DOLFINx: The next generation FEniCS problem solving environment

IGOR A. BARATTA, University of Cambridge, United Kingdom
JOSEPH P. DEAN, University of Cambridge, United Kingdom
JØRGEN S. DOKKEN, Simula Research Laboratory, Norway
MICHAL HABERA, Rafinex S.à r.l., Luxembourg and University of Luxembourg, Luxembourg
JACK S. HALE, University of Luxembourg, Luxembourg
CHRIS N. RICHARDSON, University of Cambridge, United Kingdom
MARIE E. ROGNES, Simula Research Laboratory, Norway
MATTHEW W. SCROGGS, University College London, United Kingdom
NATHAN SIME, Carnegie Institution for Science, USA
GARTH N. WELLS, University of Cambridge, United Kingdom

DOLFINx is the next generation problem solving environment from the FEniCS Project; it provides an expressive and performant framework for solving partial differential equations using the finite element method. Designed for parallelism from the ground up, DOLFINx supports arbitrary-degree finite elements on a wide range of (possibly curved) cell shapes across the full de Rham complex, as well as user-defined custom elements. It preserves the high level of mathematical abstraction associated with the FEniCS project, while enabling extensibility and fine-grained customization via user-defined element kernels and direct access to low-level data structures. At its core, DOLFINx adopts a modern, data-oriented and functional-inspired design and avoids the deep class hierarchies typical of more traditional finite element libraries. This approach yields a compact, maintainable, and flexible library that enables the development of high-performance programs in different languages and which enables efficient integration with other widely used numerical libraries and tools.

CCS Concepts: • **Mathematics of computing → Mathematical software**; *Numerical analysis*; *Partial differential equations.*

Additional Key Words and Phrases: partial differential equations, finite element methods, scientific software, parallel computing, domain-specific languages, automatic code generation

Authors' addresses: Igor A. Baratta, ia397@cam.ac.uk, Department of Engineering, University of Cambridge, Trumpington Street, Cambridge, United Kingdom, CB2 1PZ; Joseph P. Dean, jpd62@cam.ac.uk, Department of Engineering, University of Cambridge, Trumpington Street, Cambridge, United Kingdom, CB2 1PZ; Jørgen S. Dokken, dokken@simula.no, Simula Research Laboratory, Oslo, Norway, 0164; Michal Habera, michal.habera@uni.lu, Rafinex S.à r.l., 16 Ginzegaass, Senningerberg, Luxembourg, L-1670, Institute of Computational Engineering, Department of Engineering, Faculty of Science, Technology and Medicine and University of Luxembourg, 6 avenue de la Fonte, Esch-sur-Alzette, Luxembourg, L-4364; Jack S. Hale, jack.hale@uni.lu, Institute of Computational Engineering, Department of Engineering, Faculty of Science, Technology and Medicine, University of Luxembourg, 6 avenue de la Fonte, Esch-sur-Alzette, Luxembourg, L-4364; Chris N. Richardson, cnr12@cam.ac.uk, Institute for Energy and Environmental Flows, University of Cambridge, Bullard Laboratories, Madingley Road, Cambridge, United Kingdom, CB3 0EZ; Marie E. Rognes, meg@simula.no, Simula Research Laboratory, Oslo, Norway, 0164; Matthew W. Scroggs, matthew.scroggs.14@ucl.ac.uk, Department of Mathematics, University College London, Gower Street, London, United Kingdom, WC1E 6BT; Nathan Sime, nsime@carnegiescience.edu, Earth and Planets Laboratory, Carnegie Institution for Science, Washington, D.C., USA, 20015; Garth N. Wells, gnw20@cam.ac.uk, Department of Engineering, University of Cambridge, Trumpington Street, Cambridge, United Kingdom, CB2 1PZ.

1 INTRODUCTION

The finite element method emerged in the 1950s, driven by a need for accurately solving structural mechanics problems in the aeronautical industry [6, 29, 33, 61, 113]. The following decades saw the advent of general-purpose finite element software [17, 48, 57, 78, 108, 110] and more complex features, such as handling time-dependent and non-linear problems, adaptivity, enhanced user interfaces and input/output-integration [73, 109]. By the 1990s, object-oriented finite element libraries such as Diffpack and deal.II [7, 14, 24, 72] provided abstractions for finite element concepts, such as meshes and finite element spaces.

The FEniCS Project [2] pioneered a software pipeline consisting of a domain-specific language for defining weak forms (Unified Form Language (UFL)) [3], a finite element form compiler for generating low-level finite element code (FEniCS Form Compiler (FFC)) [76] and an automated finite element problem solving environment (Dynamic Object-oriented Library for FINite element computation (DOLFIN)) [75]. Higher levels of abstraction became available in other packages too, such as AceGen, Feel++, NGSolve and Firedrake [2, 3, 45, 55, 69, 70, 75–77, 90, 91, 101] thanks to automatic code generation. The new levels of abstraction in FEniCS enabled advanced features such as automated adjoints and derivatives of finite element models [43, 83], automated error control and adaptivity [97], shape optimization [40, 52, 100], uncertainty quantification [115], reduced order modeling [13], and hybrid finite element and neural network models [84], to mention but a few.

FEniCS has made a significant impact across fields in engineering and the natural sciences. We can point to examples in geophysics [114], biomechanics [54], biomedicine [35], structural mechanics [50, 95], fluid mechanics [85], inverse problems [88] and optimization [39]. In addition, FEniCS has been used as a foundational tool for developing packages for new numerical approaches, e.g. fictitious-domain finite element methods [25, 41], hybrid particle-mesh methods [79] as well as for the numerical verification of results from mathematical analysis of new discretization techniques [34, 60, 107] and preconditioners [21].

Despite these successes, a number of issues became evident with the design of the FEniCS components:

- *Age.* Parts of the FEniCS software were approaching twenty years old. Design decisions were based on technologies and software engineering paradigms of the time, with now apparent limitations.
- *Maintainability.* The FEniCS pipeline was becoming increasingly cumbersome to maintain due to its large code base, including core and non-core features, and design limitations.
- *Extensibility*. Overly encapsulated data storage and a lack of fine-grained control made it difficult to extend FEniCS non-intrusively to new methods when the available abstractions were not sufficient. This also presented barriers to experimentation on new hardware architectures, e.g. GPUs.
- *Performance.* Parallelism was not pervasive when DOLFIN was created and parallel support was retrofitted. Numerous algorithms were either not optimal or not parallel, resulting in an inconsistent user experience and hindering performance.

We present the finite element library DOLFINx and its context in the revised FEniCS ecosystem (FEniCSx). DOLFINx is a ground-up redevelopment of DOLFIN using modern C++ and idiomatic Python. The new architecture is fundamentally data-oriented and functional, rather than relying on object hierarchies and polymorphism. The main design goals are that DOLFINx should:

- Be suitable for experimental or research work without requiring intrusive extension of the core library with possibly immature and experimental technologies. This is achieved by ensuring that the library has a stable

public application programming interface (API) that provides access to low-level data structures alongside the higher-level functionality that we expect the majority of users will use.
- Integrate seamlessly with modern Python tools such as NumPy [53] and Numba [71] for implementing finite element kernels, creating meshes and finite element assembly algorithms, without a significant performance penalty over C++ implementations.
- Be designed for massively parallel computations. To ensure this, we no longer accept contributions that do not meet this goal.

1.1 Overview of FEniCSx

FEniCSx is composed of four libraries: UFL is a domain-specific language for expressing finite element forms [3]; Basix constructs finite elements and provides associated functionality [102]; FEniCSx Form Compiler (FFCx) generates C cell kernels from UFL forms; and DOLFINx manages finite element meshes, assembly over meshes, linear algebra data structures and solution, and input/output (I/O). UFL and FFCx are written in Python. Basix and DOLFINx are written in C++ with Python interfaces to the majority of their functionality. These libraries, and the modules of DOLFINx's Python interface are summarized in figure 1.

Figure 2 shows how the components of FEniCSx interact when a user develops an application in Python. The user defines elements with Basix and writes symbolic forms in UFL. The forms are then passed to FFCx which just-in-time generates element-level kernels that DOLFINx then executes across the mesh. Users can develop application codes in C++ (see figure 3). In this case, FFCx can be used for ahead-of-time generation of element-level kernels. DOLFINx is also designed to be used without FFCx generated code. Element-level kernels can be written in C or C++ and executed via the C++ or Python interfaces of DOLFINx, or written using the Python-based JIT compilation library Numba. A workflow using Numba is illustrated in figure 4 and discussed in section 8.3.

The layout of this paper is as follows: we introduce a typical application problem in section 2 and discuss the underlying design principles in section 3. We then cover the main features of DOLFINx, consisting of:

- A design that supports efficient parallel computation (section 4);
- Arbitrary degree finite elements on interval, triangle, quadrilateral, tetrahedral and hexahedral cells, including unstructured meshes without special ordering [102, 105] (section 5);
- User-defined custom finite elements (section 5);
- Scalable, distributed meshes with flat or curved cells (section 6);
- Parallel I/O with e.g. ADIOS2 [47] and HDF5 [111] (section 6);
- Interpolation of functions between different function spaces built on different finite elements, and on different (non-matching) meshes, including meshes using non-affine geometry (section 8);
- Code generation for finite element kernels from forms written using UFL [3] (section 8);
- Assembly and solvers using different floating point scalar types (section 8);
- Assembly into blocked and nested matrices, supporting the efficient and scalable [116] implementation of physics-based block preconditioners [42] (section 8);
- Assembly of custom element kernels written in Python using Numba [71] (section 8);
- Ability to non-intrusively support different linear algebra backends, e.g., NumPy, Portable, Extensible Toolkit for Scientific Computation (PETSc) [12], Trilinos [112] and Eigen [49] (section 9);

Finally, we provide concluding remarks in section 10.

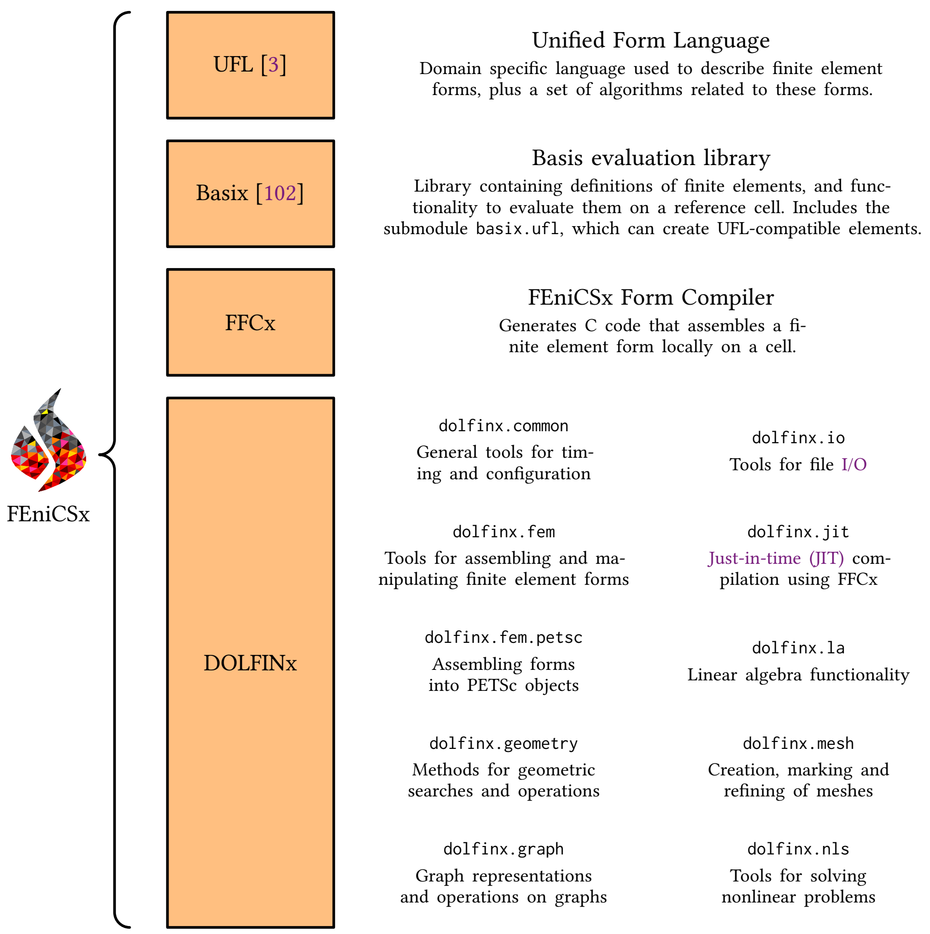

UFL [3]
Unified Form Language
Domain specific language used to describe finite element forms, plus a set of algorithms related to these forms.
Basix [102]
Basis evaluation library
Library containing definitions of finite elements, and functionality to evaluate them on a reference cell. Includes the submodule basix.ufl, which can create UFL-compatible elements.
FFCx
FEniCSx Form Compiler
Generates C code that assembles a finite element form locally on a cell.
FEniCSx
DOLFINx
dolfinx.common
General tools for timing and configuration
dolfinx.io
Tools for file I/O
dolfinx.fem
Tools for assembling and manipulating finite element forms
dolfinx.jit
Just-in-time (JIT) compilation using FFCx
dolfinx.fem.petsc
Assembling forms into PETSc objects
dolfinx.la
Linear algebra functionality
dolfinx.geometry
Methods for geometric searches and operations
dolfinx.mesh
Creation, marking and refining of meshes
dolfinx.graph
Graph representations and operations on graphs
dolfinx.nls
Tools for solving nonlinear problems

Fig. 1. The components that make up FEniCSx, and the Python submodules of DOLFINx.

1.2 Development, license and availability

FEniCSx development takes place at https://github.com/FEniCS. DOLFINx is released under the LGPL version 3 or later licenses. The other first-party components of the FEniCS Project, namely FFCx and UFL are also released under the LGPL version 3 or later licenses, while Basix is released under the MIT license. DOLFINx is available as Debian and Ubuntu packages, via the Conda [32] and Spack [44] package managers and in container images [51].

The DOLFINx codebase is remarkably compact for a finite element library with a wide range of functionality. The C++ library has approximately 27 000 lines of code. The Python interface has around 3600 lines of nanobind C++ binding code and 2100 lines of Python, excluding tests. The important dependencies, Basix, FFCx and UFL contain an additional 22 000 lines of C++ and 20 000 lines of Python in total.

The full source code for the code snippets in this paper is available in the supporting material [15].

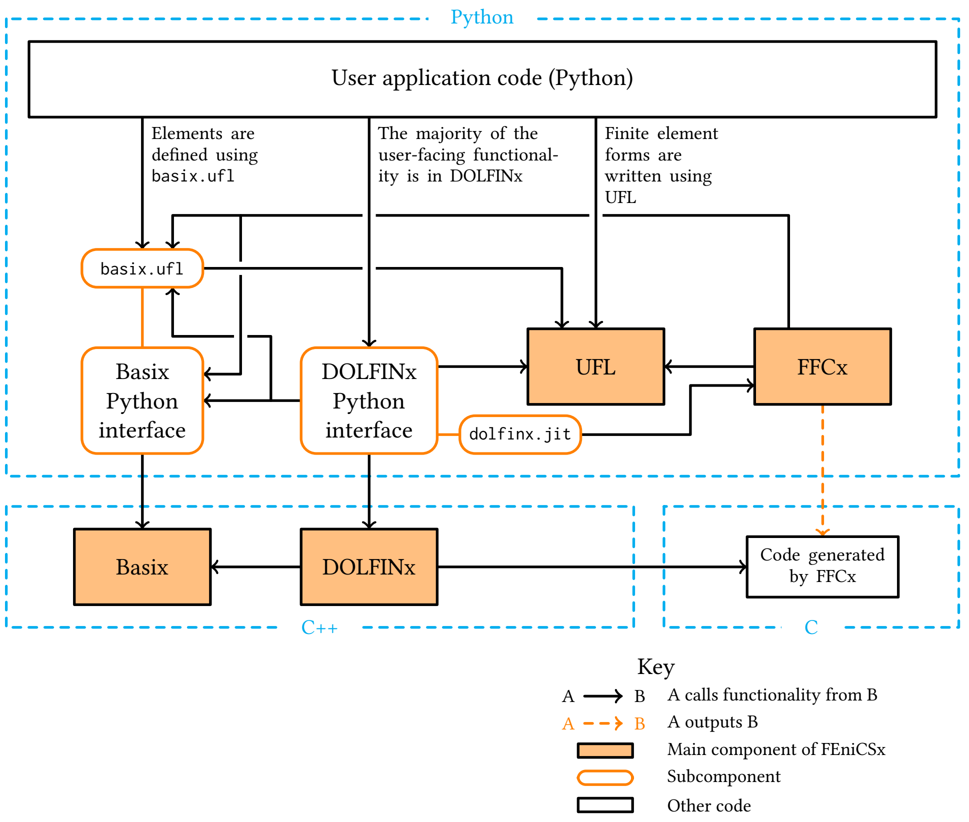

Python
User application code (Python)
Elements are defined using basix.ufl
The majority of the user-facing functionality is in DOLFINx
Finite element forms are written using UFL
basix.ufl
Basix Python interface
DOLFINx Python interface
UFL
FFCx
dolfinx.jit
Basix
DOLFINx
Code generated by FFCx
C++
C
Key
A → B A calls functionality from B
A ⇢ B A outputs B
Main component of FEniCSx
Subcomponent
Other code

Fig. 2. The interdependence of the core components of FEniCSx (DOLFINx, FFCx, Basix, and UFL) in a typical user application code written in Python.

