FEniCSx: Theory and Practice

A Didactic Introduction to the Finite Element Method and Its Implementation in FEniCSx

Will Robertson, with the help of AI

April 5, 2026

Abstract

This document is intended as a truly 'soft ramp' into the finite element method using FEniCSx. The current version of the document has been largely written using AI but I (WR) intend to expand and supplement and modify this material over time...

Contents

1 Introduction

The **Finite Element Method** (FEM) is a powerful and widely used numerical technique for solving partial differential equations (PDEs) arising in science and engineering. **FEniCSx** is an open-source computing platform for solving PDEs using the finite element method [1, 2], which provides high-level abstractions that allow users to express variational formulations close to their mathematical description, while transparently generating efficient low-level code.

1.1 What is FEniCSx?

FEniCSx consists of several components:

- **DOLFINx** — the core C++/Python library providing data structures for meshes, function spaces, and linear algebra.
- **UFL** (Unified Form Language) — a domain-specific language for expressing variational forms in near-mathematical notation.
- **FFCx** (FEniCS Form Compiler X) — compiles UFL forms to efficient C code.
- **Basix** — a finite element tabulation library providing basis functions for a wide range of element families.

1.2 Scope of This Document

This document provides a didactic introduction to the mathematical foundations underlying the finite element method, and their mapping to FEniCSx concepts and terminology. It is structured as follows:

1. Section 2 reviews the mathematical foundations: function spaces, variational formulations, and the Galerkin method.
2. Section 3 describes how these mathematical constructs map to FEniCSx data structures and workflow.
3. Section 4 presents worked examples that can be run directly with FEniCSx.

2 Mathematical Foundations

2.1 What Types of Problems Are Represented by PDEs?

Partial differential equations (PDEs) arise wherever a quantity varies continuously in space and/or time. Below are representative examples across disciplines, organised by the spatial dimension of the domain Ω.

2.1.1 One-Dimensional Problems

One-dimensional problems arise when the physical domain is a line segment $\Omega = (a, b) \subset \mathbb{R}$ (e.g. a rod, cable, or pipe cross-section).

Steady heat conduction in a rod. Given a rod of length L with thermal conductivity $\kappa > 0$ and a distributed heat source $f(x)$, the temperature $u(x)$ satisfies

$$-\kappa\, u'' = f \quad \text{in } (0, L), \qquad u(0) = T_0, \quad u(L) = T_L,$$

an *elliptic* (boundary-value) problem in 1D.

Vibrating string (wave equation). The transverse displacement $u(x, t)$ of a taut string with wave speed c obeys

$$u_{tt} = c^2\, u_{xx} \quad \text{in } (0, L) \times (0, T),$$

a *hyperbolic* PDE requiring both initial displacement and velocity as data.

Advection–diffusion equation. Transport of a scalar concentration $u(x, t)$ with velocity b and diffusivity $\varepsilon > 0$:

$$u_t + b\, u_x = \varepsilon\, u_{xx} \quad \text{in } \mathbb{R} \times (0, T).$$

This *parabolic* equation models pollutant transport in rivers or solute migration along a column.

2.1.2 Two-Dimensional Problems

When the domain is a planar region $\Omega \subset \mathbb{R}^2$, the spatial coordinates are (x, y) and we encounter a rich variety of problem types.

Membrane deflection (Poisson equation). The small deflection $u(x, y)$ of a pre-stressed elastic membrane under a transverse load f satisfies

$$-\Delta u = f \quad \text{in } \Omega \subset \mathbb{R}^2, \qquad u = 0 \quad \text{on } \partial\Omega.$$

The same equation governs electrostatic potential in a grounded 2D conductor and steady-state groundwater pressure.

Time-dependent heat conduction. Transient temperature in a thin plate:

$$\rho c_p \, u_t - \nabla \cdot (\kappa \, \nabla u) = f \quad \text{in } \Omega \times (0, T),$$

a *parabolic* PDE where ρ is density and c_p is specific heat.

Linear elasticity (plane stress/strain). Displacement field $\mathbf{u}(x, y) \in \mathbb{R}^2$ in a loaded structural component:

$$-\nabla \cdot \boldsymbol{\sigma}(\mathbf{u}) = \mathbf{f} \quad \text{in } \Omega,$$

where $\boldsymbol{\sigma}$ is the Cauchy stress tensor related to strain by Hooke's law. This is an *elliptic system* of two coupled PDEs.

Shallow-water equations. Depth-averaged flow in coastal regions is described by the 2D shallow-water system (a first-order *hyperbolic* system) governing water height and depth-averaged velocity.

