisely, we can compute these integrals as

$$A = \sum_{K\in\mathcal{T}_h} \int_K |\nabla\varphi|^2 \,\mathrm{d}x, \quad b = \sum_{K\in\mathcal{T}_h} \int_K \varphi \,\mathrm{d}x.$$

It can be shown that

$$A = 16, \quad b = \frac{1}{3},$$

so

$$c = \frac{b}{A} = \frac{1/3}{16} = \frac{1}{48},$$

and

$$u_h = \frac{1}{48}\varphi.$$

Getting Started: Return to Poisson

The goal of this chapter is to introduce several basic methods and operators provided by the MFEM library by solving (1). These methods and operators are used to perform common operations on vectors and matrices, such as taking norms, calculating dot products, and solving linear systems. We will also discuss the use of the `Coefficient` class to define functions that depend on the spatial coordinates of a mesh.

Structure of Programs

Before we start writing our first MFEM program, it is important to understand the structure of a typical MFEM program.

MFEM programs usually start with the following code:

```
#include "mfem.hpp"

using namespace std;
using namespace mfem;
```

The first line above is always necessary, as the MFEM header file `mfem.hpp` contains all the necessary declarations and definitions for using the MFEM library. The second and third lines are optional but recommended, as they allow us to use the `std` and `mfem` namespaces without having to prefix every class or function with the namespace name, making the code cleaner and easier to read.

Next, we need to define the main function of our program:

```
int main(int argc, char *argv[])
```

The parameters `argc` and `argv` are used to pass command-line arguments to the program, which can be useful for configuring the behavior of the program at runtime. We will make use of these parameters to provide different meshes and polynomial orders without having to modify the code.

The contents of `main` typically follow the structure below:

1. Parse command line options.
2. Read the mesh from the given mesh file.
3. Define a finite element space on the mesh.
4. Extract the list of all the boundary DOFs.
5. Define the solution `x` as a finite element grid function in a finite element space.
6. Set up the linear form `b` corresponding to the right-hand side.
7. Set up the bilinear form `a`.
8. Form the linear system `A X = B`.
9. Solve the system.
10. (Optional) View the solution `x`.

Command Line Options

At runtime, MFEM applications can be configured using command line options. This is useful for avoiding compiling the code every time a parameter changes. The command line options are parsed using the **`OptionsParser`** class, which is defined in `general/optparser.hpp`. We can use this class to define options of different types, such as floats, booleans, and strings. The options can be set to default values, and the user can override them by passing command line arguments when running the application.

Let's try solving (1) with a square mesh and polynomial order two. In Listing 1, we set parameters for the mesh file and polynomial order. In lines 9 and 10, we use the **`AddOption`** method to parse command line arguments, which requires a pointer to the variable that will hold the value of the argument, the short and long names of the argument, and a description[10]. The **`ParseCheck`** method checks if the given arguments are valid and prints an error message if they are not. When running our program `poisson.cpp`, we pass arguments in the following manner: `./poisson -o 2` or `./poisson --mesh ../data/ref-square.mesh`.

[10] This particular set of requirements is necessary for **`int`** and **`string`** arguments. **`bool`** arguments, for example, have more requirements.

Meshes

Now that we have chosen a mesh, we must load it in MFEM through the **`Mesh`** class. Listing 2 shows how to load a mesh from a file and refine it five times uniformly.

```cpp
// 1. Parse command line options.
string mesh_file = "../data/ref-square.mesh";
int order = 2;

OptionsParser args(argc, argv);
args.AddOption(&mesh_file, "-m", "--mesh", "Mesh file to use.");
args.AddOption(&order, "-o", "--order", "Polynomial degree.");
args.ParseCheck();
```

Listing 1: Parsing command line options using the `OptionsParser` class.

```cpp
// 2. Read the mesh from the given mesh file,
//    and refine five times uniformly.
Mesh mesh(mesh_file);
for (int k = 0; k < 5; k++)
{
   mesh.UniformRefinement();
}
```

Listing 2: Loading a mesh from a file and refining it uniformly.

See our chapter on the `Mesh` class for more details on meshes in MFEM.

Finite Element Spaces

MFEM provides a variety of finite element spaces that can be used to discretize partial differential equations (PDEs). For our purposes, we will utilize Lagrange finite elements, which are found in the `H1_FECollection` class, defined in `fem/fe_coll.hpp`.

```cpp
// 3. Define a finite element space on the mesh. Here we use H1
//    continuous high-order Lagrange finite elements of the
//    given order.
H1_FECollection fec(order, mesh.Dimension());
FiniteElementSpace fespace(&mesh, &fec);
cout << "Number of unknowns: " << fespace.GetTrueVSize() << endl;
```

We create a corresponding finite element collection variable `fec`, which requires the polynomial order and the mesh dimension, acquired through the `Dimension` method, as arguments. Lastly, we define a finite element space on the mesh using the `FiniteElementSpace` class, which takes a `Mesh` and `FiniteElementCollection` as arguments. The number of degrees of freedom in the finite element space are outputted using the `GetTrueVSize` method.

Recall that `order == 2` and `mesh.Dimension() == 2` for our example.

Boundaries and GridFunctions

It is now that we impose the boundary conditions in (1) and define the object that will hold our discrete solution. We acquire the degrees of freedom on the boundary of our mesh through the `GetBoundaryTrueDofs` method and store them in an `Array<int>` object `boundary_dofs`. This `boundary_dofs` array acts as a marker for which degrees of freedom should respect the Dirichlet boundary conditions, i.e., be set to zero, and will be used later when we assemble the linear system.

```
// 4. Extract the list of all the boundary DOFs. These will be
//    marked as Dirichlet in order to enforce zero boundary
//    conditions.
Array<int> boundary_dofs;
fespace.GetBoundaryTrueDofs(boundary_dofs);
```

Our discrete solution is stored as a `GridFunction` object, a subclass of the `Vector` class, and stores the degrees of freedom associated with the finite element space.

```
// 5. Define the solution x as a finite element grid function in
//    fespace. Set the initial guess to zero, which also sets the
//    boundary conditions.
GridFunction x(&fespace);
x = 0.0;
```

As `x` is also a `Vector`, we can write `x = 0.0` to set all its entries to zero, enforcing $u = 0$ on $\partial\Omega$ in (1).

At times, we would like to impose a Dirichlet boundary condition $u = g$ on a *subset* of the boundary, $\Gamma \subset \partial\Omega$. In this case, we can mark Γ in the mesh through the attributes of the mesh elements and use the `ProjectBdrCoefficient` method of the `GridFunction` class to set the values of the degrees of freedom on Γ to g. See our chapter on the `Mesh` class for more details.

Functions

Say we desired to implement a method to solve (1) with $f(x) = -|x|$. There are a few ways to represent f in MFEM. The first we present is the simplest and uses a std function to represent f; see Listing 3.

In line 1 of Listing 3, we're introduced to two fundamental types in MFEM, `real_t` and `Vector`. The former is a typedef for `float` or `double`, depending on whether MFEM is compiled with the

```
real_t negabs(const Vector &x)
{
  // Compute |x| (Euclidean norm)
  real_t norm = 0.0;
  for (int i = 0; i < x.Size(); i++)
  {
    norm += x(i) * x(i);
  }
  return -sqrt(norm);
}
```

Listing 3: This `negabs` function is used to define the function $f(x) = -|x|$ in MFEM. The function `sqrt` is provided by the `cmath` library, which is included in many of the MFEM source files.

`MFEM_USE_SINGLE` or `MFEM_USE_DOUBLE` preprocessor directive for precision, and is defined in `config/config.hpp`. The latter is a class that represents a vector[11] in MFEM and the type is defined in `linalg/vector.hpp`.

[11] That is, an element of a Euclidean space $\mathbb{R}^n$.

Lines 4 through 8 are readily seen to compute $-|x|$. In the `for` loop, we are introduced to one of the many methods available to the `Vector` class, the `Size` method, which returns the number of components of a `Vector`. The reader should take note that the condition `i < x.Size()` is used, as opposed to `i <= x.Size()`, as objects of type `Vector` are zero-indexed[12]. Further, we use `()` to access the components of a `Vector` in line 7; this can also be accomplished with `[]`, as in `x[i]`.

[12] The relevance of zero-indexing, of course, is not unique to the `Vector` class.

Of course, the operation of taking a norm is common enough that the MFEM developers have provided a method to do so. The `Norml2` method of the `Vector` class returns the Euclidean norm of the vector. Thus, Listing 3 can be simplified to the following.

```
real_t negabs(const Vector &x)
{
  return -x.Norml2();
}
```

Listing 4: An alternative implementation of the function $x \mapsto -|x|$; cf. Listing 3.

Taking advantage of such methods is a good practice, as it makes your code more readable, simpler, and possibly less error-prone. The `Norml2` method is a simple example of this, but the MFEM library provides many more such methods that can be used to simplify code. In our chapter on the `Vector` class, we tabulate some other common methods of the `Vector` class in Table 2.

Another way to represent f is to inherit from the `Coefficient` class[13] and define a new class, say `NegAbsCoefficient`, to represent f. The `Coefficient` class is used to represent functions, such as f, in variational formulations of PDEs. In Listing 5, we use `NegAbsCoefficient` to provide an implementation of f equivalent to Listing 3.

[13] Functions such as these were dubbed "coefficients" due to their presence as coefficients in elliptic PDEs. See our chapter on the `Coefficient` class for more information.

```
1  class NegAbsCoefficient : public Coefficient
2  {
3  public:
4     virtual real_t Eval(ElementTransformation &T, const
          IntegrationPoint &ip)
5     {
6        Vector x(T.GetDimension());
7        T.Transform(ip, x);
8
9        return -x.Norml2();
10    }
11 };
```

Listing 5: This `NegAbsCoefficient` class is used to define the function $f(x) = -|x|$ in MFEM.

In Listing 5, we're introduced to the `Eval` method in line 4. The `Eval` method takes an `ElementTransformation` and an `IntegrationPoint` as arguments and returns the value of the coefficient at the given point. The `ElementTransformation` class provides the transformation from the reference element to the physical element, and the `IntegrationPoint` class provides the coordinates of the quadrature point in the reference element, storing them in `ip`. In line 7 of Listing 5, the `Transform` method of the `ElementTransformation` class is used to map the reference point to the physical point, which is stored and represented by the `Vector` object `x`. The `Vector` `x` is initialized in line 6 with the dimension of the physical space in which the element resides, which is acquired through the `GetDimension` method of the `ElementTransformation` class. Such a definition allows us to use the same `NegAbsCoefficient` class for elements in any dimension. We note that it is not necessary to initialize `x` with a size—this can be done later with the `SetSize` method of the `Vector` class.