2 A PROTOTYPE PROBLEM

The design of FEniCSx is centred around the mathematical abstraction of the finite element method. Let Ω be a domain with boundary $\partial\Omega$, and let $\mathcal{T}$ be a mesh of the domain. A large class of finite element problems can be formulated as follows [22]: find $u \in U$ such that

$$F(u; v) = 0 \quad \forall v \in V,$$

where U and V are finite element spaces and $F : U \times V \to \mathbb{C}$ is a functional. As an example, for the Helmholtz equation with homogeneous Neumann boundary conditions on $\partial\Omega$, we have

$$F(u, v) := \int_\Omega \nabla u \cdot \nabla \bar{v} - k^2 u\bar{v} - f\bar{v}\,\mathrm{d}x,$$

where k is the wave number, f is a prescribed function and $\bar{v}$ denotes the complex conjugate of v. Problems that are (sesqui-)linear in u can be rephrased as: find $u \in U$ such that

$$a(u, v) = L(v) \quad \forall v \in U,$$

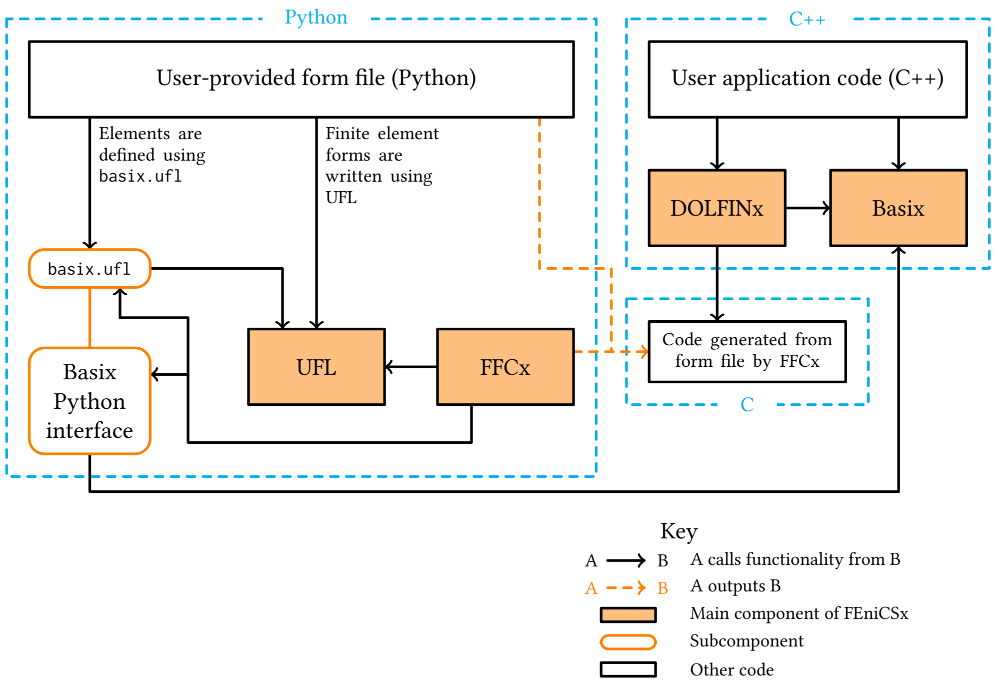

Python
C++
User-provided form file (Python)
User application code (C++)
Elements are defined using basix.ufl
Finite element forms are written using UFL
DOLFINx
Basix
basix.ufl
Code generated from form file by FFCx
C
Basix Python interface
UFL
FFCx
Key
A → B A calls functionality from B
A ⇢ B A outputs B
Main component of FEniCSx
Subcomponent
Other code

Fig. 3. The interdependence of the core components of FEniCSx (DOLFINx, FFCx, Basix, and UFL) in a typical user application code written primarily in C++. The C kernels are automatically generated from the Python UFL form file by FFCx.

where $a : U \times U \to \mathbb{C}$ and $L : U \to \mathbb{C}$, and we have set $U = V$ as is typical in the Galerkin method. For the Helmholtz equation, the *bilinear form* a and *linear form* L are defined by

$$a(u, v) := \int_\Omega \nabla u \cdot \nabla \bar{v} - k^2 u \bar{v} \, \mathrm{d}x \quad \text{and} \quad L(v) := \int_\Omega f \bar{v} \, \mathrm{d}x. \tag{1a}$$

Figure 5 shows a complete Python solver for this problem using DOLFINx, which is concise and expressive. The mesh is a unit cube with tetrahedral cells, and we use a degree 3 Lagrange finite element space.

3 DESIGN PRINCIPLES

DOLFINx adopts a data-oriented and functional design philosophy, where data structures are designed to be as simple and transparent as possible, and operations are implemented as pure functions acting on data. The design is modular, where high-level functionality is constructed from lower-level functions. The lower-level functions are available to the user for fine-grained control when required. This is a fundamentally different approach to DOLFIN, which relied heavily on object-oriented design patterns with deep class hierarchies. The design of DOLFINx is driven by high-performance, flexibility and usability, support for modern computer hardware (e.g. GPUs), language interoperability, and lessons learned from the legacy DOLFIN library. DOLFINx is distributed memory parallel from the ground up, with a focus on scalability and consistent functionality in serial and parallel.

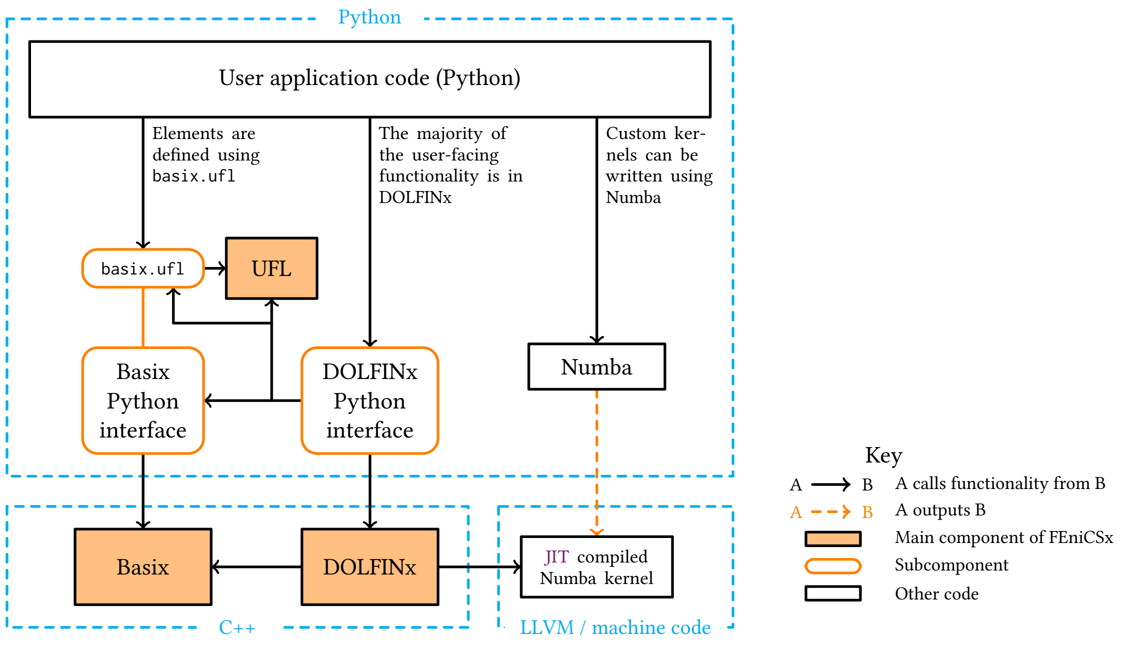

Python
User application code (Python)
Elements are defined using basix.ufl
The majority of the user-facing functionality is in DOLFINx
Custom kernels can be written using Numba
basix.ufl
UFL
Basix Python interface
DOLFINx Python interface
Numba
Basix
DOLFINx
JIT compiled Numba kernel
C++
LLVM / machine code
Key
A → B A calls functionality from B
A ⇢ B A outputs B
Main component of FEniCSx
Subcomponent
Other code

Fig. 4. A user workflow when using a custom assembler written in Numba.

3.1 Data-oriented design

DOLFINx's data-oriented design allows custom, low-level operations to be implemented without intrusive changes to the core library. This approach also lends itself to extensions to different hardware technologies, e.g. GPUs, without intrusive changes, since the underlying data can be easily accessed and operated on, with the layout of data described programmatically. Data encapsulation is used where appropriate, but not dogmatically, and the loss of some data encapsulation is often of little consequence in scientific computing as performance considerations often place strong constraints on how data can be stored and accessed.

3.2 Functional-style design

Where possible, DOLFINx favours pure functions over functions that modify their arguments; this aligns with the mathematical structure that we aim to follow, simplifies reasoning about code, and can have performance benefits in parallel [19]. Modern C++ features (e.g. move semantics and return value optimization) allow pure functions to often be used without a performance penalty. Rather than relying on polymorphism and virtual functions to provide user-defined functionality, DOLFINx instead accepts functions as arguments, leading to simpler and more flexible code, and consistent interfaces across languages. Both modern C++ and Python provide simple, native support for passing functions, including with captures.

```python
import numpy as np
from mpi4py import MPI

import dolfinx.fem.petsc
from dolfinx import fem, io, mesh
from ufl import SpatialCoordinate, TestFunction, TrialFunction, cos, dx, grad, inner

# Create mesh and define function space
msh = mesh.create_unit_cube(MPI.COMM_WORLD, 12, 16, 12)
V = fem.functionspace(msh, ("Lagrange", 3))

# Define variational problem
u, v = TrialFunction(V), TestFunction(V)
x, y, z = SpatialCoordinate(msh)
k = 4 * np.pi
f = (1.0 + 1.0j) * k**2 * cos(k * x) * cos(k * y)
a = inner(grad(u), grad(v)) * dx - k**2 * inner(u, v) * dx
L = inner(f, v) * dx

# Solve a(u, v) = L(v)
problem = dolfinx.fem.petsc.LinearProblem(
    a,
    L,
    petsc_options_prefix="helmholtz_",
    petsc_options={"ksp_type": "preonly", "pc_type": "lu", "ksp_error_if_not_converged": True},
)
uh = problem.solve()

# Save solution in VTX (.bp) format
with io.VTXWriter(msh.comm, "helmholtz.bp", [uh]) as vtx:
    vtx.write(0.0)
```

helmholtz.py

Fig. 5. DOLFINx solver for the Helmholtz problem on a unit cube with homogeneous Neumann boundary conditions.

3.3 Minimal code generation

DOLFINx (optionally) leverages code generation for finite element kernels for user-specified problems. In all other areas, DOLFINx prefers library implementations. A guiding principle is that all operations should be possible with traditional development techniques, complemented by targeted code generation tools. DOLFIN employed more extensive code generation e.g. for basis function tabulation, which led to increased library complexity, reduced maintainability, limited extensibility and customizability, and slowed the rate at which new features and performance improvements could be made. The DOLFINx design overcomes these issues without loss of the most appealing features of DOLFIN.

3.4 Extensibility

DOLFINx is designed to be extensible, supporting the development of new and experimental features outside of the core library. Broadly applicable features, once matured, are considered for incorporation into the library. This extensibility is made possible by the data- and function-oriented design. For example, GPU accelerated finite element operators implemented outside of the DOLFINx library can access the underlying mesh and degree of freedom (DOF) map data structures, as demonstrated in [94] for creating a scalable solver for GPU-accelerated supercomputers using HIP, and shown in a CUDA demo in section 8.3.2.

3.5 Language interoperability

The core of DOLFINx is written in C++20 [31], with an idiomatic Python interface built using nanobind [63]. Although most users use the Python interface, the C++ API is feature complete, and thanks to modern C++ features, closely mirrors the Python syntax.

The data-oriented and functional design of DOLFINx makes language interoperability straightforward. Most core data structures are based on contiguous arrays with known shape and strides, exposed through a low-level API. This allows the data to be shared between languages, without copying, as, for example, NumPy arrays in Python, or `std::span` or `std::mdspan` in C++23. Pure functions eliminate most memory management issues when passing data into the C++ library, and the object lifetime support in nanobind [63] supports memory management for data shared into the Python layer.

Functions can be passed through the Python/C++ language barrier. The user may provide, for example, a graph partitioning function, a function to interpolate into a finite element space, or a custom finite kernel for assembly. Python performance limitations can be overcome using NumPy's [53] vectorised operations, JIT compilation with Numba [71], or C Foreign Function Interface (CFFI)-compiled C-signature functions.

3.6 Fine-grained control

The legacy DOLFIN library provided mathematically expressive high-level interfaces, but because they were not constructed from lower-level, fine-grained interfaces, extension to problems that did not match the abstractions could be challenging. Moreover, their implicit nature could hide performance issues, making it easy for users to inadvertently write slow code – for example, by creating and destroying non-trivial objects like sparse matrices within a time loop, or triggering JIT compilation/cache lookups.

The DOLFINx API is designed to (i) expose the lower-level steps in the solution of a finite element problem in a fine-grained manner and (ii) be explicit rather than implicit, e.g. potentially expensive operations, steps that might not be required in all cases, or caching of objects, should be explicitly controlled by the user.

High-level interfaces that enhance usability are still used in DOLFINx, but they are composed from granular, low-level functions and are more explicit for performance transparency. Interfaces that were frequently (mis-)used in ways that led to poor performance have been removed.

4 PARALLELISM

DOLFINx is designed from the outset to support distributed memory parallelism using the Message Passing Interface (MPI), and parallel efficiency has been a major consideration in its design. The library has been used to solve problems with more than 1 trillion cells. MPI neighborhood collectives (or point-to-point operations that replicate neighborhood collectives where they are not supported by an implementation) are used extensively in DOLFINx as they are naturally suited to finite element computations and sparse linear algebra. Good mesh partitioning and data ordering strategies [65, 81, 98] keep communication neighborhoods small and independent of problem size and number of MPI ranks. Sometimes, the complete neighborhood communication graph might not be known i.e. a process might know which other process holds data that it requires, but not which processes require (some of) its data; for example, this can occur when constructing a mesh from an external file. When such neighborhood information is not available, we build the neighborhood communication graphs using the non-blocking NBX consensus algorithm [58]. DOLFINx does not use any MPI all-to-all communication functions.

A full description of the parallel design of DOLFINx would require extensive coverage, so we focus on two key components used throughout DOLFINx: index maps and scatterers.

4.1 Index maps

A common pattern is a range of contiguous indices that is partitioned across parallel processes, with each index having a global index and a local index, where the contiguous local indices start form 0 on each process. Each index has a unique owner, but some indices will be present on more than one process (ghost indices). Examples where such index ranges are useful include mesh entities (which are identified by an index) and degrees of freedom. Mesh vertices on the partition boundary of a distributed mesh are adjacent to cells on more than one process, and thus will appear on more than one process.