2.1.3 Three-Dimensional Problems

Full 3D domains $\Omega \subset \mathbb{R}^3$ with coordinates (x, y, z) are required when out-of-plane or depth variation cannot be neglected.

3D Poisson / Laplace equation. The electric potential ϕ in a dielectric medium satisfies

$$-\nabla \cdot (\varepsilon \, \nabla \phi) = \rho_e \quad \text{in } \Omega \subset \mathbb{R}^3,$$

where ε is permittivity and ρ_e is charge density. Setting $\rho_e = 0$ gives the Laplace equation, governing steady diffusion in the absence of sources.

3D linear elasticity. Structural analysis of a 3D component under mechanical loading:

$$-\nabla \cdot \boldsymbol{\sigma}(\mathbf{u}) = \mathbf{f} \quad \text{in } \Omega \subset \mathbb{R}^3,$$

an *elliptic system* of three coupled PDEs for the displacement components (u_1, u_2, u_3).

Incompressible Navier–Stokes equations. Viscous fluid flow in 3D:

$$\begin{aligned} \rho\big(\mathbf{u}_t + (\mathbf{u} \cdot \nabla)\mathbf{u}\big) &= -\nabla p + \mu\, \Delta \mathbf{u} + \mathbf{f}, \\ \nabla \cdot \mathbf{u} &= 0, \end{aligned}$$

where $\mathbf{u}$ is velocity, p pressure, ρ density, and μ dynamic viscosity. This coupled *parabolic–elliptic* system governs aerodynamics, cardiovascular blood flow, and atmospheric dynamics.

Maxwell's equations. Electromagnetic wave propagation in 3D:

$$\begin{aligned} \nabla \times \mathbf{H} &= \mathbf{J} + \partial_t(\varepsilon \mathbf{E}), \\ \nabla \times \mathbf{E} &= -\partial_t(\mu_0 \mathbf{H}), \end{aligned}$$

a first-order *hyperbolic system* for the electric field $\mathbf{E}$ and magnetic field $\mathbf{H}$, with applications in photonics, antenna design, and MRI.

2.2 What Is a PDE Mathematically?

2.2.1 Definition

Definition 2.1 (Partial Differential Equation)**.** *A* ***partial differential equation*** *(PDE) is an equation relating an unknown function $u\colon \Omega \to \mathbb{R}$ (or $\mathbb{R}^m$) to its partial derivatives. For u depending on independent variables $\mathbf{x} = (x_1, \ldots, x_d)$ (and possibly time t), a PDE of* order k *has the general form*

$$F\left(\mathbf{x},\, u,\, \frac{\partial u}{\partial x_1}, \ldots, \frac{\partial^k u}{\partial x_i^{k_i} \cdots}\right) = 0, \tag{1}$$

where F is a given function and the highest-order derivative appearing in F determines the order *of the PDE.*

2.2.2 Notation

Throughout this document we adopt the following notational conventions.

- $\Omega \subset \mathbb{R}^d$ denotes an open, bounded domain with Lipschitz boundary $\partial\Omega = \partial\Omega$.[1] Spatial coordinates are $\mathbf{x} = (x_1, \ldots, x_d)$, often written (x) in 1D, (x, y) in 2D, and (x, y, z) in 3D.
- $t \in (0, T]$ is time (for time-dependent problems).
- Partial derivatives are written $\partial_t u \equiv u_t$, $\partial_{x_i} u \equiv u_{x_i}$, etc.
- The **gradient** of a scalar u is $\nabla u = (\partial_{x_1} u, \ldots, \partial_{x_d} u)^T \in \mathbb{R}^d$.
- The **divergence** of a vector field $\mathbf{q} = (q_1, \ldots, q_d)^T$ is $\nabla \cdot \mathbf{q} = \sum_{i=1}^d \partial_{x_i} q_i$.
- The **Laplacian** of a scalar u is $\Delta u = \nabla \cdot \nabla u = \sum_{i=1}^d \partial_{x_i}^2 u$.
- **Multi-index notation:** a multi-index $\alpha = (\alpha_1, \ldots, \alpha_d) \in \mathbb{N}_0^d$ with $|\alpha| = \sum_i \alpha_i$ denotes $D^\alpha u = \dfrac{\partial^{|\alpha|} u}{\partial x_1^{\alpha_1} \cdots \partial x_d^{\alpha_d}}$.