Some methods require inputs. The `DistanceSquaredTo` method of the `Vector` class is such an example, and as its name suggests, it returns the square of the Euclidean distance to another vector, $|x-y|^2$. Defining $x = (1,2,3), y = (1,0,0)$, we execute the following to compute $|x-y|^2$.

```
Vector x(3), y(3);

x[0] = 1.0; x[1] = 2.0; x[2] = 3.0;
y[0] = 1.0; y[1] = 0.0; y[2] = 0.0;

real_t dist = x.DistanceSquaredTo(y);
```

Listing 6: The `DistanceSquaredTo` method can be used to compute the square of the Euclidean distance between two vectors, $|x-y|^2$. Of course, it would make no difference if we wrote `y.DistanceSquaredTo(x)`.

Returning to our function f, we can use either Listing 3 or Listing 5 to use f in MFEM. If we were to use Listing 3, then we would write `FunctionCoefficient f(negabs)` within `main`; if we were to use

Listing 5, then we would write `NegAbsCoefficient f`.

Linear Forms

Using our work in the previous section, we are now ready to implement the linear form b defined $b(v) = \int_\Omega fv \, \mathrm{d}x$ in (2).

```
// 6. Set up the linear form b(.) corresponding to the right-hand
//    side.
FunctionCoefficient f(negabs);
LinearForm b(&fespace);
b.AddDomainIntegrator(new DomainLFIntegrator(f));
b.Assemble();
```

The `DomainLFIntegrator` class is used to represent our linear form b. It takes a `Coefficient` object as input, the function f in (2). The `AddDomainIntegrator` method of the `LinearForm` class is used to add the integrator to the form[14], and the `Assemble` method of the `LinearForm` class is used to assemble the vector—that is, sum over all integrators—associated with `b`, the vector with entries $b(\phi_i)$ for each basis function ϕ_i.

See our chapter on the `LinearForm` class for more details on how to implement linear forms in MFEM.

[14] "Add" is appropriate because this is literally the action taking place. In MFEM, linear and bilinear forms are added to memory, so we can sum linear forms together by calling `AddDomainIntegrator` with various integrators.

Bilinear Forms

How do we implement $-\Delta$? Recall that $-\Delta$ yields the bilinear form a defined $a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x$ in the weak formulation of Poisson's equation. It is this through the `DiffusionIntegrator` class that we represent our bilinear form a. The `DiffusionIntegrator` class inherits from `BilinearFormIntegrator`, which is the base class for all integrators that compute bilinear forms.

```
// 7. Set up the bilinear form a(.,.) corresponding to the -Delta
//    operator.
BilinearForm a(&fespace);
a.AddDomainIntegrator(new DiffusionIntegrator);
a.Assemble();
```

Listing 7: The `DiffusionIntegrator` class is used to represent the bilinear form $(u, v) \mapsto \int_\Omega \nabla u \cdot \nabla v \, \mathrm{d}x$ in the weak formulation of Poisson's equation.

The keyword `new` is used to create an instance of the `DiffusionIntegrator` class, and the `AddDomainIntegrator` method of the `BilinearForm` class is used to add the integrator to the form. The `Assemble` method of the `BilinearForm` class is used to assemble the matrix associated with the bilinear form.

We could instead define `BilinearForm* di = DiffusionIntegrator` and pass `di` to the `AddDomainIntegrator` method, but explicitly assigning memory to any particular integrator is often unnecessary.

Linear Systems

Lastly, we solve the associated linear system.

```
// 8. Form the linear system A X = B. This includes eliminating
//    boundary conditions, applying AMR constraints, and other
//    transformations.
SparseMatrix A;
Vector B, X;
a.FormLinearSystem(boundary_dofs, x, b, A, X, B);
```

The `FormLinearSystem` method takes the boundary degrees of freedom, the solution vector, the right-hand side vector, and returns the sparse matrix and the vectors for the linear system. The boundary conditions are applied, and the system is ready to be solved.

```
// 9. Solve the system using PCG with symmetric Gauss-Seidel
//    preconditioner.
GSSmoother M(A);
PCG(A, M, B, X, 1, 200, 1e-12, 0.0);
```

MFEM provides many tools to solve linear systems, including iterative solvers like the Preconditioned Conjugate Gradient (PCG) method. In this example, we use a symmetric Gauss-Seidel preconditioner to solve the system. The parameter `200` is the maximum number of iterations, `1e-12` is the relative tolerance, and `0.0` is the absolute tolerance.

Obtaining and Viewing the Solution

Using the `RecoverFEMSolution` method from the `BilinearForm` class, we can recover the finite element solution from the linear system we solved in the previous section in the `GridFunction` `x`.

```
// 10. Recover and view the solution as a finite element grid
//     function.
a.RecoverFEMSolution(X, b, x);

char vishost[] = "localhost";
int  visport   = 19916;
socketstream sol_sock(vishost, visport);
sol_sock.precision(8);
sol_sock << "solution\n" << mesh << x << flush;
```

The other half of the code above configures GLVis to visualize our solution `x`. See Figure 5 for the resulting visualization.

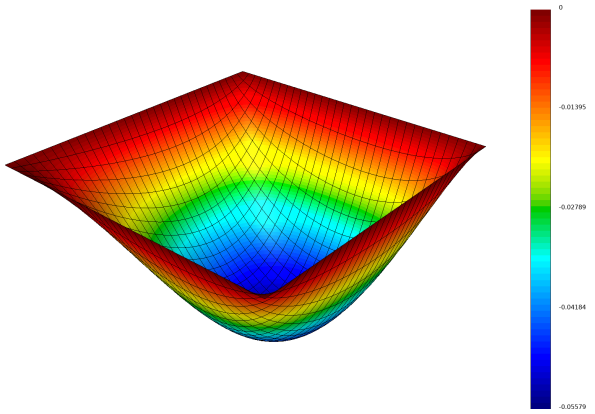


Figure 5: Discrete solution to (2) with $f : \mathbb{R}^2 \to \mathbb{R}$ defined $f(x) = -|x|$. Visualized using GLVis.

Our chapter on the `Mesh` class includes more examples of using GLVis in MFEM.

Vectors

THE GOAL OF THIS CHAPTER is introduce the `Vector` class in MFEM. `Vector` objects are commonly used in the definition of functions (i.e. `Coefficient` objects). The source for this class can be found in `linalg/vector.hpp` and `linalg/vector.cpp`.

Vector Objects

The `Vector` class in MFEM is a general-purpose vector class that can be used to represent vectors. An object of type `Vector` can be created simply by writing `Vector x`. It is not necessary to specify the size of the vector when creating it, but its size should be set before using it. The size of the vector can be set by either using the `SetSize` method, or upon allocation. For example, to create a vector of size 10, one could write `Vector x(10)` or

```
Vector x;
x.SetSize(10);
```

There too are multiple ways to populate a vector. One such way is specify its entries one-by-one through `[]` (or `()`); see Listing 8.

```
Vector x(10);
for (int i = 0; i < 10; i++)
  x[i] = i;
```

Listing 8: Setting the entries of a vector one-by-one.

Another method is to specify the entries of the vector upon allocation through an array. The code below is equivalent to the code in Listing 8; note that the size of `x` is automatically deduced from the size of the array `arr`.

```
int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
Vector x(arr);
```

One last method to populate a vector is similar to the previous method, and avoids the allocation of an array by directly passing the array upon vector allocation. As before, the code below is equivalent to the code in Listing 8.

```
Vector x({1, 2, 3, 4, 5, 6, 7, 8, 9, 10});
```

A special case of population is when we desire to initialize all entries of a vector to the same value. This can be done by passing a single value to the vector upon allocation. For example, the code below creates a vector of size 10 with all entries set to 3.

```
Vector x(10);
x = 3;
```

Vector Operators

We have been exposed to a few vector operators, such as `/` and `*` for component-wise division and the dot product, respectively. These and many other vector operators in MFEM are defined after line 285 in `linalg/vector.hpp`. There are comments that describe the purpose of each operator, so we will not list and describe them all here. Nontetheless, we will mention some of the more common ones in Table 1; their definitions are quite intuitive.

Operator	Description	Use
`*`	Inner product	`u * v`
`=`	Copy assignment	`u = v`
`+=`	Component-wise sum, assignment	`u += c`, `u += v`
`*=`	Component-wise scaling, assignment	`u *= c`, `u *= v`
`/=`	Component-wise division, assignment	`u /= c`, `u /= v`
`(),[]`	Component reference	`u(i)`, `u[i]`

Table 1: A few of the more common vector operators in MFEM. Here, `u`,`v` are of type `Vector`, `c` is of type `real_t`, and `i` is of type `int`.

Note that the operators `+=`, `*=`, and `/=` can be used with scalars *and* vectors[15]. Whenever used, the operation is performed component-wise. For example, `u += v` will add the components of `v` to the corresponding components of `u`, and `u *= 2` will multiply each component of `u` by two. The reader is cautioned that `u = u + v` is *not* equivalent to `u += v`, as the former is not even defined. Indeed, the expression `u + v` is simply not defined, and `+` *cannot* be used between objects of type `Vector`. The addition of vectors must be accomplished using a function.

[15] This is known as *overloading*, a property that MFEM makes much use of.

Vector Methods and Functions

The difference between *methods* and *functions* is that methods are applied to an object, while functions are given an object.

We will now discuss *some* of the methods that are available for the `Vector` class, which are defined in `linalg/vector.hpp`. See Table 2.

Method	Description	Use	Math
`Size`	Size of vector	`u.Size()`	NA
`Norml2`	ℓ^2 or Euclidean norm	`u.Norml2()`	$\|u\|_{\ell^2}$
`Normlinf`	ℓ^∞ norm	`u.Normlinf()`	$\|u\|_{\ell^\infty}$
`Norml1`	ℓ^1 norm	`u.Norml1()`	$\|u\|_{\ell^1}$
`Normlp`	ℓ^p norm	`u.Normlp(p)`	$\|u\|_{\ell^p}$
`Max`	Maximal element	`u.Max()`	$\max_i u_i$
`Min`	Minimal element	`u.Min()`	$\min_i u_i$
`Sum`	Sum of entries	`u.Sum()`	$\sum_i u_i$
`DistanceSquaredTo`	Euclidean distance squared	`u.DistanceSquaredTo(v)`	$\|u-v\|_{\ell^2}^2$
`DistanceTo`	Euclidean distance	`u.DistanceTo(v)`	$\|u-v\|_{\ell^2}$
`Add`	Add a scalar multiple of a vector	`u.Add(a,v)`	$u_{\text{new}} = u_{\text{old}} + av$
`Set`	Assign a scalar multiple of a vector	`u.Set(a,v)`	$u_{\text{new}} = av$
`cross3D`	Cross product	`u.cross3D(v,w)`	$u \times v = w$
`Print`	Print contents to stream out	`u.Print()`	NA
`PrintMathematica`	Print contents in Mathematica format	`u.PrintMathematica()`	NA

Table 2: Some methods available to objects of type `Vector`. Here, `u`, `v`, `w` are of type `Vector`, and `p`, `a` are of type `real_t`.

Of course,

```
u.Normlp(2) == u.Norml2()   and   u.Normlp(1) == u.Norml1().
```

In Table 3, we list some of the functions that are available for the `Vector` class, which are defined in `linalg/vector.hpp`.