The parallel layout of index ranges in DOLFINx is described by `IndexMap` objects. An `IndexMap` instance partitions a set of $n_g \in \mathbb{N}$ contiguous indices across a set of processes P. Each process $p \in P$ owns a contiguous subset of indices $[i_p, i_p + n_p)$, where the offsets are given by $i_p = \sum_{i=0}^{p-1} n_p$ i.e. the first partition is owned by process 0, the next partition by process 1, and so on. The union of these owned partitions for a disjoint cover of $[0, n_g)$. An `IndexMap` also stores ghost indices and their owning rank. Indices in an index map have a local index, with owned indices numbered first, followed by ghost indices; the global index for owned entities is simply the local position (index) plus the process offset i_p, so storage is very light. Additionally, each process stores lists of 'source' and 'destination' ranks (processes that own ghost indices and processes that ghost owned indices, respectively). This data supports the creation of MPI neighborhood communicators. The use of index maps in the partitioning of meshes is discussed in section 6.1.

4.2 Scatterers

A `Scatterer` builds upon an `IndexMap` to communicate data associated with distributed objects. It provides forwards communication (sending data associated with owned indices to ghosting processes) and reverse communication (sending data associated with ghost indices to owning processes). An `IndexMap` only stores information for the reverse scatter (i.e. communication graph edges to processes that own ghosted indices), whereas a `Scatterer` has communication graphs for both forward and reverse communication patterns. Since these communication patterns are sparse (each process exchanges data with only a small number of other processes), we use MPI neighborhood collectives [1]. A `Scatterer` also supports any required packing and unpacking operations in-and-out of communication buffers. To support communication between GPUs, the `Scatterer` class is templated over the storage type and supports the passing of custom pack and unpack functions, allowing host/device transfers to be avoided.

We note that other libraries use similar concepts, e.g. index sets (`IS`) and vector scatters (`VecScatter`) in PETSc [12].

5 FINITE ELEMENTS AND BASIX

FEniCSx usually uses Basix [102] for finite element definitions. Basix provides a wide range of arbitrary-order finite elements on different cell shapes, including $H(\mathrm{div})$- and $H(\mathrm{curl})$-conforming elements. It offers fine-grained control over element construction and supports user-defined elements.

[1]DOLFINx provides an implementation based on MPI point-to-point communication functions to replicate neighborhood collective functionality for MPI implementations that do not support neighborhood collectives.

Element	Supported cells	Supported degrees
Lagrange	interval, triangle, quadrilateral, tetrahedron, hexahedron	$\geqslant 0$
Nédélec first kind (N1) [86]	triangle, quadrilateral, tetrahedron, hexahedron	$\geqslant 1$
Raviart–Thomas (RT) [92]	triangle, quadrilateral, tetrahedron, hexahedron	$\geqslant 1$
Nédélec second kind (N2) [87]	triangle, quadrilateral, tetrahedron, hexahedron	$\geqslant 1$
Brezzi–Douglas–Marini (BDM) [23]	triangle, quadrilateral, tetrahedron, hexahedron	$\geqslant 1$
Regge [27, 93]	triangle, tetrahedron	$\geqslant 0$
Hellan–Herrmann–Johnson (HHJ) [10]	triangle	$\geqslant 0$
Crouzeix–Raviart (CR) [36]	triangle, tetrahedron	1
discontinuous polynomial cubical (DPC) [9]	quadrilateral, hexahedron	$\geqslant 0$
serendipity [8]	interval, quadrilateral, hexahedron	$\geqslant 1$
bubble [67]	interval, triangle, quadrilateral, tetrahedron, hexahedron	$\geqslant 3$ (triangles), $\geqslant 4$ (tetrahedron), $\geqslant 2$ (other cells)
iso [18]	interval, triangle, quadrilateral, tetrahedron, hexahedron	$\geqslant 1$

Table 1. Elements that are supported in Basix and the cells and degrees for which they are supported. Note that we start the numbering of Raviart–Thomas (RT) and Nédélec first kind (N1) elements from degree 1. The lowest-degree RT elements that we refer to as RT degree 1 are referred to as RT degree 0 in some sources (for example, [11, 103]).

5.1 Basix supported elements

Basix provides functionality to define finite elements, evaluate basis functions at a set of points, and apply push-forward and pull-back operations to map between reference and physical cells. Basix also provides information about the layout of DOFs on each cell. It is written in C++ and includes a Python interface to its public API. Users can choose from a wide range of built-in elements (see table 1) or define their own custom elements (see section 5.2).

For Lagrange and discontinuous polynomial cubical (DPC) elements, the spacing of the points that define the element can be controlled. Tensor-product elements typically use the Gauss–Legendre (GL) or Gauss–Lobatto–Legendre (GLL) points. For non-equispaced simplex cells, Basix can position points using one of three methods: warped points [56], centroid-based points [20], and Isaac's method [62]. For elements that are defined using integral moments against Lagrange or DPC elements, variants control the choice of polynomials. By default, continuous higher-degree Lagrange elements use GL or GLL points and integral moments are taken with Legendre polynomials.

5.2 User-defined finite elements

Basix allows custom finite elements to be defined through its Python or C++ interfaces. Custom elements integrate seamlessly with UFL and FFCx in the same way as built-in elements. Before illustrating this with an example, we briefly recall the Ciarlet definition [28] of a finite element, which Basix follows:

Definition 5.1 (Ciarlet finite element). A finite element is defined by the triplet $(R, \mathcal{P}, \mathcal{L})$, where

- $R \subset \mathbb{R}^d$ is the reference cell, usually a polygon or polyhedron;
- $\mathcal{P}$ is a finite dimensional polynomial space on R of dimension n;
- $\mathcal{L} := \{l_0, \dots, l_{n-1}\}$ is a basis of the dual space $\mathcal{P}^* := \{f : \mathcal{P} \to \mathbb{R} \,|\, f \text{ is linear}\}$.

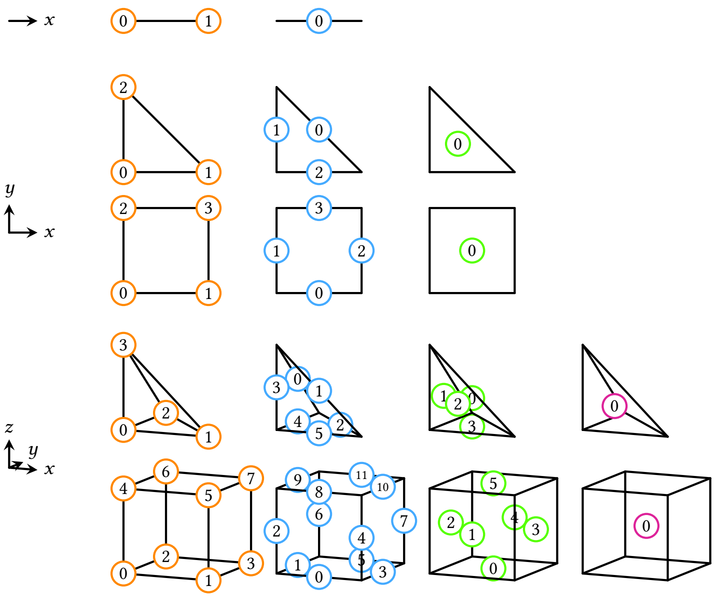


Fig. 6. The numbering of local entities on each cell type. These figures are taken from [104].

The basis functions $\{\phi_0, \dots, \phi_{n-1}\}$ of the space $\mathcal{P}$ are defined by

$$l_i(\phi_j) = \begin{cases} 1 & i = j, \\ 0 & i \neq j. \end{cases} \tag{2}$$

Each functional $l_i \in \mathcal{L}$ is a DOF and is associated with a sub-entity of the cell. The numbering of local entities for a range of cell types used in Basix is shown in figure 6.

To demonstrate the construction of custom elements, we consider the degree 1 tiniest tensor (TNT) element [30]. The TNT element is defined by:

- $R_{\text{TNT}} := [0, 1]^2$ is the unit square.
- $\mathcal{P}_{\text{TNT}} := \operatorname{span}\{1, x, y, xy, x^2, x^2y, y^2, xy^2\}$.
- $\mathcal{L}_{\text{TNT}} := \{l_0, \dots, l_7\}$, where l_0 to l_3 are point evaluations at vertices 0 to 3 and l_4 to l_7 are integrals of the function on edges 0 to 3. Each functional is associated with the vertex or edge that is used to define it.

The code to create this element in Basix from Python (C++ can also be used) is shown in figure 7. The polynomial space $\mathcal{P}_{\text{TNT}}$ is specified by a coefficient matrix expressing a basis of $\mathcal{P}_{\text{TNT}}$ in terms of a set of orthogonal polynomials on the quadrilateral. The degree 2 orthogonal polynomials on a quadrilateral are ordered so that the nth polynomial is in the span of the first $n + 1$ elements of the set $\{1, y, y^2, x, xy, xy^2, x^2, x^2y, x^2y^2\}$. The order that Basix uses for arbitrary-degree orthogonal polynomials on any cell type can be found in the Basix documentation. The elements of the dual basis $\mathcal{L}_{\text{TNT}}$ that are associated with each sub-entity of the reference cell are defined by providing a set of points and weights. These define each functional: the functional can be applied to a function by evaluating the function at the points, multiplying by the corresponding weights, and then summing. For example, functionals l_4 to l_7 use a set of quadrature points and weights. The Python element is then created with `basix.ufl.create_custom_element`, and any of the functionality of Basix or UFL can immediately be used.

```python
1  import numpy as np
2  
3  import basix
4  import basix.ufl
5  
6  # Coefficients defining the polynomial space in terms of orthogonal polynomials on the cell
7  wcoeffs = np.eye(8, 9)
8  
9  geometry = basix.geometry(basix.CellType.quadrilateral)
10 topology = basix.topology(basix.CellType.quadrilateral)
11 
12 # Points and weights used to define functionals on each sub-entity of the cell
13 x = [[], [], [], []]
14 M = [[], [], [], []]
15 
16 # Associate one point evaluation with each vertex
17 for v in topology[0]:
18     x[0].append(np.array(geometry[v]))
19     M[0].append(np.ones([1, 1, 1, 1]))
20 
21 # Associate an integral with each edge
22 pts, wts = basix.make_quadrature(basix.CellType.interval, 1)
23 for e in topology[1]:
24     v0 = geometry[e[0]]
25     v1 = geometry[e[1]]
26     # Map points on the reference interval to each edge of the quadrilateral
27     edge_pts = np.array([v0 + p * (v1 - v0) for p in pts])
28     x[1].append(edge_pts)
29 
30     mat = np.zeros((1, 1, pts.shape[0], 1))
31     mat[0, 0, :, 0] = wts
32     M[1].append(mat)
33 
34 # Associate 0 DOFs with the interior of the cell
35 x[2].append(np.zeros([0, 2]))
36 M[2].append(np.zeros([0, 1, 0, 1]))
37 
38 tnt_degree1 = basix.ufl.custom_element(
39     basix.CellType.quadrilateral, [], wcoeffs, x, M, 0,
40     basix.MapType.identity, basix.SobolevSpace.H1, False, 1, 2)
```

basix_element.py

Fig. 7. Creating a degree 1 TNT element in Basix. The polynomial space $\mathcal{P}_{\mathrm{TNT}}$ is defined by `wcoeffs` (line 7); in this example `wcoeffs` is an 8×9 matrix that is an 8×8 identity plus an extra column of zeros. The functionals in $\mathcal{L}_{\mathrm{TNT}}$ are defined by a set of points `x` and a 4-dimensional array `M` for each sub-entity of the cell (lines 13 to 36). The element is initialized in lines 38–40 using Basix's UFL submodule; the element can then be used directly with UFL.

6 MESHES

DOLFINx supports distributed, unstructured meshes composed of simplex or tensor-product cells with arbitrary overlaps (support for mixed-topology meshes and multiple geometric map types is experimental). Local and uniform refinement of distributed simplex-cell meshes is provided using the algorithm from [89], which relies solely on scalable local neighborhood communication. 'Sub-meshes' can be created from subsets of mesh entities. Since a sub-mesh is a full `Mesh` object, any functionality that works on meshes also works on sub-meshes. Sub-meshes are useful for multiphysics and coupled problems where different fields are defined over different domains (see [37]). Mesh representation follows the graph-centric approach described in [74] for non-distributed meshes (see also [68]), but the algorithms and data structures differ. A full discussion of mesh data structures, algorithms and parallel treatment would require extensive treatment; we limit our discussion to some main points.

Mesh topology and geometry are separated in DOLFINx; a `Mesh` consists of (i) a `Topology` and (ii) a `Geometry` object. In its simplest form, a mesh `Topology` holds the cells of a mesh, with a cell defined by its vertices. Algorithms are provided that can number (in parallel) entities of other topological dimensions (i.e. edges and faces), which are required when, for example, creating high-order function spaces. A mesh `Topology` can also compute and store the connectivity between entities of different topological dimensions.

A mesh `Geometry` describes the geometry of the cells; it consists of (i) a finite element (typically Lagrange) that provides the map for how a cell is transformed from a reference configuration to a physical configuration, (ii) the coordinates of the geometric DOFs, and (iii) a DOF map that for each cell gives the indices of the geometric DOFs. `Geometry` is templated over the float type used to represent the mesh geometry.

6.1 Creating meshes

Distributed meshes can be created through interfaces at different levels of abstraction. At the lowest level, users can create a `Topology` and a `Geometry` directly, but this is uncommon since all partitioning, distribution, separation of geometric and topological data, and local data reordering for data locality must have already been applied.

Most users instead call the higher-level `create_mesh` function, which facilitates fully-distributed and memory-scalable mesh creation. The following C++ example creates a mesh of two quadrilateral cells.

```cpp
auto e = std::make_shared<basix::FiniteElement<float>>(
    basix::create_element<float>(
        basix::element::family::P, basix::cell::type::quadrilateral, 1,
        basix::element::lagrange_variant::unset,
        basix::element::dpc_variant::unset, false));
fem::CoordinateElement<float> cmap(e);
std::vector<float> x{0.0, 0.0, 1.0, 0.0, 2.0, 0.0,
                     0.0, 1.0, 1.0, 1.0, 2.0, 1.0};
std::vector<std::int64_t> cells{0, 1, 3, 4, 1, 2, 4, 5};
mesh::Mesh mesh
    = mesh::create_mesh(MPI_COMM_WORLD, cells, cmap, x, {x.size() / 2, 2},
                        mesh::GhostMode::none);
```

mesh.cpp

The float type for the mesh geometry is inferred from the coordinate array `x`, and embedding dimension is determined from the coordinate array's shape (a quadrilateral cell mesh could be embedded in two- or three-dimensions). The input data may be distributed across any of the MPI processes or reside on one process. Internally, DOLFINx constructs a distributed dual graph, partitions the cells, reorders them for data locality, and then creates the distributed mesh.