2.2.3 Classification of Second-Order PDEs

A second-order linear PDE in two variables (x_1, x_2) with constant coefficients has the form $A\,u_{x_1x_1} + B\,u_{x_1x_2} + C\,u_{x_2x_2} + \text{lower order} = 0$. Its *type* is determined by the discriminant $\mathcal{D} = B^2 - 4AC$:

Type	**Condition**	**Prototype**
Elliptic	$B^2 - 4AC < 0$	Laplace / Poisson equation
Parabolic	$B^2 - 4AC = 0$	Heat (diffusion) equation
Hyperbolic	$B^2 - 4AC > 0$	Wave equation

Each type requires different boundary and/or initial data to be *well-posed* in the sense of Hadamard: existence, uniqueness, and continuous dependence on data:

[1] Lipschitz means the boundary is locally representable as a graph with bounded (i.e., non-infinite) slope, excluding cusps and fractal irregularities. Note that this use of the ∂ symbol is unrelated to a partial derivative. This notation is standard in topology, differential geometry, and differential topology.

- **Existence:** There exists a solution u that satisfies the PDE and the prescribed boundary/initial conditions.
- **Uniqueness:** If u_1 and u_2 both satisfy the PDE with the same data, then $u_1 = u_2$.
- **Continuous dependence on data:** Small changes in the boundary/initial data lead to small changes in the solution (in an appropriate norm).

2.2.4 Boundary Conditions

PDEs on bounded domains require *boundary conditions* (BCs) on $\partial\Omega$ to select a unique solution. We must specify appropriate boundary conditions that model how the exterior influences the domain. Boundary conditions can be one of thee types: Dirichlet, Neumann, or Robin. Let $\{\Gamma_D, \Gamma_N, \Gamma_R\}$ be a non-overlapping partition of $\partial\Omega$ ($\Gamma_D \cup \Gamma_N \cup \Gamma_R = \partial\Omega$).

Dirichlet (essential) BC. This type of boundary condition is is used when the surroundings enforce a fixed state. The solution value is prescribed on Γ_D:

$$u = g_D \quad \text{on } \Gamma_D.$$

Examples: fixed temperature, clamped displacement, grounded potential.

Neumann (natural) BC. This is used when there is a specifed flux (or no flux) with the surrounding. The normal flux (outward derivative) is prescribed on Γ_N:

$$\kappa \frac{\partial u}{\partial n} \equiv \kappa \nabla u \cdot \mathbf{n} = g_N \quad \text{on } \Gamma_N,$$

where $\mathbf{n}$ is the outward unit normal to $\partial\Omega$. Examples: insulated boundary ($g_N = 0$), prescribed heat flux. The special case $g_N = 0$ is also called a *no-flux* or *free* boundary condition.

Robin (mixed) BC. This boundary condition is used when there is exchange with an environment outside the boundary. A linear combination of value and normal flux is prescribed on Γ_R:

$$\kappa \frac{\partial u}{\partial n} + \alpha\, u = \beta \quad \text{on } \Gamma_R,$$

where $\alpha > 0$ (required to ensure the bilinear form remains coercive and the problem is well-posed). Examples: convective heat transfer (Newton cooling), impedance BC in acoustics.

2.2.5 Initial Conditions

For time-dependent problems ($u = u(\mathbf{x}, t)$), the state of the system at time $t = 0$ must be specified. For a first-order problem in time:

$$u(\mathbf{x}, 0) = u_0(\mathbf{x}) \quad \text{in } \Omega.$$

Second-order hyperbolic problems (e.g. the wave equation) additionally require the initial velocity:

$$u_t(\mathbf{x}, 0) = v_0(\mathbf{x}) \quad \text{in } \Omega.$$

2.2.6 Strong Form of a General PDE Problem

Combining the PDE with boundary and initial conditions gives a *boundary-value problem* (BVP) or *initial-boundary-value problem* (IBVP). As a concrete example, a general parabolic diffusion problem reads:

$$\begin{cases} u_t - \nabla \cdot (\kappa \nabla u) = f & \text{in } \Omega \times (0, T), \\ u = g_D & \text{on } \Gamma_D \times (0, T), \\ \kappa \nabla u \cdot \mathbf{n} = g_N & \text{on } \Gamma_N \times (0, T), \\ u(\cdot, 0) = u_0 & \text{in } \Omega. \end{cases} \tag{2}$$

This *strong form* requires the solution u to satisfy (2) pointwise. As we shall see in Section 2.3, relaxing this pointwise requirement by multiplying by a test function and integrating leads to the *weak (variational) form*, which is the starting point for the finite element method.

2.3 Strong and Weak Formulations

Consider a domain $\Omega \subset \mathbb{R}^d$ ($d = 1, 2, 3$) with boundary $\partial\Omega = \partial\Omega$. A PDE in *strong form* prescribes the equation pointwise throughout Ω. For example, the **Poisson equation** with homogeneous Dirichlet boundary conditions reads:

$$-\Delta u = f \quad \text{in } \Omega, \qquad u = 0 \quad \text{on } \partial\Omega, \tag{3}$$

where $f \in L^2(\Omega)$ is a given source term and u is the unknown.