Function	Description	Use	Math
`Neg`	Scalar negation	`Neg(u)`	$u_{\text{new}} = -u_{\text{old}}$
`Reciprocal`	Component-wise reciprocal	`Reciprocal(u)`	$u^i_{\text{new}} = 1/u^i_{\text{old}}$
`Swap`	Swap entries of two vectors	`Swap(u,v)`	$u_{\text{new}} = v_{\text{old}}, v_{\text{new}} = u_{\text{old}}$
`add`	Addition	`add(u,v,w)`	$u + v = w$
`subtract`	Subtraction	`subtract(u,v,w)`	$u - v = w$
`InnerProduct`	Dot product	`InnerProduct(u,v)`	$u \cdot v$

Table 3: Some functions available to objects of type `Vector`. Here, `u`, `v`, `w` are of type `Vector`.

Of course,

```
Neg(u) == (u *= -1)   and   InnerProduct(u,v) == u * v.
```

As introduced in the previous section, MFEM makes frequent use of *overloading*. This is also true in the function context. For example, there are actually *four* functions named `add`, each with a different

number (and type) of arguments. The `add` function in Table 3 is the simplest one, which takes three arguments of type `Vector`. Another version takes in three `Vector` inputs and one `real_t` input:

```
/// Set v = v1 + alpha * v2.
friend void add(const Vector &v1, real_t alpha, const Vector &v2,
    Vector &v);
```

The reader is cautioned that there are more functions named `add` than `subtract`[16]. Thus, while `subtract(u,v,w)` is a valid call, `subtract(u,a,v,w)` is not.

[16] This is true as of MFEM v4.8.

Block Vectors

We can represent block vectors in MFEM using the `BlockVector` class. A block vector is a vector that is composed of smaller vectors, or *blocks*. Consider the following block vector:

$$v = \left[\begin{array}{c} 1 \\ 2 \\ 3 \\ \hline 8 \\ 9 \end{array}\right].$$

First, we define an array which contains the offsets:

```
Array<int> offsets(3);
offsets[0] = 0;
offsets[1] = 3;
offsets[2] = 2;
offsets.PartialSum();
```

Our array `offsets` has three entries, *exactly one more* than the number of blocks in the block vector. In the last $b-1$ entries of `offsets`, where b is the number of blocks, we store the sizes of each block in the order they appear in the block vector. Next, we call the `PartialSum` method of the `BlockVector` class to convert these sizes into offsets. The `PartialSum` method modifies the array such that each entry at index i contains the sum of all previous entries from index 0 to index $i-1$. Thus, after calling `PartialSum`, we have `offsets == {0, 3, 5}`. Of course, we could have directly defined `offsets` as such, but using `PartialSum` allows us to think in terms of block sizes instead of offsets.

Next, we define and populate the underlying vector:

```
Vector w({1.0, 2.0, 3.0, 8.0, 9.0});
```

Finally, we can define the block vector using the underlying vector `w` and the array `offsets`:

```
BlockVector v(w, offsets);
```

Running the code

```
for (int i = 0; i < v.NumBlocks(); i++)
{
  Vector block;
  v.GetBlockView(i, block);
  block.Print();
}
```

will output

```
1 2 3
8 9
```

as expected.

Matrices

In this chapter, we will explore the `DenseMatrix` class. The source for this class can be found in `linalg/densemat.hpp` and `linalg/densemat.cpp`.

Matrix Objects

There are several ways to create a matrix in MFEM; see Table 4.

Function	Description
`DenseMatrix A`	Initialization
`DenseMatrix A(B)`	Initialize `A` with contents of `B`
`DenseMatrix A(n)`	Initialize `A` with size `n` by `n`
`DenseMatrix A(m,n)`	Initialize `A` with size `m` by `n`

Table 4: Some ways to intialize a matrix. Here, `B` is of type `DenseMatrix`, and `m`, `n` are of type `int`.

How do we populate matrices? The syntax is similar to the `Vector` class. Below, we use a C-style array to populate a matrix.

```
real_t values[3][3] = {
   {1, 2, 3},
   {4, 5, 6},
   {7, 8, 9}
};
DenseMatrix A(values);
```

Listing 9: Populating a `DenseMatrix` object with a C-style array.

We can also use a braced initializer list to populate a matrix.

```
DenseMatrix B({
   {1, 2, 3},
   {4, 5, 6},
   {7, 8, 9}
});
```

Listing 10: Populating a `DenseMatrix` object through a braced initializer list.

Yet another way is to use an `std::vector`.

```
std::vector<real_t> values_vec = {1, 4, 7, 2, 5, 8, 3, 6, 9};
DenseMatrix C(values_vec.data(), 3, 3);
```

In the listings above, `A == B == C`. The first three elements of `C` are the first column of `A`, and so on. This is because the matrix data is stored in a column-major order.

Matrix Operators

These and many other matrix operators in MFEM are defined after line 616 in `linalg/densemat.cpp`. There are comments that describe the purpose of each operator, so we will not list and describe them all here. Nonetheless, we will mention some of the more common ones in Table 5; their definitions are quite intuitive.

Operator	Description	Use
`*`	(Frobenius) Inner product	`A * B`
`=`	Copy assignment	`A = c`, `A = B`
`+=`	Entry-wise sum, assignment	`A += c`, `A += B`
`-=`	Entry-wise difference, assignment	`A -= B`
`*=`	Entry-wise scaling, assignment	`A *= c`, `A *= B`
`()`	Entry reference	`A(i,j)`

Table 5: A few of the more common vector operators in MFEM. Here, `A` and `B` are of type `DenseMatrix`, `i`, `j` are of type `int`, and `c` is of type `real_t`. Recall that the Frobenius inner product of two matrices A, B is defined as $A : B = \text{tr}(A^T B) = \sum_{i,j} A_{ij} B_{ij}$.

Matrix Methods and Functions

We will now discuss *some* of the methods that are available for the `DenseMatrix` class, which are defined in `linalg/densemat.hpp`, and make a few comments regarding some of the methods listed in Table 6:

- The `Size` and `Width` methods are synonymous.
- The `Mult` method supports overloading and can accept types other than `Vector`, such as `real_t`.
- There are also `RightScaling` and `InvRightScaling` methods.
- The `Exponential` method currently only supports 2×2 matrices.
- MFEM can compute (generalized) eigenvectors and eigenvalues[17], assuming the user has compiled MFEM with LAPACK.

There are many methods we did not list here, including the "copy"[18] methods.

[17] See line 275 in `linalg/densemat.hpp`.

[18] See line 368 in `linalg/densemat.hpp`.

Method	Description	Use	Math
`SetSize`	Resize	`A.SetSize(h)`, `A.SetSize(h,w)`	NA
`Height`	Get height	`A.Height()`	NA
`Width`	Get width	`A.Width()`	NA
`Mult`	MVM	`A.Mult(x,y)`	$Ax = y$
`MultTranspose`	TMVM	`A.MultTranspose(x,y)`	$A^T x = y$
`AddMult`	Add scalar multiple of MVM	`A.AddMult(x,y,c)`	$y_{\text{new}} = y_{\text{old}} + cAx$
`AddMultTranspose`	Add scalar multiple of TMVM	`A.AddMultTranspose(x,y,c)`	$y_{\text{new}} = y_{\text{old}} + cA^T x$
`InnerProduct`	Inner product	`A.InnerProduct(x,y)`	$y^T Ax, Ax \cdot y$
`LeftScaling`	Scale by diagonal matrix	`A.LeftScaling(x)`	$A_{\text{new}} = \text{Diag}(x)A_{\text{old}}$
`InvLeftScaling`	Scale by inverse diagonal matrix	`A.InvLeftScaling(x)`	$A_{\text{new}} = \text{Diag}(1/x)A_{\text{old}}$
`Trace`	Compute trace	`A.Trace()`	$\sum_i a_{ii}$
`Inverse`	Compute matrix inverse	`A.Invert()`	$A_{\text{new}} = A_{\text{old}}^{-1}$
`Exponential`	Compute matrix exponential	`A.Exponential()`	$A_{\text{new}} = \exp A_{\text{old}}$
`Set`	Assign scalar multiple of matrix	`A.Set(c,B)`	$A_{\text{new}} = cB$
`Add`	Add scalar multiple of matrix	`A.Add(c,B)`	$A_{\text{new}} = A_{\text{old}} + cB$
`Neg`	Scalar negation	`A.Neg()`	$A_{\text{new}} = -A_{\text{old}}$
`MaxMaxNorm`	Compute ℓ^1 norm	`A.MaxMaxNorm()`	$\max_{ij} \lvert a_{ij} \rvert$
`FNorm`	Compute Frobenius norm	`A.FNorm()`	$\lVert A \rVert_{\text{F}}$
`Diag`	Create diagonal matrix	`A.Diag(c,n)`	$A = \text{Diag}(c)$
`Transpose`	Transpose	`A.Transpose()`, `B.Transpose(A)`	$A_{\text{new}} = A_{\text{old}}^T, B = A^T$
`Symmetrize`	Symmetrize matrix	`A.Symmetrize()`	$A_{\text{new}} = (A_{\text{old}} + A_{\text{old}}^T)/2$
`PrintMatlab`	Print matrix to stdout	`A.PrintMatlab()`	NA

Table 6: Some methods available to objects of type `DenseMatrix`. Here, `A` is of type `DenseMatrix`, `x`, `y` are of type `Vector`, `h`, `w` are of type `int`, and `c` is of type `real_t`. "(T)MVM" stands for "(transpose) matrix-vector multiplication."

We will now discuss *some* of the functions that are available for the `DenseMatrix` class, which are defined in `linalg/densemat.hpp`; see Table 7.

Function	Description	Use	Math
`Add`	Linear combination assignment	`Add(alpha,A,beta,B,C)`	$\alpha A + \beta B = C$
`Mult`	Matrix-matrix multiplication	`Mult(A,B,C)`	$AB = C$
`AddMult`	"	`AddMult(A,B,C)`	$C_{\text{new}} = C_{\text{old}} + AB$
`CalcInverse`	Compute (left) inverse	`CalcInverse(A,B)`	$B = A^{-1}$
`CalcInverseTranspose`	Compute inverse transpose	`CalcInverseTranspose(A,B)`	$B = (A^{-1})^T$

Table 7: Some functions available to objects of type `DenseMatrix`. Here, `A`, `B`, `C` are of type `DenseMatrix`, and `alpha`, `beta` are of type `real_t`.

There are many methods we did not list here, including the many "add" and "multiply"[19] functions.

[19] See line 558 in `linalg/densemat.hpp`.