By default, partitioning is handled by a graph partitioning library e.g. PT-SCOTCH [26], ParMETIS [64] or KaHiP [99], but the functional-oriented design of DOLFINx easily allows users to provide their own partitioning function:

```cpp
auto e = std::make_shared<basix::FiniteElement<float>>(
    basix::create_element<float>(
        basix::element::family::P, basix::cell::type::quadrilateral, 1,
        basix::element::lagrange_variant::unset,
        basix::element::dpc_variant::unset, false));
fem::CoordinateElement<float> cmap(e);
```

```cpp
  std::size_t rank = dolfinx::MPI::rank(comm);
  std::vector<float> x;
  std::vector<std::int64_t> cells;
  // We assign data on only one process
  if (rank == 0)
  {
    x = {0.0, 0.0, 1.0, 0.0, 2.0, 0.0,
         0.0, 1.0, 1.0, 1.0, 2.0, 1.0};
    cells = {0, 1, 4, 3, 1, 2, 5, 4};
  }

  auto part_fn
      = [](MPI_Comm comm, int nparts, std::vector<mesh::CellType> cell_type,
           const std::vector<std::span<const std::int64_t>>& cells)
      -> graph::AdjacencyList<std::int32_t>
  {
    // Compute destination rank(s) for each cell in cells. Cells sent to
    // more than one rank will be ghosts on some ranks. Return
    // destinations for each cell as an adjacency list.

    // Leave cells on current rank
    std::size_t num_vertices_per_cell = dolfinx::mesh::num_cell_vertices(cell_type.front());
    std::vector<std::int32_t> dest(cells.front().size()/num_vertices_per_cell, dolfinx::MPI::rank(
 comm));
    std::vector<std::int32_t> offsets(dest.size() + 1);
    std::iota(offsets.begin(), offsets.end(), 0);
    return graph::AdjacencyList<std::int32_t>(dest, offsets);
  };
  mesh::Mesh mesh = mesh::create_mesh(comm, comm, cells, cmap, comm, x,
                                      {x.size() / gdim, gdim}, part_fn);
```

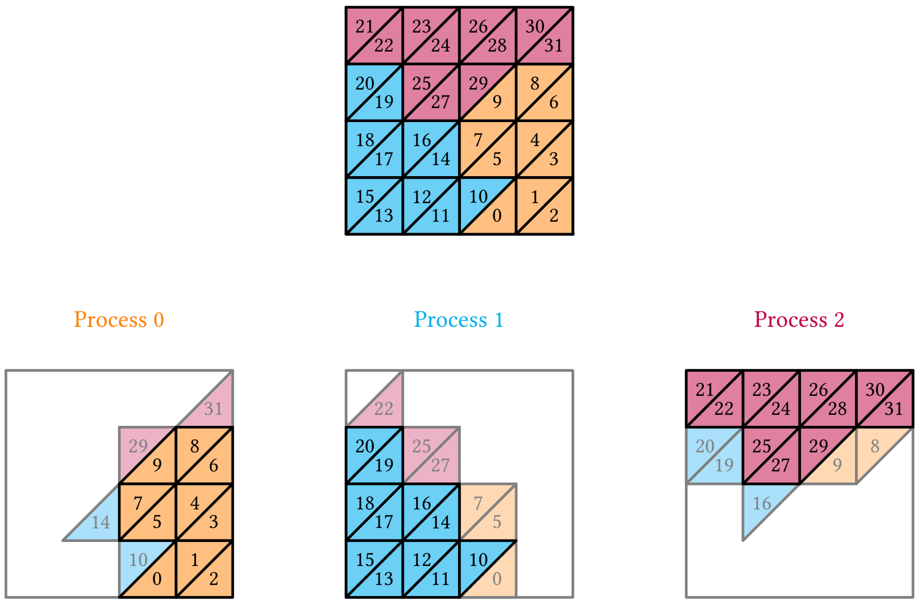
mesh_custom_part.cpp

Similarly, users can pass custom cell re-ordering functions. The Python interface for creating a mesh and providing custom partitioning and reordering functions is very similar.

The DOLFINx file input interfaces read topology and geometry data from parallel file formats, with each process reading a chunk of data, and then call `create_mesh`. For unsupported formats, users can read the data themselves, apply any required permutation to match DOLFINx's conventions, and then call `create_mesh`. Programmatic mesh generation libraries, such as Gmsh [46], can also be used for distributed mesh generation without intermediate output to disk. DOLFINx also provides built-in mesh generators for some simple geometric shapes.

6.2 Arranging and accessing mesh data

A `Topology` object stores an `IndexMap` for each created entity type that describes how the entities are numbered and distributed (see section 4.1). Similarly, a `Geometry` object has an index map to describe the ownership of the geometry DOFs across processes. For example, figure 8 shows a mesh distributed across three processes and the index maps for the cells on each process. Similarly, figure 9 illustrates the distribution of vertices of the mesh and the corresponding index maps.

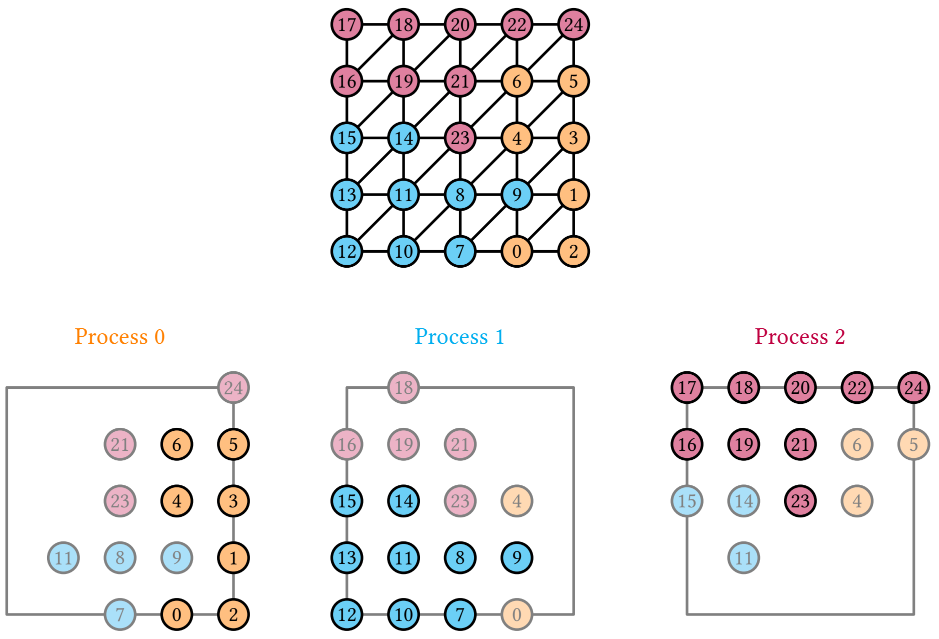


Local index	Global index	
0	0	
1	1	
2	2	
3	3	
4	4	Owned
5	5	
6	6	
7	7	
8	8	
9	9	
10	10	
11	14	Ghosts
12	29	
13	31	

Local index	Global index	
0	10	
1	11	
2	12	
3	13	
4	14	
5	15	Owned
6	16	
7	17	
8	18	
9	19	
10	20	
11	0	
12	5	
13	7	Ghosts
14	27	
15	25	
16	22	

Local index	Global index	
0	21	
1	22	
2	23	
3	24	
4	25	
5	26	Owned
6	27	
7	28	
8	29	
9	30	
10	31	
11	16	
12	19	
13	20	Ghosts
14	9	
15	8	

Fig. 8. The cell index map for a mesh of triangles distributed over three processes. Each cell is owned by exactly one process but may appear as a ghost on other processes. Ghosts are stored at the end of the index map array. Here, cells that are not owned by a particular process but that share a facet with an owned cell have been added as ghost cells. Color is added as a visual aid to indicate the process owning each index.



Local index	Global index	
0	0	Locally owned
1	1	
2	2	
3	3	
4	4	
5	5	
6	6	
7	7	Ghosts
8	8	
9	9	
10	23	
11	11	
12	21	
13	24	

Local index	Global index	
0	7	Locally owned
1	8	
2	9	
3	10	
4	11	
5	12	
6	13	
7	14	
8	15	
9	23	Ghosts
10	16	
11	19	
12	0	
13	4	
14	21	
15	18	

Local index	Global index	
0	16	Locally owned
1	17	
2	18	
3	19	
4	20	
5	21	
6	22	
7	23	
8	24	
9	6	Ghosts
10	5	
11	14	
12	11	
13	15	
14	4	

Fig. 9. The index maps for the vertices of the mesh shown in figure 8. Each process is aware of the vertices of all of the cells in its index map in figure 8. Color is added as a visual aid to indicate the process owning each index.

Connectivities (incidence relationships) between entities in DOLFINx follow a graph-centric approach. The (d_0, d_1) connectivity is a bipartite graph linking d_0-dimensional entities to incident d_1-dimensional entities. Adjacency lists provide a natural data structure for this, which are stored as simply a data array and an array of offsets. This conforms to our data-oriented approach and allows mesh data to be operated on efficiently and shared between libraries without overhead. The following example illustrates computing and accessing the vertex-to-cell connectivity data, which could then be copied to a GPU for on-device computation.

```cpp
auto mesh = dolfinx::mesh::create_rectangle<double>(
    comm, {{{0.0, 0.0}, {2.0, 1.0}}}, {32, 16}, dolfinx::mesh::CellType::triangle);

int tdim = mesh.topology()->dim();
mesh.topology()->create_connectivity(
    0, tdim); // Connectivity vertices -> cells
auto c = mesh.topology()->connectivity(0, tdim);

// Access underlying connectivity data
std::span<const std::int32_t> c_data = c->array();
std::span<const std::int32_t> c_offsets = c->offsets();

// Copy data to GPU for operations on the device
// ...
```

mesh_data.cpp

6.3 Cell orientation encoding

A distinguishing feature of DOLFINx is its support for high-order finite element spaces on general unstructured meshes without requiring special cell orderings. This is enabled by computing, for each cell, the local orientation of each sub-entity relative to a reference orientation and encoding this information into a single 32-bit integer for each cell. In DOLFINx, the vertices of a cell each have a local index based on the reference numbering of the cell (see figure 6) and a global index (the vertex's index in the mesh topology). Edges are oriented from low-to-high vertex index. We use one bit per edge to store whether the local and global edge orientations agree. For faces, the lowest-indexed vertex is used as an origin, and the face is oriented based on the direction towards the next-lowest-indexed vertex connected by an edge. For each face, two bits encode the number of rotations needed to align the local and global origins, and a third bit encodes whether a reflection is needed. (For faces with more than four vertices, more than two bits would be required for rotations.) Tetrahedral, square-based pyramid, triangular prism, and hexahedral cells require 18, 23, 24, and 30 bits, respectively, to encode this information, and so can be stored in a single 32-bit integer per cell.

7 FUNCTION SPACES, FUNCTIONS, AND INTERPOLATION

DOLFINx supports the creation of arbitrary-degree function spaces for a wide range of finite elements. Functions can be interpolated into finite element spaces and from one space to another. Moreover, user-defined expressions can be written in UFL and used to postprocess simulation results.

7.1 Degree-of-freedom maps

A DOF map specifies which global DOF numbers are associated with the local DOFs on each cell. DOLFINx constructs distributed DOF maps from the mesh topology and the layout of DOFs on the reference cell. The DOF layout is element-specific and associates each DOF of an element with either a vertex, an edge, a face, or the cell. This information is provided by Basix and is encapsulated in the `ElementDofLayout` class. The mesh topology is used to ensure that the local DOFs associated with mesh entities that are shared by multiple cells are all assigned the same global DOF number. A `DofMap` object stores the DOF map data in an adjacency list.

7.2 Function spaces and finite element functions

Modeling the mathematical structure of a function space, DOLFINx function spaces consist of a mesh (the domain), an element (the local space) and a DOF map (required global regularity). In the simplest case, a function space is created from a mesh and a finite element type, e.g. a (continuous) Lagrange space of degree 2 can be created by:

```python
msh = mesh.create_mesh(...)
V = fem.functionspace(msh, ("Lagrange", 2))
```

A strength is the capability to create function spaces on mixed elements. The snippet below illustrates the creation of a mixed finite element space composed of Raviart–Thomas and discontinuous Lagrange spaces.

```python
msh = mesh.create_mesh(...)
E0 = basix.ufl.element("Raviart-Thomas", msh.basix_cell(), 3)
E1 = basix.ufl.element("DG", msh.basix_cell(), 2)
E = basix.ufl.mixed_element([E0, E1])
V = fem.functionspace(msh, E)
```

Mixed elements can be nested arbitrarily. The degree-of-freedom map is constructed from data associated with the elements. It is also possible to construct a function space with a user-provided DOF map.

A number of operations on function spaces are supported, including extracting subspaces (views) and collapsing subspaces (creating a new space from a view). One of the most powerful features is interpolation into and between spaces, which is presented in section 7.3.

Finite element functions can be created on a given function space. A `Function` is an object that holds a function space and the DOFs coefficients associated with the function. Functions with different scalar types for the coefficient values can be created as follows:

```python
V = fem.functionspace(...)
u0 = fem.Function(V, dtype=np.float64)
u1 = fem.Function(V, dtype=np.complex128)
```

Analogous to function spaces, sub-functions (views) can be extracted and 'collapsed', and interpolated to and from.

7.3 Interpolation

DOLFINx supports the interpolation of user-provided expressions into finite element spaces. This builds on the Basix infrastructure for defining finite elements via the dual basis.

Let f be an expression that we want to interpolate into a finite element space. The (local) interpolant of f is denoted by $\tilde{f} \in \mathcal{P}$ and is defined by

$$\tilde{f}(\boldsymbol{x}) = \sum_{i=0}^{n-1} l_i(f)\phi_i(\boldsymbol{x}),$$

where $l_i \in \mathcal{L}$ is the element dual basis and ϕ_i is the (primal) basis that spans the finite element space $\mathcal{P}$.

The snippet below illustrates the interpolation of the expression $f(x) = \sin \pi x \sin \pi y$ into a continuous Lagrange space of degree 3. It also highlights the functional design – the user provides the mathematical expression to apply to the (coordinate) data. Vectorized evaluation avoids performance issues that can affect interpreted languages, and it also permits the library to make decisions on how many points should be evaluated each time.

```
V = functionspace(msh, ("Lagrange", 3))
u = Function(V)
u.interpolate(lambda x: np.sin(np.pi * x[0]) * np.sin(np.pi * x[1]))
```

7.3.1 Point evaluations versus integral moment definitions of the dual basis. To obtain optimal interpolation accuracy for elements with integral moment DOFs, DOLFINx uses quadrature evaluation of integral moments by default, rather than simple point evaluations, as often used in other finite element libraries (see e.g. [4, 38, 66, 82]).

Several finite elements have integral moment DOFs, including N1 [86], Nédélec second kind (N2) [87], RT [92], serendipity [8], and Brezzi–Douglas–Marini (BDM) [23] elements among others. For example, the DOFs for a degree p N1 element on a tetrahedron are:

- on each edge, moments of the tangential components against a basis of the set of degree $p-1$ polynomials on an interval;
- (if $p > 1$) on each face, moments of each component against a basis of the set of degree $p-2$ polynomials on a triangle;
- (if $p > 2$) on the interior of the cell, moments of each component against a basis of the set of degree $p-3$ polynomials on a tetrahedron.

In the lowest-order case ($p = 1$), the DOFs are simply integral moments against a constant on each edge, so they can be replaced by point evaluations of the tangential component of the function at the midpoint of each edge. However, for higher-degree elements, replacing integral moment evaluation with point evaluations results in sub-optimal interpolation accuracy. We demonstrate this by comparing the default (integral-moment-based) N1 element in DOLFINx with a point-evaluation-based N1 element (implemented using Basix's custom element interface, see section 5.2). We interpolate the function $\boldsymbol{g}(x, y, z) = (\sin(8x), 2^y \cos(3z), x)$ and show the interpolation error in figure 10 as a function of both cell size h and polynomial degree. We see that the integral-moment elements converge at the expected theoretical rate under h-refinement, whereas point-evaluation elements are sub-optimal by one degree. Moreover, for a given polynomial degree, integral-moment-based elements are one or two orders of magnitude more accurate.

Despite the added complexity of proper integral-moment functional evaluation, the DOLFINx interface is very straightforward. For example, a function can be interpolated in a degree 3 N1 space as follows:

```
g_h = dolfinx.fem.Function(nedelec)
g_h.interpolate(lambda x: (np.sin(8*x[0]), 2**x[1]*np.cos(3*x[2]), x[0]))
```

7.3.2 Expressions. Expressions in DOLFINx allow symbolic UFL expressions to be evaluated. This is particularly helpful when postprocessing simulation data to compute derived quantities, such as the stress or strain from a displacement field. For example, figure 11 creates an `Expression` to calculate the von Mises stress, represented in a discontinuous Lagrange finite element space.

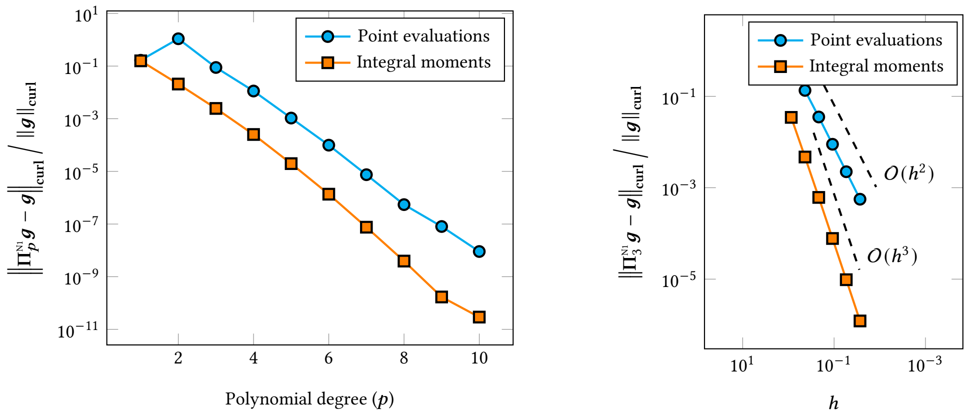


Fig. 10. The interpolation error when the function $g(x, y, z) = (\sin(8x), 2^y \cos(3z), x)$ is interpolated into a space of N1 elements on a tetrahedron defined using either point evaluation (circles) or integral moment (squares) functionals. The left plot shows the errors as we increase p on a mesh of the unit cube with 750 cells. The right plot shows the errors for a degree 3 element as we decrease h, where the dashed lines show $O(h^2)$ and $O(h^3)$ convergence.