The strong form requires u to be twice continuously differentiable, a condition that is often too restrictive for discontinuous or rough data (for instance, a heat source which turns on abruptly, or an impulse force in vibration system). The *weak* (variational) form relaxes this requirement.

Why does (3) require this of u? The strong form requires $-\nabla^2 u = f$ to hold at every point in Ω. Since the Laplacian involves second-order partial derivatives, u must be twice differentiable for the left-hand side to even be defined pointwise. If f is rough or discontinuous, no such twice-differentiable solution may exist.

2.3.1 Deriving the Weak Form

We want to relax the pointwise requirement on u. The idea is to instead require the equation to hold *on average* against all functions in some suitable test space. We multiply both sides by an arbitrary *test function* v and integrate over Ω — if the equation holds for all such v, it captures the same information as the pointwise statement, but under weaker assumptions on u.

In practice, when the domain is discretised into a mesh, the test functions are typically low-order polynomials — often piecewise linear or quadratic — defined on each element. The global space V_0 is then approximated by a finite-dimensional subspace spanned by these local polynomial pieces, sewn together with appropriate continuity conditions at element boundaries.

To derive the weak form, multiply (3) by a *test function* $v \in V_0 = H_0^1(\Omega)$ (see Def. 2.3 on the following page) and integrate over the domain Ω:

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

Applying Green's first identity (integration by parts):

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

where $\mathbf{n}$ is the outward unit normal to $\partial\Omega$. Since $v = 0$ on $\partial\Omega$ (by the choice of V_0), the boundary term vanishes. The **weak formulation** of (3) then reads:

$$\text{Find } u \in V_0 \text{ such that } a(u, v) = L(v) \quad \forall v \in V_0, \tag{4}$$

where the *bilinear form* a and the *linear form* L are defined as

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

$$L(v) = \int_\Omega f \, v \, \mathrm{d}x. \tag{6}$$

2.4 Sobolev Spaces and Function Spaces

When we multiply a PDE by a test function and integrate, we must decide what space the test function lives in — and, relatedly, what space we seek our solution in. These choices are not arbitrary: they must be broad enough to accommodate physically meaningful solutions that may lack classical smoothness, yet structured enough to ensure the mathematics is well-posed. For instance, in the weak formulation of the Poisson problem we required ∇u and ∇v to be square-integrable, which means neither u nor v can be an arbitrary $L^2(\Omega)$ function. This motivates us to formalise the notion of a function space that tracks not just the size of a function, but also the size of its derivatives.

The natural setting for variational problems is a *Sobolev space*.

Definition 2.2 (Sobolev Space H^1)**.** *The Sobolev space $H^1(\Omega)$ consists of all square-integrable functions whose first-order weak derivatives are also square-integrable:*

$$H^1(\Omega) = \left\{ v \in L^2(\Omega) \mid \nabla v \in [L^2(\Omega)]^d \right\},$$

equipped with the norm $\|v\|_{H^1}^2 = \|v\|_{L^2}^2 + \|\nabla v\|_{L^2}^2$.

Definition 2.3 (Homogeneous Dirichlet Space H_0^1)**.** *$H_0^1(\Omega)$ is the subspace of $H^1(\Omega)$ whose elements satisfy $v = 0$ on $\partial\Omega$ in the trace sense.*

In FEniCSx, function spaces are created using the `functionspace` function and are characterised by the mesh, element family, and polynomial degree.

2.5 The Galerkin Finite Element Method

The Galerkin method converts the infinite-dimensional problem (4) into a finite-dimensional approximation by replacing V_0 with a finite-dimensional subspace $V_h \subset V_0$. In other words, rather than seeking a solution over the full infinite-dimensional space V_0, we instead seek an approximation expressed as a linear combination of a finite set of basis functions that span V_h.

2.5.1 Mesh and Triangulation

Let $\mathcal{T}_h$ be a *triangulation* (mesh) of Ω into non-overlapping elements (triangles in 2D, tetrahedra in 3D, intervals in 1D). The subscript h denotes the *mesh size* (characteristic element diameter).

2.5.2 Finite Element Basis

On each element $K \in \mathcal{T}_h$, a polynomial space is defined. For the standard continuous Lagrange element of degree p, the local space is $\mathcal{P}_p(K)$ – the space of polynomials of total degree at most p. The global finite element space is

$$V_h = \left\{ v_h \in C^0(\bar{\Omega}) \mid v_h|_K \in \mathcal{P}_p(K) \ \forall K \in \mathcal{T}_h \right\},$$

where continuity across element boundaries is enforced by shared degrees of freedom (DOFs). The homogeneous Dirichlet subspace is $V_{h,0} = V_h \cap H_0^1(\Omega)$.