Block Matrices

There is not much novelty in the `BlockMatrix` class when compared to the `SparseMatrix` class. Consider the following block matrix:

$$M = \left[\begin{array}{c|c} A & C \\ \hline B & D \end{array}\right] = \left[\begin{array}{ccc|cc} 1 & & & 1.2 & \\ & 2 & & & 2.2 \\ & & 3 & & \\ \hline & 1.5 & & 4 & \\ & & 2.5 & & 5 \end{array}\right].$$

We can represent M in a few different ways. One way is to specify the blocks *row-wise*[20]. First, we define an array which contains the row-wise offsets:

[20] The row-wise definition of block matrices is in fact the default in MFEM.

```
Array<int> offsets(3);
offsets[0] = 0;
offsets[1] = 3;
offsets[2] = 5;
```

Our array `offsets` has three entries, *exactly one more* than the number of block-rows in the block matrix. The first entry is the starting index of the first row, the second entry is the starting index of the second row, and so on. The last entry is the total number of rows in the matrix. Next, we can define and populate the blocks:

```
SparseMatrix A(3,3);
SparseMatrix B(2,3);
SparseMatrix C(3,2);
SparseMatrix D(2,2);

A.Set(0, 0, 1.0); A.Set(1, 1, 2.0); A.Set(2, 2, 3.0);
B.Set(0, 1, 1.5); B.Set(1, 2, 2.5);
C.Set(0, 0, 1.2); C.Set(1, 1, 2.2);
D.Set(0, 0, 4.0); D.Set(1, 1, 5.0);
```

Finally, we can create the block matrix:

```
BlockMatrix M(offsets);
M.SetBlock(0, 0, &A);
M.SetBlock(0, 1, &C);
M.SetBlock(1, 0, &B);
M.SetBlock(1, 1, &D);
M.Finalize();
```

Note that in setting the blocks, we need to pass the block by reference.

Meshes

In this chapter, we discuss meshes. Meshes are a fundamental part of any FEM application, as they define the geometry and topology of the problem domain. In MFEM, meshes are represented by the `Mesh` class, whose source can be found in `mesh/mesh.hpp` and `mesh/mesh.cpp`.

Mesh Definitions

MFEM comes with a variety of mesh formats that can be read, and they can all be found within `data`. For example, `data/disc-nurbs.mesh` corresponds to a mesh of a disc with a NURBS boundary. We can visualize this mesh through GLVis by running `glvis -m data/disc-nurbs.mesh` in the commandline.

It is not uncommon that we wish to perform certain transformations to our mesh. The following code snippet[21] scales this mesh by a factor of $1/(2\sqrt{2})$ and in effect, transforms the disc into a unit disc.

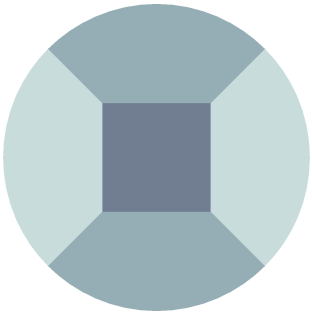


Figure 6: The mesh `data/disc-nurbs.mesh` visualized using GLVis.

[21] This code was taken from `examples/ex36.cpp`.

```
const char[] mesh_file = "../data/disc-nurbs.mesh";
Mesh mesh(mesh_file, 1, 1);
GridFunction *nodes = mesh.GetNodes();
*nodes /= 2*sqrt(2);
```

We can also create a mesh from scratch within MFEM. The following code snippet creates a 3D hexahedral mesh with dimensions $1 \times 1 \times 1$.

The already-present `data/ref-cube.mesh` produces the same mesh as in the following code snippet.

```
Mesh mesh = Mesh::MakeCartesian3D(
  /* nx */ 1, /* ny */ 1, /* nz */ 1,
  /* type */ Element::HEXAHEDRON,
  /* sx */ 1.0, /* sy */ 1.0, /* sz */ 1.0
);
```

The parameters `nx`, `ny`, and `nz` correspond to the number of elements in the x, y, and z directions, respectively. The `type` parameter specifies

the type of element to be used, which in this case is a hexahedron. Finally, the parameters `sx`, `sy`, and `sz` specify the size of the mesh in each direction; the hexadron represented mathematically is $[0, \texttt{sx}] \times [0, \texttt{sy}] \times [0, \texttt{sz}]$.

The `MakeCartesian3D` function, in general, creates a mesh for a parallelepiped.

Exporting a Mesh

We can export this mesh to a file `unit-hexahedron.mesh` using the `Save` method: `mesh.Save("unit-hexahedron.mesh")`. As before, we can visualize this mesh using GLVis by running `glvis -m unit-hexahedron.mesh` in the commandline.

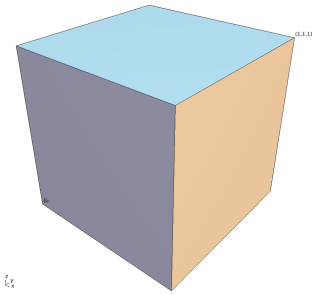


Figure 7: The mesh `unit-hexahedron.mesh` visualized using GLVis.

Mesh Files

Mesh files are used to define the geometry and topology of the mesh. They contain information about the nodes, elements, and how they connect.

The MFEM mesh file format is a simple text-based format that describes the mesh structure. It includes information about the mesh dimension, the number of elements, their types, and the coordinates of the vertices. Listing 11 shows the general structure of a simple MFEM mesh file.

The mesh elements and boundary elements are defined by their geometry type, which is specified as a nonnegative integer. The geometry types and their corresponding indices are listed in Table 8.

Geometry Type	Index
Point	`0`
Segment	`1`
Triangle	`2`
Square	`3`
Tetrahedron	`4`
Cube	`5`
Prism	`6`

Table 8: Geometry types and their corresponding indices in MFEM mesh files.

Attributes

In MFEM, the *attributes* of a mesh are used to label different parts of the mesh, such as boundaries or faces. Attributes are particularly useful for defining boundary conditions, material properties, or other characteristics that vary across the mesh. In Listing 12, we demonstrate how to set attributes for the boundary elements of a hexahedral mesh.

```
MFEM mesh v1.0

# Space dimension: 2 or 3
dimension
<dimension>

# Mesh elements, e.g. tetrahedrons (4)
elements
<number of elements>
<element attribute> <geometry type> <vertex index 1> ... <vertex index m>
...

# Mesh faces/edges on the boundary, e.g. triangles (2)
boundary
<number of boundary elements>
<boundary element attribute> <geometry type> <vertex index 1> ... <vertex index m>
...

# Vertex coordinates
vertices
<number of vertices>
<vdim>
<coordinate 1> ... <coordinate <vdim>>
...
```

Listing 11: The general structure of a (simple) MFEM mesh file.

```
const char[] mesh_file = "../data/ref-cube.mesh";
Mesh mesh(mesh_file, 1, 1);

// Mark the bottom boundary of the cube as attribute 1,
// the rest as attribute 2
mesh.GetBdrElement(0)->SetAttribute(1);
for (int i = 1; i < mesh.GetNBE(); i++)
{
   mesh.GetBdrElement(i)->SetAttribute(2);
}
mesh.SetAttributes();
```

Listing 12: Setting attributes for mesh boundaries.

Lines 1 and 2 are familiar. In line 6, we set the attribute of the first boundary element (the bottom face of the cube) to 1 using the `GetBdrElement` and `SetAttribute` methods. The subsequent loop iterates through all other boundary elements and sets their attributes to 2. The `GetNBE` methods returns the number of boundary elements in the mesh, which in this case, is 6. Lastly, we call `SetAttributes` to finalize our changes.

In more complex scenarios, such as when the mesh has multiple boundaries or when attributes need to be set based on specific geometric criteria, we can use a loop to iterate through the boundary elements and set their attributes conditionally. In Listing 13, we demonstrate how to set attributes based on the centroid of each boundary facet. Listings 12 and 13 are equivalent.

```
const char[] mesh_file = "data/ref-cube.mesh";
Mesh mesh(mesh_file, 1, 1);

// Mark the bottom boundary of the solid as attribute 1,
// the rest as 2
for (int i = 0; i < mesh.GetNBE(); i++)
{
  Element *facet = mesh.GetBdrElement(i);
  Array<int> vertices;
  facet->GetVertices(vertices);

  // Compute the centroid of the facet
  real_t z_centroid = 0.0;
  for (int j = 0; j < vertices.Size(); j++)
  {
    z_centroid += mesh.GetVertex(vertices[j])(dim-1);
  }
  z_centroid /= vertices.Size();

  if (abs(z_centroid) < 1e-8)
  {
    facet->SetAttribute(1);
  }
  else
  {
    facet->SetAttribute(2);
  }
}
mesh.SetAttributes();
```

Listing 13: Setting attributes for mesh boundaries.

Boundary Conditions

In the previous section, we discussed how to define boundary attributes in a mesh. In this section, we will explore how to apply boundary conditions using these attributes.

Suppose we wish to impose a Dirichlet boundary condition on a subset of the boundary of a mesh, Γ. In particular, we wish to impose $u = g$ on the bottom face of a cube, whose mesh is given by `data/ref-cube.mesh`. First, we need a `Coefficient` object, say `g_coeff`, that defines the function g; confer our chapter on Coefficients and GridFunctions for how to accomplish this. Second, we see that the bottom face of the cube has boundary attribute 1[22] by inspecting `data/ref-cube.mesh`. Third, we create an `Array<int>` called `ess_bdr` that essentially acts as a switch to turn on or off the boundary condition $u = g$ for each boundary attribute. In this case, we set the first entry of `ess_bdr` to `1`, which corresponds to the bottom face of the cube, and all other entries to `0`:

[22] If the attributes of our mesh were not already set, we could proceed by following e.g. Listing 13.

```
// Define the mesh
const char *mesh_file = "data/ref-cube.mesh";
Mesh mesh(mesh_file, 1, 1);

// Define the array of essential boundary attributes
Array<int> ess_bdr(mesh.bdr_attributes.Max());
ess_bdr = 0; ess_bdr[0] = 1;
```

The term "essential" in this context is a synonym for "Dirichlet", as in, *essential boundary conditions* are those that specify the value of the solution on the boundary.

Fourth, we create another `Array<int>` called `ess_tdof_list`, which will hold the degrees of freedom corresponding to the essential boundary conditions; in MFEM, we call these "essential true degrees of freedom."

```
Array<int> ess_tdof_list;
fespace.GetEssentialTrueDofs(ess_bdr, ess_tdof_list);
```

In the above, `fespace` is of type `FiniteElementSpace`.

Fifth, we impose $u = g$ on the bottom face Γ through the `ProjectBdrCoefficient` method:

```
GridFunction u(&fespace);
u = 0.0;
u.ProjectBdrCoefficient(g_coeff, ess_bdr);
```

Lastly, assuming our bilinear form `a` and linear form `b` are already defined and assembled, we enforce $u = g$ directly within our system with the `FormLinearSystem` method.

```
OperatorPtr A;
Vector B, X;
a.FormLinearSystem(ess_tdof_list, u, b, A, X, B);
```

Mesh Methods

We will now discuss *some* of the methods that are available in the `Mesh` class, which are defined in `mesh/mesh.hpp`. See Table 9.