To see how this works, we consider the simpler case of computing the derivative of a linear continuous Lagrange function $\tilde{f} \in \mathcal{P}_1$ in the x_1-direction, yielding a scalar-valued degree 0 Lagrange function $\tilde{g} \in \mathcal{P}_0$. We have

$$\tilde{g}(x) = \partial_{x_1} \tilde{f}(x) = \sum_{i=0}^{m-1} \psi_i(x) l_i \left(\sum_{j=0}^{n-1} \partial_{x_1} \phi_j f_j \right) = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} \psi_i(x) l_i(\partial_{x_1} \phi_j) f_j. \tag{3}$$

Here l_i are the m elements of the dual basis of $\mathcal{P}_0$ with associated vector-valued basis functions ψ_i, ϕ_j are the n basis functions associated with $\mathcal{P}_1$ and f_j are the coefficients of $\tilde{f}$. Computing $\tilde{g}$ (or equivalently, its coefficients g_i) requires (i) the evaluation of the derivatives of the basis functions ϕ_i at functionals in the dual basis of $\tilde{g}$, (ii) their contraction with the coefficients of $\tilde{f}$, and (iii) assembly into the global vector of coefficients g_i associated with $\tilde{g}$. FFCx can generate a local expression kernel that performs the first two operations, and DOLFINx provides the necessary routines to execute expression kernels over cells.

7.3.3 Interpolation between elements. Fast interpolation between different finite element spaces, including to/from elements with moment functionals, is also supported in DOLFINx. This is useful in a range of applications, including for visualization. For example, $H(\mathrm{div})$- and $H(\mathrm{curl})$-conforming finite element spaces can be visualized exactly by interpolating into a sufficiently rich discontinuous Lagrange space prior to visualization.

8 ASSEMBLY

Assembling finite element forms into matrices, vectors, and scalars is central to any finite element library. This involves iterating over mesh entities, computing the local element tensor using a finite element kernel, and adding the contributions into the correct locations.

Assembly performance depends heavily on the performance of the element kernel. Since the PDE and element type determine the kernel, users often need to write problem-specific kernels in traditional finite element libraries.

DOLFINx eases the burden of kernel creation by using a domain-specific language (UFL) and a code generate FFCx to automatically generate optimised finite element kernels from a high-level, mathematical description. This approach can

```python
def sigma(v):
    """An expression for the stress given a displacement field v."""
    def eps(v): return sym(grad(v))
    return 2.0 * mu * eps(v) + lmbda * tr(eps(v)) * Identity(len(v))


# Define deviatoric and von Mises stress as UFL expressions
sigma_dev = sigma(uh) - (1 / 3) * tr(sigma(uh)) * Identity(len(uh))
sigma_vm = sqrt((3 / 2) * inner(sigma_dev, sigma_dev))

# Interpolate von Mises stress into a finite element space
W = fem.functionspace(msh, ("Discontinuous Lagrange", 0))
interpolation_points = W.element.interpolation_points
sigma_vm_expr = fem.Expression(sigma_vm, interpolation_points)
sigma_vm_h = fem.Function(W)
sigma_vm_h.interpolate(sigma_vm_expr)
```

expression.py

Fig. 11. Interpolating a UFL expression of von Mises stress of a continuous vector-valued finite element solution into a discontinuous scalar-valued finite element space. For brevity, we omit the solution of the linear elasticity problem.

be highly-effective and save time for a wide range of cases. However, we stress that DOLFINx also supports a traditional workflow where users provide their own element kernel instead of relying on generated code. This is an important departure from legacy DOLFIN, which also featured automatic kernel generation but could not easily support cases which fell outside of the abstractions of UFL; requiring code generation can be a significant burden when exploring novel numerical schemes. By contrast, the data- and functional-oriented design of DOLFINx retains the attractive features of DOLFIN and overcomes its limitations by supporting a range of kernel creation approaches: generated kernels (C++ or Python); JIT compiled Numba kernels (Python); and hand-coded, compiled kernels with a C interface (C++ or Python). Additionally, these approaches can be combined. Novel is the support for custom, performant and parallel assembly functions in Python using Numba.

8.1 Assembling generated kernels

Finite element forms expressed in UFL can be automatically complied by FFCx and assembled over mesh entities by DOLFINx into tensors. In C++, the generated code can be compiled into a program. In Python, the code is JIT compiled using CFFI [96] and passed to a DOLFINx assembly function. The snippet below demonstrates the assembly of a mass operator into a DOLFINx native distributed sparse matrix using single-precision complex floating point numbers.

```python
1 V = fem.functionspace(...)
2 u, v = ufl.TrialFunction(V), ufl.TestFunction(V)
3 a = ufl.inner(u, v) * ufl.dx
4 a = fem.form(a, dtype=np.complex64)  # Compile the bilinear form into a concrete instance
5 A = fem.assemble_matrix(a)
6 A.scatter_reverse()
```

In line 3, `a` is an abstract, symbolic representation of the form. It can be further manipulated using UFL functions. It can be compiled into a finite element kernel using the `form` function (line 4). This is made explicit in DOLFINx so that users can control when JIT compilation occurs to better manage resources. It also avoids opaque memory resource demands, avoids mutable states that make programs harder to reason with, and discourages user code that introduces repeat preprocessing of a UFL form, which can have a non-negligible cost. Users have complete control over compilation parameters; for example, the same abstract form `a` can be compiled with different scalar types:

```python
a0 = fem.form(a, dtype=np.float32)
a1 = fem.form(a, dtype=np.complex64)
```

We avoid hidden caching or dynamic attaching of data to objects, e.g. the sparse matrix is not attached to the form `a`, as this can introduce opaque, and sometimes unnecessary, increases in memory usage. Except in the very highest level interfaces, DOLFINx does not hide parallel communication steps. Attempting to hide communication steps in many cases introduces more communication operations than are required as the library cannot anticipate how a user will next use an object.

The C++ interface code for the same operation is very similar, as shown by the following snippet:

```cpp

  auto element = basix::create_element<float>(
      basix::element::family::P, basix::cell::type::triangle, 1,
      basix::element::lagrange_variant::unset,
      basix::element::dpc_variant::unset, false);

  auto V = std::make_shared<fem::FunctionSpace<float>>(
      fem::create_functionspace<float>(mesh, std::make_shared<fem::FiniteElement<float>>(element)));
  auto a = std::make_shared<fem::Form<float>>(
      fem::create_form<float>(*form_mass_a, {V, V}, {}, {}, {}, {}));
```

mass_form.cpp

In this case, `form_mass_a` is a pointer to a form struct (generated by FFCx) which provides a kernel.

8.2 Local kernel interface

Local kernels that are executed by the DOLFINx assembly functions have the following C signature, as defined in Unified Form-assembly code for FEniCSx (UFCx) [16]:

```c
void kernel(T* restrict A,
            const T* restrict w,
            const T* restrict c,
            const T2* restrict coordinate_dofs,
            const int* restrict entity_local_index,
            const uint8_t* restrict quadrature_permutation,
            void* custom_data);
```

where `T` is the data type (e.g. `float`, `double`, `float`, `float _Complex` or `double _Complex`), `T2` is the geometry data type (e.g. `float` or `double`) and `A` is the local element tensor (as an in/out argument). The array `w` contains the coefficients attached to the form. This coefficient array is a list of finite element coefficient values for the given cell, ordered as $(c_0, \ldots, c_{N_c-1}, d_0, \cdots, d_{N_d-1}, \ldots)$, where c_i are the coefficients of a function c in a finite element space with N_c DOFs per cell, and d_i are from a space with N_d DOFs per cell. The array `c` contains constants (of any rank) attached to the form. The array `coordinate_dofs` holds the geometric degrees-of-freedom values for a cell. The integer `entity_local_index` is the local index (relative to the cell) of the sub-entity that the kernel is executed over, and `quadrature_permutation` points to an integer that encodes how the sub-entity should be permuted to ensure that the orientation of the sub-entity is agreed upon when viewed from two neighboring cells. The final `custom_data` argument can be used to pass any additional information into the

```python
@numba.cfunc(c_signature, nopython=True)
def tabulate_A(A_, w_, c_, coords_, entity_local_index,
               cell_orientation, custom_data):
    # Wrap pointers as a Numpy arrays
    A = numba.carray(A_, (dim, dim))
    coordinate_dofs = numba.carray(coords_, (3, 3))

    x0, y0 = coordinate_dofs[0, :2]
    x1, y1 = coordinate_dofs[1, :2]
    x2, y2 = coordinate_dofs[2, :2]

    # Compute Jacobian determinant and fill the output array with
    # precomputed mass matrix scaled by the Jacobian
    detJ = abs((x0 - x1) * (y2 - y1) - (y0 - y1) * (x2 - x1))
    A[:] = detJ * M_hat
```

numba_custom_kernel.py

Fig. 12. A Numba function that captures the already-computed cell mass matrix on the reference cell $\hat{R}$ as a compile-time constant `M_hat` to compute the cell mass matrix on a physical cell R.

kernel by casting to a void pointer. For example, this argument may be used if a user is writing updated kernels for cut finite element methods [25] where custom quadrature rules are needed for each cut cell. Generated or hand-coded kernels that conform to this interface can exploit the built-in DOLFINx assembly functions. The simplicity of the kernel interface eases possible integration into other finite element solvers, e.g. [80].

8.3 Custom kernels and assemblers

The data- and function-oriented design of DOLFINx allows users to implement custom operations by either (i) supplying a custom kernel to the built-in assembly functions, (ii) use their own assembly functions with C kernels generated from FFCx, or (iii) provide their own kernel and own assembler. These approaches can even be combined within a single application. In Python, both kernels and assemblers can be written using Numba [71], which JIT compiles a subset of Python and NumPy code to machine code using LLVM. Alternatively, CFFI can be used to compile C functions that can be used in Python and called from within the existing DOLFINx C++ assemblers. For all cases, the functionality is enabled by the data- and function-oriented design of DOLFINx. We demonstrate these features in this section, but for clarity the examples presented are deliberately simple and could also be implemented using code generation.

8.3.1 Custom kernels. We first consider creating an element matrix kernel using Numba. The high-level workflow is shown in figure 4. Consider assembling a mass matrix on affine triangles R using degree 1 Lagrange basis functions with local matrix entries $\mathbf{A} := (a_{ij}) \in \mathbb{R}^{3\times 3}$ given by

$$a_{ij} = \int_R \varphi_i \cdot \varphi_j \,\mathrm{d}x = |\det J| \int_{\hat{R}} \hat{\varphi}_i \cdot \hat{\varphi}_j \,\mathrm{d}\hat{x} = |\det J|\, \hat{m}_{ij}. \tag{4}$$

The matrix $\hat{\mathbf{M}} := (\hat{m}_{ij}) \in \mathbb{R}^{3\times 3}$ is a mass matrix on the reference triangle $\hat{R}$.

The Numba code for this kernel is shown in figure 12. The `c_signature` in the decorator has type `numba.types` and defines a C interface for the function, which conforms to the interface defined in section 8.2. FFCx contains helper functions to construct `c_signature`.

```python
@numba.cfunc(c_signature, nopython=True)
def tabulate_A_wrapped(A_, w_, c_, coords_, entity_local_index,
                       cell_orientation=ffi.NULL, custom_data=None):
    A = numba.carray(A_, (dim, dim))

    # Allocate new Numpy array where temporary tabulation is stored
    M = np.zeros_like(A)

    w = numba.carray(w_, (dim, ))
    c = numba.carray(c_, (1, ))

    # Call the compiled kernel (from_buffer is required to extract the
    # underlying data pointer)
    ufcx_kernel(ffi.from_buffer(M), ffi.from_buffer(w),
                ffi.from_buffer(c), coords_, entity_local_index,
                cell_orientation, empty_void_pointer())

    # Row sum matrix M and store into diagonal of the output matrix A
    np.fill_diagonal(A, np.sum(M, axis=1))
```

numba_wrapped_kernel.py

Fig. 13. A Numba kernel that calls an FFCx generated kernel (`ufcx_kernel`) and applies row sum lumping.

A pointer to the Numba compiled function can then be passed to the `Form` initializer, creating a form object equipped with our custom kernel. Any required constants of coefficients can also be passed to the form initializer. Additional data can also be captured as a compile-time constant from outside of the kernel's scope.

The code generation capabilities of FFCx can be composed with lower-level Numba operations. A user can compile UFL forms with the FFCx JIT compiler, and then call that kernel from within a custom Numba kernel. This can be useful for implementing operations where one may wish to modify or view the output of a standard kernel, e.g. to apply static condensation, mass matrix lumping or to inspect the local element matrix. Figure 13 shows an example of calling a FFCx generated kernel from within a Numba kernel. The compiled kernel `ufcx_kernel` computes the cell mass matrix A. As an illustrative example, the cell mass matrix is then modified using the method of row-sum lumping. 5

8.3.2 Custom assemblers. The data-oriented approach followed by DOLFINx makes it possible to write complete and efficient assembly functions in other languages, including Python. The data underpinning key objects, including meshes and DOF maps, can be easily accessed as C data types and shared across language interfaces.

As an example, in figures 14 and 15 we show a Numba CUDA assembler for the action of the mass matrix a_{ij} on a vector f_j into a vector u_i

$$u_i = a_{ij} f_j, \tag{5}$$

with a_{ij} defined similarly to section 8.3.1, except that we allow for the Lagrange finite element basis functions to be of arbitrary polynomial order. This GPU-accelerated kernel could be used inside a Krylov solver to compute the operator action without explicitly forming the sparse matrix A. For further details we refer to the figure captions.

8.4 Degree-of-freedom permutations and transformations

For high-degree elements, neighboring cells must agree on the orientation of their shared sub-entities. For meshes of simplex (i.e. interval, triangle or tetrahedron) cells, this can be achieved by a suitable local ordering of the vertices of

```python
import numba
import numba.cuda as cuda
import numpy as np
from mpi4py import MPI

import basix
import dolfinx
import ufl
```

cuda_assembler.py

```python
dtype = np.float32

mesh = dolfinx.mesh.create_unit_cube(comm, 100, 100, 100,
                                     dolfinx.mesh.CellType.hexahedron, dtype=dtype)
```

cuda_assembler.py

```python
# Assemble the action u = Mf

@cuda.jit
def assemble_vector(u, f, dofmap, M_hat, detJ):
    """Kernel to assemble the mass action u = \\int f v dx.

    - One thread per degree-of-freedom vector
    - One cell per block

    Args:
        u: Vector to assemble action into.
        f: Vector to compute action on.
        dofmap: cell to degree of freedom map.
        M_hat: Mass matrix on the reference cell.
        detJ: Determinant of the Jacobian for each physical cell.
    """
    thread_id = cuda.threadIdx.x  # Local thread ID (max: 1024)
    block_id = cuda.blockIdx.x  # Block ID (max: 2147483647)

    # f, restricted to element, shared across block (gather)
    _f = cuda.shared.array(shape=(num_dofs_per_cell,), dtype=dtype)
    dof = dofmap[block_id, thread_id]
    _f[thread_id] = f[dof]
    cuda.syncthreads()

    # Each thread computes the dot product of row i of M with _f
    val = 0.0
    for i in range(num_dofs_per_cell):
        val += M_hat[thread_id, i] * _f[i]

    val *= detJ[block_id]

    # scatter val into global u
    cuda.atomic.add(u, dof, val)
```

cuda_assembler.py

Fig. 14. Part 1: A custom assembler written using Numba and CUDA. A kernel to compute the action of the mass matrix on a vector $\boldsymbol{u} = A\boldsymbol{f}$. This kernel is executed in figure 15. For brevity we do not show the computation of $|\det J|$ via another CUDA kernel.

```python
e = basix.ufl.element("Lagrange", cell_type, degree=5, dtype=dtype)
V = dolfinx.fem.functionspace(mesh, e)

# Function to compute action on
f = dolfinx.fem.Function(V, dtype=dtype)
f.interpolate(lambda x: x[1])

# Create quadrature rule of sufficient order
pts, wts = basix.make_quadrature(cell_type, e.degree * 2 + 1)

# Compute mass matrix on reference cell
phi = e.tabulate(0, pts)[0, :, :]
M_hat = np.asarray(np.einsum("qi,q,qj->ij", phi, wts, phi), dtype=dtype)

u = dolfinx.fem.Function(V, dtype=dtype)

# Move required data to the device for assembling action
dofmap_d = cuda.to_device(V.dofmap.list)
u_d = cuda.to_device(u.x.array)
f_d = cuda.to_device(f.x.array)
M_hat_d = cuda.to_device(M_hat)

# Get the number of blocks and threads per block
num_dofs_per_cell = e.dim
num_blocks = num_cells
threads_per_block = num_dofs_per_cell

# Invoke kernel with the specified grid configuration and arguments
assemble_vector[num_blocks, threads_per_block](u_d, f_d, dofmap_d, M_hat_d, detJ_d)
cuda.synchronize()
```

cuda_assembler.py

Fig. 15. Part 2: A custom assembler written using Numba and CUDA. The element mass matrix is computed, the necessary data is transferred to the CUDA device, and the mass matrix action kernel defined in figure 14 is executed. For full discussion of CUDA execution semantics (grid, block, thread) see e.g. [5].

each cell. An ordering approach can also be used for quadrilateral and hexahedral cells, but the ordering operation is not local to each cell and is not possible to consistently order all hexahedral cell meshes [1, 59].