2.5.3 Discrete Variational Problem

Replacing V_0 by V_h in (4) gives the *discrete variational problem*:

$$\text{Find } u_h \in V_h \text{ such that } a(u_h, v_h) = L(v_h) \quad \forall v_h \in V_h. \tag{7}$$

Writing $u_h = \sum_{j=1}^{N} U_j \phi_j$ in terms of the nodal basis $\{\phi_j\}_{j=1}^{N}$ of V_h, problem (7) becomes the linear system

$$\mathbf{A}\,\mathbf{U} = \mathbf{b}, \tag{8}$$

where

$$A_{ij} = a(\phi_j, \phi_i) = \int_\Omega \nabla\phi_j \cdot \nabla\phi_i \,\mathrm{d}x,$$
$$b_i = L(\phi_i) = \int_\Omega f\,\phi_i \,\mathrm{d}x.$$

The matrix $\mathbf{A}$ is the *stiffness matrix* and $\mathbf{b}$ is the *load vector*.

2.6 Boundary Conditions

2.6.1 Dirichlet Boundary Conditions

A **Dirichlet** (essential) boundary condition enforces the value of the unknown on a portion $\Gamma_D \subseteq \partial\Omega$:

$$u = g_D \quad \text{on } \Gamma_D.$$

In the discrete problem, Dirichlet conditions are imposed by modifying the linear system (8). In FEniCSx, this is done via `dirichletbc` objects passed to the `solve` function or the `LinearProblem` class.

2.6.2 Neumann Boundary Conditions

A **Neumann** (natural) boundary condition prescribes the normal flux on $\Gamma_N \subseteq \partial\Omega$:

$$\frac{\partial u}{\partial n} = g_N \quad \text{on } \Gamma_N.$$

Neumann conditions appear naturally in the weak form as a boundary integral added to the linear form:

$$L(v) = \int_\Omega f\, v \,\mathrm{d}x + \int_{\Gamma_N} g_N\, v \,\mathrm{d}s.$$

2.6.3 Robin (Mixed) Boundary Conditions

A **Robin** condition combines Dirichlet and Neumann conditions:

$$\frac{\partial u}{\partial n} + \alpha u = \beta \quad \text{on } \Gamma_R.$$

This introduces both a boundary bilinear term $\alpha \int_{\Gamma_R} uv \,\mathrm{d}s$ and a boundary linear term $\int_{\Gamma_R} \beta v \,\mathrm{d}s$ into the weak form.

2.7 A Priori Error Estimates

A fundamental result for conforming Galerkin methods is *Céa's lemma*, which states that the discrete solution is a quasi-best approximation in V_h:

Theorem 2.4 (Céa's Lemma)**.** *Let u be the solution to the continuous problem (4) and u_h the solution to the discrete problem (7). If $a(\cdot,\cdot)$ is continuous (bounded) and coercive on V with constants M and $\alpha > 0$, then*

$$\|u - u_h\|_{H^1} \leq \frac{M}{\alpha} \inf_{v_h \in V_h} \|u - v_h\|_{H^1}.$$

Combined with standard interpolation estimates, for a regular mesh and $u \in H^{p+1}(\Omega)$:

$$\|u - u_h\|_{H^1} \leq C\, h^p \, \|u\|_{H^{p+1}},$$

confirming that the error decreases at rate h^p as the mesh is refined.

3 Implementation Overview

This section maps the mathematical descriptions from Section 2 to FEniCSx constructs.

3.1 The FEniCSx Workflow

A typical FEniCSx workflow follows these steps:

1. **Create or import a mesh.**
2. **Define a function space** on the mesh.
3. **Define trial and test functions.**
4. **Specify boundary conditions.**
5. **Write the variational problem** using UFL.
6. **Assemble and solve** the linear system.
7. **Post-process and visualise** the solution.

3.2 Meshes in DOLFINx

FEniCSx provides built-in mesh generators for simple domains and supports import from external tools (Gmsh, XDMF/HDF5 files).

Listing 1: Creating a unit-square mesh in FEniCSx.

```python
import dolfinx.mesh
from mpi4py import MPI
import numpy as np

# Create a unit square mesh with 32x32 quadrilateral cells
domain = dolfinx.mesh.create_unit_square(
    MPI.COMM_WORLD, 32, 32,
    cell_type=dolfinx.mesh.CellType.triangle
)
```

Key mesh attributes available in DOLFINx:

- `domain.topology` – connectivity information (vertices, edges, faces, cells).
- `domain.geometry` – coordinate data.
- `domain.comm` – MPI communicator (enables parallel computation).