Method	Description	Use
`Dimension`	Dimension of element	`mesh.Dimension()`
`SpaceDimension`	Dimension of physical space	`mesh.SpaceDimension()`
`MeshGenerator`	Identify present element types	`mesh.MeshGenerator()`
`HasBoundaryElements`	Check if mesh has boundary elements	`mesh.HasBoundaryElements()`
`GetBoundingBox`	Get minimum and maximum corners	`mesh.GetBoundingBox(min,max)`
`GetNV`	Get number of vertices	`mesh.GetNV()`
`GetNE`	Get number of elements	`mesh.GetNE()`
`GetNBE`	Get number of boundary elements	`mesh.GetNBE()`
`GetNEdges`	Get number of edges	`mesh.GetNEdges()`
`GetNFaces`	Get number of faces	`mesh.GetNFaces()`

Table 9: Some methods available to objects of type `Mesh`. Here, `mesh` is of type `Mesh`, and `min`, `max` are of type `Vector` with size equal to `mesh.SpaceDimension()`.

We note that the `GetNFaces` method is only available for 3D meshes.

Coefficients

In this chapter, we will discuss the `Coefficient`. In MFEM, objects of type `Coefficient` are used to represent functions that can be evaluated at a given point in space. These functions are often used to define the coefficients of PDEs, such as diffusion coefficients or source terms. For example, the function f in (1) would be represented in MFEM through a `Coefficient` object. The source for the `Coefficient` class can be found in `fem/coefficient.hpp` and `fem/coefficient.cpp`.

Definitions

Scalar Coefficients

As we saw previously, there are multiple ways to define an object of type `Coefficient`. One way is to inherit from the `Coefficient` class and utilize the `Eval` method to evaluate the function at a given point. This is shown in Listing 14.

```
class CoefficientType : public Coefficient
{
public:
  virtual double Eval(ElementTransformation &T, const
      IntegrationPoint &ip)
  {
    Vector x(T.GetDimension());
    T.Transform(ip, x);

    y = f(x);

    return y;
  }
};
```

Listing 14: Psuedocode to define a function f by inheriting from the `Coefficient` class.

To create an object based on this class, the user could, within `main`, write `CoefficientType f_coeff` to create an object called `f_coeff`.

A possibly simpler way is to define an std function and pass it to the constructor of the `Coefficient` class; see Listing 15.

```
real_t function(const Vector &x)
{
  return f(x);
}

// In main
FunctionCoefficient scaler_coeff(function);
```

Listing 15: Psuedocode to define a function f by using an std function.

In this case, both methods can be used interchangeably. However, the former may be necessary in contexts where the user needs to interact with the mesh; cf. Listing 23.

Sometimes, a simple `ConstantCoefficient` type suffices. To represent the mapping $\mathbb{R}^n \ni x \mapsto c \in \mathbb{R}$ for some $c \in \mathbb{R}$, we can use the `ConstantCoefficient` type; see Listing 16.

```
// In main
real_t c;
ConstantCoefficient const_coeff(c);
```

Listing 16: Psuedocode to define a constant function f.

We can also create `Coefficient` objects that are defined piecewise, according to the attributes of a mesh. Such a circumstance is encountered in linear elasticity, with a linear solid composed of two distinct materials with distinct Lamé parameters λ and μ. Material 1 may have Lamé parameters λ_1, μ_1, while material 2 may have Lamé parameters λ_2, μ_2. In this scenario, it is most appropriate to use `PWConstCoefficient`, a subclass of `Coefficient` that represents piecewise-constant[23] `Coefficient` objects. This is accomplished through defining a `Vector` object, as large as the number of attributes our mesh contains, populated according to the values it takes at each mesh attribute. In particular, the $i-1$th entry of this `Vector` corresponds to the ith mesh attribute[24]. See Listing 17.

[23] The "piecewise" present in this terminology is referring to the different pieces, i.e. attributes, of the mesh that this `Coefficient` object is defined on.

[24] Here, we're assuming $i \geq 1$.

```
Vector lambda(mesh->attributes.Max());
real_t l1; real_t l2;
lambda(0) = l1; lambda(1) = l2;
PWConstCoefficient lambda_func(lambda);

Vector mu(mesh->attributes.Max());
real_t m1; real_t m2;
m(0) = m1; m(1) = m2;
PWConstCoefficient mu_func(mu);
```

Listing 17: Psuedocode to define a piecewise-constant functions, with values according to the attributes of a mesh.

Vector Coefficients

The `VectorCoefficient` type is used to represent (mathematical) vector-valued functions. We can initialize an object of type `VectorCoefficient` by using an std function. The code in Listing 18 shows how to define the mapping $\mathbb{R}^n \ni x \mapsto \sin(\pi x_i) \in \mathbb{R}^n$ that takes a `Vector` as input and returns a `Vector` as output.

```
void vec_function(const Vector &x, Vector &f)
{
  for (int i = 0; i < x.Size(); i++)
  {
     f(i) = sin(M_PI * x(i));
  }
}

// In main
VectorFunctionCoefficient vec_coeff(dim, vec_function);
```

Listing 18: Defining an object of type `VectorFunctionCoefficient`. Here, `dim` is the dimension of the mesh in the finite element space.

Matrix Coefficients

The `MatrixCoefficient` type is used to represent (mathematical) matrix-valued functions. We can initialize an object of type `MatrixCoefficient` by using an std function. The code in Listing 19 shows how to define a particular `MatrixFunctionCoefficient`.

```
void mat_function(const Vector &x, DenseMatrix &K)
{
  K.SetSize(x.Size(),x.Size());
  for (int i = 0; i < K.Height(); i++)
  {
     for (int j = 0; j < K.Width(); j++)
     {
        K(i,j) = i*i + j*j;
     }
  }
}

// In main
MatrixFunctionCoefficient mat_coeff(dim, mat_function);
```

Listing 19: Defining an object of type `MatrixFunctionCoefficient`. Here, `dim` is the dimension of the mesh in the finite element space.

GridFunction Coefficients

There are other `Coefficient` types that are defined according to an operator on a `GridFunction` object. For example, we may wish to store

∇u or curl $\mathbf{u}$ as a `Coefficient`, where u represents a `GridFunction` object. We tabulate such `Coefficient` types in Table 10.

Class	Math
`GradientGridFunctionCoefficient`	∇u
`CurlGridFunctionCoefficient`	curl $\mathbf{u}$
`DivergenceGridFunctionCoefficient`	div $\mathbf{u}$

Table 10: `Coefficient` subclasses used to represent `Coefficient` objects defined according to an operator applied on a `GridFunction` object. Here, u mathematically represents a `GridFunction` object.

Scalar-Arithmetic Coefficients

We can also perform arithmetic operations on `Coefficient` objects. We tabulate such `Coefficient` types in Table 11.

Class	Use	Math
`SumCoefficient`	`SumCoefficient sum(f,g,alpha,beta)`	$\alpha f + \beta g$
`ProductCoefficient`	`ProductCoefficient prod(f,g)`	fg
`RatioCoefficient`	`RatioCoefficient quo(f,g)`	f/g
`PowerCoefficient`	`PowerCoefficient pow(f,alpha)`	f^α

Table 11: `Coefficient` subclasses used to represent arithmetic operations on `Coefficient` objects. Here, `f`, `g` are of type `Coefficient`, and `alpha`, `beta` are of type `real_t`.

We note that the first argument in the `ProductCoefficient` type can be of type `real_t`, so to define a `Coefficient` corresponding to αf, we can write `ProductCoefficient scale(alpha,f)`. Similarly, to define a `Coefficient` corresponding to α/f or f/β, we can write

`ProductCoefficient squo(alpha,f)` or `ProductCoefficient squo(f,beta)`,

respectively.

Vector-Arithmetic Coefficients

We can also perform arithmetic operations on `VectorCoefficient` objects. We tabulate such `VectorCoefficient` types in Table 12.

Class	Use	Math
`VectorSumCoefficient`	`VectorSumCoefficient vsum(f,g,alpha,beta)`	$\alpha\mathbf{f} + \beta\mathbf{g}$
`ScalarVectorProductCoefficient`	`ScalarVectorProductCoefficient svprod(g,f)`	$g\mathbf{f}$
`InnerProductCoefficient`	`InnerProductCoefficient innerprod(f,g)`	$\mathbf{f}\cdot\mathbf{g}$
`VectorCrossProductCoefficient`	`VectorCrossProductCoefficient crossprod(f,g)`	$\mathbf{f}\times\mathbf{g}$

Table 12: `VectorCoefficient` subclasses used to represent arithmetic operations on `VectorCoefficient` objects. Here, `f` is of type `VectorCoefficient`, `g` is of type `Coefficient` or `VectorCoefficient`, and `alpha`, `beta` are of type `real_t`.

To represent $\mathbf{f} + \mathbf{g}$, we can write `VectorSumCoefficient vsum(f,g);` to represent $\alpha\mathbf{f}$, we can write

`ScalarVectorProductCoefficient vscale(alpha,f).`

Matrix-Arithmetic Coefficients

We can also perform arithmetic operations on `MatrixCoefficient` objects. We tabulate such `MatrixCoefficient` types in Table 13.

Class	Use	Math
`MatrixSumCoefficient`	`MatrixSumCoefficient matsum(A,B,alpha,beta)`	$\alpha A + \beta B$
`MatrixProductCoefficient`	`MatrixProductCoefficient matprod(A,B)`	AB
`ScalarMatrixProductCoefficient`	`ScalarMatrixProductCoefficient smatprod(f,A)`	fA

Table 13: `MatrixCoefficient` subclasses used to represent arithmetic operations on `MatrixCoefficient` objects. Here, `A`, `B` are of type `MatrixCoefficient`, `f` is of type `Coefficient`, and `alpha`, `beta` are of type `real_t`.