To achieve consistent sub-entity orientations in DOLFINx on arbitrary meshes, we use a method of DOF permutations and transformations [105]. DOF permutations and transformations determine how the local basis functions and DOFs should be adjusted to account for differences in the orientations of cell sub-entities on the reference cell and the physical mesh. DOLFINx orients entities as described in section 6.3. DOF permutations and transformations are applied whenever the reference orientations derived from the local indices do not match the physical orientations derived from the global indices. This method allow us to use arbitrary degree finite elements on meshes of any cell type.

Consider a Lagrange element. Orientation differences for edges and faces can be accounted for by permuting the DOF numbering. For example, if the direction of an edge on the reference cell disagrees with the direction of an edge that it corresponds to in the physical mesh, the difference may be resolved by reversing the order of the global DOF numbers assigned to each local DOF on that edge. Now consider a more general case, for example N1 elements [86] of degree greater than one on a tetrahedron, for which the definition of the DOFs on each face includes dot products with respect to tangent vectors on the face. The face tangent vectors on adjacent cells must be consistently aligned. In this case, a DOF permutation is not adequate and a more general transformation is required [105].

Function name	Operation
`FiniteElement::T_apply`	$\mathbf{TA}$
`FiniteElement::Tt_apply`	$\mathbf{T}^{\mathsf{T}}\mathbf{A}$
`FiniteElement::Tinv_apply`	$\mathbf{T}^{-1}\mathbf{A}$
`FiniteElement::Tt_inv_apply`	$\mathbf{T}^{-\mathsf{T}}\mathbf{A}$
`FiniteElement::T_apply_right`	$\mathbf{AT}$
`FiniteElement::Tt_apply_right`	$\mathbf{AT}^{\mathsf{T}}$
`FiniteElement::Tinv_apply_right`	$\mathbf{AT}^{-1}$
`FiniteElement::Tt_inv_apply_right`	$\mathbf{AT}^{-\mathsf{T}}$

Fig. 16. Summary of Basix DOF transformation functions that apply a transformation operator in-place to $\mathbf{A}$.

8.4.1 Transformation operators. The value of a scalar finite element function f_h at some point within a cell can computed as

$$f_h = \boldsymbol{c}^{\mathsf{T}}\boldsymbol{\phi} = \tilde{\boldsymbol{c}}^{\mathsf{T}}\tilde{\boldsymbol{\phi}}, \tag{6}$$

where $\boldsymbol{c}$ is a vector of DOFs (restricted to the cell), $\boldsymbol{\phi}$ holds the basis functions, all relative to the physical cell ordering, and $\tilde{\boldsymbol{c}}$ and $\tilde{\boldsymbol{\phi}}$ are the equivalents following the reference cell ordering. We encode a transformation in a matrix $\mathbf{T}$ such that

$$\boldsymbol{\phi} = \mathbf{T}\tilde{\boldsymbol{\phi}}.$$

Inserting this into (6) leads to $\boldsymbol{c}^{\mathsf{T}}\boldsymbol{\phi} = \boldsymbol{c}^{\mathsf{T}}\mathbf{T}\tilde{\boldsymbol{\phi}} = (\mathbf{T}^{\mathsf{T}}\boldsymbol{c})^{\mathsf{T}}\tilde{\boldsymbol{\phi}} = \tilde{\boldsymbol{c}}^{\mathsf{T}}\tilde{\boldsymbol{\phi}}$, therefore vectors of DOFs transform according to

$$\tilde{\boldsymbol{c}} = \mathbf{T}^{\mathsf{T}}\boldsymbol{c}.$$

While $\mathbf{T}$ is often orthogonal, this not the case for all elements (see [105, section 4.1.4]). It follows straightforwardly that for an element matrix $\tilde{\mathbf{A}} \in \mathbb{C}^{m\times n}$ following the reference cell ordering that the matrix for the physical cell ordering is $\mathbf{A} = \mathbf{T}_1\tilde{\mathbf{A}}\mathbf{T}_2^{\mathsf{T}}$, where $\mathbf{T}_1$ and $\mathbf{T}_2$ are the transformations associated with the test and trial function elements, respectively. The algorithm used to compute $\mathbf{T}$ from the orientation of mesh entities and the definition of the element degrees-of-freedom is presented in [106].

8.4.2 Transformation and permutation functions. Basix provides functions to apply the transformations from the preceding section to data in-place. If the transformation is a permutation only, the Basix function `FiniteElement::permute` applies the $\mathbf{T}$ operation to a vector, and `FiniteElement::permute_inv` applies the inverse transformation ($\mathbf{T}^{-T}$, since all permutation operators are orthogonal). In practice, the permute functions are not called during assembly over cells, but when constructing a DOF map. For elements that require a more general transformation, transformations must be applied to cell-wise data, e.g. to restricted DOF arrays, local right-hand side vectors and element matrices. The Basix-provided functions for applying the transformation $\mathbf{T}$ in-place are listed in figure 16.

The Basix permute and transform functions take as arguments the data to permute/transform and a 32-bit unsigned integer that encodes the orientation of each cell sub-entity relative to the reference cell. Internally, Basix computes (small) permutation or (small) transformation matrix operators for each cell sub-entity; one for each edge to apply the effect of reversing an edge direction, and two for each face for applying the effect of rotations and reflections. The transformations are then applied in-place (see [106]).

For custom elements implemented using Basix, as presented in section 5.2, Basix determines the required transformations automatically.

8.4.3 Transformations for interpolation. When interpolating a function $f \in V$ into a finite element space V_h, we get from Basix the points on the reference cell where f needs to be evaluated for interpolation, push these points forward to the physical cells and then evaluate f. The values of f at evaluation points are then pulled back to the reference cell, and we apply an element interpolation matrix to get the local coefficients $\tilde{\boldsymbol{c}}$ on V_h for each cell. Since $\boldsymbol{c} = \mathbf{T}^{-\top}\tilde{\boldsymbol{c}}$, we apply the function `FiniteElement::Tt_inv_apply` to compute $\boldsymbol{c}$. The same approach can be used when interpolating between different finite spaces.

8.4.4 Transformations for custom kernels and assemblers in Python. For Numba-based custom kernels or assemblers that require degree-of-freedom transformations beyond simple permutations, and therefore must be applied during assembly, the submodule `basix.numba_helpers` provides efficient implementations of `T_apply` and `Tt_apply_right`.

9 LINEAR ALGEBRA

There exists a rich range of linear algebra libraries providing data structures and solvers, e.g. PETSc, Trilinos [112] and Eigen [49]. Our experience is that third-party linear algebra libraries are best supported *not* through extensive wrappers, but non-intrusively and allowing users direct access to the complete interface of the 3rd-party library. The approach we advocate lends itself to sustainability, maintainability and extensibility, with users able to introduce new linear algebra backends and use all features of the backend without modification of the DOLFINx library. It also avoids hidden inconsistencies, where different libraries perform nominally similar operations differently.

We will show examples of how the functional and data-oriented interfaces supports such a non-intrusive design.

9.1 Vectors/arrays

DOLFINx provides a distributed vector (array) class (`Vector`) that builds on the `IndexMap` and `Scatterer` functionality described in section 4. The class is templated over the scalar type and a container type (typically a `std::vector`); templating over the container type allows control of where the vector data is placed in memory, e.g. on a host or on a GPU device. Vectors are an example where performance dictates the storage layout, with the linear memory model (contiguous) the only reasonable choice. Therefore, we do not encapsulate the storage. To interface with linear algebra libraries, the DOLFINx `Vector` data can be wrapped by other libraries, e.g. by PETSc (C++ and Python) or NumPy (Python).

DOLFINx assemblers for vectors assemble into memory wrapped by a `std::span`, which allows direct and no-copy assembly in to any data structure that conforms to the requirements of `std::span`; examples include DOLFINx `Vector`s, NumPy arrays, plain C-style arrays and many other array-like data structures.

9.2 Matrices and solvers

DOLFINx also provides a distributed CSR matrix class (`MatrixCSR`), which is templated over the scalar type and the internal storage container types (like for `Vector`, this allows placement in host or device memory). The underlying CSR data arrays can be accessed, allowing memory to be shared with other libraries, e.g. to create SciPy sparse matrices that share data.

In general, sparse matrix data structures are considerably more complex than vectors, with a wide range of storage formats, interfaces and implementations. Two main requirements are (i) initializing a sparse matrix and (ii) inserting into a sparse matrix. For formats that require *a priori* knowledge of the sparsity pattern, given a bilinear form `a`, the code below illustrates how a matrix sparsity pattern can be constructed.

```
    auto V = std::make_shared<fem::FunctionSpace<float>>(
        fem::create_functionspace<float>(mesh, std::make_shared<fem::FiniteElement<float>>(element)));
    auto a = std::make_shared<fem::Form<float>>(
        fem::create_form<float>(*form_mass_a, {V, V}, {}, {}, {}, {}));

    la::SparsityPattern pattern = fem::create_sparsity_pattern(*a);
    pattern.finalize();
    auto [nonzeros, row_offset] = pattern.graph();
```

sparsity_pattern.cpp

The last line above returns the adjacency list data for non-zero entries; `nonzeros` holds column indices for non-zero columns and `row_offset` points to the start of each row in `nonzeros`. The `SparsityPattern` holds index map for the rows and columns that describe the parallel layout and row/column index block sizes, which can also be accessed. By exposing the sparsity as an adjacency list, a user can use the data to initialize a (distributed) sparse matrix. A native DOLFINx sparse matrix can be constructed directly from a sparsity pattern, and for convenience native DOLFINx and PETSc matrix factory functions are also provided:

```
    la::MatrixCSR<float> A(pattern);
    Mat B = la::petsc::create_matrix(comm, pattern);
```

sparsity_pattern.cpp

Insertion into a sparse matrix is typically via a library function call. A conventional approach to supporting different matrix backends is to wrap a linear algebra object with a native library class, possibly derived from a base class, to avoid exposing the specific linear algebra backend directly across a library. This common pattern was followed by DOLFIN. It can require a substantial amount of additional code and considerable wrapping of functionality of the third-party library (which is inevitably never comprehensive), and the use of C++ classes can make working across languages more difficult. In DOLFINx, different linear algebra libraries are supported by the functional design of DOLFINx assemblers, with matrix assemblers accepting a function that inserts local contributions into a (sparse) matrix. DOLFINx matrix assembly functions require an 'insertion' function with the following signature to be passed as an argument:

```
std::function<int(std::span<const std::int32_t>, std::span<const std::int32_t>, std::span<const T>)>
```

where the first two arguments are the row and column indices of the matrix to be added, respectively, and the last argument holds the values to be inserted with `T` being a scalar type. Anonymous (lambda) functions make the creation of insertion functions that require captured data straightforward. For example, to assemble into a PETSc matrix, we can define:

```
    auto V = std::make_shared<fem::FunctionSpace<PetscScalar>>(
        fem::create_functionspace<PetscScalar>(mesh, std::make_shared<fem::FiniteElement<PetscScalar
   >>(element)));
    auto a = std::make_shared<fem::Form<PetscScalar>>(
        fem::create_form<PetscScalar>(*form_laplace_a, {V, V}, {}, {}, {}, {}));

    la::SparsityPattern pattern = fem::create_sparsity_pattern(*a);
    pattern.finalize();
    Mat A = la::petsc::create_matrix(comm, pattern);
```

```python
V = functionspace(msh, ("Lagrange", 2, (msh.geometry.dim,)))
Q = functionspace(msh, ("Lagrange", 1))
W = MixedFunctionSpace(V, Q)

(u, p) = TrialFunctions(W)
(v, q) = TestFunctions(W)
a = inner(grad(u), grad(v)) * dx + inner(p, div(v)) * dx + inner(div(u), q) * dx
a_blocked = form(extract_blocks(a))

A = assemble_matrix(a_blocked, kind="mpi")
A.assemble()
```

blocked_stokes.py

Fig. 17. Snippet showing assembly of mixed discretisation of Stokes operator into a PETSc blocked sparse matrix.

```cpp

  auto add_vals = [A](std::span<const std::int32_t> rows,
                      std::span<const std::int32_t> cols,
                      std::span<const PetscScalar> vals)
  {
    PetscErrorCode ierr
        = MatSetValuesLocal(A, rows.size(), rows.data(), cols.size(),
                            cols.data(), vals.data(), ADD_VALUES);
    return ierr;
  };

  // Assemble bilinear form 'a' into matrix 'A'
  fem::assemble_matrix(add_vals, *a, {});
```

assemble_mat_petsc.cpp

In the lambda function syntax, `[A]` copies `A` (a pointer in this case) from the outer scope into the scope of the lambda function. The DOLFINx assembly function calls the `add_vals` function to perform insertion. If, for example, the PETSc matrix required a different integer pointer type, the lambda function can capture a memory buffer for copying integer arrays prior to calling the PETSc function. DOLFINx uses this approach for assembly into its native sparse matrix format, and for convenience provides insertion functions for PETSc matrices, including blocked and nested formats appropriate for efficiently storing matrices associated with mixed finite element methods or multi-physics problems. A simple example for the mixed discretisation of the Stokes problem is given in figure 17.

Assembly into other library formats, e.g. Trilinos or Eigen, is straightforward using the above pattern and in all cases the core DOLFINx assembly code is unaware of the matrix library interface/implementation.

Once vectors and matrices have been assembled, handles to the underlying liner algebra objects and be accessed and the full native library interfaces used to solve linear systems. For example, from Python the full petsc4py [12] or SciPy interface can be used.

10 CONCLUSIONS

We have presented an overview of the design principles and implementation of the DOLFINx finite element library. DOLFINx inherits the strengths of the earlier FEniCS library DOLFIN in providing a high level of mathematical abstraction, exploiting code generation techniques for finite element kernels and providing C++ and Python interfaces, with the Python interface supported by JIT compilation. DOLFINx overcomes the shortcomings and criticisms of the DOLFIN approach by following new design principles, leading to much greater extensibility, improved performance and substantially improved cross-language support. Noteworthy is that DOLFINx is a very compact library, despite the wide range of functionality that it supports, with the C++ component having fewer than 30 000 lines of code.

11 SUPPLEMENTARY MATERIALS

The source code for the snippets in this paper (MIT license), the source code to the FEniCSx components and a Docker image containing an environment to run the snippets are available at [15]. The snippets in this paper are compatible with DOLFINx 0.10.0, Basix 0.10.0, FFCx 0.10.1 and UFL 2025.2.0.

ACKNOWLEDGMENTS

We would like to thank Matt Moelter for his feedback on a pre-print of this manuscript. We thank Drew Parsons and Min Ragan–Kelley for their dedicated work maintaining the Debian and Anaconda packages of the FEniCSx components, respectively. We gratefully acknowledge the contributions of many people to the codebase of DOLFINx and the other components of FEniCSx, including (but not limited to) Francesco Ballarin, Michele Castriotta, Massimiliano Leoni and Sarah Roggendorf. The FEniCS Project is a fiscally sponsored project of NumFOCUS. MH and IB have received funding from the Google Summer of Code Program via the NumFOCUS umbrella organization.