3.3 Function Spaces

A *finite element function space* V_h is created from a mesh and an element specification.

Listing 2: Creating a Lagrange function space.

```
from dolfinx.fem import functionspace
import basix.ufl

# Continuous Lagrange elements of degree 1 (
    piecewise linear)
V = functionspace(domain, ("Lagrange", 1))

# Degree-2 Lagrange for higher accuracy
V2 = functionspace(domain, ("Lagrange", 2))
```

The string `"Lagrange"` specifies the *element family*; the integer is the polynomial degree p. Other supported families include `"DG"` (discontinuous Galerkin), `"N1curl"` (Nédélec curl-conforming elements), and `"RT"` (Raviart–Thomas divergence-conforming elements).

3.4 Trial and Test Functions

In UFL, *trial functions* u represent the unknown solution and *test functions* v are used to form the variational problem.

Listing 3: Defining trial and test functions.

```
import ufl

u = ufl.TrialFunction(V)    # trial function (
    unknown)
v = ufl.TestFunction(V)     # test function (
    weight)
```

After solving, the numerical solution is stored in a `Function` object:

Listing 4: Creating a Function to hold the solution.

```
from dolfinx.fem import Function

uh = Function(V)  # will store the discrete
    solution u_h
```

3.5 Variational Forms in UFL

UFL provides a high-level syntax for expressing bilinear and linear forms that mirrors mathematical notation.

Listing 5: Expressing the Poisson variational form in UFL.

```
import ufl
from dolfinx.fem import Constant

f = Constant(domain, 1.0)  # right-hand side f = 1

# Bilinear form a(u, v)
a = ufl.inner(ufl.grad(u), ufl.grad(v)) * ufl.dx

# Linear form L(v)
L = f * v * ufl.dx
```

Table 1: Common UFL operators and their mathematical counterparts.

UFL	Math	Description
`ufl.grad(u)`	∇u	Gradient
`ufl.div(u)`	$\nabla \cdot \mathbf{u}$	Divergence
`ufl.curl(u)`	$\nabla \times \mathbf{u}$	Curl
`ufl.inner(a,b)`	$a \cdot b$	Inner product
`ufl.dot(a,b)`	$a \cdot b$	Dot product
`ufl.dx`	$\mathrm{d}x$	Volume integration measure
`ufl.ds`	$\mathrm{d}s$	Exterior facet measure
`ufl.dS`	$\mathrm{d}S$	Interior facet measure

3.6 Boundary Conditions in FEniCSx

Listing 6: Applying homogeneous Dirichlet boundary conditions.

```
import dolfinx.fem as fem
import numpy as np

# Locate all boundary facets
domain.topology.create_connectivity(
    domain.topology.dim - 1, domain.topology.dim
)
boundary_facets = dolfinx.mesh.
   exterior_facet_indices(domain.topology)
```

```
9
10 # Find DOFs on the boundary
11 boundary_dofs = fem.locate_dofs_topological(V,
       domain.topology.dim - 1,
12                                                  boundary
                                                       )
13
14 # Create homogeneous Dirichlet condition u = 0
15 bc = fem.dirichletbc(
16     fem.Constant(domain, 0.0),
17     boundary_dofs, V
18 )
```

For non-homogeneous conditions, pass a `Function` or a scalar/vecto value in place of `Constant(domain, 0.0)`.

3.7 Assembly and Solving

FEniCSx assembles the stiffness matrix and load vector from the UFL forms and solves the resulting linear system. The high-level `LinearProblem` interface handles assembly and solving in one call:

Listing 7: Solving with LinearProblem.

```
1 from dolfinx.fem.petsc import LinearProblem
2
3 problem = LinearProblem(a, L, bcs=[bc],
4                         petsc_options={"ksp_type"
                              : "preonly",
5                                        "pc_type":
                                             "lu"})
6 uh = problem.solve()
```

For more control (e.g. custom preconditioners, iterative solvers), the forms can be assembled separately:

Listing 8: Manual assembly using `assemble_matrix` and `assemble_vector`.

```
1 import dolfinx.fem.petsc as petsc_fem
2 from petsc4py import PETSc
3
4 A = petsc_fem.assemble_matrix(fem.form(a), bcs=[
      bc])
5 A.assemble()
6
7 b = petsc_fem.assemble_vector(fem.form(L))
8 petsc_fem.apply_lifting(b, [fem.form(a)], [[bc]])
```

```
b.ghostUpdate(addv=PETSc.InsertMode.ADD,
              mode=PETSc.ScatterMode.REVERSE)
petsc_fem.set_bc(b, [bc])
```

3.8 Visualisation

FEniCSx integrates with **ParaView** via XDMF/VTK output and supports in-notebook rendering with **pyvista**.