To represent $A + B$, we can write `MatrixSumCoefficient matsum(A,B);` to represent αA, we can write

```
ScalarMatrixProductCoefficient matscale(alpha,A).
```

Functions

There are only two functions defined for `Coefficient` and `VectorCoefficient` objects, notably, `ComputeLpNorm` and `ComputeGlobalLpNorm`. Suggestively, these functions compute the L^p norm of a scalar-valued or vector-valued function. The difference between them is that `ComputeGlobalLpNorm` should be used when running MFEM in parallel. In the case $f : \mathbb{R}^n \to \mathbb{R}$, we can use `ComputeLpNorm` to compute

$$\|f\|_{L^p(\Omega)} = \left(\int_\Omega |f|^p \, \mathrm{d}x \right)^{1/p},$$

and in the case $f : \mathbb{R}^n \to \mathbb{R}^m$, we can use the same (overloaded) function to compute

$$\|f\|_{L^p(\Omega)} = \left(\sum_{i=1}^m \|f_i\|_{L^p(\Omega)}^p \right)^{1/p} = \left(\sum_{i=1}^m \int_\Omega |f_i|^p \, \mathrm{d}x \right)^{1/p}.$$

We can even take $p = \infty$ by writing `real_t p = infinity()`. See Listing 20.

```cpp
#include "mfem.hpp"
#include <iostream>

using namespace mfem;

// f(x,y) = x^2 + y
real_t f_std(const Vector &x)
{
   return x(0) * x(0) + x(1);
}

int main(int argc, char *argv[])
{
   // 1. Parse command line options.
   const char *mesh_file = "data/star.mesh";
   const int order = 4;
   const real_t p = 2.0;

   // 2. Read the mesh from the given mesh file.
   Mesh mesh(mesh_file, 1, 1);
   const int dim = mesh.Dimension();

   // 3. Define the coefficient.
   FunctionCoefficient f_coeff(f_std);

   // 4. Set up integration rules for all geometry types.
   const IntegrationRule *irs[Geometry::NumGeom];
   for (int i = 0; i < Geometry::NumGeom; i++)
   {
      irs[i] = &IntRules.Get(i, order);
   }

   // 5. Compute the Lp norm.
   real_t norm = ComputeLpNorm(p, f_coeff, mesh, irs);
   std::cout << "L^" << p << " norm of f: " << norm << std::endl;

   return 0;
}
```

Listing 20: Computing the L^2 norm of the function $f : \mathbb{R}^2 \to \mathbb{R}$ defined by $f(x, y) = x^2 + y$ over a star-shaped domain.

GridFunctions

In this chapter, we will discuss the `GridFunction` class. The `GridFunction` class is used to represent functions defined on a finite element space. The source for the `GridFunction` class can be found in `fem/gridfunc.hpp` and `fem/gridfunc.cpp`.

Definitions

A `GridFunction` object can be thought of as a vector of coefficients that define the function in the finite element basis. The `GridFunction` class provides methods to evaluate the function at a given point, compute its gradient, and perform other operations.

To create a `GridFunction` object, the user must first create a finite element space and then pass it to the `GridFunction` constructor. Listing 21 shows how to create a `GridFunction` object based on a finite element space and a `Coefficient` object.

```
// In main
FiniteElementSpace fespace(&mesh, &fec);
GridFunction f(&fespace);
f.ProjectCoefficient(f_coeff);
```

Listing 21: Defining a `GridFunction` object from a `Coefficient` object `f_coeff`. Here, `mesh` is of type `Mesh` and `fec` is of type `FiniteElementCollection`.

Plotting GridFunctions

We can use GLVis to plot `GridFunction` objects. For example, defining $\Omega \subset \mathbb{R}^2$ as the unit square in the first quadrant, we can plot the function

$$f : \Omega \to \mathbb{R} \quad \text{defined by} \quad f(x, y) = \sin(\pi x) \sin(\pi y)$$

through code in Listing 22. A plot of this function is shown in Figure 8.

See the Appendix for more information on GLVis.

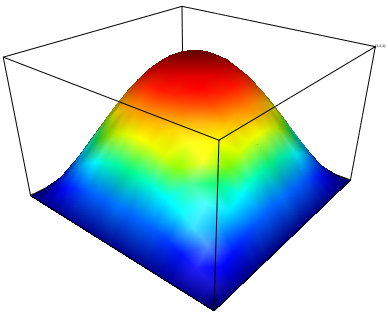


Figure 8: A plot of the function f defined in Listing 22. The plot was generated using GLVis.

```
real_t sinsin(const Vector &x)
{
  return sin(M_PI * x(0)) * sin(M_PI * x(1));
}

// In main
// Define the mesh as a disc
const char* mesh_file = "mfem/data/ref-square.mesh";
Mesh mesh(mesh_file, 1, 1);
const int dim = mesh.Dimension();

// Define the finite element collection
int order = 2;
H1_FECollection fec(order, dim);
FiniteElementSpace fespace(&mesh, &fec);

// Define f
FunctionCoefficient f_coeff(sinsin);
GridFunction f(&fespace);
f.ProjectCoefficient(f_coeff);

// Set up the visualization socket
char vishost[] = "localhost";
int visport = 19916;
socketstream sol_sock(vishost, visport);
sol_sock.precision(8);

// Plot f
sol_sock << "solution\n" << mesh << f << std::flush;
```

Listing 22: Plotting the function $(x, y) \mapsto \sin(\pi x)\sin(\pi y)$ through a `GridFunction`.

Methods

We will now discuss *some* of the methods that are available in the `GridFunction` class; see Table 14. These methods can are defined in `fem/gridfunc.hpp` and implemented in `fem/gridfunc.cpp`.

Method	Description	Use	Math
`ProjectCoefficient`	`Coefficient` to `GridFunction`	`f.ProjectCoefficient(f_coeff)`	NA
`ComputeL2Error`	L^2 distance	`f.ComputeL2Error(g_coeff)`	$\|f-g\|_{L^2}$
`ComputeH1Error`	H^1 distance	`f.ComputeH1Error(&g_coeff,&g_grad_coeff)`	$\|f-g\|_{H^1}$

Table 14: Some methods available to objects of type `GridFunction`. Here, `f` is of type `GridFunction`, `f_coeff`, `g_coeff` are of type `Coefficient`, and `g_grad_coeff` is of type `VectorCoefficient`.

Using the Normal

At times, we may need to use the unit normal vector in our computations, for example, when we need to compute the flux across a boundary. This is demonstrated in Listing 23, where we compute $\nabla u \cdot n$ for a given `GridFunction` `u`.

```
1  virtual real_t Eval(ElementTransformation &T, const
       IntegrationPoint &ip)
2  {
3     int dim = T->GetDimension();
4
5     // Set the integration point to ip
6     T->SetIntPoint(&ip);
7
8     // Compute gradient at the point ip
9     Vector grad_u;
10    u->GetGradient(T, grad_u);
11
12    // Set the integration point to the center of the geometry
13    T->SetIntPoint(&Geometries.GetCenter(T->GetGeometryType()));
14
15    // Compute the (unscaled) normal
16    Vector normal(dim);
17    CalcOrtho(T->Jacobian(), normal);
18
19    // Normalize the normal
20    normal /= normal.Norml2();
21
22    return grad_u * normal;
23 }
```

Listing 23: Acquire the unit normal n to the boundary $\partial\Omega$ and gradient ∇u of a function u and return $\nabla u \cdot n$, at a point.

In Listing 23, lines 1–3 are familiar. Here, `u` is of type `GridFunction*` and was defined outside of Listing 23. In line 6, we

set the integration point of the transformation `T` to `ip`. This is necessary because the transformation `T` may be used to evaluate multiple points, and we need to specify which point we are interested in. In line 10, the `GetGradient` method of the `GridFunction` class is used to compute the gradient of a given function at the most recently set integration point. We apply `GetGradient` to `u` and store its value in `grad_u`. This is why we need not initialize `grad_u` with a size—its size is given through `u`[25]. In line 17, the `CalcOrtho` method can be used to compute the normal vector at the given point. The `Jacobian` method of the `ElementTransformation` class returns the Jacobian of a given transformation. Lastly, we use the intuitive `normal /= normal.Norml2()` to normalize the vector and return the dot product `grad_u * normal`.

We caution that the vector returned by the `CalcOrtho` method in line 17 is *not* normalized by default.

[25] See `fem/gridfunc.cpp`.

Linear Forms

In this chapter, we discuss `LinearForm` and `LinearFormIntegrator` objects in MFEM, which are used to represent linear forms on finite element spaces. Linear forms are the right-hand side of a linear system, and they can be defined by integrating a function against a finite element basis. The linear form $v \mapsto \int_\Omega fv \, \mathrm{d}x$ for given $f \in L^2(\Omega)$ is a linear form on $H_0^1(\Omega)$. The source for this class can be found in `fem/lininteg.hpp` and `fem/lininteg.cpp`.

Definition

Linear forms are defined through a two-step process. First, we define a `LinearForm` object, `LinearForm b(&fespace)`, where `fespace` is of type `FiniteElementSpace`. The linear form can then be populated with integrators that define how the linear form is computed. The integrators are literally added using the `AddDomainIntegrator` method, which requires a pointer to a `LinearFormIntegrator` object.

Suppose, for example, we want to use the linear form

$$b : H^1(\Omega) \to \mathbb{R} \quad \text{given by} \quad b = b_1 + b_2,$$

where

$$b_1(v) = \int_\Omega fv \, \mathrm{d}x, \quad b_2(v) = \int_\Omega gv \, \mathrm{d}x,$$

and $f, g \in L^2(\Omega)$. We can use the integrator `DomainLFIntegrator` to accomplish this. Denoting $h = f + g$, one simple method to define this linear form in MFEM is first to note that

$$b(v) = \int_\Omega (f + g)v \, \mathrm{d}x = \int_\Omega hv \, \mathrm{d}x.$$

We can then define b as in Listing 24.

We can proceed differently by defining the linear form as a sum of two linear forms, each defined by a different function. In this case, we can use the `AddDomainIntegrator` method twice, once for each function; see Listing 25. Of course, such a practice is not necessary for b, but highlights the flexibility of the `LinearForm` class.

```
// In main
FunctionCoefficient h_coeff(h);
LinearForm b(&fespace);
b.AddDomainIntegrator(new DomainLFIntegrator(h_coeff));
b.Assemble();
```

Listing 24: Defining the linear form $v \mapsto \int_\Omega hv \,\mathrm{d}x$ in MFEM. Here, `h` is a function defined outside of `main` representing h, and `fespace` is of type `FiniteElementSpace`.

```
// In main
FunctionCoefficient f_coeff(f);
FunctionCoefficient g_coeff(g);
LinearForm b(&fespace);
b.AddDomainIntegrator(new DomainLFIntegrator(f_coeff));
b.AddDomainIntegrator(new DomainLFIntegrator(g_coeff));
b.Assemble();
```

Listing 25: Defining the linear form $v \mapsto \int_\Omega fv \,\mathrm{d}x + \int_\Omega gv \,\mathrm{d}x$ in MFEM. Here, `f`, `g` are functions defined outside of `main` representing f, g, and `fespace` is of type `FiniteElementSpace`.

We will discuss the use of the `Assemble` method in the following section.

Construction

`LinearFormIntegrator` is an abstract base class for integrators that compute contributions to a linear form. It is used in the assembly process to define how the linear form is constructed from the finite element basis functions and the problem data.

Suppose we wanted to use the linear form $b : H_0^1(\Omega) \to \mathbb{R}$ defined by $b(v) = \int_\Omega fv \,\mathrm{d}x$ in MFEM, for given $f \in L^2(\Omega)$. If $V_h \subset H_0^1(\Omega)$ is a finite element space with basis $\{\varphi_i\}_{i=1}^N$, we need to compute $b(\varphi_i)$ for each basis function φ_i. Using our mesh $\mathcal{T}_h$ for Ω, we can proceed as follows:

$$b(\varphi_i) = \int_\Omega f\varphi_i \,\mathrm{d}x = \sum_{K\in\mathcal{T}_h} \int_K f\varphi_i \,\mathrm{d}x.$$

Next, we can use the change of variables formula to express the integral over each element K in terms of the local coordinates:

$$\int_K f\varphi_i \,\mathrm{d}x = \int_{\widehat{K}} (f\circ T_K)(\varphi_i \circ T_K)|\det J_{T_K}| \,\mathrm{d}\widehat{x},$$

where J_{T_K} is the Jacobian of the transformation $T_K : \widehat{K} \to K$ from the reference element $\widehat{K}$ to the element K.