IB and JSD acknowledge the support of EPSRC under grants EP/L015943/1 and EP/W026260/1. JPD acknowledges the support of EPSRC under grants EP/L015943/1 and EP/W026635/1. MH acknowledges the support of the Luxembourg National Research Fund under grant COAT/17205623 and the Luxembourg Ministry of Economy under the grant FEDER 2018-04-024. MWS acknowledges the support from EPSRC under grants EP/S005072/1 and EP/W007460/1. NS acknowledges the support from the NSF-EAR under grant 2021027 and the Carnegie Institution for Science with a President's Fellowship. MER acknowledges support and funding from the Research Council of Norway via FRIPRO grant agreement #324239 (EMIx) and the U.S.–Norway Fulbright Foundation for Educational Exchange. CNR and GNW acknowledge the support of EPSRC under grants EP/P013309/1, EP/V001396/1, EP/V001345/1, EP/S005072/1, EP/W00755X/1 and EP/W026635/1.

GLOSSARY

API application programming interface. 3, 9, 11

BDM Brezzi–Douglas–Marini. 11, 20

CFFI C Foreign Function Interface. 9, 22, 24
CR Crouzeix–Raviart. 11

DOF degree of freedom. 8, 11, 12, 14, 15, 19, 20, 23, 25, 27, 28
DOLFIN Dynamic Object-oriented Library for FINite element computation. 2, 6, 8, 9, 22, 31, 32
DPC discontinuous polynomial cubical. 11

FFC FEniCS Form Compiler. 2
FFCx FEniCSx Form Compiler. 3, 4, 5, 6, 11, 21, 22, 23, 24, 25

GL Gauss–Legendre. 11
GLL Gauss–Lobatto–Legendre. 11

HHJ Hellan–Herrmann–Johnson. 11

I/O input/output. 3, 4

JIT just-in-time. 3, 4, 7, 9, 22, 24, 25, 32

MPI Message Passing Interface. 9

N1 Nédélec first kind. 11, 20, 21, 27
N2 Nédélec second kind. 11, 20
NumPy . 9

PETSc Portable, Extensible Toolkit for Scientific Computation. 3, 30, 31

RT Raviart–Thomas. 11, 20

TNT tiniest tensor. 12, 13

UFCx Unified Form-assembly code for FEniCSx. 23
UFL Unified Form Language. 2, 3, 4, 5, 6, 11, 12, 13, 18, 20, 21, 22, 25

REFERENCES

[1] Rainer Agelek, Michael Anderson, Wolfgang Bangerth, and William L. Barth. 2017. On orienting edges of unstructured two- and three-dimensional meshes. *ACM Trans. Math. Software* 44, 1, Article 5 (2017), 22 pages. https://doi.org/10.1145/3061708

[2] Martin Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E. Rognes, and Garth N. Wells. 2015. The FEniCS project version 1.5. *Archive of Numerical Software* 3, 100 (2015), 9–23. https://doi.org/10.11588/ans.2015.100.20553

[3] Martin S. Alnæs, Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. 2014. Unified Form Language: A Domain-specific Language for Weak Formulations of Partial Differential Equations. *ACM Trans. Math. Softw.* 40, 2 (March 2014), 9:1–9:37. https://doi.org/10.1145/2566630

[4] Robert Anderson, Julian Andrej, Andrew Barker, Jamie Bramwell, Jean-Sylvain Camier, Jakub Cerveny, Veselin Dobrev, Yohann Dudouit, Aaron Fisher, Tzanio Kolev, et al. 2021. MFEM: A modular finite element methods library. *Computers & Mathematics with Applications* 81 (2021), 42–74. https://doi.org/10.1016/j.camwa.2020.06.009

[5] Richard Ansorge. 2022. *Programming in Parallel with CUDA: A Practical Guide.* Cambridge University Press, Cambridge. https://doi.org/10.1017/9781108855273

[6] J. H. Argyris, I. Fried, and D. W. Scharpf. 1968. The TUBA Family of Plate Elements for the Matrix Displacement Method. *The Aeronautical Journal* 72, 692 (1968), 701–709. https://doi.org/10.1017/S000192400008489X

[7] Daniel Arndt, Wolfgang Bangerth, Denis Davydov, Timo Heister, Luca Heltai, Martin Kronbichler, Matthias Maier, Jean-Paul Pelteret, Bruno Turcksin, and David Wells. 2021. The deal.II finite element library: Design, features, and insights. *Computers & Mathematics with Applications* 81 (2021), 407–422. https://doi.org/10.1016/j.camwa.2020.02.022

[8] Douglas N. Arnold and Gerard Awanou. 2011. The serendipity family of finite elements. *Foundations of Computational Mathematics* 11, 3 (2011), 337–344. https://doi.org/10.1007/s10208-011-9087-3

[9] Douglas N. Arnold and Anders Logg. 2014. Periodic table of the finite elements. *SIAM News* 47 (2014).

[10] Douglas N. Arnold and Shawn W. Walker. 2020. The Hellan–Herrmann–Johnson method with curved elements. *SIAM Journal on Numberical Analysis* 58, 5 (2020), 2829–2855. https://doi.org/10.1137/19M1288723

[11] C. Bahriawati and Carsten Carstensen. 2005. Three Matlab Implementations of the Lowest-order Raviart–Thomas Mfem with a Posteriori Error Control. *Computational Methods in Applied Mathematics* 5, 4 (2005), 333–361. https://doi.org/10.2478/cmam-2005-0016

[12] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil M. Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, Jacob Faibussowitsch, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. 2023. PETSc Web page. https://petsc.org/

[13] Francesco Ballarin, Alberto Sartori, and Gianluigi Rozza. 2015. RBniCS: Reduced Order modelling in FEniCS. Poster presented at MoRePaS 2015. https://doi.org/10.14293/P2199-8442.1.SOP-MATH.PUQ0WD.v1

[14] Wolfgang Bangerth, Ralf Hartmann, and Guido Kanschat. 2007. deal.II—a general-purpose object-oriented finite element library. *ACM Trans. Math. Software* 33, 4 (2007), 24–es. https://doi.org/10.1145/1268776.1268779

[15] Igor Baratta, Joseph Dean, Jørgen Schartum Dokken, Michal Habera, Jack S. Hale, Chris Richardson, Marie E. Rognes, Matthew W. Scroggs, Nathan Sime, and Garth N. Wells. 2025. *Supplementary Material. DOLFINx: The next generation FEniCS problem solving environment.* https://doi.org/10.5281/zenodo.13963645

[16] Igor Baratta, Jørgen Dokken, Michal Habera, Jack S. Hale, Chris Richardson, Matthew Scroggs, Garth Wells, et al. 2023. UFCx: FEniCSx Unified Form-assembly Code. GitHub. https://github.com/FEniCS/ffcx/blob/main/ffcx/codegeneration/ufcx.h [Online; accessed 25-September-2023].

[17] Klaus-Jürgen Bathe. 1986. Finite elements in CAD and ADINA. *Nuclear Engineering and Design* 98, 1 (1986), 57–67. https://doi.org/10.1016/0029-5493(86)90120-2

[18] Michel Bercovier and Olivier Pironneau. 1979. Error estimates for finite element method solution of the Stokes problem in the primitive variables. *Numer. Math.* 33 (1979), 211–224. https://doi.org/10.1007/BF01399555

[19] Guy E. Blelloch. 2010. Functional parallel algorithms. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming (ICFP '10)*. Association for Computing Machinery, New York, NY, USA, 247. https://doi.org/10.1145/1863543.1863579

[20] M. G. Blyth and C. Pozrikidis. 2006. A Lobatto interpolation grid over the triangle. *IMA Journal of Applied Mathematics* 71, 1 (2006), 153–169. https://doi.org/10.1093/imamat/hxh077

[21] Wietse M. Boon, Martin Hornkjøl, Miroslav Kuchta, Kent-Andre Mardal, and Ricardo Ruiz-Baier. 2021. Parameter-robust methods for the Biot–Stokes interfacial coupling without Lagrange multipliers. (2021). arXiv:2111.05653 arXiv 2111.05653.

[22] Susanne Brenner and Ridgway Scott. 2008. *The Mathematical Theory of Finite Element Methods* (3 ed.). Springer-Verlag, New York. https://doi.org/10.1007/978-0-387-75934-0

[23] Franco Brezzi, Jim Douglas, and L. Donatella Marini. 1985. Two families of mixed finite elements for second order elliptic problems. *Numer. Math.* 47 (1985), 217–235. https://doi.org/10.1007/BF01389710

[24] Are Magnus Bruaset and Hans Petter Langtangen. 1997. A comprehensive set of tools for solving partial differential equations: Diffpack. In *Numerical Methods and Software Tools in Industrial Mathematics*. Springer, 61–90. https://doi.org/10.1007/978-1-4612-1984-2_4

[25] Erik Burman, Susanne Claus, Peter Hansbo, Mats G. Larson, and André Massing. 2015. CutFEM: Discretizing geometry and partial differential equations. *Internat. J. Numer. Methods Engrg.* 104, 7 (2015), 472–501. https://doi.org/10.1002/nme.4823

[26] C. Chevalier and F. Pellegrini. 2008. PT-Scotch: A tool for efficient parallel graph ordering. *Parallel Comput.* 34, 6 (2008), 318–331. https://doi.org/10.1016/j.parco.2007.12.001 Parallel Matrix Algorithms and Applications.

[27] Snorre H. Christiansen. 2011. On the linearization of Regge calculus. *Numer. Math.* 119, 4 (2011), 613–640. https://doi.org/10.1007/s00211-011-0394-z

[28] Philippe G. Ciarlet. 1978. Chapter 2: Introduction to the Finite Element Method. In *The Finite Element Method for Elliptic Problems*, J. L. Lions, G. Papanicolaou, and R. T. Rockafellar (Eds.). Studies in Mathematics and Its Applications, Vol. 4. Elsevier, 36–109. https://doi.org/10.1016/S0168-2024(08)70181-4

[29] Ray W. Clough. 1990. Original formulation of the finite element method. *Finite elements in analysis and design* 7, 2 (1990), 89–101. https://doi.org/10.1016/0168-874X(90)90001-U

[30] Bernardo Cockburn and Weifeng Qiu. 2014. Commuting diagrams for the TNT elements on cubes. *Math. Comp.* 83 (2014), 603–633. https://doi.org/10.1090/S0025-5718-2013-02729-9

[31] C++ Standards Committee et al. 2020. ISO International Standard ISO/IEC 14882: 2020, Programming Language C++. https://www.iso.org/standard/79358.html [Online; accessed 20-October-2023].

[32] conda contributors. [n. d.]. *conda: A system-level, binary package and environment manager running on all major operating systems and platforms.* https://github.com/conda/conda

[33] Richard Courant. 1943. Variational methods for the solution of problems of equilibrium and vibrations. *Bull. Amer. Math. Soc.* 49, 1 (1943), 1–23. https://doi.org/10.1090/S0002-9904-1943-07818-4

[34] Matteo Croci and Giacomo Rosilho de Souza. 2022. Mixed-precision explicit stabilized Runge-Kutta methods for single- and multi-scale differential equations. *J. Comput. Phys.* 464 (2022), 111349. https://doi.org/10.1016/j.jcp.2022.111349

[35] Matteo Croci, Vegard Vinje, and Marie E. Rognes. 2019. Uncertainty quantification of parenchymal tracer distribution using random diffusion and convective velocity fields. *Fluids and Barriers of the CNS* 16, 1 (2019), 1–21. https://doi.org/10.1186/s12987-019-0152-7

[36] Michel Crouzeix and Pierre-Arnaud Raviart. 1973. Conforming and nonconforming finite element methods for solving the stationary Stokes equations. *Revue Française d'Automatique, Informatique et Recherche Opérationnelle* 3 (1973), 33–75. https://doi.org/10.1051/m2an/197307R300331

[37] Joseph Dean. 2023. *Mathematical and computational aspects of solving mixed-domain problems using the finite element method.* Ph. D. Dissertation. Apollo - University of Cambridge Repository. https://doi.org/10.17863/CAM.108292

[38] Joseph Dean, Brandon Keith, Socrates Petrides, and Tzanio Kolev. 2022. MFEM issue #2949: Suboptimal convergence rate of discrete curl on tetrahedral meshes. https://github.com/mfem/mfem/issues/2949

[39] Jørgen S. Dokken, Simon W. Funke, August Johansson, and Stephan Schmidt. 2019. Shape Optimization Using the Finite Element Method on Multiple Meshes with Nitsche Coupling. *SIAM Journal on Scientific Computing* 41, 3 (2019), A1923–A1948. https://doi.org/10.1137/18M1189208

[40] Jørgen S. Dokken, Sebastian K. Mitusch, and Simon W. Funke. 2020. Automatic shape derivatives for transient PDEs in FEniCS and Firedrake. (2020). arXiv:2001.10058 arXiv 2001.10058.

[41] Michel Duprez and Alexei Lozinski. 2020. ϕ-FEM: A Finite Element Method on Domains Defined by Level-Sets. *SIAM J. Numer. Anal.* (2020). https://doi.org/10.1137/19M1248947 Publisher: Society for Industrial and Applied Mathematics.

[42] Howard Elman, David Silvester, and Andy Wathen. 2014. *Finite Elements and Fast Iterative Solvers: with Applications in Incompressible Fluid Dynamics.* Oxford University Press. https://doi.org/10.1093/acprof:oso/9780199678792.001.0001

[43] Patrick E. Farrell, David Ham, Simon Funke, and Marie E. Rognes. 2013. Automated Derivation of the Adjoint of High-Level Transient Finite Element Programs. *SIAM Journal on Scientific Computing* 35, 4 (2013), C369–C393. https://doi.org/10.1137/120873558

[44] Todd Gamblin, Matthew LeGendre, Michael R. Collette, Gregory L. Lee, Adam Moody, Bronis R. de Supinski, and Scott Futral. 2015. The Spack Package Manager: Bringing Order to HPC Software Chaos *(Supercomputing 2015 (SC'15))*. Austin, Texas, US. https://tgamblin.github.io/pubs/spack-sc15.pdf

[45] Christophe Geuzaine. 2007. GetDP: a general finite-element solver for the de Rham complex. In *Proceedings in Applied Mathematics and Mechanics*, Vol. 7. Wiley Online Library, 1010603–1010604. Issue 1. https://doi.org/10.1002/pamm.200700750

[46] Christophe Geuzaine and Jean-François Remacle. 2009. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *Internat. J. Numer. Methods Engrg.* 79, 11 (2009), 1309–1331. https://doi.org/10.1002/nme.2579

[47] William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, Axel Huebl, Mark Kim, James Kress, Tahsin Kurc, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrouchov, Manish Parashar, Franz Poeschel, David Pugmire, Eric Suchyta, Keichi Takahashi, Nick Thompson, Seiji Tsutsumi, Lipeng Wan, Matthew Wolf, Kesheng Wu, and Scott Klasky. 2020. ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management. *SoftwareX* 12 (2020), 100561. https://doi.org/10.1016/j.softx.2020.100561

[48] Gerald L. Goudreau and J. O. Hallquist. 1982. Recent developments in large-scale finite element Lagrangian hydrocode technology. *Computer Methods in Applied Mechanics and Engineering* 33, 1-3 (1982), 725–757. https://doi.org/10.1016/0045-7825(82)90129-3

[49] Gaël Guennebaud, Benoît Jacob, et al. 2010. Eigen v3. http://eigen.tuxfamily.org [Online; accessed 23-November-2023].