Listing 9: Writing solution to an XDMF file for ParaView.

```
from dolfinx.io import XDMFFile

with XDMFFile(domain.comm, "poisson.xdmf", "w")
    as file:
     file.write_mesh(domain)
     file.write_function(uh)
```

4 Example Applications

4.1 Example 1: Poisson Equation on the Unit Square

We solve the Poisson equation

$$-\Delta u = f \quad \text{in } \Omega = [0,1]^2, \qquad u = 0 \quad \text{on } \partial\Omega, \tag{9}$$

with the manufactured solution $u_{\text{exact}}(x, y) = \sin(\pi x)\sin(\pi y)$, which requires $f = 2\pi^2 \sin(\pi x)\sin(\pi y)$.

Listing 10: Complete FEniCSx script for the Poisson equation.

```
import numpy as np
from mpi4py import MPI
import dolfinx
import dolfinx.mesh
from dolfinx.fem import (functionspace, Function,
    Constant,
                         locate_dofs_topological
                            , dirichletbc, form)
from dolfinx.fem.petsc import LinearProblem
import ufl

# 1. Mesh
domain = dolfinx.mesh.create_unit_square(
    MPI.COMM_WORLD, 32, 32,
    cell_type=dolfinx.mesh.CellType.triangle
```

```
)

# 2. Function space
V = functionspace(domain, ("Lagrange", 1))

# 3. Trial and test functions
u = ufl.TrialFunction(V)
v = ufl.TestFunction(V)

# 4. Source term
x = ufl.SpatialCoordinate(domain)
f = 2.0 * ufl.pi**2 * ufl.sin(ufl.pi * x[0]) *
    ufl.sin(ufl.pi * x[1])

# 5. Variational forms
a = ufl.inner(ufl.grad(u), ufl.grad(v)) * ufl.dx
L = f * v * ufl.dx

# 6. Dirichlet boundary condition
domain.topology.create_connectivity(
    domain.topology.dim - 1, domain.topology.dim
)
boundary_facets = dolfinx.mesh.
    exterior_facet_indices(domain.topology)
boundary_dofs = locate_dofs_topological(
    V, domain.topology.dim - 1, boundary_facets
)
bc = dirichletbc(Constant(domain, 0.0),
    boundary_dofs, V)

# 7. Solve
problem = LinearProblem(a, L, bcs=[bc],
                        petsc_options={"ksp_type"
                            : "preonly",
                                       "pc_type":
                                           "lu"})
uh = problem.solve()

# 8. Compute L2 error
u_exact = ufl.sin(ufl.pi * x[0]) * ufl.sin(ufl.pi
    * x[1])
error_form = form((uh - u_exact)**2 * ufl.dx)
import dolfinx.fem
error_L2 = np.sqrt(
    domain.comm.allreduce(
        dolfinx.fem.assemble_scalar(error_form),
            op=MPI.SUM
```

```
    )
)
if domain.comm.rank == 0:
    print(f"L2 error: {error_L2:.3e}")
```

4.2 Example 2: Transient Heat Equation

The heat equation models time-dependent diffusion:

$$\frac{\partial u}{\partial t} - \Delta u = f \quad \text{in } \Omega \times (0,T], \qquad u = 0 \text{ on } \partial\Omega, \qquad u(\cdot,0) = u_0 \text{ in } \Omega. \tag{10}$$

Applying the **backward Euler** time-stepping scheme with step Δt:

$$u^{n+1} - u^n = \Delta t \left(\Delta u^{n+1} + f^{n+1} \right),$$

the weak form at each time step becomes:

$$\underbrace{\int_\Omega u^{n+1} v \, \mathrm{d}x + \Delta t \int_\Omega \nabla u^{n+1} \cdot \nabla v \, \mathrm{d}x}_{a(u^{n+1}, v)} = \underbrace{\int_\Omega u^n v \, \mathrm{d}x + \Delta t \int_\Omega f^{n+1} v \, \mathrm{d}x}_{L(v)}. \tag{11}$$