Using quadrature, we can approximate the integral above as

$$\begin{aligned}\int_{\widehat{K}} (f\circ T_K)(\varphi_i \circ T_K)|\det J_{T_K}| \,\mathrm{d}\widehat{x} \\ \approx \sum_{q=1}^{Q} w_q (f\circ T_K)(\widehat{x}_q)(\varphi_i \circ T_K)(\widehat{x}_q)|\det J_{T_K}(\widehat{x})|,\end{aligned}$$

where $\{(\widehat{x}_q, w_q)\}_{q=1}^Q$ are the quadrature points and weights on the reference element $\widehat{K}$.

Finally, we can assemble the contributions from all elements to obtain

$$b(\varphi_i) \approx \sum_{K \in \mathcal{T}_h} \sum_{q=1}^{Q} w_q (f \circ T_K)(\widehat{x}_q)(\varphi_i \circ T_K)(\widehat{x}_q) |\det J_{T_K}(\widehat{x})|. \tag{8}$$

Thus, it is through (8) that MFEM internally represents the linear form b. More precisely, `LinearFormIntegrator` objects compute the inner sum, and `LinearForm` objects compute the outer sum through the `Assemble` method.

Implementation

Let's now take a look at the implementation of the linear form $v \mapsto \int_\Omega f v \, dx$ in MFEM. This linear form is defined in `fem/lininteg.hpp` as `DomainLFIntegrator`; a stripped-down version of the definition is show below.

```
1  /// Class for domain integration $ L(v) := (f, v) $
2  class DomainLFIntegrator : public DeltaLFIntegrator
3  {
4    Vector shape;
5    Coefficient &Q;
6    int oa, ob;
7  public:
8    /// Constructs a domain integrator with a given Coefficient
9    DomainLFIntegrator(Coefficient &QF, int a = 2, int b = 0)
10     : DeltaLFIntegrator(QF), Q(QF), oa(a), ob(b) { }
11
12   bool SupportsDevice() const override { return true; }
13
14   /// Method defining assembly on device
15   void AssembleDevice(const FiniteElementSpace &fes, const
         Array<int> &markers, Vector &b) override;
16
17   /** Given a particular Finite Element and a transformation (Tr)
18      computes the element right hand side element vector, elvect.
           */
19   void AssembleRHSElementVect(const FiniteElement &el,
         ElementTransformation &Tr, Vector &elvect) override;
20
21   using LinearFormIntegrator::AssembleRHSElementVect;
22 };
```

The key parts of this class are in lines 9–10 and line 19. Lines 9–10 allow us to interact with the class and input our $L^2(\Omega)$ function f, which is `QF` and `Q` in the code. The `AssembleRHSElementVect` method (line 19) computes the right-hand side vector for a given element (i.e. the inner sum in (8)).

A stripped-down implementation of the `AssembleRHSElementVect` method is shown below, and taken from `fem/lininteg.cpp`.

```
1  void DomainLFIntegrator::AssembleRHSElementVect(const
       FiniteElement &el, ElementTransformation &Tr, Vector &elvect)
2  {
3    int dof = el.GetDof();
4
5    shape.SetSize(dof); // vector of size dof
6    elvect.SetSize(dof);
7    elvect = 0.0;
8
9    const IntegrationRule *ir = GetIntegrationRule(el, Tr);
10
11   if (ir == NULL)
12   {
13     ir = &IntRules.Get(el.GetGeomType(), oa * el.GetOrder() + ob);
14   }
15
16   for (int q = 0; q < ir->GetNPoints(); i++)
17   {
18     const IntegrationPoint &ip = ir->IntPoint(q);
19
20     Tr.SetIntPoint(&ip);
21     real_t val = Tr.Weight() * Q.Eval(Tr, ip);
22
23     el.CalcPhysShape(Tr, shape);
24
25     add(elvect, ip.weight * val, shape, elvect);
26   }
27 }
```

The vectors `shape` and `elvect` hold the shape functions φ_i and the value of the inner sum in (8) for all shape functions, respectively. Of course, the size of these vectors is equal to the number of degrees of freedom (dof) in the finite element space, `el.GetDof()`, and is set in lines 5–6.

In line 18, we obtain the point $\widehat{x}_q$ in the reference element, the q-th quadrature point, and store it in `ip`. In line 20, we set the transformation `Tr` to the quadrature point $\widehat{x}_q$, which informs the program that we wish to evaluate T_K and/or $\det T_K$ at $\widehat{x}_q$.

In line 21, we compute the product $|\det J_{T_K}(\widehat{x}_q)|(f \circ T_K)(\widehat{x}_q)$ in (8). The code `Tr.Weight()` computes $\det J_{T_K}(\widehat{x}_q)$; `Q.Eval(Tr, ip)` computes $(f \circ T_K)(\widehat{x}_q)$. Line 23 computes $(\varphi_i \circ T_K)(\widehat{x}_q)$ for all shape functions φ_i and stores them in `shape`.

The code `ip.weight` returns the associated weight w_q with the quadrature point $\widehat{x}_q$. The last line, line 25, is equivalent to `elvect += ip.weight * val * shape`.

Tables of Linear Forms

We now provide a list of linear forms available in MFEM. We partition the list into two tables: the first one contains the linear forms that occur on the entire domain Ω, while the second one contains the linear forms that occur on the boundary $\partial\Omega$. When using domain integrators, the method `AddDomainIntegrator` is used, while for boundary integrators, the method `AddBoundaryIntegrator` is used. See Tables 15 and 16.

Name	Description	Domain, Range	Elements
`DomainLFIntegrator`	$(f, v)_\Omega$	$(\mathbb{R}^n \to \mathbb{R}, \mathbb{R}^n \to \mathbb{R})$	H^1, L^2
`DomainLFGradIntegrator`	$(f, \nabla v)_\Omega$	$(\mathbb{R}^n \to \mathbb{R}^n, \mathbb{R}^n \to \mathbb{R})$	H^1
`VectorDomainLFIntegrator`	$(f, v)_\Omega$	$(\mathbb{R}^n \to \mathbb{R}^n, \mathbb{R}^n \to \mathbb{R}^n)$	H^1, L^2
`VectorDomainLFGradIntegrator`	$(f, \nabla v)_\Omega$	$(\mathbb{R}^m \to \mathbb{R}^{m\times m}, \mathbb{R}^m \to \mathbb{R}^m)$	H^1
`VectorFEDomainLFCurlIntegrator`	$(f, \operatorname{curl} v)_\Omega$	$(\mathbb{R}^3 \to \mathbb{R}^3, \mathbb{R}^3 \to \mathbb{R}^3)$	Nedelec
`VectorFEDomainLFDivIntegrator`	$(f, \operatorname{div} v)_\Omega$	$(\mathbb{R}^n \to \mathbb{R}, \mathbb{R}^n \to \mathbb{R})$	Raviart-Thomas
`VectorFEDomainLFIntegrator`	$(f, v)_\Omega$	$(\mathbb{R}^n \to \mathbb{R}^n, \mathbb{R}^n \to \mathbb{R}^n)$	Nedelec, Raviart-Thomas

Table 15: Linear forms on the entire domain Ω. The test function is denoted by v, $n \geq 1$, and $m = 2, 3$.

Name	Description	Domain, Range	Elements
`BoundaryLFIntegrator`	$(g, v)_{\partial\Omega}$	$(\mathbb{R}^n \to \mathbb{R}, \mathbb{R}^n \to \mathbb{R})$	H^1, L^2
`BoundaryNormalLFIntegrator`	$(g \cdot n, v)_{\partial\Omega}$	$(\mathbb{R}^n \to \mathbb{R}^n, \mathbb{R}^n \to \mathbb{R})$	H^1, L^2
`BoundaryTangentialLFIntegrator`	$(g \cdot \tau, v)_{\partial\Omega}$	$(\mathbb{R}^2 \to \mathbb{R}^2, \mathbb{R}^2 \to \mathbb{R})$	H^1, L^2
`VectorBoundaryLFIntegrator`	$(g, v \cdot n)_{\partial\Omega}$	$(\mathbb{R}^n \to \mathbb{R}, \mathbb{R}^n \to \mathbb{R}^n)$	H^1, L^2
`VectorBoundaryFluxLFIntegrator`	$(g, v)_{\partial\Omega}$	$(\mathbb{R}^n \to \mathbb{R}^n, \mathbb{R}^n \to \mathbb{R}^n)$	H^1, L^2
`VectorFEBoundaryFluxLFIntegrator`	$(g, v \cdot n)_{\partial\Omega}$	$(\mathbb{R}^n \to \mathbb{R}, \mathbb{R}^n \to \mathbb{R}^n)$	Raviart-Thomas
`VectorFEBoundaryNormalLFIntegrator`	$(g \cdot n, v \cdot n)_{\partial\Omega}$	$(\mathbb{R}^n \to \mathbb{R}^n, \mathbb{R}^n \to \mathbb{R}^n)$	Raviart-Thomas
`VectorFEBoundaryTangentLFIntegrator`	$(n \times g, v)_{\partial\Omega}$	$(\mathbb{R}^n \to \mathbb{R}^n, \mathbb{R}^n \to \mathbb{R})$	Nedelec

Table 16: Linear forms on the boundary $\partial\Omega$. The test function is denoted by v, the normal vector is denoted by n, the tangential vector is denoted by τ, and $n \geq 1$.

Writing Your Own Linear Forms

All available linear form integrators in MFEM can be found in `fem/lininteg.hpp` (with implementation in `fem/lininteg.cpp`). As such, to write your own linear form integrator, you will need to create a new class that inherits from `LinearFormIntegrator` within `fem/lininteg.hpp`. However, the trouble with this approach is that you will need to recompile MFEM after each change you make to your new linear form integrator. A better approach is to create a new class that inherits from `LinearFormIntegrator` in a separate file, and then include that file in your main program.

Suppose our linear form integrator is called `MyLFIntegrator`. We may define this integrator in a file `mylininteg.hpp`:

```
#include "mfem.hpp"

namespace mfem
{

class MyLFIntegrator : public LinearFormIntegrator
{
   // Define required methods here
}

}
```

We can write our implementation of `MyLFIntegrator` in a file `mylininteg.cpp`:

```
#include "mfem.hpp"
#include "mylininteg.hpp"

namespace mfem
{

void MyLFIntegrator::AssembleRHSElementVect(
   const FiniteElement &el,
   ElementTransformation &Tr,
   Vector &elvect)
{
  // Implement required methods here
}

}
```

Our project file `project.cpp` should include the header file corresponding to our linear form integrator:

Finally, we must include the files `mylininteg.hpp` and `mylininteg.cpp` in our makefile; see Listing 26.