[50] Jack S. Hale, Matteo Brunetti, Stéphane P. A. Bordas, and Corrado Maurini. 2018. Simple and extensible plate and shell finite element models through automatic code generation tools. *Computers & Structures* 209 (2018), 163–181. https://doi.org/10.1016/j.compstruc.2018.08.001

[51] Jack S. Hale, Lizao Li, Christopher N. Richardson, and Garth N. Wells. 2017. Containers for Portable, Productive, and Performant Scientific Computing. *Computing in Science & Engineering* 19, 6 (Nov. 2017), 40–50. https://doi.org/10.1109/MCSE.2017.2421459

[52] David A. Ham, Lawrence Mitchell, Alberto Paganini, and Florian Wechsung. 2019. Automated shape differentiation in the Unified Form Language. *Structural and Multidisciplinary Optimization* 60 (2019). Issue 5. https://doi.org/10.1007/s00158-019-02281-z

[53] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362. https://doi.org/10.1038/s41586-020-2649-2

[54] Paul Hauseux, Jack S. Hale, Stéphane Cotin, and Stéphane P. A. Bordas. 2018. Quantifying the uncertainty in a hyperelastic soft tissue model with stochastic parameters. *Applied Mathematical Modelling* 62 (2018), 86–102. https://doi.org/10.1016/j.apm.2018.04.021

[55] F. Hecht. 2012. New development in FreeFem++. *Journal of Numerical Mathematics* 20, 3-4 (2012), 251–265. https://doi.org/10.1515/jnum-2012-0013

[56] Jan S. Hesthaven and Tim Warburton. 2008. Beyond one dimension. In *Nodal Discontinuous Galerkin Methods: Algorithms, Analysis, and Applications.* Springer New York, New York, NY, Chapter 6, 169–241. https://doi.org/10.1007/978-0-387-72067-8_6

[57] H. D. Hibbitt. 1984. ABAQUS/EPGEN – A general purpose finite element code with emphasis on nonlinear applications. *Nuclear Engineering and Design* 77, 3 (1984), 271–297. https://doi.org/10.1016/0029-5493(84)90106-7

[58] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. 2010. Scalable Communication Protocols for Dynamic Sparse Data Exchange. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Bangalore, India) *(PPoPP '10)*. Association for Computing Machinery, New York, NY, USA, 159–168. https://doi.org/10.1145/1693453.1693476

[59] Miklós Homolya and David A. Ham. 2016. A parallel edge orientation algorithm for quadrilateral meshes. *SIAM Journal on Scientific Computing* 38, 5 (2016), S48–S61. https://doi.org/10.1137/15M1021325

[60] Q. Hong, J. Kraus, M. Kuchta, M. Lymbery, K. A. Mardal, and M. E. Rognes. 2022. Robust approximation of generalized Biot–Brinkman problems. *Journal of Scientific Computing* 93, 77 (2022), 77:1–77:28. https://doi.org/10.1007/s10915-022-02029-w

[61] A. Hrennikoff. 2021. Solution of Problems of Elasticity by the Framework Method. *Journal of Applied Mechanics* 8, 4 (2021), A169–A175. https://doi.org/10.1115/1.4009129

[62] Tobin Isaac. 2020. Recursive, Parameter-Free, Explicitly Defined Interpolation Nodes for Simplices. *SIAM Journal on Scientific Computing* 42, 6 (2020), A4046–A4062. https://doi.org/10.1137/20M1321802

[63] Wenzel Jakob. 2022. nanobind: tiny and efficient C++/Python bindings. https://github.com/wjakob/nanobind.

[64] George Karypis and Vipin Kumar. 1996. Parallel Multilevel K-Way Partitioning Scheme for Irregular Graphs. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing* (Pittsburgh, Pennsylvania, USA) *(Supercomputing '96)*. IEEE Computer Society, USA, 35–es. https://doi.org/10.1145/369028.369103

[65] George Karypis, Kirk Schloegel, and Vipin Kumar. 1997. *PARMETIS: Parallel Graph Partitioning and Sparse Matrix Ordering Library.* Report. http://conservancy.umn.edu/handle/11299/215345 Accepted: 2020-09-02T15:04:02Z.

[66] Robert C. Kirby. 2004. Algorithm 839: FIAT, a New Paradigm for Computing Finite Element Basis Functions. *ACM Trans. Math. Software* 30, 4 (2004), 502–516. https://doi.org/10.1145/1039813.1039820

[67] Robert C. Kirby, Anders Logg, Marie E. Rognes, and Andy R. Terrel. 2012. Common and unusual finite elements. In *Automated solution of differential equations by the finite element method*, Anders Logg, Kent-Andre Mardal, and Garth N. Wells (Eds.). Vol. 84. Springer, Berlin, Heidelberg, 95–119. https://doi.org/10.1007/978-3-642-23099-8_3

[68] Matthew G. Knepley and Dmitry A. Karpeev. 2009. Mesh Algorithms for PDE with Sieve I: Mesh Distribution. *Sci. Program.* 17, 3 (2009), 215–230. https://doi.org/10.1155/2009/948613

[69] Jože Korelc. 1997. Automatic generation of finite-element code by simultaneous optimization of expressions. *Theoretical Computer Science* 187, 1 (1997), 231–248. https://doi.org/10.1016/S0304-3975(97)00067-4

[70] Jože Korelc. 2002. Multi-language and Multi-environment Generation of Nonlinear Finite Element Codes. *Engineering with Computers* 18, 4 (2002), 312–327. https://doi.org/10.1007/s003660200028

[71] Siu Kwan Lam, Antoine Pitrou, and Stanley Seibert. 2015. Numba: A LLVM-Based Python JIT Compiler. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC* (Austin, Texas) *(LLVM '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 5 pages. https://doi.org/10.1145/2833157.2833162

[72] Hans Petter Langtangen. 1994. Diffpack: Software for partial differential equations. In *Proceedings of the Second Annual Object-Oriented Numerics Conference (OON-SKI'94)*, Vol. 94. Sunriver, OR, USA, 1–12.

[73] Wing Kam Liu, Shaofan Li, and Harold S. Park. 2022. Eighty years of the finite element method: Birth, evolution, and future. *Archives of Computational Methods in Engineering* 29, 6 (2022), 4431–4453. https://doi.org/10.1007/s11831-022-09740-9

[74] Anders Logg. 2009. Efficient representation of computational meshes. *International Journal of Computational Science and Engineering* 4, 4 (2009), 283–295. https://doi.org/10.1504/IJCSE.2009.029164

[75] Anders Logg and Garth N. Wells. 2010. DOLFIN: Automated Finite Element Computing. *ACM Transactions on Mathematical Softwareare* 37, 2 (2010), 20:1–20:28. https://doi.org/10.1145/1731022.1731030

[76] Anders Logg, Kristian B. Ølgaard, Marie E. Rognes, and Garth N. Wells. 2012. FFC: the FEniCS Form Compiler. In *Automated Solution of Differential Equations by the Finite Element Method*, A. Logg, K.-A. Mardal, and G. N. Wells (Eds.). Lecture Notes in Computational Science and Engineering, Vol. 84. Springer, Chapter 11, 227–238. https://doi.org/10.1007/978-3-642-23099-8_11

[77] Kevin R. Long. 2003. Sundance rapid prototyping tool for parallel PDE optimization. In *Large-scale pde-constrained optimization.* Springer, 331–341. https://doi.org/10.1007/978-3-642-55508-4_20

[78] R. H. MacNeal and C. W. McCormick. 1971. The NASTRAN computer program for structural analysis. *Computers & Structures* 1, 3 (1971), 389–412. https://doi.org/10.1016/0045-7949(71)90021-6

[79] Jakob M. Maljaars, Chris N. Richardson, and Nathan Sime. 2021. LEoPart: A particle library for FEniCS. *Computers & Mathematics with Applications* 81 (2021), 289–315. https://doi.org/10.1016/j.camwa.2020.04.023

[80] Arnaud Mazier, Sidaty El Hadramy, Jean-Nicolas Brunet, Jack S. Hale, Stéphane Cotin, and Stéphane P. A. Bordas. 2023. SOniCS: develop intuition on biomechanical systems through interactive error controlled simulations. *Engineering with Computers* (2023). https://doi.org/10.1007/s00366-023-01877-w

[81] Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2017. Parallel Graph Partitioning for Complex Networks. *IEEE Trans. Parallel Distrib. Syst.* 28, 9 (2017), 2625–2638. https://doi.org/10.1109/TPDS.2017.2671868

[82] Lawrence Mitchell, David Ham, Jack S. Hale, Robert C. Kirby, and Patrick E. Farrell. 2020. FIAT issue #40: Better nodes for RT/Nedelec. https://github.com/FEniCS/fiat/issues/40

[83] Sebastian K. Mitusch, Simon W. Funke, and Jørgen S. Dokken. 2019. dolfin-adjoint 2018.1: automated adjoints for FEniCS and Firedrake. *Journal of Open Source Software* 4, 38 (2019), 1292. https://doi.org/10.21105/joss.01292

[84] Sebastian K. Mitusch, Simon W. Funke, and Miroslav Kuchta. 2021. Hybrid FEM-NN models: Combining artificial neural networks with the finite element method. *J. Comput. Phys.* 446 (2021), 110651. https://doi.org/10.1016/j.jcp.2021.110651

[85] Mikael Mortensen and Kristian Valen-Sendstad. 2015. Oasis: A high-level/high-performance open source Navier–Stokes solver. *Computer Physics Communications* 188 (2015), 177–188. https://doi.org/10.1016/j.cpc.2014.10.026

[86] Jean-Claude Nédélec. 1980. Mixed finite elements in $\mathbb{R}^3$. *Numer. Math.* 35, 3 (1980), 315–341. https://doi.org/10.1007/BF01396415

[87] Jean-Claude Nédélec. 1986. A new family of mixed finite elements in $\mathbb{R}^3$. *Numer. Math.* 50, 1 (1986), 57–81. https://doi.org/10.1007/BF01389668

[88] Noemi Petra, Hongyu Zhu, Georg Stadler, Thomas J. R. Hughes, and Omar Ghattas. 2012. An inexact Gauss–Newton method for inversion of basal sliding and rheology parameters in a nonlinear Stokes ice sheet model. *Journal of Glaciology* 58, 211 (2012), 889–903. https://doi.org/10.3189/2012JoG11J182

[89] A. Plaza and G. F. Carey. 2000. Local refinement of simplicial grids based on the skeleton. *Applied Numerical Mathematics* 32, 2 (2000), 195–218. https://doi.org/10.1016/S0168-9274(99)00022-7

[90] Christophe Prud'Homme, Vincent Chabannes, Vincent Doyeux, Mourad Ismail, Abdoulaye Samake, and Gonçalo Pena. 2012. Feel++: A computational framework for galerkin methods and advanced numerical methods. In *ESAIM: Proceedings*, Vol. 38. EDP Sciences, 429–455. https://doi.org/10.1051/proc/201238024

[91] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. 2016. Firedrake: automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Softwareare* 43, 3 (2016), 407–422. https://doi.org/10.1145/2998441

[92] Pierre-Arnaud Raviart and Jean-Marie Thomas. 1977. A mixed finite element method for 2nd order elliptic problems. In *Mathematical aspects of finite element methods*, Ilio Galligani and Enrico Magenes (Eds.). Vol. 606. Springer, 292–315. https://doi.org/10.1007/BFb0064470

[93] Tullio Regge. 1961. General relativity without coordinates. *Il Nuovo Cimento* 19, 3 (1961), 558–571. https://doi.org/10.1007/BF02733251

[94] Chris N. Richardson, Igor A. Baratta, Joseph P. Dean, Adrian Jackson, and Garth N. Wells. 2025. An efficient multigrid solver for finite element methods on multi-GPU systems. *Procedia Computer Science* 267 (2025), 82–91. https://doi.org/10.1016/j.procs.2025.08.235

[95] Chris N. Richardson, Nathan Sime, and Garth N. Wells. 2019. Scalable computation of thermomechanical turbomachinery problems. *Finite Elements in Analysis and Design* 155 (2019), 32–42. https://doi.org/10.1016/j.finel.2018.11.002

[96] Armin Rigo, Maciej Fijałkowski, and Google Inc. 2021. CFFI: C foreign function interface for Python. https://cffi.readthedocs.io [Online; accessed 25-September-2023].

[97] Marie E Rognes and Anders Logg. 2013. Automated goal-oriented error control I: Stationary variational problems. *SIAM Journal on Scientific Computing* 35, 3 (2013), C173–C193. https://doi.org/10.1137/10081962X

[98] Peter Sanders and Christian Schulz. 2013. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Experimental Algorithms, 12th International Symposium, SEA 2013, Rome, Italy, June 5-7, 2013. Proceedings*, Vol. 7933. Springer, 164–175.

[99] Peter Sanders and Christian Schulz. 2013. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *Experimental Algorithms*, Vincenzo Bonifaci, Camil Demetrescu, and Alberto Marchetti-Spaccamela (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 164–175. https://doi.org/10.1007/978-3-642-38527-8_16

[100] Stephan Schmidt. 2018. Weak and Strong Form Shape Hessians and Their Automatic Generation. *SIAM Journal on Scientific Computing* 40, 2 (2018), C210–C233. https://doi.org/10.1137/16M1099972

[101] Joachim Schöberl. 2014. *C++ 11 implementation of finite elements in NGSolve*. Technical Report 30/2014. Vienna University of Technology. https://www.asc.tuwien.ac.at/~schoeberl/wiki/publications/ngs-cpp11.pdf

[102] Matthew W. Scroggs, Igor A. Baratta, Chris N. Richardson, and Garth N. Wells. 2022. Basix: a runtime finite element basis evaluation library. *Journal of Open Source Software* 7, 73 (2022), 3982. https://doi.org/10.21105/joss.03982

[103] Matthew W. Scroggs, Timo Betcke, Erik Burman, Wojciech Śmigaj, and Elwin van 't Wout. 2017. Software frameworks for integral equations in electromagnetic scattering based on Calderón identities. *Computers & Mathematics with Applications* 74, 11 (2017), 2897–2914. https://doi.org/10.1016/j.camwa.2017.07.049 arXiv:1703.10900

[104] Matthew W. Scroggs, Pablo D. Brubeck, Joseph P. Dean, Jørgen S. Dokken, India Marsden, Nuno Nobre, et al. 2025. DefElement: an encyclopedia of finite element definitions. https://defelement.org. [Online; accessed 25-August-2025].

[105] Matthew W. Scroggs, Jørgen S. Dokken, Chris N. Richardson, and Garth N. Wells. 2022. Construction of arbitrary order finite element degree-of-freedom maps on polygonal and polyhedral cell meshes. *ACM Trans. Math. Software* 48, 2 (2022), 18:1–18:23. https://doi.org/10.1145/3524456

[106] Matthew W. Scroggs and Garth N. Wells. 2025. Algorithm XXX: Computation of finite element degree-of-freedom transformation matrices. In preparation.

[107] Nathan Sime, Jakob M. Maljaars, Cian R. Wilson, and Peter E. van Keken. 2021. An Exactly Mass Conserving and Pointwise Divergence Free Velocity Method: Application to Compositional Buoyancy Driven Flow Problems in Geodynamics. *Geochemistry, Geophysics, Geosystems* 22, 4 (2021), e2020GC009349. https://doi.org/10.1029/2020GC009349

[108] J. A. Swanson. 1971. *ANSYS user's manual.* Vol. I.

[109] John A. Swanson. 1994. Keynote Paper: Current and future trends in finite element analysis. *International Journal of Computer Applications in Technology* 7, 3-6 (1994), 108–117. https://doi.org/10.1504/IJCAT.1994.062518

[110] J. A. Swanson and J. F. Patterson. 1971. Application of Finite Element Methods for the Analysis of Thermal Creep, Irradiation Induced Creep, and Swelling for LMFBR Design. In *Proceedings of The First International Conference on Structural Mechanics in Reactor Technology, Berlin, Germany.* IASMiRT, 293–310.

[111] The HDF Group. [n. d.]. *Hierarchical Data Format, version 5.* https://github.com/HDFGroup/hdf5

[112] The Trilinos Project Team. 2020. The Trilinos Project Website. https://trilinos.github.io [Online; accessed 23-November-2023].

[113] M .Jon Turner, Ray W. Clough, Harold C. Martin, and L. J. Topp. 1956. Stiffness and deflection analysis of complex structures. *Journal of the Aeronautical Sciences* 23, 9 (1956), 805–823. https://doi.org/10.2514/8.3664

[114] H. Juliette T. Unwin, Garth N. Wells, and Andrew W. Woods. 2016. CO_2 dissolution in a background hydrological flow. *Journal of Fluid Mechanics* 789 (2016), 768–784. https://doi.org/10.1017/jfm.2015.752

[115] Umberto Villa, Noemi Petra, and Omar Ghattas. 2021. HIPPYlib: An Extensible Software Framework for Large-Scale Inverse Problems Governed by PDEs: Part I: Deterministic Inversion and Linearized Bayesian Inference. *ACM Trans. Math. Software* 47, 2, Article 16 (2021), 34 pages. https://doi.org/10.1145/3428447

[116] Martin Řehoř and Jack S. Hale. 2023. FEniCSx Preconditioning Tools (FEniCSx-pctools). (2023). https://hdl.handle.net/10993/58088

Received