Listing 11: Time-stepping loop for the heat equation (FEniCSx).

```
# (Continuing from Example 1 setup -- mesh and V already created)
import numpy as np
from dolfinx.fem import Function, Constant
from dolfinx.fem.petsc import LinearProblem
import ufl

dt = 0.01
T  = 0.5

u_n = Function(V)     # solution at previous time step
u_n.interpolate(lambda x: np.sin(np.pi * x[0]) * np.sin(np.pi * x[1]))

u  = ufl.TrialFunction(V)
v  = ufl.TestFunction(V)
f  = Constant(domain, 0.0)

a = (u * v + dt * ufl.inner(ufl.grad(u), ufl.grad(v))) * ufl.dx
L = (u_n * v + dt * f * v) * ufl.dx
```

```
t = 0.0
while t < T:
    t += dt
    problem = LinearProblem(a, L, bcs=[bc],
                            petsc_options={"
                                ksp_type": "
                                preonly",
                                          "
                                              pc_typ
                                              ":
                                              "lu
                                              "})
    uh = problem.solve()
    u_n.x.array[:] = uh.x.array
```

4.3 Example 3: Linear Elasticity

Linear elasticity governs the displacement $\mathbf{u}$ of an elastic body:

$$-\nabla \cdot \boldsymbol{\sigma}(\mathbf{u}) = \mathbf{f} \quad \text{in } \Omega,$$

where the *stress tensor* $\boldsymbol{\sigma}$ is related to the *strain tensor* $\boldsymbol{\varepsilon}$ through *Hooke's law* (isotropic material):

$$\boldsymbol{\sigma}(\mathbf{u}) = \lambda(\nabla \cdot \mathbf{u})\mathbf{I} + 2\mu\,\boldsymbol{\varepsilon}(\mathbf{u}),$$
$$\boldsymbol{\varepsilon}(\mathbf{u}) = \tfrac{1}{2}\big(\nabla \mathbf{u} + (\nabla \mathbf{u})^T\big),$$

with Lamé parameters $\lambda, \mu > 0$.

The variational form for vector-valued displacement $\mathbf{u}$ is:

$$a(\mathbf{u}, \mathbf{v}) = \int_\Omega \boldsymbol{\sigma}(\mathbf{u}) : \boldsymbol{\varepsilon}(\mathbf{v})\,\mathrm{d}x = \int_\Omega \mathbf{f} \cdot \mathbf{v}\,\mathrm{d}x. \tag{12}$$

In FEniCSx, vector-valued problems use a *vector element*:

Listing 12: Setting up a vector function space for elasticity.

```
# Vector Lagrange space of degree 1 in 2D
V_vec = functionspace(domain, ("Lagrange", 1, (
    domain.geometry.dim,)))

lam = Constant(domain, 1.0)
mu  = Constant(domain, 1.0)

def epsilon(u):
    return ufl.sym(ufl.grad(u))
```

```
def sigma(u):
    return lam * ufl.div(u) * ufl.Identity(len(u)
        ) + 2 * mu * epsilon(u)

u = ufl.TrialFunction(V_vec)
v = ufl.TestFunction(V_vec)
f = ufl.as_vector([0.0, -1.0])

a = ufl.inner(sigma(u), epsilon(v)) * ufl.dx
L = ufl.inner(f, v) * ufl.dx
```

Conclusion

This document has introduced the theoretical foundations of the Finite Element Method as implemented in FEniCSx. The key concepts are:

- Derivation of the *weak (variational) form* by multiplying the strong PDE by test functions and integrating by parts.
- The *Galerkin discretisation* that restricts the problem to a finite-dimensional subspace V_h defined by a mesh and a polynomial degree.
- FEniCSx abstractions: `mesh`, `FunctionSpace`, `TrialFunction`/`T` `dirichletbc`, and UFL forms assembled by `LinearProblem`.
- Worked examples: Poisson equation, heat equation (transient), and linear elasticity.

For further reading, the official FEniCSx tutorial [3] and the FEniCS book [1] are excellent starting points.

References

[1] A. Logg, K.-A. Mardal, G. N. Wells (eds.), *Automated Solution of Differential Equations by the Finite Element Method*, Springer, 2012. doi:10.1007/978-3-642-23099-8

[2] M. W. Scroggs, J. S. Dokken, C. N. Richardson, G. N. Wells, Construction of Arbitrary Order Finite Element Degree-of-Freedom Maps on Polygonal and Polyhedral Cell Meshes, *ACM Transactions on Mathematical Software*, 48(2), 2022. doi:10.1145/3524456

[3] J. S. Dokken, *The FEniCSx Tutorial*, Simula Research Laboratory, 2023. https://jsdokken.com/dolfinx-tutorial/

[4] S. C. Brenner, L. R. Scott, *The Mathematical Theory of Finite Element Methods*, 3rd ed., Springer, 2008.

[5] A. Ern, J.-L. Guermond, *Theory and Practice of Finite Elements*, Springer, 2004.