Finally, we can run `make` to compile our project.

```cpp
#include "mfem.hpp"
#include "mylininteg.hpp"

using namespace mfem;

int main()
{
   // Write main code here
}
```

```
# Use the MFEM build directory
MFEM_DIR ?= ~/mfem
MFEM_BUILD_DIR ?= ~/mfem
SRC = project.cpp
CONFIG_MK = $(MFEM_BUILD_DIR)/config/config.mk

MFEM_LIB_FILE = mfem_is_not_built
-include $(CONFIG_MK)

# Target executable
TARGET = project

# Custom integrator dependencies
CUSTOM_SOURCES = mylininteg.cpp
CUSTOM_HEADERS = mylininteg.hpp

# Rule for building the target
$(TARGET): $(SRC) $(CUSTOM_SOURCES) $(CUSTOM_HEADERS)
   $(MFEM_LIB_FILE) $(CONFIG_MK) $(MFEM_CXX) $(MFEM_FLAGS)
   $< $(CUSTOM_SOURCES) -o $@ $(MFEM_LIBS)

# Generate an error message if the MFEM library is not built
# and exit
$(MFEM_LIB_FILE):
   $(error The MFEM library is not built)

clean:
   rm -f $(TARGET)

.PHONY: all clean
```

Listing 26: Makefile for compiling an MFEM program that uses a custom linear form integrator.

Appendix

Building MFEM

MFEM can only be built on UNIX-based operating systems (i.e. Linux, macOS). If the user wishes to build and use MFEM in a machine running Windows, they can use WSL (Windows Subsystem for Linux) to accomplish this. The guide in `https://mfem.org/building` provides a step-by-step guide to building MFEM. For more information on building MFEM, the user is encouraged to read `INSTALL`.

We encourage the user to build the debug version of MFEM by running `make debug` (or `make pdebug` for parallel builds). The debug build "enables a number of messages and consistency checks that may simplify bug-hunting."

user.mk

MFEM builds with certain settings by default set in `config/defaults.mk`. Among these are the destination location of `make install`, the number of threads to use, and whether to use METIS 5. Having to pass these options and others every time you build MFEM can be cumbersome—using a `user.mk` file allows you to set these options once and for all and makes the build process simpler. This can be accomplished by simply copying `config/defaults.mk` to `config/user.mk` and modifying the options you want to change within `config/user.mk`.

LSPs

Language Server Protocols (LSPs) are a standardized way for editors and IDEs to communicate with *language servers*, which provide features such as code completion, linting, refactoring, and go-to definition. It is highly encouraged to use an LSP or a tool providing similar functionality when working with MFEM due to the size of the codebase. A popular LSP for C++ is clangd; VSCode users can instead install the C/C++ extension.

Indentation

After opening a file such as examples/ex0.cpp, the user is prompted with a shocking discovery—the file uses tabs[26] of length three for indentation. This, in fact, is true for any MFEM file. Most likely, the user's editor is configured to use tabs of length two, four, or perhaps even eight. Thus, how can set three-space indentation *only* for MFEM files?

One simple method is to utilize a `.editorconfig`[27] file. A simple `.editorconfig` file for use in MFEM files is shown in Listing 27.

[26] By *tab*, we mean either the literal tab character or an indentation using spaces.

[27] EditorConfig. https://editorconfig.org

```
# Root configuration file
root = true

[*.{cpp,hpp,c,h}]
indent_style = space
indent_size = 3
```

Listing 27: A simple `.editorconfig` file.

The user can place this file in the root of their project and their editor will automatically apply[28] the specified settings.

If the user uses Vim, they can bypass `.editorconfig` and create a file called `~/.vim/after/ftplugin/cpp.vim` and populate it with the code given in Listing 28.

[28] Not all editors come preconfigured for `.editorconfig`; some require plugins.

```
function! SetMFEMIndentation()
  let l:lines = readfile(expand('%'), '', 10)
  let l:content = join(l:lines, "\n")

  if l:content =~# '#include\s\+<mfem.hpp>' || l:content =~#
      'using\s\+namespace\s\+mfem'
    setlocal expandtab
    setlocal shiftwidth=3
    setlocal tabstop=3
  endif
endfunction

autocmd VimEnter *.cpp,*.hpp call SetMFEMIndentation()
```

Listing 28: A Vim function to set proper indentation for MFEM files.

We note also that the MFEM repository uses Artistic Style for code formatting. The user can find the configuration file for Artistic Style used by MFEM in config/mfem.astylerc and, assuming Artistic Style is installed, the user can run `make style` to format all code in the repository.

Environment Variables

MFEM, BY DEFAULT, does not come included with environment variables. We, of course, are welcome to define our own. For example, we could define an environment variable `MFEM_DIR` to point to the directory where MFEM is installed. This can be accomplished by appending to the user's `.bashrc` or `.zprofile`; see Listing 29.

```
export MFEM_DIR="$HOME/mfem"
```

Listing 29: Defining a `MFEM_DIR` environment variable.

Using environment variables in MFEM allows the user to write portable code—not only across different systems, but across different projects. For example, if the user desires to use one of the provided meshes, they can use the `MFEM_DIR` environment variable to locate them.

```
const char* mfem_dir = getenv("MFEM_DIR");
if (!mfem_dir) mfem_dir = "../mfem"; // Default fallback

std::string mesh_file = std::string(mfem_dir) + "/data/star.mesh";
```

Listing 30: Reading and accessing a `MFEM_DIR` environment variable.

Makefiles

ONE'S FIRST APPROACH for getting acquainted with MFEM is most likely to be experimenting within the 🗀 `examples` directory. After the user becomes comfortable with the library, they may want to start their own project. Thus, starting a new project `project.cpp` in the 🗀 `examples` directory seems natural—the provided makefile 🗋 `examples/makefile` can be used for compilation. However, this constrains the user into using the 🗀 `examples` directory as their project root. A more attractive alternative is to place the project root wherever the user so desires, and to compile within that root. This can be accomplished by creating a custom makefile. In Listing 31, a sample makefile is provided to make a MFEM project named `project.cpp`.

Our makefile is modeled after 🗋 `examples/makefile` and can be easily modified according to the user's needs. The user must specify the MFEM directory and build directory in the makefile in lines 2 and 3. Furthermore, the user must specify the project source file in line 4, and the target executable in line 10. The user can then compile the project by simply running `make` in the project root.

```
# Use the MFEM build directory
MFEM_DIR ?= $(HOME)/mfem
MFEM_BUILD_DIR ?= $(HOME)/mfem
SRC = project.cpp
CONFIG_MK = $(MFEM_BUILD_DIR)/config/config.mk

MFEM_LIB_FILE = mfem_is_not_built
-include $(CONFIG_MK)

# Target executable
TARGET = project

# Rule for building the target
$(TARGET): $(SRC) $(MFEM_LIB_FILE) $(CONFIG_MK)
   $(MFEM_CXX) $(MFEM_FLAGS) $< -o $@ $(MFEM_LIBS)

# Generate an error message if the MFEM library is not built and
    exit
$(MFEM_LIB_FILE):
   $(error The MFEM library is not built)

clean:
   rm -f $(TARGET)

.PHONY: all clean
```

Listing 31: A sample makefile for an MFEM project.

For more information on makefiles, consult the GNU Make Manual or Chase Lambert's online tutorial[29].

[29] GNU make. https://www.gnu.org/software/make/manual/make.html; and Makefile Tutorial. https://makefiletutorial.com

GLVis

"GLVis is a *lightweight* tool for *accurate* and *flexible* finite element visualization[30]." Many of the written examples use GLVis to visualize their output.

[30] GLVis: OpenGL Finite Element Visualization Tool. glvis.org

Building GLVis not a difficult task; all instructions are available in glvis.org/building. Furthermore, we encourage the reader to read the README.md file in the GLVis GitHub repository to learn more about the tool, particularly its (well-documented) mouse and keyboard controls.

MFEM Web

The MFEM website contains a wealth of information about the library. The source of the website, just like the source of MFEM itself, is hosted publicly on GitHub at https://github.com/mfem/web. Anyone is welcome to contribute to the website by submitting a pull request to this repository. We will discuss how to build and test the website locally before submitting changes, and assume the repository has already been cloned into web/.

1. An install of Python version at least 3.6.9 and less than 3.10 is required. One of the best ways to manage multiple Python versions is with pyenv. After installing and setting up pyenv, you can install a specific version of Python, say Python 3.9.25, with the command:

```
$ pyenv install 3.9.25
```

2. Next, install the virtualenv package[31] if you don't have it already:

[31] Linux users may install this package through their respective package manager instead of using pip.

```
$ pip install virtualenv
```

and in the web/ directory, create a new virtual environment with:

```
$ virtualenv -p 3.9.25 mfem-web-venv
```

3. Activate the virtual environment with:

```
$ source mfem-web-venv/bin/activate
```

4. Install the required Python packages with:

```
$ pip install -r requirements.txt
```

   You may also need to install the `setuptools` Python package if it is not already installed:

```
$ pip install setuptools
```

5. Finally, preview the website locally with:

```
$ mkdocs serve
```

6. When you are done previewing your changes, exit the virtual environment with:

```
$ deactivate
```

Additional Resources

Learning MFEM is made easier with the help of additional resources that provide further insights, examples, and community support. Here are some valuable resources to explore:

MFEM Tutorial The MFEM tutorial is a comprehensive guide that covers various aspects of the library, including installation, basic usage, and advanced features. It is an excellent starting point for new users. You can access it at `https://mfem.org/tutorial`.

MFEM Documentation The official MFEM documentation is an essential resource for understanding the library's features, functions, and usage. You can find it at `https://mfem.org/dox`.

GitHub Discussion The MFEM GitHub repository has a discussion section where you can ask questions, share ideas, and engage with the MFEM community. Visit `https://github.com/orgs/mfem/discussions` to participate.

Bibliography

EditorConfig. https://editorconfig.org.

GLVis: OpenGL Finite Element Visualization Tool. glvis.org.

GNU make. https://www.gnu.org/software/make/manual/make.html.

Makefile Tutorial. https://makefiletutorial.com.

Richard Courant. Variational methods for the solution of problems of equilibrium and vibrations. *Bulletin of the American Mathematical Society*, 49(1):1–23, 1943. DOI: 10.1090/s0002-9904-1943-07818-4.

Leszek F. Demkowicz. *Mathematical Theory of Finite Elements*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2023. DOI: 10.1137/1.9781611977738.

L.C. Evans. *Partial Differential Equations*. Graduate studies in mathematics. American Mathematical Society, 2010. ISBN 9780821849743.

Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo. *C++ Primer*. Addison-Wesley, Upper Saddle River, NJ, 5 edition, 2013. ISBN 9780321714114.

mfem. MFEM: Modular finite element methods [Software]. https://mfem.org.